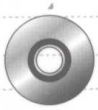
程序员书库

初学者的入门宝典，程序员的百科全书



DVD-ROM

12小时多媒体视频讲解

本书特色

- 对概念进行了精炼的总结，并结合插图进行说明
- 对文件的安全编程和网络编程等内容进行了深入剖析
- 对代码进行了详细注释，阅读起来很容易理解
- 全书提供了114个实例和1个综合案例，非常实用

Linux编程

从入门到精通

宫虎波　等编著



化学工业出版社

Linux
编程从入门到精通

本书主要内容

- shell编程方法
- C语言编程原理
- Gcc编译器
- make的使用
- 编译链接的原理及过程
- 异常处理及排错方法
- 自定义函数库的应用
- GUI界面程序设计——QT
- GUI界面程序设计——GTK+
- 文件属性编程
- 文件权限编程
- 标准输入/输出库编程
- 进程的控制
- 信号的处理
- 消息队列
- 共享内存
- 信号量
- 套接口网络编程
- 银行代理收费服务器案例

ISBN 978-7-122-05688-7
9 787122 056887 >
定价: 49.80元
含1DVD-ROM
销售分类建议: 计算机/程序设计

Linux编程

从入门到精通

程序员书库



12小时多媒体视频讲解

Linux编程从入门到精通

宫虎波　等编著

化学工业出版社
·北京·

随着 Linux 平台的广泛使用，与之相关专业的应用开发正变得越来越重要。本书由浅入深，系统地介绍了在Linux平台下使用C语言进行程序开发的技术，通过丰富的编程实例，使读者可以快速掌握Linux平台下应用C语言进行开发的技能。

本书共分为4篇21章，详细讲解了如何在Linux环境下进行C语言的开发。第一篇介绍了Linux的系统环境，主要包括Linux系统简介和shell环境等内容；第二篇介绍了C语言及编程环境，主要包括C语言简介，vi编辑器、gcc编译器、make的使用，程序调试和创建与使用库等内容；第三篇介绍了输入输出及进程管理，主要包括文件操作，标准输入输出库，界面程序设计——QT、GTK+、进程、信号等内容；第四篇介绍了进程间通信及网络编程，主要包括进程间通信——管道、消息队列、共享内存、信号量，Linux网络环境，基本套接口编程和综合实例——银行代理收费服务器等内容。

本书适合没有编程基础的C语言初学者作为入门教程，也可作为大、中专院校师生和培训班的教材。对于Linux下C语言开发的爱好者，本书也有较大的参考价值。

图书在版编目（CIP）数据

Linux编程从入门到精通 / 宫虎波等编著. —北京：化学工业出版社，2009.8
（程序员书库）
ISBN 978-7-122-05688-7
ISBN 978-7-89472-111-2（光盘）

Ⅰ. L… Ⅱ. 宫… Ⅲ. Linux操作系统-程序设计 Ⅳ. TP316.89

中国版本图书馆CIP数据核字（2009）第077915号

策划编辑：王思慧　陈　静　　装帧设计：蓝色印象
责任编辑：陈　静　　责任校对：蒋　宇

出版发行：化学工业出版社(北京市东城区青年湖南街13号 邮政编码100011)
印　　装：化学工业出版社印刷厂
787mm×1092mm　1/16　印张25¾　字数607千字　2009年8月北京第1版第1次印刷

购书咨询：010-64518888(传真：010-64519686)　售后服务：010-64518899
网　　址：http://www.cip.com.cn
凡购买本书，如有缺损质量问题，本社销售中心负责调换。

定　　价：49.80元(含1DVD—ROM)

版权所有　违者必究

出版者的话

近年来我国 IT 产业发展迅猛，IT 从业人数也不断增多，越来越多的人开始学习计算机技术。而作为计算机核心技术之一的程序设计更是受到了很多人的青睐，尤其是年轻人。如今很多年轻人开始学习编程，以使得自己在未来的职业生涯中更有竞争力。

但很多人刚开始学习时总觉得不得要领，感觉有一堵无形的墙竖在面前而无法逾越。究其原因是学习方法或者思路有问题所致。作为自学人员，学编程首先需要一本好书，适合自己阅读的书，这样才不至于走弯路，学习效果也好。然而图书市场上的图书虽然很多，但要真想找本很好的入门编程书却并不容易，很多书都是写给专家看的，把真正需要学习的人都给挡在了门外。基于这种原因，我们策划了这套《程序员书库》丛书，希望能降低读者学习编程的门槛，带领读者快速跨入编程的大门。

丛书包含书目

《Java 从入门到精通》
《C++从入门到精通》
《PHP 从入门到精通》
《Visual C++从入门到精通》
《Visual Basic 从入门到精通》
《ASP.NET 从入门到精通》
《Linux 编程从入门到精通》
《JavaScript 从入门到精通》
《CSS 从入门到精通》
《ActionScript 3.0 从入门到精通》
《Excel VBA 从入门到精通》
《SQL Server 2008 从入门到精通》

丛书主要特色

1. 由浅入深讲解到位，轻松入门

本丛书对内容的讲解都是从最基本的配置和概念讲起，然后层层深入，最后还安排了综合案例，很适合读者学习，可以达到轻松入门、快速提高的效果。

2. 程序代码详尽注释，易于理解

书中给出了典型的程序代码，而且对这些代码都进行了详尽的注释，读者阅读起来很容易理解，这对理解编程有很大的帮助。

3. 以大量实例为示范，快速掌握

无论是对基本的语法讲解，还是应用开发，书中都列举了大量实例，最后还

提供了综合实例，非常实用，读者可以通过这些例子很快掌握所学内容，学习效果好。

4．多媒体视频讲解，加速学习

丛书的每本书都配专门制作的多媒体学习光盘，方便读者学习，另外也提供了书中所涉及的源代码，以方便读者使用。

5．完善的售后服务，后顾无忧

本丛书提供技术论坛（http://www.rzchina.net）和 QQ 群（QQ 群号：21948169）答疑，读者可以在上面提问和交流。

丛书适合的读者对象

本丛书定位于如下没有编程基础的入门人员。

- ❑ 自学编程的入门人员。
- ❑ 各类程序设计爱好者。
- ❑ 想学习一门技术以便找工作的人员。
- ❑ 做课题设计和毕业设计的学生。
- ❑ 需要迅速掌握一门编程语言的人员。
- ❑ 大中专院校或电脑学校的学生。

学习建议

学习编程并不是想象的那么难，关键是要有合适的教程和正确的学习方法。如果您已经选择了这套书，那么恭喜您已经解决了第一个问题。但我们还有如下建议。

- ❑ 入门人员自然按书中安排的顺序阅读最好。
- ❑ 如果你有一定的基础，那也不妨跳跃式阅读，选择阅读。
- ❑ 先弄懂书中的内容，然后可以适当拓展。
- ❑ 要勤思考，多动手，必要时一定要上机操作。
- ❑ 要学会利用网络资源，经常光顾各技术论坛，有不懂的地方可以利用百度来搜索解决方法。

希望这套书能成为您初学编程的良师益友，能带您轻松跨入编程的大门，并逐步成为编程高手。

化学工业出版社

2009 年 5 月

前　言

Linux 操作系统是由 Linus Torvalds 于 1991 年开发的，经过十几年的发展，目前已经成为全球最受欢迎的操作系统之一。Linux 在服务器领域和桌面应用中都有杰出的表现，大至规模庞大的数据中心，小至可以置于掌心的手持设备，都可以看到 Linux 的身影。掌握 Linux 平台下的程序设计技术，是一个优秀程序员的必修课。

要学习好 Linux 平台下的程序设计技术，关键是选择一本正确的入门书籍。目前市面上关于 Linux 的编程书籍很多，但许多书不是专门为初学者写的，因此，对初学者而言，会觉得内容过多过深，难以掌握。

为了使 Linux 平台编程的初学者少走弯路，快速而熟练地掌握 Linux 编程技术，笔者总结自身十多年的 Linux 应用开发经验，向广大读者奉献本书。书中结合大量生动翔实的代码实例，使读者在学习理论知识的同时，能够灵活地与实践相结合，达到迅速掌握编程技巧的目的。

本书特点

相比同类图书，本书具有以下明显特色。

1. 内容翔实，重点突出

本书从初学者的角度出发，全面介绍了 Linux 环境下 C 语言编程的基础知识。本书涉及了 Linux 环境下编程的方方面面，在讲解过程中并不是泛泛地列举知识点，而是在对知识点进行介绍的同时，选择当前最热门、应用最广的一些技术进行了深入的讨论。

2. 概念准确，易于理解

作为一本 Linux 环境下 C 语言编程的入门图书，书中概念描述准确，让读者易于理解。本书对每个概念都使用准确而且精炼的语言总结，并结合丰富的插图，使读者易于理解编程概念。

3. 实例丰富，强调实践

为了让读者易于掌握 Linux 环境下 C 语言编程的技巧，本书列举了大量实例进行讲解。通过这些实例，读者便可更加深入地理解相关概念，从而达到熟练掌握 Linux 环境下程序设计技术的目的。另外，本书重点强调实践性，本书中的很多例子都来源于编者的实际开发，大多数实例都来源于实际项目。通过对这些例子的学习，可以增强读者的动手能力。

4. 代码规范，注释丰富

本书中程序源代码结构清晰，语句简洁，体现了良好的编码风格，有利于读者养成良好的编写代码习惯。

本书内容

本书共分为四篇，从理论介绍到具体实践，循序渐进地讲述了 Linux 系统下 C 语言的编程技术。

第一篇（第 1 章～第 2 章）介绍了 Linux 操作系统和 Shell 编程环境。

第二篇（第 3 章～第 8 章）介绍了 C 语言及其编程环境，包括 C 语言的常见内容以及常见的 vi 编辑器和 gcc 编译器，最后还详细分析了 make 的使用、程序调试、创建与使用库的方法。

第三篇（第 9 章～第 14 章）介绍了 Linux 系统下的输入输出及进程管理，包括 Linux 环境中的文件操作和标准输入输出库，同时还介绍了两种界面程序设计方法——QT 和-GTK+ 最后介绍了进程和信号等内容。

第四篇（第 15 章～第 21 章）介绍了 Linux 系统下的进程间通信及网络编程，并重点讲解了 4 种比较重要的进程间通信机制——管道、消息队列、共享内存、信号量，最后介绍了 Linux 网络环境、基本套接口编程和综合实例——银行代理收费服务器等内容。

本书读者

- Linux 系统下 C 语言编程的初学人员。
- Linux 服务器领域的开发人员。
- Linux 桌面应用的开发人员。
- 想了解 Linux 系统下 C 语言编程的其他人员。

本书编者

本书主要由宫虎波编写，其他参与编写和资料整理的人员有刘成、马臣云、潘娜、阮履学、陶则熙、王大强、王磊、徐琦、许少峰、颜盟盟、杨娟、杨瑞萍、于海波、俞菲、曾苗苗、赵莹、朱存等。

编者

2009 年 6 月

目　录

第一篇　系统环境

第 1 章　Linux 系统概述 2

1.1 计算机操作系统简介 2
1.1.1 操作系统的概念 2
1.1.2 操作系统的基本功能 3
1.1.3 主要操作系统介绍 3
1.2 Linux 操作系统介绍 4
1.2.1 Linux 的来源 4
1.2.2 什么是 Linux 5
1.2.3 Linux 的特性及优点 7
1.2.4 为什么要选择 Linux 8
1.2.5 内核的组成 8
1.3 主流 Linux 操作系统及发行版本 10
1.3.1 Linux 内核的版本 10
1.3.2 Linux 的发行版本 10
1.4 小结 11

第 2 章　shell 环境 12

2.1 shell 介绍 12
2.1.1 shell 的种类 12
2.1.2 如何进入 shell 13
2.1.3 如何使用 shell 14
2.2 shell 编程基础 15
2.2.1 创建和运行 shell 脚本程序 15
2.2.2 shell 环境变量 16
2.2.3 常用的 shell 命令 18
2.2.4 管道与重定向的使用 19
2.2.5 shell 变量的使用 21
2.2.6 shell 运算符的应用 22
2.2.7 在 shell 脚本中进行条件控制 23
2.2.8 在 shell 脚本中使用 for 循环 25

2.2.9 在 shell 脚本中使用 while 循环 26
2.2.10 在 shell 脚本中使用 until 循环 27
2.2.11 在 shell 脚本中使用函数 27
2.3 综合实例 28
2.3.1 实例需求 28
2.3.2 系统设计 28
2.3.3 程序代码 28
2.4 小结 33

第二篇 C 语言及编程环境

第 3 章 C 语言简介 36

3.1 C 语言概述 36
3.1.1 C 语言的发展简史 36
3.1.2 C 语言的特点 37
3.2 C 语言的组成元素 37
3.2.1 字符集 37
3.2.2 标识符 38
3.2.3 关键字 38
3.2.4 常量的类型 38
3.2.5 变量的类型 39
3.2.6 变量的存储类型 40
3.2.7 变量的作用域 42
3.2.8 运算符 42
3.2.9 注释方法 43
3.3 语句与控制结构 43
3.3.1 表达式语句的组成 43
3.3.2 复合语句 44
3.3.3 函数调用语句 44
3.3.4 控制语句 44
3.3.5 循环控制语句 47
3.3.6 转移语句 48
3.4 函数与程序结构 48
3.4.1 库函数 48
3.4.2 用户自定义函数 49
3.5 数组 49
3.5.1 一维数组的定义和使用 49

3.5.2 多维数组的定义和使用 50
3.6 结构 52
3.6.1 结构的定义 52
3.6.2 结构成员的引用 53
3.7 指针 53
3.7.1 指针的概念 54
3.7.2 指针的定义和使用 54
3.7.3 指针变量的运算 55
3.8 综合实例 56
3.8.1 冒泡排序算法原理 56
3.8.2 冒泡排序算法实现 56
3.9 小结 58
第 4 章 vi 编辑器 59
4.1 vi 编辑器概述 59
4.1.1 vi 的启动 59
4.1.2 vi 的操作方式 59
4.1.3 vi 编辑器的功能键 60
4.1.4 退出 vi 编辑器 61
4.2 vi 编辑器的命令 62
4.2.1 光标移动命令 62
4.2.2 滚动屏幕命令 62
4.2.3 文本编辑命令 63
4.2.4 文本删除命令 63
4.2.5 文本修改命令 64
4.2.6 文本移动命令 64
4.2.7 搜索命令 65
4.2.8 ex 转义命令 65
4.3 vi 编辑器的选项 66
4.3.1 选项的含义 66
4.3.2 选项的设置方式 67
4.4 小结 68
第 5 章 gcc 编译器 69
5.1 编译过程简述 69
5.1.1 预编译过程 69
5.1.2 编译的过程 71
5.1.3 优化及汇编的过程 71

5.2 链接过程简述 72
5.2.1 链接的过程 72
5.2.2 静态链接与动态链接 73
5.3 gcc 编译器简述 73
5.3.1 程序的编译与链接 74
5.3.2 gcc 编译器的工作过程 74
5.4 gcc 编译器语法 75
5.4.1 常用语法 75
5.4.2 用 gcc 编译器生成可执行文件 76
5.4.3 用 gcc 编译器生成动态链接库 76
5.4.4 如何使用动态链接 77
5.4.5 gcc 编译器常见错误排除 78
5.5 小结 78
第 6 章 make 的使用 79
6.1 makefile 介绍 79
6.2 makefile 的书写规则 80
6.2.1 基本语法规则 80
6.2.2 定义变量 80
6.2.3 环境变量 81
6.2.4 通配符的使用 81
6.2.5 使用条件判断 82
6.2.6 在 makefile 中使用函数 82
6.2.7 使用 make 与直接使用 gcc 脚本的区别 84
6.3 make 工具 84
6.3.1 运行 make 命令 84
6.3.2 make 命令的工作过程 85
6.3.3 在 makefile 中使用伪目标 85
6.3.4 make 命令的返回值 86
6.4 综合实例 86
6.4.1 makefile 应用的环境 86
6.4.2 makefile 的实现及解释 86
6.5 小结 87
第 7 章 程序调试 88
7.1 错误处理 88
7.1.1 使用标准错误输出 88
7.1.2 使用 errno 全局变量 89

7.1.3 使用错误信号处理.....90
7.1.4 使用 assert 断言.....91
7.1.5 内存泄露的检查.....92
7.1.6 其他可能的内存错误.....93
7.2 gdb 介绍.....94
7.2.1 利用 gdb 调试的前提.....94
7.2.2 启动 gdb 的方法.....94
7.2.3 gdb 的基本功能.....95
7.3 使用 gdb 进行调试实例.....97
7.4 小结.....98
第 8 章 创建与使用库.....99
8.1 函数库介绍.....99
8.1.1 系统函数库的使用.....99
8.1.2 用户自定义函数库的创建和使用.....100
8.2 库函数与系统调用.....102
8.2.1 系统调用介绍.....103
8.2.2 库函数介绍.....104
8.3 动态库的创建与使用.....105
8.4 综合实例.....108
8.5 小结.....109
第三篇 输入输出及进程管理
第 9 章 文件操作.....112
9.1 Linux 文件系统简述.....112
9.1.1 逻辑磁盘分区管理.....112
9.1.2 文件系统的建立与挂载.....114
9.1.3 虚拟文件系统.....117
9.1.4 ext2 文件系统.....117
9.1.5 文件类型.....119
9.1.6 文件权限管理.....120
9.2 文件基本操作.....125
9.2.1 文件编程的基本概念.....125
9.2.2 文件的创建与打开.....126
9.2.3 文件的读写.....130
9.2.4 文件的关闭与删除.....133

9.2.5 文件的随机存取.....135
9.3 文件安全编程.....137
9.3.1 文件的属主及用户组编程.....137
9.3.2 设置文件权限（UGO 模式）.....139
9.3.3 设置文件权限（ACL 模式）.....141
9.4 文件属性编程.....145
9.5 目录编程.....148
9.6 综合实例.....152
9.6.1 ELF 文件格式.....152
9.6.2 程序实现.....152
9.7 小结.....156
第 10 章 标准输入输出库.....157
10.1 标准 I/O 的基本概念.....157
10.1.1 流.....157
10.1.2 缓存.....158
10.1.3 标准输入、标准输出及标准错误输出.....160
10.2 使用标准 I/O 进行文件操作.....162
10.2.1 打开关闭流文件.....162
10.2.2 单字符方式读写.....162
10.2.3 行方式读写.....164
10.2.4 二进制方式读写.....166
10.2.5 格式化输入/输出.....168
10.2.6 在流文件中定位.....170
10.3 综合实例.....171
10.4 小结.....174
第 11 章 界面程序设计——Qt.....175
11.1 Qt 简述.....175
11.1.1 Qt 的组成.....175
11.1.2 Qt 的优点.....175
11.2 Qt 开发包的安装.....176
11.3 Qt 集成开发环境介绍.....177
11.3.1 启动设计器.....177
11.3.2 设计器界面元素介绍.....178
11.4 Qt 程序开发.....179
11.4.1 建立新项目.....179
11.4.2 设计窗口.....179

11.4.3 添加事件处理程序.....181
11.4.4 添加主程序.....182
11.5 Qt 程序的生成.....183
11.6 小结.....184
第 12 章 界面程序设计——GTK+.....185
12.1 GNOME 与 KDE.....185
12.1.1 Linux 图形界面原理.....185
12.1.2 X 协议.....185
12.1.3 GNOME 与 KDE 的启动.....186
12.1.4 GNOME 与 KDE 的区别.....187
12.2 使用 GTK+进行开发.....187
12.2.1 GTK+的安装.....187
12.2.2 GTK+程序的初始化与退出.....187
12.2.3 GTK+的事件处理.....188
12.2.4 使用 GTK+实现 HelloWorld.....189
12.2.5 编译 GTK+程序.....191
12.2.6 在 GTK+中使用控件.....191
12.3 综合实例.....192
12.3.1 实例需求.....192
12.3.2 实例代码及解释.....192
12.4 小结.....194
第 13 章 进程.....195
13.1 进程的基本概念.....195
13.1.1 进程的属性.....195
13.1.2 进程的内存映像.....197
13.1.3 进程组.....198
13.1.4 进程的会话.....199
13.1.5 进程的控制终端.....201
13.1.6 进程的状态.....202
13.1.7 进程的优先级.....204
13.2 进程的运行环境.....206
13.2.1 进程的入口函数.....206
13.2.2 进程的环境变量.....211
13.2.3 进程的内存分配.....213
13.3 进程的创建.....214
13.3.1 调用 fork 创建进程.....214

13.3.2 调用 exec 系列函数执行程序 216
13.3.3 调用 system 创建进程 219
13.4 进程的终止 220
13.4.1 调用 exit 退出进程 220
13.4.2 调用 wait 等待进程退出 220
13.5 小结 222

第 14 章 信号 223

14.1 信号的基本概念 223
14.1.1 信号的定义 223
14.1.2 信号的来源 224
14.1.3 信号的分类 224
14.2 信号的安装及处理 226
14.2.1 信号的处理方式 226
14.2.2 用 signal 安装信号 226
14.2.3 用 sigaction 安装信号 228
14.2.4 信号的阻塞处理 230
14.2.5 信号集的操作 232
14.2.6 未决信号的处理 233
14.2.7 等待信号 234
14.2.8 信号处理函数的实现 236
14.3 信号的发送 238
14.3.1 使用 kill 发送信号 238
14.3.2 使用 sigqueue 发送信号 239
14.4 SIGALRM 信号 240
14.4.1 安装 SIGALRM 信号 240
14.4.2 设置定时器 241
14.5 SIGCLD 信号 242
14.5.1 子进程的退出过程 242
14.5.2 SIGCLD 信号的处理 243
14.6 小结 244

第四篇 进程间通信(IPC)及网络编程

第 15 章 进程间通信——管道 246

15.1 进程间通信概念 246
15.2 管道的概念及分类 247

15.2.1 管道的概念及特点.....247
15.2.2 管道的分类.....248
15.3 管道编程.....249
15.3.1 创建管道.....249
15.3.2 读写管道.....251
15.3.3 关闭管道.....253
15.3.4 管道 I/O.....253
15.4 命名管道编程.....255
15.4.1 创建管道.....255
15.4.2 打开管道及读写.....257
15.4.3 管道的删除.....259
15.5 小结.....259
第 16 章 进程间通信——消息队列.....260
16.1 System V 进程间通信概述.....260
16.1.1 Shell 环境控制 IPC.....260
16.1.2 进程间通信关键字.....262
16.1.3 进程间通信标识符.....262
16.1.4 IPC 权限许可结构.....263
16.2 消息队列基本概念.....264
16.2.1 队列.....264
16.2.2 消息.....264
16.2.3 消息队列.....265
16.3 消息队列编程.....266
16.3.1 键值生成函数.....266
16.3.2 创建消息队列.....267
16.3.3 消息发送.....270
16.3.4 消息接收.....272
16.3.5 控制消息队列.....274
16.4 小结.....276
第 17 章 进程间通信——共享内存.....277
17.1 共享内存基本概念.....277
17.1.1 共享内存编程模型.....277
17.1.2 共享内存的映射.....277
17.1.3 共享内存数据结构.....278
17.2 共享内存编程.....279
17.2.1 创建共享内存.....279

17.2.2 映射共享内存.....281
17.2.3 删除共享内存映射.....282
17.2.4 控制共享内存.....283
17.3 小结.....285
第 18 章 进程间通信——信号量.....286
18.1 PV 操作原理.....286
18.1.1 PV 操作的来源.....286
18.1.2 PV 操作的定义.....287
18.1.3 PV 操作的应用.....288
18.2 信号量基本概念.....290
18.2.1 Linux 信号量简介.....290
18.2.2 信号量的控制结构.....291
18.3 信号量编程.....291
18.3.1 创建信号量.....291
18.3.2 信号量操作.....293
18.3.3 信号量控制.....296
18.4 综合实例——利用信号量实现生产者-消费者模型.....299
18.4.1 需求.....300
18.4.2 需求分析与设计.....300
18.4.3 实现代码及分析.....300
18.5 小结.....303
第 19 章 Linux 网络环境.....304
19.1 计算机网络基础.....304
19.1.1 计算机网络分类.....304
19.1.2 网络拓扑结构.....305
19.1.3 网络通信协议.....306
19.1.4 OSI 参考模型.....307
19.2 TCP/IP 协议概述.....309
19.2.1 TCP/IP 分层模型.....309
19.2.2 TCP/IP 协议族.....310
19.2.3 网络地址.....311
19.2.4 端口.....313
19.3 客户机/服务器模型.....314
19.4 传输控制协议.....316
19.4.1 连接建立.....316
19.4.2 连接关闭.....316

19.4.3 TCP 数据报格式 318
19.5 用户数据报协议 319
19.6 小结 319
第 20 章 基本套接口编程 320
20.1 套接口编程简述 320
20.1.1 半相关与全相关 321
20.1.2 地址族与协议族 321
20.1.3 面向连接与面向无连接 322
20.1.4 套接口类型 322
20.1.5 字节序 323
20.1.6 套接口连接方式 324
20.1.7 数据传输方式 325
20.2 套接口数据结构 326
20.2.1 套接口地址结构 326
20.2.2 通用套接口地址结构 327
20.2.3 主机名称数据结构 327
20.2.4 服务名称数据结构 329
20.2.5 通用数据收发结构 330
20.3 基本套接口函数 331
20.3.1 字节操作函数 331
20.3.2 字节序操作函数 333
20.3.3 地址转换函数 334
20.3.4 套接口函数 337
20.4 套接口选项 352
20.4.1 套接口选项函数 352
20.4.2 SO_KEEPALIVE 选项 352
20.4.3 SO_LINGER 选项 353
20.4.4 SO_RCVBUF 和 SO_SNDBUF 选项 355
20.4.5 SO_RCVTIMEO 和 SO_SNDTIMEO 选项 356
20.4.6 SO_REUSEADDR 和 SO_REUSEPORT 选项 357
20.5 TCP 套接口编程 357
20.5.1 重复服务器编程 357
20.5.2 并发服务器编程 357
20.6 UDP 套接口编程 362
20.6.1 UDP 编程模型 363
20.6.2 UDP 客户/服务器编程 363
20.7 小结 366

第 21 章　综合实例——银行代理收费服务器 367

21.1　程序需求 367

21.2　程序实现 367

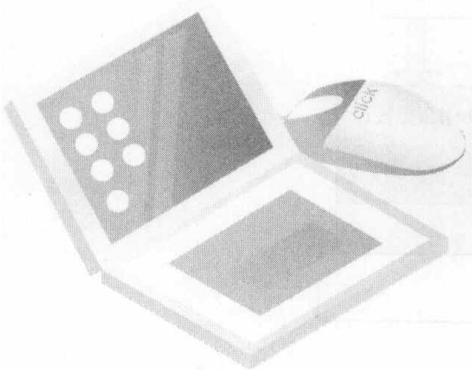
21.3　小结 379

附录　Linux 下常见 C 函数字母索引 380

第一篇　系统环境

第 1 章　Linux 系统概述

第 2 章　shell 环境



第 1 章　Linux 系统概述

自上世纪 90 年代初，Linux 操作系统以其强大的功能和开源的特性以惊人的速度占领了服务器市场和桌面应用领域。Linux 被业界认为是最有前途的操作系统之一，有人断言 Linux 迟早将会代替 UNIX 在服务器市场的龙头老大的地位。本章简要介绍了 Linux 操作系统的功能及特点。通过本章的学习，读者应重点掌握如下内容。

- 理解计算机操作系统的概念与作用。
- 初步了解 Linux 操作系统的基本功能。
- 对 Linux 的组成及内核工作原理有基本的了解。

1.1　计算机操作系统简介

计算机系统由硬件和软件两部分组成。操作系统是计算机中的底层软件，其他软件如数据库系统、编译系统和各种应用软件都依赖于操作系统的支持。

1.1.1　操作系统的概念

操作系统是用户与计算机硬件系统间的接口，是用于控制和管理系统资源，方便用户使用计算机程序的集合。通过操作系统，用户可以安全、快捷地操纵计算机硬件系统并运行自己的程序。同时，操作系统也便于对计算机系统的各项资源进行管理。

用户使用计算机操作系统主要通过两种方式进行：一是通过执行 shell 命令方式，典型的如 DOS 操作系统的 command，UNIX 及 Linux 操作系统的 bsh、csh 等；二是编写程序，通过调用操作系统提供的系统调用接口，访问操作系统的各种资源。用户使用操作系统的两种方式如图 1-1 所示。

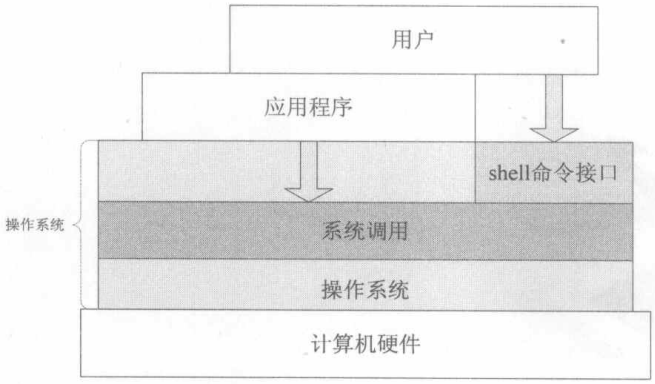


图 1-1　操作系统接口

1.1.2　操作系统的基本功能

典型的计算机系统中，通常包含多种软、硬件资源，如处理器（CPU）、存储器（内存、磁盘存储器）、输入/输出设备、数据或程序等。其组织结构如图 1-2 所示。

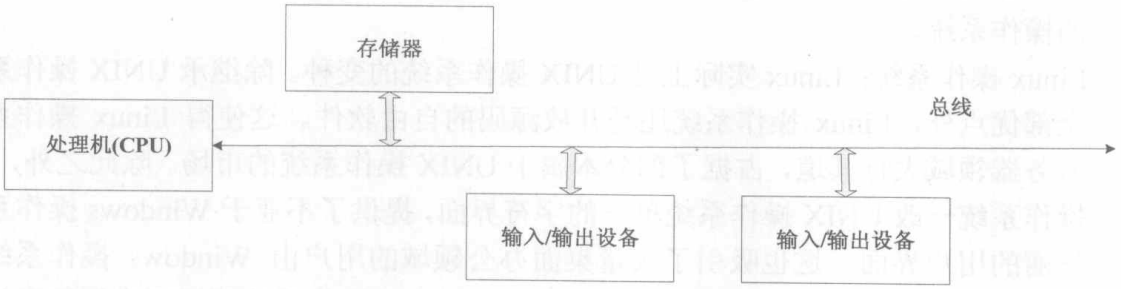


图 1-2　计算机系统的组织结构

操作系统的主要功能正是对上述这些资源进行有效管理。简要地说，操作系统的基本功能包括以下几点。

- 处理机管理：对处理机进行分配，并对其运行进行有效的控制和管理。通常情况下，处理机的分配以进程为单位，所以对处理机的管理也可以归结为对进程的管理。
- 存储器管理：负责内存的分配与回收。为程序的运行分配内存空间，为程序提供良好的运行环境，方便用户合理使用存储器。
- 输入/输出设备管理：负责输入/输出设备的分配与控制，如向打印机打印输出、从键盘输入数据等。
- 文件管理：负责文件的存取。通常情况下，程序和数据总是以文件的形式存储在磁盘等介质中供用户使用。文件管理的主要任务，就是对文件进行有效的管理，以方便用户使用。

1.1.3　主要操作系统介绍

操作系统从诞生至今已有近 50 年的历史。上世纪 60 年代至 70 年代是操作系统发展的活跃时期。到上世纪 80 年代，操作系统已发展得比较成熟。经过多年的发展，操作系统的功能日趋完善，性能更加优化。按照通常的划分方法，操作系统可以划分为以下几种。

- MS-DOS 操作系统：提到 MS-DOS，读者可能会立即回忆起那熟悉的“C>:”提示符。MS-DOS 是在 1981 年推出的，随 IBM 公司的 IBM-PC 计算机一起发行。随着 IBM-PC 计算机的畅销，同时也由于 MS-DOS 强大的功能，MS-DOS 成为 16 位微型计算机上操作系统的标准。
- Windows 系列操作系统：1990 年，Microsoft 推出了 Windows 3.0 版本，它以其友好的图形界面征服了广大计算机用户。该操作系统同时具有支持多任务、易学好用的特点，因而在微机领域迅速流行。1993 年推出的 Windows NT 是真正的 32 位多任务的操作系统，在服务器操作系统领域占有了很大市场。后来从 Windows 95、Windows 98、Windows XP 一直发展至现在的 Windows Vista，Windows 系列操作系

统在微型机领域仍然是绝对的主流。

- UNIX 操作系统：UNIX 操作系统是由美国贝尔实验室开发的多用户、多任务的操作系统。本书将要介绍的Linux操作系统也是在UNIX系统的基础上发展而来。UNIX操作系统不仅可以在微机上使用，而且支持小型机乃至大型计算机。同时，UNIX操作系统具有优良的性能、强大的安全控制机制，到目前仍然是服务器领域最重要的操作系统。
- Linux 操作系统：Linux 实际上是 UNIX 操作系统的变种。除继承 UNIX 操作系统的全部优点外，Linux 操作系统还是开放源码的自由软件。这使得 Linux 操作系统在服务器领域大行其道，占据了部分本属于 UNIX 操作系统的市场。除此之外，Linux操作系统一改 UNIX 操作系统单一的字符界面，提供了不亚于 Windows 操作系统的华丽的用户界面，这也吸引了大量桌面办公领域的用户由 Windows 操作系统转向 Linux 阵营。本书将要介绍的内容，就是在 Linux 操作系统环境下进行程序设计的方法与技巧。

1.2 Linux 操作系统介绍

Linux 操作系统的出现，是源于荷兰赫尔辛基大学的学生 Linus Torvalds。他想要开发一个 Minix 操作系统的替代品，可用于 x86 体系结构的个人计算机上，并且具有 UNIX 操作系统的全部功能。这就是 Linux 诞生的开端。

1.2.1 Linux 的来源

Linux 的来源最早要追溯到 UNIX 操作系统。1969 年，贝尔实验室的 Ken Thompson 利用一台闲置的 PDP-7 计算机开发了一种多用户、多任务的操作系统。后来，另一个研究人员 Dennis Richie 加入了该项目，在他们的共同努力下诞生了最早的 UNIX 操作系统。UNIX 操作系统最早是用汇编语言实现的。

UNIX 操作系统诞生后，由于其强大的功能和优良的性能，很多商业公司及研究机构纷纷加入到这个操作系统的研发中如此一来，产生了大量的 UNIX 版本，如 AT&T 的 System V、加州大学的 BSD、IBM 的 AIX 等。这些操作系统版本之间互不兼容，只支持基于各自硬件体系结构的计算机系统，并且不支持个人计算机。这一情况成为 Linux 操作系统诞生的重要因素。

Linux 的诞生源于荷兰的一个大学生的需求。1991 年，荷兰赫尔辛基大学的学生 Linus Torvalds 需要一款多用户、多任务的操作系统。限于其财力有限，所能购买得起的只有 Minix 操作系统。而 Minix 功能简单，在当时是被用于教学的一个类似于 UNIX 的操作系统。Linus Torvalds 对 Minix 很不满意，决定自己编写一款新的操作系统。

由于 UNIX 操作系统具有优良的性能，Linus Torvalds 决定以 UNIX 操作系统为原型开始进行开发工作，并且进展很快。到 1991 年，这个操作系统的 0.02 版已经编写完成，被命名

为 Hobby。受开放源码思想的影响，Linus Torvalds 将他新编写的操作系统放置到互联网上并开放源码与其他开发人员共享。随着大量开发人员的加入，使得这个新的系统发展很快，并最终成为一个真正意义上的操作系统。这个操作系统就是 Linux 的最早版本。

提示：1991 年，Linus Torvalds 在 BBS 中发布了一个公告（如图 1-3 所示），成为 Linux 系统诞生的重要标志。其中文意思大致为：各位，我正在开发一款操作系统（只是个人兴趣，不会做得像 GNU 那么大那么专业）。从这段公告可以看出，Linus Torvalds 也没有想到，Linux 会发展到今天这个程度。

这个新诞生的操作系统是基于 UNIX 操作系统的，所以被称为 Linus's UNIX，简称为 Linux。其标志是一个活泼可爱的企鹅形象，如图 1-4 所示。

Hello everybody out there using minix-
I'm doing a (free) operation system (just a hobby,
won't be big and professional like gnu) for 386(486) AT
clones.

图 1-3　Linus 的 BBS 公告



图 1-4　Linux 的形象

1.2.2　什么是 Linux

Linux 是一个多用户、多任务的操作系统。它由 UNIX 操作系统发展而来，但又比 UNIX 操作系统功能更为强大，尤为重要的是它可以免费使用并自由传播。由于 Linux 的开源特性，自从 Linux 诞生以来，来自世界各地的程序开发人员不断对其功能进行丰富完善。发展到现在，Linux 已经成为服务器领域及桌面应用领域非常重要的操作系统之一。

Linux 具有十分重要的用途，它被广泛应用于服务器联机事务处理、办公自动化桌面应用及个人工作站等领域。由于 Linux 是一种类似于 UNIX 的操作系统，这就使得其在服务器领域与 UNIX 操作系统具有同等的功能。加上 Linux 的开源特性，在服务器领域的联机事务处理方面，Linux 已经取得非凡的成绩。而在办公自动化方面，Linux 提供了 KDE、GNOME 等大量的图形操作环境供用户使用。其应用领域如图 1-5 所示。

Linux 主要由 4 部分内容组成：系统内核、shell、文件系统和实用工具。内核可以看作是操作系统的“心脏”，它对外提供一种称为“系统调用”的接口，供外部程序调用内核提供的服务。其余的三部分（shell、文件系统和实用工具）都是通过访问系统调用实现各自功能的。

系统内核是 Linux 操作系统的核心，它包含了诸如进程管理、存储管理等核心功能的处理。Linux 自诞生以来，产生了多种不同版本的内核，目前最新的内核是 2.6 版本。Linux 操作系统内核的版本在系统登录时可以看到，如图 1-6 所示，该系统的内核版本为 2.6.5。该版本号也可以通过 Linux 执行命令 uname –r 获得。

shell 是操作系统对用户提供的交互操作的接口，它的作用类似于 DOS 操作系统的 command。shell 接收用户输入的命令并提交给系统内核去执行。传统意义上的 shell 是基于字符界面的，较常见的 shell 如 Bourne shell、C shell、Korn shell 等。Bourne Shell 的操作界面如

图 1-7 所示。

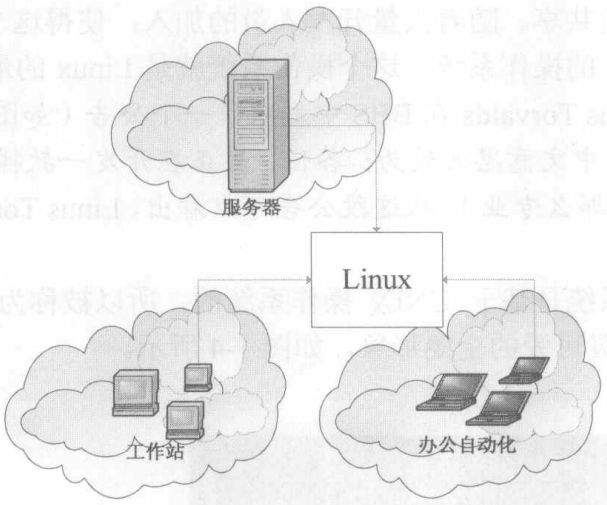


图 1-5　Linux 应用领域

```
Welcome to SUSE LINUX Enterprise Server 9 (i586) - Kernel 2.6.5-7.97-default (0.
linux login:
```

图 1-6　Linux 内核版本

```
linux login: root
Password:
You have new mail in /var/mail/root.
Last login: Mon Nov 10 20:21:07 from 192.168.1.101
linux:~ #
```

图 1-7　Bourne Shell 操作界面

除这些字符界面的 shell 外，Linux 还提供了像 KDE、GNOME 等的图形操作界面。这些图形操作环境提供了一种可视化的操作方式供用户使用，这与 Windows 操作系统颇为类似。如图 1-8 所示是典型的 KDE 图形操作界面。

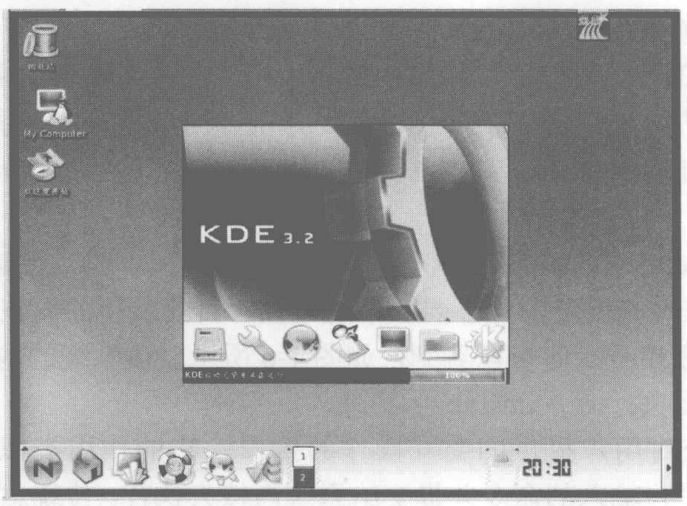


图 1-8　KDE 图形操作界面

文件系统用于管理存储在磁盘设备上的文件。Linux 文件系统是基于树形结构的，支持文件和目录。Linux 操作系统基本的目录结构可以通过执行 tree 命令输出，如图 1-9 所示。在 Linux 系统中，一切文件，包括对硬件设备的操作也抽象为对设备文件的操作。为保证安全性，Linux 提供了复杂的安全管理功能。它既支持传统的 UNIX 操作系统下的文件权限方式，也支持其独特的访问控制列表（ACL）方式。

```
.
|-- bin
|-- boot
|   |-- boot -> .
|   `-- grub
|-- dev
|   |-- ataraid
|   |-- button
|-- etc
|   |-- SuSEconfig
|   |-- X11
|   |-- YaST2
|-- home
|   `-- test
|-- lib
|   |-- YaST
|   |-- evms
|   |-- i686
|-- media
|   |-- cdrecorder
|   `-- floppy
|-- mnt
|-- opt
|   |-- gnome
|   `-- kde3
|-- proc
|   |-- 1
|-- root
|   |-- Desktop
|   `-- bin
|-- sbin
|   `-- conf.d
|-- srv
|   |-- ftp
|   `-- www
|-- sys
|   |-- block
|   |-- bus
|-- usr
|   |-- X11R6
|   |-- bin
`-- var
    |-- X11R6
    |-- adm
```

图 1-9　Linux 的目录结构

实用工具是 Linux 系统提供给用户使用的各种工具软件。在不同厂商的操作系统中，这些实用工具各不相同，如编辑工具软件、计算器工具软件等。

1.2.3　Linux 的特性及优点

Linux 以它的高效和灵活性著称。在服务器领域，它实现了 UNIX 操作系统的全部功能，而且由于它的开源特性，Linux 提供了比 UNIX 系统更多的应用支持软件。在桌面应用领域，Linux 提供了丰富的界面编程、多媒体编程工具，具有如同 Windows 操作系统一样的强大功能。

Linux 主要具有以下功能。

- 存储管理：Linux 系统利用芯片的保护模式，在存储管理中采用了页式存储管理。
- 进程管理：进程是 Linux 操作系统的核心，所有的任务都是以进程出现的。
- 文件系统：Linux 系统提供了对各种文件系统的支持，甚至可以在 Linux 下访问 Windows 操作系统的文件。
- 网络通信：Linux 系统提供了强大的网络通信功能，支持 TCP/IP 协议及其他多种网络通信协议。

- 多媒体功能：Linux 提供了多种多样的多媒体功能，支持各种音频、视频设备及即插即用设备，并且提供了强大的界面开发工具包，可以开发出与 Windows 相媲美的桌面应用程序。

Linux 具有以下优点。

- 真正支持多任务、多用户：Linux 的内核采用了基于时间片的任务调度方式，这也是 UNIX 系统所采用的调度方式。在多 CPU 的环境下，可以实现真正的多任务并发处理。
- 支持多种文件系统：Linux 支持多达 32 种以上的文件系统，这一数字还在不断增加。Linux 自身提供的文件系统包括 ext2、ext3 和 reiserfs 文件系统。其他的文件系统像 Windows 的 FAT 和 NTFS，光盘文件系统 ISO-9660 等都在 Linux 系统下获得广泛支持。
- 开源特性：可以免费获取整个操作系统的源码并支持用户自行修改。这是 Linux 最吸引人的地方之一。也正是由于这一点，Linux 才能取得今天的成就。
- 运行稳定，功能强大且性能优异：Linux 是由 UNIX 系统派生而来，继承了 UNIX 系统稳定、高效的优点。运行 Linux 操作系统的主机可以保持若干年长时间高效运行而不宕机。
- 免费使用：Linux 是基于 GPL 的，因此任何人或者组织都可以无偿获取和使用 Linux，而不必支付大量的版权费用。

1.2.4 为什么要选择 Linux

目前主流的操作系统有很多，且各自有其吸引用户的特性。为何要选择 Linux 操作系统呢？Linux 之所以受到用户的青睐，主要基于以下两方面的原因。

- Linux 属于开源的自由软件。用户无须支付任何费用即可拥有操作系统及其源码。如果有必要，甚至可以根据用户的需要自行对操作系统进行修改，从而打造适合用户自身需要的操作系统。从另一方面看，也更适合用户学习、掌握 Linux 系统。
- Linux 功能强大。它实现了服务器操作系统 UNIX 的全部功能。不仅如此，它还在 UNIX 系统的基础上对操作系统进行了扩充，提供了比 UNIX 系统更为丰富的应用软件。

基于上述两方面的原因，Linux 成为时下主流的操作系统，并且在瓜分 Windows 桌面应用市场的同时，大有替代 UNIX 系统在服务器市场地位的趋势。作为应用开发人员，掌握好 Linux 系统下的编程显得更加重要。况且，Linux 与 UNIX 系统的编程接口是完全兼容的，在学习 Linux 的同时，也掌握了 UNIX 系统的编程方法，何乐而不为。

1.2.5 内核的组成

Linux 内核是一个非常复杂的软件系统，在整个操作系统中处于核心地位。内核主要由进程调度、内存管理、虚拟文件系统和进程间通信机制等模块组成。

进程调度用于控制进程对 CPU 资源的访问。在多任务操作系统中，系统中同时存在多个可以被调度的任务。系统需要根据一定的策略选择占用 CPU 资源投入运行的进程，如图 1-10 所示。

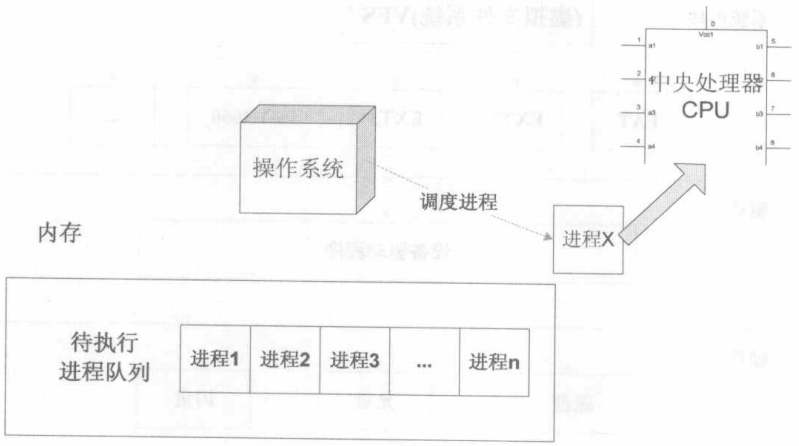


图 1-10　进程调度

如果某个进程在等待其他资源，则该进程将处于睡眠状态。当所等待的资源就绪后，该进程再被唤醒。Linux 将 CPU 的资源划分成大小相同的时间片（time slice），每个占用 CPU 的进程运行完一个时间片后将释放 CPU 资源供其他进程使用。在调度策略上，Linux 使用按照优先级的进程调度算法选择新的进程。其主要有 3 种调度方法：分时调度（SCHED_OTHER）、先到先服务的实时调度（SCHED_FIFO）、时间片轮转的实时调度（SCHED_RR）。

内存是 Linux 管理最重要的资源之一。Linux 支持虚拟内存的方式进行内存管理。所谓虚拟内存，就是将磁盘当做内存的扩展。一个程序被调入内存后，如果其程序、堆栈和数据的总量超过实际内存的大小，内核只把当前使用的程序块保留在内存中，其余的内容则保留在磁盘中。在磁盘中的内容需要执行时，内核再负责在磁盘和内存间交换数据。这里的磁盘就是 Linux 交换区。在命令提示符下输入 fdisk –l 命令可以看到有一行名称为 Linux swap 的设备，这就是 Linux 系统的交换区。fdisk 命令的输出如下所示。

```
Disk /dev/sda: 2147 MB, 2147483648 bytes
255 heads, 63 sectors/track, 261 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Device    Boot      Start         End      Blocks   Id  System
/dev/sda1               1          32      257008+  82  Linux swap
/dev/sda2   *          33         261     1839442+  83  Linux
Disk /dev/sdb: 1073 MB, 1073741824 bytes
255 heads, 63 sectors/track, 130 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
```

Linux 内核基本上对所有文件系统都提供了强大支持。为隐藏各种硬件的具体细节和所有的文件系统提供统一的接口，Linux 采用了一种称为虚拟文件系统（VFS）的机制。通过 VFS，Linux 向用户提供了一个统一的访问接口，如图 1-11 所示。

Linux 系统内核支持多种进程间通信机制，主要包括用于本机进程间通信的消息队列、共享内存、信号量和用于网络环境下进程间通信的套接口。除 UNIX 系统中所支持的 System

V 进程间通信机制外，Linux 还支持 POSIX 进程间通信机制。

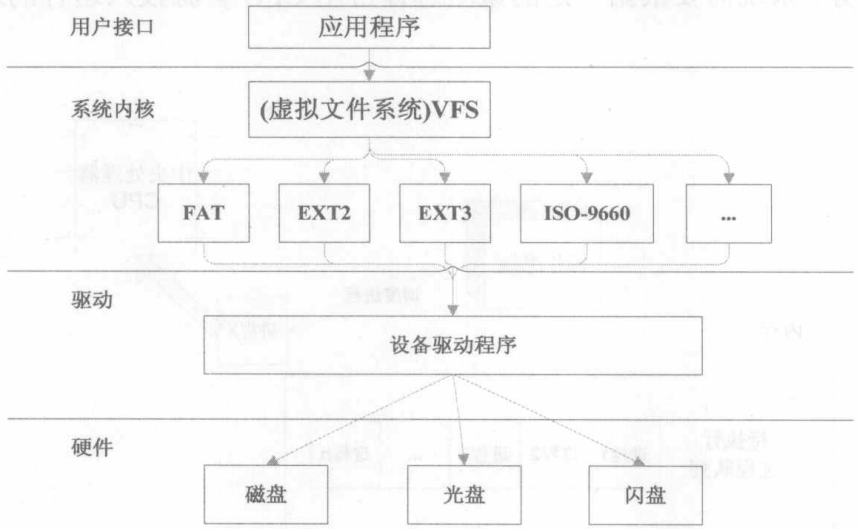


图 1-11　虚拟文件系统

1.3　主流 Linux 操作系统及发行版本

Linux 操作系统自产生以来，随着功能的不断发展完善，产生了多个版本。同时，各个不同的 Linux 厂商也分别在核心的基础上发行了各自的操作系统。

1.3.1　Linux 内核的版本

Linux 自发行以来，随着功能的日渐完善，其内核版本也经过多次升级，由最初的 0.01 版本到目前最新的 2.6 版本。读者需要注意的是，这里的版本与操作系统的发行版本是不同的概念。不同的 Linux 版本的操作系统，使用的内核版本可能是相同的。

最早的 Linux 内核版本是 0.01 版本，它是于 1991 年 9 月发行，是正式对外公布的第一个 Linux 内核版本。到目前为止，Linux 最新的内核版本为 2.6 版本，这也是功能最为强大、最为成熟的一个版本。相信随着时间的推移，在广大 Linux 爱好者的努力下，将会有更多功能强大、性能稳定的内核版本推陈出新。

1.3.2　Linux 的发行版本

随着 Linux 的不断发展，众多的软件厂商参与到 Linux 中来。每个厂商都在统一的规范内开发出各自的 Linux 操作系统，它们在功能上都是兼容的。Linux 主要的发行版本包括以下几个。

- RedHat Linux：由 Red Hat 公司发行。目前 RedHat 有两个版本，一个是收费的

RedHat Enterprise Linux，另外一个就是免费的 Fedora。

- SuSe Linux：德国著名的 Linux 发行版本，在世界范围内也享有很高声誉，特别是其开发的 yast 管理系统。笔者所使用的就是 SuSeLinux 9 版本。
- Debian Linux：Debian 是完全遵循 GNU 规范的 Linux 操作系统，并且 100%免费，它在世界范围内有庞大的用户群。
- Ubuntu Linux：Ubuntu 基本具有 Debian 的全部优点，在 Debian 的基础上又有所扩充，特别是 Ubuntu 的安装配置过程非常人性化。

除此之外，还有许多，如 Turbo Linux、gentoo Linux 等，在此不再一一列举。

要注意 Linux 内核版本与发行版本的区别。不同发行版本的 Linux 系统，其采用的内核版本可能是相同的。如 Suse Linux 版本 9 采用的是 Linux2.6 内核，而 Ubuntu Linux 版本 8 采用的也是 2.6 版本的内核。这里的版本 9 和版本 8 都是指定特定厂商的发行版，而 2.6 指的是 Linux 的内核版本。可以通过执行命令 uname –a 相看当前操作系统的内核版本，其输出如下所示。

```
Linux linux 2.6.5-7.97-default #1 Fri Jul 2 14:21:59 UTC 2004 i686 i686 i386 GNx
```

1.4　小结

Linux 操作系统功能强大、性能稳定，在服务器领域和桌面办公领域都有杰出的表现。再加上 Linux 的开源特性，Linux 必将得到更快的发展。因此，掌握 Linux 系统下的编程非常重要。本章简要介绍了操作系统的基本概念及 Linux 操作系统的基础知识，通过本章的学习，读者对 Linux 操作系统有了初步的了解，这将为后续编程环节的学习打好基础。

第 2 章　shell 环境

shell 是 Linux 为方便用户访问核心服务而提供的命令接口。大多数情况下，用户要使用 Linux 提供的各种服务必须通过 shell 方可进行。本章介绍了 Linu 系统下 shell 的功能及特点，重点讲解了 Linux 系统下使用 shell 进行编程的方法。通过本章的学习，读者应重点掌握如下内容。

- ❑ 了解 shell 的基本功能。
- ❑ 掌握 shell 的使用方法。
- ❑ 掌握使用 shell 进行编程的方法。

2.1　shell 介绍

任何操作系统都提供命令访问接口供用户使用。通过命令访问接口，用户可以管理与控制操作系统，实现特定的功能。不同的操作系统有着不同的命令访问接口，如 DOS 环境下是通过 command 访问系统的。在 Linux 系统下，这一接口称作 shell。本节将对登录 shell 的过程及如何使用 shell 进行简要介绍。

2.1.1　shell 的种类

shell 的中文字面含义是“外壳”，这个名称表示与系统内核相对应的外壳层应用。Linux 系统下有多种 shell，如常见的 Bourne Again shell（简称 Bash）、C shell（简称 Csh）、Korn shell 等。主要的 Linux shell 如图 2-1 所示。

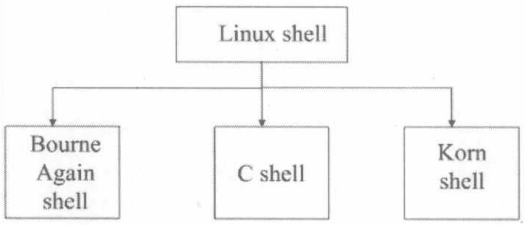


图 2-1　Linux shell 分类

Bash 是 Linux 系统下标准的 shell。当系统创建一个新的用户时，如果不特别指定，其采用的 shell 默认就是 Bash。Bash 是在 1979 年由 Stephen Bourne 开发完成的。它具有功能简洁、运行速度快的特点。另外 Bash 具有 DOS 操作系统下 doskey 的功能，用户以前输入过的命令可以通过上下箭头快速查阅。这也是 Bash 吸引用户的另外一个重要原因。

Csh 是于 20 世纪 70 年代末发展起来的，它由加州大学伯克利分校的 Bill Joy 设计，提供了部分 Bash 所不具备的功能。Csh 是 BSD 版本的 UNIX 系统中标准的 shell。在进行 shell 程序设计时，Csh 支持 C 语言类似的语法，这也是其所以被称为 Csh 的原因。Csh 的缺点是运行速度较 Bash 慢。

Korn shell（简称 Ksh）产生于 20 世纪 80 年代中期，它实际上是 Bash 的一个超集。Ksh 集成了 Bash 和 Csh 的优点，同时在运行速度上有所提升。由于这些特点，Ksh 受到 Linux 用户的广泛欢迎。

shell 是在增加用户时指定的，增加用户可以通过执行命令 Useradd 进行，也可以通过操作系统提供的实用工具进行。Useradd 的语法如下所示。

```
Useradd [参数]  主要的参数描述如下
-d homedir      指定新建用户的主目录，当用户登录系统时，默认就位于主目录下
-u uid          指定新建用户的用户 ID
-g gid          指定新建用户的组 ID
-s shell        指定新建用户的 shell，默认是采用 Bash
```

用户创建完成后，如果需要修改用户默认的 shell，可以通过执行命令 usermod 完成，也可以通过直接修改文件/etc/passwd 达到同样的目的。/etc/passwd 是 Linux 系统中保存用户配置信息的文件，其格式如下所示。

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/bin/bash
daemon:x:2:2:Daemon:/sbin:/bin/bash
…
```

每当在系统中增加一个用户时，该文件就会增加一行。文件中每一行描述了一个用户的配置信息。每行数据由若干要素组成，各个要素之间用冒号分隔。每行数据的格式如下所示。

```
用户名称:加密后的口令:用户 ID:组 ID:账号的说明或者描述:主目录:登录 shell
```

提示：在 Linux 系统中，Bash 与 UNIX 系统中的 Bsh 是类似的。Bash 是 Bourne Again Shell 的缩写，是在 Bourne Shell 基础上经过扩充发展而来的。该 shell 是由 Steven Bourne 编写出来的，为了纪念他，就将这种 shell 称作 Bourne Shell。

2.1.2　如何进入 shell

要进入 shell 环境，需要执行登录（login）操作。所谓登录，就是进行用户身份校验的过程。在 Linux 登录提示符下输入正确的用户名和口令，即可登录进入系统。登录进入系统后，系统将根据 shell 的类型执行一些初始化工作，如对于 Bash 来说，将执行.profile 文件，然后系统将给出 shell 命令提示符。提示符的形式可以自己定义。通常情况下，普通用户的 shell 命令提示符是$，超级用户的命令提示符是#。登录及退出的过程如下所示。

```
Login:<输入用户名称>
Passworg:<输入用户密码>
$<输入命令>
…
$ctrl+d（exit）
Login:
```

上述登录过程是基于字符界面的登录过程。除了这一登录方式外，Linux 还支持图形界面的登录方式。这与读者经常使用的 Windows 操作系统有些类似，如图 2-2 所示。通过图形界面成功登录系统后，展示给用户的将是图形操作界面。关于这一图形操作界面，将在后续的内容中向读者介绍。如图 2-2 所示是 SUSE Linux 的图形登录界面，可以从左侧的用户列表选择要登录的用户，也可以直接输入要登录的用户名和密码，单击“登录”按钮即可进入系统。

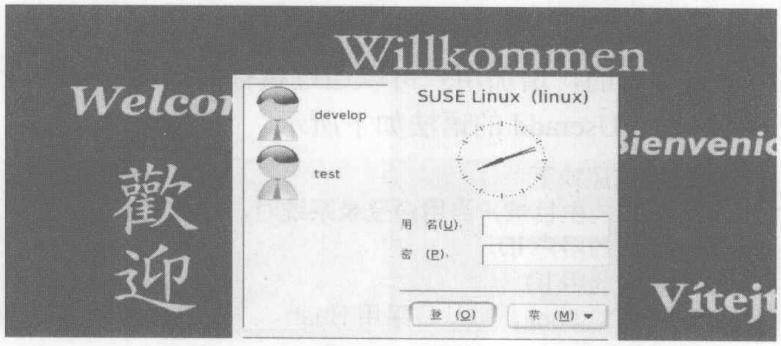


图 2-2　图形方式登录 Linux

系统登录过程中，采用字符界面还是图形界面是由系统配置文件/etc/inittab 中的配置项决定的。如果文件中的 initdefault 配置项的值为 3，则系统登录时将以字符操作界面进行。如果该配置项的值为 5，则展示给用户的界面为图形操作环境。如图 2-3 所示展示了/etc/inittab 中相应配置项的信息。

```
# Copyright (c) 1996-2002 SuSE Linux AG, Nuernberg, Germany.  All rights reserved.
#
# This is the main configuration file of /sbin/init, which
# is executed by the kernel on startup. It describes what
# scripts are used for the different run-levels.
.
.
.
# The default runlevel is defined here
id:3:initdefault:
```

图 2-3　默认登录方式

2.1.3　如何使用 shell

无论何种 shell，其功能最终都在于解释用户输入的命令并执行。在接收用户输入的命令后，shell 将根据环境变量 PATH 指定的路径，搜寻相应的命令。如果未找到，将给出错误提示信息。如果找到匹配的命令，将执行该命令。在大多数情况下，系统已将重要的系统目录设置到 PATH 环境变量中，通过执行命令 echo $PATH 可以查看当前的 PATH 环境变量信息。其输出如下所示。

```
/sbin:/usr/sbin:/usr/local/sbin:/root/bin:/usr/local/bin:/usr/bin:/usr/X11R6/bin:/bin:/usr/games:/opt/gnome/bin:/opt/kde3/bin:/usr/lib/java/jre/bin:.
```

在上述输出中，多个目录间以“:”进行分隔。在 shell 搜索用户输入的命令时，将按照 PATH 环境变量给出的路径依次进行搜索。如果搜索到匹配的命令，则执行相应程序；如果全部目录都搜索结束也未找到匹配的命令，则提示错误信息。

提示：在 PATH 环境变量的使用过程中，需要特别注意当前目录的使用。从 PATH 内容可以看到，最后一项“.”就是指当前目录。按照上述 PATH 环境变量的设置，在/sbin、/usr/sbin 等目录搜索结束后，shell 将在当前目录下寻找匹配的命令。为防止当前目录下的某些恶意程序进行破坏，注意不要在 PATH 的配置中将当前目录“.”放到重要系统目录前，或者干脆不要将当前目录设置到 PATH 中。

shell 给用户提供了一个较为人性化的执行环境，它允许用户自定义 shell 的使用方式和习惯。这些可以自定义的行为包括设置终端的种类、设置环境变量、设置新建文件的权限掩码、设置自定义的 shell 命令提示符等。对于 Bash 来说，这些设置在初始化脚本文件.profile 中进行定义。对于 Csh 来说，这些设置在.cshrc 中进行定义。

如果是对于本机的所有用户的设置，也可以统一在/etc/profile 中设置。在用户登录的过程中，不论是何种 shell 都将首先执行该文件。例如，Bash 的用户登录过程中执行的初始化文件依次如下所示。

```
/etc/profile
~/.bash_profile  ~/.bash_login  ~/.profile 三个其中之一。其中，~在 shell 环境中表示用户的主目录。
```

提示：在 Bash 中，有一个重要的隐藏文件.bash_history。该文件中保存了当前用户输入过的全部命令。可以通过在 shell 中按上、下方向键，调出曾经使用过的命令来执行。通过这一方式，可以大大提高用户使用 shell 的效率。

除了解释 shell 命令外，shell 还支持脚本编程。可以将多个 shell 命令以一定的格式组织成 shell 程序，程序中可以实现复杂的逻辑控制功能，如条件判断、循环操作等。关于 shell 脚本编程的内容将在下一节详细介绍。

2.2　shell 编程基础

shell 提供了脚本编程的功能。这一功能非常强大，甚至可以代替高级语言编程实现相应功能。在 DOS 操作系统中，如果需要执行一些重复性的命令，可以通过编写一个批处理文件（扩展名.bat），将这些命令写到批处理文件中。在需要执行时，直接执行这个批处理文件就可以了。与 DOS 相同，Linux 也提供了批处理的命令。不过 Linux 的批处理命令比 DOS 更加强大，甚至可以与通常的高级语言一比高低。这就是本章将要介绍的 shell 编程。

2.2.1　创建和运行 shell 脚本程序

要创建一个 shell 脚本程序的过程非常简单。首先新建一个文本文件，然后在文本文件中按照 shell 编程规则输入 shell 命令即可。通常情况下，在 Linux 中建立一个文本文件会用到 vi 编辑器。关于该编辑器的详细用法，将在后续章节中进行介绍。

要运行一个 shell 程序，主要有两种方式：一是赋予 shell 程序的执行权限，如通过 chmod 命令给脚本文件增加执行权，然后在命令行直接运行即可；二是通过执行另外的 shell 加载脚本，如通过命令“sh 脚本文件名称”的方式运行。

提示：对于使用 shell 编程的程序员来说，使用普通用户而不是超级用户是一个很好的安全习惯。在普通用户身份下，大多涉及安全的命令是限制执行的。而在超级用户身份下，如果执行涉及系统安全的命令，将没有任何警告直接执行，这很容易给系统造成严重的破坏甚至崩溃。

【范例 2-1】创建一个 shell 脚本 echodate，输出当前主机的系统日期。其实现过程如示例代码 2-1 所示。

示例代码 2-1

```
1    vi echodate                              #用 vi 编辑文件 echodate
2    #%Y 是 date 命令的格式化输出，表示 4 位的年份，
3    #%m 表示 2 位的月份
4    #%d 表示 2 位的日期
5    echo 'date +%Y%m%d'                      #文件的内容是调用 echo 命令显示执行 date 命令的结果
```

【运行结果】通过执行命令 chmod 755 echodate 赋予 echodate 执行权限，在 shell 下运行./echodate，其结果如下所示。

```
1    20081001
```

提示：也可以在 shell 命令提示符下运行 sh ./echodate 直接运行该脚本，而不对脚本赋予执行权。

2.2.2 shell 环境变量

所谓环境变量，就是包含了系统和当前登录用户的一些环境信息（或者称为配置信息）的一些字符串。也可以看作是一些在 shell 环境中定义的全局变量，这些变量都指定了相应的值。这些环境变量可以在 shell 程序中引用（在后续章节中可以看到，也可以在 C 语言程序中引用环境变量）。

操作系统本身会包含一部分环境变量，同时每个用户也可以设置自己不同的环境变量。通常情况下，环境就是用于保存一些用户的配置信息。较为常见的环境变量有 TERM（用于保存终端类型，TERM=Vt220）、PATH（用于保存 shell 命令的搜索路径）等。

shell 的环境变量可以从系统进行继承，也可以由用户自行定义。从系统继承的环境变量是在文件/etc/profile 中定义的，用户自行定义的环境变量大多在文件.profile 中（根据 shell 的不同，该文件名称也不相同）。环境变量为用户提供了保存配置信息很好的方法。对于一些需要配置的信息，用户可以定义成环境变量，然后在程序中进行引用。如果该配置信息的值需要改变，只需要修改环境变量的值即可。用户登录系统时，环境变量的继承情况如图 2-4 所示。

可以通过如下方式设置环境变量。

```
export 环境变量的名称=环境变量的值
```

要查看系统中已定义的环境变量，可在 shell 下执行 env 命令。如果要单独查看某个环境变量的值，则可以通过如下命令进行。

```
echo $环境变量的名称
```

环境变量的命名与使用必须遵守一定的规则。在定义环境变量过程中，需要注意以下几点。

- 变量名与变量的值必须以等号“=”来连接，等号两边不能有空格符。
- 变量名称只能是英文字母与数字，但是数字不能是开头字符，通常使用大写字母定义环境变量。
- 变量的引用需要通过$+环境变量名称的方式进行。需要注意的是，在使用引号时，如果需要在引号中间引用环境变量，则应该使用双引号。
- 如果除当前 shell 程序中使用该环境变量外，其他程序也需要引用该环境变量，则应该使用 export 将该环境变量输出。

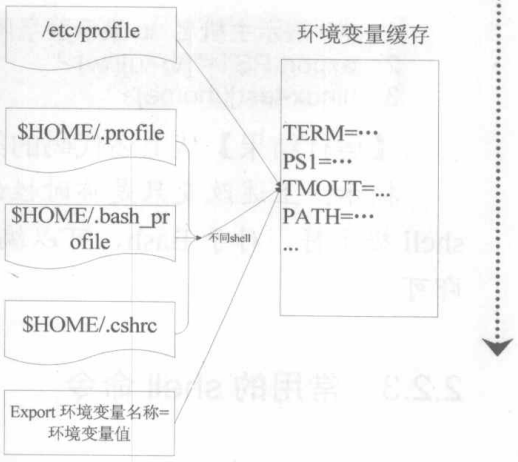


图 2-4　环境变量的继承

常见的由系统提供的环境变量如下。

- PATH：shell 命令搜索路径。如果存在多个路径，则路径间以冒号为分隔符。通常情况下，应该把当前目录所在的路径“.”放在第一项，这可以有效避免部分木马病毒的攻击。常见的 PATH 定义如 PATH=.:/usr/bin:/bin:/usr/sbin。
- HOME：用户的主目录的路径名。建立用户时，为每个用户指定了主目录。当用户登录进入系统时，默认状态即进入该主目录。
- TERM：终端类型，终端类型定义了用户所使用的终端的种类，常见的如 vt100、vt220、ansi 等。
- TMOUT：shell 自动退出的时间，单位为秒。若设为 0 则禁止 shell 自动退出。
- PS1：主命令提示符。用户可根据需要自行定义。
- PS2：二级命令提示符。命令执行过程中要求输入数据时用。
- LD_LIBRARY_PATH：寻找动态链接库的路径。如果存在多个路径，路径间用冒号分隔。

默认情况下，Linux 的 shell 命令提示符是“用户名@主机名:当前工作目录”。用户可以根据自己的需要进行定制。定制的方法是通过修改用户的 shell 环境变量 PS1。在定制过程中，shell 支持如下格式的特殊字符。

```
\d 显示当前日期
\h 显示主机名
\t 显示当前时间
\u 显示当前用户的用户名
\w 显示当前工作目录的路径
```

【范例 2-2】通过设置环境变量 PS1 将命令提示符修改为“[主机名-用户名][工作目录]>”的形式。其实现过程如示例代码 2-2 所示。

示例代码 2-2

```
1  #\h 表示主机名 \u 表示登录用户名 \w 表示当前工作目录
2  export PS1="[\h-\u][\w]>"
3  [linux-test][/home]>
```

【运行结果】从上述代码的第 2 行可以看出，命令提示符已经发生了改变。

提示：上述改变只是临时性的，在 shell 退出后将失效。如果需要永久性改变该用户的 shell 提示符，对于 Bash，可以编辑用户主目录下的.profile 文件，在其中加上 export PS1=... 即可。

2.2.3 常用的 shell 命令

shell 命令既可以在命令提示符下直接使用，也可以在 shell 编程的脚本中使用。本小节将详细讲解其常见的应用。

1．更改账号密码

通过执行 passwd 命令可以修改用户的登录口令。定期修改用户口令是一个良好的安全习惯。超级用户 root 可以修改任何用户的口令而无需输入旧口令，普通用户修改口令则需要输入旧口令进行验证。通过执行 passwd 命令修改口令的语法如下所示。

```
$passwd
Old Password：<输入旧密码>
New Password：<输入新密码>
Retype New Password：<再输入一次新密码>
```

2．帮助

在 Linux 系统下，可以通过执行 man 命令查找帮助。man 命令类似 Windows 系统下的 F1 帮助。通过 man 命令，用户可以更好地掌握命令的用法甚至函数的调用方法。这一命令在 Linux 中非常实用，读者应该好好加以利用。man 命令的语法如下所示。

```
$man <命令名称>
```

3．列表文件或目录

执行命令 ls 可以查看文件或者目录的信息。这一命令是 Linux 系统中最为经常使用的一个命令。ls 命令有许多选项，通过设置不同的选项，可以灵活输出不同的文件信息。ls 命令的语法如下所示。

```
$ls <文件或者目录名称>
```

4．改变工作目录

通过执行 cd 命令可以改变当前的工作目录。在使用 shell 过程中，当前所在的目录就是工作目录。用户刚登录时，其工作目录通常是其主（HOME）目录。如果要修改工作目录，可以通过执行命令 cd 进行。cd 命令的语法如下所示。

```
$cd <目录绝对路径或者相对路径>
```

5．复制文件

通过执行 cp 命令可以复制文件，带参数-r 可以将整个目录一起进行复制。cp 命令的语法如下所示。

```
$cp <源路径名称> <目标路径名称>
```

6．删除文件

通过 rm 执行命令可以删除文件，如果指定-r 选项，则可以删除一个目录。另外一个经常使用的选项是-f，使用-f 选项可以强制删除指定的文件，即使该文件正在被使用。rm 命令的语法如下所示。

```
$rm <文件名称>
```

7．修改文件访问权限

通过执行 chmod 命令可以修改文件或者目录的访问权限。通过使用-r 选项，可以连同所有子目录下的文件权限一并修改。关于文件权限的详细信息，请参见第 9 章中相关内容。chmod 命令的语法如下所示。

```
$chmod <权限信息> <文件名称>
```

上面只是列举了常用的部分命令。除此之外，还有许多其他 shell 命令，在此不再一一列举。

提示：在 Linux 系统中，执行 shell 命令的一个好习惯是使用绝对路径名。如用户通常输入的 ls 命令，安全的方法是/bin/ls。要执行当前目录下的程序，则输入“./程序名”执行。如果感觉这样比较麻烦，可以使用 shell 中的别名功能。将/bin/ls 用“alias myls='/bin/ls'”定义为别名 myls，在以后的使用中，直接执行 myls 命令就可以了。

2.2.4　管道与重定向的使用

Linux 系统支持在 shell 中使用管道。所谓管道，就是将一个命令的输出作为另一个命令的输入。例如，要统计当前目录下文件的数量可以用下述命令实现。

```
ls|wc –l
```

其中的“|”就是管道符号。通过管道，将 ls 命令的输出作为 wc 命令的输入。因此上述命令的输出就是 wc 统计的 ls 命令输出的行数，即文件的数量。除了支持管道操作外，Linux 还支持输入/输出的重定向。默认情况下，shell 的输入是键盘，而输出是屏幕。通过重定向技术，可以将输入/输出重新定向到指定的目标，如某个文件或打印机等，如图 2-5 所示。

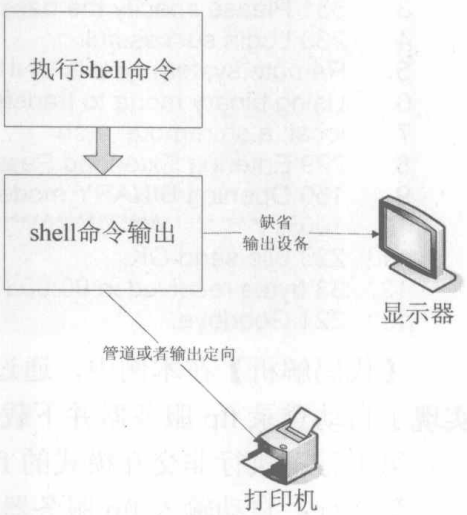


图 2-5　管道和重定向

Linux 的输出重定向有以下两个操作符。

- >：重定向，如果是重定向至文件，则目标文件被覆盖。

- >>：重定向，如果重定向至文件，则将在目标文件的尾部追加。

例如，要将一个文件输出到打印机，除了执行 lp 命令外，还可以通过输出重定向的方式完成，其过程如下所示。

```
cat 文件名称>/dev/lp0
```

输入操作的重定向也包含两个操作符。

- <：输入重定向。
- <<：也称为 here 文档操作符。它通知 shell 当前运行的命令的输入来自命令行。<< 操作符后面的输入用一个分隔符开始，用同样的另外一个分隔符结束。两个分隔符间的内容就是要执行的命令的输入。例如，要统计一段文字单词的数量，如下所示。

```
wc<<delim
This is a test program!
This is the second line.
delim
```

在使用<<操作符时，要注意最后的分隔符要以一个新行开始。

【范例 2-3】创建一个 shell 脚本 autoftp，实现自动登录 ftp 服务器并下载某文件。其实现过程如示例代码 2-3 所示。

示例代码 2-3

```
1    ftp -n 192.168.1.10<<!                    #使用非交互模式的 ftp,用!作分隔符
2    user abc password                         #用户名和口令
3    get $1                                    #以命令行参数 1 作为要获取的文件名
4    bye                                       #退出 ftp
5    !                                         #!分隔符，命令结束
```

【运行结果】通过执行命令 chmod 755 autoftp 赋予 autoftp 执行权限，执行 autoftp a.sh 的结果如下所示。

```
1    Connected to 192.168.1.10.
2    220 (vsFTPd 1.2.1)
3    331 Please specify the password.
4    230 Login successful.
5    Remote system type is UNIX.
6    Using binary mode to transfer files.
7    local: a.sh remote: a.sh
8    229 Entering Extended Passive Mode (|||32230|)
9    150 Opening BINARY mode data connection for a.sh (33 bytes).
10   100% |**************************************|    33        10.14 KB/s    00:00 ETA
11   226 File send OK.
12   33 bytes received in 00:00 (7.90 KB/s)
13   221 Goodbye.
```

【代码解析】在本例中，通过执行非交互模式（-n 选项）的 ftp，使用输入重定向符<<，实现了自动登录 ftp 服务器并下载指定文件的功能。源码的各行解释如下。

第 1 行：执行非交互模式的 ftp，通过<<指定 ftp 的输入来自于后续的文本。

第 2 行：自动输入 ftp 服务器的用户名和密码。

第 3 行：调用 ftp 的 get 命令从服务器获取指定的文件。

第 4 行：关于与 ftp 服务器的连接。

第 5 行：重定向的第二个分隔符，注意该符号应该从一个新行开始。

2.2.5　shell 变量的使用

与任何编程语言相同，shell 也支持变量的使用。shell 环境中的变量有多种类型，一是环境变量，可以在 shell 脚本中直接用“$环境变量名称”的形式引用；二是用户自定义变量，将在后面进行详细介绍；三是系统全局变量，这些变量代表了特定的含义，不能为用户所修改。主要的全局变量包括以下几个。

```
$#：执行 shell 脚本时的命令行参数个数，不包括脚本本身
$?：执行上一条 shell 命令的返回值
$0：shell 脚本程序自身的名称
```

shell 是一种解释执行的脚本语言，所以用户自定义变量的使用不需要事先定义。通常情况下，使用变量时直接对变量赋初值，具体的方法如下。

```
变量名称=变量初始值（注意等号两边不能够有空格）
```

在用户定义自己的变量时，要遵循以下命名规则。

- 首个字符必须为字母（a～z、A～Z）。
- 中间不能有空格，可以使用下画线（_）。
- 不能使用标点符号。
- 不能使用 shell 里的关键字（可执行 help 命令查看保留关键字）。

要在脚本程序中使用变量，则通过“$变量名称”的方式进行。即在变量的前面加$符号。另外，在 shell 编程中存在一种称为位置变量的变量。这种变量类似于 C 语言编程中的命令行参数，它代表了执行 shell 程序时的命令行参数。要在 shell 脚本中引用这些位置变量，通过$0,$1,$2,...的方式进行。其中，$0 代表 shell 命令程序本身，而$1 代表 shell 命令行的第一个参数，$2 代表第二个，以此类推。

【范例 2-4】创建一个 shell 脚本 sum，计算并输出两个输入数据的和。其实现过程如示例代码 2-4 所示。

示例代码 2-4

```
1    vi sum                                  #调用 vi 编辑 sum 文件
2    #调用 expr 计算两个命令行参数的和
3    sum=`expr $1 + $2`                      #调用 expr 计算命令行参数 1 和命令行参数 2 的和
4    #输出求和的结果
5    echo $1+$2=$sum                         #调用 echo 显示结果
```

【运行结果】在 shell 下运行./sum 1 2，其结果如下所示。

```
6    1+2=3
```

【代码解析】在本例中，首先调用 expr 进行计算，将计算结果存放到变量 sum 中，最后将求和结果输出。

提示：在使用 expr 进行计算时，注意运算符两边应该加空格。

2.2.6 shell 运算符的应用

当进行 shell 编程时，如果需要进行科学计算，则需要用到 expr 命令。通过执行 expr 命令可以进行下述计算。

```
+：加法
-：减法
*：乘法
/：除法
%：求余
:：字符串比较。如果两个字符串相等，则返回字符串的长度；否则返回 0
>，>=，<，<=，=，!=：比较运算，包括大于、大于等于、小于、小于等于、等于、不等于
&，|：与和或运算
```

除去 expr 命令外，还有一个 test 命令用于进行逻辑测试。可以用 test 命令进行测试的情况包括以下 4 种。

1．整数测试运算

```
test int1 -eq int2：判断两个整数是否相等
test int1 -ne int2：判断两个整数是否不相等
test int1 -gt int2：判断整数 1 是否大于整数 2
test int1 -ge int2：判断整数 1 是否大于等于整数 2
test int1 -lt int2：判断整数 1 是否小于整数 2
test int1 -le int2：判断整数 1 是否小于等于整数 2
```

2．字符串测试运算

```
test –z string：判断字符串的长度是否为 0，即判断串是否为空，为空返回真，非空返回假
test –n string：判断字符串的长度是否不为 0，即判断串是否非空，为空返回假，非空返回真
test str1=str2：判断两个字符串是否相等
test str1! = str2：判断两个字符串是否不等
```

3．文件测试运算

```
test -r filename：判断用户对文件 filename 是否有读权限
test -w filename：判断用户对文件 filename 是否有写权限
test -x filename：判断用户对文件 filename 是否有可执行权限
test -f filename：判断文件 filename 是否为普通文件
test -d filename：判断文件 filename 是否为目录
test -c filename：判断文件 filename 是否为字符设备文件
test -b filename：判断文件 filename 是否为块设备文件
test -s filename：判断文件 filename 是否大小不为零
test -t fnum：判断与文件描述符 fnumb（默认值为 1）相关的设备是否是一个终端设备
```

4．逻辑运算

```
test 表达式 1 –a 表达式 2：与（and）逻辑判断。如果两个表达式同时为真则返回真，否则返回假
test 表达式 1 –o 表达式 2：或（or）逻辑判断。只要两个表达式有一个为真则返回真，否则返回假
```

【范例 2-5】创建一个 shell 脚本 diff，比较从命令行输入的两个整数的值是否相等。其实现过程如示例代码 2-5 所示。

示例代码 2-5

```
1    #判断两个命令行参数是否相等
2    if test $1 = $2                    #调用 if 判断命令行参数 1 和命令行参数 2 是否相等
3    #相等
4    then
5        echo $1=$2                     #输出两个参数相等的信息
6    #不等
7    else
8        echo $1!=$2                    #输入两个参数不等的信息
9    fi                                 #if 结束
```

【运行结果】在 shell 下运行./diff 1 2，其结果如下所示。

```
10   1!=2
```

【代码解析】在本例中，调用 test 命令判断两个输入的整数是否相等，并根据判断结果输出相应信息。源码的各行解释如下。

第 2～3 行：输出相等的信息。

第 5～6 行：输出不相等的信息。

2.2.7　在 shell 脚本中进行条件控制

在进行 shell 设计过程中，支持各种条件判断和循环控制。其中用于条件判断的有 if 表达式和 case 表达式。进行循环控制的有 for 循环、while 循环和 until 循环。使用 if 表达式进行条件判断的语法如下所示。

```
if [条件表达式 1]
then
    if [条件表达式 2]
    then
    …
    else
    …
    fi
    命令串;
else
    命令串;
fi
```

上述 if 语法中，中括号“[ ]”中的内容用于进行条件测试。使用“[ ]”进行条件测试时，要注意空格的使用。在 if 与“[”间要有空格，在“[”与后面的条件测试表达式间要有空格，在“]”与前面的条件测试条件式间也要有空格。这一点读者要特别注意，经常在 shell 编程中因少用空格而导致错误。

使用 case 表达式进行条件判断的语法如下所示。

```
case string in
str1)
命令串 1;;
str2)
命令串 2;;
*)
```

```
默认处理命令串;
esac
```

【范例 2-6】创建一个 shell 脚本 weekday，输入数字，输出该数字对应的星期几。其实现过程如示例代码 2-6 所示。

示例代码 2-6

```
1   #判断命令行参数个数是否小于 1
2   if [ $# -lt 1 ]                    #判断命令行参数个数是否小于 1
3   #无命令行参数，提示用法后退出
4   then
5       echo Usage:$0  数字            #提示用法
6       exit
7   fi
8   #用 case 表达式输入的数字
9   case $1 in
10  #星期一
11  1)
12      echo monday;;                  #显示 monday
13  #星期二
14  2)
15      echo tuesday;;                 #显示 tuesday
16  #星期三
17  3)
18      echo wednesday;;               #显示 wednesday
19  #星期四
20  4)
21      echo thursday;;                #显示 thursday
22  #星期五
23  5)
24      echo friday;;                  #显示 friday
25  #星期六
26  6)
27      echo saturday;;                #显示 saturday
28  #星期日
29  7)
30      echo sunday;;                  #显示 sunday
31  #其他任意字符
32  *)
33      echo invalidate;;              #输出无效信息量
34  esac                               #case 结束
```

【运行结果】在 shell 下运行./weekday 2，其结果如下所示。

```
1   tuesday
```

【代码解析】在本例中，调用 case 进行判断输入的星期数，并输出英文的星期几信息。源程序中各行的详细解释如下。

第 1～7 行：判断执行脚本时的命令行参数个数，如果没有带参数，则提示用法后退出。

第 9～34 行：通过 case 表达式判断输入的数字，并输出对应的星期几。

提示：在 shell 中，注释是通过首字符输入#进行的，这与 C 语言不同，需要读者注意。

2.2.8　在 shell 脚本中使用 for 循环

上一节介绍过 shell 可以使用 3 种循环，分别是 for 循环、while 循环和 until 循环。这三种循环方式均可用于需要循环执行某段代码的场合。for 循环有两种不同的使用方式，其中第一种形式如下所示。

```
for var in list
do
命令串
done
```

参数说明如下。

- list：列表，可以是由空格分隔的变量（$a $b）或者是值（1 2 3 等）。对于 list 中的每一项，都将循环一次。
- var：每次循环的值。对于 list 中的每一项都要进行一次循环，而每次循环时，就取出 list 中的第几项放在 var 中，可以在命令串中通过$var 的方式进行引用。

for 循环的第二种方式如下所示。

```
for var
do
命令串
done
```

可以看到，在第二种 for 循环中，只是缺少了 list 项。在这种情况下，for 循环省略了 list，list 值由当前脚本程序的命令行参数代替。也就是说，在这种情况下，每循环一次，var 中存储的都是一个命令行参数。

【范例 2-7】创建一个 shell 脚本 computesum，从命令行输入多个以空格分隔的数字，输出全部数字的和。其实现过程如示例代码 2-7 所示。

示例代码 2-7

```
#定义求和结果存储的变量，并赋初值为 0
sum=0                                                    #初始化变量 sum
#调用 for 循环的第二种格式，以命令行参数为 for 循环值列表
for ccc                                                  #for 循环开始
do                                                       #do 关键字
#判断是否为数字
    if ['expr $ccc \>= 0' = 1 ]                          #判断是否为数字
    then                                                 # then 关键字
#输出加数
        echo -e "$ccc+\c"                                #输出加数
#累加命令行参数至 sum 变量
        sum='expr $sum + $ccc'                           #调用 expr 求和
    fi                                                   #if 结束
done                                                     #do 结束
#输出求和的结果
echo -e "\b=$sum"                                        #输出求和结果
```

【运行结果】在 shell 下运行./computesum 2 3 4，其结果如下所示。

```
17  2+3+4=9
```

【代码解析】在本例中，调用 for 循环依次读取命令行参数，并判断是否是数字。如果是数字则求和，最后输出求和结果。源程序中各行的详细解释如下。

- 第 4～14 行：利用 for 循环，获取全部的命令行参数并求和。
- 第 10 行：显示加数。注意：最后的“\c”表示输出加数后不按<Enter>键。
- 第 12 行：调用 expr 累加命令行参数至 sum 变量。注意：在“+”运算符的两侧要有空格。
- 第 16 行：显示求和的结果。注意：其中的“\b”是退格的意思。由于显示的加数最后多余一个“+”号，按退格键将该“+”号删除。

2.2.9 在 shell 脚本中使用 while 循环

在上一节介绍的 for 循环中，使用了循环变量判断是否应该退出循环。当循环变量的值超出预定范围后，循环结束。在某些情况下，需要依据某个条件进行判断，如果条件为真则继续循环，否则结束循环。这种情况下，就需要使用 while 循环。while 循环是当某个判定条件的值为真时，一直循环直到该判定条件为假才退出循环。while 的语法如下所示。

```
while 条件表达式
do
命令串;
done
```

【范例 2-8】创建一个 shell 脚本 testcmdline，从命令行输入多个以空格分隔的参数，使用 while 枚举全部参数。其实现过程如示例代码 2-8 所示。

示例代码 2-8

```
1   #定义变量，用于存储当前命令行参数的顺序号
2   n=1                                          #初始化变量 n
3   #while 循环，测试$*是否为空。
4   #$*包含了全部的命令行参数，在循环体中调用 shift 向参数向左移除，导致$*最终变为空而循环
    结束
5   while [ -n "$*" ]                           #while 循环开始
6   do                                           #do 关键字
7   #显示每个参数的信息
8       echo 第$n 个参数=$1                      #显示第几个参数的提示和参数值
9   #顺序号加 1
10      n='expr $n + 1'                          #调用 expr 计算下一个参数的序号
11  #命令行参数左移一个
12      shift                                    #调用 shift 左移命令行参数
13  done                                         #do 结束
```

【运行结果】在 shell 下运行./ testcmdline a b c，其结果如下所示。

```
1   第 1 个参数=a
2   第 2 个参数=b
3   第 3 个参数=c
```

【代码解析】在本例中，调用 while 循环依次输出每个命令行参数。注意：在循环体中使用 shift 将命令行参数左移。这样，每循环一次，命令行参数将从左侧被移走一个。使原来处于第 2 位的命令行参数变成为第 1 位。最终，$*所代表的全部命令行参数将为空，循环结束。

2.2.10　在 shell 脚本中使用 until 循环

until 循环与 while 循环实现的功能基本相同，不过 until 是判定条件表达式为假时才继续循环。until 的语法如下所示。

```
until 条件表达式
do
命令串;
done
```

2.2.11　在 shell 脚本中使用函数

对于在脚本中重复使用的功能模块，可以封装成为函数。在 shell 脚本中使用函数，可以使程序代码更为简洁，程序的可读性更好。shell 脚本中的函数定义可以使用如下两种方式。

```
函数名()
{
...
}
```

或者

```
function 函数名()
{
...
}
```

在调用函数时，可以向函数传递任意个数的参数。在函数中可以通过位置参数$1、$2 的方式进行引用。函数也可以有返回值，可以通过查询全局变量$?的值获取函数的返回值。

【范例 2-9】创建一个 shell 脚本 fact.sh，从命令行输入数字，计算该数字的阶乘并输出。其实现过程如示例代码 2-9 所示。

示例代码 2-9

```
1   #计算阶乘的函数，函数中使用了递归。函数的两个参数分别为$1=本次阶乘因子、$2=上次阶乘结果。
2   function fact()                                    #定义函数 fact，用于计算阶乘
3   {
4   #如果阶乘因子等于 0，表明阶乘计算完成，输出结果返回
5       if [ $1 -eq 0 ]                                #调用 if 判断本次阶乘的值是否为 0
6       then
7           echo $2                                    #是 0，则输出阶乘的结果
8           return                                     #返回
9       fi                                             #if 结束
10  #将当前的阶乘因子与上一次递归计算所得的阶乘结果相乘，并将结果存放于变量 t 中
```

```
11        t='expr $2 \* $1'                                    #调用 expr 计算 两个数的积
12    #递归调用
13        fact 'expr $1 - 1' $t                                #递归调用当前函数
14    }
15    #调用函数 fact 计算阶乘，第一个参数为从命令行输入的要求阶乘的变量的值
16    #第二个参数是初始的递归结果，设为 1
17    fact $1 1                                                #调用函数 fact 计算命令行参数 1 的阶乘
```

【运行结果】在 shell 下运行./fact.sh 5，其结果如下所示。

```
1    120
```

【代码解析】在本例中，首先定义了递归函数 fact，用于计算一个数字的阶乘。该函数有两个参数，第一个参数是本次阶乘的因子，第二个参数是上次阶乘的结果。从输出可以看到，所求 5 的阶乘为 120。

2.3 综合实例

在前面小节中，已经详细讲解了关于 shell 编程的各种基础内容。在本小节中，将通过一个典型的综合案例来介绍如何使用 shell 编程。

2.3.1 实例需求

用户在进行 Linux 系统管理的过程中，经常需要用到查看进程的信息、用户的信息等常用的功能。本例针对这一需求，使用 shell 编程实现了基本的系统管理功能。通过本程序，可以按照要求实现查看进程信息、查看当前登录系统的用户等功能。程序主界面颜色可以更改，也可以显示帮助信息。

2.3.2 系统设计

针对本例的情况，首先考虑到界面的美观性，将主界面设计为菜单形式。各系统管理的功能作为其中的一个菜单项。当用户选择某项功能后，则执行相应的代码段完成功能并输出。输出结束后再返回主菜单，供用户继续选择。流程图如图 2-6 所示。

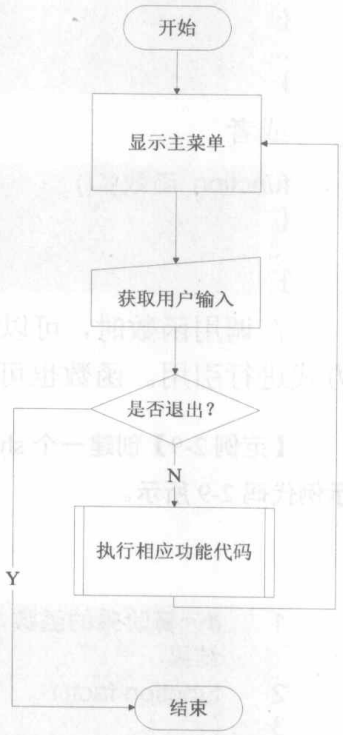


图 2-6 流程图

2.3.3 程序代码

按照上述设计思想，结合本章中介绍的 shell 编程方法予以实现。程序可以划分为几个模块：界面修改模块、主界面模块、查看进程信息模块、查看用户信息模块和帮助模块。各模

块都封装成函数的形式予以实现。

1．界面修改模块

通过函数 colour 实现，函数中根据用户的不同选择，调用 echo 函数向终端发送控制命令实现颜色的变更，其实现代码如下所示。

```
1   colour()                          #定义函数 colour,根据输入颜色信息修改界面颜色
2   {
3   case $1 in                        #调用 case 判断函数参数 1 的值
4   black_green)                      #黑底绿字
5   echo -e "\033[40;32m"             #调用 echo，通过向显示器输出控制字符达到特殊效果
6   ;;                                #case 节结束符
7   black_yellow)                     #黑底黄字
8   echo -e "\033[40;33m"             #调用 echo，通过向显示器输出控制字符达到特殊效果
9   ;;
10  black_white)                      #白底黑字
11  echo -e "\033[40;37m"
12  ;;
13  black_white)                      #黑底白字
14  echo -e "\033[40;36m"
15  ;;
16  black_blue)                       #黑底蓝字
17  echo -e "\033[40;34m"
18  ;;
19  esac
20  }
```

2．主界面模块

主界面模块负责主菜单的显示，并接受用户的输入。根据用户输入的功能选项，调用不同的模块执行相应的功能。其实现代码如下所示。

```
21  echo -e "\033[2J"                                    #清屏
22  trap "" 1 2 3                                       #设置信号处理
23  mday='date +%d/%m/%y'                               #日期信息
24  mhost='hostname'                                    #机器名信息
25  mwho='whoami'                                       #当前用户信息
26  while :                                             #循环显示主菜单
27  do
28  cat <<mmenu                                         #显示主菜单，mmenu 作为 cat 命令的结束符
29          ------------------------------------------------------
30            $mwho              $mhost              $mday
31          ------------------------------------------------------
32                      1: 改变字体颜色
33                      2: 查看进程信息
34                      3: 查看用户信息
35                      h: 帮助
36                      Q: 退出
37          ------------------------------------------------------
38  mmenu                                               #cat 命令结束
39  echo -e -n "\tEnter Your Choice[1,2,3,h,q]:"        #提示用户输入选项
40  read Cho                                            #读取用户输入的选择项
41  case $Cho in                                        #调用 case 判断输入选项
42  1)                                                  #用户输入 1
```

```
while :                                                    #调用 while 循环，接收用户输入的颜色选择
do                                                         #while 循环关键字
cat <<kcol                                                 #显示颜色选择菜单
        ------------------------------------------------------
          $mwho              $mhost              $mday
        ------------------------------------------------------
          1:黑绿    2:黑黄   3:白黑   4:黑白   5:黑蓝   0:返回
        ------------------------------------------------------
kcol                                                       #颜色选择菜单结束
echo -e -n "\t 请输入选择的颜色[1,2,3,4,5]:"              #提示用户输入选项
read choice                                                #读取要设置的颜色信息
            if [ "$choice" = "1" ]                         #判断用户输入的颜色选择
            then
            colour black_white                             #调用 colour 函数设置颜色
            elif [ "$choice" = "2" ]
            then
            colour black_yellow                            #调用 colour 函数设置颜色
            elif [ "$choice" = "3" ]
            then
            colour black_green                             #调用 colour 函数设置颜色
            elif [ "$choice" = "4" ]
            then
            colour black_blue                              #调用 colour 函数设置颜色
            elif [ "$choice" = "5" ]
            then
            then
            colour black_cyan                              #调用 colour 函数设置颜色
            elif [ "$choice" = "0" ]
            then
            break                                          #退出当前菜单
            else
            echo -e "\033[2J";                            #清屏
            continue
            fi                                             #if 结束
clear
done                                                       #while 结束
;;
```

3．查看进程信息模块

该模块通过调用执行 ps 命令查看当前系统中的进程信息，并对进程信息进行排序后输出前 10 条进程记录。其实现代码如下所示。

```
2)                                                         #用户要查看进程信息
ps aux|sort -rn|head -10                                   #调用 ps 命令输出相应进程信息
;;
```

4．查看用户信息模块

该模块通过调用 who 命令查看当前系统中已登录的用户信息并输出。其实现代码如下所示。

```
3)                                                         #用户要查看用户信息
who                                                        #调用 who 命令输出用户信息
;;
```

5．帮助模块

该模块通过调用 who 命令查看当前系统中已登录的用户信息并输出。其实现代码如下所示。

```
86  H|h)                                                    #用户要查看帮助信息
87  cat <<mmenu                                             #调用 cat 显示帮助信息
88          选择改变颜色可以修改当前界面的字体颜色
89          选择查看进程信息可以获取当前占用资源最高的前 10 个进程
90          选择查看用户可以获取当前登录系统的用户信息
91  mmenu                                                   #显示帮助信息结束
92  ;;
93  Q|q)                                                    #用户要退出主菜单
94  exit 0                                                  #退出主程序
95  ;;
96  *)                                                      #用户输入其他的选项
97  echo -e "\033[2J";                                      #清屏
98  continue;                                               #重新让用户选择
99  ;;
100 esac                                                    #case 结束
101 echo -e -n "\t 按任意键继续"                             #执行完命令后的提示
102 read J                                                  #执行完命令后等待用户按键继续
103 clear                                                   #清屏
104 done                                                    #循环结束，退出程序
```

【运行结果】在 shell 下运行上述脚本程序，主界面如下所示。

```
1
2           develop              linux              10/10/08
3
4                    1: 改变字体颜色
5                    2: 查看进程信息
6                    3: 查看用户信息
7                    h: 帮助
8                    Q: 退出
9
10
```

其中修改字体颜色的二级菜单如下所示。

```
11          Enter Your Choice[1,2,3,h,q]:1
12
13          develop              linux              10/10/08
14
15          1:黑白    2:蓝黄    3:黑绿    4:黑蓝    5:黑灰    0:返回
16
17          请输入选择的颜色[1,2,3,4,5]:
```

查看进程信息的输出如下所示。

```
18
19          develop              linux              10/10/08
20
21                   1: 改变字体颜色
22                   2: 查看进程信息
23                   3: 查看用户信息
24                   h: 帮助
```

```
25                    Q: 退出
26          ----------------------------------------------------------
27          Enter Your Choice[1,2,3,h,q]:2
28  USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
29  root          9  0.0  0.0      0     0 ?        S    Oct09   0:00 [kswapd0]
30  root       8706  0.0  4.2  22708 10816 ?        S    01:20   0:08 /opt/kde3/bin/kdk
31  root          8  0.0  0.0      0     0 ?        S    Oct09   0:18 [pdflush]
32  root          7  0.0  0.0      0     0 ?        S    Oct09   0:00 [pdflush]
33  root          6  0.0  0.0      0     0 ?        S<   Oct09   0:00 [khelper]
34  root        525  0.0  0.0      0     0 ?        S<   Oct09   0:00 [kcopyd]
35  root          5  0.0  0.0      0     0 ?        S<   Oct09   0:00 [kblockd/0]
36  root       4790  0.0  0.6   2696  1624 pts/2    Ss+  Oct09   0:00 bash
37  root       4788  0.0  1.2   8252  3080 ?        S    Oct09   0:00 xterm
38          按任意键继续
```

查看已登录用户信息的输出如下所示。

```
39          ----------------------------------------------------------
40            develop              linux              10/10/08
41          ----------------------------------------------------------
42                      1: 改变字体颜色
43                      2: 查看进程信息
44                      3: 查看用户信息
45                      h: 帮助
46                      Q: 退出
47          ----------------------------------------------------------
48          Enter Your Choice[1,2,3,h,q]:3
49  root     tty1          Oct  9 13:07
50  root     pts/2         Oct  9 23:13
51  root     pts/3         Oct 10 06:33 (192.168.1.7)
52  root     :0            Oct  9 23:12 (console)
53          按任意键继续
```

【代码解析】源程序中各行的详细解释如下。

第 1～20 行：函数 colour 的定义，根据输入的命令行参数修改文本的颜色。在修改颜色时，使用了 echo 命令输出具有特殊含义的字符，从而达到设置颜色的目的。

第 21 行：调用 echo 命令输出控制字符，实现清屏。

第 22 行：调用 trap 命令实现信号的屏蔽。trap 命令定义了程序接收到信号时的处理方法，trap 的语法规则如下所示。

```
trap 命令 信号
```

当命令是""时，表示忽略信号。当命令是-时，表示采用默认的处理方式。

信号可以用数字代替，可以同时定义多个信号的处理，信号间用空格分隔。

第 23～25 行：获得当前时间、主机名、当前用户等信息存放到变量中，用于显示到菜单顶部。

第 26～38 行：调用 while 循环显示菜单，获取用户的输入。注意：在显示菜单时使用了输入重定向<<。

第 39～40 行：读取用户的选择。

第 41～100 行：调用 case 表达式分别处理用户的选择。

第 42～79 行：当用户输入 1 改变颜色后的处理。显示二级菜单，由用户选择颜色并调用 colour 函数实现颜色的变更。

第 80～82 行：当用户输入 2 显示进程信息的处理。调用 ps 命令显示头十条进程的信息。

第 83～85 行：当用户输入 3 显示用户信息的处理。调用 who 命令输出当前已登录的用户信息。

第 86～92 行：当用户输入 h 帮助时，显示相应的帮助信息。

第 93～95 行：当用户输入 q 时，退出程序。

第 96～99 行：当用户输入其他字符时，清屏并继续循环显示主菜单。

2.4　小结

shell 是用户访问 Linux 系统的用户接口，熟练掌握 shell 的使用方法，对于程序开发人员具有重要意义。本章简要介绍了 Linux 下 shell 的种类和使用方法，重点讲解了使用 shell 进行程序设计的方法。通过本章的学习，读者应该意识到 shell 是在 Linux 系统下编程必须要使用的工具，认真掌握好 shell 的相关命令和脚本编程方法，对于提高 Linux 系统下程序设计水平有很大帮助。shell 的功能强大，读者可以结合使用 Linux 系统的过程中加深对 shell 的理解。

第二篇　C 语言及编程环境

第 3 章　C 语言简介

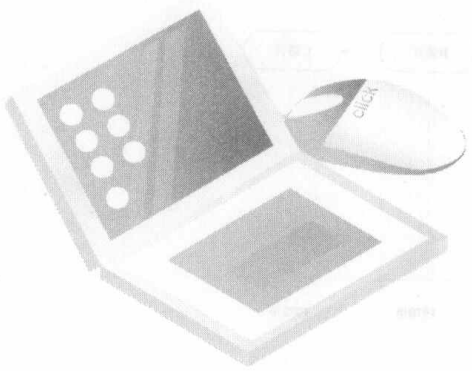
第 4 章　vi 编辑器

第 5 章　gcc 编译器

第 6 章　make 的使用

第 7 章　程序调试

第 8 章　创建与使用库



第 3 章　C 语言简介

C 语言是一种通用的高级程序设计语言，它以运行效率高、可移植性强而著称。绝大多数操作系统都提供了 C 语言编译环境，Linux 系统也不例外。本章将向读者简单介绍 C 语言的基础知识及基本语法。需要注意的是，本章的内容并不是全面的 C 语言参考手册，如果读者需要详细的语法参考，请参见相关的 C 语言书籍。在本章的学习中，读者应重点掌握如下内容。

- 了解 C 语言的发展历史及基础知识。
- 掌握 C 语言的基本语法。
- 理解使用 C 语言进行程序设计的方法。

3.1　C 语言概述

C 语言是当前最为流行的高级编程语言。特别是在系统程序设计过程中，C 语言具有明显的优势。Linux 操作系统自身就是由 C 语言编写而成的。

3.1.1　C 语言的发展简史

C 语言是由 ALGOL 60 语言发展来的。1963 年，剑桥大学的科研人员在 ALGOL 60 的基础上创建了 CPL 语言，后来又对 CPL 语言进行了优化，产生了 BCPL 语言。

1970 年，美国的贝尔实验室成功研制了著名的 UNIX 操作系统。最初，UNIX 操作系统是以汇编语言编写的。为提供更好的可移植性，贝尔实验室的科研人员决定研制一种新的开发语言用于实现 UNIX 操作系统。这种新研制的编程语言就是 C 语言的前身 B 语言。B 语言是由 BCPL 语言演化而来的，其主要设计思想源于 BCPL 语言。

B 语言是解释执行的，数据类型不够丰富。1973 年，D.M.Ritchie 在 B 语言的基础上实现了 C 语言，并与 K.Thompson 一起用 C 语言重新实现了 UNIX 操作系统。C 语言的发展演变历程，如图 3-1 所示。

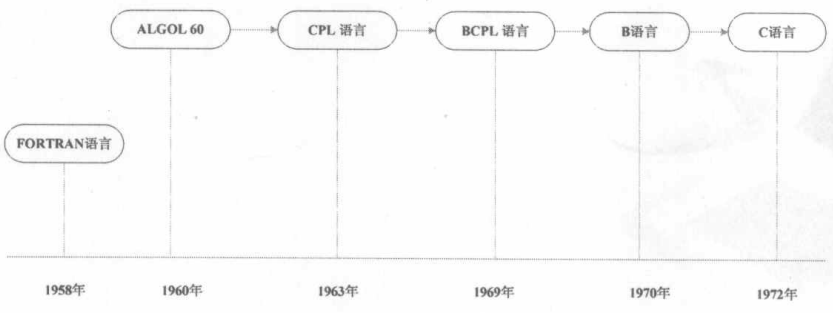


图 3-1　C 语言的演变历程

3.1.2　C 语言的特点

C 语言具有多方面的优点，主要包括以下几点。

- 通用性好，语言表达能力强。C 语言被绝大多数的计算机系统所支持，许多操作系统本身就是用 C 语言实现的。因此，C 语言具有良好的通用性。另外，C 语言支持对地址的运算及按位操作。这在某种程度上可以取代汇编语言的部分功能。
- 语言简洁，使用灵活。C 语言只包括 32 个关键字和 9 种控制语句，支持结构化程序设计。
- 提供丰富的运算符和数据类型。C 语言提供了 34 种不同的运算符，支持对二进制位的直接运算。通过各种运算符的组合，可以实现各种复杂的数学、逻辑运算。在数据类型上，C 语言提供了诸如整型、浮点型等多种数据类型。特别是其提供的指针类型，提供了对地址直接访问的功能。另外，C 语言还支持自定义数据类型的功能。
- 生成的代码运行效率高。实践证明，用 C 语言编写的代码，其运行效率仅比汇编语言代码低 10%～20%。因此，在编写系统软件过程中，C 语言是不错的首选语言。
- 可移植性好。所谓可移植，是指程序可以不加修改的从一个环境中移植到另外一个完全不同的环境中运行。在一台机器上用 C 语言编写的程序，只需在新的环境下重新编译链接一下即可运行，具有良好的可移植性。

3.2　C 语言的组成元素

本节将介绍 C 语言的组成元素，主要包括 C 语言的字符集、标识符、关键字及常量变量的定义等内容。

3.2.1　字符集

C 语言支持的字符集包括英文大小写字母、10 个阿拉伯数字以及下画线字符，这些字符集将用来构成后面介绍的标识符和变量等元素。不在 C 语言字符集中的字符，可以出现在 C 语言的注释中，或者是出现在字符串变量的内容中。

在有些情况下，需要在程序中输入非可见字符，如常见的回车符、退格符等。C 语言提供了对转义字符的支持，可以将这些不能直接在程序中输入的字符以转义的形式输入到程序中。常见的转义字符包括以下几种。

\n　换行符。
\t　水平制表符。
\v　垂直制表符。
\b　退格符。
\f　换页符。

\r　回车符。

除了上述以“\”+单字母方式的转义字符外，C 语言还支持以八进制和十六进制数字方式表示的转义。用八进制方式表示的转义方法为“\ddd”，其中 ddd 是三位八进制数字给出的 ASCII 值。用十六进制方式表示的转义方法为“\xddd”，其中 ddd 是三位十六进制数字给出的 ASCII 值。例如，退格符“\b”的八进制转义表示为“\010”，而十六进制转义表示为“\x008”。

提示：C 语言是区分大小写的，例如，变量 a 和变量 A 是两个不同的变量。

3.2.2　标识符

标识符是用户在编程过程中，对使用到的常量变量名称、函数名称等所起的名字。在 C 语言中，标识符必须以字母或者下画线开头。标识符可以由字母、数字或者下画线组成，例如，abc、_name、abc123 等都是合法的标识符。

在定义标识符时，应该注意以下几点。

- C 语言不限制标识符的长度，但标识符长度受 C 编译器限制。通常情况下，标识符长度不应超过 31 个字节。
- 标识符的大小写是有区别。例如，标识符 a 和标识符 A 是两个不同的标识符。
- 读者在进行程序设计的过程中，应该尽量使用有意义的标识符，以增强程序的可读性。

3.2.3　关键字

关键字是指对于编译程序来说具有特殊意义的标识符，用户自定义的标识符不能与关键字冲突。C 语言的关键字如表 3-1 所示。

表 3-1　C 语言的关键字

auto	do	goto	signed	unsigned
break	double	if	sizeof	void
case	else	int	struct	volatile
char	enum	long	static	while
const	extern	register	switch	typedef

提示：在 C 语言中，所有关键字都是小写的。

3.2.4　常量的类型

在程序设计中，除去关键字外，数量最多的就是各种常量和变量。在程序执行过程中，常量的值是不能变化的。C 语言中主要有整型、浮点型、字符型、字符串型 4 种常量。

整型常量用来表示一个整数。在程序中，可以用十进制、八进制和十六进制的方式表示

常量。十进制的方式是缺省的表示方式，直接在程序中使用即可。例如，赋值语句“int a=10;”，其中的 10 就是一个整型常量。如果用八进制表示，则应该在数字前加 0 。如果用十六进制表示，则应该在数字前加 0x。在前面的例子中，如果用八进制和十六进制实现，分别为“int a=012;”、“int a=0x0a;”。

浮点型常量用来表示一个有符号的实数。浮点型常量包括整数、小数点和指数 3 部分。其中的整数部分为零时，可以略去。例如，3.14、3.0E2 都是合法的浮点型常量。

字符常量是指单个字符表示的常量。该字符必须包含在单引号内。在前面介绍过的转义字符也是一种字符常量，如'a'、'\n'、'\010'。

字符串常量在C语言中用于表示一个字符串，是由一对双引号括起的字符序列。字符常量在内存中占据一个字节的存储空间，而字符串常量占据多个字节的存储空间。字符串常量的末尾包含一个字符串结束符，因此字符串常量占据的内存空间比其包含的字符串多 1。字符串常量“China”在内存中占用的存储空间是 6 个字节，其内存映像，如图 3-2 所示。

C	h	i	n	a	\0

图 3-2　字符串常量的内存映像

提示：在C语言中，字符串常量或者变量的末尾都包含了一个字符串结束符。该结束符是一个字符常量“\0”，其 ASCII 码值为 0。因此，字符串常量或者变量的长度实际上要比其数据内容长度大 1。

3.2.5　变量的类型

在程序运行过程中，其值可以变化的量称为变量。常量可以不加声明直接在程序中使用，而变量则不同，在使用前必须进行声明。通常情况下，变量统一在函数的开始进行声明。变量的声明可以包括 3 个部分：一是变量的类型，二是变量的存储类型，三是变量的作用域。

C语言支持多种变量类型，基本上可以包括 3 类：简单类型、构造类型和指针类型。简单变量类型主要有整型变量 int、长整型变量 long（无符号长整型 unsigned long）、短整型变量 short（无符号短整型 unsigned short）、浮点型变量 float、双精度型变量 double、字符变量 char。

构造类型的变量是由简单类型变量组合而成的，主要包括数组变量、结构变量、联合变量等。指针变量是一种特殊的也是非常有用的变量，其值用来表示某个变量在内存中的地址。指针类型在 C 语言中占有重要的地位，同时由于指针的误用会导致不可预料的后果，导致指针为部分程序设计人员所诟病。如图 3-3 所示描述了整型指针变量 p，该指针变量指向的地址内存储的数据为 0x1234。可以通过运算符*取得指针所指向的存储区的内容。如图 3-3 中*p 的值即为 0x1234。

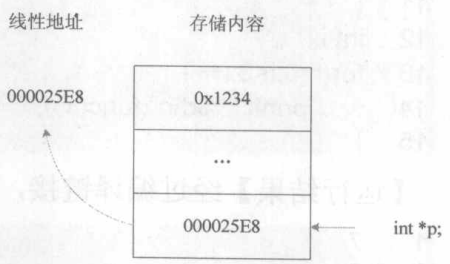


图 3-3　指针变量

3.2.6 变量的存储类型

在 C 语言中，变量还有其存储类型 auto、static、extern 和 register。存储类型规定了变量的存储方式。

1．auto 存储类型

auto 存储类型的变量也称为自动变量。在函数中声明的变量，如果不进行特殊说明都是自动变量。在系统调用变量所在的函数时，系统将从堆栈中自动分配存储空间，在函数返回后将由系统自动释放这些空间。如果要声明一个自动变量，可以用 auto 关键字进行说明，也可以省略存储类型说明，系统默认存储类型就是 auto 类型。例如：

```
int func(int x)
(
int a;
a=x;
…
)
```

这里的变量 x 和 a 都是自动变量。

2．static 存储类型

static 存储类型用于表示该变量在两次调用间保持原值。也就是说，用 static 存储类型声明的变量，在两次调用过程中，其存储空间是不会被系统自动释放的。static 类型的局部变量在编译的过程中赋初值。在运行过程中，除非使用赋值语句，否则不会对变量赋值。

【范例 3-1】验证 static 局部变量的性质。其实现过程如示例代码 3-1 所示。

示例代码 3-1

```
1   #include <stdio.h>                              /*头文件*/
2   /* 函数 func 的定义，计算整数与静态变量 n 的和*/
3   int func(int x)
4   {
5   static int n=5;          /*静态局部变量 n 定义，注意：其初始化过程在编译阶段已完成*/
6   n = n+1;                                        /*对静态变量加 1*/
7   return n+x;                                     /*返回静态变量与整数的和*/
8   }
9   /*主函数*/
10  main()
11  {
12  int i;                                          /*循环变量*/
13  for(i=0;i<3;i++)                                /* for 循环*/
14      printf("%d\n",func(1));                     /*调用函数 func 并输出结果*/
15  }
```

【运行结果】经过编译链接，运行上述程序，结果如下所示。

```
1   7
2   8
3   9
```

【代码解析】本例中，在函数 func 中声明了静态局部变量 n。在主函数中用同样的参数

值“i”循环调用该函数，根据运行结果可以看到，3 次循环输出的结果是不同的，这表明在函数 func 中的静态局部变量在这 3 次循环过程中保留了其值。这也正是 static 类型局部变量的特点。

在本例中，各源代码的解释如下。

第 1 行：头文件。

第 3～8 行：函数 func 的定义。在函数中，声明了 static 类型的局部变量 n，并赋初始值 5。根据对 static 的介绍可以知道，该赋值过程是在编译时完成的，在运行过程中，该赋初值的语句是不会被执行的。

第 10～15 行：主函数的定义。在主函数中循环调用 func，并输出函数 func 的返回结果。

提示：static 存储类型用于定义全局变量时，具有不同的含义，指该变量只在当前的源程序中可见。

3．extern 存储类型

extern 变量也称为外部变量，是在函数外部定义的变量。如果在外部变量定义之前的函数中要使用外部变量，则应该用关键字 extern 对该变量进行声明。通常情况下，可以在头文件中定义外部变量，在源代码中用 extern 进行声明后引用。外部变量使用不宜过多，否则将造成函数间耦合度过高的问题。

【范例 3-2】验证 extern 变量的性质。其实现过程如示例代码 3-2 所示。

示例代码 3-2

```
1    #include <stdio.h>                    /*头文件*/
2    main()                                /*主函数*/
3    {
4    extern int n;                         /*外部变量的声明*/
5    printf("%d\n",++n);                   /*输出变量加 1 后的值*/
6    }
7    int n = 10;                           /*变量的定义*/
```

【运行结果】经过编译链接，运行上述程序，结果如下所示。

```
1    11
```

【代码解析】本例中，在程序末尾 main 函数的外部声明了全局变量 n，在主函数引用该变量。由于在主函数引用时，该变量尚未声明，对于主函数是不可见的。因此，在主函数中用 extern 声明该外部变量。在本例中，各源代码的解释如下。

第 1 行：头文件。

第 2～6 行：主函数的定义。在函数中，声明了 extern 存储类型的变量 n，将 n 值加 1 后进行输出。

第 7 行：全局变量 n 的声明。

4．register 存储类型

register 变量也称为寄存器变量，是为了提高程序的运行效率，将使用频率高的变量直接存储到寄存器中，而不是保存到内存中。

3.2.7 变量的作用域

C 语言中所有的变量都有自己的作用域。所谓作用域，就是指该变量在什么范围内是可见的。C 语言中的变量根据作用域的不同可以划分为局部变量和全局变量。

局部变量也称为内部变量，是在函数内部进行定义的，一旦离开该函数，该变量将是不可见的。不同的函数内，可以使用相同名称的局部变量。大多数情况下，在程序中使用的变量都是局部变量。全局变量是在函数外部定义的变量，它并不属于某个固定的函数，而是在整个程序范围内可见。如果全局变量名称与函数内部的局部变量名称重复，则在函数内部将使用局部变量。

提示：在程序设计过程中，不建议过多使用全局变量。多个函数中需要共享的变量可以通过传递参数的方法加以解决。

3.2.8 运算符

所谓运算符，就是指在 C 语言中进行各种运算所使用的符号。在 C 语言中，可以进行算术运算、关系运算、逻辑运算等若干运算。其对应的运算符可分为以下几类。

- 算术运算符：用于进行各类数值的运算。包括加（+）、减（-）、乘（*）、除（/）、求余（或称模运算，%）、自增（++）、自减（--）共 7 种。在使用++和--运算符时，要特别注意运算符在变量前与变量后的区别。当运算符在变量前时，是先对变量进行加减运算再取变量值；当运算符在变量后时，是先取变量值再对变量进行加减运算。
- 关系运算符：用于比较运算。包括大于（>）、小于（<）、等于（=）、大于等于（>=）、小于等于（<=）和不等于（!=）6 种。
- 逻辑运算符：用于逻辑运算。包括与（&&）、或（||）、非（!）3 种。
- 位操作运算符：参与运算的量，按二进制位进行运算。包括位与（&）、位或（|）、位非（～）、位异或（^）、左移（<<）、右移（>>）6 种。
- 赋值运算符：用于赋值运算，分为简单赋值（=）、复合算术赋值（+=、-=、*=、/=、%=）和复合位运算赋值（&=、|=、^=、>>=、<<=）3 类共 11 种。
- 条件运算符：这是一个三目运算符，用于条件求值（?:）。
- 逗号运算符：用于把若干表达式组合成一个表达式（,）。
- 指针运算符：包括取内容（*）和取地址（&）两种运算符。
- 求字节数运算符：用于计算数据类型所占的字节数（sizeof）。
- 特殊运算符：有括号( )，下标[ ]，成员（→,.）等。

【范例 3-3】验证 sizeof 与 strlen 的区别。其实现过程如示例代码 3-3 所示。

示例代码 3-3

```
1    #include <stdio.h>                    /*头文件*/
2    main()                                /*主函数*/
```

```
3    {
4    char *p="abcdef";                          /*定义字符串指针变量*/
5    char q[9]="12345678";                      /*定义字符数组*/
6    printf("%d %d\n",sizeof(p),strlen(p));     /*输出字符串指针的尺寸和字符串的长度*/
7    printf("%d %d\n",sizeof(q),strlen(q));     /*输出字符数组的尺寸和字符数组的长度*/
8    }
```

【运行结果】经过编译链接，运行上述程序，结果如下所示。

```
1    4 6
2    9 8
```

【代码解析】本例中，定义了字符串指针变量 p 和字符串数组变量 q。在主函数中分别用 sizeof 和 strlen 求二者的长度。从输出可以看到，sizeof(p)求得的值为 4，这是由于 p 是一个指针变量，指针类型的变量长度就是 4；strlen(p)求得的长度为 6，这是字符串“abcdef”的长度。用 sizeof(q)求得的长度为 9，这是字符串数组的大小；而用 strlen(q)求得的是 8，是字符串数组中有效字符串（不包括字符串结束符）的长度。在本例中，各源代码的解释如下。

第 1 行：头文件。

第 4 行：定义字符串指针变量 p。

第 5 行：定义字符串数组变量 q。

第 6～7 行：分别调用 sizeof 和 strlen 求变量的长度。

提示：要注意 sizeof 运算符与求字符串长度函数 strlen 的区别。sizeof 可以用于数据类型，也可以用于任意类型的变量；而 strlen 只能用于字符串变量。用 sizeof 求字符串变量的长度时，是指整个字符串数组的尺寸；而用 strlen 求得的字符串长度则不包括最后的字符串结束符。

3.2.9　注释方法

注释是为增强源程序的可读性而在代码中添加的说明性文字。在编程过程中，应该养成使用注释的良好编程习惯。C 语言中的注释格式为/*注释内容*/。注释内容可以是除“*/”之外的任何字符组合。在编译过程中，C 语言的编译器将忽略注释部分的内容。

3.3　语句与控制结构

编程语言的主要成分是语句。语句实现程序的功能，同时也控制着程序的执行流程。C 语言中的语句有多种，主要包括表达式语句、函数调用语句、控制语句、复合语句和空语句。

3.3.1　表达式语句的组成

表达式语句由表达式加上分号“;”组成，其形式如下所示。

```
表达式;
```

在 C 语言中，表达式语句使用非常多。如 x=y+1;x++;等都是合法的表达式语句。表达式

语句的语法结构如图 3-4 所示。

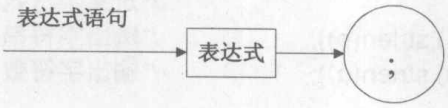


图 3-4　表达式语句的语法结构

3.3.2　复合语句

C 语言中，复合语句由若干个有序语句组成。这些语句用花括号{}括起来就构成了复合语句。复合语句的形式，如下所示。

```
{
语句 1;
语句 2;
┆
语句 n;
}
```

从语法上看，复合语句等价于一条语句，在程序中作为单独的一个语句发挥作用。不过，复合语句的最后是没有“;”的。复合语句的语法结构，如图 3-5 所示。

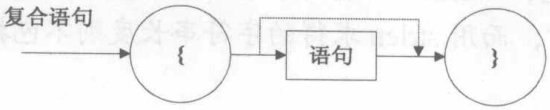


图 3-5　复合语句的语法结构

3.3.3　函数调用语句

在 C 语言中，可以将实现某个独立功能的代码封装成为函数，供程序中需要的地方进行调用。函数调用语句由函数名称、实际参数和分号“;”组成，其形式如下所示。

```
函数名称(实际参数…);
```

函数调用语句执行时，将把实际参数传入给相应函数，执行函数体中的语句并将值返回给调用的程序。

3.3.4　控制语句

控制语句用于控制程序执行的流程。C 语言有 9 种控制语句，可分成条件语句、循环语句和转移语句 3 类。

- 条件语句：包括 if 语句、switch 语句。
- 循环语句：包括 do while 语句、while 语句、for 语句。
- 转移语句：包括 break 语句、goto 语句、continue 语句、return 语句。

程序在执行时，一般是按照语句的先后顺序进行的。但是在解决复杂的问题时，常常需要依据条件来改变执行的顺序，选择需要执行的语句。C 语言提供了 if-else 结构和 switch 结构解决这一问题。

1．if 语句

if-else 结构的形式如下所示。

```
If(条件表达式)
语句;
else if(条件表达式)
语句;
 ┆
else
语句;
```

if-else 条件语句的语法结构如图 3-6 所示。

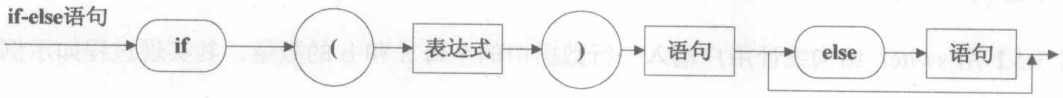


图 3-6　if-else 的语法结构

【范例 3-4】用 if-else 结构实现判断两个整型变量大小的函数。其实现过程如示例代码 3-4 所示。

示例代码 3-4

```
1   #include <stdio.h>                                    /*头文件*/
2   /* 函数 maxint 的定义 返回两个输入参数中大的一个 */
3   int maxint(int x,int y)                               /*函数定义*/
4   {
5       if(x > y)                                         /*判断 x 是否大于 y*/
6           return x;                                     /*如果 x 大于 y，则返回 x*/
7       else
8           return y;                                     /*如果 x 小于等于 y，则返回 y*/
9   }
10  main()                                                /*主函数*/
11  {
12      printf("%d \n",maxint(5,6));                      /*调用函数 maxint 输出两个整数中较大的一个*/
13  }
```

【运行结果】经过编译链接，运行上述程序，结果如下所示。

```
14  6
```

【代码解析】本例中，定义了函数 maxint。在函数中用 if-else 结构实现了条件判断，返回数值较大的输入参数，在主函数中用参数 5 和 6 调用 maxint。从输出可以看到，返回的较大的整数是 6。在本例中，各源代码的解释如下。

第 1 行：头文件。

第 3～9 行：使用 if-else 结构判断输入的两个参数的大小，返回较大的一个。

第 12 行：用参数 5 和 6 调用函数 maxint，并输出返回的较大的整数。

2. switch 语句

switch 语句也称为开关语句，它实现的功能类似于日常生活中的开关。switch 语句的形式如下所示。

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

在执行 switch 语句时，首先计算表达式的值，其结果一定为整数或者字符型值。将该计算结果与 case 中的常量表达式相比较，如果相等，则执行后面的对应语句。如果没有任何匹配的常量表达式，则执行 default 后面的语句。

【范例 3-5】用 switch 结构统计用户输入一行数据中的字母 a 和 b 的数量。其实现过程如示例代码 3-5 所示。

示例代码 3-5

```
#include <stdio.h>                          /*头文件*/
main()                                      /*主函数*/
{
    char c;                                 /*字符变量，用于保存键盘输入的字符*/
    int  ca,cb;                             /*整型变量，分别用于保存字符 a 和字符 b 的输入数量*/
    int  bQuit;                             /*整型变量，用于判断是否退出循环。为 1 时退出循环*/
    ca=0;cb=0;                              /*初始化变量*/
    bQuit = 0;                              /*初始化变量*/
    for(;;)                                 /*for 循环*/
    {
        switch((c=getchar()))               /* switch 分支语句，判断从键盘输入的单个字符*/
        {
            case 'a':                       /*输入了字母 a 的处理*/
              ca++;                         /*相应变量加 1*/
              break;
            case 'b':                       /*输入了字母 b 的处理*/
              cb++;                         /*相应变量加 1*/
              break;
            case '\n':                      /*输入了换行符后的处理，退出循环*/
              bQuit = 1;                    /*置退出标志*/
              break;
            default:                        /*输入其他值的处理，继续循环*/
              break;
        }
        if(bQuit) break;                    /*输入换行符后，退出循环*/
    }
    printf("The number of a=%d;The number of b=%d\n",ca,cb); /*输出字母 a 和字母 b 的输入个数*/
}
```

【运行结果】经过编译链接，运行上述程序，结果如下所示。

```
29   asjdhfkacbsa /*随意输入字符串*/
30   The number of a=3;The number of b=1
```

【代码解析】本例中，循环调用 getchar 函数从键盘输入字符，直至输入换行符中断循环。在循环中调用 switch 结构统计字母 a 和 b 的数量，最后输出输入的一行数据中字母 a 和 b 各自的数量。在本例中，各源代码的解释如下。

- 第 7～8 行：对局部变量进行初始化。
- 第 9～26 行：for 循环结构，用于循环从键盘接收输入的数据。
- 第 13～15 行：判断是否为字母 a。如果是，则对变量 ca 值加 1。
- 第 16～18 行：判断是否为字母 b。如果是，则对变量 cb 值加 1。
- 第 19～21 行：判断是否为换行符，如果是换行，则置退出标志为 1。
- 第 22～23 行：如果为其他字符，则继续循环。

3.3.5　循环控制语句

在很多实际问题中都会遇到具有规律的重复性运算，因此程序中需要对某些语句重复执行。循环语句主要由两部分组成：循环体和循环终止条件。在循环语句中，一组被重复执行的语句称为循环体。而每循环一次，通常都需要对是否中断循环进行判断，这就是循环终止条件。C 语言中有 3 种循环语句，分别是 do-while 循环、while 循环和 for 循环。

do-while 循环的形式如下所示。

```
do
{
循环体;
}while(表达式);
```

do-while 循环的执行过程为：首先执行循环体，然后判断表达式的值，如果表达式的值为真，则继续循环；如果表达式的值为假，则退出循环。while 循环的形式如下所示。

```
while(表达式)
{
循环体;
}
```

while 循环的执行过程为：首先判断表达式的值，如果表达式的值为真，则执行循环体然后继续循环；如果表达式的值为假，则退出循环。while 循环与 do-while 循环的区别在于：do-while 循环无论在什么情况下，都将至少被执行一次。而 while 循环在表达式为假的情况下，循环体可能一次都不会执行。for 循环的形式如下所示。

```
for(表达式 1;表达式 2;表达式 3)
{
循环体;
}
```

在 for 循环中，表达式 1 用于给循环变量赋初值，该表达式在循环过程中只在循环开始时被执行一次。表达式 2 用于判断是否满足循环终止的条件，如果表达式 2 为真，则继续循环；否则终止循环。表达式 3 通常用于对循环变量进行增加或者减少一定值，循环每执行一次，都会调用表达式 3 一次。

【范例 3-6】用 for 循环对 1 至 100 的整数求和。其实现过程如示例代码 3-6 所示。

示例代码 3-6

```
#include <stdio.h>                          /*头文件*/
main()                                      /*主函数*/
{
    int i;                                  /*循环变量*/
    int sum=0;                              /*整型变量，保存求得的和*/
    for(i=1;i<=100;i++)                     /*for 循环,从 1 循环至 100*/
        sum += i;                           /*累加至变量 sum */
    printf("%d\n",sum);                     /*输出求得的和*/
}
```

【运行结果】经过编译链接，运行上述程序，结果如下所示。

```
5050
```

【代码解析】本例中，使用 for 循环对 1 至 100 的整数进行求和运算。各源代码的解释如下。

- 第 5 行：定义整型变量 sum 并赋初值 0，用于保存求得的和。
- 第 6 行：for 循环。首先对循环变量 i 赋初值 1，每循环一次 i 的值加 1，一直循环至 i 值大于 100 退出循环。

3.3.6 转移语句

转移语句用于从程序当前执行的位置跳转到指定的位置，主要包括 break、continue、return、goto 等语句。goto 语句由于破坏了结构化程序设计的原则，在实际编程中应尽量减少使用。break 语句用于终止当前的循环，一般用在循环体内部，在符合终止的条件时，调用 break 语句退出循环。continue 语句用于结束本次循环，继续进行下一次循环。return 语句用于从函数中返回，可以返回指定类型的值，也可以没有返回值。

3.4 函数与程序结构

C 语言的源程序是由函数组成的。一个可执行的 C 程序必须包含且仅包含一个 main 函数。函数封装了特定的功能代码。通过函数，用户可以编写出可读性强、结构化程度高的程序。从函数定义的角度看，函数可以分为库函数和用户自定义函数两种。

3.4.1 库函数

库函数是由系统提供的，用户无需定义，也不必在程序中进行类型说明。要在程序中使用库函数，只需在程序前包含有该函数原型的头文件即可。经常用到的 printf、scanf 等函数均属于库函数。

大多数操作系统都提供了丰富的库函数，常见的如字符串处理库函数、输入/输出库函数、数学计算库函数、日期时间库函数等。

3.4.2　用户自定义函数

用户自定义函数是指根据用户的需要自行编写的函数。对于自定义函数，需要在调用模块中对函数进行类型说明后才能使用。函数的定义如下所示。

```
返回值　函数名(参数列表)
(
函数体
)
```

函数的参数在定义时称为形式参数，在调用时称为实际参数。函数可以有返回值，也可以没有返回值。如果有返回值，则在函数体中应该用 return 语句返回相应类型的值；如果无返回值，定义函数时，其返回值应为 void 类型。

3.5　数组

在程序设计过程中，把具有相同数据类型的变量顺序组织起来就成为数组。大多数编程语言都提供了对数组的支持。

3.5.1　一维数组的定义和使用

数组是一种线性的结构。第一个成员没有前驱，有惟一确定的后续。最后一个成员没有后续，有惟一确定的前驱。其他的每一个成员都有惟一确定的前驱和后继。

在 C 语言中，一维数组的定义方式如下所示。

```
类型说明符 数组名 [常量表达式];
```

类型说明符可以是任一种基本数据类型或构造数据类型，这里所声明的类型实际上是定义了数组的每一个成员的类型。其中，数组名表示用户定义的数组标识符，常量表达式表示数据元素的个数，也称为数组的长度。该表达式计算的结果应该是确定的常量，而不能包含变量。例如：int n[2];定义了一个长度为 2 的整型数组。

对数组成员的引用通过数组变量的下标完成。所谓下标，就是数组成员在整个数组中的位置顺序号，下标的值应为非负整数。在程序中要引用数组，要通过如下方式进行。

```
数组名称[下标]
```

一维数组可以在定义时初始化，也可以在程序中通过编码实现。在定义时初始化的语法如下所示。

```
类型说明符 数组名[常量表达式]={值,值,...};
```

其中大括号中的以逗号分隔的值就是数组的初始值。

提示：在C语言中使用数组时，要注意其下标是由0开始编号的。所以一个长度为n的数组，其最后一个成员应该用“数组名[n-1]”的方式引用。

【范例3-7】定义一个整型的一维数组并初始化，输出其值。其实现过程如示例代码3-7所示。

示例代码3-7

```
1   #include <stdio.h>                          /*头文件*/
2   main()                                      /*主函数*/
3   {
4       int i;                                  /*循环变量*/
5       int n[5]={3,1,4,1,5};                   /*定义整型数组，并初始化*/
6       for(i=0;i<5;i++)                        /*for 循环*/
7           printf("%d\n",n[i]);                /*输出数组成员的值*/
8   }
```

【运行结果】经过编译链接，运行上述程序，结果如下所示。

```
3
1
4
1
5
```

【代码解析】本例中，首先定义了一个一维的整型数组并初始化，然后循环输出数组的各成员的值。各源代码的解释如下。

- 第5行：定义一维数组n并进行初始化。
- 第6～7行：循环输出数组n的所有成员。

3.5.2 多维数组的定义和使用

要使用多维数组，首先要理解维的概念。所谓维，可以理解为是一个方向。例如，空间是三维的，有长度、宽度和高度3个方向。一维、二维和三维数组可以形象的表示，如图3-7所示。

由于多维数组的复杂性，本书仅介绍二维数组。三维乃至多维数组与二维数组原理相同。二维数组定义的一般形式如下所示。

```
类型说明符 数组名[常量表达式 1][常量表达式 2]
```

其中，常量表达式1表示第一维下标的长度，常量表达式2表示第二维下标的长度。可以看到，二维数组实际是比一维数组多了一组下标。例如，可以用以下方式定义一个二维的整型数组。

```
int m[2][5];
```

在这个定义中，第一维长度是2，第二维长度是5。这表明，m是由两个一维数组组成的，其中每一个一维数组由5个整数组成。引用二维数组的方式与一维数组类似，也是通过下标实现的。

```
数组名称[下标 1][下标 2]
```

二维数组初始化也是在数组定义时给各成员变量赋初值，可按行分段赋值，也可按行连续赋值。例如，可以用以下方式为上述数组 m 赋初值。

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

这两种方式都是合法的，但是第一种方式更为清晰，可读性也强。建议读者在使用时通过第一种方式赋初值。

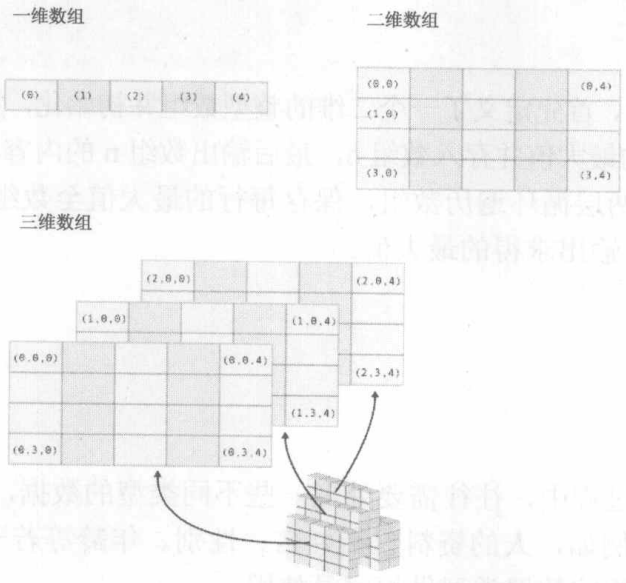


图 3-7　多维数组

提示：在 C 语言中使用字符数组时要注意，如果字符数组要当作字符串进行处理，则要注意数组的最后一位应该是字符串结束符。这样，一个 n 位长度的字符串数组，实际上只能存储 n-1 位字符数据。

【范例 3-8】定义并初始化一个二维整型数组。要求编程计算每行最大的一个元素，将这些元素组成一维数组并输出。其实现过程如示例代码 3-8 所示。

示例代码 3-8

```
#include <stdio.h>                                          /*头文件*/
main()                                                      /*主函数*/
{
    int i,j;                                                /*循环变量*/
    int n[4];                                               /*整型数组，用于保存求得的每行的最大值*/
    int maxvalue;                                           /*整型变量，临时保存较大值*/
    int m[4][3]={ {3,1,4},{2,0,5},{8,1,3},{9,2,6} };        /*定义二维整型数组并初始化*/
    for(i=0;i<4;i++)                                        /*for 循环，循环次数为数组第一维的长度*/
    {
        maxvalue = m[i][0];             /*以第一个数组成员为缺省的最大值*/
        for(j=1;j<3;j++)                /*for 循环，循环次数为数组第二维的长度减 1 */
        {
            if(m[i][j] > maxvalue)      /*判断数组成员是否比当前最大值还要大*/
                maxvalue = m[i][j];     /*设置新的最大值*/
        }
```

```
16              n[i] = maxvalue;                /*查找一行最大值结束，将求得的最大值保存到数组 n 中*/
17          }
18          for(i=0;i<4;i++)                    /*循环输出数组 n 的值*/
19              printf("%d\n",n[i]);
20      }
```

【运行结果】经过编译链接，运行上述程序，结果如下所示。

```
1    4
2    5
3    8
4    9
```

【代码解析】本例中，首先定义了一个二维的整型数组并初始化，然后两层嵌套循环遍历数组的各成员，找出每行的最大值并存入数组 n，最后输出数组 n 的内容。各源代码的解释如下。

- 第 7～16 行：两层循环遍历数组，保存每行的最大值至数组 n。
- 第 17～18 行：输出求得的最大值。

3.6 结构

在解决实际问题的过程中，往往需要处理一些不同类型的数据，而且这些数据是要作为一个整体进行处理的。例如，人的资料包括姓名、性别、年龄等若干信息。这种情况下，C 语言提供了一种称为结构的数据类型供程序员使用。

3.6.1 结构的定义

定义一个结构的一般形式如下所示。

```
struct 结构名
    {成员列表};
```

成员列表由若干个成员组成，每个成员都是该结构的一个组成部分。对每个成员也必须进行类型说明，其形式如下所示。

```
类型说明符 成员名;
```

同一结构的不同成员可以具有同一数据类型，也可以具有不同的数据类型。

结构定义完成后，需要定义该结构类型的变量。变量的定义可以在定义结构时进行，也可以通过结构名称单独进行。这两种定义结构变量的方式如下所示。

```
struct 结构名{成员列表}  结构变量名称 1,结构变量名称 2,...;
struct 结构名   结构变量名称 1,结构变量名称 2,...;
```

其中，第一种方式用于在定义结构的同时定义该结构类型的变量，第二种方式用于先定义了结构类型，再定义该结构类型的变量。在定义结构变量时，可以一并进行结构变量的初始化。初始化方式与数组的初始化相似，如下所示。

```
struct personal
{
```

```
char name[10];
int age;
char sex[2];
} user1={"name1",20,"男"};
```

3.6.2　结构成员的引用

要引用结构变量中的成员，通过运算符“.”进行。如果结构变量是一个指针，则要将“.”方式的引用变为“->”方式的引用，如下所示。

```
struct personal user1,*user2;
int n = name1.age;
int m = name2->age;
```

【范例 3-9】定义个人资料的结构变量并初始化，输出结构的每个成员的值。其实现过程如示例代码 3-9 所示。

示例代码 3-9

```
1   #include <stdio.h>                          /*头文件*/
2   main()                                      /*主函数*/
3   {
4       struct personal                         /*定义结构类型 personal*/
5       {
6           char name[10];                      /*结构成员，姓名，字符数组*/
7           int age;                            /*结构成员，年龄，整型变量*/
8           char sex[3];                        /*结构成员，性别，字符数组*/
9       } user1={"name1",20,"男"};              /*定义结构变量，并初始化*/
10      printf("姓名=%s  年龄=%d  性别=%s\n",user1.name,user1.age,user1.sex); /*输出结构成员
    的值*/
11  }
```

【运行结果】经过编译链接，运行上述程序，结果如下所示。

```
12  姓名=name1  年龄=20  性别=男
```

【代码解析】本例中，首先定义了结构类型 personal 和一个结构变量 user1 并初始化，然后输出结构变量的各成员。各源代码的解释如下。

- 第 4～9 行：定义结构类型和结构变量并初始化。
- 第 10 行：输出结构变量的成员值。

3.7　指针

指针是C语言中被广泛使用的一种数据类型，大量使用指针也是C语言的编程风格之一。指针可以用于各种变量，如整型指针、字符串指针、结构指针等，它被定义为指向某个内存地址的变量。

3.7.1 指针的概念

要了解指针的概念，首先要理解计算机存储器的基本原理。在程序运行过程中，数据是存储在内存中的。内存被划分成为一个个的存储单元，这些单元的基本单位是字节。C 语言中不同的数据类型占据不同大小的内存空间。例如，整型数据占用 2 个字节，而字符型数据占用 1 个字节等。为方便对这些存储单元的访问，给这些内存单元分配了地址。每个内存单元都分配惟一的地址，如图 3-8 所示。

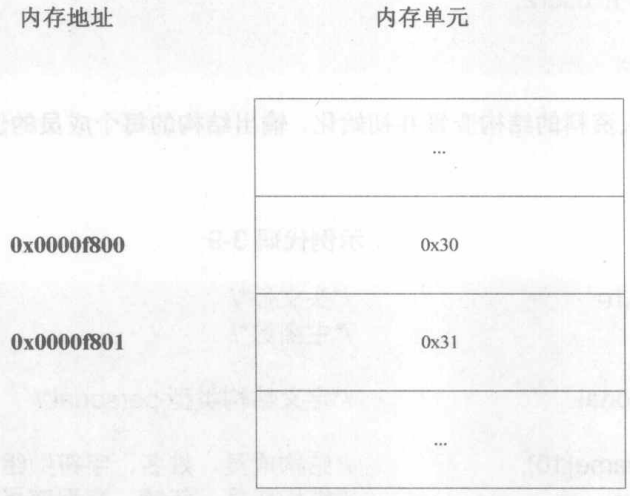


图 3-8 内存单元

通过前面的介绍可以知道，根据内存的地址就可以找到内存单元，这些地址就称为指针，即通过这个指针指向真正的内存单元。所以说指针就是指向一个变量的地址。例如：图 3-8 中地址 0x0000f800 指向的内存单元保存了字符 0（ASCII 码 0x30），这里的 0x0000f800 就是一个指针。

3.7.2 指针的定义和使用

要定义一个指针，需要使用操作符“*”。其定义的形式如下所示。

```
类型说明符 *变量名;
```

类型说明符定义了指针所指向的变量的类型。例如，int 表明该指针可用于指向一个整型变量。指针定义后，可以通过运算符“&”取一个变量的地址赋给指针。例如，int *p = &n; 表示把整型变量 n 的地址存放到指针变量 p 中。所以，&运算符也称为取地址运算符。

如何取得指针所指变量的值呢？这需要用到运算符“*”。注意：这里的“*”与定义指针时的“*”是不一样的。这里的“*”是一个运算符，意思要取地址（指针）所指向的变量（内存单元）的内容。例如，上面定义的指针 p，要取得该指针所指变量 n 的内容，通过*p 即可实现。所以，“*”也称为取内容运算符。

3.7.3　指针变量的运算

指针变量是一种较特殊的变量。在 C 语言中，指针变量的长度是固定的。在 32 位的计算机上这个值是 4 个字节，这是由于指针变量实际上就是一个地址。指针变量有以下很多特有的运算方式。

1．指针变量的赋值运算

可以将一个指针变量赋值给另外一个指针变量。经过这种赋值后的两个指针变量，指向同一内存地址，如 int *p = &n; int *q = p;。也可以将字符串的首地址赋值给一个指针变量，如 char *s="123"，这个操作实际上就相当于把内存中字符串的首地址存储到指针变量 s 中。还可以把数组的首地址存储于一个变量中，如 int m[10]; int *p=m;，这表示把数组 m 的第一个成员的地址赋给指针 p。int *p = m;等价于 int *p = &m[0]。

2．指针变量的加减运算

除赋值运算外，还可以对指针进行加减操作。通常情况下，指针加减经常用于指针指向一个数组的情况。这时指针变量的加减，并不是将指针所代表的地址增加 1 或者减去 1，而是加上或者减去所指向的变量类型的长度。例如，int m[5];int *p = m;p+1;经过上述操作后，p 实际上是指向 m 数组中第 2 个变量的地址。

【范例 3-10】编写函数对传入的整数减 1。其实现过程如示例代码 3-10 所示。

示例代码 3-10

```
1   #include <stdio.h>                          /*头文件*/
2   /*定义函数 sub_1*/
3   void sub_1(int *x)                          /*函数定义*/
4   {
5       *x -= 1;                                /*将 x 所指向的整型变量的值减 1*/
6   }
7   main()                                      /*主函数*/
8   {
9       int n=10;                               /*整型变量*/
10      sub_1(&n);                              /*调用函数 sub_1*/
11      printf("%d\n",n);                       /*输出减 1 后的整数*/
12  }
```

【运行结果】经过编译链接，运行上述程序，结果如下所示。

```
1   9
```

【代码解析】本例中，首先定义了函数 sub_1，在函数中将传入的整数减 1 后返回。各源代码的解释如下。

- 第 3～6 行：定义函数 sub_1。注意该函数的参数是一个整型指针。
- 第 5 行：将 x 所指向的整型变量的值减 1。
- 第 9 行：定义整型变量 n，并赋初始值为 10。
- 第 10 行：调用函数 sub_1。注意：这里传递的参数是指向 n 的指针。

3.8 综合实例

排序在数据结构中是很重要的一种应用。排序方法有多种，如快速排序、插入排序、堆排序等。本节重点介绍用 C 语言编程实现冒泡排序的算法及程序实现。

3.8.1 冒泡排序算法原理

排序是对一组无序的记录按照某种规则重新排列的过程。可选择的排序方法有很多种，如插入排序、选择排序等。

冒泡排序的原理是：首先将第一条记录与第二条记录进行比较，如果第一条的值大于第二条的值，则将两个记录进行交换；然后再将第二条与第三条进行比较，一直到第 n-1 条与第 n 条进行比较，每次比较都是把较大的值向后移动。这样的一次过程称为一趟冒泡排序。经过一次冒泡排序后，最大的记录已成为最后一个记录。

第二次冒泡排序的过程与第一次相似，只不过只比较到第 n-1 个记录，这是由于第 n 个记录已经是排序好的最大值。同样，第三次排序比较到第 n-2 个记录。第 n-1 次排序比较到第 2 个记录。至第 n-1 趟排序结束时，整个记录的排序已经完成。这一过程将较小的值像水中的气泡一样向上不停地冒，所以形象地称为冒泡排序。对一组无序记录的排序过程如图 3-9 所示。



图 3-9 冒泡排序过程

3.8.2 冒泡排序算法实现

采用双向链表进行数据存储，按照冒泡排序算法实现的过程如下所示。

```
1    #include <stdio.h>                                    /*头文件*/
2    /*自定义数据类型 NODE，用于表示每一个待排序的数据*/
```

```
typedef struct node
{
    int data;                                   /*数据的值*/
    struct node *next;                          /*下一个结点的指针*/
    struct node *pre;                           /*上一个结点的指针*/
} NODE;
/*插入新结点 node 至双向链表 head 中*/
InsertLt(NODE **head,NODE *node)
{
    /* 链表为空 ,以要插入的节点为头节点*/
    if(*head == NULL)                           /*第一个节点*/
    {
        (*head) = node;                         /*以新节点为链表头*/
        (*head)->next=NULL;                     /*后续节点为空*/
        (*head)->pre=NULL;                      /*前驱节点为空*/
    }
    /* 链表非空,将新加入的节点作为头节点，插入链表的头部*/
    else
    {
        node->next = *head;                     /*新节点插入链表头部*/
        (*head)->pre = node;
        (*head) = node;
        (*head)->pre = NULL;
    }
}
/*一趟冒泡排序的函数,从链表的第一个节点开始比较，直到 node 节点结束*/
PerBubble(NODE *head,NODE *node)
{
    NODE *tmp,*p;                               /*临时变量定义*/
    int n;                                      /*整型变量，用于临时存储节点值*/
    for(tmp = head;tmp!=node;tmp=tmp->next)     /*从第一个节点开始比较，到节点结束*/
    {
        if(tmp->next == NULL) break;            /*链表结束，终止循环*/
        /* 当前节点的值大于后续节点的值，交换两个节点*/
        if(tmp->data > tmp->next->data)
        {
            n=tmp->data;                        /*交换两个节点在链表中的位置*/
            tmp->data = tmp->next->data;
            tmp->next->data = n;
        }
    }
}
/*冒泡排序算法函数 head 为待排序的链表*/
BubbleSort(NODE *head)
{
    NODE *tmp = NULL;                           /*临时变量定义*/
    for(tmp=head;tmp->next!=NULL;)              /*查找双向链表的最后一个节点*/
        tmp=tmp->next;
    /*从最后一个节点开始，一直到第一个元素，分别进行一趟排序过程*/
    for(;tmp!= NULL;tmp = tmp->pre)
    {
        PerBubble(head,tmp);                    /*调用排序函数*/
    }
}
main()                                          /*主函数*/
```

```
{
    int n;                                          /*循环变量定义*/
    NODE *head = NULL;                              /*链表的头指针，初始化为空*/
    NODE *tmp,*p;                                   /*指针变量，临时保存 NODE 变量的指针*/
    for(;;)  /*循环调用 scanf 从键盘输入数据，根据输入的数据建立双向链表。输入负数则退出循环*/
    {
        scanf("%d",&n);                             /*从标准输入设备输入一个整数*/
        if(n < 0) break;                            /*输入值小于 0，退出循环*/
        tmp = (NODE *)malloc(sizeof(NODE));         /*动态申请内存*/
        memset(tmp,0x00,sizeof(tmp));               /*清空申请的内存*/
        tmp->data=n;                                /*保存输入的值*/
        tmp->next=NULL;
        tmp->pre=NULL;
        InsertLt(&head,tmp);                        /*插入到双向链表*/
    }
    BubbleSort(head);                               /*进行冒泡排序*/
    for(tmp = head;tmp!= NULL;)     /*循环输出排序后的结果，同时释放动态申请的内存空间*/
    {
        printf("%d\n",tmp->data);                   /*输出结果*/
        p = tmp;
        tmp = tmp->next;                            /*跳转至下一节点*/
        p->next = NULL;
        free(p);                                    /*释放内存*/
    }
}
```

【运行结果】经过编译链接，运行上述程序，结果如下所示。

```
1   7
2   2
3   8
4   9
5   1
6   -1
7   1
8   2
9   7
10  8
11  9
```

【代码解析】本例中，采用双向链表保存待排序的数据。首先从键盘输入待排序的数据，然后调用冒泡排序算法进行排序，最后输出已排序完成的数据。

3.9 小结

C 语言是一种功能十分强大的语言，很多读者也许对 Windows 系统下的 C 语言编程已经十分熟悉，本章的主要目的是将 C 语言中重要的语言现象进行梳理。在 Linux 系统下，C 语言中的重要元素依然使用广泛，如控制语句、函数、数组和指针等。这些基础的语法内容，在 Linux 系统下和 Windows 系统下是共通的。在本章的最后，还使用一个最常见的排序例子，来讲解如何在 Linux 系统下综合应用 C 语言。

第 4 章　vi 编辑器

vi 编辑器是 Linux 系统下提供的重要的文本编辑工具。vi 功能强大，不逊色于任何编辑器。大多在 Linux 系统下的源程序是使用 vi 书写的。本章由浅入深地介绍了 vi 的使用方法。通过本章的学习，读者应重点掌握如下内容。

- 了解 vi 编辑器的工作方式。
- 掌握 vi 编辑器常用的编辑功能。
- 能够使用 vi 编辑器灵活地进行文件编辑操作。

4.1　vi 编辑器概述

编辑器是操作系统提供的必不可少的工具之一。Windows 操作系统下的记事本、word 等都是编辑器。为方便用户的使用，Linux 系统提供了多种编辑器，如 ex、ed 等，这些编辑器多是基于行方式进行编辑。相比而言，vi 编辑器功能更加强大，它提供了直观的、面向全屏幕的编辑窗口，更符合用户的编辑需要，成为 Linux 系统下的编辑利器。同时，vi 使用方法较为复杂，读者应结合实践用心加以掌握。

4.1.1　vi 的启动

要启动 vi 编辑器，只需要在命令提示符下输入 vi 即可。如果要编辑一个已存在的文件，则可以使用“vi 文件名”的方法启动编辑器。vi 编辑器启动功能后，将显示一个全屏幕的编辑窗口。窗口的最后一行是状态栏，显示了文件名及文件的读写属性、文件大小等信息。vi 启动后的初始界面如图 4-1 所示。

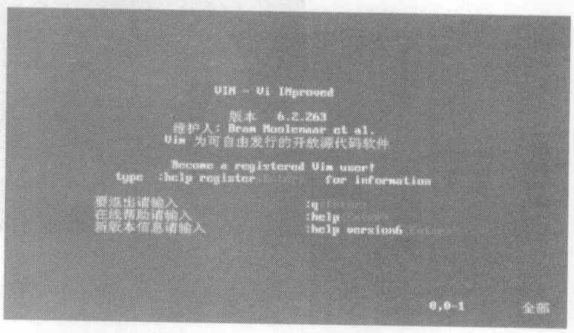


图 4-1　vi 初始界面

4.1.2　vi 的操作方式

vi 编辑器提供了 3 种不同的操作方式：命令方式、编辑方式和 ex 转义方式。

在命令方式下，从键盘输入的数据作为 vi 命令提交给编辑器。vi 编辑器启动时处于命令方式下。在这种模式下，用户可以在文件中随意移动，或者进行剪切、复制和粘贴等操作。可以在命令方式下输入部分命令进入编辑方式，如附加数据的 a 命令、插入数据的 i 命令等。也可以在命令方式下进行到 ex 转义方式，方法是输入“:”命令。此时，光标将移动至 vi 编

辑器的底部，并显示 ex 提示符“:”。在此状态下，可以在 ex 命令提示符后直接输入 ex 命令进行编辑操作。

在编辑方式下，从键盘输入的数据作为要输入的文本记录到待编辑的文件中。从编辑方式进入命令模式的方法是按<Esc>键。如图 4-2 所示是用户在命令方式下输入“i”命令进入编辑方式的截图。从图中可以看到，vi 编辑器的状态行出现提示“INSERT”，表明当前是处于插入状态。

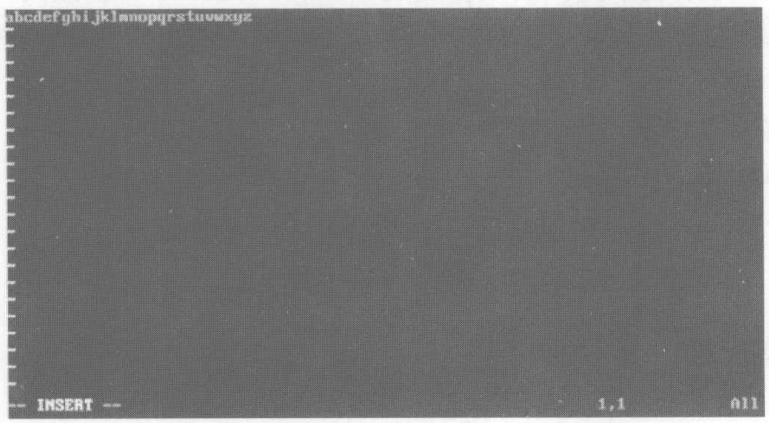


图 4-2　ex 编辑方式

第三种方式就是所谓的 ex 转义方式。ex 是 Linux 系统下的另一个编辑工具，其与 vi 的区别在于用户界面不同。所谓 ex 转义方式，就是在 vi 编辑器中调用 ex 编辑器的命令完成相应的编辑操作。如图 4-3 所示，在 ex 命令方式下，用户通过执行 ex 命令完成对被编辑文件中字符串的替换。

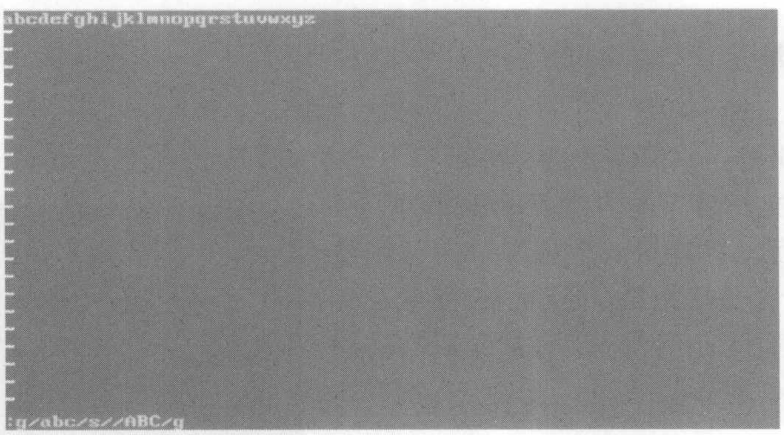


图 4-3　ex 命令方式

提示：需要注意的是，vi 编辑器是对编辑缓冲区进行操作的。其所有的修改都记录到编辑缓冲区中，只有调用执行 w 等命令向磁盘写入后，对文件所做的修改才会真正写入文件。

4.1.3　vi 编辑器的功能键

vi 编辑器中有若干键是有特殊含义的。在命令方式下，主要的功能键包括以下几个。

- <Esc>键：用于从插入方式返回到命令方式。
- <Enter>键：用于 ex 转义方式时，结束 ex 命令。或者用于编辑方式时，产生一个新行。
- /：用于从当前位置开始向后查找一个指定的串。在进行查找的过程中，可以随时输入 n 命令查找下一个匹配的项。
- ?：用于从当前位置开始向前查找一个指定的串。在进行查找的过程中，可以随时输入 n 命令查找下一个匹配的项。如图 4-4 所示是在文件中查找字符串“mn”的屏幕截图。在输入“/”命令或者“?”命令后，光标将跳转到编辑器底部，并显示“/”或者“?”命令提示符。在此状态下，直接输入要查找的串按<Enter>键即可。如果需要查找多个匹配的项，可以在命令方式下输入 n 命令即可。

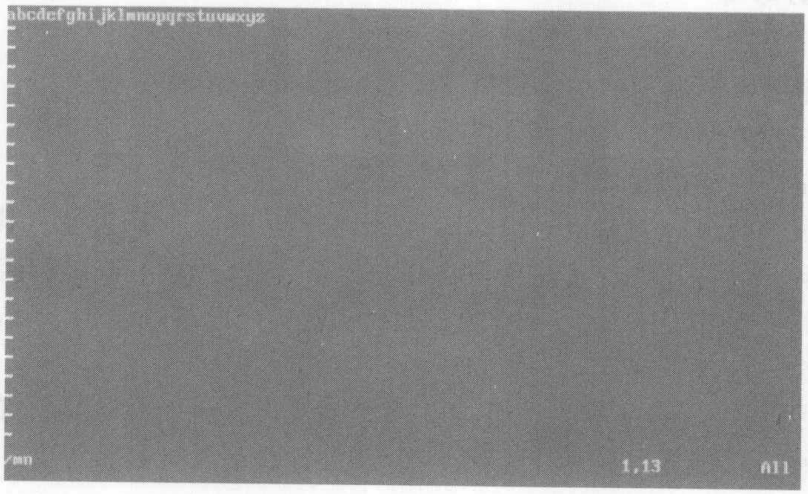


图 4-4　查找匹配的串

- :：在命令状态输入冒号将进入 ex 模式。结束 ex 命令可以按<Enter>键。

在编辑方式下，主要的功能键包括以下几个。

- 退格键：用于从当前位置删除一个字符。
- <Ctrl>+<U>键：用于将光标移回至本次编辑的起始位置重新开始编辑。刚刚进行的编辑操作将被取消。
- <Ctrl>+<V>键：消除下一个键入字符的特殊含义。通常情况下，可以用<Ctrl>+<V>键插入控制字符。例如，要向文件中插入字符“Esc”，只需要输入<Ctrl>+<V>键后，按下<Esc>键即可。
- <Ctrl>+<W>键：把光标移回至上次编辑的单词的第一个字符上。

4.1.4　退出 vi 编辑器

退出 vi 编辑器主要有以下 3 种方法。

- 命令 ZZ：在命令方式下，输入 ZZ 将保存当前的修改并退出 vi 编辑器。
- ex 转义命令“:x”：通过输入 ex 转义命令“:x”，可以保存当前的修改并退出 vi 编辑器。
- ex 转义命令“:q!”：输入 ex 转义命令“:q!”，可以不保存当前的修改，强制退出

vi 编辑器。

4.2 vi 编辑器的命令

vi 编辑器包含了复杂的命令。在命令方式下输入的数据都被解释为 vi 命令，如果格式正确，命令将被执行；否则，vi 编辑器将响铃报错。

4.2.1 光标移动命令

光标移动命令用于变换当前的编辑位置，其具体命令如下。

- 向前（右）移动：l、空格键、右箭头键。
- 向后（左）移动：h、退格键、左箭头键。
- 移动到下一行：+、<Enter>键、j、<Ctrl>+<N>键、下箭头键。其中，<Enter>键将移动至下一行的开头，其他命令将移动至下一行，而列的位置与当前行一致。
- 移动到上一行：k、<Ctrl>+<P>键、上箭头键。光标移动至上一行，列的位置保持在当前列的位置。
- 移动到行首：0（数字零）、^。其中命令 0 将光标移动到当前行的第一个字符上，而命令^则移动光标到当前行的第一个非空、非制表符的字符上。
- 移动到行尾：$。
- 移动到某行：[行号] G。执行该命令后，光标将跳转至指定行的第一个字符的位置。如果未指定行号，则将跳转到文件的最后一行的头部。
- 移动到某列：[列号] |。光标移动到当前行指定的列上。如果未指定列号，则跳至当前行的第一列。
- 向前移动一个单词：w、W。将光标移动至下一个单词的开始。
- 向后移动一个单词：b、B。光标回退至当前单词的开始。
- 光标移动至句首（尾）：句首（，句尾）。
- 光标移动至段首（尾）：段首{，段尾}。

4.2.2 滚动屏幕命令

在编辑过程中，有时需要滚动屏幕。vi 编辑器提供了多种滚动屏幕的命令，具体的命令如下。

- 滚动半页：<Ctrl>+<U>键（向上滚动）、<Ctrl>+<D>键（向下滚动）。
- 滚动一页：<Ctrl>+<F>键（向前滚动）、<Ctrl>+<B>键（向后滚动）。
- 刷新屏幕：<Ctrl>+<R>键、<Ctrl>+<L>键。该命令在屏幕显示内容混乱时特别有用。

4.2.3　文本编辑命令

文本编辑命令是由命令状态进入编辑状态的途径。vi 主要提供了以下几种编辑命令。

- 在当前位置插入：i 命令。输入 i 命令后，将由命令状态转换为编辑状态，新的文本被插入到光标所在位置之前。退出插入状态只需要按下<Esc>键即可。
- 在当前位置附加：a 命令。在当前光标所在字符的后面附加新的内容。
- 在当前行末尾附加：A 命令。在当前行的末尾附加新的内容。
- 开始新行：o、O 命令。o 命令在当前行的下面建立新的行，并进入编辑状态。O 命令在当前行的上面建立新行并进入编辑状态。

4.2.4　文本删除命令

文本删除命令如下。

- 删除字符：x 和 X 命令。x 命令删除当前光标所在位置的字符，X 命令删除光标前的那个字符。如果在 x 命令前带有前置计数，如 5x，表明要删除光标当前位置开始的相应个数的字符。
- 删除部分字符：vi 编辑器提供了 d 命令用于对部分文本进行删除。d 命令的格式为：d 位置指示符。常用的位置指标符有：w 删除当前位置的一个单词，$从当前位置删除至行尾，d 指示符删除当前行。在使用删除命令时，需要注意的是，删除操作所删除的文本被存放到一组缓冲栈中，这组缓冲栈的编号从 1 至 9。最后一次删除的内容缓存在缓冲栈 1 中，倒数第二次删除的内容在缓冲栈 2 中，以此类推。可以通过执行 p 命令把已删除的文本放到当前编辑行的下面。p 命令的格式为："+缓冲栈编号+p。

【范例 4-1】使用 vi 编辑器新建文件 a，在其中插入两行数据。两行数据内容分别为“123”和“abc”，执行 d 命令删除第一行（内容为 123），并执行 p 命令将删除的文本放到原第二行（内容为 abc）后。其实现过程如示例代码 4-1 所示（单引号中的内容使用控制字符，括号中的文字为注释）。

示例代码 4-1

```
01   vi a                                          #编辑文件 a
02   i(插入命令)123(插入第一行内容)<Enter>键          #插入数据 123
03   abc(插入第二行内容)'Esc 键'                     #插入数据 abc，并退出编辑状态，进入命
令状态
04   k                                             #向上移动一行，使当前位置成为第 1 行
05   dd                                            #删除当前行
06   p                                             #将刚刚删除的数据放到当前行的下面
```

【运行结果】调用 vi 命令，按照上述过程运行，结果如下所示。

```
01   abc
02   123
```

4.2.5 文本修改命令

文本修改命令主要包括修改的回退、重复执行上条命令、替换等。具体命令如下。

- 回退修改：u 和 U 命令。在编辑过程中，经常会遇到修改错误的情况。此时，需要用到撤销修改的功能。小写 u 命令仅仅回退上一条命令，大写 U 命令可以回退至进入本行来所做的全部修改。在编辑过程中，有时可能会误删除数据，此时 u 命令将非常有用。
- 重复执行命令：.（点）命令。在有些情况下，需要重复执行一些命令。例如，需要在所有行的最后附加相同的内容，这时，可以在其中某行插入，然后移动光标至其他行，执行“.”命令即可完成重复插入。
- 替代单个字符：s 命令。执行 s 命令可以替代当前位置的字符为其他内容。这里的其他内容可以是一个字符，也可以是多个字符。当用户在命令状态下输入 s 时，位于光标当前位置的字符将会变为“$”，此时输入的一个或者多个字符将代替该字符。
- 替代当前行：R 命令。执行 R 命令将从光标所在位置开始，替代当前行的内容。此时，vi 编辑器底部将出现“REPLACE”字样，指示当前正在处于替换状态。
- 连接两行：J 命令。可以执行 J 命令将两行内容连接成为一行。

4.2.6 文本移动命令

在编辑过程中，vi 编辑器支持将部分内容从一个位置移动或者复制至另外一个位置。文本移动命令的格式较为复杂，本书仅介绍其较常用的使用方法。

如果需要复制部分内容，则可以组合 y 命令和 p 命令来实现。即先执行 y 命令将要复制的内容复制到 vi 的缓冲区，然后移动光标至要复制的目标位置。执行 p 命令即可将在 vi 缓冲区中的要复制的数据复制到当前位置。

如果需要移动部分内容，则可以组合 d 命令（或者其他删除命令）和 p 命令来实现。在用户执行 d 命令删除数据时，数据在被删除的同时将被存储到 vi 的缓冲区中。在新的编辑位置输入 p 命令即可将被删除的数据恢复到当前位置，从而完成移动的功能。如图 4-5 所示，通过执行 d 和 p 命令完成了一行数据的移动。

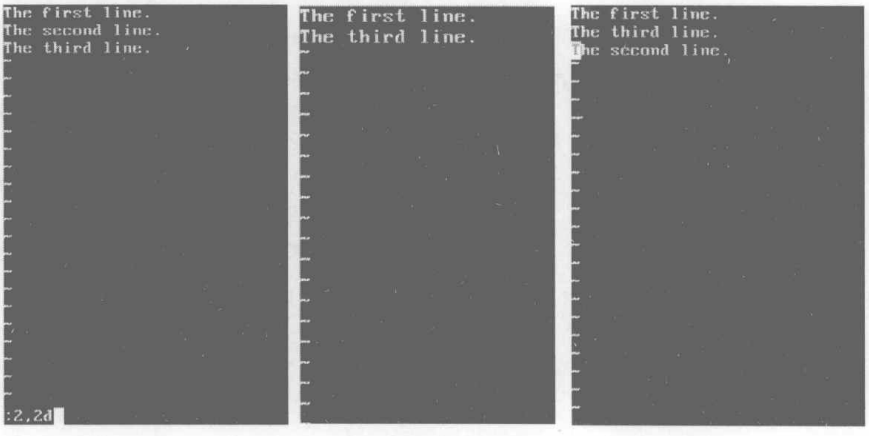


图 4-5 移动数据

在图 4-5 中，第一次操作是执行删除，执行 ex 命令方式，输入命令“2,2d”表示删除第 2 行数据。也可以将光标移动到第 2 行，直接通过执行命令 dd 删除。第二次操作将光标移动到第 2 行，然后输入 p 命令。从输出结果可以看到，被删除的原第 2 行数据已经移动到了文件的最后。y 命令和 p 命令的详细说明如下。

- 抽取命令：y 命令。所谓抽取就是将文件内容复制到无名缓冲区中，y 命令前可以加行数前缀。例如，要抽取一行执行命令 yy，要从当前位置抽取 5 行用 5yy 等。
- 放入命令：p 命令。从文件中执行 y 命令抽取到内容后，将光标移动到需要插入的位置。输入 p 命令即可实现内容的复制，vi 将存储在缓冲区中的内容复制到当前行的下面。

4.2.7　搜索命令

对于比较大的文件，搜索命令将会起到非常重要的作用。使用搜索命令可以从文件中检索指定的内容。在 ex 命令模式下，还可以利用搜索命令替换指定的内容。

搜索命令：/（?）命令、n 命令、N 命令。/（?）命令的功能是搜索指定的内容。在 vi 命令状态下，输入/，光标将跳转至屏幕底部。此时，在/后面输入要搜索的内容后按<Enter>键。如果文件中有指定的内容，则光标将跳转至第一个匹配的位置。/命令与?命令的区别是，/命令从当前位置向下搜索，?命令从当前位置向上搜索。n 命令用于按照上次搜索相同的方向继续搜索下一个匹配项。N 命令用于按照上次搜索相反的方向继续搜索下一个匹配项。

4.2.8　ex 转义命令

所谓 ex 转义命令，就是指在 vi 编辑器中支持 ex 编辑器的命令。ex 编辑器是 Linux 系统下的行编辑器，其具有强大的命令功能。但是，由于 ex 面向行方式而不是全屏幕方式编辑，在实际应用中感到诸多不便。vi 编辑器通过支持 ex 命令，将 vi 和 ex 二者的优点加以结合，提供了更加强大的编辑功能。

要进入 ex 转义方式，只需要在 vi 的命令方式下输入“:”即可。此时，光标将跳转至屏幕底部，等待接收 ex 命令。ex 命令非常繁杂，本书仅就其中经常使用的加以说明。读者如果需要了解其他的 ex 命令，请参考相关手册。

- 编辑命令：在 ex 转义模式下，输入 e+"文件名"可以打开一个新的外部的文件进行编辑。如果当前编辑的文件未进行保存，vi 将会提示用户是否进行保存。
- 写命令：在 ex 转义模式下，输入 w 命令可以对当前编辑的文件进行保存。
- 读入命令：在 ex 转义模式下，输入 r+"文件名"可以读入外部文件内容附加到当前编辑行的后面。利用这一功能可以进行文件的合并。

使用全程查找替换命令可以完成非常复杂的修改其语法格式如下所示。

```
g/模式串/命令串
```

其中，模式串是指要查找的内容的模式，最简单的模式就是一个字符串；命令串是指查找到相应的模式串后要执行的命令，较常用的命令是将查找到的模式串替换为指定的其他内

容。替换文件中的小写字母 i 为大写字母 I 的过程如图 4-6 所示。

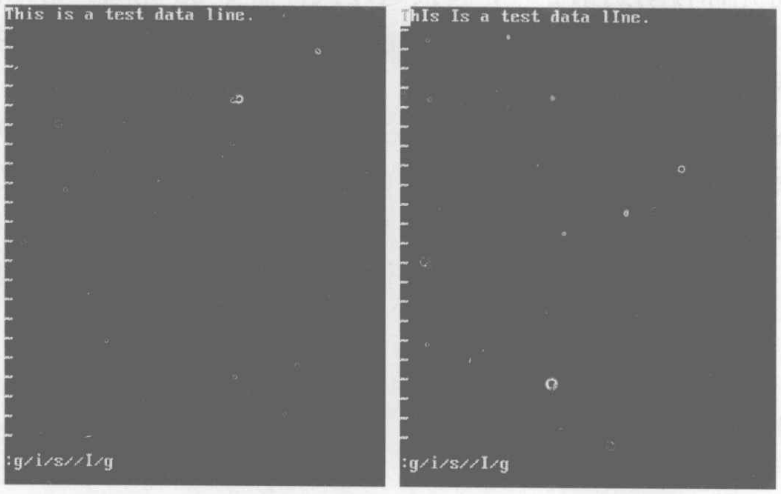


图 4-6　查找及替换

下面列出两条常见的全程查找替换命令如下。

- g/s1/s//s2/g：替换当前文件中串 s1 为串 s2。
- g/^/s//s1/g：在当前文件的所有行的开始插入字符串 s1。

4.3　vi 编辑器的选项

vi 编辑器提供了若干选项，供用户在操作过程中使用。灵活使用这些选项将有效提高用户编辑的效率。

4.3.1　选项的含义

在 vi 中的选项可以分为两类：一类是开关选项，如果要打开这类选项则使用“set 选项名”即可，而要关闭这类选项则使用“set no 选项名”；另一类是字符串选项，设置这类选项的格式为“set 选项名=选项值”。可以在 vi 编辑器的 ex 命令方式下，输入命令“set all”显示所有的 vi 选项的当前设置，如图 4-7 所示。

常用的 vi 选项有以下几个。

- ai：即 autoindent，指定每行数据总与上行数据进行对齐。缺省值为 no ai。该选项对于程序员编程时较为有用，可以打开该选项，每行程序将自动与上行进行对齐。
- directory：该选项指定了缺省的编辑缓冲区目录。vi 在编辑过程中，未保存的数据实际上是在缓冲区进行缓存的。vi 编辑器的缓冲区体现为一个临时文件，该选项就是指定临时文件的存放目录的。缺省值为/tmp。
- list：显示制表符和行尾信息。如果打开该选项，则每行的行末都会显示“$”符号，文本中的制表符则会显示“^I”。缺省值为 no list。

- exrc：如果该选项被打开，则 vi 编辑器执行时的工作目录下的.exrc 文件中的选项信息将发挥作用。否则将忽略当前工作目录下的.exrc 文件，而只检查用户主目录下的.exrc 文件。缺省状态为关闭。
- ic：即 ignorecase，指定在 vi 编辑器中进行搜索时，是否忽略大小写间的区别。其缺省值是 no ic，即不忽略大小写。
- nu：即 number，在编辑过程中对各行数据进行编号。打开该选项后，每行数据前将显示行号信息。缺省值为 no number。
- sm：即 showmatch，在输入右边的圆括号、花括号或者方括号时，将显示左边的对应的另一半括号。在编写程序的过程中，该选项比较有用。缺省值为 no sm。

```
:set all
--- Options ---
  aleph=224           fileencoding=       modeline            switchbuf=
noarabic              fileformat=unix     modelines=0         syntax=
  arabicshape         filetype=           modifiable          tabstop=8
noallowrevins       nofkmap               modified            tagbsearch
noaltkeymap           foldclose=          more                taglength=0
  ambiwidth=single    foldcolumn=0        mouse=              tagrelative
noautoindent          foldenable          mousemodel=extend   tagstack
noautoread            foldexpr=0          mousetime=500       term=vt100
noautowrite           foldignore=#        number            notermbidi
noautowriteall        foldlevel=0       nopaste               termencoding=
  background=light    foldlevelstart=-1   pastetoggle=       noterse
  backspace=          foldmethod=manual   patchexpr=          textauto
nobackup              foldminlines=1      patchmode=        notextmode
  backupcopy=auto     foldnestmax=20    nopreserveindent      textwidth=0
  backupext=~         formatoptions=tcq   previewheight=12    thesaurus=
  backupskip=/tmp/*   formatprg=       nopreviewwindow     notildeop
nobinary            nogdefault            printdevice=        timeout
nobomb                helpheight=20       printencoding=      timeoutlen=1000
  bufhidden=        nohidden              printfont=courier notitle
  buflisted           history=20          printoptions=       titlelen=85
  buftype=          nohkmap             noreadonly            titlestring=
-- More --
```

图 4-7　vi 编辑器选项

4.3.2　选项的设置方式

设置 vi 编辑器选项可以通过两种方式进行，一是直接在使用 vi 编辑器过程中通过执行 set 命令进行设置，还有一种方式是在 vi 编辑器的启动文件.exrc 中进行设置。前一种设置方式只在本次编辑过程中发生作用，编辑器退出后选项将失效，后一种方式则可以使选项对于该用户长期有效。

通常情况下，vi 编辑器的启动文件.exrc 位于用户的主目录下。在启动 vi 编辑器时，将检查当前用户主目录下的.exrc 文件，根据.exrc 文件的内容进行相应选项的设定。一个典型的.exrc 文件内容如图 4-8 所示。

```
1 set number$
2 set list$
3 set ai$
~
~
~
~
~
```

图 4-8　.exrc 文件

4.4 小结

vi 编辑器是 Linux 系统下重要的文本编辑工具。vi 编辑器功能强大，但也为读者在学习方面带来了不少的困难，特别是对于初学者来说。在编程的过程中，熟练应用 vi 编辑器将大大提高编码的效率。本章仅向读者介绍了 vi 编辑器的一些常用的功能，同时，还介绍了 vi 编辑器中常见的操作命令。对于这些功能和命令，需要读者多加练习，才能在使用过程中得心应手。

第 5 章　gcc 编译器

用户编写的源程序要投入运行，必须经过编译链接的过程。在 Windows 系统下，大多数集成开发环境都内置了编译器的功能，无需用户操作。在 Linux 系统下，必须要手动进行编译链接操作。Linux 下 C 语言的编译器是 gcc，本章将重点介绍 gcc 编译器的语法。通过本章的学习，读者应重点掌握如下内容。

- 理解程序编译链接的原理。
- 熟练掌握 gcc 编译器的常见用法。
- 能用 gcc 进行编译链接，生成可执行程序。
- 学会分析 gcc 编译链接过程中的常见错误。

5.1　编译过程简述

编译链接是使用高级语言编程所必需的操作。一个源程序只有经过编译、链接操作后，才会真正变为可以被计算机所理解并执行的二进制可执行文件。本节将向读者简要介绍编译、链接的过程和原理。由于本书以 C 语言作为编程的实现语言，因此，讲解编译的过程也是以 C 语言为示例语言进行的。

编译是指根据用户书写的源程序，进行词法和语法分析，将高级语言转换为功能有效的汇编语言代码，再由汇编语言转换为机器语言的过程。通常情况下，编译的过程可以划分为预编译、编译、优化和汇编 4 个子过程，如图 5-1 所示。

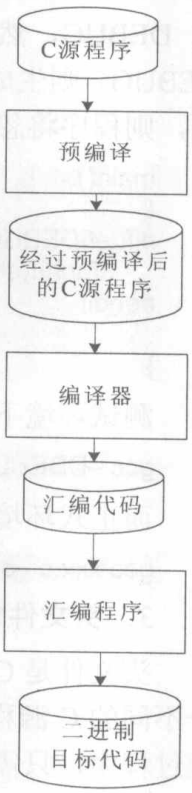


图 5-1　编译过程

5.1.1　预编译过程

预编译过程的主要工作是对源程序中的伪指令（即以#开头的指令）和特殊符号进行处理的过程。在 C 语言中，常见的伪指令包括宏定义指令、条件编译指令、头文件包含指令和特殊符号。经过预编译处理，将生成一个没有宏定义、没有条件编译指令、没有特殊符号的输出文件。在使用 gcc 编译的过程中，可以用-E 选项输出经过预编译处理后的源程序。

1．宏定义指令

经常使用的宏指令有#define 和#undef。对于#define 伪指令，其使用方法为“#define name string”。预编译程序将源程序中出现“name”的地方用“string”进行替换，但是对于字符串常量中的 name 则不会进行替换。而#undef 伪指令则将取消对某个宏的定义，取消宏定义后，从取消的位置开始，后面出现

“name”串的地方将不再被替换。

```
#define NAME "user"
#include <stdio.h>
main()
{
    fprintf(stderr,"NAME=%s\n",NAME);
}
```

经过预编译后，上述源代码将呈现的如下形式。

```
 ┆
main()
{
    fprintf(stderr,"NAME=%s\n","user");
}
```

可以看到，预编译操作将输出的变量 NAME 替换成为宏定义的值“user”，而在字符串常量“NAME=”中的 NAME 则没有被替换。

2．条件编译指令

条件编译指令包括#ifdef、#ifndef、#else、#elif、#endif 等。通过条件编译指令，程序员可以在源程序中通过定义不同的宏来决定编译程序对哪些代码进行处理。预编译程序将根据宏定义的配置信息，将那些不必要的代码过滤掉。对那些需要运行在不同平台环境下的程序来说，条件编译指令特别有用。

可以根据不同平台的宏定义，使用不同的源程序语句实现相同的功能。例如，在编码过程中经常需要打印调试信息，而正式发行的程序则需要将调试信息删除。此时可以通过定义宏 DEBUG，然后使用条件编译指令进行条件编译达到目的。在测试环境下，编译时定义宏 DEBUG，则生成的程序将包含调试信息。在正式发行环境下，编译过程中不使用 DEBUG 选项，则程序将忽略调试信息的编译。

```
main()
{
#ifdef DEBUG
    fprintf(stderr,"Debug Information...\n");
#endif
…
}
```

测试环境下编译的过程如下所示。

```
gcc –DDEBUG xxx.c –o xxx
```

而正式环境下的编译过程如下所示。

```
gcc xxx.c –o xxx
```

3．头文件包含指令

头文件是 C 语言中特有的一种文件，采用头文件的目的主要是为了使某些定义可以供多个不同的 C 源程序使用。头文件主要包含各种外部符号，如函数、数据类型等的声明。在编程过程中，只需要将相应的头文件用头文件包含指令#include 将其包含进来，即可在程序中加以引用。

在预编译过程中，预编译程序将把头文件中的定义都加入到它所产生的输出文件中，以

供编译程序对之进行处理。包含到 C 源程序中的头文件可以是系统提供的标准头文件，这些头文件一般位于/usr/include 目录下。在程序中#include 它们要使用尖括号（<>），如#include <stdio.h>。另外开发人员也可以定义自己的头文件，这些文件一般与 C 源程序放在同一目录下，此时在#include 中要用双引号（" "），如#include "test.h"。

4．特殊符号

除上述 3 种伪指令外，预编译程序还可以识别一些特殊的符号。例如，在源程序中出现的 LINE 标识将被解释为当前行号（十进制），FILE 则被解释为当前被编译的 C 语言源程序的名称。预编译程序对于在源程序中出现的这些特殊符号将用相应的值进行替换。

5.1.2　编译的过程

经过预编译处理后，将产生新的源程序。在该源程序中，只有常量、变量的定义，以及 C 语言的关键字等信息。编译程序将通过词法分析和语法分析，确认所有的指令都符合语法规则，并将其翻译成等价的中间代码或汇编代码。

在使用 gcc 进行编译时，缺省情况下是不会输出这个汇编代码的文件的。如果读者需要查看这个汇编文件的内容，可以在编译时指定-S 选项。这样，gcc 编译器将输出同名的汇编语言文件，如下所示。

```
        .file   "a.c"
        .text
.globl main
        .type    main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        addl    $15, %eax
        addl    $15, %eax
        shrl    $4, %eax
        sall    $4, %eax
        subl    %eax, %esp
        leave
        ret
        .size   main, .-main
        .section        .note.GNU-stack,"",@progbits
        .ident  "GCC: (GNU) 3.3.3 (SuSE Linux)"
```

从上面的代码看，并不是读者所熟悉的 Intel 80X86 汇编语言。实际上，这是 AT&T 汇编语言代码。读者如果有兴趣，可以查看相关资料进一步了解这种汇编语言的细节。

5.1.3　优化及汇编的过程

优化处理是编译系统中一项比较深奥的技术。它不仅同编译技术本身有关，而且同机器的硬件环境也有很大的关系。优化工作包括两部分优化，一种是对中间代码的优化，这种优

化不依赖于具体的计算机硬件，主要的工作是删除公共表达式、循环优化、无用值的删除等。另一种则主要针对目标代码的生成而进行的，这种优化与计算机的硬件结构密切相关。最主要的是考虑如何根据不同计算机体系结构，充分利用计算机的各个硬件特点，提高代码的执行效率。

要对编译过程进行优化，可以通过两种手段进行。一种方法是在 gcc 编译器编译选项中，指定-O 选项对代码进行优化。另外一种方法是定义环境变量 CFLAGS，在环境变量中指定优化的选项。涉及优化的编译器选项主要有两个：-O 和-march。

- -O 选项表示设定优化级别。优化级别共分为 3 级，由高至低分别为 O3、O2、O1。大多数情况下，指定 O2 优化选项已经足够。
- -march 选项表示为特定的 CPU 类型编译二进制代码。如果编译时指定了该选项，则编译出的代码不能在比 march 指定的级别低的 CPU 上运行。

提示：指定-march 选项时，对于不同体系结构的计算机系统，其所支持的 cpu 类型不同。对于 IBM RS/6000 系列的计算机，其包括的选项则包括 power、power2、power3、power4、power5、powerpc、powerpc64、rios、rios1、rios2、rsc 等。

代码经过优化后，再经过汇编程序的翻译，即可生成可为计算机系统所识别的二进制代码。汇编的过程实际上是将汇编语言代码翻译成为机器语言的过程。经过汇编生成的代码实际上只是目标代码，并不能直接执行。要想将目标代码变成可执行的程序，还需要进行链接的操作。关于链接的详细内容，将在下一节进行详细介绍。

5.2 链接过程简述

经过编译操作后，编译程序将生成二进制的目标代码。这些目标代码虽然是机器语言代码，但还不能直接执行。只有进行链接操作后，才会生成真正可以执行的可执行程序。本节将向读者简要介绍链拉过程的原理和过程。

5.2.1 链接的过程

链接就是把目标代码中的各种符号引用和符号定义转换为可执行文件中的信息的过程。为更好地理解编译的过程，举个简单的例子向读者加以说明。假设某个用户编写了如下程序。

```
#include <stdio.h>
main()
{
    func1();
    printf("OK\n");
    return;
}
void func1()
{
    printf("In func1\n");
}
```

经过编译后，将会生成一个二进制的目标文件。这个目标文件中调用了自定义的函数 fun1，同时还调用了函数 printf。对于函数 fun1 的执行，用户应该可以理解，直接调用用户编写的代码函数就可以。而对于函数 printf，其函数声明位于头文件 stdio.h 中，其目标代码在什么地方呢？答案是 libc.a。这是一个系统的函数库，用户平时调用的大量函数实际上都是以函数库的形式存储在指定的系统目录下（/usr/lib、/lib 等）。链接操作最重要的步骤就是将这些函数库中相应函数的代码组合到目标文件中。

除上述组合目标代码的工作外，链接程序还做了若干其他操作，如重定位等。链接过程如图 5-2 所示。链接过程可以分为静态链接和动态链接，在下一小节中将向读者进行介绍。

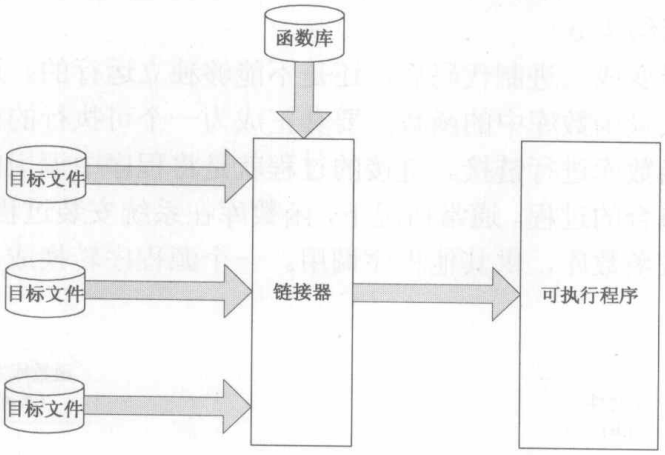


图 5-2　链接过程

5.2.2　静态链接与动态链接

所谓静态链接，就是把各种函数库中包含的目标代码静态添加到可执行文件中。而动态链接则是将这些函数库的路径、函数名等信息添加到可执行文件中，在执行过程中动态加载相应函数库。

通常情况下，静态链接生成的可执行文件较大，而动态链接生成的可执行文件较小。静态链接的优点是可以独立运行，不需要额外动态库的支持；缺点是生成的文件较大，一旦需要程序升级则需要更新整个目标程序。动态库的优点则在于生成的可执行代码较小，程序升级时较为灵活。如下所示，a.c 是源程序，a1 是通过动态链接生成的可执行程序，a2 是通过动态链接生成的可执行程序。

```
-rw-r--r--   1 abc   users     51   Nov 11 17:41 a.c
-rwxr-xr-x   1 abc   users 2968   Nov 11 18:05 a1*
-rwxr-xr-x   1 abc   users 8679   Nov 11 18:05 a2*
```

5.3　gcc 编译器简述

gcc 是 GNU C Compiler 的缩写，是 Linux 下一款功能强大，性能突出的编译器。它支持多种硬件平台，并且编译出的可执行代码执行效率较高。缺省状态下，部分 Linux 操作系统

未安装 gcc 编译器，此时，需要从安装介质上查找 gcc 安装包并安装。

5.3.1 程序的编译与链接

读者平时所执行的各种 Linux 命令，实际上是一个可执行程序。那么，如何从一个文本格式的源程序转换成一个二进制格式的可执行文件呢？这需要经过程序的编译与链接的过程。所谓编译，就是将源程序转换成为计算机可以执行的二进制代码，也称为目标文件。在 Windows 系统下，这些二进制代码称为 obj 文件（或者 dll 文件、lib 文件）。在 Linux 系统下，目标文件的缺省后缀为.o。

一个程序经过编译变成二进制代码后，还是不能够独立运行的。这是由于这些生成的二进制代码中，引用了大量函数库中的函数。要真正成为一个可执行的程序，还必须要将编译出的目标文件与这些函数库进行链接。链接的过程就是将程序中调用的函数库的目标代码与程序的目标文件进行整合的过程。通常情况下，函数库在系统安装过程中自动安装到系统中，也可以由用户自己生成函数库，供其他程序调用。一个源程序转换成为可执行程序的过程如图 5-3 所示。

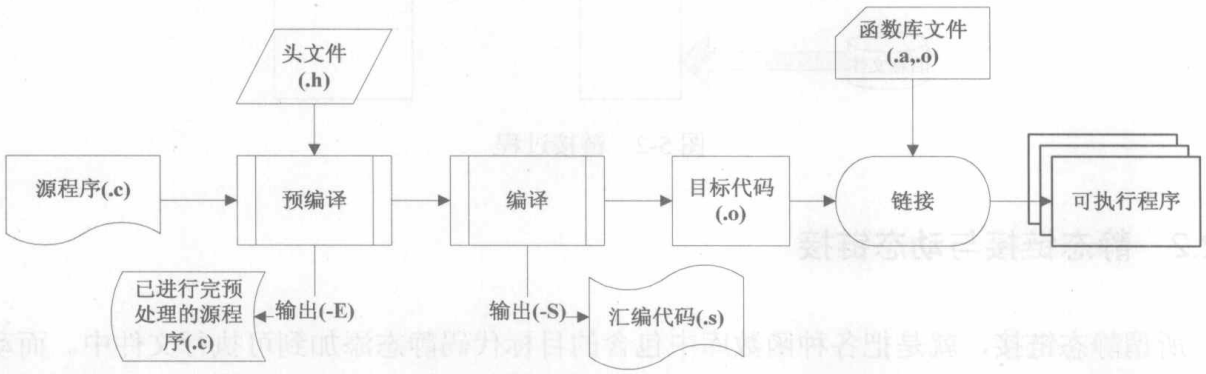


图 5-3 编译链接的过程

5.3.2 gcc 编译器的工作过程

使用 gcc 编译器由源代码生成可执行文件的过程包括以下 3 个相互关联的步骤。

（1）预处理（Preprocessing）。预处理的过程也称为预编译，主要功能是对各种预处理命令进行处理，如头文件的包含（include）、宏定义的扩展（define）以及条件编译（ifdef...endif）等。

（2）编译（Compilation& Assembly）。编译的过程是由源代码翻译成为机器语言的过程。首先进行源程序的语法分析、词法分析，然后将源程序转换为汇编语言代码（在编译时选择-S 选项，可以看到生成的汇编代码在.s 文件中）。再由汇编程序将汇编代码（扩展名.s）转换为目标代码（扩展名.o）。在这个转换过程中，可以根据不同的硬件平台对生成的汇编代码进行优化（在 gcc 编译器中通过-O 选项实现优化）。经过编译后，将产生二进制的目标文件（扩展名为.o）。

（3）链接（Linking）。编译完成后，由链接程序将目标文件与各库函数进行链接并重定位。至此，一个源程序变为真正意义上的可执行程序。链接可以分为静态链接和动态链接，在下一节中将进行详细讲解。

提示：在 gcc 编译器环境下，常见文件扩展名的含义如下。

- .c：C 源程序。
- .h：头文件。
- .s：汇编程序代码。
- .o：目标代码。
- .a：函数库。

5.4　gcc 编译器语法

gcc 编译器不仅支持对 C 语言的编译链接，而且支持面向对象的 C++语言的编译。因此，gcc 的语法非常复杂，受篇幅所限，在本节中将重点介绍其常用的功能。

5.4.1　常用语法

本小节将首先详细介绍 gcc 编译器的常见语法，具体如下。

- -c：只进行编译，而不进行链接。-c 选项输出的是与源文件同名的.o 后缀的文件。
- -S：输出汇编代码文件。在编译时如果指定该选项，将生成一个与源文件同名的.s 后缀的文件，文件中包含的是生成的汇编代码。
- -o：链接生成可执行文件。该选项可以附加生成的可执行文件名称。如果未指定生成的可执行文件名称将用缺省名 a.out 代替。
- -g：产生符号调试工具（gdb）所必要的符号信息。要想对源代码进行调试，需要在编译时加入这个选项。
- -O：在编译链接过程中进行优化。gcc 编译器将根据当前的硬件平台等信息在编译、链接过程中进行优化处理，产生的可执行文件的执行效率可以得到提高。
- -Idirname：将 dirname 所指的目录加入到“头文件搜索目录列表”中，在预编译过程中使用该参数。在 C 源程序中，可以使用两种方式对头文件进行包含，一种是用“#include <>”，另外一种是用“#include" "”。对于第一种情况，预编译程序将会在系统缺省的头文件目录（一般为/usr/include）中进行搜索。而对于第二种情况，预编译程序将首先在当前目录中进行搜索，如果未找到相应的头文件，则根据“头文件搜索目录列表”中指定的目录逐一进行搜索，直至发现为止。
- -Ldirname：将 dirname 所指出的目录加入到“函数库搜索目录列表”中。与预编译过程需要寻找头文件类似，在链接过程中需要寻找相应的库文件（后缀.a）。
- -lname：在连接过程中加载名字为“libname.a”的函数库。该函数库位于系统预设的目录或者由-L 选项确定的目录下。例如，-lc 表示连接名为“libc.a”的函数库。

如果要加载系统函数库（在/lib 及/usr/lib 等目录下的.a），是不需要定义-L 的。如果要加载自定义的函数库，则必须要先通过-L 选项指定，然后才可以用-l 选项加载。

提示：在加载函数库时，要注意-l 选项后面的参数与库文件名称的联系。库函数名称为 libxxx.a，则-l 选项后面应该使用 xxx。

5.4.2 用 gcc 编译器生成可执行文件

要使用 gcc 编译器生成一个可执行文件，需要经过编译、链接的过程。编译过程通常编译的脚本如下所示。

```
gcc –c –O  -I dirname x.c
```

其中-c 选项表明只是进行编译，而不进行链接。

- -O：表示需要进行优化。
- -I：指定程序 x.c 中用到的头文件所在的目录。如果只使用了系统头文件（/usr/include 中的头文件），则不必指定该选项。

编译成功后，将生成与源程序同名的扩展名为.o 的目标文件。

链接过程通常的链接脚本如下所示。

```
gcc –L dirname –lyyy x.o –o x
```

- -L：指定了链接时用到的库文件所在的目录。如果程序中只使用了系统库文件（/usr/lib 和/lib 下的.a 文件），则不必指定该选项。
- -lyyy：指定要链接函数库 libyyy.a。
- -o x：指定了链接后生成的可执行文件名称。如果未指定该文件名，则将生成到文件 a.out 中。

5.4.3 用 gcc 编译器生成动态链接库

在 Linux 系统下，函数库分为两种不同的类型：静态函数库和动态链接库。静态函数库是指当应用程序的目标代码与函数库进行链接时，函数库中被引用的代码将被复制到最终生成的可执行文件。与静态函数库链接后生成的可执行文件，是可以脱离函数库直接运行的。动态链接库则不然，它在内存只包含一份可执行的映像。所有链接到动态链接库的应用程序都共享这一映像。动态链接库与静态函数库的区别如图 5-4 所示。

使用动态链接库不仅可以节省文件系统及内存的空间，而且它还提高了系统的安全性。在应用动态链接库到一些较大的应用程序中时，可以使用应用程序的加载时间。在 Linux 系统下，生成动态链接库的过程如下。

（1）编译。其语法如下所示。

```
gcc -fPIC -c xxx.c
```

-fPIC 选项的含义是通知 gcc 编译器产生可以重定位的目标代码。对于需要被动态链接的函数库，这一点尤其重要。

（2）链接。其语法如下所示。

```
gcc -shared -o libxxx.so xxx.o
```

经过编译链接，即可以生成动态链接库，其扩展名为.so。执行 file 命令查看文件类型时，可以看到类似“ELF 32-bit LSB shared object”的信息。

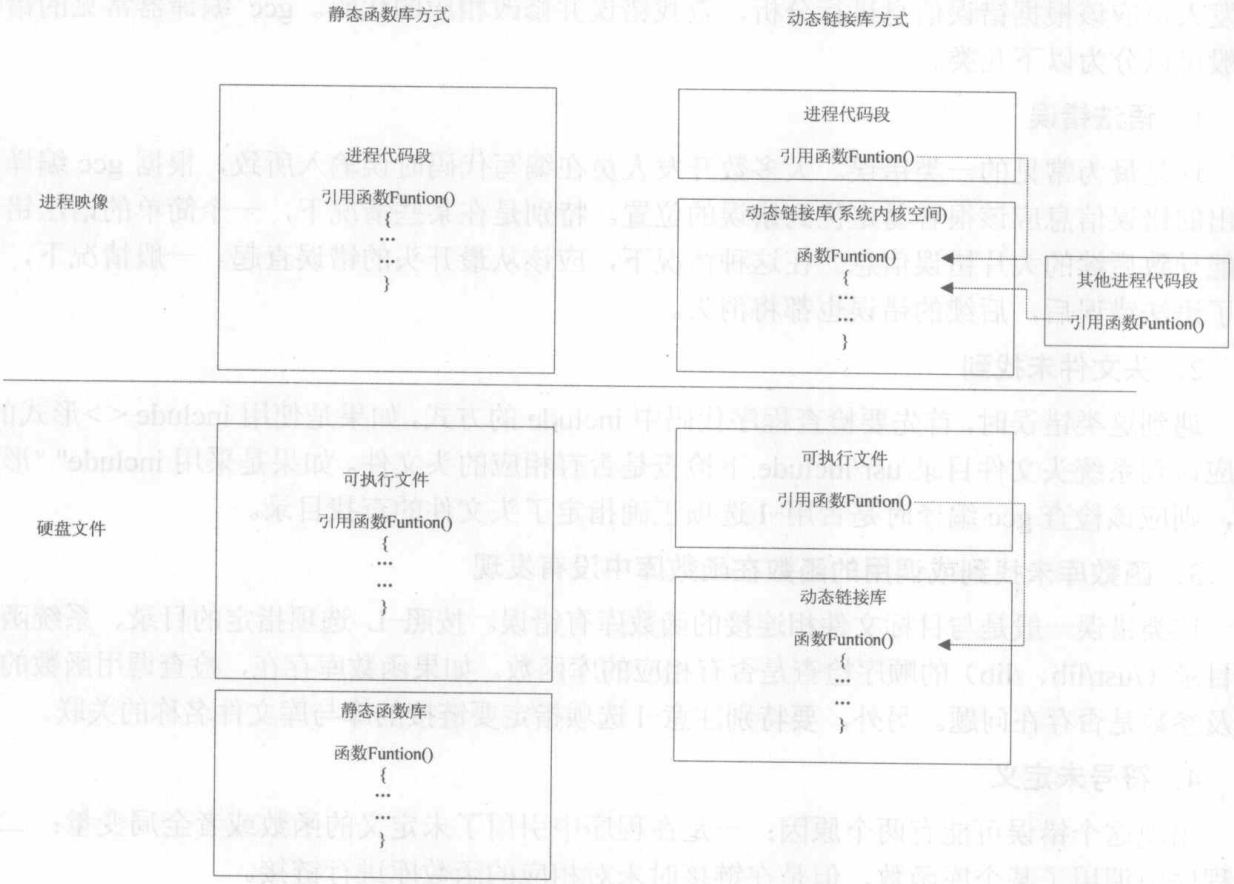


图 5-4　静态函数库与动态链接库

5.4.4　如何使用动态链接

在上一节中已经向读者介绍了使用动态链接的好处。那么，如向在应用程序中引用生成的动态链接库呢？要使用系统已有的动态链接库，可以有两种方式：一是通过设置环境变量 LD_LIBRARY_PATH 的方式；二是通过 dlopen 系列系统调用动态打开的方式。

通过设置环境变量的方式使用起来比较简单，只需要设置环境变量 LD_LIBRARY_PATH 为动态链接库文件所在的目录即可，如 export LD_LIBRARY_PATH=/home/sharedlib 等。也可以在.profile 文件中进行设置以保证长期有效。一旦设置了该变量，在程序执行的过程中，将根据 LD_LIBRARY_PATH 指定的路径进行查找并进行动态链接。

dlopen 系列系统调用主要包括 dlopen、dlsym、dlclose 等函数。其中 dlopen 函数用于打开一个动态库，可以在函数中直接指定动态库的路径名，dlsym 函数用于将动态库中的函数映射到本地函数指针。经过映射后，就可以在本地应用程序中调用动态库中的函数完成相应

功能；最后，调用 dlclose 完成动态库的卸载。

5.4.5 gcc 编译器常见错误排除

在编译链接的过程中，如果发现错误，gcc 将给出错误信息并中止编译链接过程。程序开发人员应该根据错误信息进行分析，查找错误并修改相应的代码。gcc 编译器常见的错误一般可以分为以下几类。

1．语法错误

这是最为常见的一类错误。大多数开发人员在编写代码时误输入所致，根据 gcc 编译器给出的错误信息应该很容易定位到错误的位置。特别是在某些情况下，一个简单的语法错误可能导致后续的大片错误信息。在这种情况下，应该从最开头的错误查起。一般情况下，修改了语法错误后，后续的错误也都将消失。

2．头文件未找到

遇到这类错误时，首先要检查程序代码中 include 的方式，如果是使用 include < >形式的，则应该到系统头文件目录/usr/include 下检查是否有相应的头文件。如果是采用 include" "形式的，则应该检查 gcc 编译时是否用-I 选项正确指定了头文件的查找目录。

3．函数库未找到或调用的函数在函数库中没有发现

这类错误一般是与目标文件相连接的函数库有错误。按照-L 选项指定的目录、系统函数库目录（/usr/lib、/lib）的顺序检查是否有相应的库函数。如果函数库存在，检查调用函数的名称及参数是否存在问题。另外，要特别注意-l 选项指定要链接的库与库文件名称的关联。

4．符号未定义

出现这个错误可能有两个原因：一是在程序中引用了未定义的函数或者全局变量；二是在程序中调用了某个库函数，但是在链接时未对相应的函数库进行链接。

5.5 小结

编译、链接操作是在 Linux 系统下进行编程时必需的步骤。而 gcc 编译器是 Linux 系统下进行程序开发时必不可少的工具。gcc 编译器功能非常强大，在本书中只是就常见的功能进行了讲解。其中，重点分析和讲解了 gcc 编译器的语法功能，读者可以通过这部分内容了解 gcc 编译器的使用方法。同时，对其 gcc 编译器的其他功能可以在编程过程中根据实际需要灵活加以运用。

第 6 章　make 的使用

在 Linux 系统下，make 是一个非常重要的编译工具。利用 make 工具，用户可以将大型的开发项目分解成为多个更易于管理的模块，对于一个包括几百个源文件的应用程序，使用 make 及 makefile 工具就可以简洁明快地理顺各个源文件之间纷繁复杂的相互关系。因此，有效地利用 make 及 makefile 工具可以大大提高项目开发的效率。本章重点介绍 make 的使用规则，通过本章的学习，读者应重点掌握如下内容。

- 了解 make 命令的工作原理。
- 学会 makefile 的编写规则。
- 掌握 make 的基本命令选项。

6.1　makefile 介绍

make 工具最主要也是最基本的功能就是通过 makefile 文件来描述源程序之间的相互关系并自动维护编译工作。而 makefile 文件需要按照某种语法进行编写，文件中需要说明如何编译各个源文件并连接生成可执行文件，并要求定义源文件之间的依赖关系。在 Linux 系统下，默认使用 makefile 作为 makefile 文件。如果要使用其他文件为 makefile，则可在使用 make 时指定–f 选项使用不同的 makefile 文件。makefile 文件作为一种描述文档，一般需要包含以下内容。

1．显式规则

显式规则定义了如何生成一个或者多个的目标文件，即显式地指出要生成的文件、文件的依赖项、生成的命令等。

2．隐含规则

在 make 工具中包含有一些内置的或隐含的规则，这些规则定义了如何从不同的依赖文件建立特定类型的目标文件。Linux 系统通常支持一种基于文件扩展名即文件名后缀的隐含规则。这种后缀规则定义了如何将一个具有特定文件名后缀的文件（如.c 文件），转换成为具有另一种文件名后缀的文件（如.o 文件）。

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

系统中默认的常用文件扩展名及其含义。

- .o：目标文件。
- .c：C 源文件。
- .s：汇编源文件。
- .y：Yacc-C 源语法。

- .l：Lex 源语法。

3．宏（变量）定义

makefile 中允许使用简单的宏指代源文件及其相关编译信息，在 Linux 中也称宏为变量。在引用宏时只需在变量前加$符号，但值得注意的是，如果变量名的长度超过一个字符，在引用时就必须加圆括号()。

4．注释

与 UNIX Shell 类似，在 makefile 中只有行注释，在行首使用“#”字符即表示本行内容为注释。如果要在 makefile 中使用“#”，则可使用反斜杠转义“\”。

6.2 makefile 的书写规则

上一小节已经简单介绍了 makefile 的内容，本小节将介绍 makefile 的详细编写规则，其主要内容包括语法规则以及如何使用其他语法元素。

6.2.1 基本语法规则

makefile 的基本语法规则如下所示。

```
Targets:Prerequisites
Command
```

其中各元素的含义如下。

- Targets：文件名，以空格分开，可以使用通配符。一般来说，我们的目标基本上是一个文件，但也有可能是多个文件。
- Prerequisites：目标所依赖的文件（或依赖目标）。如果其中的某个文件要比目标文件新，那么，目标就被认为是“过时的”，被认为是需要重新生成的。
- Command：命令行，如果其不与“Targets:Prerequisites”在一行，那么，必须以<Tab>键开头，如果和 Prerequisites 在一行，那么可以用分号作为分隔。Command 如果命令太长，可以使用反斜框（\）作为换行符。

提示：需要注意的是，所有的 makefile 中的命令行都应以<Tab>键开始。

6.2.2 定义变量

在 makefile 中定义的变量，就像是 C 语言中的宏一样，它代表了一个文本字串，在 makefile 中执行的时候会自动原模原样地展开其在所使用的位置。与 C 语言所不同的是，用户可以在 makefile 中改变其值。在 makefile 中，变量可以使用在“目标”、“依赖目标”、“命令”或是 makefile 的其他部分中。

变量的命名字可以包含字符、数字、下画线（可以是数字开头），但不应该含有“:”、

“#”、“=”或是空字符（<Space>键、<Enter>键等）。变量的大小写敏感的，“abis”、“Abis”和“ABIS”是 3 个不同的变量名。传统的 makefile 的变量名是全大写的命名方式。

变量在声明时需要给予初值，而在使用时，需要给在变量名前加上“$”符号，但最好用小括号“()”或是大括号“{}”把变量给包括起来。如果要使用真实的“$”字符，那么需要用“$$”来表示。

提示：需要注意的是一些自动化（或者叫预定义）变量。在 Linux 系统下，自动化变量包括$@、$?、$<、$(@D) 等。$@：表示规则中的目标文件集；$+：所有的依赖文件，以空格分开，并以出现的先后为序，可能包含重复的依赖文件；$^：所有的依赖文件，以空格分开，不包含重复的依赖文件；$?：所有比目标新的依赖目标的集合；$<：第一个依赖文件的名称；$*：不包含扩展名的目标文件名称。

6.2.3　环境变量

make 工具支持环境变量，通过设置环境变量的值，可以改变 make 的默认动作。下面详细介绍各种常见的工程变量的内容。

1．VPATH 环境变量

在一些大的工程中，有大量的源文件，我们通常的做法是把许多的源文件分类，并存放在不同的目录中。所以，当 make 需要去寻找文件的依赖关系时，可以在文件前加上路径，但最好的方法是把一个路径告诉 make，让 make 在自动去找。

makefile 文件中的特殊变量“VPATH”就是完成这个功能的，如果没有指明这个变量，make 只会在当前的目录中去找寻依赖文件和目标文件。如果定义了这个变量，那么 make 就会在找不到当前目录的情况下，到所指定的目录中去找寻文件了。例如：

```
VPATH = /home/abis/src: ../headers
```

上面的定义指定“/home/abis/src”和“../headers”两个目录，make 会按照这个顺序进行搜索。如果存在多个目录，则目录间由“冒号”分隔。

2．CFLAGS 环境变量

指定了 C 语言编译器参数。如果指定了该环境变量，则在编译时将引用这些参数。

3．LDFLAGS 环境变量

指定了 C 语言链接器参数。如果指定了该环境变量，则在链接时将引用这些参数。

6.2.4　通配符的使用

如果用户想定义一系列比较类似的文件，可以使用通配符。make 支持 3 个通配符：“*”、“?”和“[...]”。这和 UNIX 的 B-Shell 是相同的。波浪号（～）字符在文件名中也有比较特殊的用途。如果是“～/test”，这就表示当前用户的$HOME 目录下的 test 目录。

6.2.5 使用条件判断

使用条件判断，可以让 make 根据运行时的不同情况选择不同的执行分支。条件表达式可以是比较变量的值，或是比较变量和常量的值。条件表达式的语法如下所示。

```
<conditional-directive>
<text-if-true>
endif
```

以及：

```
<conditional-directive>
<text-if-true>
else
<text-if-false>
endif
```

<conditional-directive>代表条件关键字，共有 4 个：ifeq、ifneq、ifdef、ifndef。

ifeq 的语法如下所示。

```
ifeq (<arg1>, <arg2>)
ifeq '<arg1>' '<arg2>'
ifeq "<arg1>" "<arg2>"
ifeq "<arg1>" '<arg2>'
ifeq '<arg1>' "<arg2>"
```

ifeq 用于比较参数 arg1 和 arg2 的值是否相同。如果相同则返回真，否则返回假。

ifneq 的语法如下所示。

```
ifneq (<arg1>, <arg2>)
ifneq '<arg1>' '<arg2>'
ifneq "<arg1>" "<arg2>"
ifneq "<arg1>" '<arg2>'
ifneq '<arg1>' "<arg2>"
```

ifneq 用于比较参数 arg1 和 arg2 是否不同，如果不同则返回真，否则返回假。

ifdef 的语法如下所示。

```
ifdef <variable-name>
```

如果变量<variable-name>的值非空，那么表达式为真；否则，表达式为假。

ifndef 的语法如下所示。

```
ifndef <variable-name>
```

如果变量<variable-name>的值为空，那么表达式为真；否则，表达式为假。

提示：使用条件判断时，不应该加前缀<Tab>键，否则，将被解释为命令。

6.2.6 在 makefile 中使用函数

makefile 支持函数调用。通过使用函数，可以使 makefile 的使用更为灵活。函数调用后的返回值可以在 makefile 中获取，用户可以根据返回值进行下一步的处理。在 makefile 中使用函数的语法如下所示。

```
$(<函数名称> <函数参数 1> <函数参数 2> <...>)
```

其中，函数名称定义了要调用的函数名，后面的函数参数则是传递给被调用函数的参数，参数的个数可以是多个。函数名称及函数参数间用空格进行分隔。常用的 makefile 函数包括以下几点。

1．字符串替换函数 subst

语法格式如下所示。

```
$(substr <from>,<to>,<text>)
```

该函数实现的功能是将字符串 text 中 from 的内容替换为 to。

2．去空格函数 strip

语法格式如下所示。

```
$(strip <string>)
```

该函数实现的功能是将字符串 string 中的前后空格去除。

3．查找字符串函数 findstring

语法格式如下所示。

```
$(findstring <find>,<in>)
```

该函数实现的功能是在字符串 in 中查找与 find 指定内容相匹配的子串。如果查找成功，则返回 find 指定的串，否则返回空串。

4．排序函数 sort

语法格式如下所示。

```
$(sort <list>)
```

其中，list 是待排序的字符串集合；返回值是经过排序后的字符串集合。需要注意的是：sort 函数将对重复的串进行过滤，只保留一次。

5．取目录函数 dir

语法格式如下所示。

```
$(dir <路径名>)
```

该函数实现的功能是从路径名中取出目录部分并返回。

6．取前缀函数 basename

语法格式如下所示。

```
$(basename <文件名>)
```

该函数实现的功能是从文件名中取出，去掉后缀（.及后面的部分）后的内容并返回。

7．变量来源函数 orgin

语法格式如下所示。

$(orgin <变量名>)

该函数的作用返回变量的来源。可能的来源包括以下几点。

- Undefined：变量未定义过。
- Default：默认的变量定义。

- File：在 makefile 中定义的变量。
- Command line：在命令行中定义的变量，如在 make 的命令行中定义的变量等。

6.2.7 使用 make 与直接使用 gcc 脚本的区别

make 可以定义项目中不同源文件之间的依赖关系，这种依赖关系在多源文件的程序编译中尤其重要。通过这种依赖关系的定义，make 工具可避免许多不必要的编译工作。虽然利用 Shell 脚本也可以达到自动编译的效果，但是 gcc 脚本将全部编译任何源文件，包括那些不必要重新编译的源文件，而 make 工具则可根据目标上一次编译的时间和目标所依赖的源文件的更新时间，而自动判断应当编译哪个源文件。

【范例 6-1】一个名为 prog 的程序由两个 C 源文件 filea.c、fileb.c 以及库文件 libtest.a 编译生成。库文件位于当前目录下。这两个源文件还分别包含自己的头文件 a.h 、b.h。假设 filea.c 要用到一个名为 defs.h 的头文件，但 fileb.c 不用。编写 makefile 实现对该程序的编译链接。其实现过程如示例代码 6-1 所示。

示例代码 6-1

```
01  prog : filea.o fileb.o                  #prog 依赖于 filea.o fileb.o
02  cc filea.o fileb.o –L./ -ltest -o prog  #对于 prog 目标进行的动作
03  filea.o : filea.c a.h defs.h            #filea.o 依赖于 filea.c a.h defs.h
04  cc -c filea.c                           #对于 filea.o 目标进行的动作
05  fileb.o : fileb.c b.h                   #fileb.o 依赖于 fileb.c b.h
06  cc -c fileb.c                           #对于 fileb.o 目标进行的动作
```

6.3 make 工具

makefile 编写完成后，就可以通过执行 make 命令进行编译操作。在本小节中，将详细介绍 make 工具的详细使用方法。

6.3.1 运行 make 命令

要执行 make，只需要在命令行下输入 make 即可。make 可以包含若干选项，主要的选项及其功能描述如下。

- -b 、-m：忽略和其他版本 make 的兼容性。
- -B、--always-make：强制重新编译所有的目标。
- -C <dir>、--directory=<dir>：指定读取 makefile 的目录。
- -debug[=<options>]：输出 make 的调试信息。它有几种不同的级别可供选择，如果没有参数，那就是输出最简单的调试信息。

其中，<options>的取值如下。

- a：all，输出所有的调试信息。

- b：basic，只输出简单的调试信息，即输出不需要重新编译的目标。
- v：verbose，在 b 选项的级别之上。输出的信息包括哪个 makefile 被解析、不需要被重新编译的依赖文件（或是依赖目标）等。
- i：implicit，输出所有的隐含规则。
- j：jobs，输出执行规则中命令的详细信息，如命令的进程标识、返回码等。
- m：makefile，输出 make 读取 makefile、更新 makefile、执行 makefile 的信息。

6.3.2　make 命令的工作过程

make 命令的工作是按照一定规则进行的。通常情况下，一个 make 命令的工作过程如下。

（1）读入 makefile，make 是在当前目录下查找 makefile 或者 Makefile 的。如果用户通过-f 指定其他 makefile 名称，则将按照用户指定的文件去查找。

（2）初始化 makefile 文件中的变量，如果 makefile 包含了其他的 makefile 文件，则加载之。

（3）推导隐含规则，并分析所有规则，检查各种要素间的依赖关系，为所有的目标文件创建依赖关系链。

（4）根据依赖关系，检查哪些依赖项发生了变更，分析哪些文件需要重新生成。

（5）执行生成命令。

6.3.3　在 makefile 中使用伪目标

每个 makefile 都会定义目标。所谓目标，就是 make 操作要生成的对象。对于程序设计过程中使用的 makefile 来说，这个目标可以理解为要生成的目标文件。通常情况下，一个 makefile 会存在一个最终目标，如要生成某个可执行文件。其他的目标都是为生成这个最终目标服务的，如生成一个目标文件或者函数库文件。

多数情况下，一个 makefile 中可能会指定多个目标，如果在 make 时指定要生成某个目标，则 make 将会按照要求单独生成某个目标。按照 GNU 规范的要求，允许在使用 make 的过程中在 makefile 中定义伪目标。这些伪目标并不是强制性的，而是建议性的。常用的伪目标有以下几种。

- all：这个伪目标是所有目标的目标，其功能一般是编译所有的目标。
- clean：这个伪目标的功能是删除所有被 make 创建的文件。
- install：这个伪目标的功能是安装已编译好的程序，其实就是把目标执行文件复制到指定的目标中去。
- print：这个伪目标的功能是输出改变过的源文件。
- tar：这个伪目标的功能是把源程序打包备份，也就是一个 tar 文件。
- dist：这个伪目标的功能是创建一个压缩文件，一般是把 tar 文件压成 Z 文件，或是 gz 文件。
- TAGS：这个伪目标的功能是更新所有的目标，以备完整地重编译使用。

- check 和 test：这两个伪目标的功能是测试 makefile 的流程。

6.3.4 make 命令的返回值

make 命令执行完成后，可以根据 make 的返回码查询 make 运行的结果。查看该返回码的方式就是用 shell 执行命令 echo $?。make 返回码有以下 3 种。

- 0：表示成功执行。
- 1：如果 make 运行时出现任何错误，返回 1。
- 2：如果使用了 make 的-q 选项，并且 make 使得部分目标不需要更新，则返回 2。

6.4 综合实例

为了更好地掌握 makefile 的使用方法，本节将结合一个具体的 makefile 例子进行讲解。本例介绍的是一个实际开发应用项目的 makefile 文件，具有一定代表性。希望通过本节的介绍，能够使读者对于 makefile 的使用有一个初步的了解。

6.4.1 makefile 应用的环境

本节的示例程序是一个可独立运行的可执行程序，共有两个源程序，分别是 xywjk.c 和 jktools.c。另外，需要连接数据库提供的接口函数库 libsybdb.a 和用户自行编写的目标函数文件 instlogup.o。详细环境如下所示。

```
数据库系统：Sybase 数据库
编程语言：C 语言
头文件：sybfront.h、sybdb.h
库文件：libsybdb.a instlogup.o
```

6.4.2 makefile 的实现及解释

根据上述环境，结合 makefile 的编写规则，实现的 makefile 如下所示。

```
1   # Make system openserver version 1.0 programs.                    /*注释信息*/
2   #
3   # makefile 1.0 2008.06.03
4   #
5   # Change the following definitions as appropriate for your site:
6   SYBASE      = /home/Sybase                                        /*定义变量 SYBASE */
7   INCDIR      = $(SYBASE)/include                                   /*定义变量 INCDIR*/
8   LIBDIR      = $(SYBASE)/lib                                       /*定义变量 LIBDIR*/
9   HEADERS     = $(INCDIR)/sybfront.h $(INCDIR)/sybdb.h              /*定义变量 HEADERS*/
10  DBLIBS = $(LIBDIR)/libsybdb.a                                     /*定义变量 DBLIBS*/
11  INCLUDE= -I$(INCDIR)                                              /*定义变量 INCLUDE*/
```

```
12  CC=cc                                                    /*定义变量 CC*/
13  OBJ=xywjk.o jktools.o instlogup.o                        /*定义伪目标 OBJ，该伪目
    标依赖于 xywjk.o、jktools.o 和 instlogup.o*/
14
15  xywjk:$(OBJ)                                             /*本 makefile 的最终目标，
    要生成名为 xywjk 的可执行程序，xywjk 依赖于 OBJ 变量所指的内容*/
16  cc $(OBJ) $(DBLIBS)  -ltinfo -lnsl -lsocket  -o xywjk    /*生成 xywjk 所执行的操
    作，可以看到，该操作实际是链接操作*/
17  $(OBJ):$(HEADERS)                    /*OBJ 中所包含的目标依赖于变量 HEADERS 所指的
    内容，实际就是头文件*/
18  .c.o:                                /*定义由.c 生成.o 的生成规则*/
19  cc -c -g $(INCLUDE) $*.c             /*由.c 生成.o 所执行的操作，实际上是编译操作*/
```

在本项目中，用户要生成的是可执行的目标程序 xywjk，该目标依赖于 OBJ 变量所指内容（第 15 行），即 xywjk.o、jktools.o、instlogup.o 三个目标文件，所要执行的操作是链接操作（第 16 行）。对于 OBJ 变量中所包含的项目又要如何生成呢？从第 17 行可以看到，OBJ 中包含的项目依赖于 HEADERS 变量的内容，就是各种头文件。OBJ 中包含的项目的扩展名为.o，第 18～19 行定义了由. C 源程序生成.o 文件的方法，就是调用 cc 进行编译操作（第 19 行）。

6.5　小结

make 为程序开发人员提供了更为方便的编译工具。在开发过程中，用户可以根据自己的需要决定在命令行中使用 gcc 还是使用 make。如果是较小的项目，直接使用 gcc 也未尝不可。但是对于较大的项目，建议使用 make 工具完成编译链接工作。本章主要介绍了 makefile 的语法规则和常见的工具，通过这些内容，用户可以了解 Linux 环境下 makefile 的使用方法。同时，本章最后结合一个综合案例，具体分析了 makefile 的使用方法。

第7章 程序调试

在编程的过程中，不可避免地存在这样或者那样的错误，此时需要对程序进行调试。Linux系统提供了gdb工具进行程序调试，本章主要介绍使用gdb进行程序调试的基本方法。通过本章的学习，读者应重点掌握如下内容。

- 了解Linux环境下编程的错误处理方法。
- 掌握使用gdb调试程序的基本方法。

7.1 错误处理

有过Windows系统下可视化程序设计环境经验的读者可能知道，在Windows系统下的集成开发环境，提供了若干自动化的错误处理机制，如try...catch机制、断点跟踪机制等。在Linux系统下，没有这些智能化的异常处理手段。要进行错误处理，大多需要通过手工编码的方式进行。本节将向读者介绍Linux系统下错误处理的方法。

7.1.1 使用标准错误输出

在Linux系统的编程环境下，每个运行的进程默认都将打开3个文件描述符：标准输入、标准输出及标准错误输出。其中标准错误输出就是用于向标准错误输出设备输出错误信息的。标准错误输出文件的文件描述符是2，在标准I/O库中，它被定义为stderr。

在默认状态下，标准错误输出的内容被输出至屏幕。也就是说，在程序中向标准错误输出写入的内容将显示在屏幕上。利用这一特性，编程人员可以在程序的不同地方输出所需要的调试信息或者错误提示代码如下所示。

```
#include <stdio.h>
main()
{
    char ch;
    scanf("%c",&ch);
    if(isdigit(ch))
    {
        fprintf(stderr,"Please input a digit char!\n");
    }
}
```

在上述程序中，要求用户输入一个数字类型的字符，即从0至9的任意数字。如果用户输入的不是数字，则显示错误提示信息。在实际编程应用过程中，这一手段被大量使用。读者在编程过程中，也可以灵活应用标准错误输出这一机制，在适当的位置显示错误提示信息，甚至是调试信息。

7.1.2　使用 errno 全局变量

在 Linux 编程模式中，系统预先定义了全局变量 errno。每当调用函数或者系统调用失败时，由系统内核将一个整形的错误代码保存至 errno。编程人员在调用函数失败后，可以从 errno 变量的值去获取错误的详细信息。这个 errno 变量的声明位于头文件 errno.h 中。为了更方便地查看错误的详细信息，Linux 系统提供了两个格式化错误信息的函数：perror 和 strerror。perror 函数的声明位于头文件 stdio.h 中，其原型如下所示。

```
#include <stdio.h>
void perror(__const char *__s);
```

其中，__s 是一个指向字符串的指针。调用该函数后，将向标准错误输出设备输出如下信息“s:错误的详细信息”。

同样，strerror 函数也可以对指定的错误代码进行格式化处理，产生错误的详细描述。不过，strerror 并不直接将错误信息输出，而是返回指向错误信息的字符串指针。用户可以根据需要对错误信息进行进一步的处理。在需要对错误信息进行进一步处理、而不是简单的输出的情况下，strerror 函数更加适用。strerror 函数的原型如下所示：

```
#include <string.h>
char *strerror(int __errnum);
```

其中，__errnum 是错误的值，可以直接使用 errno。调用该函数后，将返回__errnum 对应的详细错误信息。

【范例 7-1】调用标准 I/O 库函数 fopen 以只读方式打开当前目录下的文件“test.txt”。其实现过程如示例代码 7-1 所示。

示例代码 7-1

```
1   #include <stdio.h>                              /*头文件*/
2   main()                                          /*主函数*/
3   {
4       FILE *fp;                                   /*流结构指针变量定义*/
5       If((fp = fopen("./test.txt","r")) == NULL)  /*只读方式打开文件*/
6       {
7           perror("fopen");                        /*打开失败后，提示错误信息*/
8           return;
9       }
10      printf("Open file success!\n");             /*显示打开成功信息*/
11      fclose(fp);                                 /*关闭文件*/
12      return;
13  }
```

【运行结果】经过编译链接，在 shell 下运行上述程序，系统输出如下所示。

```
1   fopen: No such file or directory
```

【代码解析】从输出结果可以看到，由于在当前目录下不存在文件./test.txt，所以程序执行第 7 行的错误输出得到相应的错误信息。在本例中，源程序各行的具体含义如下。

- 第 1 行：头文件信息。stdio.h 是标准输入/输出库的头文件。

- 第 4 行：流结构指针变量定义。当用 fopen 打开一个文件时，返回的就是指向该结构的指针。
- 第 5 行：调用 fopen 以只读方式打开文件./test.txt。
- 第 7 行：如果 fopen 调用失败，则调用 perror 输出当前的错误信息。perror 将根据 errno 变量的值进行格式化，并向标准错误输出（即屏幕）输出错误信息。
- 第 10 行表示：输出打开文件成功的信息。
- 第 11 行表示：关闭已打开的文件。

提示：errrno 中的内容应该在出错后立即使用。因为如果出错后不立即使用，可能在调用其他函数时又产生了错误，导致 errno 的原有内容被覆盖。

7.1.3 使用错误信号处理

使用标准错误输出和 errno 变量进行错误处理，都是在可预知的情况下进行的错误处理。也就是说，编码人员可以预知此处可能发生错误，所以预先编码进行错误处理。除此之外，还有另外一种情况，就是编码人员不知道何时会出现异常，此时就可以使用信号进行错误处理。

例如，某个用户编程实现了一个简单的除法程序。从屏幕输入被除数和除数，计算它们的商后输出。如果用户输入的除数是 0，此时就会产生错误。对于这种情况，一种方法是可以在程序中判断除数是否为 0，如果为 0 则向标准错误输出提示错误信息；另外一种方法就是捕获信息。在被 0 除的情况下，系统内核将向进程发送信息 SIGFPE，编程人员可以在程序中捕获该信息进行处理。

【范例 7-2】编程实现除法。其实现过程如示例代码 7-2 所示。

示例代码 7-2

```
1   #include <stdio.h>                          /*头文件*/
2   #include <signal.h>                         /*头文件*/
3   void Handle_Sigfpe(int signo)               /*信号处理函数*/
4   {
5       fprintf(stderr,"捕获到 SIGFPE 信号！\n"); /*输出提示信息*/
6       exit(0);                                /*退出程序*/
7   }
8   main()                                      /*主函数*/
9   {
10      int i,m,n;                              /*变量定义*/
11      signal(SIGFPE,Handle_Sigfpe);           /*安装信号*/
12      for(i=0;i<10;i++)                       /*循环*/
13      {
14          scanf("%d %d",&m,&n);               /*从键盘输入两个整数*/
15          printf("%d / %d = %d\n",m,n,m/n);   /*求商并输出*/
16      }
17  }
```

【运行结果】经过编译链接，在 shell 下运行上述程序，系统输出如下所示。

```
4 2
4 / 2 = 2
3 0
捕获到 SIGFPE 信号！
```

【代码解析】从输出结果可以看到，在输入数据正常时，程序可以显示正确的输出。而当输入的除数为 0 时，将通过信号处理函数捕获到异常并进行处理。在本例中，源程序各行的具体含义如下。

- 第 1～2 行：头文件信息。stdio.h 是标准输入/输出库的头文件，signal 是信号处理库的头文件。
- 第 3～7 行：SIGFPE 信号处理函数。进程捕获到 SIGFPE 信号时，将执行该函数。
- 第 11 行：安装 SIGFPE 信号处理函数。
- 第 12～16 行：循环从键盘输入被除数和除数，计算它们的商并输出。

7.1.4　使用 assert 断言

Linux 系统提供了 assert 断言供开发人员调用。所谓断言，就是程序设计人员在编程时可以预知相应的状态，如某个条件表达式的值或者某个变量的值，此时可以对该变量或者条件表达式使用断言。一旦该值与所预想的不符，程序将输出错误信息后结束运行。

assert 的工作原理是计算表达式 expression ，如果其值为假，那么它先向标准错误输出打印一条错误信息，然后通过调用 abort 来终止程序运行。assert 宏的原型定义在<assert.h>中，其作用是如果它的条件返回错误，则终止程序执行。该函数的原型定义如下所示。

```
#include <assert.h>
void assert( int expression );
```

【范例 7-3】编程实现文件打开的操作体会断言的使用方法。其实现过程如示例代码 7-3 所示。

示例代码 7-3

```
#include <stdio.h>                                  /*头文件*/
#include <assert.h>
#include <stdlib.h>
int main( void )                                    /*主函数*/
{
    FILE *fp;                                       /*流结构指针变量定义*/
    fp = fopen( "test.txt", "w" );/*以可写的方式打开一个文件，如果不存在就创建一个同名文件*/
    assert( fp );                                   /*断言，这里不会出错*/
    fclose( fp );                                   /*关闭文件*/
    fp = fopen( "noexitfile.txt", "r" );/*以只读的方式打开一个文件，如果不存在就打开文件失败*/
    assert( fp );                                   /*这里出错*/
    fclose( fp );                                   /*该语句将不会执行*/
    return 0;
}
```

【运行结果】经过编译链接，在 shell 下运行上述程序，系统输出如下所示。

```
example7_3: example7_3.c:11: main: Assertion'fp'failed.
已放弃
```

7.1.5 内存泄露的检查

在 Linux 系统下，C 语言程序完全由编程人员自主申请（malloc）和释放内存（free）。如果用户申请了内存而忘记释放，就会在系统中导入内存错误。内存错误往往非常严重，特别是在长时间运行的环境下，可能会带来诸如系统崩溃、内存耗尽等严重的后果。但是，由于内存错误通常具有隐蔽性，它们很难再现，并且内存泄露的症状通常无法直接在源代码中发现。这是由于即使存在内存泄露，可能程序也在相对短的时间范围内运行正常。如下所示是一段存在内在泄露的程序。

```
void func()
{
        char *p;
        p1 = malloc(100);
        sprintf(p,"this is a memory leak program. ");
        fprintf(stderr,"%s\n",p);
        //free(p);
}
```

在上述代码中，程序员通过 malloc 动态分配了 100 字节的内存空间。但在使用完毕后，并没有调用 free 进行释放（见被注释的行）。如此一来，这 100 个字节的内存就被泄露了。该函数只被调用一次或者几次，内存泄露的后果可能并不严重，但是如果该函数被循环调用，那么导致的后果就是内存最终被耗光。

再看另外一个例子，这个例子中没有显式地调用 malloc 分配内存，但是同样存在内存泄露。

```
#include <stdio.h>
void catfile(char *filename)
{
        FILE *fp;
        char buf[1025];
        memset(buf,0x00,sizeof(buf));
        if((fp = fopen(filename,"r")) == NULL)
            return;
        while(fgets(buf,1024,fp) ! = NULL)
        {
            fprintf(stderr,"%s\n",buf);
            memset(buf,0x00,sizeof(buf));
        }
        //fclose(fp);
}
```

在这个例子中，程序员通过 fopen 打开了一个文件并遍历输出其内容。但是在函数的最后，程序员忘记关闭已打开的文件。由于打开文件后在内核会分配内存区用于存储文件信息，所以这时也会存在内存泄露的问题。

内存泄露通常情况下是不易察觉的。因为在短时间内，存在内存泄露的程序运行时不会出现任何异常。只有在长时间运行的过程中，才会发现内存逐渐被消耗干净。为保证用户编写的程序无内存泄露，需要程序员有相应的内存泄露检查机制。

那么如何才能发现内存存在泄露呢？本节向读者介绍两种较为简便的方法。一种是通过 ps 命令查看内存消耗的情况，查看是否存在内存的动态持续增长。另外一种方法是通过代码分析工具对代码进行分析，根据分析报告判断是否存在内存泄露。通过执行 ps 命令查看内存

泄露的方法相对较为简单，直接通过执行 ps –aux 等命令即可。ps 命令的输出如下所示。

```
USER       PID  %CPU  %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
develop   4004   0.0   1.1    6668  2884 pts/1    S    07:49   0:00 su - develop
develop   4005   0.0   0.7    4628  1848 pts/1    S+   07:49   0:00 -bash
postfix   4782   0.0   1.0    5908  2652 ?        S    17:38   0:00 pickup -l -t fifo
root      4832   0.0   0.3    1856   856 ?        Ss   17:45   0:00 in.telnetd: 10.23
root      4833   0.0   0.9    5180  2536 ?        Ss   17:45   0:00 login -- root
```

在上述输出中，VSZ 对应的列就是相应进程占据的内存空间。可以间隔一段时间运行 ps 命令，根据相应进程的内存增长情况进行判断。如果内存动态地、持续地增长，则可以初步断定存在内存泄露问题。

通过代码分析工具进行检查是另外一种进行内存泄露检查的方法。可以用于这方面检查的工具软件有 splint、PC-LINT、BEAM 等，由于这些软件大多具有专用工具，本书在此不作详细介绍，读者有兴趣可以自行下载相关软件试用。

提示：在 Windows 系统下的.Net 编程或者 Java 编程中，实际上是由系统自动进行内存回收的，这样程序员不需要担心内存的泄露问题。而在 Linux 系统下，这一过程是需要手工处理的。

7.1.6　其他可能的内存错误

除上一节介绍过的内存泄露外，还有其他可能的内存错误，主要包括使用未初始化的内存指针和内存溢出。本节将就这两种问题向读者进行介绍。

1．未初始化的指针

所谓未初始化的指针，是指程序员使用指针变量访问内存，而该内存实际上并未分配。为更好地描述这一情况，见下面的示例程序。

```
void func()
{
    char *s;
    sprintf(s,"this is a test program\n");
    return;
}
```

在上述程序中，s 是一个字符串指针变量。在调用 sprintf 时，该指针变量并未分配内存空间，此时，调用 sprintf 将导致进程终止。正确的做法是应该先为 s 分配空间，然后使用，最后再释放分配的空间。正确的程序如下所示。

```
void func()
{
    char *s;
    s = malloc(100);
    sprintf(s, "this is a test program\n");
    …
    free(s);
    return;
}
```

2．内存溢出

内存溢出是指某变量的空间大小是 n 个字节，而在向该变量中写入数据时，写入的数据

超过了该变量的最大空间，此时，就会产生内存溢出。如下所示就是一个内存溢出的例子。

```
void func()
{
    int i,n[9];
    for(i=0;i<10;i++)
        n[i] = i;
}
```

在上述示例程序中，程序员声明了包含 9 个元素的整型数组。在后续的循环中，实际上循环的次数为 10，在最后一次循环中，向数组元素赋值操作的语句实际上为 n[9] = 9;在 C 语言中，数组下标由 0 开始，n[9]表明为第 10 个元素赋值。此时，就将出现内存溢出的问题。

7.2 gdb 介绍

除上一节中介绍过的错误处理机制外，在 Linux 编程环境下另外一个重要的程序调试工具就是 gdb。gdb 是 Linux 系统下进行程序调试的利器，使用户能在程序运行时观察程序的内部结构和内存的使用情况。通过 gdb 可以在程序中设置断点、显示变量的值、单步调试等。

7.2.1 利用 gdb 调试的前提

不是所有的程序都可以使用 gdb 进行调试的。要开始 gdb，必须具备以下几个前提。

- 在程序编译过程中使用-g 选项，以生成 gdb 工作所必需的调试信息。
- 要调试的可执行程序需要有源代码，并且通常源代码应该放在与可执行文件相同的目录下。

7.2.2 启动 gdb 的方法

要启动 gdb 进行调试，可以有多种方法。调试的对象也可以有多种，如可执行文件、core 文件等。本书仅以调试可执行文件为例进行介绍。启动 gdb 可以通过两种方法，一是运行 gdb 可执行文件名启动调试；二是先启动 gdb，再执行 file 命令装入待调试的可执行程序。gdb 启动后的状态如图 7-1 所示。

```
GNU gdb 6.1
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 "i586-suse-linux".
(gdb)
```

图 7-1　启动 gdb

7.2.3　gdb 的基本功能

gdb 提供了多种命令以方便编程人员进行调试。这些命令主要包括以下几点。

- file：装入想要调试的可执行文件。
- kill：终止正在调试的程序。
- list：执行一行源代码但不进入函数内部。
- next：执行一行源代码但不进入函数内部。
- step：执行一行源代码且进入函数内部。
- run：执行当前被调试的程序 。

执行 run 命令时，可以向被执行的程序传递参数。例如，“(gdb)run example7_1 a b”，其中的“a”和“b”就是传递给程序 example7_1 的命令行参数。可以通过命令 set args 修改程序运行时的命令行参数。命令行参数一旦使用，在后续使用 run 命令启动程序执行时，如果未使用命令行参数，如“(gdb)run example7_1”，此时，gdb 仍然将使用上一次调用的命令行参数“a b”调用 example7_1 程序。可以通过执行命令 show args 查看当前调试程序的命令行参数。执行 set agrs 命令的方法如下所示。

```
(gdb)set args  -b  -x
```

print 的主要功能是显示数据，利用 print 命令可以显示程序中各个变量的当前值。另外，还有一个 whatis 命令，可以显示变量的数据类型，如下所示。

```
(gdb)whatis i
type = int *
```

print 命令除可以用于显示变量的值外，还可以直接调用程序的函数或者显示程序中复杂的组合数据类型的值。调用程序中函数的方法如下所示。

```
(gdb) print   func(1,2)
```

break 命令的功能是在代码里设置断点。设置断点操作将使程序执行到这里时被挂起。设置断点的方法有以下 4 种。

- break line-number：使程序恰好在执行给定行之前停止，如下所示。

```
(gdb) break 11
Breakpoint 1 at 0x8048501: file example7_2.c, line 11.
```

- break function-name：使程序恰好在进入指定的函数之前停止。
- break line-or-function if condition：如果 condition 是真，程序到达指定行或函数时停止。
- break routine-name：在指定例程的入口处设置断点。

可以通过执行命令 info break 显示当前设置的断信息，如下所示。

```
(gdb)info break
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x08048501 in main at example7_2.c:11
```

设置断点后，可以用 delete breakpoint 将断点删除，不带参数的 delete 命令将删除所有断点。可以在 delete breakpoint 后面跟上断点序号，可以单独删除某个断点。也可以不删除断点，而只是临时禁用一下，在适当的时候再启用断点。也可以通过命令 disable breakpoint 和 enable

breakpoint 完成。

- make：不退出 gdb 重新产生可执行文件。
- shell：不退出 gdb 执行 shell 命令。

【范例 7-4】假设存在某个程序如示例代码 7-4 所示，该程序实现了对 1 至 9 的数字求和。用 gdb 设置断点调试运行。

示例代码 7-4

```
1    #include <stdio.h>                    /*头文件*/
2    main()                                /*主函数*/
3    {
4        int i;                            /*循环变量*/
5        int sum;                          /*整型变量，用于保存求得的和*/
6        for(i=0;i<10;i++)                 /*循环*/
7            sum += i;                     /*累加*/
8        printf("%d\n",sum);               /*输出和*/
9    }
```

将上述程序进行编译链接 cc –g example7_4.c –o example7_4。运行上述程序，结果如下所示。

```
1    1073832717
```

从上述结果可以看出，该结果显然是不正确的。启动 gdb example7_4 进行调试，输入命令 list main 显示 main 的信息，如下所示。

```
(gdb) list main
1        #include <stdio.h>
2        main()
3        {
4            int i;
5            int sum=0;
6            for(i=1;i<10;i++)
7                sum += i;
8            printf("%d\n",sum);
9        }
```

在上述 list 的输出中，是 main 函数的程序列表。用户要在函数的第 7 行 sum +=i 处设置断点，检查每次循环前 sum 的值分别是多少。输入 break 7（代表断点所在的行号）命令，如下所示。

```
(gdb) break 7
Breakpoint 1 at 0x80483b7: file example7_1.c, line 7.
```

然后输入 run 命令运行该程序。当运行至主函数第 7 行设置的断点时将中断，如下所示。

```
(gdb) run
Starting program: /home/develop/section7/example7_1

Breakpoint 1, main () at example7_4.c:7
7                   sum += i;
```

此时，在 gdb 命令提示符下输入 print sum 显示 sum 变量的值，输出如下所示。

```
(gdb) print sum
$1 = 1073832672
```

根据此处的输出可以看到，在第一次进行加运算前 sum 的值是不合理的。仔细查看代码可以发现，sum 变量未进行初始化。将源程序的第 5 行修改为 int sum=0;问题即可得到解决。

7.3　使用 gdb 进行调试实例

假设用户需要计算整数 100 除以 5、4...直到 1 的值，假定程序代码如示例代码 7-5 所示。

示例代码 7-5

```
1    #include <stdio.h>                                    /*头文件*/
2    void func()                                           /*函数 func 定义*/
3    {
4        int i;                                            /*循环变量*/
5        for(i=5;i>=0;i--)                                 /*循环*/
6            printf("%d\n",100/i);                         /*求商并输出*/
7    }
8    main()                                                /*主函数*/
9    {
10       func();                                           /*调用 func*/
11   }
```

【运行结果】经过编译链接，运行上述程序，结果如下所示。

```
1    20
2    25
3    33
4    50
5    100
6    浮点数例外
```

从程序运行的输出可以看到，第 6 行输出时报错“浮点数意外”。为查找错误原因，启用 gdb 进行调试。在 gdb 中运行程序，结果如下所示。

```
20
25
33
50
100
Program received signal SIGFPE, Arithmetic exception.
0x080483b7 in func () at example7_2.c:6
6                   printf("%d\n",100/i);
```

根据 gdb 的输出可以看到，程序是在 printf 语句时出现错误。在 gdb 中打印当前变量 i 的值，如下所示。

```
(gdb) print i
$1 = 0
```

可以看到，此时变量 i 的值为 0。很明显，发生了除零异常。根据这一错误信息，调整程序重新编译链接即可。

7.4 小结

Linux 系统下的错误处理是每个开发人员必须掌握的内容。虽然 Linux 系统下的错误处理机制不像 Windows 系统下那么智能，但功能却毫不逊色。gdb 是 Linux 系统下进行程序调试时非常重要的工具，其功能非常强大。本章只是就一些较为简单的错误进行了调试，读者可以在实际开发过程中，使用 gdb 定位错误并进行程序修正，以提高程序质量及效率。

第 8 章　创建与使用库

函数库在 Linux 编程中扮演着重要的角色，在程序中所调用的大部分函数实际上都是函数库中的内容。本章将从函数库的概念讲起，介绍库函数与系统调用的区别，重点介绍静态库、动态库的创建与使用。通过本章的学习，读者应重点掌握如下内容。

- 理解库函数与系统调用的区别。
- 理解静态库与动态库的概念及区别。
- 重点掌握创建动态库及调用动态库的方法。

8.1　函数库介绍

在进行 Linux 编程过程中，不可避免地需要使用函数库。函数库中提供了各种各样的函数供用户编程过程中进行调用。

8.1.1　系统函数库的使用

在进行程序设计的过程中，经常需要调用一些系统函数。这些函数实际上就是 Linux 系统提供的开发函数库。通常情况下，这些函数库位于/usr/lib 和/lib 目录下。例如，通常使用的 fprintf 函数就是由标准输入或输出函数库提供的，其对应的函数库文件是/usr/lib/libc.a。

使用系统函数库的方法很简单，只要在源程序中包含相应的头文件，在程序中就可以直接使用。像上述提到的 fprintf 函数，只要包含头文件 stdio.h 即可。在程序进行链接时，指定链接 libc.a 就可以生成可独立运行的程序。可以通过执行命令 ar 查询系统函数库的组成。通常函数库是由若干个目标文件组成的。查看函数库内容的过程如下所示。

```
ar tv libc.a
```

如果需要查看某个目标文件中提供哪些函数，可以先通过 ar 命令的 x 选项将目标文件从函数库中分离出来，再执行 nm 命令查看相应的目标文件中提供哪些函数。以 libc.a 为例，这一过程如下所示。

```
ar x libc.a fprintf.o
nm fprintf.o
```

输出信息如下所示。

```
00000000 W _IO_fprintf
00000000 T fprintf
         U vfprintf
```

根据输出信息可以看到，fprintf.o 中包含了函数 fprintf 的定义。

提示：对函数库进行链接通过指定-l 选项完成。要注意链接时使用-lname 指定的函数库是 libname.a。

8.1.2 用户自定义函数库的创建和使用

除了使用系统的函数库，用户也可以将常用的功能整理成函数库，供开发过程中调用。

（1）要形成用户自定义的函数库，需要经过以下几个步骤。

1）将功能整理成函数。将总结归纳的代码形成 C 源程序。通常情况下，每个源程序可以包含若干个功能函数。

2）将函数编译成目标文件。将源程序进行编译（不进行链接），形成二进制的目标文件（扩展名通常为.o）。

3）将目标文件打包成函数库。执行 ar 命令的 c 参数（新建）或者 r 参数（附加），将多个目标文件打包成函数库。

（2）当函数库创建完成后，要在用户的程序中引用，需要经过以下几个步骤。

1）在源程序中包含相关头文件的信息。头文件中定义了函数库中提供的函数声明原型。

2）在源程序中调用函数库中相关的函数。

3）在链接时通过-l 选项指定要链接的函数库。例如，用户自定义的函数库的名称为 libtool.a，在链接时，应使用-ltool 选项将程序的目标代码与函数库进行链接，以形成最终的可执行文件。

【范例 8-1】在编程过程中，经常需要用到字符串的处理功能。编程实现用户自定义的字符串处理函数库 libmystring.a。假定包含两个源程序文件，分别是 string1.c 和 string2.c。源程序 string1.c 实现的功能主要包括字符串的去空格操作和字符串的比较操作；源程序 string2.c 实现的是字符串大小写转换的功能，包括大写转换小写、小写转换大写两个函数。两个源程序分别封装了部分字符串处理的函数。这两个程序的实现代码如示例代码 8-1 和示例代码 8-2 所示。

示例代码 8-1

```
#include <stdio.h>                                  /*头文件*/
#include <stdlib.h>
#include <string.h>
char* TrimStringInRight(char* pszDest)              /*去掉字符串右侧的空格*/
{
    int n;                                          /*定义整型变量*/
    if (pszDest != NULL)                            /*判断输出串是否为空*/
    {
        for (n = strlen(pszDest); n > 0; n--)       /*从字符串最右边开始按字节循环*/
        {
            if (!isblank(pszDest[n - 1]))           /*如果不是空字符，则中止循环*/
                break;                              /*中止循环*/
        }
        pszDest[n] = '\0';                          /*是空字符，则将该字符清空*/
    }
    return pszDest;                                 /*返回处理过的字符串*/
}
```

```
18  char * TrimStringInLeft(char* pszDest)                    /*去掉字符串左侧的空格*/
19  {
20      int n, nLen;                                          /*定义整型变量*/
21      if (pszDest != NULL)                                  /*判断输入串是否为空*/
22      {
23          nLen = strlen(pszDest);                           /*求输入字符串的长度*/
24          for (n = 0; n < nLen; n++)                        /*从字符串最左侧开始循环*/
25          {
26              if (!isblank(pszDest[n]))                     /*如果不是空字符，则中断循环*/
27                  break;
28          }
29          strcpy(pszDest, pszDest + n);                     /*是空字符，则把从当前位置开始的后
    续的字符复制到开始位置*/
30      }
31      return pszDest;                                       /*返回处理完成的字符串*/
32  }
33  int CmpStringNoCase(char *pszAString,char *pszBString) /*不区别大小写比较两个字符串*/
34  {
35      register int i = 0;                                   /*定义整型变量*/
36      while(*(pszAString + i) && *(pszBString + i))         /*分别从两个字符串的开始进行循环*/
37      {
38          if((*(pszAString + i)& 0x0F) != (*(pszBString + i) & 0x0F))   /*如果两个字符串的当前
    位置不相等*/
39              return(*(pszAString + i) - *(pszBString + i));            /*返回两个字符的差*/
40          i ++;                                             /*循环变量加 1，比较下一个字符*/
41      }
42      if(*(pszAString + i))          /*如果字符串 A 还未结束，则返回字符串 A 当前位置的字符*/
43          return(*(pszAString + i));
44      if(*(pszBString + i))          /*如果字符串 B 还未结束，则返回字符串 B 当前位置的字符*/
45          return(*(pszBString + i));
46      return 0;                                             /*返回 0，表示两个字符串相等*/
47  }
```

对程序 string1.c 进行编译，并生成库函数文件的过程如下所示。

```
1   cc –c –O string1.c
2   ar cv string1.o libmystring.a
```

【代码解析】在编译 string1 完成后，调用档案库维护命令 ar 创建了一个新的函数库 libmystring.a。由于该函数库是一个新的函数库，所以使用参数 c。在后续介绍向 libmystring.a 中追加 string2.o 时，可以看到使用了向函数库附加目标文件的选项 r。

通过上述操作，可以看到在当前目录下已经生成了新的函数库文件 libmystring.a。在编写程序的过程中，可以随时引用这个函数库中已实现的函数。在需要链接时通过-l 选项指定链接该函数库即可（-lmystring）。

示例代码 8-2

```
1   #include <stdio.h>                                        /*头文件*/
2   #include <stdlib.h>
3   #include <string.h>
4   char *LowerString(char *pszInputString)                   /*转换为小写的函数*/
5   {
6       int i;                                                /*循环变量*/
7       for( i = 0 ;pszInputString[i] != 0; i ++)             /*从头循环检查字符串*/
```

```
8               pszInputString[i] = tolower(pszInputString[i]);     /*转换为小写*/
9           return pszInputString;                                  /*返回转换后的串*/
10  }
11   char *UpperString(char *pszInputString)                        /*转换大写的函数*/
12  {
13          int i;                                                  /*循环变量*/
14          for( i = 0 ;pszInputString[i] != 0; i ++)               /*从字符串开始循环检查*/
15              pszInputString[i] = toupper(pszInputString[i]);     /*转换为大写*/
16          return pszInputString;                                  /*返回转换后的串*/
17  }
```

对程序 string2.c 进行编译，并生成库函数文件的过程如下所示。

```
1   cc –c –O string2.c
2   ar rv string2.o libmystring.a
```

【代码解析】在编译 string2 完成后，调用档案库维护命令 ar 向函数库 libmystring.a 追加了目标文件 string2.o。此时，libmystring.a 就是一个包含 2 个目标代码文件的函数库。如果有必要，也可以随时向其中增加其他的目标代码。用户编程过程中，可以引用这些目标代码中的任意函数。

8.2 库函数与系统调用

Linux 系统下的编程过程中，除了可以使用各种函数库外，还可以使用系统调用。系统调用是操作系统开放给用户的调用接口。实质上也是一种函数。不过，这种函数在执行过程中需要由用户态切换到核心态。

从可移植性角度看，使用库函数具有更好的可移植性。从执行效率看，由于部分库函数的实现过程中都调用了系统提供的系统调用完成一定的功能，在这种情况下，调用系统调用比调用库函数效率要高一些。但是也存在这种情况，部分库函数的实现没有调用系统调用的接口。这种情况下，使用系统调用由于存在由用户态到核心态的切换过程，反而不如函数库效率更高。Linux 系统下系统调用与库函数的层次如图 8-1 所示。

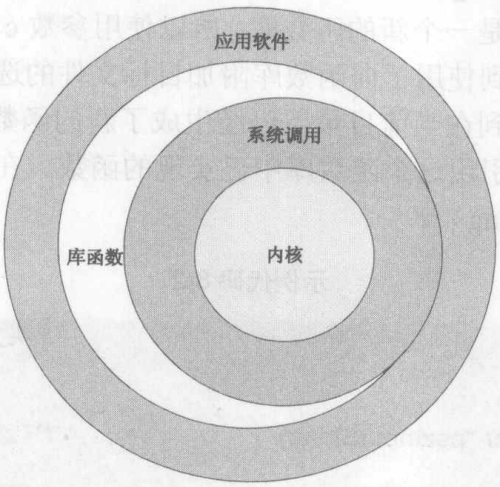


图 8-1　系统调用与库函数

8.2.1　系统调用介绍

Linux 内核中设置了一组用于实现各种系统功能的子程序，称为系统调用。用户可以通过系统调用命令在自己的应用程序中调用它们。系统调用和普通的函数调用非常相似。它们的区别在于系统调用由操作系统内核提供，运行于核心态；而普通的函数调用由函数库或用户自己提供，运行于用户态。

通常情况下，进程是不能访问内核的。它不能访问内核所占内存空间也不能调用内核函数。这是由 CPU 的硬件保护措施决定的，也就是所谓的“保护模式”。系统调用是这些规则的一个例外。其原理是进程先用适当的值填充寄存器，然后调用一个特殊的指令，这个指令会跳到一个事先定义的内核中的一个位置，这个位置是用户进程可读但是不可写的。这一过程称为“自陷”。通过自陷机制，用户的进程就可以访问在核心态运行的内核系统调用。系统调用的实现过程如图 8-2 所示。

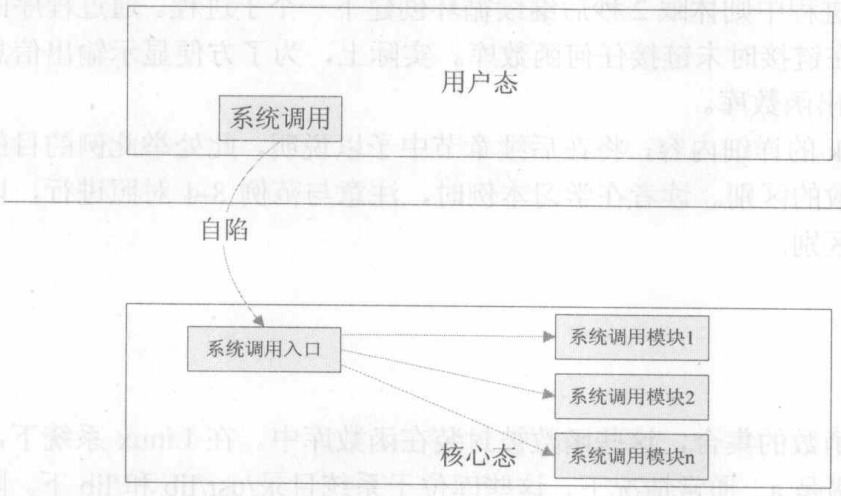


图 8-2　系统调用的实现

【范例 8-2】编写程序调用系统调用 fork 创建子进程。其实现过程如示例代码 8-3 所示。

示例代码 8-3

```
#include <unistd.h>                                          /*头文件*/
main()                                                       /*主函数*/
{
    int I;                                                   /*定义整型变量*/
    int nPid;                                                /*定义整型变量*/
    for(i=0;i<5;i++)                                         /*循环创建 5 个子进程*/
    {
        switch((nPid = fork()))                              /*调用 switch 判断 fork 的返回值*/
        {
            case -1:                                         /*调用 fork 失败*/
                    break;                                   /*中断循环*/
            case 0:                                          /*子进程*/
                    printf("Child process ID is %d\n",getpid());/*输出子进程 ID*/
                    exit(0);                                 /*子进程退出*/
```

```
17                  default:                                    /*父进程*/
18                          sleep(2);                           /*休眠 2 秒*/
19                          break;                              /*中断循环*/
20              }
21      }
22  }
```

对该程序进行编译链接，生成可执行程序的过程如下所示。

```
23  cc   example8_3.c –o example8_3
```

【运行结果】在 shell 下运行范例代码 8-3 中的程序，其结果如下所示。

```
1   Child process ID is 2922
2   Child process ID is 2923
3   Child process ID is 2924
4   Child process ID is 2925
5   Child process ID is 2926
```

【代码解析】在示例代码 8-3 中，通过系统调用 fork 循环创建子进程。在子进程中输出子进程的 ID，而在父进程中则休眠 2 秒后继续循环创建下一个子进程。通过程序的编译链接过程可以看到，程序在链接时未链接任何函数库。实际上，为了方便显示输出信息，在程序中引用了标准输入/输出函数库。

关于系统调用 fork 的详细内容，将在后续章节中予以说明，此处举此例的目的仅是为了说明系统调用与库函数的区别。读者在学习本例时，注意与范例 8-4 对照进行，以明确系统调用与库函数之间的区别。

8.2.2 库函数介绍

库函数实际上是函数的集合，这些函数被封装在函数库中。在 Linux 系统下，库函数也称为档案文件，其后缀是.a。通常情况下，这些库位于系统目录/usr/lib 和/lib 下。除去系统提供的函数库外，程序员也可以根据自己平时的积累，形成自己的函数库。可以把一些常用的功能进行标准化，再封装成函数库。这一过程的基本操作为以下几点。

（1）编写标准的函数。根据平时的积累，抽象出部分功能并用函数的形式加以实现。

（2）编译这部分函数，形成目标文件。将这些包含函数的源程序进行编译，形成.o 文件。

（3）将若干目标文件打包成函数库。在 Linux 系统下，提供了 ar 命令用于生成或者维护函数库。

【范例 8-3】调用数学库函数 sqrt 实现计算输入整型数的平方根。其实现过程如示例代码 8-4 所示。

示例代码 8-4

```
1   #include <stdio.h>                                  /*标准输入输出头文件*/
2   #include <math.h>                                   /*数学库所需要的头文件*/
3   main()                                              /*主函数*/
4   {
5       int I;                                          /*定义整型变量*/
6       int num;                                        /*定义整型变量，用于保存输入的整数*/
7       double result;                                  /*定义双精度变量，用于保存求平方根的结果*/
```

```
8        for(i=0;i<5;i++)                          /*循环 5 次*/
9        {
10           scanf("%d\n",&num);                   /*从键盘接收整数*/
11           if(num < 0) break;                    /*如果输入数据小于 0，中断循环*/
12           result = sqrt(num);                   /*计算平方根*/
13           printf("\n%.2f\n",result);            /*输出计算的结果*/
14       }
15   }
```

对该程序进行编译链接，生成动态库的过程如下所示。

```
1    cc example8_4.c –lm –o example8_4
```

【运行结果】在 shell 下运行范例代码 8-4 中的程序，其结果如下所示。

```
1    5
2    2.24
3    3
4    1.73
```

【代码解析】在示例代码 8-4 中，系统通过调用数学函数库中的计算平方根函数 sqrt，对用户从键盘输入的整数求平方根。从程序编译链接的过程看，链接的过程中使用了-m 选项，其作用是将目标代码与库函数 libm.a 进行链接。这说明，在调用函数库时，需要在链接时指定要链接的库文件。而系统调用是不需要执行与相关函数库链接的，请读者注意区分这一区别。

8.3　动态库的创建与使用

动态链接库也是一种函数库。与静态库不同的是，动态链接库中的内容并不链接到可执行文件中去。在编译链接时，只是将动态链接库所在位置的信息链接写入可执行文件。在运行过程中，动态链接库的内容也是为当前系统全局共享的。如果有多个进程都调用某动态链接库，则内存中只有该动态链接库的一份拷贝。动态库与静态库的关系，类似于 Windows 环境下 dll 与 lib 的关系，但又并不完全相同。

在 Linux 系统下要创建动态链接库，需要经过以下两个步骤。

（1）编译时指定-fPIC 选项。该选项通知 gcc 编译器产生可以重定位的与位置无关的目标代码。

（2）链接时指定-shared 选项。通过这一选项，使 gcc 编译器生成动态链接库。

动态链接库创建完成后，如何在用户的程序中应用呢？要调用动态链接库中的函数，主要有两种方式：一种是通过设置环境变量 LD_LIBRARY_PATH；另外一种是通过 dlopen 系列函数完成。在第 5.2 节中已向读者介绍过通过设置环境变量的方式，在本节中不再赘述。下面着重向读者介绍 dlopen 方式加载动态链接库。dlopen 系列函数主要包括 4 个函数：dlopen、dlsym、dlclose 和 dlerror。下面分别介绍各个函数在调用动态链接库时的作用。

1．dlopen 函数

Dlopen 函数打开动态链接库并且映射到当前进程的内存，并且返回句柄。该函数的声明在头文件<dlfcn.h>中，其原型如下所示。

```
#include <dlfcn.h>
void *dlopen (__const char *__file, int __mode);
```

（1）参数说明如下。

- __file：输入参数，动态链接库文件路径。文件路径可以是绝对路径，也可以为相对路径。如果未使用路径，而是直接用动态链接库文件名称，则根据环境变量 LD_LIBRARY_PATH 的值查找相应的动态库文件。
- __mode：输入参数，打开方式。可以选择的打开方式有两种。一种是 RTLD_LAZY，只打开动态库文件，而暂时不解析外部对动态库的引用，直到动态库函数被执行时再解析对动态库的引用；第二种是 RTLD_NOW，打开动态库文件，在 dlopen 返回前解析所有的对动态库的引用。

（2）返回值说明如下。

- NULL：调用 dlopen 函数失败。可以调用 dlerror 获取详细的错误信息。
- 其他：打开成功，返回值为访问动态链接库的句柄。

2．dlsym 函数

dlsym 函数的功能是返回一个指向被请求入口点的指针，其原型如下所示。

```
#include <dlfcn.h>
void *dlsym (void *__restrict __handle,__const char *__restrict __name);
```

（1）参数说明如下。

- __handle：输入参数，动态链接库句柄。该句柄由 dlopen 函数返回。
- __name：输入参数，要调用的动态链接库中的函数名称。

（2）返回值说明如下。

- NULL：调用 dlsym 函数失败。可以调用 dlerror 函数获取详细的错误信息。
- 其他：打开成功，返回值为指向要调用的函数的指针。在定义 dlsym 函数的返回值变量（一个指向函数的指针）时，要注意与要调用的函数的形式保持一致。

3．dlclose 函数

dlclose 函数的功能是关闭已打开的动态链接库，其原型如下所示。

```
#include <dlfcn.h>
int dlclose (void *__handle);
```

（1）参数说明如下。

- __handle：输入参数，动态链接库句柄。该句柄由 dlopen 函数返回。

（2）返回值说明如下。

- 0：调用 dlclose 函数成功。
- 其他：调用 dlclose 函数失败。

4．dlerror 函数

dlerror 函数用于输出动态库操作过程中的错误信息。一般情况下，在调用 dlopen 函数或者 dlsym 函数失败时调用，其原型如下所示。

```
#include <dlfcn.h>
char *dlerror (void);
```

返回调用 dlopen 系列函数时的详细错误信息。

【范例 8-4】编写动态库实现两个整型数相加。编写程序打开该动态库，并调用其加法函数。其实现过程如示例代码 8-5（动态库）和示例代码 8-6（调用程序）所示。

示例代码 8-5

```
#include <stdio.h>                                  /*头文件*/
int addfunc(int x,int y)                            /*函数定义，实现两个整数相加*/
{
    return x+y;                                     /*返回两个整数的和*/
}
```

对该程序进行编译链接，生成动态库的过程如下所示。

```
cc –fPIC –c example8_5.c
cc –shared example8_5.o –o example8_5.so
```

示例代码 8-6

```
#include <stdio.h>                                  /*头文件*/
#include <dlfcn.h>                                  /*头文件，包含了 dl 系列函数的声明*/
main()                                              /*主函数*/
{
    void *handle;                                   /*用于保存动态链接库的句柄*/
    int (*callfunc)(int,int);                       /*指向函数的指针，用于保存查找到的函数 addfunc 的指针*/
    if((handle = dlopen("./example8_1.so",RTLD_LAZY)) == NULL) /*打开动态库并返回句柄*/
    {
        fprintf(stderr,"dlopen:%s\n",dlerror());    /*输出错误信息*/
        exit(0);                                    /*退出程序*/
    }
    dlerror();                                      /*清除 dl 的错误缓冲区，以便进行下一次 dl 调用*/
    if((callfunc = dlsym(handle,"addfunc")) == NULL) /*在打开的动态库中查找函数 addfunc*/
    {
        fprintf(stderr,"dlsym:%s\n",dlerror());     /*输出错误信息*/
        exit(0);                                    /*退出程序*/
    }
    printf("1+2=%d\n",callfunc(1,2));               /*调用动态库中的函数，完成功能并输出结果 */
    dlclose(handle);                                /*关闭动态链接库*/
}
```

对该程序进行编译链接，生成可执行程序的过程如下所示。

```
cc –c   -O example8_6.c
cc example8_6.o –ldl –o example8_6
```

【运行结果】在 shell 下运行示例代码 8-6 中的程序，其结果如下所示。

```
1+2=3
```

【代码解析】在示例代码 8-5 中，实现了加法函数 addfunc，然后编译生成动态链接库。在示例代码 8-6 中，通过 dlopen 打开该动态链接库，并调用其中的函数计算两个整数的和。在示例代码 8-6 中，源码的各行解释如下。

- 第 1～2 行：头文件信息。

- 第 5 行：动态链接库句柄定义。
- 第 6 行：函数指针定义，用于保存 dlsym 返回的函数指针。
- 第 7～11 行：调用 dlopen 函数打开示例代码 8-5 所创建的动态链接库。
- 第 12 行：调用 dlerror 函数清除前面操作产生的错误，以确保后续调用 dlerror 函数时不遗留原来的错误。
- 第 13～17 行：调用 dlsym 函数映射函数 addfunc 函数到当前进程地址空间。
- 第 18 行：调用加法函数实现求两个整数的和并输出。
- 第 19 行：调用 dlclose 函数关闭动态链接库。

提示：在调用 dlopen 系列函数时，要在程序中包含头文件<dlfcn.h>，在链接时要链接函数库 libdl.a。

8.4 综合实例

在 Linux 系统下，提供了数学函数库，其中常用的静态库的库文件名是/usr/lib/libm.a，动态库的文件名是/usr/lib/libm.so。要求通过 dlopen 方式编程实现调用动态数学库中的函数 sqrt 计算一个输入整数的平方根。根据上述综合实例的要求，结合动态链接库的动态加载方法，实现代码如下所示。

```
1   #include <stdio.h>                              /*头文件*/
2   #include <stdlib.h>                             /*进行 atof 计算时需要用到此头文件*/
3   #include <dlfcn.h>                              /*dlopen 系列函数使用该头文件*/
4   main(int argc,char *argv[])  /*主函数，带有命令行参数。命令行参数 1 表示要求平方根的正数*/
5   {
6       float n;                                    /*用于保存从命令行输入的正数*/
7       void *handle;                               /*动态库的句柄*/
8       double (*callsqrt)(double);                 /*指向函数 sqrt 的指针。定义函数时注意参数及返回
    值要与动态库中一致*/
9       if(argc < 2 ||(n=atof(argv[1])) < 0)        /*检查是否有输入参数或者输入的数据是否是正数*/
10      {
11          fprintf(stderr,"Usage:%s 正数\n",argv[0]);   /*输出用法提示信息*/
12          exit(0);                                     /*退出程序*/
13      }
14      if((handle = dlopen("/usr/lib/libm.so",RTLD_NOW))  == NULL) /*调用 dlopen 打开动态库
    libm.so*/
15      {
16          fprintf(stderr,"dlopen:%s\n",dlerror());     /*输出错误提示*/
17          exit(0);                                     /*退出程序*/
18      }
19      dlerror();                          /*清除 dl 的错误缓冲区,以便进行下次 dl 系列函数调用*/
20      if((callsqrt = dlsym(handle,"sqrt")) == NULL) /*在动态库中查找函数 sqrt 并返回函数的指针*/
21      {
22          fprintf(stderr,"dlsym:%s\n",dlerror());
23          exit(0);
24      }
25      printf("%f\n",callsqrt(n));             /*调用函数实现求平方根的功能，并输出所求得的结果*/
26      dlclose(handle);                        /*关闭动态链接库的句柄*/
```

```
27  }
```

【运行结果】在 shell 下经过编译链接后运行程序，其结果如下所示。

```
1   ./example8_0 5
2   2.236068
```

8.5　小结

函数库是 Linux 系统下编程必不可少的部分。特别是动态库的使用，在某些环境中是非常有用的。读者应该在编程过程中多加积累，形成自己的函数库，以方便后续的开发工作。根据实际的工程要求，本章首先重点分析了库函数与系统调用的区别，让读者对库函数和系统调用的概念有深刻的理解，然后重点介绍了静态库、动态库的创建与使用，最后通过一个综合案例分析了函数库的具体使用方法。

第三篇　输入输出及进程管理

第 9 章　文件操作

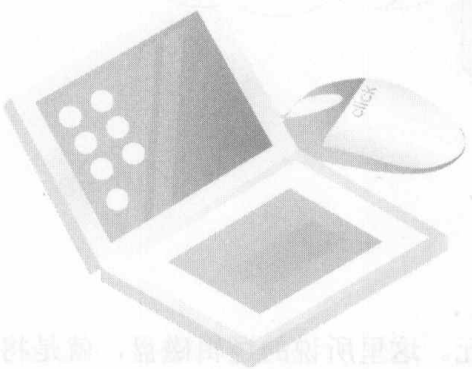
第 10 章　标准输入输出库

第 11 章　界面程序设计——Qt

第 12 章　界面程序设计——GTK+

第 13 章　进程

第 14 章　信号



第9章 文件操作

尽管内核是 Linux 系统的核心，但文件却是用户与操作系统交互所采用的主要工具，这对 Linux 来说尤为如此。文件系统作为操作系统的一部分，与用户联系最为密切。绝大部分用户接触系统，首先是从访问文件开始的。因此了解怎样建立文件、如何使用文件便成为系统入门的最初工作。本章首先简要介绍了 Linux 文件系统的基本结构，在此基础之上，结合多个具体实例对不带缓冲的 I/O 操作进行了详细讲解。通过本章的学习，读者应重点掌握如下内容。

- 了解 Linux 文件系统的结构。
- 掌握 Linux 文件的种类及文件的权限管理，特别是文件权限的管理。
- 掌握普通文件的建立、打开及读写等操作。
- 理解目录项的结构及目录管理操作。
- 学会使用系统调用获取及修改文件的属性。

9.1 Linux 文件系统简述

文件系统是操作系统的一个重要组成部分，用于对磁盘进行存储管理及输入输出。它向用户提供了一致的、友好的访问接口，屏蔽了对物理设备的操作细节。Linux 系统可以同时支持多个物理存储设备，而每个设备上又可以划分成一个或多个文件系统。每个文件系统由逻辑上的功能块的序列组成，这些功能块包括引导块、超级块、i 节点块、数据块等。文件系统的总体结构如图 9-1 所示。

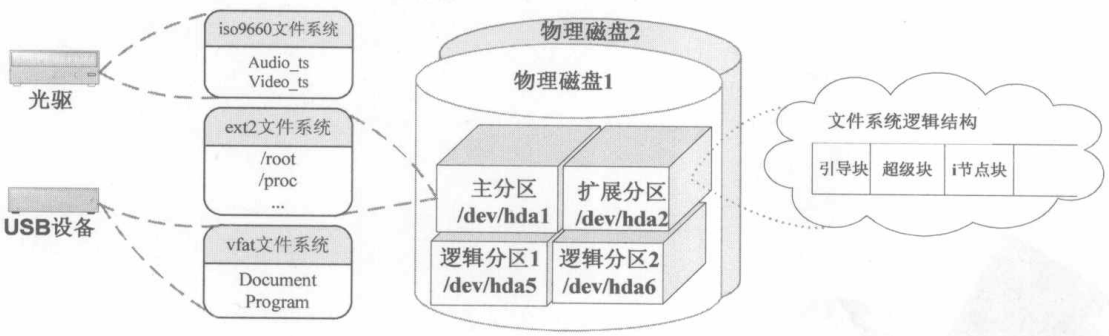


图 9-1 文件系统总体结构

9.1.1 逻辑磁盘分区管理

在 Linux 系统中，逻辑磁盘分区是基本的文件存储单元。这里所说的逻辑磁盘，就是将

一个或者多个物理磁盘进行逻辑划分，从而形成逻辑上的磁盘分区。Linux 将每个逻辑划分视为一个独立的磁盘，每个逻辑磁盘在 Linux 系统中表现为一个单独的设备文件。系统通过存放在/dev 目录下的设备文件对逻辑设备进行 I/O 操作。逻辑磁盘分区是建立 Linux 文件系统的基础。只有创建了磁盘分区后，才能在分区上建立文件系统。建立磁盘逻辑分区可通过执行命令 fdisk 来完成。

【范例 9-1】查看系统的逻辑分区。其实现过程如示例代码 9-1 所示。

示例代码 9-1

```
1    fdisk –l                    /*-l 选项表示要获取分区信息*/
```

【运行结果】在 shell 下运行上述脚本，其结果如下所示。

```
1    Disk /dev/sda: 36.4 GB, 36401479680 bytes
2    255 heads, 63 sectors/track, 4425 cylinders
3    Units = cylinders of 16065 * 512 = 8225280 bytes
4    Device    Boot      Start         End      Blocks   Id  System
5    /dev/sda1              1         129     1036161   82  Linux swap
6    /dev/sda2    *       130        4425    34507620   83  Linux
```

【代码解析】在本例中，fdisk 是用来查看或调整磁盘分区的命令。读者如果以前用过 Windows 系统下的磁盘分区工具的话，对于 PartitionMagic 应该不会陌生。fdisk 是 Linux 系统环境下的磁盘分区工具。“-l”选项表示要查看磁盘分区。在本例中，输出信息的含义具体如下。

- 第 1 行：第一个磁盘的容量是 36.4GB。
- 第 2 行：第一个磁盘的参数信息，有 255 个磁面、63 个扇区、4425 个柱面。
- 第 3 行：该磁盘的每个柱面的容量是 16065 * 512 字节，约为 7.84MB。
- 第 4 行：明细输出的标题，对应的解释分别为磁盘分区名称、是否是引导分区，分区开始的柱面、分区终止的柱面、分区容量（单位为 K）、分区类型 Id（7 表示 NTFS 分区）分区类型。计算分区容量可通过如下公式进行：(End-Start)*每柱面容量。
- 第 5 行：该磁盘是系统中第一块盘的第一个分区，从柱面 1 至 129，共 1036161 块。该分区是 Linux 操作系统的交换分区。
- 第 6 行：该磁盘是系统中第一块盘的第二个分区，从柱面 130 至 4425，共 1036161 块。该分区是可引导分区。

在上述结果中，第 5 行的输出信息/dev/sda1 中，s 代表这是 SCSI 设备或者移动存储设备（如果是 IDE 设备则用 h 代表），a 代表第一块磁盘设备，1 代表第一个分区。如果系统中存在第二块磁盘设备，则以/dev/sdb 来代表，依此类推。而磁盘设备内的分区用数字指代，如/dev/sda2 代表 SCSI 驱动器上的第 2 个分区。

提示：swap 分区即交换分区。它的作用是在系统的物理内存紧张的时候，将物理内存的部分数据交换出来，以供当前正在运行的程序使用。Linux 系统按照一定的调度算法决定哪些内存空间被交换出来。这部分被交换出来的空间的数据被临时保存到 swap 分区中，等到这部分数据要使用时，再从 swap 中恢复保存的数据到内存中。

9.1.2 文件系统的建立与挂载

要想正式使用文件系统，仅对物理存储设备进行逻辑分区是远远不够的。一个逻辑分区只有建立了某种文件系统后，这个分区才能使用。所以，在分区之后还需要对新增的逻辑分区进行格式化。建立文件系统的过程，就是用相应格式化工具格式化逻辑分区的过程。这个过程和在 windows 中格式化某个分区为 NTFS 分区类似。在 Linux 系统下，进行分区格式化的命令是 mkfs。

实际上，mkfs 只是个前端程序，读者在实际使用中应该通过执行 mkfs.fstype 命令建立各种不同格式的文件系统。例如，执行 mkfs.ext2 命令可以建立 ext2 文件系统，执行 mkfs.vfat 命令可以建立 vfat 文件系统。

【范例 9-2】在逻辑磁盘分区/dev/hda5 上创建 ext2 文件系统。其实现过程如示例代码 9-2 所示。

示例代码 9-2

```
1    #mkfs.ext2 /dev/hda5      /*mkfs.ext2 命令用于创建 ext2 文件系统  /dev/hda5 指逻辑磁盘分区*/
```

【运行结果】在 shell 下运行上述脚本，系统输出如下所示。

```
1    mke2fs 1.34 (25-Jul-2003)
2    Filesystem label=
3    OS type: Linux
4    Block size=1024 (log=0)
5    Fragment size=1024 (log=0)
6    49152 inodes, 196024 blocks
7    9801 blocks (5.00%) reserved for the super user
8    First data block=1
9    24 block groups
10   8192 blocks per group, 8192 fragments per group
11   2048 inodes per group
12   Superblock backups stored on blocks:
13       8193, 24577, 40961, 57345, 73729
14   Writing inode tables: done
15   Writing superblocks and filesystem accounting information: done
16   This filesystem will be automatically checked every 36 mounts or
17   180 days, whichever comes first. Use tune2fs -c or -i to override.
```

【代码解析】在本例中，创建文件系统输出的含义具体如下。

- 第 1 行：mkfs 的版本信息。
- 第 2 行：新文件系统的标签。
- 第 3 行：新文件系统的类型，此处是 Linux。
- 第 4～11 行：输出新建立的文件系统的 i 节点信息、文件块信息等。
- 第 14～17 行：文件系统建立完成。

Linux 支持多种文件系统，常见的如 ext2、ext3、swap、vfat、ISO9660、日志文件系统等。各文件系统的内容包括以下几点。

- ext2：Linux 中自带的文件系统类型。该文件系统是 Linux 中原来使用的 ext 文件系统的后续版本，在 Linux 的早期发行版本中一直使用 ext2 作为操作系统默认使用的文件系统。

- ext3：ext3 是 ext2 文件系统的后续版本，是在 ext2 文件系统上加入了文件系统日志的管理机制。
- swap：swap 文件系统在 Linux 中作为交换分区使用，交换分区用于操作系统管理内存的交换空间。在安装 Linux 操作系统时，交换分区是必须建立的，并且其类型一定是 swap。交换分区由操作系统自动管理，用户不需要对其进行过多的操作。
- vfat：在 Linux 中把 DOS 下的所有 FAT 文件系统统称为 vfat，其中包括 FAT12/FAT16/FAT32。在 Redhat 9 中即可以使用同系统中已存在的 FAT 分区，也可以建立新的 FAT 分区。
- ISO9660：ISO9660 是光盘所使用的标准文件系统，在 Linux 中对光盘有非常好的支持，不仅可以读取光盘中的文件，还可以进行光盘的刻录。
- 日志文件系统：提供日志功能是文件系统发展的趋势，尤其对于像 Linux 这样通常作为服务器使用的操作系统更是如此。

在文件系统建立后，必须挂载到系统中才能通过文件管理系统对其进行访问。所谓挂载，就是将逻辑磁盘分区与文件系统的指定目录之间建立关联。对于 Windows 系统，在逻辑分区建立完成后，将根据分配的盘符（C 盘、D 盘等）由操作系统将逻辑磁盘自动挂载到系统上。在 Linux 下没有盘符的概念，而是挂载到某个指定的目录。挂载文件系统有两种方式，执行 mount 命令进行手工挂载和通过 fstab 文件进行自动挂载。

手工挂载文件系统在 Linux 系统中是通过执行 mount 命令完成的，通常适用于光盘、软盘等不需要经常使用的存储设备，当然也可以挂载硬盘分区。通过 mount 挂载的外部文件系统，在使用完毕后，可通过执行 umount 命令将文件系统卸载。

对于系统运行必须使用或系统中长期使用的文件系统，需要设置其在系统运行时自动运行挂载。文件系统的自动挂载文件是通过配置 fstab 文件实现的。在 Linux 启动过程中，init 进程会自动读取文件/etc/fstab 中的配置内容挂载相应的文件系统，所以只需要在 fstab 文件中加入相应的设置行，就可以使系统在启动时自动挂载指定的文件系统。

【范例 9-3】安装光盘到当前系统的/mnt/cdrom 目录下。其实现过程如示例代码 9-3 所示。

示例代码 9-3

```
1    #mount -t iso9660 -o ro /dev/cdrom /mnt/cdrom  /* -t 选项指定文件系统的类型;–o 指定文件系统的读写权限*/
```

【运行结果】在 shell 下运行上述脚本，系统无错误输出则表示安装成功。通过执行 df 命令显示当前系统已安装的文件系统，输出如下所示。

```
2    # df –T
3    Filesystem     Type        1K Blocks      Used      Available   Use%  Mounted on
4    /dev/sda2      reiserfs     34506524    6325272     28181252    19%   /
5    /dev/cdrom     iso9660        614400     614400           0    100%  /mnt/cdrom
```

【代码解析】在本例的输入中，包括 4 个输入参数，分别是文件系统类型（iso9660）、文件系统的权限（ro：只读）、包含文件系统的物理块设备（/dev/cdrom）以及此新文件系统要安装到的目录名称（/mnt/cdrom）。上述命令执行成功后，将把光盘挂载到/mnt/cdrom 目录下，如上述输出信息的第 4 行所示。在输出信息中，第一列代表设备文件名称，第二列代

表文件系统类型，第三至第六列是文件系统的大小及使用情况，最后一列是文件系统的挂载目录。

读者平时在使用 Windows 系统时，可能会留意到 Windows 系统对外部设备是“即插即用”的，即系统会自动完成设备的挂载工作。最典型的是在使用 U 盘时，将设备插入 USB 接口后，系统将自动识别并分配盘符。而在低版本的 Linux 系统中，是需要手工执行命令挂载的。在较高版本的 Linux 系统中，也实现了与 Windows 系统类似的“即插即用”的功能。

本例的输出是通过执行 df 命令查看的。在 Linux 系统下，df 命令被广泛地用来生成文件系统的统计数据，它能显示系统中所有文件系统的信息，包括总容量、可用的空闲空间、目前的安装点等。-T 选项表示输出文件系统的类型。在本例中，输出信息的含义具体如下。

- 第 3 行：在磁盘分区 sda2 上挂载了 reiserfs 文件系统。文件系统大小是 34506524 个块（每块 1024 字节），已使用 19%，文件系统挂载点是“/”。从这里读者也可以看到，这是 Linux 的根文件系统。Linux 根文件系统是在系统启动过程中自动加载的。
- 第 4 行：在设备 cdrom 上挂载了 iso9660 文件系统。文件系统大小是 614400 个块，挂载点是/mnt/cdrom。在访问该文件系统时，直接访问挂载目录/mnt/cdrom 即可。

在执行 mount 命令挂载文件系统时，应该注意以下几点。

- 挂载点必须是目录，不允许是文件。
- 挂载的目录可以不为空，但在挂载成功后，该目录原来的文件将暂时不可访问。待文件系统卸载后才能访问。

文件/etc/fstab 存放的是系统中的文件系统信息。在文件中，每行对应一种文件系统。在每一行中，选择<Tab>键或者<Space>键分隔各字段。fstab 中定义的文件系统是有序的。在进行文件系统检查 fsck 及挂载 mount 等操作时，将按照 fstab 文件中的顺序进行。文件中各字段的解释如下。

- 字段一：文件系统的定义。描述文件系统所在的设备或者远程文件系统。一般情况下，IDE 设备描述为/dev/hdSN。S 是指 IDE 设备顺序号（a,b,c...），N 是逻辑分区号（1,2...）。SCSI 设备描述为/dev/sdSN。对于网络文件系统（NFS），该字段定义为 host:dir 的格式。host 是指主机名称，dir 是指主机上的目录。
- 字段二：表示文件系统的挂载点（mount point）。
- 字段三：表示文件系统的类型，如 ext2、vfat 等。
- 字段四：文件系统选项。定义了挂载文件系统时需要指定的特殊选项。
- 字段五：为 dump 程序所用。判断哪个文件系统需要进行 dump。
- 字段六：为 fsck 程序所用。用来确定系统重启时，对文件系统进行检查的顺序。

文件系统挂载完成后，就可以正式使用了。Linux 文件系统采用多级目录的树型层次结构对文件进行管理。树型结构的最上层是根目录，用“/”表示。其他的所有目录都是属于根目录的子结点。系统在安装时将创建默认的目录，而每个目录都有其特殊的功能，用户不能随意修改和删除。Windows 系统的文件系统也采用树型结构，不过 Windows 文件系统的树型结构的根是磁盘分区的盘符。

有几个分区就有几个树型结构，它们之间的关系是并列的。但在 Linux 中，无论操作系统管理多少个逻辑磁盘分区，目录树只有以“/”为根的一棵树。其他文件系统都将通过挂载的方式，接入到根文件系统的指定目录上。典型的 Linux 文件系统结构如图 9-2 所示。

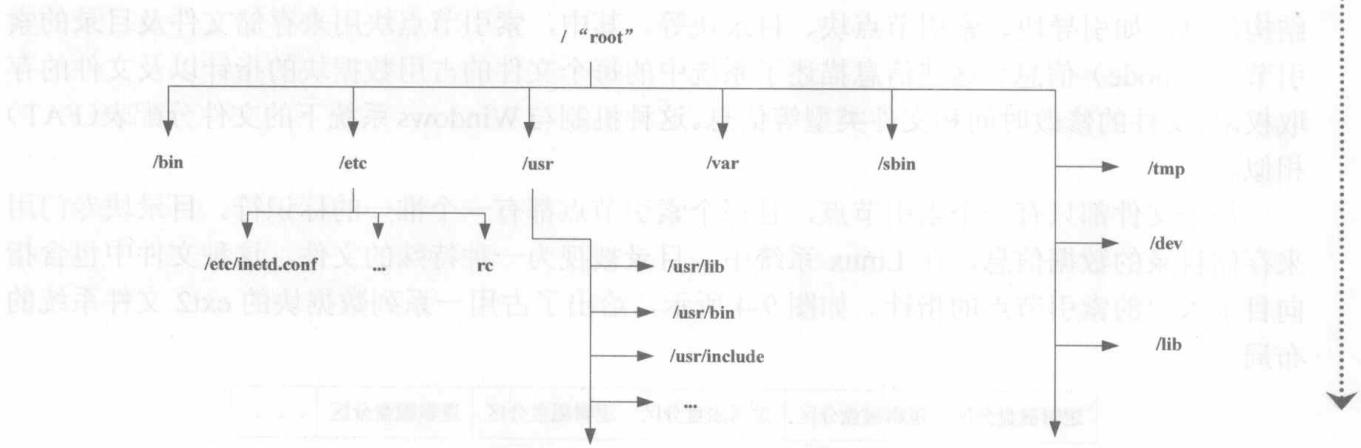


图 9-2　典型的 Linux 文件系统结构

9.1.3　虚拟文件系统

Linux 系统的一个重要特征是支持多种不同的文件系统，如 FAT、ext2、ext3、MINIX、MSDOS 等。为此，必须使用一种统一的接口对文件系统进行管理，这就是虚拟文件系统（VFS）。通过 VFS 将不同文件系统的实现细节隐藏起来，从外部看上去，所有的文件系统都是一样的。也就是说，VFS 是建立在具体文件系统之上的，它为用户程序提供一个统一的、抽象的、虚拟的文件系统界面。VFS 与实际文件系统的关系如图 9-3 所示。

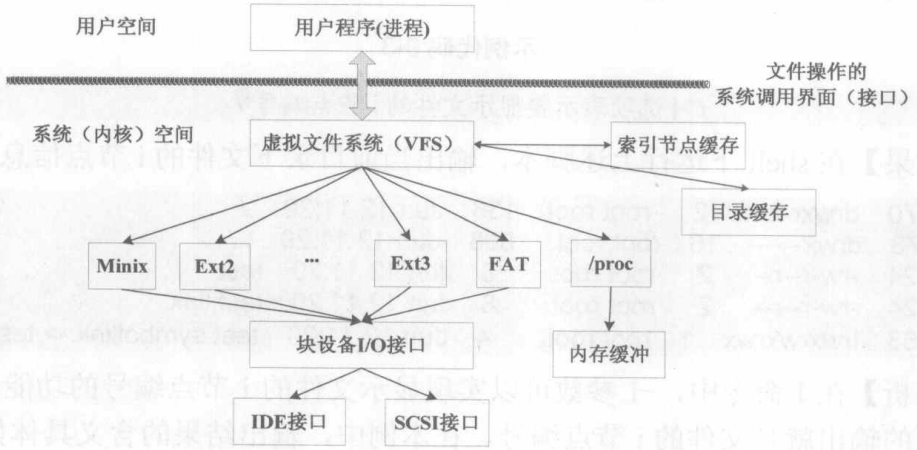


图 9-3　VFS 与实际文件系统的关系

9.1.4　ext2 文件系统

ext2 即第二代扩展文件系统。它是在 ext（Extended File System）文件系统的基础上发展起来的。它弥补了 ext 文件系统在性能方面的一些不足，是 Linux 系统下主流的文件系统。与 ext 文件系统相比，ext2 的特点是存取文件性能好，并且比 ext 文件系统更加稳定。

ext2 文件系统中的数据是以数据块的方式存储的。每个数据块大小相同，其值在文件系统创建时被指定。除了用于保存文件数据的数据块外，也有部分存储区用来保存文件系统的

结构信息，如引导块、索引节点块、目录块等。其中，索引节点块用来存储文件及目录的索引节点（inode）信息。这些信息描述了系统中的每个文件的占用数据块的指针以及文件的存取权限、文件的修改时间和文件类型等信息。这种机制与 Windows 系统下的文件分配表（FAT）相似。

每个文件都只有一个索引节点，且每个索引节点都有一个惟一的标识符。目录块专门用来存储目录的数据信息。在 Linux 系统中，目录被视为一种特殊的文件。这种文件中包含指向目录入口的索引节点的指针。如图 9-4 所示，给出了占用一系列数据块的 ext2 文件系统的布局。

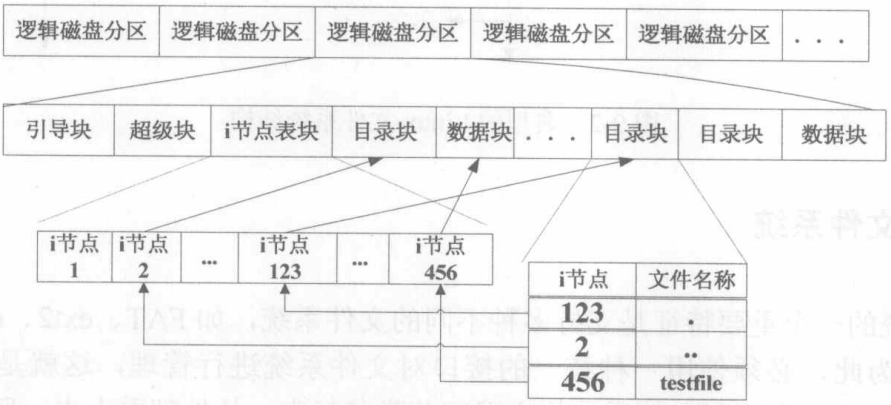


图 9-4　ext2 文件系统的布局

【范例 9-3】查看文件的 inode 编号。其实现过程如示例代码 9-3 所示。

示例代码 9-3

```
1    #l –i                    /*-i 选项表示要显示文件的 i 节点编号*/
```

【运行结果】在 shell 下运行上述脚本，输出当前目录下文件的 i 节点信息，如下所示。

```
1    183370   drwxr-xr-x   2    root root   136   Jun 12 11:20   ./
2      9373   drwx------   16   root root   808   Jun 12 11:20   ../
3    183424   -rw-r--r--   2    root root     8   Jun 12 11:20   test
4    183424   -rw-r--r--   2    root root     8   Jun 12 11:20   test.link
5    183363   lrwxrwxrwx   1    root root     4   Jun 12 11:20   test.symbollink -> test /*符号链接*/
```

【代码解析】在 1 命令中，-i 参数可以实现显示文件的 i 节点编号的功能。在命令的输出中，第一列的输出就是文件的 i 节点编号。在本例中，输出结果的含义具体如下。

- 第 1～2 行：表示当前目录及父目录。
- 第 3～4 行：文件 test 及 test.lnk 的文件信息。可以看到，两个文件的 i 节点编号 183424 是相同的，这是由于 test.lnk 是 test 的链接文件。通过链接方式产生的文件与被链接文件的 i 节点编号是相同的。在文件被链接后，该文件的引用计数会加 1。关于文件类型的更多信息将在介绍文件类型时详细说明。
- 第 5 行：文件 test.symbollink 的文件信息。从标志“->”中可以看到这是一个符号链接文件。它的 i 节点编号 183363 与被链接文件 test 是不同的。

ext2 的超级块中存储的是文件系统的各种参数，包括数据块大小、空闲索引节点数目等。索引节点块则包括了文件系统中所有文件的索引节点信息。其中，每个索引节点存储的内容

包括文件类型及权限、文件大小、时间戳信息及数据块的指针。

在 Linux 中，索引节点中的数据块指针有 4 种：直接指针、一级间接指针、二级间接指针、三级间接指针。直接指针所指向的盘块中放有该文件的数据，这种盘块称为直接块。而一级间接指针所指向的盘块（间接块）中放有直接块的块号表。为了通过间接块存放文件数据，核心必须先读出间接块，找到相应的直接块项，然后从直接块中读取数据。二级间接指针所指向的盘块中放有一级间接块号表。同样，三级间接指针所指向的盘块中放有二级间接块号表。

在 Linux 的索引节点中，共有 12 个直接指针及 3 种间接指针各 1 个。因此，利用前 12 个直接指针可以存放的文件不超过 12KB。如果文件大小超过 12KB，则可以用一级间接指针；若更大，则可用二级间接指针；以此类推，使用三级间接指针最大的文件可达 16GB。

9.1.5　文件类型

谈到文件类型，读者可能立刻联想到 Windows 环境下的文件类型，如常见的.txt、.doc、.mp3、.exe 等，根据文件的后缀就能判断文件的类型。但是，在 Linux 系统中，文件的扩展名与文件类型没有必然的联系，文件类型主要与文件的属性有关。虽然如此，读者自行创建的文件，如果有必要还是可以加上扩展名，这样做的目的仅仅是为了在应用时方便。在 Linux 系统下，查看文件类型可以直接通过执行 file 命令完成，也可以通过 ls-lh 来查看某个文件的属性，从而获得文件的类型信息。

Linux 文件类型常见的有普通文件、目录文件、设备文件、链接文件、管道文件等。

- 普通文件：最为常见的文件类型，其特点是不包含有文件系统的结构信息。通常所接触到的文件，包括图形文件、数据文件、文档文件、声音文件等都属于这种文件。
- 目录文件：目录文件是用于存放文件名及其相关信息的文件，是内核组织文件系统的基本节点。目录文件可以包含下一级目录文件或普通文件。
- 设备文件：Linux 中较为特殊的文件。正是由于它的存在，使得 Linux 系统可以十分方便地访问外部设备。Linux 系统为外部设备提供一种标准接口，将外部设备视为一种特殊的文件。通常情况下，Linux 系统将设备文件存放在/dev 目录下，设备文件使用设备的主设备号和次设备号来指定某外部设备。
- 链接文件：链接文件是一种特殊的文件，实际上是指向一个真实存在的文件的链接。这有点类似于 Windows 下的快捷方式。
- 管道文件（Named Pipes/FIFO）：管道文件是 Linux 系统中另一种很特殊的文件，主要用于不同进程间的信息传递。

【范例 9-4】查看文件类型。其实现过程如示例代码 9-4 所示。

示例代码 9-4

```
1    # l /etc /dev/hda ./          /*显示/etc 目录、/dev/had 文件和当前目录的信息*/
```

【运行结果】在 shell 下运行上述脚本，其结果如下所示。

```
1    -rw-r--r--      1   root   root   2087 Dec 8 2006          /etc/passwd
```

```
2    drwxrwxrwx  96  root  root  9.0K 2008-06-06 15:34  xinetd.d
3    brw-rw----   1  root  disk  3, 0  Jul  1 2004      /dev/had
4    lrwxrwxrwx   1  root  root  11    2008-06-10 16:26  passwd -> /etc/passwd
5    prw-r--r--   1  root  root  0     Jun 10 16:37      testpipe
```

【代码解析】在此命令的输出中，从第一列开始依次显示的信息是文件权限信息、文件引用计数、文件的属主、文件的组、文件大小、文件修改日期、文件路径。在本例中，输出结果的含义具体如下。

- 第 1 行：普通文件，Linux 环境下的用户信息文件，用符号-代表。
- 第 2 行：目录文件，Linux 的网络服务配置目录，用字母 d 代表。
- 第 3 行：块设备文件，第一块 IDE 磁盘的设备文件，用字母 b 代表。
- 第 4 行：符号链接文件，用户信息文件的符号链接文件，用字母 l 代表。
- 第 5 行：管道文件，用于测试的管道文件，用字母 p 代表。

提示：在 Linux 系统中，文件链接分为硬链接及符号链接两种。所谓硬链接，就是链接文件与被链接文件物理上是同一个文件。对文件进行硬链接后，只是增加了文件的引用计数，并没有物理上增加文件。硬链接是通过执行命令 ln 完成的。符号链接是一个物理上真实存在的文件，但是该文件的内容是指向被链接文件的指针。符号链接文件与原文件的 i 节点编号是不同的。符号链接通过执行命令 ln –s 完成。符号链接与 Windows 环境下的快捷方式非常相似。

9.1.6 文件权限管理

文件权限，是指对文件或者目录的访问权限，包括对文件的读、写、删除、执行。在 Windows 操作系统中，文件的权限管理也包括只读、隐藏等属性设置。而在 Windows 环境下的 NTFS 文件系统中，则更进一步，允许针对不同用户对文件设置不同的访问权限。Linux 的文件权限管理比 Windows 更为复杂，安全机制也更为完善。

Linux 系统下文件权限管理包括两种不同的类型：一是传统方式的文件权限管理，也称作 UGO（User Group Other）方式；一是访问控制列表方式的权限管理，即 ACL（Access Control List）方式。传统方式的文件权限管理在 Linux 中已沿用多年，灵活性上不如访问控制列表方式。访问控制列表为计算机提供更好的访问控制机制，它的作用是限制包括 root 用户在内的所有用户对文件、资源或者套接字的访问。下面将分别为读者介绍这两种权限管理方式。

1．UGO 方式

Linux 是多用户操作系统，系统允许多个用户同时登录和工作。因此，Linux 下的每一个文件必须严格地归属于一个用户和一个组。同时，在 Linux 中的每一个文件或目录都包含有访问权限信息。这些访问权限决定了谁能访问和如何访问这些文件和目录。

UGO 方式的权限管理主要是通过文件的 9 个权限控制位对文件访问进行控制的。这 9 个权限位可以划分为 3 组：文件拥有者（File Owner）的权限、与文件归属的用户组（Group）同组用户的权限及其他用户（Other）的访问权限。对于每一组权限位，又可以划分为 3 种不同的权限：读、写及执行，分别用字母 r、w、x 表示。在传统的文件权限管理模式下，任何文件或者目录都包含有这 9 个权限位。

【范例 9-5】输出当前目录下文件的权限信息。其实现过程如示例代码 9-5 所示。

示例代码 9-5

```
1    [root@localhost ~]# l
```

【运行结果】在 shell 下运行上述脚本，列出文件的权限信息如下所示。

```
1  -rw-r--r--    1 root    root        6766 Nov  4  2004  clamav.conf    /*644*/
2  -rwx--x—x     1 root    root          87 Apr  5  2004  cls*           /*711*/
3  –rwxrwxrwx    1 root    root       57032 Apr  8  2005  config_file*   /*777*/
4  drwxr-xr-x    2 root    root          96 Dec  1  2006  cron.d/        /*755*/
```

【代码解析】文件信息的输出中，第一列的输出就是文件或者目录的权限信息，其中第一位是文件类型，后面的 9 位是上述的文件权限信息位。在本例中，输出结果的含义具体如下。

- 第 1 行：root 用户对于文件 clsmac.conf 具有读写权限，其他用户则只有读权限。
- 第 2 行：root 用户对于文件 cls 有全部权限，而 root 用户组的用户及其他用户只有执行权。
- 第 3 行：root 用户、root 用户组内的用户、其他用户对于文件 config_file 具有全部权限。
- 第 4 行：root 用户对目录 cron.d 有读、写及执行权，其他用户则只有读和执行权限。

提示：Linux 对文件的权限与目录的权限的定义是不同的。文件的读权限是指可以读取文件的内容（通过执行 cat 命令查看），目录的读权限是指可以查看（执行 l 命令）目录所包含的文件列表；文件的写权限是指可以对文件的内容进行修改（执行 vi 命令修改文件内容），而目录的写权限是指可以在目录中新建或者删除文件的权限；文件的执行权限是指可以在 shell 环境下运行该文件，而目录的执行权则是指可以访问该目录（通过执行 cd 命令进入目录）。

在 UGO 方式下，可以通过执行 chmod 命令修改文件的权限信息。在进行文件权限修改时，Linux 系统提供了两种不同的权限表示方式：字母方式及数字方式。字母方式就是上述的读用 r 表示，写用 w 表示，执行用 x 表示。

在用字母方式进行权限修改时，用“+”表示增加权限，用“-”表示删除权限。数字方式则是将 9 位的权限位按照文件拥有者、组内用户及其他用户分组后，每组内的三位权限位转换为八进制数字表示。转换的具体方式是：如果增加该权限，该则位置“1”，否则置“0”，然后每三位为一组，转换为八进制，形成 3 个八进制的权限信息。关于文件权限表示的八进制方式如图 9-5 所示。

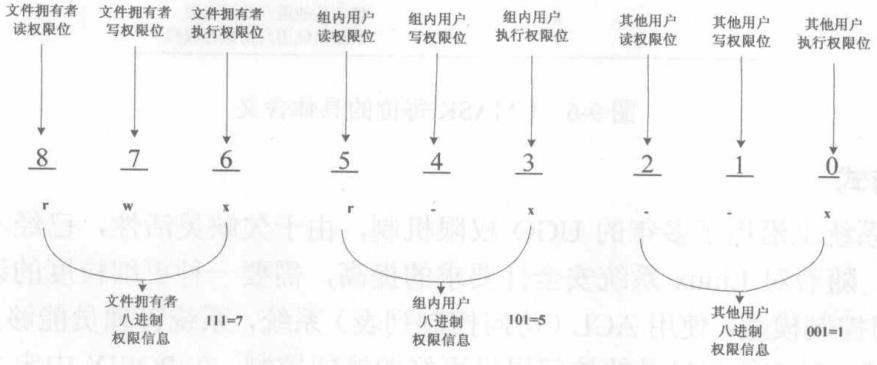


图 9-5　文件权限的八进制表示

【范例 9-6】修改文件 cls 的权限信息，使组内的用户具有读权限。其实现过程如示例代码 9-6 所示。

示例代码 9-6

```
1    [root@localhost ~]# chmod g+r cls          /*同组用户增加读权限*/
2    [root@localhost ~]# chmod 751 cls          /*此命令与第 1 行命令是等效的*/
```

【运行结果】在 shell 下运行上述脚本。权限修改完成后，通过执行 l 命令输出文件的信息如下所示。

```
1    -rwxr-x--x    1 root    root    87 Apr  5  2004  cls*
```

【代码解析】从文件信息的输出中可以看到 cls 文件的相应权限已被修改。

在本例中，输出结果表示文件 cls 对于 root 组内的用户具有读的权限。而在修改前，是不具备读权限的。

通常，新建文件的权限还与另外一个因素有关，这就是当前用户的文件权限掩码。这个掩码可以通过访问环境变量 UMASK 得到，可以通过执行命令 umask 进行修改。文件权限掩码的作用过程是：在新建文件时，将 UMASK 值按二进制位取反，与指定的文件权限位进行与（AND）操作，以此来决定文件最终的权限。默认方式下，umask 通过八进制数字的方式来定义用户创建文件或目录的权限掩码。

如果要通过字母方式进行修改，则需要定义-S 选项。UMASK 的作用对于文件和目录稍有不同。对于文件来说，UMASK 的设置是在假定文件拥有八进制 666 权限上进行，新建文件的权限就是 666 减去 UMASK 的掩码数值。对于目录来说，UMASK 的设置是在假定文件拥有八进制 777 权限上进行，目录八进制权限 777 减去 UMASK 的掩码数值。UMASK 环境变量一般是在用户相关的 SHELL 的配置文件中进行定义，如用户主目录下的.profile 或者.bash_profile。UMASK 每位的具体含义如图 9-6 所示。

UMASK位	取值	含义
第一位	0	允许文件拥有者任何权限
	4	禁止文件拥有者的读权限
	2	禁止文件拥有者的写权限
	1	禁止文件拥有者的执行权限
第二位	0	允许文件的用户组任何权限
	4	禁止文件的用户组的读权限
	2	禁止文件的用户组的写权限
	1	禁止文件的用户组的执行权限
第三位	0	允许其他用户任何权限
	4	禁止其他用户的读权限
	2	禁止其他用户的写权限
	1	禁止其他用户的执行权限

图 9-6　UMASK 每位的具体含义

2．ACL 方式

在 Linux 系统上沿用了多年的 UGO 权限机制，由于欠缺灵活性，已经不能适应系统安全管理的需要。随着对 Linux 系统安全性要求的提高，需要一种更细粒度的访问控制模型来代替 UGO 访问控制模型。使用 ACL（访问控制列表）系统，系统管理员能够为每个用户（包括 root 用户在内）对文件和目录的访问提供更好的访问控制。在 POSIX 中定义了一种访问控制叫做 POSIX ACL，可以实现基于单独用户的控制。

虽然 2.6 版本的 Linux 内核添加了对 ACL 权限管理的支持，但是在默认状态下，系统使用的权限管理模式仍然是 UGO。为在系统中启用 ACL 权限管理，必须在挂载文件系统时手工指定启用 ACL。在执行 mount 命令挂载文件系统时，通过选项–o acl 可以完成这一功能。为了能够在开机后直接应用 ACL 权限管理方式，可以将这一选项写入文件系统挂载的配置文件/etc/fstab 中。

ACL 可以划分为两种权限模式：一种是最小化的 ACL 模式，在这种模式下，仅仅包含了属主权限项、组用户权限项、其他用户权限项，这与 UGO 模式是相同的；一种是扩展的 ACL 模式，在扩展 ACL 模式下，除了最小代 ACL 的模式外，还包括一个掩码项及多个命名用户权限项或者命名用户组权限项。扩展模式下的 ACL 权限项列表如图 9-7 所示。

权限项名称	宏定义	内容/格式
Owner（属主用户）	ACL_USER_OBJ	user::rwx
Owning group（组内用户）	ACL_GROUP_OBJ	group::rwx
Other（其他用户）	ACL_OTHER	other::rwx
Named user（命名用户）	ACL_USER	user:username:rwx
Named group（命名用户组）	ACL_GROUP	group:groupname:rwx
Mask（权限掩码）	ACL_MASK	mask::rwx

图 9-7　扩展模式下的 ACL 权限项列表

所谓命名用户（Named User）权限，就是指单独为某个用户分配相应权限。同样，命名用户组权限就是单独为某个用户组分配权限。通常情况下，这里所分配的权限是有效的。但是，如果权限项里定义了 mask 项，则有效的 ACL 权限将是所分配的权限与 mask 做与（AND）操作后的权限。例如，假定为用户 testuser 分配了对某个文件的读权限和执行权限，同时又定义了 mask 为读权限及写权限。最终，用户 testuser 对该文件所拥有的权限是只读权限，如图 9-8 所示。

权限项名称	权限信息	权限
Named user（命名用户）	user:testuser:r-x	r - x
Mask（权限掩码）	mask::rw-	r w -
有效权限		r - -

图 9-8　ACL mask 的作用示例

【范例 9-7】创建一个简单的文件系统并采用 ACL 权限管理模式。其实现过程如示例代码 9-7 所示。

示例代码 9-7

```
1    [root@localhost ~]# dd if=/dev/zero of=/opt/testfs count=512   /*用 dd 建立一个空白文件*/
2    [root@localhost ~]# losetup /dev/loop0 /opt/testfs      /*建立循环设备，用于模拟文件系统*/
3    [root@localhost ~]# mke2fs /dev/loop0                  /*在模拟文件系统上建立 ext2 文件系统*/
4    [root@localhost ~]# mount -o rw,acl /dev/loop0 /mnt   /*以可读可写权限挂载文件系统于/mnt 目录*/
```

【运行结果】在 shell 下依次运行上述命令，即可在新建立的 ext2 文件系统上采用 ACL 权限管理模式。

【代码解析】在本例中，各命令的含义具体如下。

- 第 1 行：建立一个 512K 大小的空白文件。

- 第 2 行：建立一个循环设备。循环设备可以把文件虚拟成区块设备，从而模拟一个文件系统。
- 第 3 行：在循环设备上建立 ext2 文件系统。
- 第 4 行：将新建的文件系统挂载到目录/mnt 下，并采用 ACL 权限管理模式。

ACL 权限管理包含 6 种规则，分别是 ACL_USER_OBJ、ACL_GROUP_OBJ、ACL_OTHER、ACL_MASK、ACL_USER、ACL_GROUP，其含义具体如下。

- ACL_USER_OBJ：用户属主权限，对应于 UGO 模式的文件拥有者权限。
- ACL_GROUP_OBJ：用户组属主权限，对应于 UGO 模式的组内用户权限。
- ACL_OTHER：其他用户权限，对应于 UGO 模式的其他用户权限。
- ACL_USER：为某个用户添加的规则。
- ACL_GROUP：为某个组添加的规则。
- ACL_MASK：规定了 ACL_USER、ACL_GROUP 和 ACL_GROUP_OBJ 的最大值。

【范例 9-8】查看文件 cls 的 ACL 权限并单独为用户 testuser 添加读和执行权限。其实现过程如示例代码 9-8 所示。

示例代码 9-8

```
1   linux:/etc # getfacl cls                          /*得到文件 cls 的访问控制列表信息*/
2   # file: cls
3   # owner: root
4   # group: root
5   user::rwx
6   group::--x
7   other::--x
8   linux:/etc # setfacl -m user:testuser:r-x cls     /*调用 setfacl 为用户 testuser 设置对文件 cls 的
    权限*/
9   linux:/etc # getfacl cls                          /*调用 getfacl 获取文件 cls 的权限*/
10  # file: cls
11  # owner: root
12  # group: root
13  user::rwx
14  user: testuser:r-x
15  group::--x
16  mask::r-x
17  other::--x
```

【运行结果】在 shell 下依次运行第 1 行、第 8 行、第 9 行命令，即可在文件 cls 上为用户 testuser 赋予读和执行的权限。

【代码解析】在本例中，各输出信息的含义具体如下。

- 第 1 行：查看文件 cls 的 ACL 权限信息。
- 第 2～4 行：分别指示文件名称、文件的拥有者、文件的用户组信息。
- 第 5～7 行：分别指示文件的拥有者权限、组内用户权限、其他用户权限。
- 第 8 行：修改文件 cls 的 ACL 信息，为用户 testuser 增加读的执行权限。
- 第 14 行：用户 testuser 已经具有了单独的权限，即对文件 cls 的读和执行权。

9.2　文件基本操作

Linux 系统下文件操作的编程方法主要有两种：原始文件 I/O 编程模式及标准 I/O 库编程模式。原始文件 I/O 模式是指 Linux 系统提供的文件编程 API，是一种不带缓存的文件 I/O。与 Windows 环境下的编程 API 类似，是一种底层的文件操作系统调用。

所谓不带缓存是指对每个文件操作函数都调用系统内核的一个系统调用，标准 I/O 库则是带缓存的文件 I/O。标准 I/O 代替用户进行许多缓存细节方面的处理，如缓存的分配、执行 I/O 操作时优化缓存长度等。关于标准 I/O 编程，将在下一章中为读者详细介绍。

9.2.1　文件编程的基本概念

在详细介绍文件编程之前，本小节将介绍几个和文件编程相关的重要概念。了解这些概念将是用户进行文件编程的重要基础，希望用户可以认真掌握。

1．文件描述符

文件描述符在文件编程中是一个很重要的概念。它是一个非负整数，其取值范围是 0～OPENMAX。当进程创建一个新文件或者打开现有的文件时，系统内核将向进程返回一个文件描述符。文件描述符在进程范围内是惟一的，并且每个进程可以同时打开的文件数目不能大于 OPENMAX。OPENMAX 是一个宏定义，它的取值在不同版本的 Linux 中是不同的。可以通过执行命令 ulimit –n 获取它的当前值。当对文件进行 I/O 操作时，大多数函数都用文件描述符作为参数，代表要操作的文件。

2．标准输入、标准输出与标准错误输出

在 Linux 进程启动时，内核默认为每个进程打开 3 个文件：标准输入文件、标准输出文件、标准错误输出文件。同样，这 3 个文件也分配了文件描述符，分别是 0、1、2。标准输入文件被映射至键盘，而标准输出文件及标准错误输出文件则被映射至监视器，也就是计算机屏幕。对于这 3 个已由内核自动打开的文件描述符，在程序中可以直接引用。引用的方式可以是直接用数字，也可以使用它们的宏定义：STDIN_FILENO、STDOUT_FILENO、STDERR_FILENO。

3．设置-用户-ID 与设置-组-ID

每个文件都有自己的文件属主（用户 ID）及组（用户组 ID）。如果这个文件是可执行的程序，那么当该文件被执行时就会在操作系统中形成进程。每个进程都有一个有效用户 ID 和一个有效组 ID。这两个与进程相关联的 ID 的作用是进行文件存取许可检查。除此之外，每个进程还有另外两个 ID，分别称作实际用户 ID 和实际组 ID。这两个 ID 就是执行该程序用户的用户 ID 及组 ID。

在通常情况下，进程是有效用户 ID 就是实际用户 ID，进程的有效组 ID 就是实际组 ID。但是，Linux 提供了一种特殊的机制，可以在文件的属主权限中设置一个标志。这个标志的作用是：当执行该文件时，将进程的有效用户 ID 设置为该文件所有者的用户 ID。

如果设置了该标志，那么当一个文件被执行时，其有效用户 ID（用于文件存取许可检查）不再是调起该进程的用户，而是该文件的属主 ID。这种机制就称作设置-用户-ID，也称作是 suid。同样，对于组的权限也可以设置这样的标志，称作是设置-组-ID，也就是 sgid。suid 可以通过执行命令 chmod u+s 完成。同样，sgid 则通过执行命令 chmod g+s 完成。

下面向读者介绍一下 passwd 命令的工作原理。如果读者了解了 passwd 的工作机制，就会对 suid 有更深刻的了解。在 Linux 操作系统下，用户的口令文件是最重要的数据文件之一。该文件存放于/etc/passwd 及/etc/shadow 中。通过执行 l 命令可以看到文件/etc/shadow 的存取权限是：-rw-r----- 1 root shadow 425 2008-06-27 13:57 /etc/shadow。可以看到，文件对于 root 用户是可以写的，而对于其他所有用户都是只读或者无权限的。这说明，每个普通用户是无权对 shadow 文件进行写操作的。

那么，在普通用户通过执行 passwd 命令修改口令时，却是实实在在地向 shadow 文件写入数据的。这是如何实现的呢？这就是 suid 作用的结果。原来，/usr/bin/passwd 这个命令设置了 suid 权限：-rwsr-xr-x 1 root shadow 86218 2004-06-30 15:09 /usr/bin/passwd。请读者注意 passwd 命令的属主权限最后一位是 s。

这个 s 就是 suid 标志。如果这个 s 是在组权限上，就是 sgid 的标志。而 passwd 命令文件的属主是 root。也就是说，普通用户在通过执行 passwd 命令修改口令时，其有效用户 ID 会变成 root，所以可以完成对口令文件的修改。

9.2.2 文件的创建与打开

文件操作最基本的部分就是文件的创建，它是其他文件操作的基础。用文件创建的函数有多个如 creat、open、mknod、mkdir 等。mknod 用于设备文件的创建，mkdir 用于目录文件的创建。本节只介绍 creat 及 open 系统调用，其他内容在后续章节中将陆续向读者予以介绍。

1. creat 函数

Creat 函数用来创建一个普通文件其函数原型如下所示。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int creat(const char *pathname, mode_t mode );
```

（1）参数说明如下。

- pathname：要创建文件的路径，可以使用绝对路径，也可以使用相对路径。
- mode：新建文件的访问权限。文件访问权限的定义在头文件<sys/stat.h>中，如图 9-9 所示。

（2）返回值说明如下。

- 若成功，返回 int 型的文件描述符。
- 若出错，返回-1。

mode值	意义
S_IRUSR	文件拥有者-读
S_IWUSR	文件拥有者-写
S_IXUSR	文件拥有者-执行
S_IRGRP	组内用户-读
S_IWGRP	组内用户-写
S_IXGRP	组内用户-执行
S_IROTH	其他用户-读
S_IWOTH	其他用户-写
S_IXOTH	其他用户-执行

图 9-9 文件访问权限的定义

（3）在使用 creat 系统调用时，应该注意以下几点。

- 要在一个目录中新建文件，必须要有对该目录的写权限及执行权限。
- 如果文件已存在，则调用 creat 的进程应该对该文件有写权限及执行权限。creat 成功后，原文件内容被清除，而文件归属的用户 ID、组 ID 及权限信息保留。
- 如果文件不存在，则新生成文件的用户 ID 及组 ID 分别是当前进程的有效用户 ID 及有效组 ID。
- creat 调用成功后，文件以只写方式打开。如果在文件操作中需要对文件进行读操作，应先关闭文件后，再用 open 打开文件进行操作。

【范例 9-9】使用 creat 在当前目录下创建一个文件拥有者可读权限的文件，并在文件中写入文本串。其实现过程如示例代码 9-9 所示。

示例代码 9-9

```
1   #include <sys/stat.h>                                   /*头文件*/
2   #include <sys/types.h>
3   #include <fcntl.h>
4   #include <unistd.h>
5   #include <errno.h>                                      /*头文件，包含全局变量 errno 的定义*/
6   main()
7   {
8   int fd;                                                 /*整型变量，用于保存文件描述符*/
9   char szErrorBuffer[256];                                /*字符串数组，用于保存错误信息*/
10  fd = creat("./HelloWorld.txt", S_IRUSR | S_IWUSR); /*调用 creat 创建新文件，使文件属主有读
    写权限*/
11  if (fd == -1)                  /*创建失败，调用函数 strerror 格式化 errno 得到详细的错误信息*/
12  {
13      memset(szErrorBuffer,0x00,sizeof(szErrorBuffer));          /*清空缓冲区*/
14      sprintf(szErrorBuffer,"creat:%s\n",(char *)strerror(errno));  /*向缓冲区中输出错误信息*/
15      write(STDERR_FILENO,szErrorBuffer,strlen(szErrorBuffer));    /*向标准输出设备输出错
    误信息*/
16      return;
17  }
18  write(fd, "Hello world!\n", 13);                         /*向新建立的文件中写入 HelloWorld*/
19  close(fd);                                              /*关闭文件*/
20  }
```

【运行结果】经过编译、链接，在命令提示符下执行程序，可以看到当前目录下已成功新建文件 HelloWorld.txt。

【代码解析】在本例中，各程序行的具体含义如下。

- 第 1～5 行：包含头文件。程序中用到的 STDERR_FILENO、S_IRUSR 等都在其中定义。
- 第 6 行：main 函数。每个可执行的 C 程序中必须包含并且保证惟一的函数。
- 第 8 行：文件描述符变量定义。定义了一个 int 型的变量，用于保存 creat 返回的文件描述符。
- 第 9 行：错误输出缓冲区定义。定义了一个字符串数组，在 creat 失败时用于输出错误信息。

- 第 10 行：创建文件。通过调用 creat 创建一个名为 HelloWorld.txt 的文件，并设置读写权限。
- 第 11 行：创建文件失败。creat 返回-1，表明创建文件出现了错误。
- 第 13 行：清空错误输出缓冲区。局部变量定义后，在使用前要注意初始化。
- 第 14 行：格式化错误信息至错误输出缓冲区。errno 是系统内核变量，可以在程序中直接引用。
- 第 15～16 行：向标准错误输出写错误信息并返回。
- 第 18 行：向成功建立的文件中写入字符串 Hello world!。关于 write 将在后面章节中介绍。
- 第 19 行：关闭打开的文件。在程序退出前，必须关闭已打开的文件描述符。

读者在程序设计中，要注意 memcpy（memset、memcmp）与 strcpy（strcmp）的区别。memcpy 系统函数是以内存中的通用类型为操作对象，其中每一个函数都指定了要操作的内存区的长度。而 strcpy 系统函数是以内存中的字符串为操作对象，而字符串的结束符是“\0”。strcpy 系列函数在遇到字符串结束符时将结束后续操作。

读者从上面的例子中可以看到 Linux 系统下的一种错误处理方法：使用 errno 变量。在系统调用出现错误时，内核将把错误值写入到变量 errno 中。用户的应用程序可以从这个变量中获取错误的原因。程序中使用 errno 时，必须首先包含<errno.h>。注意在系统调用返回失败时，必须紧接着引用 errno 变量，避免程序中下一条系统调用时内核覆盖 errno 的值。errno 的值可以进行格式化输出。Linux 系统提供了两种方法，分别是 perror 及 strerror。perror 是将错误信息输出到终端，而 strerror 则是将错误信息输出到缓冲区。

2．open 函数

Open 函数用来打开一个现有的文件或者创建一个文件，其函数原型如下所示。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char * pathname, int oflag,.../*, mode_t mode * / ) ;
```

（1）参数说明如下。

- pathname：要创建文件的路径，可以使用绝对路径，也可以使用相对路径。
- oflag：打开文件的方式。可以由多个方式进行组合。常用的打开文件方式的定义如图 9-10 所示。

打开方式	意义	可选
O_RDONLY	只读打开	三选一
O_WRONLY	只写打开	三选一
O_RDWR	读写打开	三选一
O_APPEND	追加方式打开	可选
O_CREAT	文件不存在则创建	可选
O_EXCL	如定义了O_CREAT,且文件已存在,则出错	可选
O_TRUNC	如文件存在,且以只读或只写打开,则截断文件	可选

图 9-10　常见的文件打开方式

- mode：可选参数，只在创建新文件时有效。用于定义新建文件的访问权限。

（2）返回值说明如下。

若成功，返回 int 型的文件描述符；若出错，返回-1。文件打开失败错误描述如图 9-11 所示。

错误代码	解释
EEXIST	参数pathname 所指的文件已存在,却使用了O_CREAT和O_EXCL标志
EACCESS	参数pathname所指的文件不符合所要求的权限
EROFS	欲分配写入权限的文件存在于只读文件系统内
EFAULT	参数pathname指针超出可存取内存空间
EINVAL	参数mode 不正确
ENAMETOOLONG	参数pathname过长
ENOTDIR	参数pathname不是目录
ENOMEM	核心内存不足
ELOOP	参数pathname有过多符号连接问题
EIO I/O	存取错误

图 9-11　文件打开失败错误描述

（3）在使用 open 系统调用时，应该注意以下几点。

- 文件打开方式参数应在 O_RDONLY、O_WRONLY、O_RDWR 中选择一个，然后通过|与其他的选取参数进行组合。
- 如果调用 open 函数创建新文件，则用户掩码（见 9.1.6 节文件权限中对 umask 的说明）会影响到被创建文件的访问权限。open 调用里给出的模式值将与当时的用户掩码 umask 取反做 AND 操作。例如，如果用户掩码被设置为 001，并且在调用 open 时指定了 S_IXOTH（其他用户具有可执行权限）模式标志，那么实际结果是其他用户对创建的文件不会拥有执行权限。
- 可以利用 O_EXCL 参数的特殊性质，采用打开方式参数组合 O_CREAT|O_EXCL 实现互斥访问。While((fd=open(path,O_WRONLY|O_CREAT|O_EXCL,0666)) == -1 && errno == EEXIST){代码段}。从 O_EXCL 的性质知道，在文件已存在的情况下，creat 系统调用将返回失败。所以一旦有进程 A 调用该代码，其他进程将不能访问代码段的内容，直至进程 A 将文件 path 删除。
- 打开方式参数为 O_WRONLY|O_CREAT|O_TRUNC 的 open 调用与 creat 系统调用是等价的。

【范例 9-10】使用 open 打开当前目录下的文件 HelloWorld.txt 文件，并在文件尾部追加字符串。其实现过程如示例代码 9-10 所示。

示例代码 9-10

```
#include <sys/stat.h>                                    /*头文件*/
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
main()                                                   /*主函数*/
{
int fd;                                                  /*文件描述符*/
char szErrorBuffer[256];                                 /*错误输出缓冲区*/
```

```
10  fd = open("./HelloWorld.txt", O_WRONLY|O_APPEND);          /*以追加的方式打开文件*/
11  if (fd == -1)                                               /*错误处理*/
12  {
13      memset(szErrorBuffer,0x00,sizeof(szErrorBuffer));       /*清空错误输出缓冲区*/
14      sprintf(szErrorBuffer,"open:%s\n",(char *)strerror(errno));  /*将格式化的错误信息复
    制到缓冲区*/
15      write(STDERR_FILENO,szErrorBuffer,strlen(szErrorBuffer));  /*输出错误信息*/
16      return;
17  }
18  write(fd, "This is a example for open.\n", 28);             /*向文件尾部附加数据*/
19  close(fd);                                                  /*关闭文件*/
20  }
```

【运行结果】经过编译、链接，在命令提示符下执行程序，可以看到文件 hello.txt 尾部已追加字符串。

【代码解析】在本例中，各程序行的具体含义见示例 9-9 的说明。第 10 行表示调用 open 打开一个文件，并定位文件读写指针至文件尾（O_APPEND）。

提示：open 函数本身可以提供创建新文件的功能。之所以单独提供 creat 系统调用基于如下原因：在早期的系统中，open 函数未提供创建新文件的功能，因此需要另外提供 creat 来实现新建文件的功能。现在的 open 函数可以通过参数 O_CREAT 及 O_TRUNC 实现文件创建，所以 creat 完全可以由 open 来替代。

9.2.3 文件的读写

在对文件的各种操作中，读写操作是最为常见，也是使用频率最高的功能之一。读文件是指从文件中获取数据，写文件是指向文件中写入数据。文件的读写操作涉及文件系统的方方面面，如文件的权限、读写的控制及效率等。在本节中，着重向读者介绍读写操作的系统调用。

1．read 函数

read 函数用于从一个打开的文件中读取数据，其函数原型如下所示。

```
#include <unistd.h>
ssize_t read(int filedes, void *buff, size_t nbytes);
```

（1）参数说明如下。

- filedes：文件描述符。
- buff：缓冲区，用于缓存 read 从文件中读取的数据。
- nbytes：要读取的字节数。

（2）返回值说明如下。

- 若成功，返回实际读取的字节数。如果已读到文件尾，则返回 0。
- 若出错，返回-1。

（3）在使用 read 系统调用时，应注意以下几点。

- read 系统调用不仅仅应用于读取文件，而且用于读取套接字及设备文件。
- 如果 read 在读到要求的字节数（由参数 nbytes 指定）前到达了文件末尾，则将返回

已读取到的数据。此时，返回值小于 nbytes 参数值。

- read 操作将自动增加文件的读写指针。在文件刚打开时，文件夹的读写指针位于文件开始处（除非指定 O_APPEND 标志）。

【范例 9-11】使用 read 读取当前目录下的文件 HelloWorld.txt 的内容，并显示在屏幕上。其实现过程如示例代码 9-11 所示。

示例代码 9-11

```
#include <sys/stat.h>                          /*头文件*/
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#define READ_LEN 100                           /*宏定义，定义了每次读取的数据长度*/
main()
{
int fd,n;                                      /*fd 是文件描述符，n 用于保存读取到的数据长度*/
char szErrorBuffer[256];                           /*错误输出缓冲区*/
char szReadBuffer[READ_LEN+1];                     /*从文件中读取的数据缓冲区*/
fd = open("./HelloWorld.txt", O_RDONLY);           /*以只读的方式打开文件*/
if (fd == -1)                                      /*打开文件失败的错误处理*/
{
    memset(szErrorBuffer,0x00,sizeof(szErrorBuffer));                  /*清空错误输出缓冲区*/
    sprintf(szErrorBuffer,"open:%s\n",(char *)strerror(errno));        /*将格式化后的错误信息复制到缓存*/
    write(STDERR_FILENO,szErrorBuffer,strlen(szErrorBuffer));          /*向错误输出设备输出错误信息*/
    return;
}
memset(szReadBuffer,0x00,sizeof(szReadBuffer));                        /*清空缓冲区*/
n = read(fd,szReadBuffer,READ_LEN);            /*调用 read 从文件中读取 READ_LEN 个字节*/
if(n == -1)                                                            /*读取数据失败的错误处理*/
{
    memset(szErrorBuffer,0x00,sizeof(szErrorBuffer));                  /*清空错误输出缓冲区*/
    sprintf(szErrorBuffer,"read:%s\n",(char *)strerror(errno)); /*将格式化后的错误信息复制到缓存*/
    write(STDERR_FILENO,szErrorBuffer,strlen(szErrorBuffer));          /*向错误输出设备输出错误信息*/
}
else if( n != READ_LEN)                                                /*读取到文件尾的处理*/
{
   write(STDOUT_FILENO,szReadBuffer,n);                                /*输出读取到的数据*/
   write(STDERR_FILENO,"Reached tail of file!\n",22);                   /*到达文件尾部*/
}
else                                                                   /*正常读取数据的处理*/
{
   write(STDOUT_FILENO,szReadBuffer,n);                                /*输出读取到的数据*/
}
close(fd);                                                             /*关闭文件*/
}
```

【运行结果】经过编译、链接，在命令提示符下执行程序，输出如下所示。

```
Hello world!
```

```
2    This is a example for open.
3    Reached tail of file!
```

【代码解析】在本例中，各程序行的含义具体如下。

- 第 12 行：调用 open 以只读方式打开文件。
- 第 13～19 行：打开文件失败后的异常处理。
- 第 20～21 行：从文件中读取指定长度的数据。
- 第 22～27 行：write 调用失败后的异常处理。
- 第 28～32 行：实际读取的长度与要求的长度不等，表明已到达文件末尾。
- 第 33～36 行：成功读取到指定长度的数据至缓冲区。
- 第 37 行：关闭打开的文件描述符。

程序的输出中具体含义如下。

- 第 1～2 行：从文件中读取到的数据。
- 第 3 行：表明已经到达文件末尾的提示信息（见源程序的第 31 行）。

读者在实际编程过程中，应注意 sizeof 的使用方法。特别是注意区别操作数是字符数组和字符指针的情况。大家知道，sizeof 是 C 语言中的一种单目操作符，用于获取操作数的存储空间大小。空间大小的单位为字节。对于一个字符数组来说，调用 sizeof 的结果是数组空间的尺寸。而对于字符指针来说，用 sizeof 获取的是指针本身的尺寸，这个值通常情况下是 4。例如，字符数组 char a[]="123456"，而字符指针 char *p="123456"。在此情况下，sizeof(a) 的结果是 7。而 sizeof(p)的结果是 4。

提示：sizeof 与 strlen 是不同的。sizeof 是获取缓冲区的长度的运算符，即使缓冲区无任何数据（全为空\0），sizeof 也可以取得其长度。而 strlen 是取得字符串的长度的函数，注意字符串是以'\0'为结束符的。strlen 在计算长度时不会包含结束符本身，而 sizeof 则包括了'\0'的长度。

2．write 函数

write 函数用于向一个打开的文件写入数据。在前面的示例中，已调用过该函数进行文件的写操作，其函数原型如下所示。

```
#include <unistd.h>
ssize_t write(int filedes, const void *buff, size_t nbytes);
```

（1）参数说明如下。

- filedes：文件描述符。
- buff：缓冲区，准备写入文件的数据。
- nbytes：要写入的字节数。

（2）返回值说明如下。

- 若成功，返回实际写入的字节数。
- 若出错，返回-1。出错常见原因是磁盘已满或者已超过文件最大长度的限制。

（3）在使用 write 系统调用时，应注意以下几点。

- 对于普通文件，写操作从当前的文件读写指针指示的位置开始。如果打开文件时，指定了 O_APPEND 选项，则将从文件末尾开始写。
- 与 read 函数一样，write 不仅用于文件的写操作，同时设备文件及套接口的写操作

也是由 write 完成的。

9.2.4　文件的关闭与删除

打开的文件在进程退出前，必须进行显式关闭。否则，内核将自动关闭这些文件。某些情况下，打开文件后未关闭会造成文件的内容丢失。多数情况下，不关闭文件可能不影响程序的运行。尽管如此，还是应该养成关闭打开的文件的编程习惯。因为打开的文件实际上在系统内核占用系统资源。如果不关闭，会造成系统的可用资源的下降。在 Linux 系统中，关闭文件的函数是 close，删除文件的函数是 unlink。

1．close 函数

在文件描述符使用完毕后，程序中应调用 close 显式关闭文件描述符。close 将解除文件描述符与内核数据结构的关联，同时释放相关资源。close 函数的调用方式参见文件的创建及打开例程。close 函数的原型如下所示。

```
#include <unistd.h>
int close (int filedes);
```

（1）参数说明如下。

- filedes：要关闭的文件描述符。

（2）返回值说明如下。

- 若成功，返回 0。
- 若出错，返回-1。

提示：在 Linux 系统中，每个进程能够同时打开的文件描述符是有限制的。这个值通过执行命令 ulimit –n 来查询及修改。如果用 ulimit 设置最大值失败，则可能是设置的值超过了系统分配的相关资源的软限制。该限制值在/etc/security/limits.conf 中进行定义。

2．unlink 函数

删除文件只是将文件的引用计数减一。只有在文件的引用计数变为零的情况下，才会物理删除一个文件。unlink 的函数原型如下所示。

```
#include <fcntl.h>
int unlink(const char *path);
```

（1）参数说明如下。

- path：要删除的文件路径。

（2）返回值说明如下。

- 若成功，返回 0。
- 若出错，返回-1。

【范例 9-12】创建一个可以在程序中使用的临时文件。其实现过程如示例代码 9-12 所示。

示例代码 9-12

```
1    #include <sys/stat.h>                    /*头文件*/
2    #include <sys/types.h>
```

```
3   #include <fcntl.h>
4   #include <unistd.h>
5   #include <errno.h>
6   main()                                          /*主函数*/
7   {
8   int fd;                                         /*文件描述符*/
9   int count;                                      /*整型变量，用于保存向文件中写的字节数*/
10  char szErrorBuffer[256];                        /*错误信息缓冲区*/
11  fd = creat("./tmpfile", S_IRUSR | S_IWUSR);     /*创建新文件*/
12  if (fd == -1)                                   /*创建新文件失败时的处理*/
13  {
14      memset(szErrorBuffer,0x00,sizeof(szErrorBuffer));              /*清空错误输出缓冲区*/
15      sprintf(szErrorBuffer,"creat:%s\n",(char *)strerror(errno));   /*向缓冲区中复制格式化
    的错误信息*/
16      write(STDERR_FILENO,szErrorBuffer,strlen(szErrorBuffer));      /*向标准错误输出设备输
    出错误信息*/
17      return;
18  }
19  if(unlink("./tmpfile") == -1)                   /*文件的引用计数减 1，此时文件的引用计数变为 0*/
20  {
21      write(STDERR_FILENO,"Unlink failed!\n",15);   /*向标准错误输出设备输出错误信息*/
22      return;
23  }
24  count = write(fd, "Hello world!", 12);                          /*向文件中写入数据*/
25  if(count < 0)                                                  /*写入失败的处理*/
26      write(STDERR_FILENO,"Write to tmpfile failed.\n",25);/*向标准错误输出设备输出错误信息*/
27  else                                                           /*写入成功，提示相应信息*/
28      write(STDERR_FILENO,"Write to tmpfile succeed.\n",26);     /*向标准错误输出设备输
    出错误信息*/
29  if((fd = open("./tmpfile",O_RDONLY)) == -1)                    /*以只读的方式打开文件./tmpfile*/
30  {
31      write(STDERR_FILENO,"Open tmpfile failed.\n",21);/*向标准错误输出设备输出错误信息*/
32  }
33  }
```

【运行结果】经过编译、链接，在命令提示符下执行程序，输出如下所示。

```
1   Write to tmpfile succeed.
2   Open tmpfile failed.
```

【代码解析】在本例中，首先调用 creat 创建一个名为 tmpfile 的文件，然后接着调用 unlink 将其删除。此时文件 tmpfile 的引用计数已为零，但是在程序中仍然可以进行写操作。这表明该文件要在当前进程退出后，才会由内核将其物理删除。而在进程未退出时，仍然可以通过文件描述符对文件进行操作。在本例中，各程序行的含义具体如下。

- 第 11～18 行：创建一个名为 tmpfile 的文件。
- 第 19～23 行：调用 unlink 删除刚刚创建的名为 tmpfile 的文件。
- 第 24～28 行：测试 tmpfile 的文件描述符是否可以写。
- 第 29～32 行：测试文件 tmpfile 是否可以通过 open 打开。

提示：在前面的章节中曾经介绍过文件链接的概念。在对文件进行硬链接后，文件的 i 节点的引用计数将加一。换句话说，可能存在多个文件名称连接到同一个物理文件（i 节点）。此时，做 unlink 操作并不会真正删除一个文件，而只是将文件引用计数减一。查看文件引用

计数的方法是通过执行 l 命令。命令的输出中，在文件访问权限与文件属主间的项就是文件的引用计数。

9.2.5　文件的随机存取

文件的读写方式分为两种：顺序读写和随机读写。所谓顺序读写，就是文件的读写操作只能从文件头部开始，顺序读取或者写入数据。而随机读写是指在读写文件时，允许移动文件内部的位置指针到需要读写的位置，再进行读写。通常意义上的文件读写，即是指顺序读写。在许多场合会用到随机读写。如打开一个文件在文件的某个指定的位置插入数据等。Linux 内核向用户提供了系统调用 lseek 用于完成这一功能。

在 Linux 系统中，每个被打开文件都有与其相关联的“当前文件位移”。它是一个非负整数，用以度量从文件开始处到文件读写指针位置的字节数。通常读、写操作都从当前文件位移处开始。在读写完成后，使文件位移增加所读或写的字节数。

当打开一个文件时，除非指定 O_APPEND 选项，否则该位移被设置为 0。在指定 O_APPEND 选项时，该位移被设置为文件的长度。lseek 函数的作用就是调整当前文件位移。不过，lseek 仅将当前的文件位移量记录在内核的数据结构中，它并不引起任何 I/O 操作。然后，该位移量用于下一个读或写操作。Lseek 函数的原型如下所示。

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd off_t offset, int whence) ;
```

（1）参数说明如下。

- fd：文件描述符。
- offset：文件偏移量。offset 是个相对偏移量，偏移的基准是由 whence 参数决定的。
- whence：偏移依据。

（2）返回值说明如下。

- 若成功，返回新的当前文件位移。
- 若出错，返回-1。

【范例 9-13】在文件 HelloWorld.txt 中“Hello World!”的后面插入字符串“Insert by lseek.”。其实现过程如示例代码 9-13 所示。

示例代码 9-13

```
#include <unistd.h>                          /*头文件*/
#include <fcntl.h>
#include <sys/types.h>
#include <errno.h>
#define READ_LEN 256                         /*宏定义  定义文件的最大长度*/
#define INSERT_POS 13                        /*宏定义  从文件头部开始算起的插入位置*/
#define INSERT_LEN 17                        /*宏定义  插入的数据长度*/
main()
{
```

```
10  int fd,n;                                          /*文件描述符*/
11  off_t lMoveLength;                                 /*要插入位置的偏移*/
12  char szErrorBuffer[256];                           /*错误信息缓冲区*/
13  char szReadBuffer[READ_LEN+1];                     /*用于保存读取文件内容的缓冲区*/
14  fd = open("./HelloWorld.txt", O_RDWR);             /*以可读可写的方式打开文件*/
15  if (fd == -1)                                      /*打开文件失败的处理*/
16  {
17      memset(szErrorBuffer,0x00,sizeof(szErrorBuffer));          /*清空错误输出缓存*/
18      sprintf(szErrorBuffer,"open:%s\n",(char *)strerror(errno)); /*向错误输出缓存复制格式化后
    的错误信息*/
19      write(STDERR_FILENO,szErrorBuffer,strlen(szErrorBuffer));
20      return;
21  }
22  lMoveLength = lseek(fd,0,SEEK_END) - INSERT_POS;           /*计算要移动的数据长度（从插
    入位置到文件尾）*/
23  lseek(fd,INSERT_POS,SEEK_SET);                     /*移动读写指针到插入位置*/
24  memset(szReadBuffer,0x00,sizeof(szReadBuffer));    /*清除缓冲区*/
25  n = read(fd,szReadBuffer,lMoveLength);             /*读取要移动的数据至缓冲区*/
26  if(n < 0)                                          /*读取数据失败的处理*/
27  {
28      close(fd);                                     /*关闭文件描述符*/
29      return;                                        /*退出*/
30  }
31  lseek(fd,INSERT_LEN+INSERT_POS,SEEK_SET);/*定位文件的读写指针到要移动数据的位置*/
32  write(fd,szReadBuffer,n);                          /*将要移动的数据块写入*/
33  lseek(fd,INSERT_POS,SEEK_SET);                     /*定位文件的读写指针到要插入的位置*/
34  write(fd, "Insert by lseek.\n", INSERT_LEN);       /*将新的数据块写入*/
35  close(fd);                                         /*关闭文件描述符*/
36  }
```

【运行结果】经过编译、链接，在命令提示符下执行程序。程序执行完毕后查看文件 HelloWorld.txt 的内容，如下所示。

```
1  Hello world!
2  Insert by lseek.
3  This is a example for open.
```

【代码解析】在本例中，首先打开文件 HelloWorld.txt；然后定位到插入的位置，将文件从插入位置开始至文件尾的内容整体向后移，移动的长度是插入数据的长度；最后定位到插入位置，将字符串写入。在本例中，各程序行的含义具体如下。

- 第 5～7 行：定义操作的宏定义。READ_LEN 表示一次读取数据的最大长度；INSERT_POS 表示要插入的位置；INSERT_LEN 表示要插入的长度。
- 第 14～21 行：以读写方式打开文件 HelloWorld.txt。
- 第 22 行：计算需要移动的文件长度，即文件的总长度减去插入位置偏移量。注意此处获取文件长度的方式，将文件偏移定位到文件的末尾，则返回值就是文件长度。
- 第 23 行：定位“当前文件位移”到插入位置。
- 第 24～30 行：读取插入位置到文件尾的内容至缓冲区。
- 第 31 行：定位“当前文件位移”到原文件内容要移动到的位置，即要插入的位置加上插入的长度。
- 第 32 行：将原文件部分内容向后移动。

- 第 33～34 行：移动“当前文件位移”至要插入的位置并写入要插入的内容。

注意：本例并未考虑文件比较大的情况。如果要在较大文件中插入，在移动原文件内容时，应采用循环的方式进行。

使用 lseek 系统调用时，应该注意使用 lseek 的几种比较特殊的具体方法如下。

- 将当前文件位移转到文件头：lseek(int fildes,0,SEEK_SET)。
- 将当前文件位移转到文件尾：lseek(int fildes,0,SEEK_END)。
- 取得当前文件位移：lseek(int fildes,0,SEEK_CUR)。

同时，文件位移量可以大于文件的当前长度。在这种情况下，对该文件的下一次写将延长该文件，并在文件中构成一个空洞。位于文件中但没有写过的字节都被设置为'\0'。

9.3　文件安全编程

在前面章节中向读者介绍了文件操作了基本函数，即文件的打开、关闭及读写等操作。本节将介绍文件的安全编程，主要包括文件的属主信息的维护及文件权限管理编程。在介绍文件权限编程时，分别介绍了两种不同的权限管理模式：UGO 模式及 ACL 模式。

9.3.1　文件的属主及用户组编程

每个文件都有惟一的用户属主，即文件的拥有者，这里的属主是 Linux 系统的用户。同样，每个文件也属于惟一的组，这里的组是指 Linux 的用户组。文件的属主信息可以通过执行命令 l 得到。文件在被创建后，可以通过系统调用对文件的属主信息进行修改。为保证文件系统的安全性，修改文件属主及文件权限的操作只允许超级用户（uid=0）和文件拥有者进行。

设置文件属主的系统调用有 3 个，分别是 chown、fchown、lchown。chown 用于修改文件的属主，lchown 用于修改符号链接本身的属主，而不是符号链接所指向的文件。fchown 与 chown 的功能是相同的。不过，fchown 是对已打开的文件进行修改。系统调用的原型声明如下。

```
#include <sys/types.h>
#include <unistd.h>
int chown(const char *pathname, uid_t owner, gid_t group) ;
int fchown(int fd,uid_t owner, gid_t group) ;
int lchown(const char *pathname, uid_t owner, gid_t group) ;
```

（1）参数说明如下。

- pathname：文件路径。
- fd：已打开文件的文件描述符，用于 fchown 系统调用。
- owner：变更的用户 ID。
- group：变更的组 ID。

（2）返回值说明如下。

- 若成功，返回 0。
- 若出错，返回-1。

【范例 9-14】修改文件 HelloWorld.txt 的文件属主为超级用户。其实现过程如示例代码 9-14 所示。

示例代码 9-14

```
1   #include <sys/stat.h>                                    /*头文件*/
2   #include <unistd.h>
3   #include <fcntl.h>
4   #include <errno.h>
5   main()                                                   /*主函数*/
6   {
7       if(chown("./HelloWorld.txt",0,0) == -1)              /*调用 chown 修改文件的属主*/
8       {
9           perror(errno);                                   /*修改失败，根据 errno 进行错误输出*/
10      }
11      else
12      {
13        write(STDERR_FILENO,"Call chown success.\n",20);    /*修改成功，输出提示信息*/
14      }
15  }
```

【运行结果】经过编译、链接，在超级用户命令提示符下执行程序。程序执行完毕后通过执行 l 命令查看文件 HelloWorld.txt 的文件信息，如下所示。

```
1   -rw-------  1 root root 58 2008-06-28 06:02 HelloWorld.txt
```

【代码解析】在本例中，由于操作系统设置了_POSIX_CHOWN_RESTRICTED 常量（关于该常量的说明，见下述的注意）。所以应在超级用户权限下执行。在本例中，第 7 行表示：修改文件 HelloWorld.txt 的属主用户 ID 为 0（root 用户），组 ID 为 0（root 组）。使用 chown 系列系统调用时，应该注意以下几点。

- 如果 owner 或者 goup 为-1，则不会更改文件的文件属主或者用户组信息。
- 调用 chown 的进程有效用户标识符必须是超级用户或者文件拥有者，否则将失败。
- 在只读文件系统上调用 chown 将返回失败。
- 对一个符号链接文件用 chown 修改属主，实际修改的是该符号链接指向的文件。如果要修改符号链接文件本身的属主，需要用 lchown 系统调用。
- 如果 chown 不是由超级用户进程调用，则调用成功后，原来文件设置的 suid 及 sgid 位将被清除。
- 不同版本的 Linux 系统对于 chown 有着不同的限制。如果操作系统定义了限制常数 _POSIX_CHOWN_RESTRICTED（该限制常数是否定义可以通过执行命令 getconf 或者函数 sysconf 获得），则 chown 适用于下述规则。

只有超级用户进程才能修改文件的拥有者 ID。只有两种情况，非超级用户进程才能修改文件的属组 ID。一是进程拥有此文件，即进程的有效用户 ID 与文件的属主 ID 相同；二是 chown 的参数 owner 等于文件的属主用户 ID，而参数 group 等于进程的有效组 ID。

9.3.2　设置文件权限（UGO 模式）

前面章节曾经介绍过 Linux 下的文件权限管理分为 UGO 模式及 ACL 模式。本节首先介绍 UGO 模式的权限管理编程，在下一节中将向读者介绍 ACL 模式的编程。在 UGO 模式下，对文件权限的修改是通过函数 chmod 及 fchmod 完成的。fchmod 与 chmod 实现的功能是相同的。不过，fchmod 是对文件描述符进行操作。这两个系统调用的原型如下所示。

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod(const char *pathname,mode_t mode) ;
int fchmod(int fd, mode_t mode) ;
```

（1）参数说明如下。

- pathname：文件路径。
- fd：已打开文件的文件描述符，用于 fchmod 系统调用。
- mode：权限组合。

（2）返回值说明如下。

- 若成功，返回 0。
- 若出错，返回-1。

【范例 9-15】修改文件 HelloWorld.txt 的文件存取权限为 644。其实现过程如示例代码 9-15 所示。

示例代码 9-15

```
1   #include <unistd.h>                                                    /*头文件*/
2   #include <sys/types.h>
3   #include <sys/stat.h>
4   #include <errno.h>
5   main()                                                                 /*主函数*/
6   {
7       if(chmod("./HelloWorld.txt",S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH) == -1) /*修改文件权限*/
8       {
9           perror(errno);                                                 /*输出错误信息*/
10      }
11      else
12      {
13          write(STDOUT_FILENO,"Call chmod success.\n",20);               /*输出成功信息*/
14      }
15  }
```

【运行结果】经过编译、链接，在 shell 下执行程序。程序执行完毕后通过执行 l 命令查看文件 HelloWorld.txt 的文件信息，如下所示。

```
1   -rw-r--r--   1 develop users   58 2008-06-28 06:02 HelloWorld.txt
```

【代码解析】在本例中，使用了宏定义代替八进制数字的权限。实际上，直接使用 chmod（"./HelloWorld.txt",0644）效果是相同的。在程序输出中，文件 HelloWorld.txt 的文件属主 develop 的权限是 rw-，而组 users 及其他用户的权限是 r--。在本例中的第 7 行表示：修改文件 HelloWorld.txt 的权限为文件所有者可读写，其他用户只读。使用 chmod 系统调用时，应该注意以下几点。

- 执行 chmod 的进程的有效用户 ID 必须等于文件的拥有者，或者是超级用户，才能修改文件访问许可。
- 调用 chmod 更改文件权限信息时，修改是 i 结点的数据结构，并没有修改文件内容。在修改完成后，通过执行 l 命令可以看到文件的修改时间并没有变化。
- 实际上，在权限组合参数中，还有一个称作文件粘住位的参数：S_ISVTX。由于实际应用中使用很少，本书中未做介绍。读者有兴趣可以查找相关资料进行深入了解。

提示：虽然允许超级用户及文件拥有者随意设置文件的访问权限。但是，对于访问权限的设置一定要慎重。否则，容易引起安全方面的漏洞。例如，文件访问权限设置了 suid 或者 sgid 等。

除了通过 chmod 修改文件的访问权限外，还可以通过 umask 为进程设置文件权限屏蔽字。所谓文件屏蔽字，就是指一旦在该调用中指定了某项访问权限，则该进程所创建的文件不允许拥有此项权限。通过这一机制，间接达到控制文件访问权限的目的。系统中该权限屏蔽字的默认值是 0022。umask 可以有效防止文件权限被意外设置，所以通过这一机制可以在进程执行期间保护所创建的文件。umask 系统调用的原型如下所示。

```
#include <sys/types.h>
#include <sys/stat.h>
mode_t umask(mode_t mask) ;
```

参数 mask 表示访问权限组合；返回值是调用 umask 前的文件权限屏蔽字。

【范例 9-16】通过 creat 创建两个文件，创建第一个时，umask 值为 0022；创建第二个时，设置 umask 值禁止组和其他用户的全部存取许可权。其实现过程如示例代码 9-16 所示。

示例代码 9-16

```
#include <sys/stat.h>                                          /*头文件*/
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
main()                                                         /*主函数*/
{
int fd1,fd2;                                                   /*文件描述符*/
mode_t nOldMode;                                    /*用于保存原来的文件访问控制掩码*/
char szErrorBuffer[256];                                       /*错误信息缓冲区*/
fd1 = creat("./file1.txt", S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
                                S_IROTH | S_IWOTH);            /*调 creat 创建新文件 file1*/
if (fd1 == -1)                                                 /*创建失败后的错误处理*/
{
      perror(errno);                                           /*输出错误信息*/
      return;
}
close(fd1);                                                    /*关闭文件*/
nOldMode = umask(S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH); /*设置当前进程的文件创建掩码*/
memset(szErrorBuffer,0x00,sizeof(szErrorBuffer));              /*清空错误输出缓存*/
sprintf(szErrorBuffer,"Old umask is %0o.\n",nOldMode);         /*向缓存中复制错误信息*/
write(STDOUT_FILENO,szErrorBuffer,strlen(szErrorBuffer));      /*输出原有的掩码信息*/
fd2 = creat("./file2.txt", S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
```

```
24                                          S_IROTH | S_IWOTH);          /*创建新文件 file2*/
25    if (fd2 == -1)                                                     /*创建失败后的错误处理*/
26    {
27        perror(errno);                                                 /*输出错误信息*/
28        return;
29    }
30    close(fd2);                                                        /*关闭文件*/
31    }
```

【运行结果】经过编译、链接，在 shell 下执行程序。程序输出结果是：Old umask is 22。程序执行完毕后通过执行 l 命令查看文件 file1.txt 及 file2.txt 的文件信息，如下所示。

```
1    -rw-r--r--    1 develop users      0 2008-06-28 11:51 file1.txt
2    -rw-------    1 develop users      0 2008-06-28 11:51 file2.txt
```

【代码解析】在本例中，首先在当前的文件权限屏蔽字（从后面的输出中可以看到是 0022）下，创建一个新文件 file1.txt。可以看到，尽管在创建文件 file1.txt 时指定了 S_IWGRP 及 S_IWOTH，但是实际创建的文件权限却没有这两种权限，这正是 umask 发生作用的结果（原来的 umask 是 0022，把 S_IWGRP 及 S_IWOTH 进行了屏蔽）。然后调用 umask 设置当前进程的文件屏蔽字，屏蔽组用户和其他用户的所有权限。从执行结果中可以看到 file2.txt 的权限信息已发生变化。

在本例中，各程序行的含义具体如下。

- 第 12～19 行：在原来的文件屏蔽字环境下，创建新文件 file1.txt。并置文件权限为 0666。
- 第 20～23 行：通过 umask 调用设置文件屏蔽字，同时返回原来的文件屏蔽字。
- 第 24～31 行：在新的文件屏蔽字环境下，创建新文件 file2.txt。

上面介绍了设置文件访问权限的几种方法。那么，如何获取一个文件的访问权限呢？这是通过系统调用 access 完成的。access 可以对文件是否具有某种权限进行检测，其原型如下所示。

```
# include <unistd.h>
int access(const char *pathname int mode) ;
```

参数说明如下。

- pathname：访问权限组合。
- mode：如图 9-12 所示的常数的或（OR）运算。

如果当前进程具有所测试的权限，则返回 0；否则，返回-1。

mode	说明
R_OK	测试读许可权限
W_OK	测试写许可权限
X_OK	测试执行许可权限
F_OK	测试文件是否存在

图 9-12　mode 定义

9.3.3　设置文件权限（ACL 模式）

通过传统的 UGO 模式对文件系统的权限控制比较简单。随着对 Linux 系统安全性要求的提高，需要一种更细粒度的访问控制模型。在 POSIX 中定义了 ACL 访问控制。要进行 ACL 编程，要具备 3 个条件：一是系统内核必须支持 ACL；二是文件系统必须支持 ACL；三是需要安装 ACL 开发包 libacl-devel。

可以通过查看文件/bool/config-版本号-default 的内容获得当前系统内核是否支持 ACL，

例如，存在类似语句 CONFIG_EXT2_FS_POSIX_ACL=y 表明内核在 ext2 文件系统上支持 ACL。其他文件系统与此相同。通过不带参数的命令 mount 可以查询当前安装的文件系统是否支持 ACL；否则，可以通过执行 mount 命令加上 ACL 选项重新挂载，或者修改/etc/fstab 文件实现自动挂载的文件系统对 ACL 的支持。安装 ACL 开发包比较简单，通过 yast 的 software 的安装功能即可从安装介质上安装 libacl-devel 开发包。ACL 提供的函数很多，涉及的数据结构也较为复杂，如图 9-13 所示。

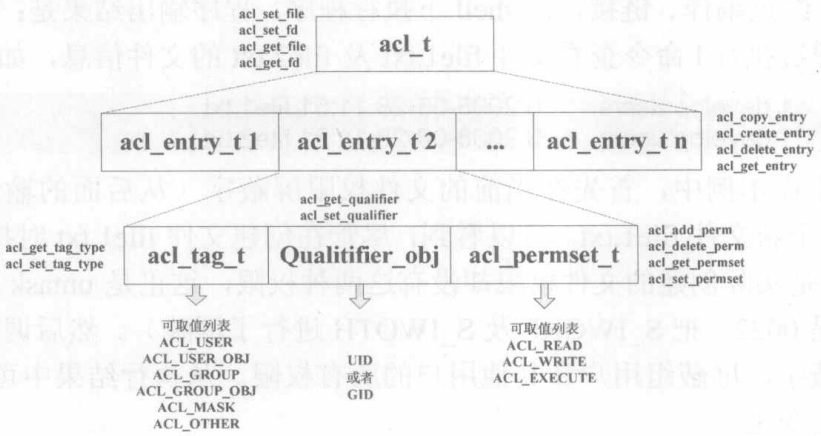


图 9-13　ACL 数据结构图

在图 9-13 中，acl_t 是基本数据结构，设置 acl 权限的函数 acl_set_file 及获取 acl 权限的函数 acl_get_file 都是以 acl_t 为参数的。acl_t 包含若干个 acl_entry_t，每个 entry 定义了一项权限结构。每个权限结构包含 3 部分内容：acl_tag_t、qualitifier_obj、acl_psermset_t。其中，acl_tag_t 是权限分配的对象类型，qualitifier_obj 是权限分配的用户 ID 或者组 ID，acl_psermset_t 是分配的权限。

1．acl_set_file、acl_set_fd 函数

acl_set_file、acl_set_fd 函数用于设置文件的 ACL 信息，二者的函数原型如下所示。

```
#include <sys/typs.h>
#include <sys/acl.h>
int acl_set_fd(int fd, acl_t acl);
int acl_set_file(const char *path_p, acl_type_t type, acl_t acl);
```

参数说明如下。

- path：要修改权限的文件路径。
- fd：已打开文件的文件描述符。
- type：修改权限的方式。
- acl: 即 ACL 的基本数据结构，其结构如下所示。

```
struct posix_acl {
        atomic_t                a_refcount;             // 结构的引用计数
        unsigned int            a_count;                // a_entries 地址中保存的 acl 权限数
        struct posix_acl_entry  a_entries[0];           // acl 权限数组的基地址
};
```

返回 0 表示设置成功；返回-1 表示失败。通过 errno 变量可以获取错误信息。

2．acl_get_file、acl_get_fd 函数

acl_get_file、acl_get_fd 函数用于获得文件的 ACL 信息，二者的函数原型如下所示。

```
#include <sys/typs.h>
#include <sys/acl.h>
acl_t acl_get_fd(int fd);
acl_t acl_get_file(const char *path_p, acl_type_t type);
```

获取 ACL 成功，则返回 ACL 信息；否则，返回 NULL。

3．acl_create_entry 函数

acl_create_entry 函数用于创建一个新的权限项。在进行文件权限的 ACL 设置时，一般需要首先创建一个新的权限项。读者从图 9-13 中可以看到，每个权限项都包含了 tag、qualitifier、permset 三方面内容。在创建一个新的权限项后，调用其他函数对这 3 项内容进行填充。最后调用 acl_set_file 进行权限设置。acl_create_entry 函数的原型如下所示。

```
#include <sys/typs.h>
#include <sys/acl.h>
int acl_create_entry(acl_t *acl_p, acl_entry_t *entry_p);
```

参数说明如下。

- acl_p：ACL 基本数据结构的指针。
- entry_p：新建权限项的指针，其结构如下所示。

```
struct posix_acl_entry {
short   e_tag;                  //权限类型
unsigned short  e_perm;         //权限 如 ACL_READ、ACL_WRITE、ACL_EXECUTE
unsigned int    e_id;           //用户 ID 或者组 ID
};
```

返回 0 表示创建成功，返回-1 表示失败。通过 errno 变量可以获取错误信息。

4．acl_set_tag_type 、acl_set_qualifier、acl_add_perm、acl_get_permset、acl_set_permset 函数

通过 acl_create_entry 创建新的权限项后，需要对权限项的内容进行设置，包括设置权限的方式、设置权限的对象及权限信息。这些内容是通过上述函数实现的，其函数原型如下所示。

```
#include <sys/typs.h>
#include <sys/acl.h>
int acl_set_tag_type(acl_entry_t entry_d, acl_tag_t tag_type);
int acl_set_qualifier(acl_entry_t entry_d, const void *tag_qualifier_p);
int acl_add_perm(acl_permset_t permset_d, acl_perm_t perm);
int acl_get_permset(acl_entry_t entry_d, acl_permset_t *permset_p);
int acl_set_permset(acl_entry_t entry_d, acl_permset_t permset_d);
```

参数说明如下。

- entry_d：acl_entry_t 类型变量。
- tag_type：分配权限的类型，如 ACL_USER、ACL_USER_OBJ 等。
- perm：权限信息，如 ACL_READ、ACL_WRITE。
- permset_d：权限信息的集合。

返回值 0 表示成功；返回-1 表示失败。通过 errno 变量可以获取错误信息。

【范例 9-17】通过 ACL 设置文件 HelloWorld.txt 的访问权限。为用户 testuser 增加对文件的写权限。其实现过程如示例代码 9-17 所示。为代码简洁起见，程序省略了基本的错误处理。

示例代码 9-17

```
1   #include <sys/types.h>
2   #include <stdio.h>
3   #include <sys/acl.h>                                    /*进行 ACL 编程所必需的头文件*/
4   #include <errno.h>
5   main()
6   {
7       uid_t uid;                                          /*用于保存用户的标识*/
8       uid=1001;                    /*初始化用户标识，在测试用机中，1001 代表用户 testuser*/
9       acl_t acl_p = NULL;                                 /*定义 acl_t 变量*/
10      acl_entry_t entry_p = NULL;                         /*定义 acl_entry_t 变量*/
11      acl_permset_t permset_p = NULL;                     /*定义 acl_permset_t 变量*/
12      acl_p = acl_get_file("./HelloWorld.txt",ACL_TYPE_ACCESS);   /*获取文件的 ACL 信息*/
13      acl_create_entry(&acl_p,&entry_p);                  /*得到当前的访问控制项*/
14      acl_get_permset(entry_p,&permset_p);                /*得到当前的访问权限信息*/
15      acl_add_perm(permset_p,ACL_WRITE);                  /*增加写权限*/
16      acl_set_tag_type(entry_p,ACL_USER);                 /*指定要为某个单独用户增加写权限*/
17      acl_set_qualifier(entry_p,(void *)&uid);            /*指定要增加写权限的用户 ID*/
18      acl_set_file("./HelloWorld.txt",ACL_TYPE_ACCESS,acl_p);     /*设置访问控制列表*/
19  }
```

【运行结果】经过编译、链接（注意链接 acl 库），在 shell 下执行程序。通过执行 getfacl 命令查看文件 HtlloWorld.txt 的 ACL 权限信息，如下所示。

```
1   # file: HelloWorld.txt
2   # owner: develop
3   # group: users
4   user::rw-
5   user:testuser:-w-
6   group::r--
7   mask::r-x
8   other::r--
```

【代码解析】在本例中，首先调用 acl_get_file 获取文件 HelloWorld.txt 的 ACL 信息；然后调用 acl_create_entry 创建一个新的权限项，并填充相关数据信息；最后调用 acl_set_file 设置 ACL 权限。从执行结果中可以看到 HelloWorld.txt 的权限信息已发生变化。在本例中，各程序行的含义具体如下。

- 第 8 行：Linux 用户 testuser 的用户 ID。
- 第 12 行：获取文件 HelloWorld.txt 的 ACL 信息。
- 第 13 行：创建新的 ACL 权限项。
- 第 14 行：获取新建 ACL 权限项的权限集合指针。
- 第 15 行：为权限集合增加写权限。
- 第 16 行：设置权限方式为 ACL_USER，即为某用户单独设置权限。
- 第 17 行：为权限集合设置用户 ID，即要为哪个用户设置 ACL 权限。
- 第 18 行：调用 acl_set_file 设置文件的 ACL 权限。

在 getfacl 的输出信息中，读者可以看到，第 5 行已为用户 testuser 增加了单独的写权限。

9.4　文件属性编程

文件在文件系统中不仅存储文件的数据，而且存储了文件的各种属性信息，如文件的大小、文件的权限等。在上一节已经向读者介绍了文件权限的编程方法，在本节将介绍文件的其他属性的编程方法。

1．文件状态的获取

在进行程序设计过程中，经常需要获取文件的状态信息，如文件的类型、大小及创建时间等。这些获取状态信息的系统调用可划分为两类，分别是获取文件的状态信息和获取文件系统的状态信息。在本节中，首先向读者详细介绍获取文件状态信息的方法，然后再对获取文件系统状态的方法做一介绍。获取文件的状态信息是通过 stat、fstat 和 lstat 实现的。获取文件状态信息的系统调用原型如下所示。

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *file, struct stat *buf) ;
int fstat(int fd,struct stat * buf) ;
int lstat(const char *file, struct stat *buf) ;
```

参数说明如下。

- file：文件名称。
- fd：已打开文件的文件描述符。
- buf：指向 stat 结构的指针，用于保存获取的文件状态数据。结构 stat 的结构如下所示。

```
struct stat {
unsigned long     st_dev;          /* 文件所在设备的标识 */
unsigned long     st_ino;          /* 文件结点号 */
unsigned short    st_mode;         /* 文件保护模式 */
unsigned short    st_nlink;        /* 硬连接数 */
unsigned short    st_uid;          /* 文件用户标识 */
unsigned short    st_gid;          /* 文件用户组标识 */
unsigned long     st_rdev;         /* 文件所表示的特殊设备文件的设备标识 */
unsigned long     st_size;         /* 总大小，字节为单位 */
unsigned long     st_blksize;      /* 文件系统的块大小 */
unsigned long     st_blocks;       /* 分配给文件的块的数量，512 字节为单元 */
unsigned long     st_atime;        /* 最后访问时间 */
unsigned long     st_atime_nsec;
unsigned long     st_mtime;        /* 最后修改时间 */
unsigned long     st_mtime_nsec;
unsigned long     st_ctime;        /* 最后状态改变时间 */
unsigned long     st_ctime_nsec;
unsigned long     __unused4;
unsigned long     __unused5;
};
```

2．文件系统状态的获取

获取文件系统的状态信息则是通过 statfs 系统调用实现的，其原型如下所示。

```
#include <sys/types.h>
```

```
#include <sys/statfs.h>
int statfs (const char *file，struct statfs *buf);
```

参数说明如下。

- file：文件名称。
- buf：指向 struct statfs 结构的指针，结构的定义如下所示。

```
struct statfs {
__u32 f_type;                    /* 文件系统类型 */
__u32 f_bsize;                   /* 经过优化的传输块大小 */
__u32 f_blocks;                  /* 文件系统数据块总数 */
__u32 f_bfree;                   /* 可用块数 */
__u32 f_bavail;                  /* 非超级用户可获取的块数 */
__u32 f_files;                   /* 文件结点总数 */
__u32 f_ffree;                   /* 可用文件结点数 */
__kernel_fsid_t f_fsid;          /* 文件系统标识 */
__u32 f_namelen;                 /* 文件名的最大长度 */
__u32 f_frsize;                  /* 分配新的文件存储空间时的最小单位 */
__u32 f_spare[5];                /* 留作扩展 */
};
```

3．文件类型的获取

宏定义	意义
S _ ISREG( )	普通文件
S _ ISDIR( )	目录文件
S _ ISCHR( )	字符特殊文件
S _ ISBLK( )	块特殊文件
S _ ISFIFO ( )	管道或FIFO
S _ ISLNK( )	符号连接
S _ ISSOCK( )	套接字

图 9-14　文件类型宏定义

在前面的章节中曾经介绍文件系统，读者对 Linux 下的文件类型有了一定的了解。那么，如何才能获得一个文件的类型呢？Linux 向用户提供了一组宏定义对文件类型进行判断。需要指出的是，这组宏定义是根据上面所讲到的 stat 的输出中 st_mode 项的内容进行判断的。文件类型宏定义，如图 9-14 所示。

【范例 9-18】获取根文件系统的可用空间及文件 /etc/passwd 的文件大小和类型。其实现过程如示例代码 9-18 所示。为代码简洁起见，程序省略了基本的错误处理。

示例代码 9-18

```
1   #include <unistd.h>                                          /*头文件*/
2   #include <unistd.h>
3   #include <sys/types.h>
4   #include <sys/stat.h>
5   #include <sys/statfs.h>
6   main()                                                       /*主函数*/
7   {
8       struct statfs buf;          /*statfs 结构变量定义，用于保存获取到的文件系统信息*/
9       struct stat fbuf;           /*stat 结构变量定义，用于保存获取到的文件信息*/
10      char szOutBuffer[256];                                   /*输出信息缓冲区*/
11      memset(&buf,0x00,sizeof(buf));                           /*清空缓冲区*/
12      statfs("/",&buf);                                        /*获取根文件系统的信息*/
13      memset(szOutBuffer,0x00,sizeof(szOutBuffer));
14      sprintf(szOutBuffer,"Available space of root filesystem is %ld MB.\n",
15              buf.f_bsize * buf.f_bavail / (1024*1024));   /*获取根文件系统的可用空间信息*/
16      write(STDERR_FILENO,szOutBuffer,strlen(szOutBuffer));    /*输出文件系统的信息*/
```

```
17          memset(&fbuf,0x00,sizeof(fbuf));
18          stat("/etc/passwd",&fbuf);                                  /*获取/etc/passwd 信息*/
19          sprintf(szOutBuffer,"The size of file /etc/passwd is %ld bytes.\n",
20                  fbuf.st_size);                                      /*获取文件的大小信息*/
21          strcat(szOutBuffer,"The type of file /etc/passwd is ");
22          if(S_ISREG(fbuf.st_mode)) strcat(szOutBuffer,"regular file.\n");   /*获取文件的类型信息*/
23          else if(S_ISDIR(fbuf.st_mode)) strcat(szOutBuffer,"directory file.\n");       /*目录文件*/
24          else if(S_ISCHR(fbuf.st_mode)) strcat(szOutBuffer,"char special file.\n");/*字符设备文件*/
25          else if(S_ISBLK(fbuf.st_mode)) strcat(szOutBuffer,"block special file.\n"); /*块设备文件*/
26          else if(S_ISFIFO(fbuf.st_mode)) strcat(szOutBuffer,"fifo file.\n");          /*管道文件*/
27          else strcat(szOutBuffer,"other file.\n");                   /*其他文件*/
28          write(STDERR_FILENO,szOutBuffer,strlen(szOutBuffer));       /* 输出文件
    /etc/passwd 的大小类型信息*/
29      }
```

【运行结果】经过编译、链接，在 shell 下执行程序，输出如下所示。

```
1    Available space of root filesystem is 96 MB.
2    The size of file /etc/passwd is 826 bytes.
3    The type of file /etc/passwd is regular file.
```

【代码解析】在本例中，各程序行的具体含义如下。

- 第 12 行：调用系统调用 statfs 获取文件系统的状态信息。
- 第 13～16 行：格式化文件系统状态信息并输出。
- 第 18 行：调用系统调 stat 获取文件的状态信息。
- 第 19～20 行：格式化文件的大小信息并输出。
- 第 21～28 行：格式化文件的类型信息并输出。

4．修改文件时间

对于系统中的每个文件，Linux 在内核记录了 3 个时间：文件数据的存取时间、文件数据的修改时间、i 节点状态的修改时间，如图 9-15 所示。

时间属性	属性描述	举例	l 命令选项
st_atime	文件数据的最后存取时间	read	-u
st_mtime	文件数据的最后修改时间	write	缺省
st_ctime	i节点状态的最后更改时间	chmod	-c

图 9-15　文件的时间属性

请读者注意修改时间（st_mtime）和状态更改时间（st_ctime）之间的区别。修改时间是文件内容最后一次被修改的时间。更改状态时间是该文件的 i 节点最后一次被修改的时间。一般情况下，修改文件属性操作修改的都是 i 节点状态。而对文件的写操作才会修改文件的修改时间。在 Linux 下，文件的存取时间和修改时间可以通过系统调用 utime 进行维护，其原型声明如下。

```
#include <sys/types.h>
#include <utime.h>
int utime(const char *file, const struct utimbuf *times) ;
```

参数说明如下。

- file：文件名称。

- times：指向 struct utimbuf 结构的指针，结构的定义如下所示。

```
struct utimbuf
  {
    __time_t actime;              /*文件的存取时间*/
    __time_t modtime;             /*文件的修改时间*/
  };
```

9.5 目录编程

在 Linux 系统中，目录是一种特殊的文件，称作目录文件。对于前面提到的 open、close、read 等文件操作同样适用于目录文件。但是，作为一种特殊的文件，目录文件有着与普通文件不同的地方。例如，不能对目录进行 write 操作，目录的创建与删除操作也与普通文件不同。为更好地进行目录文件的编程，首先向读者介绍有关目录文件的一些基本概念。

1．工作目录与用户主目录

Linux 用户在登录到系统后，在使用系统的过程中时刻都处在某个目录之中，这个目录就称作工作目录或当前目录。工作目录是可以随时改变的。当用户初始登录进入系统时，其主目录就成为工作目录（工作目录用“.”表示，其父目录用“..”表示）。

用户主目录是创建用户时创建的，每个用户都有自己的主目录，不同用户的主目录一般互不相同。查看用户主目录可以通过查看文件/etc/passwd 完成。或者通过环境变量 HOME 的值也可获得。在日常应用中，用户可以通过“～”字符来引用自己的主目录。

2．路径

路径是指从树型目录中的某个目录层次到某个文件的一条通路，路径中的各个目录层次中间用“/”分隔。路径分为相对路径和绝对路径。绝对路径是指从根（/）开始的路径，也称为完全路径；相对路径是从用户工作目录开始的路径。读者应该注意，在树型目录结构中到某个确定文件的绝对路径和相对路径均只有一条。绝对路径是固定不变的，而相对路径则随着用户工作目录的变化而不断发生变化。

涉及目录文件的操作主要有目录的创建、删除、工作目录的改变、读取目录信息等，下面向读者逐一进行介绍。

3．目录的创建

创建目录的系统调用是 mkdir，其原型声明如下所示。

```
#include <sys/types.h>
#include <sys/stat.h>
int mkdir(const char *path, mode_t mode) ;
```

（1）参数说明如下。

- path：要创建的目录文件名称，可为绝对路径或者相对路径。
- mode：新创建目录的权限。

（2）返回值说明如下。

若成功，返回 0，否则返回-1。

（3）在调用 mkdir 时，应注意以下几点。

- 新建目录的实际权限是由 mode 参数与文件权限掩码 umask 做与（AND）操作的结果，这一点与文件操作是相同的。
- 创建新目录成功后，系统将在新的目录项下自动创建两个子目录项：“.”和“..”，分别代表当前目录和父目录。

提示：在 Linux 系统下，还有另外一个系统调用可以创建目录，这就是 mknod。不过，使用 mknod 创建目录必须具有超级用户权限。同时，新建的目录中系统不会自动建立“.”和“..”两个目录项。所以，建立目录完成后，是无法执行 cd ..命令进入上级目录的，必须通过系统调用 link 建立这两个默认目录项后才能访问。

4．目录的删除

删除目录的系统调用是 rmdir。如果目录的引用计数变为 0，则释放目录占用的系统空间；否则，只是将目录的引用计数减一。其原型声明如下所示。

```
#include <sys/types.h>
#include <sys/stat.h>
int rmdir(const char *path) ;
```

（1）参数说明如下。

- path：要创建的目录文件名称，可为绝对路径或者相对路径。

（2）返回值说明如下。

若成功，返回 0，否则返回-1。

（3）在调用 rmdir 时，应注意以下几点。

只有在目录中除“.”和“..”外，没有其他文件时，才能删除目录。前面介绍过的 unlink 系统调用可以删除非空的目录，但是执行删除操作的用户必须具有超级用户权限。

5．改变和获取工作目录

当前工作目录是进程的属性，而起始目录则是用户的属性。进程调用 chdir 或者 fchdir 函数可以变更当前工作目录，而通过 getcwd 则可以获得当前工作目录。它们的原型声明如下所示。

```
#include <unistd.h>
int chdir(const char *path);
int fchdir(int fd);
char *getcwd(char *buf,size_t size);
```

（1）参数说明如下。

- path：要变更的目标目录。
- fd：要变更到的目标目录的文件描述符。该目录已被当前进程所打开。
- buf：保存得到的工作目录名称缓存。
- size：缓存大小。

（2）返回值说明如下。

- chdir 及 fchdir 调用成功，返回 0，否则返回-1。
- getcwd 调用成功，返回当前工作目录名称，否则返回 NULL。

6. 目录的读取

目录实际上是一种特殊的文件，对其也可以通过 open 打开，然后用 read 读取其内容。但是，目录文件的内容是由系统规定的数据结构组成的。直接读取目录文件的内容，进行格式化时存在诸多不便。因此，Linux 系统向用户提供了专门的系统调用，用于对目录进行操作。这些系统调用包括打开目录的 opendir、读目录的 readdir 及关闭目录的 closedir。另外，系统还提供了重绕目录的 rewinddir，用于重定位目录的读指针。它们的原型声明如下所示。

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char * pathname) ;
struct dirent *readdir(DIR * dp) ;
void rewinddir(DIR * dp) ;
int closedir(DIR * dp) ;
```

（1）参数说明如下。

- pathname：目录名称。
- dp：DIR 结构的指针。DIR 结构是 Linux 内核的目录文件结构定义。该结构对于用户来说是透明的，只需在程序中直接使用即可。

（2）返回值说明如下。

- opendir 调用成功，返回 DIR 指针，失败则返回 NULL。
- readdir 系统调用将返回一个指针，指向的结构里保存着目录中下一个目录项的相关数据。后续的 readdir 调用将返回后续的目录项，直至发生错误或者到达目录尾，此时，readdir 将返回 NULL。其中 struct dirent 的结构如下所示。

```
struct   dirent
{
 ino_t                  d_ino;       //目录文件的 inode 节点编号
 off_t                  d_off;       //目录文件开始至目录进入点的位移
 unsigned short int     d_reclen;    //d_name 的长度，不包含 NULL 字符
 unsigned char          d_type;      //d_name 所指的文件类型
char                    d_name[256]; //文件名
 };
```

- rewinddir 系统调用将目录的读指针重新返回至第一个目录项。无返回值。
- closedir 关闭已打开的目录项。调用成功，返回 0，失败，返回-1。

【范例 9-19】递归遍历指定的目录，输出遍历所得的目录名称。其实现过程如示例代码 9-19 所示。为代码简洁起见，程序省略了基本的错误处理。

示例代码 9-19

```
1   #include <sys/types.h>                          /*头文件*/
2   #include <unistd.h>
3   #include <dirent.h>                             /*进行目录操作必须包含本头文件*/
4   #include <sys/stat.h>
5   void SearchDir(char *szDir)                     /*函数定义，遍历指定的目录*/
6   {
7     DIR * dp;                                     /*目录数据结构*/
8     struct dirent *pDirent;                       /*目录项数据结构*/
9     struct stat fbuf;                             /*目录结构变量定义*/
```

```
10    char szOut[256];                                          /*输出缓存*/
11    dp = opendir(szDir);                                      /*打开目录*/
12    if (!dp)
13    {
14        return;                                               /*打开目录失败，退出*/
15    }
16    chdir(szDir);                                             /*改变工作目录至 szDir*/
17    while (pDirent=readdir(dp))                               /*循环读取目录 szDir 中的目录项*/
18    {
19        memset(&fbuf,0x00,sizeof(fbuf));                      /*清空结构*/
20        stat(pDirent->d_name,&fbuf);                          /*获取目录项的信息*/
21        if(S_ISDIR(fbuf.st_mode) && memcmp(pDirent->d_name,".",1)!= 0
22           && memcmp(pDirent->d_name,"..",2)!= 0)             /*判断是否是目录并且不是.和..*/
23        {
24                memset(szOut,0x00,sizeof(szOut));             /*清空缓存*/
25                sprintf(szOut,"%s/%-s",szDir,pDirent->d_name);      /*输出目录名称*/
26                write(STDOUT_FILENO,szOut,strlen(szOut));
27                write(STDOUT_FILENO,"\n",1);
28                SearchDir(szOut);                             /*递归调用当前函数，遍历其子目录*/
29        }
30    }
31    chdir("..");                                  /*为关闭目录，需要修改工作目录为非当前目录*/
32    closedir(dp);                                             /*关闭目录*/
33    return;
34  }
35  int main(int argc,char *argv[])                             /*主函数，带命令行参数*/
36  {
37      if (argc < 2)                                           /*参数个数不符*/
38      {
39          exit(1);
40      }
41      SearchDir(argv[1]);                                     /*遍历命令行参数 1 指定的目录*/
42      return 0;
43  }
```

【运行结果】经过编译、链接，在 shell 下执行程序。遍历/var 目录的输出如下所示。

```
1   /var/adm
2   /var/adm/YaST
3   /var/adm/YaST/y2pm
4   /var/adm/YaST/SelDB
5   …
```

【代码解析】在本例中，各程序行的具体含义如下。

- 第 5～34 行：行递归函数体，用于对某个目录进行遍历。
- 第 11 行：打开所在的目录，返回 DIR 结构指针。
- 第 16 行：更改当前工作目录至要遍历的目录。
- 第 17～30 行：循环读取一个目录的目录项，并检测目录项的类型。如果是目录，则输出目录的路径并进行递归遍历。
- 第 21～22 行：判断目录项是否是目录，且排除当前目录项（.）和父目录项（..）。
- 第 24～27 行：格式化并输出目录项的路径信息。
- 第 28 行：递归调用遍历算法进行当前目录的子目录的遍历。

- 第 31 行：变更工作目录至父目录，该行与第 16 行是对应的。执行成功后，工作目录将变回未执行第 16 行前的状态。
- 第 32 行：关闭打开的目录项，并从递归函数中返回。
- 第 35 行：程序的主函数。注意：此处使用了命令行参数。在执行程序时，第一个命令行参数是要遍历的目录名称。如果没有命令行参数，则返回失败。
- 第 41 行：主函数根据传递进来的命令行参数，调用递归遍历函数 SearchDir 对目录进行遍历。

9.6 综合实例

Linux 系统下的工具软件 readelf 是用来查看 ELF 文件格式的。编程实现 ELF 文件查看器，可以显示出 ELF 格式文件的各种属性，如 ELF 头部信息、节表等。

9.6.1 ELF 文件格式

ELF 文件是 Linux 系统下主要的二进制文件格式主要有 3 种类型。

- 可重定位文件：也就是通常称的目标文件，后缀为.o。
- 动态链接库文件：也就是通常称的库文件，后缀为.so。
- 可执行文件：就是可执行程序。

一个 ELF 文件是由一个 ELF 文件头（ELF header）和其余的文件数据构成。这些文件数据包括以下几点。

- Program header table：描述 0 个或是多个段。
- Section header table：描述 0 个或是多个节。
- 节的数据。

其中，Program header table 和 Section header table 的入口和项数是在 ELF header 中进行描述的。ELF 文件格式如图 9-16 所示。

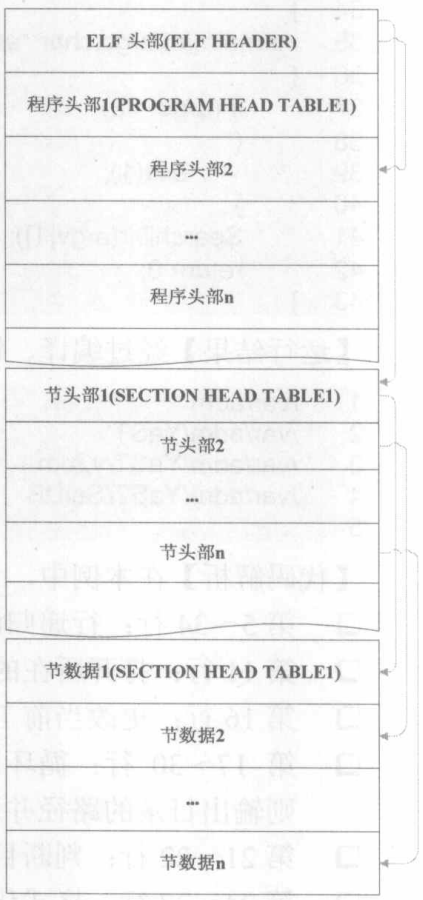


图 9-16 ELF 文件格式

9.6.2 程序实现

本小节将详细讲解如何实现上面的程序代码。由于本例的代码比较长，在下面的步骤中，将把这些代码分割为多个步骤进行讲解。

（1）头文件声明如下所示。

```
#include <stdio.h>                                          /*头文件*/
#include <elf.h>                                           /*包含了 ELF 的数据结构*/
#include <unistd.h>
#include <fcntl.h>                                         /*文件操作需要包含该头文件*/
```

（2）主函数的变量定义如下所示。

```
main(int argc,char *argv[])                                /*主函数*/
{
    int fd;                                                /*文件描述符*/
    int i,n;                                               /*循环变量*/
    unsigned char *string_table;                           /*用于分配存储空间保存程序的符号表*/
    long phdr_pos,shdr_pos;/*用于保存 Program Header Table 和 Section Header Table 入口*/
    Elf32_Ehdr elf_head;                                   /* ELF Header 结构变量*/.
    Elf32_Phdr elf_phdr;                                   /* ELF Program Header 结构变量*/
    Elf32_Shdr elf_shdr;                                   /*ELF Section Header 结构变量*/
    Elf32_Shdr elf_sshdr;      /* ELF Section Header 结构变量，用于保存符号表节的数据*/
```

（3）判断文件类型的代码如下所示。

```
    if(argc != 2)                                          /*判断是否从命令行输入了要查看的文件名*/
    {
        fprintf(stderr,"Usage:%s  文件名\n",argv[0]);      /*输出用法信息*/
        exit(0);
    }
    if((fd = open(argv[1],O_RDONLY)) ==-1 )                /*以只读的方式打开文件*/
    {
        perror("open");                                    /*输出错误信息*/
        exit(0);
    }
    n=read(fd,&elf_head,sizeof(Elf32_Ehdr));               /*读取 ELF header 信息*/
    if( n != sizeof(Elf32_Ehdr))                           /*错误处理*/
    {
        perror("read");                                    /*输出错误信息*/
        close(fd);                                         /*关闭文件*/
        exit(0);
    }
    if(memcmp(elf_head.e_ident+1,"ELF",3) != 0)            /*判断文件是否为 ELF 文件*/
    {
        fprintf(stderr,"文件%s 非 ELF 格式文件.\n",argv[1]);  /*输出错误信息*/
        close(fd);                                         /*关闭文件*/
        exit(0);
    }
    switch(elf_head.e_type)                                /*判断文件类型*/
    {
        case ET_REL:
            fprintf(stderr,"文件类型:可重定位文件\n");
            break;
        case ET_EXEC:
            fprintf(stderr,"文件类型:可执行文件\n");
            break;
        case ET_DYN:
            fprintf(stderr,"文件类型:动态库文件\n");
            break;
        case ET_CORE:
            fprintf(stderr,"文件类型:核心转储文件\n");
```

```
51                  break;
52              default:
53                  fprintf(stderr,"文件类型:其他类型文件\n");
54                  break;
55          }
```

（4）判断文件适用架构的代码如下所示。

```
56          switch(elf_head.e_machine)                          /*判断文件适用的架构*/
57          {
58              case EM_M32:
59                  fprintf(stderr,"文件架构:适合于 AT&T 架构\n");
60                  break;
61              case EM_386:
62                  fprintf(stderr,"文件架构:适合于 Intel X86 架构\n");
63                  break;
64              case EM_860:
65                  fprintf(stderr,"文件架构:适合于 Intel 860 架构\n");
66                  break;
67              case EM_MIPS:
68                  fprintf(stderr,"文件架构:适合于 MIPS 架构\n");
69                  break;
70              default:
71                  break;
72          }
```

（5）输出结果的代码如下所示。

```
73          fprintf(stderr,"程序头部信息,共有%d 个程序头部块\n",elf_head.e_phnum);  /*输出程序头部
    项的数目*/
74          fprintf(stderr,"%8s %10s %12s %12s %10s %10s %4s %4s\n",\
75                          "段类型","段偏移","虚拟地址","物理地址",\
76                          "在文件大小","在内存大小","标志","对齐");
77          phdr_pos = elf_head.e_phoff;                        /*保存 Program Header Table 的入口*/
78          shdr_pos = elf_head.e_shoff;                        /*保存 Section Header Table 的入口*/
79          lseek(fd,phdr_pos,SEEK_SET);   /*定位文件读写指针到 Program Header Table 的入口*/
80          for(i=0;i<elf_head.e_phnum;i++)    /*循环读取 Program Header 项，并进行格式化输出*/
81          {
82              n=read(fd,&elf_phdr,sizeof(Elf32_Phdr));            /*读取一个 Program Header 项*/
83              if( n != sizeof(Elf32_Phdr))                        /*错误处理*/
84              {
85                  perror("read");                                 /*输出错误信息*/
86                  close(fd);                                      /*关闭文件*/
87                  exit(0);
88              }
89              fprintf(stderr,"0x%06x 0x%08x 0x%010x 0x%010x 0x%08x 0x%08x %4d %4d\n",\
90                elf_phdr.p_type,elf_phdr.p_offset,elf_phdr.p_vaddr,elf_phdr.p_paddr,
91                elf_phdr.p_filesz,elf_phdr.p_memsz,elf_phdr.p_flags,elf_phdr.p_align);/*输出信息*/
92          }
93          fprintf(stderr,"节头部信息,共有%d 个节\n",elf_head.e_shnum);          /*输出节的数目*/
94          fprintf(stderr,"%20s %10s %10s %10s\n",\
95                          "节名称","节类型","节标志","节大小");
96          /*定位文件的读写指针到符号表数据节头部 symbol table section header*/
97          lseek(fd,shdr_pos+sizeof(Elf32_Shdr)*(elf_head.e_shstrndx),SEEK_SET);
98          /*读取符号表数据节头部的信息*/
99          n=read(fd,&elf_sshdr,sizeof(Elf32_Shdr));               /*读取符号表数据节头部的信息*/
```

```
    if( n != sizeof(Elf32_Shdr))                                    /*错误处理*/
    {
        perror("read");
        close(fd);
        exit(0);
    }
    lseek(fd,elf_sshdr.sh_offset,SEEK_SET);                         /*定位到符号表数据节*/
    string_table = (unsigned char *)malloc(
                    sizeof(unsigned char) * elf_sshdr.sh_size);                    /* 分 配 存 储 空
间，用于保存符号表节*/
    read(fd,string_table,elf_sshdr.sh_size);                        /*读取符号表节至内存*/
    lseek(fd,shdr_pos,SEEK_SET);      /*定位文件的读写指针到 Section Header 的入口*/
    for(i=0;i<elf_head.e_shnum;i++)   /*循环读取 Section Header，并输出每个节的配置信息*/
    {
        n=read(fd,&elf_shdr,sizeof(Elf32_Shdr));                    /*读取一个 Section Header*/
        if( n != sizeof(Elf32_Shdr))                                /*错误处理*/
        {
            perror("read");
            close(fd);
            exit(0);
        }
        fprintf(stderr,"%20s 0x%08x 0x%08x 0x%08x\n",\
          &string_table[elf_shdr.sh_name],elf_shdr.sh_type,elf_shdr.sh_flags,\
          elf_shdr.sh_size);                                        /*输出节头信息*/
    }
    close(fd);                                                      /*关闭文件*/
    free(string_table);                                             /*释放动态申请的空间*/
}
```

【运行结果】经过编译、链接，在命令提示符下执行程序显示某个可执行文件的结构，输出如下所示。

```
文件类型:可执行文件
文件架构:适合于 Intel X86 架构
程序头部信息,共有 8 个程序头部块
  段类型     段偏移     虚拟地址     物理地址     在文件大小 在内存大小  标志 对齐
0x000006 0x00000034 0x0008048034 0x0008048034 0x00000100 0x00000100    5    4
0x000003 0x00000134 0x0008048134 0x0008048134 0x00000013 0x00000013    4    1
0x000001 0x00000000 0x0008048000 0x0008048000 0x00000852 0x00000852    5 4096
0x000001 0x00000854 0x0008049854 0x0008049854 0x00000124 0x00000128    6 4096
0x000002 0x00000864 0x0008049864 0x0008049864 0x000000c8 0x000000c8    6    4
0x000004 0x00000148 0x0008048148 0x0008048148 0x00000020 0x00000020    4    4
0x000004 0x00000168 0x0008048168 0x0008048168 0x00000018 0x00000018    4    4
0x6474e551 0x00000000 0x0000000000 0x0000000000 0x00000000 0x00000000  6    4
节头部信息,共有 37 个节
                  节名称          节类型          节标志          节大小
                                  0x00000000      0x00000000      0x00000000
                  .interp         0x00000001      0x00000002      0x00000013
                  .note.ABI-tag   0x00000007      0x00000002      0x00000020
                  .note.SuSE      0x00000007      0x00000002      0x00000018
                  .hash           0x00000005      0x00000002      0x00000048
                  .dynsym         0x0000000b      0x00000002      0x000000d0
                  .dynstr         0x00000003      0x00000002      0x0000008b
                  .gnu.version    0x6fffffff      0x00000002      0x0000001a
                  .gnu.version_r  0x6ffffffe      0x00000002      0x00000020
                  .rel.dyn        0x00000009      0x00000002      0x00000008
```

25	.rel.plt	0x00000009	0x00000002	0x00000050
26	.init	0x00000001	0x00000006	0x00000017
27	.plt	0x00000001	0x00000006	0x000000b0
28	.text	0x00000001	0x00000006	0x00000374
29	.fini	0x00000001	0x00000006	0x0000001a
30	.rodata	0x00000001	0x00000002	0x00000042
31	.data	0x00000001	0x00000003	0x0000000c
32	.eh_frame	0x00000001	0x00000002	0x00000004
33	.dynamic	0x00000006	0x00000003	0x000000c8
34	.ctors	0x00000001	0x00000003	0x00000008
35	.dtors	0x00000001	0x00000003	0x00000008
36	.jcr	0x00000001	0x00000003	0x00000004
37	.got	0x00000001	0x00000003	0x00000038
38	.bss	0x00000008	0x00000003	0x00000004
39	.comment	0x00000001	0x00000000	0x000000d9
40	.debug_aranges	0x00000001	0x00000000	0x00000098
41	.debug_pubnames	0x00000001	0x00000000	0x0000005f
42	.debug_info	0x00000001	0x00000000	0x0000030a
43	.debug_abbrev	0x00000001	0x00000000	0x0000011d
44	.debug_line	0x00000001	0x00000000	0x0000023e
45	.debug_frame	0x00000001	0x00000000	0x00000058
46	.debug_str	0x00000001	0x00000030	0x0000011d
47	.debug_loc	0x00000001	0x00000000	0x00000047
48	.debug_ranges	0x00000001	0x00000000	0x00000018
49	.shstrtab	0x00000003	0x00000000	0x0000014f
50	.symtab	0x00000002	0x00000000	0x00000800
51	.strtab	0x00000003	0x00000000	0x00000400

9.7 小结

文件操作是 Linux 系统下编程的重要内容。本章首先介绍了 Linux 的文件系统结构，然后从程序设计的角度介绍了文件编程、文件安全编程、文件属性编程的知识。其中，文件编程的主要内容是实际开发中经常用到的文件操作，包括读写、删除和存取等。而文件编程部分只向读者介绍了不带缓存的 I/O 函数。关于标准文件 I/O 将在后续章节中向读者详细介绍。

第 10 章　标准输入输出库

在上一章中向读者介绍了 Linux 系统下的文件操作。这些文件操作都是直接调用文件操作的系统调用（也称文件操作 API）完成的。在 Linux 系统中，还提供了另外一种访问文件的方式：标准输入输出库（以下简称标准 I/O）。标准 I/O 库是在文件操作的系统调用基础上进行封装的，更加便于应用开发人员使用。本章将从流的概念讲起，介绍使用标准 I/O 库进行文件操作的方法。通过本章的学习，读者应重点掌握如下内容。

- 了解标准 I/O 与文件操作的系统调用之间的区别。
- 理解流的概念。
- 掌握使用标准 I/O 进程文件操作的方法。

10.1　标准 I/O 的基本概念

标准 I/O 操作建立在文件操作 API 的基础上，提供了更高层次的文件操作功能。在编程过程中，使用标准 I/O 进行文件操作更加方便。使用标准 I/O 时进程的映像如图 10-1 所示。

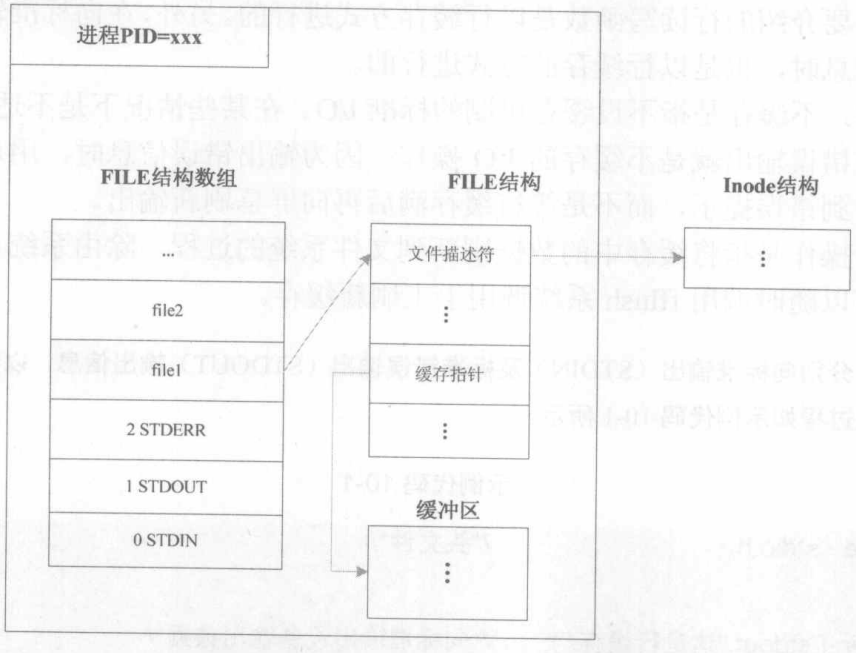


图 10-1　使用标准 I/O 时进程的映像

10.1.1　流

使用文件操作 API 访问文件系统，是针对文件描述符进行的。而使用标准 I/O 库时，所

操作的则是一个名称为“流”的对象。在使用标准 I/O 库访问文件时，需要首先打开文件。打开文件成功后，系统内核将返回流（stream）数据结构。所有后续的对该文件的操作都将针对这个数据结构进行。在文件操作中，这个流结构实际上是一个指向 FILE 结构的指针。

FILE 结构保存了标准 I/O 库用于文件操作的相关信息，如文件描述符、数据缓存的指针等。从开发的角度看，应用层开发人员不需要关心该结构的具体内容，在编程过程中，只需要将结构传递给相关的 I/O 操作函数即可。

10.1.2 缓存

标准 I/O 库与文件操作 API 的基本区别在于：标准 I/O 库是带缓存的，而文件操作 API 是无缓存机制的。缓存的含义是指在进行文件 I/O 操作时，对数据的读写并不直接对文件系统进行，而是对缓存进行。在缓存空间满时，由系统根据缓存的内容刷新到文件系统。采用标准 I/O 库机制可以降低调用文件操作的频率，并且由于标准 I/O 库提供了多种缓存方式，开发人员可以在编程过程中灵活运用。

缓存是在第一次调用标准 I/O 库进行 I/O 操作时，由系统自动调用 malloc 分配的。标准 I/O 库提供的缓存的方式主要包括以下 3 种。

- 全缓存：全缓存是指只有当前 I/O 操作的缓存被填满时，才会向文件系统进行刷新。大多数的标准 I/O 函数都是基于这种缓存方式的。
- 行缓存：行缓存是指在输入/输出过程中，如果遇到换行符，则向文件系统进行刷新。后面将要介绍的行读写函数是以行缓存方式进行的。另外，在向标准输出（STDOUT）输出信息时，也是以行缓存的方式进行的。
- 不缓存：不缓存是指不设缓存机制的标准 I/O。在某些情况下是不适宜设置缓存的，如标准错误输出就是不缓存的 I/O 操作。因为输出错误信息时，用户总是希望能够尽快看到错误提示，而不是等待缓存满后再向屏幕刷新输出。

缓存的刷新操作是指将缓存中的数据刷新到文件系统的过程。除由系统自动进行缓存刷新操作外，也可以随时调用 fflush 系统调用手工刷新缓存。

【范例 10-1】分别向标准输出（STDIN）及标准错误输出（STDOUT）输出信息，以检验行缓存和不缓存的区别。其实现过程如示例代码 10-1 所示。

示例代码 10-1

```
#include <stdio.h>                    /*头文件*/
main()
{
    fprintf(stdout,"这是行缓存.");     /*向标准输出设备输出数据*/
    fprintf(stderr,"这是无缓存.");     /*向标准错误输出设备输出数据*/
    fprintf(stdout,"\n");              /*向标准输出设备输出回车，将进行刷新*/
}
```

【运行结果】在 shell 下运行上述脚本，其结果如下所示。

```
这是无缓存.这是行缓存.
```

【代码解析】在本例中，首先向标准输出（stdout）输出一条信息，然后向标准错误输出（stderr）输出一条信息，最后再向标准输出设备输出一个换行符。通过程序的输出可以看到，第一条信息并没有首先输出在屏幕上，反而是第二条信息先显示。第一条信息直到接收到换行符后才进行输出。在本例中，源代码的详细解释如下。

- 第 1 行：头文件信息。
- 第 4 行：向标准输出设备输出一条信息。标准输出是采用行缓存的，由于未输出换行符，所以这个信息是不会立即显示的。
- 第 5 行：向标准错误输出设备输出一条信息。可以看到，本条信息照样没有换行符。但是，由于标准错误输出设备是不缓存的，所以该信息立即输出到屏幕上。
- 第 6 行：向标准输出设备输出换行符。在输出换行符后，在第 4 行输出的信息才会显示到屏幕上。

可以在程序中改变流的默认缓存方式。标准 I/O 库提供了改变缓存方式的几个函数：setbuf、setbuffer、setlinebuf 和 setvbuf。这几个函数的声明位于头文件<stdio.h>中，其原型如下所示。

```
#include <stdio.h>
void setbuf (FILE *__restrict __stream, char *__restrict __buf);
void setbuffer (FILE *__restrict __stream, char *__restrict __buf,size_t __size);
void setlinebuf(FILE *__stream);
int setvbuf (FILE *__restrict __stream, char *__restrict __buf,int __modes, size_t __size);
```

其中，setlinebuf 用于设置某个流为行缓存模式，setbuffer 与 setbuf 为公用参数。

（1）参数说明如下。

- __stream：输入参数，流对象，指向 FILE 结构的指针。调用 setbuf 和 setvbuf 时，必须针对一个已经打开的流进行。
- __buf：输入参数，自定义缓存区指针。如果该参数为空，表示设置流对象为无缓存的模式。对于 setbuf 来说，该缓存区的大小固定为 BUFSIZ。而对于 setbuffer，该缓存区大小可以自行定义，在参数__size 中指定了缓存区大小。
- __size：输入参数，缓存区大小，指定了自定义缓存区大小。

（2）返回值说明如下。

- 0：调用 setvbuf 成功。
- 其他：调用 setvbuf 失败。

（3）在设置流的缓存区时，需要注意以下几点。

- setbuf 与 setbuffer 的区别在于，setbuffer 可以自行指定缓存区大小，而 setbuf 缓存区大小是固定等于 BUFSIZ 的。
- 可以用 setvbuf 函数实现其他几个函数的功能。对于 setbuf 可以用 setvbuf 代替实现为 setvbuf(stream, buf, buf ? _IOFBF:_IONBF, BUFSIZ)；而 setlinebuf 的功能可以用如下语句替代实现：setvbuf(stream, (char *)NULL, _IOLBF, 0)。
- 要设置一个流为无缓存模式，只需调用 setbuf(stream,NULL)即可。
- 在设置某个流的缓存模式前，应确保该流是已被当前进程打开的。

【范例 10-2】分别向标准输出（STDIN）及标准错误输出（STDOUT）输出信息，以检验行缓存和不缓存的区别。其实现过程如示例代码 10-2 所示。

示例代码 10-2

```
1   #include <stdio.h>                          /*头文件*/
2   main()                                      /*主函数*/
3   {
4       setbuf(stdout,NULL);                    /*设置标准输出为无缓存模式*/
5       fprintf(stdout,"这是标准输出.");        /*验证是否无缓存*/
6       fprintf(stderr,"这是标准错误输出.");    /*向标准错误输出设备输出数据*/
7   }
```

【运行结果】经过编译链接，在 shell 下执行该程序，其结果如下所示。

```
1   这是标准输出.这是标准错误输出.
```

【代码解析】将本例与范例 10-1 进行比较可以发现，在本例中，标准输出被设置为无缓存模式。所以向标准输出写的信息马上被刷新到屏幕上，而没有经过行缓存。在本例中，源代码的详细解释如下。

- 第 1 行：头文件信息。
- 第 4 行：调用 setbuf 关闭缓存，使标准输出成为无缓存模式。
- 第 5～6 行：分别向标准输出和标准错误输出写入信息。

提示：在某些情况下，需要强制将缓存中的数据刷新至文件系统或者设备。此时，可以调用标准 I/O 库提供的函数 fflush 完成这一功能。

10.1.3 标准输入、标准输出及标准错误输出

对于每一个进程来说，系统将默认为其打开 3 个流：标准输入、标准输出和标准错误输出。在系统中分别用宏定义 stdin、stdout、stderr 表示，其代表的文件描述符分别是 0、1、2。在默认状态下，程序从键盘读取 stdin，将向 stdout 和 stderr 输出的信息显示到屏幕上。如果要改变这种默认的输入输出方式，需要使用一种名称为“输入/输出重定向”的技术。

所谓输入/输出重定向，就是将默认的输入/输出的目标重新定向到新的目标，新目标可能是某个文件，也可能是某个设备。比较典型的重定向的例子是错误输出重定向。在默认情况下，程序中向标准错误输出设备写入的错误信息都是显示在屏幕上。

为了跟踪错误及对错误进行留迹处理，有时需要把错误信息保存到文件中。在 Linux 的 shell 环境下，进行输入重定向的符号是“<”，进行输出重定向的符号是“>”和“>>”。其中，“>>”符号在向数据输出至重定向的目标时，是采取附加在目标尾部的方式进行的。例如，执行 ls>/tmp/a 命令将当前目录的文件列表输出至文件/tmp/a 中。如图 10-2 所示是进行标准输出重定向后的进程映像。

【范例 10-3】编程实现标准输出的重定向。其实现过程如示例代码 10-3 所示。

示例代码 10-3

```
1   #include <stdio.h>                                      /*头文件*/
2   main()                                                  /*主函数*/
3   {
4       FILE *fp;                                           /*流文件描述符*/
5       if((fp = fopen("./stdout.redirect","w")) == NULL)   /*以只写的方式打开文件*/
```

```
6       {
7           perror("fopen");                          /*错误处理*/
8           exit(0);
9       }
10      close(1);                                     /*关闭标准输出设备*/
11      if(dup2(fileno(fp),fileno(stdout)) == -1)     /*复制 stdout.redirect 的文件描述符至描述符 1*/
12      {
13          perror("dup2");                           /*错误处理*/
14          exit(0);
15      }
16      printf("This is a test message.\n");          /*向标准输出设备输出信息，经过重定向，应该
    输出到文件中*/
17      fclose(fp);                                   /*关闭文件*/
18  }
```

【运行结果】经过编译链接，在 shell 下运行上述程序。运行结束后，执行 l 命令的输出如下所示。

```
1    -rw-r-----    1 test    staff        24 Apr 05 2008    stdout.redirect
```

可以看到，当前目录下生成了新文件 stdout.redirect，其中的内容是“This is a test message.”。

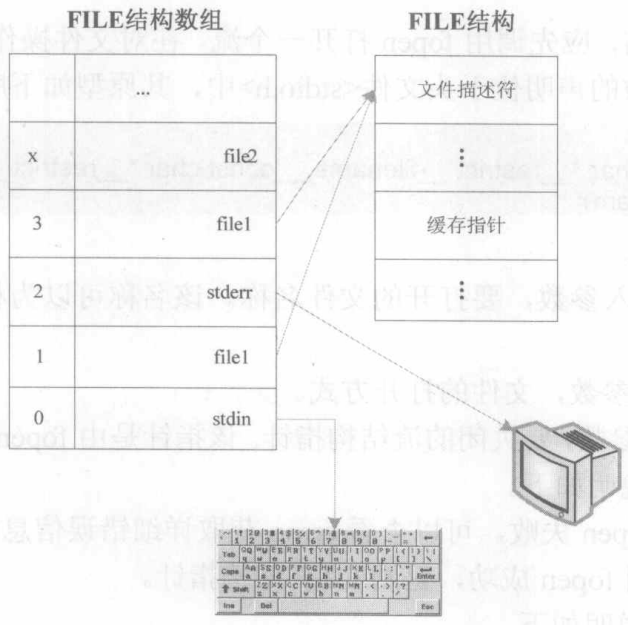


图 10-2　标准输出重定向

【代码解析】在本例中，首先打开要重定向的目标文件，然后关闭标准输出，并调用 dup2 复制目标文件的描述符至标准输出。经过如此处理后，在调用 printf 向标准输出设备输出信息时，将重定向至目标文件。在本例中，源代码的详细解释如下。

- 第 1 行：头文件信息。
- 第 4 行：FILE 结构指针，用于保存打开文件后返回的流。
- 第 5～9 行：打开新的文件 stdout.redirect 用于写，返回的流对象保存至 FILE 结构中。
- 第 10 行：调用 close 关闭文件描述符 1，文件描述符 1 代表标准输出。

- 第 11～15 行：调用 dup2 复制刚刚打开的目标文件的文件描述符至标准输出。其中，fileno 系统调用的作用是取得某个流的文件描述符。在本例中，fileno(stdout)与直接输入 1 是等价的。
- 第 16 行：向标准输出设备输出一行信息。printf 与 fprintf(stdout,…)调用是等价的。
- 第 17 行：调用 fclose 关闭一个流。此处是将重定向的目标文件关闭。在使用标准 I/O 库进行文件操作时，一定要记得调用 fclose 关闭已打开的流，否则可能出现数据丢失的情况。

10.2 使用标准 I/O 进行文件操作

要对一个流进行操作，必须先打开该流，然后调用标准 I/O 函数进行读写。本节将就如何调用标准 I/O 函数进行文件操作进行详细阐述。

10.2.1 打开关闭流文件

在进行流文件操作前，应先调用 fopen 打开一个流。在对文件操作完成后，应调用 fclose 关闭一个流。这两个函数的声明位于头文件<stdio.h>中，其原型如下所示。

```
#include <stdio.h>
FILE *fopen (__const char *__restrict __filename,__const char *__restrict __modes);
int fclose (FILE *__stream);
```

（1）参数说明如下。

- __filename：输入参数，要打开的文件名称。该名称可以为相对路径或者绝对路径的文件名。
- __modes：输入参数，文件的打开方式。
- __stream：输入参数，要关闭的流结构指针。该指针是由 fopen 等函数打开后返回的。

（2）fopen 返回值说明如下。

- NULL：调用 fopen 失败，可以查看 errno 获取详细错误信息。
- 非 NULL：调用 fopen 成功，返回 FILE 结构指针。

（3）fclose 返回值说明如下。

- EOF：调用 fclose 失败，可以查看 errno 获取详细错误信息。
- 0：调用 fclose 成功。

10.2.2 单字符方式读写

标准 I/O 库提供了多种读写方式，主要包括单字符方式读写、行读写和二进制方式读写。每种读写方式各有其不同的应用场合。单字符方式读写的函数包括 fgetc(getc)、fputc(putc)、getchar、putchar 等，这些函数的声明位于头文件<stdio.h>中，其原型如下所示。

```
#include <stdio.h>
int fgetc (FILE *__stream);
int getc (FILE *__stream);
int fputc (int __c, FILE *__stream);
int putc (int __c, FILE *__stream);
int getchar (void);
int putchar (int __c);
```

（1）参数说明如下。

- __stream：输入参数，指向流的指针，用于定义读或者写的对象。
- __c：输入参数，要向流中写入的字符。

（2）fgetc(getc)、getchar 返回值说明如下。

- EOF：读失败，可能是文件到末尾或者其他原因，检查 errno 获取详细的错误信息。
- 其他：从流中读取到的字符，已被强制转换为 int 型。

（3）fputc(putc)、putchar 返回值说明如下。

- EOF：向流中写入字符失败，可以查看 errno 获取详细错误信息。
- 其他：向流中成功写入的字符，已被强制转换为 int 型。

（4）在调用单字符方式读写函数时，应注意以下几点。

getchar 等价于 getc(stdin)，而 putchar 等价于 putc(c,stdout)。

【范例 10-4】编程实现十六进制格式显示文件内容。其实现过程如示例代码 10-4 所示。

示例代码 10-4

```
#include <stdio.h>                                          /*头文件*/
main(int argc,char *argv[])                                 /*主函数，包含命令行参数的使用*/
{
 int n,c;                                                   /*循环变量定义*/
 FILE *fp;                                                  /*流文件描述符*/
 char buf[101],hexbuf[101];                                 /*输出缓存区定义*/
 if(argc != 2)                                              /*判断是否从命令行输入了文件*/
 {
      fprintf(stderr,"Usage:%s  文件名称\n",argv[0]);   /*用法提示*/
      exit(0);
 }
 if((fp = fopen(argv[1],"r")) == NULL)                      /*以只读的方式打开文件*/
 {
      perror("fopen");                                      /*错误处理*/
      exit(0);
 }
 n = 0;
 memset(buf,0x00,sizeof(buf));                              /*清空缓存区*/
 memset(hexbuf,0x00,sizeof(hexbuf));
 while((c = fgetc(fp)) != EOF)                              /*循环从文件中读取字符*/
 {
       if( n > 15)                                          /*如果读到十六个字节数据，则换行*/
       {
            fprintf(stderr,"%48s    %s\n",hexbuf,buf); /*输出十六进制信息和原文件内容并换行*/
            n = 0;                                          /*计数器复位*/
            memset(buf,0x00,sizeof(buf));                   /*清空缓冲*/
            memset(hexbuf,0x00,sizeof(hexbuf));
```

```
28              }
29              buf[n] =(unsigned char)c;                    /*保存到原文件内容缓存区*/
30              sprintf(hexbuf + 3*n,"%02x ",c);             /*取十六进制值保存到缓存区*/
31              n++;                                         /*计数器值加 1*/
32          }
33          fprintf(stderr,"%-48s     %s\n",hexbuf,buf);     /*输出最后未满十六个字符的缓存数据*/
34          fclose(fp);                                      /*关闭文件*/
35      }
```

【运行结果】经过编译链接，在 shell 下运行上述程序，显示范例 10-3 中生成的文件 stdout.redirect，其输出如下所示。

```
1     54 68 69 73 20 69 73 20 61 20 74 65 73 74 20 6d      This is a test m
2     65 73 73 61 67 65 2e 0a                               essage.
```

【代码解析】在本例中，首先打开要显示的文件，然后循环调用 fgetc 从文件中获取数据，并按照每行 16 个字符的格式将数据显示在屏幕上。在本例中，源代码的详细解释如下。

- 第 1 行：头文件信息。
- 第 7～11 行：检查命令行参数，如果调用程序格式不符，则提示相应信息后退出。
- 第 12～16 行：调用 fopen 打开一个文件。
- 第 17～19 行：初始化变量，清空输出缓存区。
- 第 20～32 行：循环调用 fgetc 从打开的文件中读取单个字符，直到读取到文件尾。
- 第 22～31 行：以 16 个字符的文件内容为一行，将十六进制的文件内容显示到屏幕上。
- 第 33～34 行：当遇到文件尾结束循环时，将缓存区中尚未输出的信息显示到屏幕上。

10.2.3 行方式读写

大多数情况下，在对文本文件或者无格式文件进行读写的过程都是按行进行的。为此，标准 I/O 库提供了按行方式读写的函数。这些函数的声明位于头文件<stdio.h>中，其原型如下所示。

```
#include <stdio.h>
char *fgets (char *__restrict __s, int __n, FILE *__restrict __stream);
int fputs (__const char *__restrict __s, FILE *__restrict __stream);
```

（1）参数说明如下。

- __s：输入/输出参数，读或者写的缓存区。缓存区中是要读写的内容。
- __n：输入参数，要从流中读取的长度，以字节为单位。
- __stream：输入参数，流指针。

（2）fgets 返回值说明如下。

- NULL：表明已读到流的尾部，或者从流中读取数据出错。
- 其他：指向读取到的数据缓存区头部的指针。

（3）fputs 返回值说明如下。

- EOF：向流写入数据失败，可以查看 errno 获取详细错误信息。
- 非负整数：向流中写入数据成功。

（4）在调用行方式读写函数时，需要注意以下几点。

- fgets 在读取数据成功后，将读取指定长度的数据至缓存区，并且在数据的最后附加字符串结束符“\0”。
- fgets 的输入参数__n 只是指定了要读取到的数据行的最大长度。如果文件中某行数据不够长，则 fgets 并不认为是失败，而是将该行数据全部读取；如果输入的参数__n 太小，而文件某行数据过长，则将会对该行数据分多次进行读取。因此，在编程过程中，应估算要读取的文件内容，为函数 fgets 选取一个合适的读取长度。

除 fgets 和 fputs 外，标准 I/O 库还提供另外两个函数用于向标准输入（stdin）、标准输出设备（stdout）写或者读。这两个函数是 gets 和 puts，原型如下所示。

```
#include <stdio.h>
char *gets (char *__s);
int puts (__const char *__s);
```

参数及返回值说明如下。

参见 fgets 和 fputs。函数 puts 向标准输出设备写入数据后，会附加写入一个换行符。而 gets 函数从标准输入读取一行数据时，并未对缓冲区__s 的长度是否足够进行检查。如果从键盘输入过长的数据，将导致不可预料的结果。因此，在实际编程过程中，经常用 fgets 和 fputs 代替对 gets 和 puts 的使用。

【范例 10-5】编程实现按行读取文件的内容并输出至屏幕。其实现过程如示例代码 10-5 所示。

示例代码 10-5

```
1   #include <stdio.h>                                          /*头文件*/
2   main(int argc,char *argv[])                                 /*主函数，包含命令行参数的使用*/
3   {
4       FILE *fp;                                               /*流文件描述符*/
5       char buf[256];                                          /*输出缓冲区*/
6       if(argc != 2)                                           /*判断是否从命令行输入了文件名*/
7       {
8           fprintf(stderr,"Usage:%s  文件名称\n",argv[0]);     /*提示用法*/
9           exit(0);
10      }
11      if((fp = fopen(argv[1],"r")) == NULL)                   /*以只读的方式打开文件*/
12      {
13          perror("fopen");                                    /*错误处理 */
14          exit(0);
15      }
16      memset(buf,0x00,sizeof(buf));
17      while((fgets(buf,sizeof(buf),fp)) != NULL)              /*按照行的方式循环读取文件数据*/
18      {
19          printf("%s",buf);                                   /*输出读取到的每行数据*/
20          memset(buf,0x00,sizeof(buf));                       /*清空缓冲区*/
21      }
22      fclose(fp);                                             /*关闭文件描述符*/
23  }
```

【运行结果】经过编译链接，在 shell 下运行上述程序，显示范例 10-3 中生成的文件 stdout.redirect，其输出如下所示。

```
1    This is a test message.
```

【代码解析】在本例中，首先打开要显示的文件，然后循环调用 fgets 从文件中按行读取数据，并将数据显示在屏幕上。在本例中，源代码的详细解释如下。

- 第 1 行：头文件信息。
- 第 6～10 行：检查命令行参数，如果调用程序格式不符，则提示相应信息后退出。
- 第 11～15 行：调用 fopen 打开一个文件。
- 第 16 行：初始化变量，清空输出缓冲区。
- 第 17～21 行：循环调用 fgets 从打开的文件中按行读取数据，直到读取到文件尾。
- 第 19 行：将读取到的文件内容显示到屏幕上。
- 第 20 行：清空缓冲区，为读取下一行数据做准备。

10.2.4 二进制方式读写

上一小节介绍的按照行方式读写文件，适用于大多数的文本格式的文件。但在有些情况下，文件内容是需要按照一定格式存放的，如可能是存储了某个结构体等。此时如果用行方式读写的话，读取到的数据将无法恢复成原有的格式。这种情况下，标准 I/O 库提供了二进制方式的读写功能。

二进制方式读写文件类似于 Windows 环境下编程中的序列化的概念。在写入文件时进行序列化，在读出文件时进行反序列化，以获取到有格式的数据内容。标准 I/O 库提供的这两个函数为用于二进制读的 fread 和用于二进制写的 fwrite。fread 用于从文件中读取指定个数的记录。这里的记录并不代表一行，而是指存储于文件中的带格式的一条数据。而 fwite 用于向文件中写入指定个数的记录。二进制读写最常用的场合是对一个结构进行读写。

这两个函数的声明位于头文件<stdio.h>中，原型如下所示。

```
#include <stdio.h>
size_t fread (void *__restrict __ptr, size_t __size,size_t __n, FILE *__restrict __stream);
size_t fwrite (__const void *__restrict __ptr, size_t __size,size_t __n, FILE *__restrict __s);
```

（1）参数说明如下。

- __ptr：输入/输出参数，用于存储读取或者写入的数据内容。对于 fread 来说，该参数是作为输入参数出现；对于 fwrite 来说，该参数是用于输出参数。
- __size：输入参数，要读取的每个记录的长度。
- __n：输入参数，要读取的记录个数。
- __stream：输入参数，指向流的指针。

（2）返回值说明如下。

- 0：调用失败，可能的原因包括进行读操作时已到文件尾部，或者读取/写入发生错误。
- 其他：调用成功，返回值表示读取或者写入的记录个数。

（3）关于二进制方式读写，应该注意以下几点。

- 二进制方式读写的基本单位是记录，而不是行或者字符。
- 在同一操作系统环境下二进制读写是非常稳定的。如果在不同的操作系统或者编译器下，由于同一结构体中的成员偏移量可能会发生变化，所以二进制读写可能返回

不正常的结果。

【范例 10-6】编程实现对结构进行读写。其实现过程如示例代码 10-6 所示。

示例代码 10-6

```
#include <stdio.h>                                          /*头文件*/
typedef struct person{                                      /*自定义结构数据类型*/
    char name[10];                                          /*姓名*/
    int  age;                                               /*年龄*/
    float stature;                                          /*身高*/
} PERSON;                                                   /*数据类型名称*/
main()
{
    PERSON per;                                             /*定义结构变量*/
    FILE *fp;                                               /*流文件描述符*/
    memset(&per,0x00,sizeof(PERSON));                       /*清空结构*/
    strcpy(per.name,"Jack");                                /*向结构赋值*/
    per.age = 20;
    per.stature = 180.2;
    if((fp = fopen("./person.bin","w")) == NULL)            /*打开文件用于写*/
    {
        perror("fopen");                                    /*错误处理*/
        exit(0);
    }
    if(fwrite(&per,sizeof(PERSON),1,fp) <= 0)               /*向文件中写入一条记录*/
    {
        perror("fwrite");
        exit(0);
    }
    fclose(fp);                                             /*关闭文件*/
    if((fp = fopen("./person.bin","r")) == NULL)            /*打开文件用于读*/
    {
        perror("fopen");                                    /*错误处理 */
        exit(0);
    }
    memset(&per,0x00,sizeof(PERSON));                       /*清空结构变量*/
    if(fread(&per,sizeof(PERSON),1,fp) <= 0)                /*从文件中读取一条记录*/
    {
        perror("fread");                                    /*错误处理*/
        exit(0);
    }
    fclose(fp);                                             /*关闭文件*/
    printf("姓名=%s  年龄=%d  身高=%.2f\n",per.name,per.age,per.stature);     /*输出数据*/
}
```

【运行结果】经过编译链接，在 shell 下运行上述程序，其输出如下所示。

```
姓名=Jack  年龄=20  身高=180.20
```

【代码解析】在本例中，首先以只写的方式打开文件，然后 fwrite 向文件中写入一条记录，记录的格式为结构 PERSON。然后以只读的方式打开刚刚生成的文件，调用 fread 从文件中读取数据至结构类型 PERSON 中。最后，将读取到的内容格式化输出至屏幕。在本例中，源代码的详细解释如下。

- 第 1 行：头文件信息。
- 第 2～6 行：自定义类型 PERSON，该类型用于对文件进行二进制读写。
- 第 11～14 行：初始化结构，设置姓名、年龄及身高等数据。
- 第 15～19 行：调用 fopen 以只写的方式打开文件。
- 第 20～23 行：调用 fwrite 向文件中写入一条记录，记录的内容为结构 PERSON。
- 第 6～10 行：检查命令行参数，如果调用程序格式不符，则提示相应信息后退出。
- 第 25～29 行：调用 fopen 以只读的方式打开文件。
- 第 30 行：清空结构内容，以便存储后续调用 fread 时返回的数据。
- 第 31～35 行：调用 fread 从文件中读取一条记录，将读取到的数据存储到 PERSON 类型的结构中。
- 第 36～37 行：关闭打开的文件，并格式化输出读取到的数据内容。

10.2.5 格式化输入/输出

在向流文件输出数据或者从流文件读取数据的过程中，可能需要按照某种格式对数据格式化后进行输入/输出，此时需要用到格式化输入/输出函数。Linux 提供了多种格式化输入/输出函数，用于格式化输出的有 printf、fprintf、sprintf，用于格式化输入的有 scanf、fscanf、sscanf。

fprintf 用于向流文件（或者设备）格式化输出数据，fscanf 用于从流文件（或者设备）按照格式读取数据。printf 函数、sprintf 函数的功能与 fprintf 类似。只不过，printf 操作的对象是标准输出设备（stdout），sprintf 操作的对象是进程指定的缓冲区。同样，scanf 函数、sscanf 函数的作用与 fscanf 也是类似的。这几个函数的声明位于头文件<stdio.h>中，其原型如下所示。

```
#include <stdio.h>
int printf (__const char *__restrict __format, ...);
int fprintf (FILE *__restrict __stream,__const char *__restrict __format, ...);
int sprintf (char *__restrict __s,__const char *__restrict __format, ...);
int scanf (__const char *__restrict __format, ...);
int fscanf (FILE *__restrict __stream,__const char *__restrict __format, ...);
int sscanf (__const char *__restrict __s,__const char *__restrict __format, ...);
```

（1）参数说明如下。

- __stream：输入参数，流文件指针。
- __format：输入参数，指定输入/输出的格式串。如“%s %d”等。
- __s：输入/输出参数，用于 sprintf 和 sscanf 进行格式化输入/输出。在 sprintf 中，该参数用于保存格式化输出后的数据。在 sscanf 中，该参数存储了将被格式化的数据。
- ...：输入/输出参数，该参数是可变参数，参数的个数可以为 0 或者正整数，视__format 中的内容而定。在格式化输出中，该参数指定各个输入项。在格式化输入中，该参数指定格式化后数据存储的位置。

（2）标准输入函数返回值说明如下。

- EOF：调用标准输入函数失败，可能已到达文件尾或者其他错误。可以查看 errno 获取详细错误信息。

- 其他：返回输入的项数，就是...参数所指定的参数的个数。

（3）标准输出函数返回值说明如下。

- <0：表明调用标准输出函数失败，可以查看 errno 获取详细的错误信息。
- 其他：输出的字节数。

（4）在调用格式化输入/输出函数时，要注意以下几点。

- 在调用格式化输入函数时，应该注意参数...是输入参数，该参数在调用格式化输入函数成功后，将保存格式化后的数据内容。所以，该参数应该为指针类型。如果要将变量保存到整形（或者其他非字符串类型）时，应该注意变量前加&符号。
- 在调用 sprintf 时，向指定缓冲区格式化输出数据后，系统将自动在该缓冲区末尾追回字符串结束符“\0”。

【范例 10-7】编程实现格式化输入输出。其实现过程如示例代码 10-7 所示。

示例代码 10-7

```
1   #include <stdio.h>                                          /*头文件*/
2   main()                                                      /*主函数*/
3   {
4       int i,j;                                                /*循环变量定义*/
5       char *s1="123 aaa 1";                                   /*定义字符串指针，并赋初始值*/
6       char *s2="a1|a2|a3";                                    /*定义字符串指针，并赋初始值*/
7       char buf1[11];                                          /*一维数组缓冲区*/
8       char buf2[3][11];                                       /*二维数组*/
9       memset(buf1,0x00,sizeof(buf1));                         /*清空缓冲区*/
10      sscanf(s1,"%d %s %d",&i,buf1,&j);          /*从串 s1 中格式化输入到变量 I,buf1 和 j 中*/
11      printf("i=%d j=%d buf1=%s\n",i,j,buf1);                 /*输出格式化输入后的值*/
12      sscanf(s2,"%[^|]|%[^|]|%[^|]",buf2[0],buf2[1],buf2[2]); /*从串 s2 中格式化输入到二维数组 buf2 中*/
13      for(i=0;i<3;i++)                                        /*循环输出二维数组的值*/
14          printf("buf2[%d]=%s\n",i,buf2[i]);
15  }
```

【运行结果】经过编译链接，在 shell 下运行上述程序，其输出如下所示。

```
1   i=123 j=1 buf1=aaa
2   buf2[0]=a1
3   buf2[1]=a2
4   buf2[2]=a3
```

【代码解析】本例演示了通过 sscanf 从缓冲区中格式化输入数据至数个变量的过程。在本例中，源代码的详细解释如下。

- 第 1 行：头文件信息。
- 第 5 行：定义字符串 s1，字符串中包含以空格分隔的 3 个部分。
- 第 6 行：定义字符串 s2，字符串是以“|”为分隔符的。
- 第 10 行：调用 sscanf 将字符串 s1 的内容格式化输入至变量 i、j、buf1 中。注意：后面的变量参数应该使用指向变量的指针。buf1 实际上也是一种指针，等价于 &buf1[0]。
- 第 11 行：调用 printf 将获取的变量 i、j、buf1 的内容格式化输出至标准输出。
- 第 12 行：调用 sscanf 将以“|”为分隔符的字符串格式化输入至字符串数组 buf2 中。

注意：此处用%[^|]表示获取从当前位置至“|”符号前的字符串，在%[^|]后面的“|”符号表示要空过该分隔符。

- 第 13～14 行：输出格式化获取的字符串数组的值。

格式化输入输出支持多种格式，这些格式都是用一种名称为“格式控制符”的符号控制的。格式控制符以字符“%”为前导符，其格式为“%”+“修饰符”+“格式符”。

主要的修饰符包括以下几种。

- -：减号，表示按照左对齐的方式输出，如果没有该修饰符，将按照右对齐方式输出。
- 0：如果指定该修饰符，在输出数据时如果有空格，则填充 0，否则不填。
- m.n：m 指定了输出数据的长度。如果实际数据内容小于 m，则左端补空格。如果大于等于 m，则按照数据的实际长度输出。n 表示数据的精度，当输出的数据为浮点型或双精度类型时，该参数指定了输出数据的精度。如果不指定，默认的精度为小数点后 6 位。m、n 可以不同时指定。
- l：该修饰符在用于输出长整型变量指定。
- h：该修饰符在用于输出短整型变量指定。

主要的格式符包括以下几种。

- d：十进制方式整型数据输入/输出。
- x：十六进制方式整型数据输入/输出。
- o：八进制方式整型数据输入/输出。
- s：字符串输入/输出。
- u：无符号整型数据输入/输出。
- c：字符数据输入/输出。
- f：浮点型数据输入/输出。

10.2.6 在流文件中定位

在第 9 章文件操作中介绍过，可以在文件中根据要求定位文件读写指针。在标准 I/O 库中，同样支持文件读写指针的定位。这些函数包括获取文件当前读写指针的 ftell、用于定位文件读写指针的 fseek 和用于定位文件指针到文件头部的 rewind，它们位于头文件<stdio.h>中，其原型如下所示。

```
#include <stdio.h>
int fseek (FILE *__stream, long int __off, int __whence);
long int ftell (FILE *__stream);
void rewind (FILE *__stream);
```

（1）参数说明如下。

- __stream：输入参数，流指针。
- __off：输入参数，指定以参数__whence 为基准的偏移量。在__whence 不是 SEEK_SET 时，该参数可以为负数。
- __whence：输入参数，指定了定位的起始位置。

（2）fseek 返回值说明如下。

- -1：调用 fseek 失败，可查看 errno 获取详细错误信息。
- 0：调用 fseek 成功。

（3）ftell 返回值说明如下。

- -1：调用 ftell 失败，可查看 errno 获取详细错误信息。
- 其他：返回文件的当前读写位置。

【范例 10-8】编程读取文件/etc/passwd 文件最后的 10 个字符。其实现过程如示例代码 10-8 所示。

示例代码 10-8

```
1   #include <stdio.h>                                    /*头文件*/
2   main()                                                /*主函数*/
3   {
4       FILE *fp;                                         /*流文件描述符*/
5       char buf[11];                                     /*缓冲区*/
6       if((fp = fopen("/etc/passwd","r")) == NULL) /*以只读的方式打开文件*/
7       {
8           perror("fopen");                              /*错误处理*/
9           exit(0);
10      }
11      fseek(fp,-10,SEEK_END);                           /*定位文件的读写指针到文件尾前 10 个字符*/
12      memset(buf,0x00,sizeof(buf));                     /*清空缓冲区*/
13      fread(buf,10,1,fp);                               /*从当前位置读取 10 个字节*/
14      printf("%s\n",buf);                               /*输出读到的内容*/
15      fclose(fp);                                       /*关闭文件*/
16  }
```

【运行结果】经过编译链接，在 shell 下运行上述程序，其输出如下。

```
1   h
2   +:::::
```

【代码解析】本例中，首先以只读方式打开文件/etc/passwd，然后调用 fseek 将文件的读写指针定位到文件末尾前的 10 个字符，最后调用 fread 从文件当前位置读取 10 个字符并输出。在本例中，源代码的详细解释如下。

- 第 1 行：头文件信息。
- 第 6～10 行：调用 fopen 以只读方式打开文件/etc/passwd。
- 第 11 行：调用 fseek 将文件的读写指针定位到文件末尾前 10 个字节。
- 第 12～15：从文件当前位置读取 10 个字节的数据，并输出至屏幕。

10.3　综合实例

编程实现一个简单的员工档案管理系统，可以实现简单的员工资料增加、删除及查询等。要求采用二进制文件保存员工的资料信息。下面详细介绍该实例的代码过程。

（1）头文件信息的代码如下所示。

```
1   #include <stdio.h>                                    /*头文件*/
2   #include <stdlib.h>
```

```
3   #define ARFILE "./usr.ar"                          /*宏定义  档案文件的名称*/
4   struct arstruct                                    /*员工资料的结构定义*/
5   {
6       char name[10];                                 /*姓名*/
7       int age;                                       /*年龄*/
8       char tele[21];                                 /*手机*/
9   };
```

（2）实现删除功能的代码如下所示。

```
10  /*删除一个员工资料 通过复制档案库完成，要删除的员工资料不进行复制*/
11  void removeuser()
12  {
13      char name[10];                                 /*员工姓名，根据姓名进行检索*/
14      struct arstruct ar;                            /*员工资料结构变量*/
15      FILE *fp;                                      /*流文件描述符，用于打开档案文件读*/
16      FILE *fpn;                                     /*流文件描述符，用于打开新文件写*/
17      if((fpn = fopen("./tmpfile","w")) == NULL)     /*以写的方式打开新文件*/
18      {
19          return;
20      }
21      if((fp = fopen(ARFILE,"r")) == NULL)           /*以只读的方式打开档案文件*/
22      {
23          return;
24      }
25      memset(&ar,0x00,sizeof(ar));                   /*清空结构*/
26      printf("请输入员工姓名:");                     /*输出提示信息*/
27      memset(name,0x00,sizeof(name));                /*清空 name 缓存*/
28      scanf("%s",name);                              /*从键盘输入姓名*/
29      while(fread(&ar,sizeof(ar),1,fp) == 1) /*循环复制员工档案，但与输入的姓名相匹配的不复制*/
30      {
31          if(strcmp(name,ar.name) != 0)              /*比较姓名是否与档案中的姓名相同*/
32          {
33              fwrite(&ar,sizeof(ar),1,fpn);          /*不相同，则复制*/
34          }
35          memset(&ar,0x00,sizeof(ar));               /*清空结构*/
36      }
37      fclose(fp);                                    /*关闭文件*/
38      fclose(fpn);                                   /*关闭文件*/
39      remove(ARFILE);                                /*删除原档案文件*/
40      rename("./tmpfile",ARFILE);                    /*将新生成的 tmpfile 重命名为档案文件*/
41      printf("删除员工资料成功,按任意键继续...\n");  /*输出提示信息*/
42      getchar();                                     /*等待用户按任意键*/
43      getchar();
44  }
```

（3）实现查询功能的代码如下所示。

```
45  /*根据姓名查询员工资料*/
46  void queryuser()
47  {
48      int found;                                     /*标志，是否发现匹配的员工资料*/
49      char name[10];                                 /*缓存区，用于保存输入的姓名*/
50      struct arstruct ar;                            /*员工资料结构变量*/
51      FILE *fp;                                      /*流文件描述符*/
```

```
52      if((fp = fopen(ARFILE,"r")) == NULL)          /*以只读方式打开档案文件*/
53      {
54          return;
55      }
56      memset(&ar,0x00,sizeof(ar));                  /*清空结构*/
57      printf("请输入员工姓名:");                    /*输入要查询的员工姓名*/
58      memset(name,0x00,sizeof(name));
59      scanf("%s",name);
60      found=0;                                      /*置查找成功标志为假*/
61      while(fread(&ar,sizeof(ar),1,fp) == 1)        /*循环从文件中读取记录*/
62      {
63          if(strcmp(name,ar.name) == 0)             /*比较名字是否相等*/
64          {
65              found=1;                              /*置查找成功标志为真*/
66              break;                                /*查找成功，中断循环*/
67          }
68          memset(&ar,0x00,sizeof(ar));              /*清空结构*/
69      }
70      fclose(fp);                                   /*关闭文件*/
71      if(found)                                     /*查找成功，输出员工信息*/
72      {
73          printf("姓名=%s\n",ar.name);              /*输出员工的资料信息*/
74          printf("年龄=%d\n",ar.age);
75          printf("手机=%s\n",ar.tele);
76      }
77      else  /*查找失败，提示错误信息*/
78      {
79          printf("没有员工%s 的数据\n",name);       /*输出提示信息*/
80      }
81      getchar();                                    /*等待用户按任意键后再继续后续的操作*/
82      getchar();
83  }
```

（4）实现增加功能的代码如下所示。

```
84  /*增加新员工资料*/
85  void insertuser()
86  {
87      struct arstruct ar;                           /*员工资料结构变量*/
88      FILE *fp;                                     /*流文件描述符*/
89      if((fp = fopen(ARFILE,"a")) == NULL)          /*以追加的方式打开档案文件*/
90      {
91          return;
92      }
93      memset(&ar,0x00,sizeof(ar));                  /*清空结构*/
94      printf("请输入员工姓名:");                    /*输入员工资料信息*/
95      scanf("%s",ar.name);                          /*从键盘输入姓名*/
96      printf("请输入员工年龄:");                    /*提示信息*/
97      scanf("%d",&(ar.age));                        /*从键盘输入年龄*/
98      printf("请输入员工手机号码:");                /*提示信息*/
99      scanf("%s",ar.tele);                          /*从键盘输入员工手机号*/
100     if(fwrite(&ar,sizeof(ar),1,fp) < 0)           /*写入员工档案文件*/
101     {
102         perror("fwrite");                         /*错误处理*/
103         fclose(fp);                               /*关闭文件*/
```

```
104             return;
105         }
106         fclose(fp);                                          /*关闭文件*/
107         printf("增加新员工成功,按任意键继续...\n");
108         getchar();                                           /*等待用户按任意键再继续后续的操作*/
109         getchar();
110 }
```

（5）输出程序结果的代码如下所示。

```
111 main()                                                      /*主函数*/
112 {
113     char c;                                                  /*字符变量定义*/
114     while(1)                                                 /*循环显示菜单*/
115     {
116         printf("\033[2J");                                   /*清屏*/
117         printf("       *员工档案管理系统*\n")                /*显示菜单*/;
118         printf("-------------------------\n");
119         printf("       1.录入新员工档案          \n");
120         printf("       2.查看工档案              \n");
121         printf("       3.删除工档案              \n");
122         printf("       0.退出系统                \n");
123         printf("-------------------------\n");
124         switch((c=getchar()))                                /*调用 switch 判断从键盘接收的输入*/
125         {
126             case '1':                                        /*增加员工*/
127                 insertuser();                                /*调用 Insertuser 函数*/
128                 break;
129             case '2':                                        /*查询员工*/
130                 queryuser();                                 /*调用 queryuser 函数*/
131                 break;
132             case '3':
133                 removeuser();                                /*删除员工*/
134                 break;
135             case '0':                                        /*退出系统*/
136                 return 0;
137             default:                                         /*用户输入其他选择*/
138                 break;
139         }
140     }
141 }
```

10.4 小结

标准 I/O 库为 Linux 操作系统下文件的编程提供了更为简便的方法。通过本章的学习，读者了解了标准 I/O 库的编程原理及相关函数的使用方法。其中，本章最重要的内容在于如何使用标准 I/O 进行文件操作。最后，提醒用户注意，不建议将标准 I/O 函数与文件操作 API 在同一个程序中混合使用，否则有可能导致不可预料的结果。

第 11 章　界面程序设计——Qt

一个成功的应用程序，必须要有友好美观的界面支持，这一点也正是 Windows 应用程序大行其道的一个重要原因。在 Linux 系统下，传统的应用程序展现给用户的是字符界面，操作界面不够美观，对各种外设的支持也不够全面。为提供更为美观、可操作性更强的程序界面，Linux 提供了多种功能强大的界面程序设计工具包，Qt 就是其中最为著名的一个。

本章从 Qt 开发包的安装讲起，介绍了 Qt 开发的基本过程。通过本章的学习，读者应重点掌握如下内容。

- 熟悉 Qt 集成开发环境的使用。
- 掌握用 Qt 开发桌面应用程序的基本过程。

11.1　Qt 简述

Qt 是一个支持多平台（Windows、Linux 等）的面向对象（使用 C++语言）的图形用户界面应用程序框架，它为应用程序开发人员提供了建立图形界面程序所需要的全部功能。Qt 的开发过程是完全面向对象的。本书假设读者已对面向对象的 C++语言已有初步了解。如果读者对面向对象的开发过程没有概念，可以在阅读本章过程中参照一些面向对象方面的书籍。

11.1.1　Qt 的组成

Qt 有多个发布版本，主要包括企业版（专业版）、自由版。其中自由版是为开发开放源码软件而免费提供的。Qt 的企业版主要包括以下模块。

- 基本模块：包括核心、窗口组件、对话框、图形用户界面工具包及应用类库等。
- 集成开发环境：即 IDE。
- 工作区模块：提供了对多文档界面（MDI）的支持。
- OpenGL 三维图形模块：提供对 OpenGL 开发的支持。
- 网络模块：提供对套接口、TCP/UDP、DNS 等网络操作的支持。
- 表格模块：提供对电子表格进行编辑的功能。
- XML 模块：提供对 XML 进行解析的功能。
- 数据库模块：提供对数据库访问的支持。

11.1.2　Qt 的优点

Qt 被广大程序员认可的一个重要原因，就是其支持多种操作系统平台。如果读者在

Windows 平台下从事开发工作，可能会使用 Microsoft Foundation Classes（MFC）或者 C#.Net 等面向对象的编程工具。但是，使用 MFC 或者 C#开发完成的产品只能在 Windows 系统下运行，要将这些程序移植至 Linux 系统下代价是非常昂贵的。要编写在 Windows 系统下或者 Linux 系统下可移植性更强的工具就是 Qt。目前 Qt 所支持的主要操作系统包括以下几种。

- Windows 系列，如 Windows95/98、WindowsXP、Windows2003 等。
- UNIX 系列，如 HP-UX、IBM AIX、Sun Solaris 等。
- Linux 系列，如 RedHat Linux、SuseLinux、DebianLinux 等。

除 Qt 良好的可移植性之外，Qt 的另外一个优点就是生成的程序运行速度快。使用过 GUI 从事界面开发的程序员应当了解，GUI 程序运行速度通常不尽如人意。Qt 在运行速度方面具有很好的表现。这一方面是由于 Qt 的设计人员对 Qt 进行了长时间优化的结果。另外，Qt 运行速度快也是基于其实现方式。Qt 是一个 GUI 仿真开发工具包，不依赖于任何其他工具包。它通过调用不同硬件平台上的低级绘图函数实现图像处理，这使得 Qt 生成的可执行程序具有更高的运行速度。

另外，Qt 也是完全面向对象的，它具有一切面向对象编程环境的优点。同时，Qt 也是易用的，它采用完全可视化的开发环境，支持“所见即所得”的软件开发方式。

11.2 Qt 开发包的安装

要使用 Qt 开发程序，首先应该在系统中安装该开发工具包。在缺省情况下，大多系统不会安装 Qt 开发包，需要手工进行安装工作。通常情况下，Qt 安装包都包含在 Linux 的安装介质中，可以通过 yast 进行安装，输入代码如下所示。

```
#yast
```

（1）程序中出现 yast 的安装界面，如图 11-1 所示。

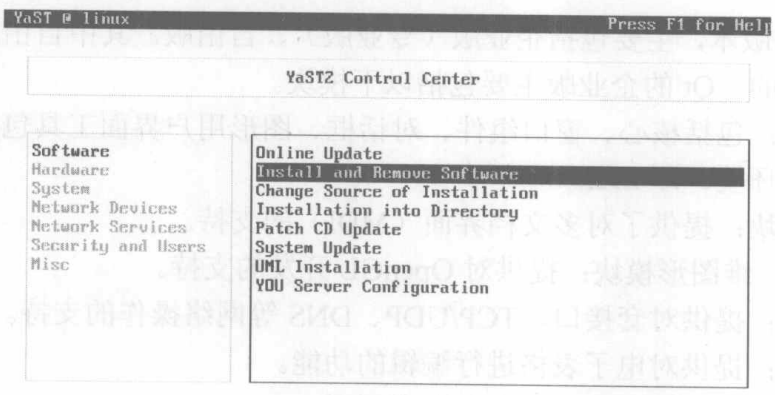


图 11-1 yast 安装界面

（2）在上述界面中，用<Tab>键选择“Install and Remove Software”选项，单击<Enter>键后出现的界面如图 11-2 所示。

（3）在界面中选择“search”选项，然后指定搜索的名称为“Qt”，可以看到结果，如图 11-3 所示。

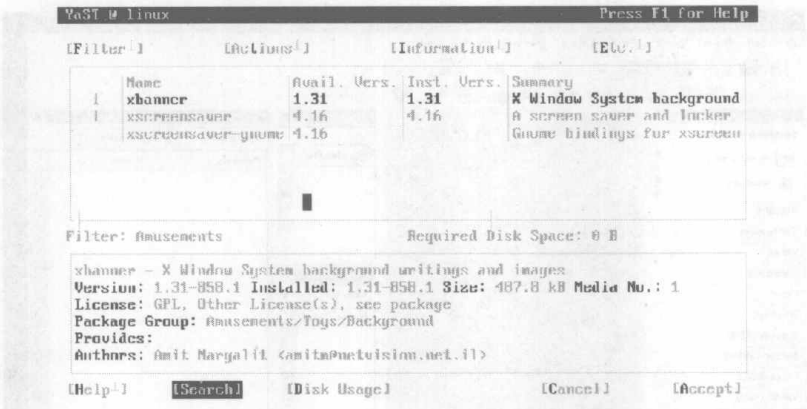


图 11-2　安装/卸载软件界面

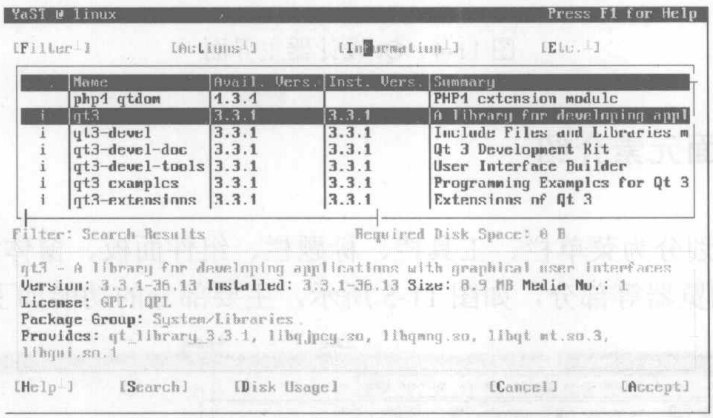


图 11-3　搜索 Qt 安装包界面

Qt 安装包包含若干模块，如开发库、开发文档、开发工具等。为简便起见，建议用户选择所有 Qt 开发的安装包。选中的方式是用方向键将光标移动到相应行，然后按<Space>键选择。选中后，相应行前面应该出现一个加号。如果在未进行选择的情况下，相应行前面已经出现“i”符号，表明该包已安装过，无需再安装。选中安装包后，按<Tab>键移动光标至“Accept”，然后按<Enter>键。系统将提示插入光盘，后续操作按照提示一步步完成即可。

提示：本书是以 SuseLinux 9 为例进行描述，其他版本的 Linux 系统下 Qt 的安装过程大同小异，可以参照相应操作系统的说明文档。

11.3　Qt 集成开发环境介绍

Qt 开发工具包提供了功能强大的集成开发环境：Qt 设计器。如果读者在 Windows 系统下用 VisualStudio.Net 或者其他面向对象的开发工具，对于该环境一定不会陌生。

11.3.1　启动设计器

在命令提示符下运行 designer 即可启动设计器，其主界面如图 11-4 所示。

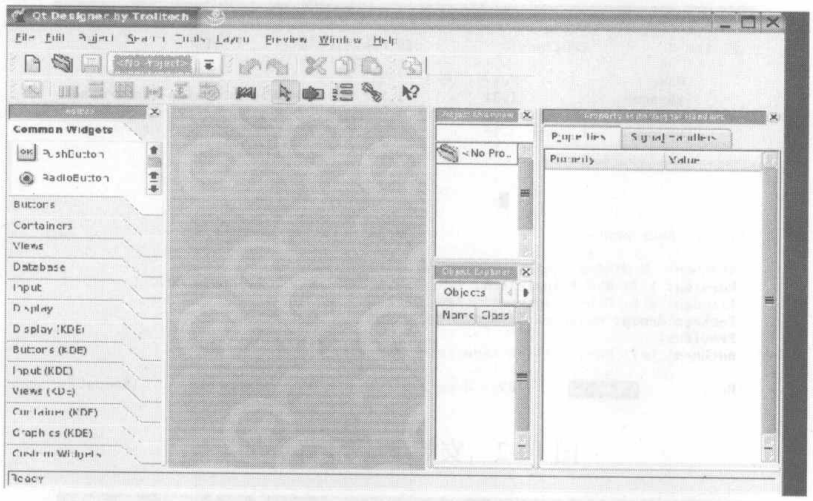


图 11-4　Qt 设计器主界面

11.3.2　设计器界面元素介绍

整个设计器可以划分为菜单栏、工具栏、标题栏、组件面板、窗体设计窗口、对象属性/事件编辑器、对象浏览器等部分，如图 11-5 所示，主要部分的功能如下。

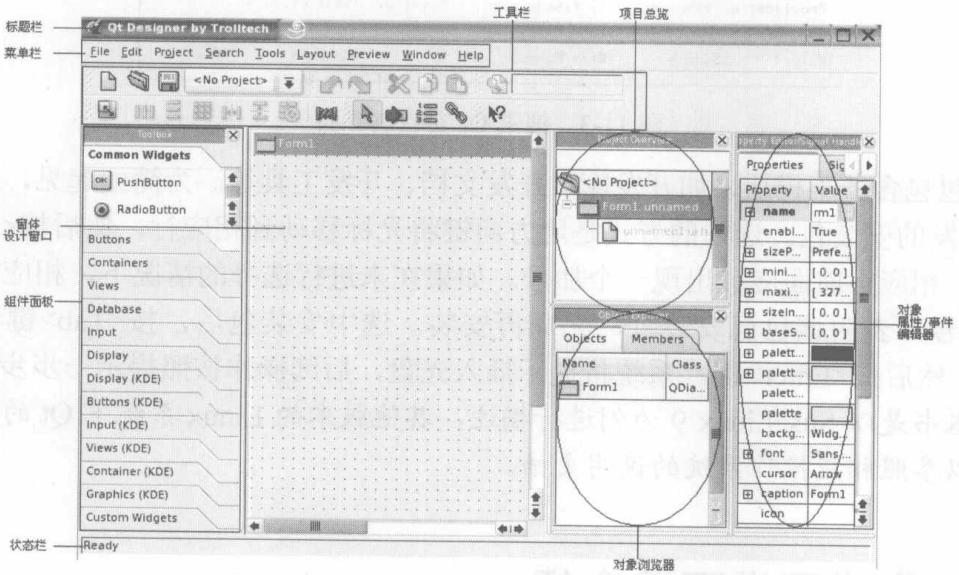


图 11-5　设计器界面元素

- 菜单栏：通过菜单栏可以执行设计器中大部分的命令，如创建新项目、打开已有项目等。菜单栏中包含的菜单组包括文件、编辑、项目、搜索、工具、设计、预览、窗口及帮助，各个菜单组下都包含了相应的菜单项。
- 工具栏：工具栏提供了快速执行部分功能的方法。通过单击工具栏中的按钮，可以达到与执行菜单项相同的效果。当光标悬停在相应工具栏时，将弹出工具栏的提示信息。
- 组件面板：组件面板提供了开发应用程序常用的组件，如按钮、文本框等。这些组

件可以拖拉至窗体设计窗口中，达到“所见即所得”的效果。

- 窗体设计窗口：窗体设计窗口是可视化设计的核心部分。可以通过拖拉的方式，将组件面板中的相应组件拖拉到窗体设计器中。当建立新的项目时，窗体中将产生默认的窗口，其标题为“Form1”，可以通过属性编辑器修改其标题。
- 对象属性或事件编辑器：该部分定义了目标对象（窗口及各种组件等）的属性及组件的事件处理。

11.4　Qt 程序开发

Qt 程序的开发是一个非常复杂的过程，其采用的是面向对象的开发方式。Qt 设计器工具支持“所见即所得”的开发方式，支持各种窗口组件。其功能之强大，可以与 Windows 系统下的 Visual Studio 相媲美。受篇幅所限，本书并不对其内容进行详解，而是以一个简单的 helloworld 示例进行介绍。读者如果有兴趣，可以参考 Qt 专门开发手册了解更详细的内容。

11.4.1　建立新项目

在 Qt 设计器中选择菜单栏中的【file】→【new】命令，然后选择 C++项目文件，在弹出的新建项目窗口中输入项目名称“helloworld”，单击“OK”按钮建立新项目，如图 11-6 所示。

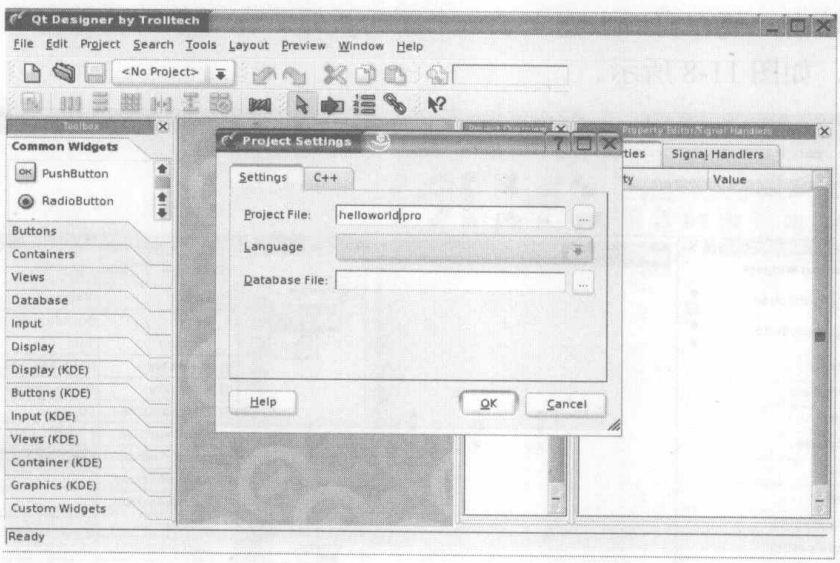


图 11-6　新建项目窗口

11.4.2　设计窗口

在 Qt 程序设计中，用户所需要做的第一件事就是创建一个主窗口，这个主窗口可以是

从 QWidget 或者 QDialog 派生的。在一个 Qt 程序中，可以创建多个窗口，但是只能有一个主窗口。在 Qt 设计器中选择菜单栏中的【file】→【new】命令，然后选择“Dialog”，创建一个新的对话框窗口。新建窗口的标题为 Form1，在对象属性或事件窗口中将其修改为 HelloWorld，如图 11-7 所示。

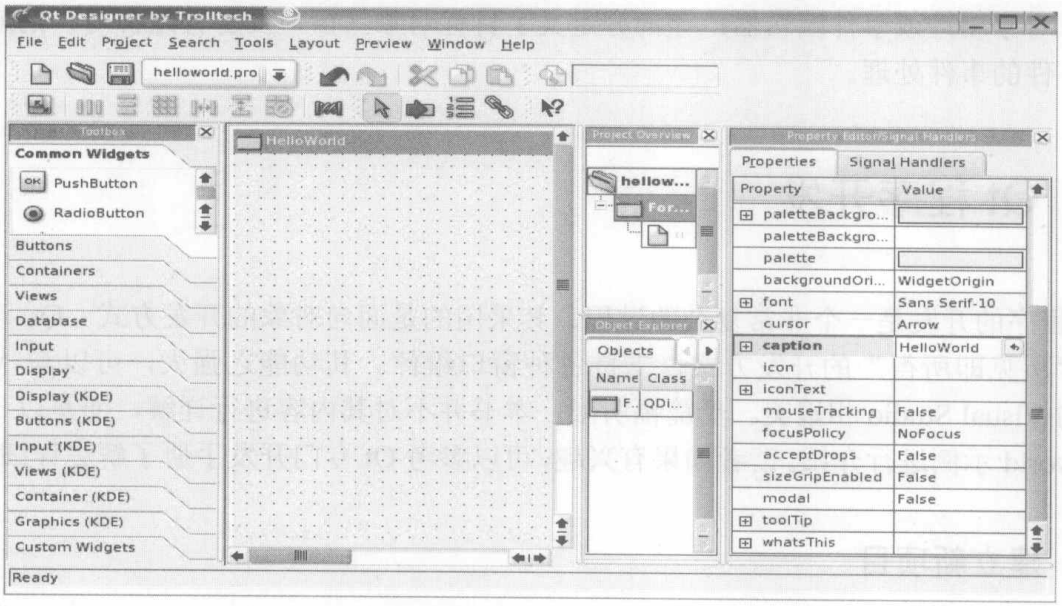


图 11-7　新建窗口

窗口建立完成后，可以从组件面板中拖动组件至窗口，并设置组件的相应属性。本例中，需要一个“确定”按钮。首先在组件面板中选中 PushButton，然后移动光标至新建的 dialog 窗口，在窗口中单击，便在窗口上建立了一个新的按钮。在对象的属性窗口中，修改其 text 属性为“确定”，如图 11-8 所示。

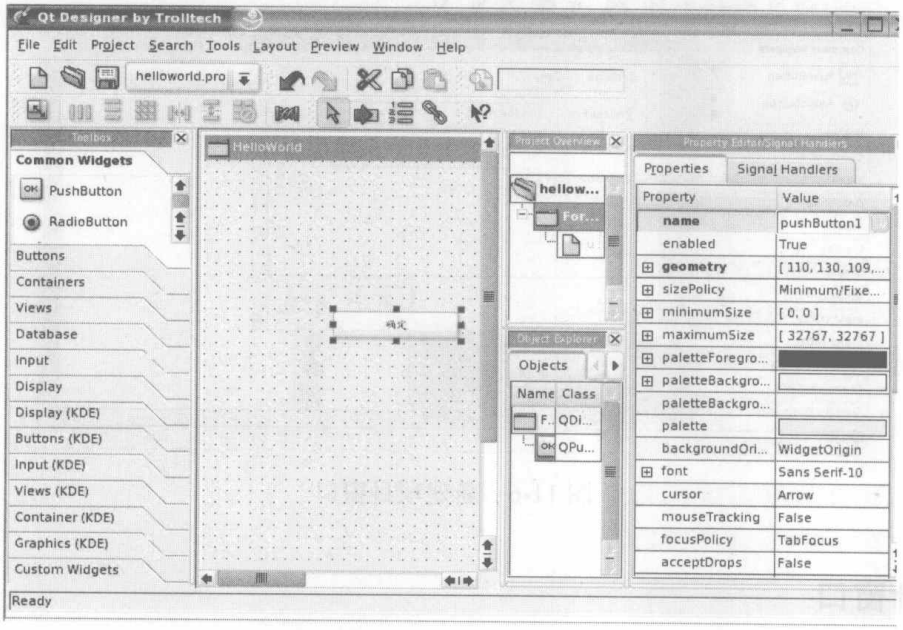


图 11-8　增加组件

经过上述操作后，系统将自动生成一个类 Form1，其实现代码位于当前工作目录的.ui 子

目录下，自动生成的代码文件分别为 form1.cpp 和 form1.h。其中 form1.cpp 的内容如下所示。

```
1   #include "form1.h"
2   #include <qvariant.h>
3   #include <qpushbutton.h>
4   #include <qlayout.h>
5   #include <Qtooltip.h>
6   #include <qwhatsthis.h>
7   #include <qimage.h>
8   #include <qpixmap.h>
9   #include "../form1.ui.h"
10  Form1::Form1( QWidget* parent, const char* name, bool modal, WFlags fl )
11      : QDialog( parent, name, modal, fl )
12  {
13      if ( !name )
14          setName( "Form1" );
15
16      pushButton1 = new QPushButton( this, "pushButton1" );
17      pushButton1->setGeometry( QRect( 90, 100, 109, 28 ) );
18      languageChange();
19      resize( QSize(287, 235).expandedTo(minimumSizeHint()) );
20      clearWState( WState_Polished );
21      // signals and slots connections
22      connect( pushButton1, SIGNAL( clicked() ), this, SLOT( pushButton1_clicked()
23   ) );
24  }
25  Form1::~Form1()
26  {
27      // no need to delete child widgets, Qt does it all for us
28  }
```

从该程序中可以看到，通过 Qt 设计器设计完成的主窗口，实际上是自动产生了一个派生自 QDialog 的类 Form1。程序运行时，将根据 Form1 类的定义产生一个主窗口和一个按钮，并将它们展现到屏幕上。

提示：在面向对象的程序设计过程中，一个类包含两个特殊的方法，称为构造函数、析构函数。构造函数的作用是在生成对象时，调用构造函数完成一些初始化的操作。析构函数的作用是在释放对象时，完成一些清理性的操作。在上述例子中，第 10 行即是 Form1 类的构造函数，第 25 行即是类 Form1 的析构函数。

11.4.3　添加事件处理程序

窗口及组件添加完成后，还需要对组件和窗口的事件进行处理。本例中，需要在单击“确定”按钮时弹出提示信息“HelloWorld”。因此，首先选中刚刚添加的按钮，然后在对象的事件处理窗口中选择“signal handlers”选项面板，在其中的 click 事件条上双击，将自动生成 click 的事件处理函数。不过，该函数是空的，需要用户根据需要在其中进行处理。本例中，只是需要弹出一个提示信息窗口。因此，在系统自动生成的事件处理程序中加入代码，如下所示。

```
#include <qmessagebox.h> /*在事件处理程序所在文件的头部*/
Void Form1::pushButton1_clicked()
{
```

```
    QMessageBox::information(NULL, "Information", "HelloWorld");
}
```

在 Qt 的事件处理机制中，使用了名为“槽”和“信号”的一种元素。所谓“槽”，可以通俗地理解为类中的一个函数。与普通函数不同的是，这个函数可以定义为在接收到某个“信号”时，触发该函数的执行。“信号”可以这样理解，当用户单击某个控件（按钮）时，将会产生一个信号。如果该信号被连接到某个槽，则槽所对应的函数将会执行。槽与信号的关系如图 11-9 所示。

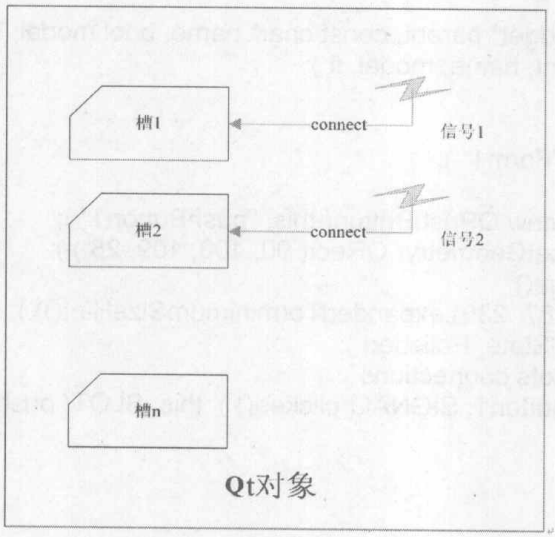


图 11-9　槽与信号的关系

11.4.4　添加主程序

要最终生成可执行的程序，还需要添加一个 main 主程序。在 Qt 设计器中选择菜单栏中的【file】→【new】命令，然后选择“C++Main File”选项，如图 11-10 所示。

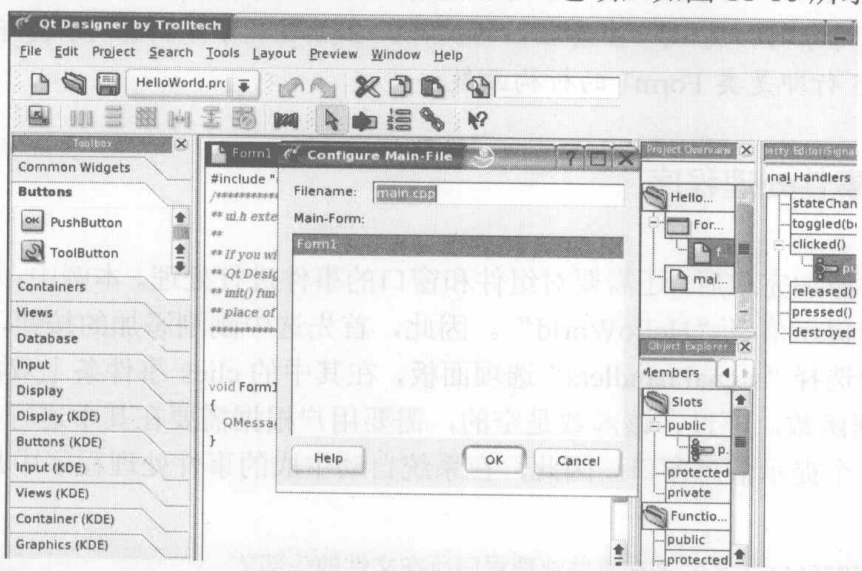


图 11-10　增加主程序

上图所示的窗口中选择当前应用的主窗口。本例中只有一个窗口，所以选择 Form1 为主窗口。单击“OK”按钮，系统将自动生成 main 函数。代码如下所示。

```
1   #include <qapplication.h>                                    /*系统头文件*/
2   #include "form1.h"                                           /*自定义头文件*/
3   int main( int argc, char ** argv )                           /*主函数*/
4   {
5       QApplication a( argc, argv );                            /*初始化应用*/
6       Form1 w;                                                 /*定义 Form 类型变量*/
7       w.show();                                                /*显示窗口*/
8       a.connect( &a, SIGNAL( lastWindowClosed() ), &a, SLOT( quit() ) );  /*安装事件处理*/
9       return a.exec();
10  }
```

在上述程序中，QApplication 是 Qt 程序所必需的全局对象。在 Qt 中，每个程序只能包含一个 QApplication 对象。

- 第 7 行调用 Form 对象的 show 方法，将 Form 对象显示在屏幕上。
- 第 8 行的 connect 调用是将主窗口关闭这一信号与函数 quit 进行了关联。在该函数调用成功后，一旦主窗口关闭这一事件发生，系统将自动调用函数 quit。这一机制将在后续章节中继续加以介绍。
- 第 9 行通过调用 QApplication 对象的 exec 方法，将程序的控制权由用户交给 Qt。Qt 将负责接收和处理用户产生的或者系统产生的事件，并把这些事件传递到相应的窗口。当应用程序关闭时，exec 函数将返回。exec 函数的功能类似于 Visual C++中的 run 函数。

在本例中，通过调用 QApplication 对象的 connect 方法，将主窗口关闭这一信号与槽所对应的函数 quit 进行关联。一旦 connect 调用成功，在主窗口关闭时，将会产生一个信号，该信号将触发函数 quit 的执行。

11.5　Qt 程序的生成

通过前面章节中对 gcc 编译器、makefile 等内容的介绍，读者应该知道要编译一个应用程序，需要编写 makefile 文件，然后通过执行 make 命令进行编译。Qt 程序的生成也有一个类似的过程。Qt 提供了专门的工具用于生成一个项目的 makefile 文件，这个工具就是 qmake。使用该工具的方法如下所示。

```
qmake 项目名称.pro
```

项目名称.pro 是 Qt 设计器生成的项目文件。执行完成上述命令后，即在当前目录下生成一个 makefile 文件。然后调用 make 工具根据该 makefile 进行编译链接，生成最终的可执行的应用程序。在命令提示符下运行该程序的结果如图 11-11 所示。

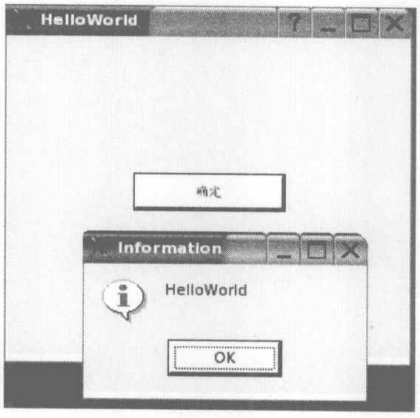


图 11-11　运行结果

11.6 小结

Qt 是 Linux 系统下非常强大的界面开发工具，使用它可以开发出非常专业的界面应用。本章仅向读者介绍了使用 Qt 的基础知识。其中，首先详细介绍 Qt 开发环境的使用方法，然后重点介绍了使用 Qt 进行程序开发的典型步骤，通过这部分的内容，用户可以了解 Qt 程序开发的一般过程。同时，读者可以根据实际应用开发的需要，开发出功能更为强大的桌面应用。

第 12 章　界面程序设计——GTK+

在上一章中介绍了 Qt 界面编程的内容。在 Linux 系统下，还有另外一个非常强大的界面编程接口，那就是 GTK+。本章从 GNOME 与 KDE 的区别讲起，介绍了两种编程模式的异同。重点介绍了用 GTK+进行编程的基本方法。通过本章的学习，读者应该重点掌握如下内容。

- 理解 Linux 系统下图形界面的原理。
- 掌握使用 GTK+进行图形程序设计的基本方法。

12.1　GNOME 与 KDE

GNOME 是 Linux 操作系统下的图形用户界面，基于该图形界面的开发需要使用 GTK+。而 KDE 同样是广泛应用的图形用户界面，不过基于 KDE 的图形应用程序开发是使用 Qt。Qt 开发方法在上一章已进行了简单介绍。本节将首先从 GNOME 与 KDE 的关系讲起，为后续章节 GTK+的开发进行知识准备。

12.1.1　Linux 图形界面原理

目前的 Linux 系统都可以提供华丽的操作界面，其美观性不亚于 Windows。但是 Linux 操作系统的图形界面与 Windows 还是存在根本区别的。Windows 系列操作系统自 Windows 95 之后，对图形的功能是在系统内核中实现的。也就是说，Windows 是一款真正从核心支持图形化界面的操作系统。

Linux 则不然，目前 Linux 系统所提供的图形操作界面只不过是 Linux 的一个程序。读者可以在字符界面输入 startx 启动图形界面。从内核上看，Linux 仍然是一个面向字符界面的操作系统。这一点与 Windows 3.X 时代，通过在 DOS 提示符下输入 win 启动 Windows 非常相似。

由图形界面切换至字符界面，可以通过<Ctrl>+<Alt>+<Backspace>组合键实现，或者通过执行命令 init 3 也可以。而要从字符界面切换回图形界面，则在命令提示符下输入“startx”即可。上面的方法只是临时提供的界面切换方法，如果要一劳永逸地设置系统刚启动时的显示方式，则可以通过修改/etc/inittab/etc/inittab 中的下述行实现。

```
id:3:initdefault:
```

id 后面的就是系统缺省的启动方式，如果是 3 则表示是字符界面，是 5 表示图形界面。

12.1.2　X 协议

在 Linux 系统下，界面的实现是通过 X 协议实现的。就像网络上的通讯协议一样，X 协

议定义了基于图形界面的应用程序与 X 服务器间进行通信的规范。例如，应用程序要在屏幕上画一个圆圈，它就会通过 X 协议通知 X 服务器，X 服务器就会根据需要在屏幕上画一个圆圈，通过与底层的图形硬件设备的接口完成图形的绘制。

12.1.3 GNOME 与 KDE 的启动

GNOME 和 KDE 都是 Linux 系统下的图形操作环境，它们各自按照不同的设计思想实现了图形化的操作界面。当用户在 Linux 的字符环境下输入“startx”时，大多数系统启动的是 KDE 图形界面，如图 12-1 所示。



图 12-1 KDE 启动界面

如果需要从 KDE 切换到 GNOME 图形界面，需要进行以下操作。

（1）按下<Ctrl>+<Alt>+<Backspace> 组合键进入字符界面。在命令提示符中输入“xinit”，显示的界面如图 12-2 所示。

（2）在 xinit 的命令提示符下输入“gnome-session”即可进入 GNOME 界面，如图 12-3 所示。

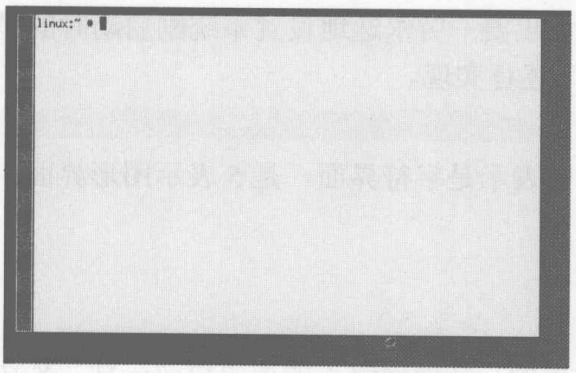


图 12-2 xinit 界面

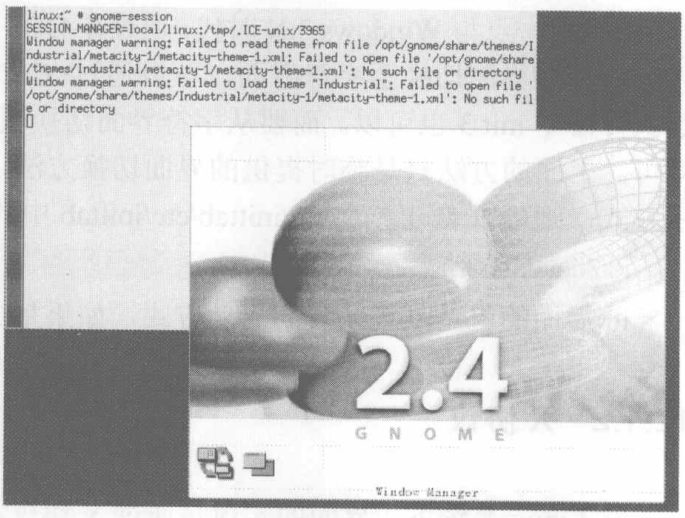


图 12-3 GNOME 界面

12.1.4　GNOME 与 KDE 的区别

GNOME 和 KDE 都是 Linux 系统中常用的图形界面操作环境。KDE 是 K Desktop Environment 的缩写；GNOME 是 GNU Network Object Model Environment 的缩写。KDE 是先于 GNOME 出现的。最初，KDE 被设计成为一个开放源码的图形用户界面和开发工具。KDE 以其强大的功能受到各 Linux 厂商的青睐。在多数的 Linux 系统中，都以 KDE 作为缺省的图形界面。但是，KDE 是基于 Qt 协议的（这里的 Qt 就是在第 11 章中介绍过的 Qt 开发包），而 Qt 协议没有遵循 GPL 协议。

基于上述因素，墨西哥的一批 Linux 专家开始设计一种完全遵循 GPL 协议的图形界面工具。经过一年多的努力，GNOME 诞生了。GNOME 提供了多种图形化的工具，很快在 Linux 世界受到欢迎。目前，多数 Linux 系统都采用 GNOME 作为缺省的图形操作环境。

12.2　使用 GTK+进行开发

GTK+提供了功能强大的图形界面开发接口，开发方法非常复杂。受篇幅所限，本书仅以 HelloWorld 为例讲解初步的应用。如果读者感兴趣，可以参考 GTK+参考手册，开发出功能强大的图形应用程序。

12.2.1　GTK+的安装

GTK 是英文 GIMP Toolkit 的简称，而 GTK+是 GTK 的升级版本。要进行 GTK+的开发，首先要安装 GTK+开发包。通常情况下，安装包都包含在 Linux 的安装介质中，可以通过 yast 等工具进行安装。安装过程不再详述。

如果安装介质中的版本过旧，也可以从网上下载最新版本的 GTK+安装包。网址是 www.gtk.org。在安装的过程中要注意的是，如果要成功安装 GTK+，必须首先安装另外一个开发工具包 glib。glib 同样可以从上述网址下载最新版本。基本的安装过程如下所示。

```
#gzip –d gtk+-.xxx.tar.gz
#tar xvf gtk+-.xxx.tar
#./configure
#./make
#./make install
#/sbin/ldconfig
```

提示：安装过程中的 xxx 代表的是 GTK+的版本号。

12.2.2　GTK+程序的初始化与退出

一个 GTK+程序的生命周期包括初始化、事件循环、结束退出 3 个部分。GTK+程序的入口函数是 main，这与所有的 Linux 程序是一致的。在 main 函数中，首先要做的就是进行 GTK+

的初始化工作。这个工作是通过调用函数 gtk_init 完成的。gtk_init 的函数原型如下所示。

```
gtk_init(int *argc,char **[]argv);
```

其中，argc 和 argv 这两个参数是指向 main 的两个参数的指针。

在初始化 GTK+完成后，就可以进行其他相关 GTK+功能的调用，如创建与显示控件、设置事件处理等。在所有这些功能调用完成后，程序将通过调用 gtk_main 函数，进入程序的主循环，此时，gtk_main 函数将会阻塞。GTK+负责接收各种事件，如鼠标的单击、键盘的输入等，根据事件的不同分别调用不同的事件处理函数。

要退出 GTK+程序，需要调用函数 gtk_main_quit 函数。调用 gtk_main_quit 函数后，被阻塞的 gtk_main 函数将返回，从而终止整个程序的运行。通常情况下，调用 gtk_main_quit 函数可以在窗口的关闭事件处理程序中进行。

12.2.3 GTK+的事件处理

通常情况下，在 Linux 系统下的编程都是面向过程的。而使用 GTK+进行编程则是基于面向对象的。有过 Windows 开发经验的读者一定知道，在 Windows 下是事件（消息）驱动的。当界面上的某个事件发生（按钮被按下）时，该事件由 Windows 系统捕获，并调用应用程序为该事件自定义的事件处理函数，从而完成事件的处理。

在 GTK+环境下，其编程原理与 Windows 系统是非常相似的。不过，Windows 系统的消息循环是通过循环调用 GetMessage 实现的。而 Linux 下这一功能是由 gtk_main 实现的。如果要退出消息循环，则可以使用 gtk_main_quit 完成。这两个函数的原型如下所示。

```
#include <gtk/gtkmain.h>
void gtk_main();
void gtk_main_quit();
```

在 GTK+环境下，事件称为信号。要处理一个事件，首先需要在程序中对该事件的处理方法进行定义。通常情况下的处理方法是定义一个回调函数。当事件发生时，由操作系统调用程序中的事件处理方法完成事件处理。用于定义事件的函数是 gtk_signal_connect，其语法如下所示。

```
gint gtk_signal_connect( GtkObject *object, gchar *name,GtkSignalFunc func, gpointer func_data );
```

参数说明如下。

- Object：输入参数，定义了将要发出信号的组件。
- Name：输入参数，希望捕获的信号名称，例如，按钮控件的 clicked 信号表明该按钮被按下。
- Func：输入参数，定义捕获信号时要调用的函数。
- Func_data：输入参数，定义了要传递给回调函数的用户参数。

回调函数就是指当事件发生时，由系统负责调用的用户定义的函数。注意：这个函数是由操作系统负责调用的，而不是由用户的程序逻辑调用的。回调函数的形式通常如下所示。

```
void callback_func( GtkWidget *widget, gpointer callback_data );
```

- widget：输入参数，指向发出信号的构件的指针。

- callback_data：输入参数，指向传递给回调函数的用户参数的指针。

12.2.4　使用 GTK+实现 HelloWorld

使用 GTK+进行开发的过程与 Qt 开发不同。GTK+通常没有集成的开发环境，编程是通过直接编码的方式进行的。在下面的内容中，将以一个 HelloWorld 的示例程序向读者介绍 GTK+编程的基本模型。

【范例 12-1】使用 GTK+编程实现 HelloWorld。其实现过程如示例代码 12-1 所示。

示例代码 12-1

```
#include <gtk/gtk.h>                                                    /*头文件*/
/*单击按钮的回调函数*/
void hello( GtkWidget *widget, gpointer data )                          /*函数定义*/
{
    g_print("HelloWorld\n");                                           /*输出 HelloWorld*/
}
                                                                        /*关闭窗口信号的回调函数*/
gint delete_event( GtkWidget *widget, GdkEvent *event, gpointer data )  /*回调函数定义*/
{
    /*返回 FALSE 将引发一个 destroy 信号，导致程序退出*/
    g_print ("delete event occurred\n");                                /*输出提示信息*/
    return(TRUE);
}
                                                                        /* 退出程序信号的回调函数*/
void destroy( GtkWidget *widget, gpointer data )
{
    gtk_main_quit() ;                                                   /*调用函数退出 GTK*/
}
int main( int argc, char *argv[] )                                      /*主函数*/
{
    GtkWidget *window;                                                  /*定义 GtkWidget 指针变量*/
    GtkWidget *button;                                                  /*定义 GtkWidget 指针变量*/
    gtk_init(&argc, &argv);
                                                                        /*创建一个主窗口*/
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
                                                                        /*设置关闭窗口的信号处理*/
    gtk_signal_connect (GTK_OBJECT (window), "delete_event", GTK_SIGNAL_FUNC (de
lete_event), NULL);
                                                                        /*设置退出程序的信号处理*/
gtk_signal_connect (GTK_OBJECT (window), "destroy",GTK_SIGNAL_FUNC (destroy), NU
LL);
                                                                        /*设置窗口的边框宽度*/
    gtk_container_set_border_width (GTK_CONTAINER (window), 10);
                                                                        /*创建一个按钮*/
    button = gtk_button_new_with_label("Hello World");
                                                                        /*设置该按钮单击的信号处理*/
    gtk_signal_connect (GTK_OBJECT (button), "clicked",GTK_SIGNAL_FUNC
(hello),NULL);
                                                                        /*设置单击按钮时退出程序*/
    gtk_signal_connect_object (GTK_OBJECT (button), "clicked",GTK_SIGNAL_FUNC(gt
```

```
40  k_widget_destroy),GTK_OBJECT (window));
41                                                          /*将按钮添加到容器窗口中*/
42      gtk_container_add (GTK_CONTAINER (window), button);
43                                                          /*显示新创建的按钮*/
44      gtk_widget_show (button);
45                                                          /*显示窗口*/
46      gtk_widget_show (window);
47      gtk_main ();                                        /*调用主函数*/
48      return(0);
49  }
```

【运行结果】经过编译链接，在图形操作环境下运行上述脚本，其结果如图 12-4 所示。

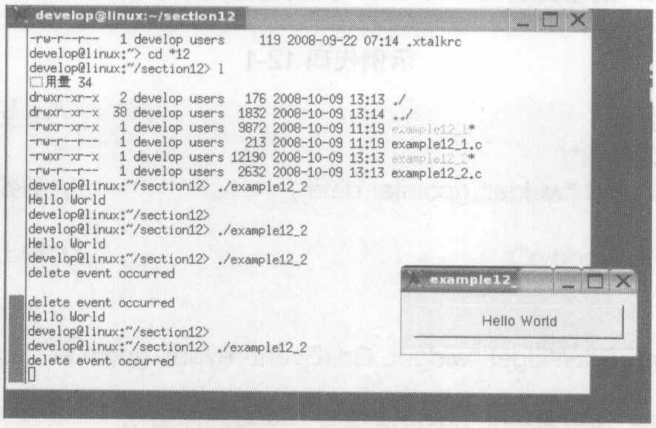


图 12-4　HelloWorld 运行结果

【代码解析】在本例中，源代码的详细解释如下。

- 第 1 行：头文件信息。
- 第 3～6 行：定义按钮单击的信号处理回调函数。可以看到，函数中只是简单输出了一行提示信息。
- 第 8～13 行：定义关闭窗口事件的回调处理函数。在该函数中，如果返回 FALSE，将引发 destroy 信号，导致程序结束退出。本例中只是输出一条信息后返回 TRUE。
- 第 15～18 行：定义结束程序信号的回调函数。在函数中，调用 gtk_main_quit 结束消息循环。
- 第 19～49 行：程序的主函数。
- 第 21～22 行：定义窗口对象变量和按钮对象变量。
- 第 23 行：调用 gtk_init 初始化 GTK 库。该语句在所有的 GTK+程序中都是必须并且是在所有 GTK 函数前被调用的。
- 第 25 行：创建一个新窗口。窗口的属性是顶级窗口，在窗口显示时，将处在所有窗口的最上层。
- 第 27～28 行：设置窗口关闭按钮被单击时的信号处理函数。
- 第 30～31 行：设置窗口退出的信号处理函数。
- 第 33 行：设置窗口的边框的宽度。
- 第 35 行：创建一个新的按钮。按钮的文本设置为“HelloWorld”。
- 第 37 行：设置按钮被单击时的信号处理函数。

- 第 39～40 行：设置当新建的按钮被单击时，退出窗口。
- 第 42 行：将新建的按钮作为子控件添加到容器窗口中。
- 第 44～46 行：显示按钮及窗口。
- 第 47 行：开始消息循环，等待事件的发生。

12.2.5　编译 GTK+程序

编译 GTK+应用程序的方法也比较特别。在使用标准的 gcc 编译器进行编译时，需要附加额外的参数，其语法如下所示。

```
Gcc xxx.c –o xxx 'gtk-config –cflags –libs'
```

经过编译链接成功后的程序，就可以在图形界面环境下运行。在范例 12-1 中，经过编译链接，在命令行下运行程序结果，如图 12-4 所示。

12.2.6　在 GTK+中使用控件

通过上述讲解，读者对 GTK+编程的过程有了基本的了解。但是，要想编写出功能强大的 GTK+程序，还必须对各种 GTK+控件的使用有更深入的了解。本节将向读者介绍 GTK+的控件。

在 GTK+中，所谓的控件实际上是对 X WINDOW 系统提供的控件的更高层的封装。通过封装，使得这些控件具有的 GTK+的特性，也更加便于程序员的使用。在 GTK+中，控件可以分为两类即容器控件和非容器控件。所谓容器控件，就是控件本身可以作为其他控件的容器。也就是说，容器控件中可以包含其他的控件。

1．非容器控件

常见的非容器控件如文本标签控件、图像控件、单行编辑控件等，如下所示。

```
label = gtk_label_new("this is a test label");          /*文本标签控件*/
image = gtk_image_new_from_file("image.png");           /*图像控件，后面参数是指定要显示的图像名*/
entry = gtk_entry_new();                                /*单行编辑控件*/
```

2．容器控件

容器控件都是从 GtkComtainer 派生而来的。根据可以包含控件的数量不同，容器控件又可以划分为只能包含一个控件的容器控件和能包含多个控件的容器控件。只包含一个控件的容器控件是基于 GtkBin 类派生而来。如果需要在这种控件中添加多个控件，则需要首先向控件中加入一个可以包含多个控件的容器控件，再向新的容器控件中添加其他控件。主要的这种容器控件包括以下几种。

- 窗口类控件（GtkWindow、GtkDialog、GtkMessageDialog，主要是窗口和各种对话框）。
- 滚动窗口（GtkScrolledWindow）。
- 框架类控件（GtkFrame、GtkAspectFrame）。
- 视口（GtkViewport）。

- 按钮（GtkButton、GtkToggleButton）。
- 浮动窗口（GtkHandleBox）。
- 事件盒（GtkEventBox）。

能包含多个控件的容器控件是最常用的容器控件，这些控件主要包括以下几种。

- 只能容纳一行或一列子控件的容器：盒状容器（GtkBox），分为成横向盒状容器（GtkHBox）和纵向盒状容器（GtkVBox）。
- 盒状容器的子类：按钮盒（GtkButtonBox），同样也分为横向按钮盒（GtkHButtonBox）和纵向按钮盒（GtkVButtonBox）。
- 能容纳多行多列子控件的容器：格状容器（GtkTable）。
- 能控制子控件的尺寸的容器：分隔面板（GtkPaned），分为横向分隔面板（GtkHPaned）和纵向分隔面板（GtkVPaned）。
- 能分页显示多个子控件的容器：多页显示控件（GtkNotebook）。

12.3 综合实例

在进行图形界面程序设计过程中，菜单是经常要使用的元素之一。本节将介绍使用 GTK+ 实现一个简单的带有菜单功能的窗口程序的过程。

12.3.1 实例需求

创建一个简单的窗口程序，要求窗口包含一个主菜单，主菜单包括“文件”、“编辑”、“视图”和“工具”菜单项。在“文件”菜单项下有 4 个菜单子项，分别为“新建”、“打开”、“保存”、“退出”。

12.3.2 实例代码及解释

根据程序的要求可以看到，本实例要实现的是一个简单的窗口程序。该程序与前面介绍过的 GTK+编程内容基本类似，只是增加了一个菜单的功能。

根据这一要求，先编写 GTK+主程序，将菜单实现部分封装到函数 CreateMenu 中加以实现。主程序实现的基本逻辑是先初始化 GTK+库，然后创建相关窗口和控件（包括菜单项），调用 gtk_main 进入消息循环。最后，在关闭窗口的事件中调用 gtk_main_quit 退出程序的运行。

（1）主程序的代码如下所示。

```
1  #include <gtk/gtk.h>                                          /*头文件*/
2  GtkWidget *CreateMenuItem(GtkWidget *MenuBar,char *test);     /*方法声明*/
3  GtkWidget *CreateMenu(GtkWidget *MenuItem);                   /*方法声明*/
4   int main(int argc,char *argv[])                              /*主函数*/
5  {
6  GtkWidget *window;                                          /*主窗口变量*/
```

```
7    GtkWidget *MenuBar;                                          /*菜单条变量*/
8    GtkWidget *box;                                              /*容器变量*/
9    GtkWidget *MenuItemFile;                                     /*文件菜单项变量*/
10
11   gtk_init(&argc,&argv);                                       /*初始化 gtk+*/
12   window=gtk_window_new(GTK_WINDOW_TOPLEVEL);                  /*创建新窗口*/
13   gtk_widget_set_usize(window,400,200);                        /*设置窗口尺寸*/
14   gtk_signal_connect(GTK_OBJECT(window),"destroy",G_CALLBACK(gtk_main_quit),NULL);
     /*设置事件*/
15
16   box=gtk_vbox_new(FALSE,0);
17   gtk_container_add(GTK_CONTAINER(window),box);                /*向主窗口中添加 box*/
18   MenuBar=gtk_menu_bar_new();                                  /*创建菜单条*/
19   gtk_box_pack_start(GTK_BOX(box),MenuBar,FALSE,TRUE,0);       /*将菜单条关联至 box*/
20
21   MenuItemFile=CreateMenuItem(MenuBar,"文件");                 /*创建菜单项*/
22   CreateMenu(MenuItemFile);                                    /*创建子菜单*/
23   CreateMenuItem(MenuBar,"编辑");                              /*创建菜单项*/
24   CreateMenuItem(MenuBar,"视图");                              /*创建菜单项*/
25   CreateMenuItem(MenuBar,"工具");                              /*创建菜单项*/
26   gtk_widget_show(box);                                        /*显示 box 窗口*/
27   gtk_widget_show(MenuBar);                                    /*显示菜单*/
28   gtk_widget_show(window);                                     /*显示主窗口*/
29   gtk_main();                                                  /*进入循环*/
30   return 0;                                                    /*退出程序*/
31   }
```

（2）设计一个建立菜单项的方法 CreateMenuItem，根据输入的菜单项标题，调用 gtk_menu_shell_append 将菜单项放到菜单条中。其实现代码如下所示。

```
32   GtkWidget *CreateMenuItem(GtkWidget *MenuBar,char *test)     /*创建菜单项的函数*/
     {
33   GtkWidget *MenuItem;                                         /*菜单项对象*/
34   MenuItem=gtk_menu_item_new_with_label(test);                 /*创建菜单项*/
35   gtk_menu_shell_append(GTK_MENU_SHELL(MenuBar),MenuItem);      /*将菜单
     项放入到菜单条*/
36   gtk_widget_show(MenuItem);                                   /*显示菜单项*/
37   return MenuItem;                                             /*返回菜单项对象*/
38   }
```

（3）设计一个建立菜单条的方法 CreateMenu，在相应的菜单项下创建菜单项。本例中是在“文件”菜单项下创建菜单项“新建”、“打开”、“保存”和“退出”。其实现代码如下所示。

```
39   GtkWidget *CreateMenu(GtkWidget *MenuItem)                   /*创建子菜单的函数*/
40   {
41   GtkWidget *Menu;                                             /*子菜单对象*/
42   GtkWidget *Exit;                                             /*退出子菜单的对象*/
43   Menu=gtk_menu_new();                                         /*创建子菜单对象*/
44   CreateMenuItem(Menu,"新建");                                 /*创建子菜单项*/
45   CreateMenuItem(Menu,"打开");
46   CreateMenuItem(Menu,"保存");
47   Exit=CreateMenuItem(Menu,"退出");                            /*创建退出子菜单项*/
```

```
48  g_signal_connect(GTK_OBJECT(Exit),"activate",G_CALLBACK(gtk_main_quit),NULL);/* 菜 单
    事件处理*/
49  gtk_menu_item_set_submenu(GTK_MENU_ITEM(MenuItem),Menu);/*将子菜单放入到父菜单中*/
50  gtk_widget_show(Menu);                                    /*显示菜单*/
51  return Menu;                                              /*返回创建的子菜单*/
52  }
```

12.4 小结

GTK+与 Qt 都可以用于进行图形界面开发。本章重点讲解了使用 GTK+进行图形程序设计的一般过程。一般典型的开发过程中，GTK+程序开发包括安装、初始化、事件处理和编译。前面章节也详细讲解了 Qt 开发的内容，在实际开发过程中，程序员可以根据自己的需要，合理选择适当的开发工具，以开发出界面美观、功能强大的桌面应用。

第13章 进 程

进程是 Linux 操作系统中最重要的概念。要掌握好 Linux 编程，离不开对进程相关概念深入细致的理解。本章从进程的基本概念入手，逐步展开，重点介绍了进程的运行环境、进程的创建及结束等内容。通过本章的学习，读者应重点掌握如下内容。

- 理解进程的基本概念，特别是进程的各个属性的含义。
- 了解进程的运行环境，特别是要加深对命令行参数、环境变量等概念的理解。
- 掌握创建进程的若干方法，能够熟练应用这些方法进行多进程编程。

13.1 进程的基本概念

简单地说，进程是指处于运行状态的程序。一个源程序经过编译、链接后，成为一个可以运行的程序。当该可执行的程序被系统加载到内存空间运行时，就称为进程。程序是静态的保存在磁盘上的代码和数据的组合，而进程是动态的概念。

13.1.1 进程的属性

进程创建后，系统内核为其分配了一系列的数据结构。这些数据结构中保存了进程的相关属性。主要的进程属性包括以下几种。

- 进程的标识符：进程创建时，内核为每个进程分配一个惟一的进程标识符。进程的标识符是一个非负整数，取值范围从 0～32767。进程 ID 是由系统循环使用的。如果当前可用进程号超过了最大值，将从 0 选择可用的整数继续循环使用。
- 进程的父进程标识符：Linux 下的全部进程组成一棵进程树。其中树根进程是 0 号进程 swapper。除根进程外，每个进程都有其对应的父进程。
- 进程的用户标识：是指运行该程序的用户 ID。当一个程序被某个用户执行而变为进程时，该用户就成为进程的用户标识。
- 进程的组标识：是指运行该程序的用户所归属的组 ID。
- 进程的有效用户标识：是指该进程运行过程中有效的用户身份。在进行文件权限许可等检查时，以该有效用户标识为依据。
- 进程的有效组标识：是指当前进程的有效组标识。在进行文件权限许可等检查时，以该有效组标识为依据。进程的用户和组相关的 4 个标识主要用于检查对文件系统的访问权限。
- 进程的进程组标识符：一个进程可以属于某个进程组。通过设置进程组，可以实现向一组进程发送信号等进程控制操作。

- 进程的会话标识符：每个进程都属于惟一的会话。在进程的属性中包含了进程的会话 ID 信息。

进程的这些属性大部分可以通过执行命令 ps 查看得到，如图 13-1 所示。

```
F S UID        PID  PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
4 S root         1     0  1  76   0    -  147 schedu 20:35 ?        00:00:04 init[3]
1 S root         2     1  0  94  19    -    0 ksofti 20:35 ?        00:00:00 [ksoftirqd/0]
1 S root         3     1  0  65 -10    -    0 worker 20:35 ?        00:00:00 [events/0]
1 S root         4     3  0  75 -10    -    0 worker 20:35 ?        00:00:00 [kacpid]
1 S root         5     3  0  65 -10    -    0 worker 20:35 ?        00:00:00 [kblockd/0]
```

图 13-1　ps 查看进程属性

【范例 13-1】通过执行 ps 命令查看到进程的属性信息。其实现过程如示例代码 13-1 所示。

示例代码 13-1

```
1    ps –alef|more                    /*输出系统内全部进程的信息*/
```

【运行结果】在 shell 下运行上述命令，其结果如下所示（有删节）。

```
1    F S UID          PID  PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME
     CMD
2    4 S root        2977  2976  0  76   0    -  945 wait4  09:59 pts/2    00:00:00
     bash
3    4 R root        3008  2977  0  77   0    -  586 -      10:06 pts/2    00:00:00
     ps -alef
4    …
```

【代码解析】在 ps 命令的输出中，包含了进程的属性信息。输出信息的解释如下。

- 第 1 行：ps 命令输出的头部信息，用于标明后续列出的进程属性数据的含义。
- 第 2 行：命令 bash 正处于睡眠状态。这是一个 shell 进程。
- 第 3 行：命令 ps –alef 正处于运行状态。进程的标识符是 3008，父进程标识符是 2977。控制终端是 pts/2。可以看出，该进程是第 2 行输出的进程 bash 的子进程。

Linux 支持通过编程的方式获取进程的属性，包括获取进程 ID 的 getpid，获取进程父进程 ID 的 getppid，获取进程用户标识的 getuid，获取进程有效用户标识的 geteuid 等。这些系统调用的声明位于头文件<unistd.h>中，其原型如下所示。

```
#include <unistd.h>
__pid_t getpid (void);                  //获取当前进程的进程 ID
__pid_t getppid (void);                 //获取当前进程的父进程 ID
__pid_t getpgrp (void);                 //获取当前进程的进程组 ID
__uid_t getuid (void);                  //获取当前进程的实际用户 ID
__uid_t geteuid (void);                 //获取当前进程的有效用户 ID
__gid_t getgid (void);                  //获取当前进程的实际用户组 ID
__gid_t getegid (void);                 //获取当前进程的有效用户组 ID
__pid_t getsid (__pid_t __pid);         //获取指定进程的会话 ID
```

返回值说明如下。

- -1：调用失败，查看 errno 获取详细错误信息。
- 其他：获取到的进程属性信息。

【范例 13-2】通过编程方式获取进程的属性信息。其实现过程如示例代码 13-2 所示。

示例代码 13-2

```
1   #include <stdio.h>                                          /*头文件*/
2   #include <unistd.h>
3   main()                                                      /*主函数*/
4   {
5       printf("process id=%d\n",getpid());                     /*进程 ID*/
6       printf("parent process id=%d\n",getppid());             /*进程的父进程 ID*/
7       printf("process group id=%d\n",getpgrp());              /*进程的组 ID*/
8       printf("calling process's real user id=%d\n",getuid());  /*进程的用户 ID*/
9       printf("calling process's real group id=%d\n",getgid());  /*进程的用户组 ID*/
10      printf("calling process's effective user id=%d\n",geteuid());  /*进程的有效用户 ID*/
11      printf("calling process's effective group id=%d\n",getegid());  /*进程的有效用户组 ID*/
12  }
```

【运行结果】经过编译链接，在 shell 下运行上述程序，其结果如下所示。

```
1   process id=3176
2   parent process id=3097
3   process group id=3176
4   calling process's real user id=1000
5   calling process's real group id=100
6   calling process's effective user id=1000
7   calling process's effective group id=100
```

【代码解析】在程序的输出中，包含了进程的属性信息。从输出可以看到，进程的 ID 与进程的进程组 ID 相同，这表示该进程是这个进程组的首进程。进程的真实用户 ID 与有效用户 ID 也是相同的。本例中，各源程序的解释如下。

- 第 1～2 行：头文件信息。
- 第 5 行：调用 getpid 获取进程的 ID。
- 第 6 行：调用 getppid 获取进程的父进程 ID。
- 第 7 行：调用 getpgrp 获取进程的进程组 ID。
- 第 8 行：调用 getuid 获取进程的真实用户 ID。
- 第 9 行：调用 getgid 获取进程的真实用户组 ID。
- 第 10 行：调用 geteuid 获取进程的有效用户 ID。
- 第 11 行：调用 getegid 获取进程的有效用户组 ID。

提示：通常情况下，进程的用户标识与有效用户标识是相同的，但是对于 suid 程序来说，其有效用户 ID 与用户 ID 是不同的。以常用的/usr/bin/passwd 命令为例，该命令即是一个 suid 程序。运行 passwd 命令的用户，其用户 ID 是运行用户的 ID，而其有效用户 ID 则是超级用户 root。这也是为什么普通用户能够通过运行 passwd 命令修改口令文件/etc/passwd（或者/etc/shadow）文件的原因。

13.1.2　进程的内存映像

一个可执行程序被系统加载后，成为一个进程。在系统内存映像中，进程主要包括代码段、数据段、BSS 段、堆栈段等部分。其内存映像如图 13-2 所示。

其中各元素的含义如下。

- 代码段：代码段是用来存放可执行文件的指令，是可执行程序在内存中的映像。对代码段的访问有严格安全检查机制，以防止在运行时被非法修改。代码段是只读的。
- 数据段：数据段用来存放程序中已初始化全局变量。
- BSS 段：BSS 段包含程序中未初始化的全局变量。
- 堆（heap）：堆是进程空间内的内存区，用于进程运行过程中动态分配内存。堆的大小不固定，可根据程序运行过程中的要求动态变化。当程序中调用 malloc 等函数申请内存时，新申请的内存被动态添加到堆内；当用 free 等函数释放内存时，被释放的内存从堆中被删除。
- 栈（stack）：栈是存放程序中的局部变量的内存区。另外，栈还被用于保存函数调用时的现场。在调用函数时，函数的参数被压入栈，由函数从栈中取出。在函数返回时，将返回值压入栈，由主调函数从栈中取出。栈是由系统自动分配的，用户程序不需要关心其分配及释放。

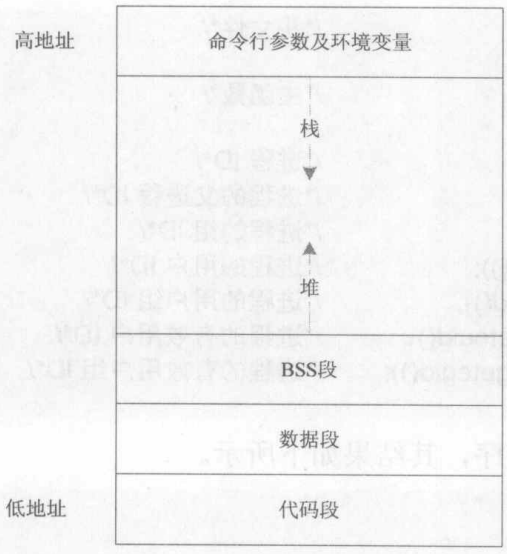


图 13-2　进程的内存映像

13.1.3　进程组

在 Linux 系统中，每个进程都惟一的归属于某个进程组。在 shell 环境中，一条 Linux 命令就形成一个进程组。这条命令可以只包含一个命令，也可以是通过管道符连接起来的若干命令。每个进程组都有一个组长进程。进程组的 ID 就是这个组长的 ID。当进程组内的所有进程都结束或者加入到其他进程组内时，该进程组就结束了。

可以通过系统调用 setpgid 修改某个进程的进程组。该函数位于头文件<unistd.h>中，其原型如下所示。

```
#include <unistd.h>
int setpgid (__pid_t __pid, __pid_t __pgid);
```

（1）参数说明如下。

- __pid：输入参数，用于指定要修改的进程 ID。如果该参数为 0，则指当前进程 ID。
- __pgid：输入参数，用于指定新的进程组 ID。如果该参数为 0，则指当前进程 ID。

（2）返回值说明如下。

- 0：表明调用成功。
- -1：表明调用失败，查看 errno 可以获取详细的错误信息。

【范例 13-3】调用 setpgid 使本进程成为新进程组的组长。其实现过程如示例代码 13-3 所示。

示例代码 13-3

```
1    #include <stdio.h>                          /*头文件*/
2    #include <unistd.h>
3    main()                                      /*主函数*/
4    {
5        setpgid(0,0);                           /*设置当前进程为新的进程组的组长*/
6        sleep(10);                              /*休眠 10 秒，以供查看进程状态*/
7    }
```

【运行结果】经过编译链接，在 shell 下运行上述程序。在另外的 shell 中通过执行 ps -ao pid,pgrp,cmd 命令查看进程的进程组 ID，其结果如下所示。

```
1      PID  PGRP   CMD
2     3518  3518   ./example13_2
3     3519  3519   ps -ao pid,pgrp,cmd
```

【代码解析】从输出可以看到，进程./example13_2 的进程 ID（PID=3518）与进程的进程组 ID（PGRP=3518）相同，这表示该进程是这个进程组的首进程。本例中，各源程序的解释如下。

- 第 1～2 行：头文件信息。
- 第 5 行：调用 setpgid 修改进程的进程组。该语句等价于 setpgrp()。
- 第 6 行：休眠 10 秒钟，以便于通过另外的 shell 查看进程信息。

提示：在 Linux 下还有另外一个函数 setpgrp 用于设置当前进程为进程组的组长。调用该函数后，将产生一个新的进程组，进程组的组 ID 为调用进程的 ID。也就是说，调用 setpgrp 将创建一个以调用进程为组长的新的进程组。该函数的功能可以用 setpgid(0,0)替代实现。

13.1.4　进程的会话

当用户登录一个新的 shell 环境时，一个新的会话就产生了。一个会话可以包括若干个进程组，但是这些进程组中只能有一个前台进程组，其他的为后台运行进程组。前台进程组通过其组长进程与控制终端相连接，接收来自控制终端的输入及信号。一个会话由会话 ID 来标识，会话 ID 是会话首进程的进程 ID。会话与进程组及进程的关系如图 13-3 所示。

Linux 提供了系统调用 setsid 用于产生一个新的会话。不过，调用 setsid 的进程应该保证不是某个进程组的组长进程。setsid 调用成功后，将生成一个新的会话。新会话的会话 ID 是调用进程的进程 ID。新会话中只包含一个进程组，该进程组内只包含一个进程：即调用 setsid 的进程，且该会话没有控制终端。setsid 系统调用的声明位于头文件<unistd.h>中，其原型如下所示。

```
#include <unistd.h>
__pid_t   setsid(void);
```

返回值说明如下。

- -1：调用 setsid 失败，查看 errno 可以获取详细的错误信息。典型的错误是调用进程是某个进程组的组长，此时 setsid 将失败，错误码 errno 为 EPERM。
- 其他值：返回进程的进程组 ID。

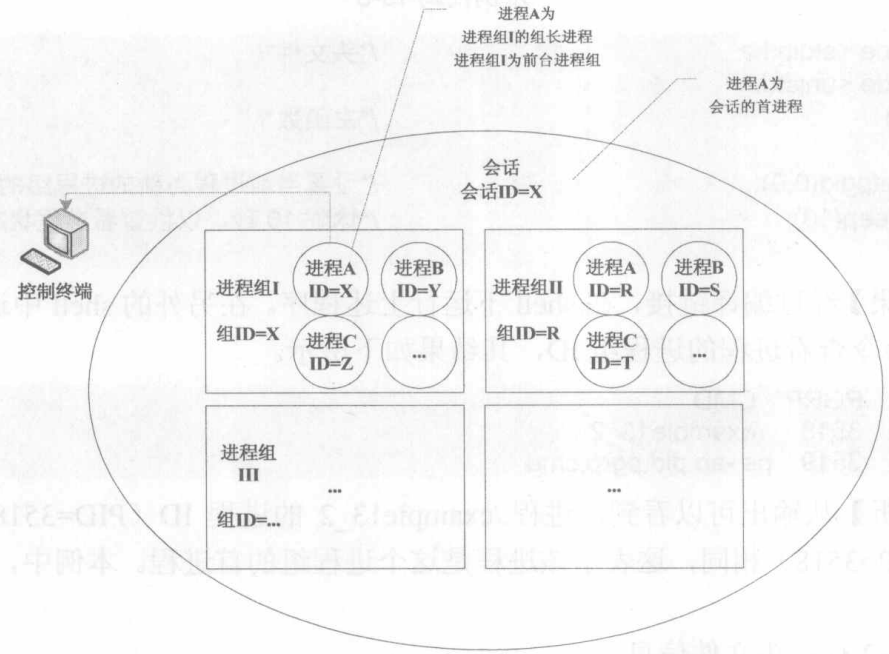


图 13-3　进程组与会话

【范例 13-4】调用 setsid 实现进程的后台运行。其实现过程如示例代码 13-4 所示。

示例代码 13-4

```
#include <stdio.h>                          /*头文件*/
#include <unistd.h>
main()                                      /*主函数*/
{
    int n;                                  /*循环变量定义*/
    __pid_t nPid;                           /*进程 ID 变量*/
    __pid_t nGroupId;                       /*进程的组长进程 ID*/
    if((nPid = fork()) < 0)                 /*创建新的子进程*/
    {
        perror("fork");                     /*创建子进程失败，错误处理*/
        exit(0);
    }
    if(nPid != 0)                           /*父进程*/
        exit(0);                            /*父进程退出*/
    nGroupId = setsid();                    /*产生新会话，返回新创建的进程组的组 ID*/
    if(nGroupId == -1)                      /*错误处理*/
    {
        perror("setsid");                   /*输出错误信息*/
        exit(0);
    }
    for(n=0;n<10;n++)                       /*循环休眠一段时间退出，供用户查看运行结果*/
        sleep(3);                           /*休眠 3 秒*/
}
```

【运行结果】经过编译链接，在 shell 下运行上述程序。在另外的 shell 中通过执行 ps -ao pid,ppid,pgrp,session,tty,cmd 命令查看进程的信息，其结果如下所示。

```
1    3087     1  3087  3087  ?    ./example13_4
```

【代码解析】在本例的输出中，分别显示的是进程 ID、进程的父进程 ID、进程的进程组 ID、进程的会话 ID、进程的控制终端、进程的命令行。从输出可以看到：进程./example13_4 的进程 ID（PID=3087）与进程的进程组 ID（PGRP=3087）相同，这表示该进程是这个进程组的首进程；进程./example13_4 的进程 ID 与其会话 ID（SESSION=3087）相同，这表明进程 example13_4 是当前会话的首进程；进程./example13_4 无控制终端（TTY=?）。本例中，各源程序的解释如下。

- 第 1～2 行：头文件信息。
- 第 8～12 行：调用 fork 创建新的子进程。fork 返回值如果小于 0，表明创建子进程出错；fork 返回值如果不等于 0，表明是在父进程中，返回值是新生成的子进程的进程 ID；fork 返回值如果等于 0，表明当前是在子进程中。
- 第 13～14 行：父进程退出。一个程序执行时，将新生成一个进程组。在本句执行前，主进程是当前进程组的组长。创建子进程后，该进程组有两个进程：主进程和刚刚创建的子进程。此时，父进程退出后，进程组仍然存在（由于子进程还在）。通过调用本语句，确保子进程不是进程组的组长进程（调用 setsid 要求调用进程不能为进程组的组长进程）。
- 第 15～20 行：调用 setsid 生成新的会话。调用后，当前进程成为新会话的首进程。
- 第 21～22 行：循环调用 sleep，以方便通过另外的 shell 查看进程信息。

提示：Linux 系统的进程大体可以分为前台进程和后台进程。所谓前台进程就是运行过程中与控制终端相连接的进程，随时可以通过控制终端与前台进程进行交互，如用户登录时的 shell 进程就是前台进程。后台进程一般无控制终端，如 Linux 系统下各种守护进程就属于后台进程。

13.1.5　进程的控制终端

作为多用户、多任务的操作系统，Linux 支持多个用户同时从终端登录系统。Linux 终端类似于 Windows 环境下的远程桌面连接。用户通过终端输入请求，提交给主机运行并显示主机的运行结果。传统的 Linux 终端是由 RS232 串口通信协议的串口终端，终端与主机间的通讯通过主机的串口进行。这种串口终端数据传输速度较慢，并且传输距离有限，现在已逐渐为网络终端所代替。网络终端与主机间通过以太网相连接，数据传输速度大为提高。Linux 系统中多用户环境下终端的使用如图 13-4 所示。

Linux 系统中，每个终端设备都有一个设备文件与其相关联，这些终端设备称为 tty。在 shell 环境下，可以通过执行命令 tty 查看当前终端的名称。用户可以通过 telnet 远程登录到某个 Linux 系统，此时其实并没有真正的终端设备。这种情况下，Linux 系统将为用户自动分配一个称为“伪终端”的终端设备。伪终端的设备文件名称类似/dev/pts/???。

前面介绍的是 Linux 系统的终端环境。而在 Linux 的进程环境中，有一个称为“控制终端”的概念。所谓控制终端，就是指一个进程运行时，进程与用户进行交互的界面。一个进程从终端启动后，这个进程的运行过程就与控制终端密切相关。可以通过控制终端输入/输出，

也可以通过控制终端向进程发送信号（可以按<Ctrl>+<C>键中止程序运行）。当控制终端被关闭时，该控制终端所关联的进程将收到 SIGHUP 信号。系统对该信号的缺省处理方式就是中止进程。

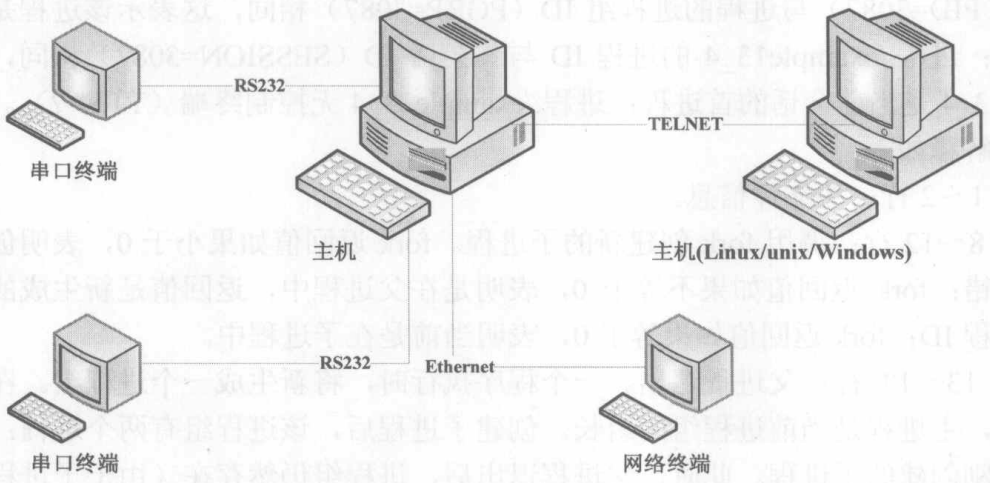


图 13-4　Linux 终端环境

提示：用户可以通过执行 shell 命令 ps –ax 查看进程的控制终端，如示例代码 13-1 所示。在 ps 的输出中，有一列名称为“TTY”的就是控制终端。如果该列中有值，表明进程是有控制终端的，否则表明进程没有控制终端。

13.1.6　进程的状态

Linux 的进程是由操作系统内核调度运行的。在调度过程中，进程的状态是不断发生变化的。这些状态主要包括可运行状态、等待状态（也称为睡眠状态）、暂停状态、僵尸状态、退出状态等，如图 13-5 所示。

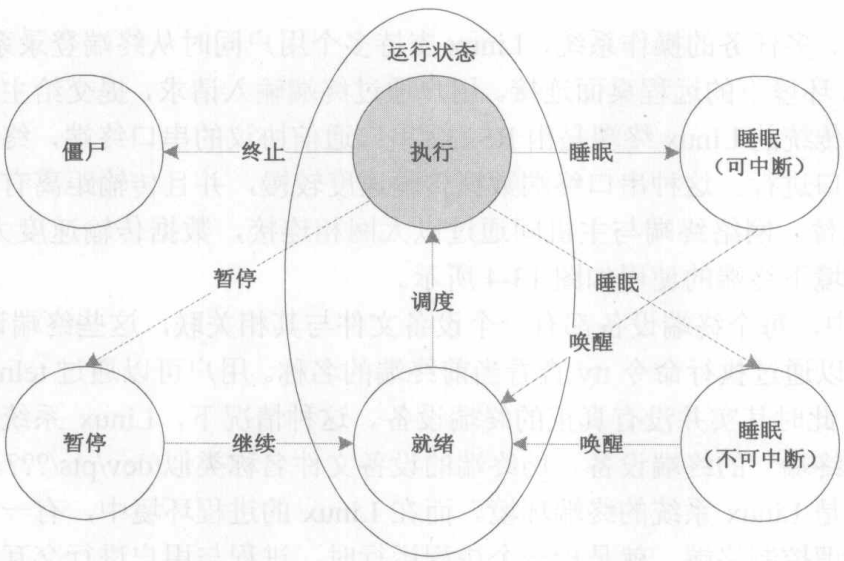


图 13-5　进程状态变化图

可运行状态又包括就绪状态和执行状态，只有处于执行状态的进程是真正占用 CPU 的进程。这些进程状态详细描述如下。

- 可运行状态（RUNNING）：该状态有两种情况，一是进程正在运行；二是处于就绪状态，只要得到 CPU 就可以立即投入运行。在第二种情况中，进程处于预备运行状态，在等待系统按照时间片轮转规则将 CPU 分配给它。
- 等待状态（SLEEPING）：表明进程正在等待某个事件发生或者等待某种资源。该状态可以分成两类：可中断的和不可中断的。处于可中断等待状态的进程，既可以被信号中断，也可以由于资源就续而被唤醒进入运行状态。而不可中断等待状态的进程在任何情况下都不可中断，只有在等待的资源准备好后方可被唤醒。
- 暂停状态（STOPPED）：进程接收到某个信号，暂时停止运行。大多数进程是由于处于调试中，才会出现该状态。
- 僵尸状态（ZOMBIE）：表示进程结束但尚未消亡的一种状态。一个进程结束运行退出时，就处于僵尸状态。进程会在退出前向其父进程发送 SIGCLD 信号（关于该信号的处理见第 14 章内容）。父进程应该调用 wait 为子进程的退出做最后的收尾工作。如果父进程未进行该工作，则子进程虽然已退出，但通过执行 ps 命令仍然可以看到该进程，其状态就是僵尸状态。在应用编程中，应尽量避免僵尸进程的出现。

在 Linux 系统下，除 ps 命令可以查看进程状态外，还有另外一个重要的进程查看工具 top。ps 命令输出的是静态的，是进程的某一时刻的信息，而 top 可以持续的动态的输出进程的信息。

【范例 13-5】通过 top 输出进程的信息。其实现过程如示例代码 13-5 所示。

示例代码 13-5

```
1    top
```

【运行结果】在 shell 下运行上述命令，其输出信息如下所示。

```
1    top - 11:02:52 up   9:15,   4 users,   load average: 0.01, 0.03, 0.00
2    Tasks:   55 total,    2 running,   53 sleeping,    0 stopped,    0 zombie
3    Cpu(s):   1.1% us,   2.3% sy,   0.0% ni, 91.7% id,   1.1% wa,   0.0% hi,   3.8% si
4    Mem:     256624k total,     163720k used,      92904k free,      43828k buffers
5    Swap:    128480k total,          0k used,     128480k free,      71432k cached
6      PID USER      PR  NI  VIRT  RES  SHR  S  %CPU %MEM    TIME+  COMMAND
7     2784 root      15   0 38000  17m  22m   S   2.0   6.9    6:59.71  X
8     2871 root      15   0  3512  1656  3276  S  1.0    0.6    3:42.27  xeyes
9     4031 root      16   0  1944  1020  1716  R  0.7    0.4    0:00.11  top
10       1 root      16   0   588   244   444   S  0.0    0.1    0:05.77  init
```

【代码解析】top 命令的输出是动态的，每间隔一定时间（可以通过-d 参数指定刷新间隔时间），该信息就会刷新一次。在本例的输出中，输出信息的详细解释如下。

- 第 1 行：当前时间（11:02:52）、系统启动时间（9:15）、当前系统登录用户数目（4 users）、平均负载（load average: 0.01, 0.03, 0.00）
- 第 2 行：当前进程的汇总信息，分别为当前进程总数（55 total）、休眠进程数（53 sleeping）、运行进程数（2 running）、僵死进程数（0 zombie）、终止进程数（0 stopped）。
- 第 3 行：当前 CPU 的消息情况，分别是用户占用（1.1%）、系统占用（2.3%）、优先进程占用（0.0%）、闲置进程占用（91.7%）。

- 第 4 行：内存状态汇总信息，分别为平均可用内存（256624k）、已用内存（163720k）、空闲内存（92904k）、缓存使用内存（43828k）。
- 第 5 行：交换区状态，分别为平均可用交换容量（128480k）、已用容量（0k）、闲置容量（128480k）、高速缓存容量（71432k）。
- 第 6 行：进程明细信息的标题，分别是进程 ID（PID）、进程的所有者用户（USER）、进程的优先级别（PR）、进程的谦让度值（NI）、进程占用的虚拟内存值（VIRT）、进程占用的物理内存值（RES）、进程使用的共享内存值（SHM）、进程的状态（S）、进程占用的 CPU 使用率（%CPU）、进程占用的物理内存的百分比（%MEM）、进程启动后占用的总的 CPU 时间（TIME+）、进程启动的启动命令行（COMMAND）。
- 第 7～10 行：系统中各进程的详细信息。该信息每隔一定时间被刷新一次。

13.1.7 进程的优先级

作为一个多任务的操作系统，Linux 操作系统中允许多个进程同时运行。但是，在 CPU 数量少于同时运行的进程数量时，是不可能实现真正同时运行的。以单 CPU 的计算机为例，如果系统中同时存在两个进程同时运行，则实际上在同一时刻只可能有一个进程占用 CPU 运行。

为实现多任务的目标，Linux 使用了一种称为“时间片轮转”的进程调度方式，为每个进程指派一定的运行时间。这个时间片通常很短，以毫秒甚至更小的时间级别为单位。系统核心依照某种规则，从大量进程中选择一个进程投入运行，其余的进程暂时等待。当正在运行的那个进程时间片用完，或进程执行完毕退出，Linux 就会重新进行进程调度，挑选下一个可用进程投入运行。由于每一个进程运行时占用的时间片很短，在用户的角度来看，就如同多个进程同时运行一样。

进程的优先级定义了进程被调度的优先顺序。优先级的数值越低，进程越是优先被调度。优先级的是由进程的优先级别（PR）和进程的谦让值（NI）两个因素联合确定的。Linux 系统内核在调度进程时，将优先级别（PR 值）和谦让值（NI）相加以确定进程的真正优先级别。对于一个进程来说，其优先级别是由父进程继承而来的，用户进程不可更改。

为方便用户修改进程运行的优先级，Linux 提供了 nice 系统调用以修改进程的谦让值。进程的谦让值在进程被创建时置为缺省值为 0。系统允许的谦让值范围为最高优先级的-20 到最低优先级的 19。通过 nice 系统调用可以修改该值。

1．nice

nice 的声明位于头文件<unistd.h>中，其原型如下所示。

```
#include <unistd.h>
int nice (int __inc);
```

（1）参数说明如下。

- __inc：输入参数，指定新的谦让值。该参数取值范围为-20～19。

（2）返回值说明如下。

- 0：调用成功。
- -1：调用失败，可以查看 errno 获取错误信息。

（3）关于使用 nice 系统调用需要注意以下几点。

只有超级用户可以在调用 nice 时指定负的谦让值。也就是说只有超级用户才可以提高进程的调度优先级别。如果不是超级用户而指定负的谦让值，则 nice 调用返回失败，errno 为 EPERM。

2．setpriority

nice 系统调用只能修改进程自身的谦让值，而另外一个系统调用 setpriority 则可以修改其他进程甚至一个进程组的谦让值。该系统调用的声明位于头文件<sys/resource.h>中，其原型如下所示。

```
#include <sys/resource.h>
int setpriority (__priority_which_t __which, id_t __who, int __prio);
```

（1）参数说明如下。

- __which：输入参数，指定设置谦让值的目标类型。setpriority 可以对 3 种目标进行谦让值设置，分别是 PRIO_PROCESS 、PRIO_PGRP 和 PRIO_USER 。PRIO_PROCESS 是为某进程设置谦让值；PRIO_PGRP 是为某进程组设置谦让值；PRIO_USER 是为某个用户的所有进程设置谦让值。
- __who：输入参数，设置谦让值的目标。对于 PRIO_PROCESS 类型的目标，该参数为进程 ID；对于 PRIO_PGRP 类型的目标，该参数为进程组 ID；对于 PRIO_USER 类型的目标，该参数为用户 ID。如果该参数为 0，对于 3 种目标类型，分别表示当前进程、当前进程组、当前用户。
- __prio：输入参数，要设置的谦让值，输入范围为-20～19。只有超级用户可以用负值调用 setpriority。

（2）返回值说明如下。

- 0：调用成功。
- -1：调用失败，可以查看 errno 获取错误信息。

进程的谦让值被修改后，如何得到进程的谦让值呢？Linux 提供了系统调用 getpriority 实现这一目的。getpriority 是与 setpriority 相对应的系统调用，其声明位于头文件<sys/resource.h>中，其原型如下所示。

```
#include <sys/resource.h>
int getpriority (__priority_which_t __which, id_t __who);
```

返回值 getpriority 系统调用比较特殊，可能返回负值，所以无法直接根据返回值确定是否调用成功。建议调用 getpriority 前置 errno 为 0，如果调用后 errno 为 0，表明成功，否则表明调用失败。

【范例 13-6】修改进程的谦让值，调用完成后输出进程的谦让值。其实现过程如示例代码 13-6 所示。

示例代码 13-6

```
1    #include <unistd.h>                                    /*头文件*/
2    #include <errno.h>
3    #include <sys/resource.h>
4    main()                                                 /*主函数*/
```

```
5   {
6       int nPr;                                          /*整型变量定义*/
7       if(nice(3) == -1)                                 /*进程的谦让值为 3，进程的优先级降低*/
8       {
9           perror("nice");                               /*错误处理*/
10          exit(0);
11      }
12      errno = 0;                                        /*设置全局错误变量为 0*/
13      nPr = getpriority(PRIO_PROCESS,getpid());         /*获得当前进程的谦让值*/
14      if(errno != 0)                                    /*错误处理*/
15      {
16          perror("getpriority");                        /*输出错误信息*/
17          exit(0);
18      }
19      printf("priority is %d\n",nPr);                   /*输出进程的谦让值*/
20  }
```

【运行结果】在 shell 下运行上述命令，其输出信息如下所示。

```
1   priority is 3
```

【代码解析】在本例中，首先调用 nice 降低当前进程的谦让值，然后调用 getpriority 得到进程的谦让值并输出。本例中，源代码的各行解释如下。

- 第 1～3 行：头文件信息。
- 第 7～11 行：调用 nice 修改进程的谦让值。此处以 3 为参数调用 nice，实际上是降低了进程的优先级。
- 第 12 行：设置系统变量 errno 为 0。此处是由于 getpriority 返回值的特殊性而添加的，详细说明见 getpriority 的返回值说明。
- 第 13～19 行：调用 getpriority 获取当前进程的谦让值并输出。

提示：除 nice 系统调用外，也可以通过执行 nice 和 rnice 命令来修改一个进程的谦让值。nice 可以在执行一个程序时，直接指定谦让值，而 rnice 命令则可以修改一个正在运行的进程的谦让值。

13.2 进程的运行环境

程序被加载到系统内存而成为一个进程是一个复杂的过程。Linux 系统为进程的运行提供了强大的进程运行环境。本节将主要介绍进程的入口函数及进程的环境变量等内容。

13.2.1 进程的入口函数

1．main 函数

众所周知，C 语言的入口函数是 main，在 Linux 下也是如此。进程开始执行时，都是从 main 函数开始的。所以，在一个可执行的 Linux 程序中，必须有包含 main 函数。main 函数的原型定义如下所示。

```
int main(int argc,char *argv[],char *env[]);
```

（1）参数说明如下。

- argc：表明程序执行时的命令行参数个数。注意：该参数个数包含了程序名称本身。也就是说，如果执行程序时，未在命令行输入任何参数，则该 argc 值为 0。该参数的值是由系统确定的，用户程序可以根据该值获得程序执行时命令行参数的个数。
- argv：命令行参数数组。其中每一个数组成员为一个命令行参数。程序名称是该数组的第一个成员 argv[0]。在 Linux 系统中，各命令行参数是以空格分隔的。
- env：环境变量数组，可以在程序中访问这些环境变量。关于环境变量的详细说明参见下一节。

（2）返回值说明如下。

main 函数的返回值可以在 shell 中获取到。

（3）关于 main 函数，需要注意以下几点。

- main 函数可以有多种格式。在上述介绍中，是格式全面的 main 函数。通常，如果不需要对命令行参数进行处理，可以直接使用 main()这种简单的方式。
- main 函数的 argc 所包含的命令行参数个数是包含程序名称自身的。命令行参数 argv[0]就是程序名称。

【范例 13-7】演示 main 函数。其实现过程如示例代码 13-7 所示。

示例代码 13-7

```
#include <stdio.h>                              /*头文件*/
int main(int argc,char *argv[],char *env[])     /*主函数*/
{
    int i;                                      /*循环变量*/
    for(i=0; i<argc; i++)                       /*循环输出全部命令行参数*/
        printf("argv[%d]=%s\n",i,argv[i]);      /*输出命令行参数*/
    for(i=0; env[i]!=NULL; i++)                 /*循环输出全部环境变量*/
        printf("env[%d]:%s\n",i,env[i]);        /*输出环境变量*/
    return 5;                                   /*返回 5*/
}
```

【运行结果】经过编译链接，在 shell 下执行程序：./example13_7 aa bb cc，其输出信息如下所示。

```
argv[0]=./example13_7
argv[1]=aa
argv[2]=bb
argv[3]=cc
env[0]:LESSKEY=/etc/lesskey.bin
env[1]:REMOTEHOST=192.168.1.5
env[2]:NNTPSERVER=news
env[3]:INFODIR=/usr/local/info:/usr/share/info:/usr/info
env[4]:MANPATH=/usr/local/man:/usr/share/man:/usr/X11R6/man:/opt/gnome/share/man
env[5]:HOSTNAME=Hubery
```

程序执行完毕后，通过 shell 执行命令 echo $?查看程序的返回值，如下所示。

```
5
```

【代码解析】在本例中，分别输出命令行参数及环境变量信息，最后程序返回，返回值为 5。本例中，源代码的各行解释如下。

- 第 1 行：头文件信息。
- 第 5～6 行：输出命令行参数信息。从输出信息的第一行可以看到，命令行参数数组的第一个值就是程序名称。
- 第 7～8 行：输出环境变量信息。环境变量也可以通过 shell 执行命令 env 获取。
- 第 9 行：程序返回，返回值为 5。

本例中，各输出信息的解释如下。

- 第 1～4 行：命令行参数信息。
- 第 5～10 行：环境变量信息。
- 第 11 行：程序的返回值。

提示：Linux 环境下，有多个 shell 系统变量，本例中用到的$?就是其中之一。主要的系统变量如下。

$#：命令行参数个数；$n：命令行参数，n 为非负整数，如$0 表示程序名称，$1 表示第一个命令行参数；$?：前一条命令的返回码；$$：本进程的进程 ID；$!：上一进程的进程 ID。

2．getopt

从上面的介绍可以知道，在编程过程中可以通过直接访问命令行参数数组获取命令行参数。另外，Linux 还提供了专门的系统调用 getopt 获取命令行参数。getopt 提供更为强大的获取命令行参数的方法。它不仅可以获取命令行参数，而且可以按照规则解析命令行参数。如执行下述程序：./test_program –i –s abc –t，在这种情况下，要获取到-s 选项的参数 abc 需要进行很复杂的判断。而通过 getopt 可以很简单地解决这一问题。

getopt 的声明位于头文件<getopt.h>中，其原型如下所示。

```
#include <getopt.h>
int getopt (int ___argc, char *const *___argv, const char *__shortopts);
int getopt_long (int ___argc, char *const *___argv,const char *__shortopts,const struct option
*__longopts, int *__longind);
```

参数说明如下。

- ___argc：输入参数，即 main 函数的 argc。
- ___argv：输入参数，即 main 函数的 argv。
- __shortopts：输入参数，选项字符串。该参数指定了解析命令行参数的规则。getopt 认可的命令行选项参数是通过“-”进行的。例如，ls –l 中的“-l”。该参数中，如果某个选项有输入数据，则在该选项字符的后面应该包含“:”，如 ls –l ./a，在指定本参数时，应该用“l:”。对于该选项，有一点需要注意，如果该参数的第一个字符是“:”，则 getopt 发现无效参数后并不返回失败，而是返回“?”或者“:”。返回“？”表示选项无效，返回“:”表示需要输入选项值。

getopt 只能支持单字符的选项，如-l、-a 等。如果需要支持多字符的选项，如-file 等，就需要用到 getopt_long。getopt_long 通过指定一个 struct option 类型的结构数组，将多字符的选项映射为单个字符，从而实现了对多字符选项的支持。struct option 的结构为如下所示。

```
struct option
```

```
{
  const char *name;
  int has_arg;
  int *flag;
  int val;
};
```

（1）在该结构中，各结构成员的说明如下。

- name：定义了多字符的选项名称。
- has_arg：定义了是否有选项值，如果该值为 0，表示没有选项值；如果该值为 1，表明该选项有选项值；如果该值为 2，表示该选项的值是可有可无的。
- flag：如果该成员定义为 NULL，那么调用 getopt_long 的返回值为该结构 val 字段值；如果该成员不为 NULL，getopt_long 调用后，将在该参数所指向的变量中填入 val 值，并且 getopt_long 返回 0。通常该成员定义为 NULL 即可。
- val 是该长选项对应的短选项名称。

（2）与 getopt_long 相关的其他参数说明如下。

- __longind：输出参数，如果该参数没有设置为 NULL，那么它是一个指向整型变量的指针。在 getopt_long 运行时，该整型变量会被赋为获取到的选项在结构数组 __longopts 中的索引值。

（3）返回值说明如下。

- “?”：表明 getopt 返回一个未在__shortopts 定义的选项。
- “:”：表明该选项需要选项值，则实际未输入选项值。
- -1：表明 getopt 解析完毕，后面已经没有选项。
- 0：在 getopt_long 的结构数组参数__longopts 中的成员 flag 定义了值。此时，getopt_long 返回 0，而选项的参数将存储在 flag 所指向的变量中。
- 其他：返回的选项字符。

（4）在使用 getopt 时，需要注意与该系统调用相关的全局变量的使用，详细说明如下。

- optind：整型变量，存放环境变量数组 argv 的当前索引值。当调用 getopt 循环取选项结束（getopt 返回-1）后，剩余的参数在 argv[optind]～argv[argc-1]中。
- optarg：字符串指针，当处理一个带有选项值的参数时，全局变量 optarg 将存放该选项的值。
- optopt：整型变量，当调用 getopt 发现无效的选项（getopt 返回?或者:）时，此时 optopt 包含了当前无效的选项。
- opterr：整型变量，如果调用 getopt 前设置该变量为 0，则 getopt 在发现错误时不输出任何信息。

【范例 13-8】利用 getopt_long 编程实现可以接受如下选项的程序。具体的程序信息如表 13-1 所示。

表 13-1　接收选项的程序

短选项	长选项	说明
-f	--flag	输入标志
-n	username	--name

其实现过程如示例代码 13-8 所示。

示例代码 13-8

```
#include <stdio.h>                                          /*头文件*/
#include <getopt.h>                                         /*getopt 系列函数要包含本头文件*/
int save_flag_arg;                                          /*全局整型变量定义*/
char *opt_arg_value;                                        /*全局字符串变量定义*/
struct option longopts[] = {                                /*结构数组，用于定义每个参数的细节*/
    { "flag", no_argument, &save_flag_arg,'f'},             /*选项无选项值*/
    { "name", required_argument, NULL,'n'},                 /*选项需要选项值*/
    {     NULL,     0,     NULL,     0},                    /*结构数组结束*/
};
int main(int argc, char *argv[])                            /*主函数*/
{
    int i,c;                                                /*整型变量定义*/
    while((c = getopt_long(argc, argv, ":n:f", longopts, NULL)) != -1) /*循环解析命令行参数*/
    {
        switch (c)                                          /*调用 switch 判断输入的选项*/
        {
        case 'n':                                           /*选项 n*/
            opt_arg_value = optarg;                         /*获得选项值*/
            printf("name is %s.\n", opt_arg_value); /*输出用户名称*/
            break;
        case 0:
            if(save_flag_arg == 'f')                        /*结构成员中定义了 flag，输入值保存在该
变量中，而 getopt 返回 0*/
            {
                printf("flag argument found!\n");    /*输出错误信息*/
            }
            break;
        case ':':                                           /*选项需要输入值，而实际未输入*/
            printf("argument %c need value.\n",optopt);  /*输出提示信息*/
            break;
        case '?':                                           /*无效的选项*/
            printf("Invalid argument %c!\n",optopt); /*输出错误信息*/
            break;
        }
    }
    return 0;
}
```

【运行结果】经过编译链接，在 shell 下执行程序：./example13_8 –name Hubery -flag。其输出信息如下所示。

```
name is Hubery.
flag argument found!
```

【代码解析】在本例中，循环调用 getopt_long 获取命令行选项。本例中，源代码的各行解释如下。

- 第 1 行：头文件信息。
- 第 3 行：全局整型变量声明。该整型变量用于存储调用 getopt_long 时返回的短选项值。
- 第 4 行：全局字符串指针声明。该指针用于保存系统全局变量中存储的当前选项的值。

- 第 5～9 行：定义结构数组，该结构数组定义了多字符选项与单字符选项的对应关系。可以看到，选项“name”需要输入参数，而选项“flag”不需要输入参数。注意：在此处如果指定了需要输入参数，则应在 getopt_long 中的__shortopts 参数中的短选项后输入“:”。
- 第 13～34 行：循环调用 getopt_long 解析输入选项。
- 第 17～24 行：输入选项为 n 或者 name 的处理。此时，optarg 中保存是的输入选项的值。
- 第 21～26 行：getopt_long 返回值为 0，表明结构数组中有成员指定了 flag。此时，flag 指针中存储的值是单字节选项。特别注意这一点，此时 getopt_long 返回值并不是选项。
- 第 27～29 行：getopt_long 返回值为“:”，表明该选项需要选项值而未输入。
- 第 30~32 行：getopt_long 返回值为“？”，表明该选项无效。

提示：使用 getopt_long 函数时，如果使用多字符格式的选项输入，则应该用“--”作为选项的前导符。如果使用单字符格式的选项输入，则可以使用“--”或者“-”格式的前导符。

13.2.2　进程的环境变量

对于每个 Linux 系统中的进程来说，都有与进程相关的环境变量。环境变量在编程中非常重要，可以用于保存一些重要的配置信息。如用户在编程过程中，需要保存某个配置项的值，一个方法是将其写入到文件中。而更好的办法是定义成环境变量。

当用户登录 shell 时，会从两个位置获得环境变量的定义。一是全局环境变量文件/etc/profile，另外一个是当前用户的环境变量文件。/etc/profile 中定义的环境变量是对所有的用户都有效的，而用户的环境变量文件只对该用户有效。用户的环境变量文件名称与用户的 shell 类型有关。如果用户的 shell 类型是 bsh，则该文件包括.profile 和.bashrc。这两个文件分别用于不同的登录方式，如果是最常使用的交互式的 shell，则使用.profile 文件。而如果是非交互式的 shell，则使用.bashrc。

可以直接在命令行增加环境变量，通过这种方式增加的环境变量只在本次会话中有效。会话一旦退出，该环境变量将会失效。如果要永久增加某个环境变量，可以在.profile（对于 bsh 来说）中进行。增加环境变量的语法如下所示。

```
export 环境变量名称=值
```

可以通过 env 变量查询当前定义的全部环境变量。如果要查询某个单独的环境变量，可以执行 echo 命令。其语法格式如下所示。

```
echo $环境变量名称
```

要删除某个环境变量的定义，可以执行 unset 命令，其语法如下所示。

```
unset $环境变量名称
```

除通过执行 shell 命令获取环境变量外，也可以通过编程的方式获取。Linux 系统提供了两个系统调用用于获取或者设置环境变量，这两个系统调用分别是 getenv 和 putenv。其声明

位于头文件<stdlib.h>中，原型如下所示。

```
#include <stdlib.h>
char *getenv (__const char *__name);
int putenv (char *__string);
```

（1）参数说明如下。

- ___name：getenv 输入参数，环境变量的名称。
- __string：setenv 输入参数，要设置的环境变量串，其格式为“环境变量名称=值”。

（2）getenv 返回值说明如下。

- NULL：表明相关的环境变量未定义。
- 其他：环境变量的值。

（3）putenv 返回值说明如下。

- -1：调用失败。
- 0：调用成功。

【范例 13-9】编程实现设置环境变量 CONFIG_PATH 的值为/etc。其实现过程如示例代码 13-9 所示。

示例代码 13-9

```
1   #include <stdio.h>                                          /*头文件*/
2   #include <stdlib.h>
3   int main()                                                  /*主函数*/
4   {
5       char *buffer;                                           /*字符串指针，用于保存环境变量*/
6       buffer = getenv ("CONFIG_PATH");                        /*获得环境变量 CONFIG_PATH*/
7       if(buffer==NULL)                                        /*如果环境变量为空，则调用 putenv 设置*/
8       {
9           putenv("CONFIG_PATH=/etc");                         /*设置环境变量*/
10      }
11      printf("CONFIG_PATH=%s\n",getenv("CONFIG_PATH"));       /*获得并输出环境变量的值*/
12      return 0;
13  }
```

【运行结果】经过编译链接，在 shell 下执行程序，其输出信息如下所示。

```
14  CONFIG_PATH=/etc
```

【代码解析】在本例中，首先调用 getenv 检查是否定义环境变量 CONFIG_PATH。如果未定义，则调用 putenv 设置该环境变量。最后输出环境变量的值。本例中，源代码的各行解释如下。

- 第 1～2 行：头文件信息。
- 第 6 行：调用 getenv 获取环境变量的值。如果该值为空，表明未定义环境变量。
- 第 9 行：调用 putenv 设置环境变量。
- 第 11 行：输出环境变量的值。

提示：修改环境变量还可以通过另外的两个系统调用 setenv 和 unsetenv 进行，其实现的功能与 putenv 大同小异，读者有兴趣可以参照相关文档。

13.2.3　进程的内存分配

在编程过程中，根据程序的需要可能需要动态申请内存。通过前面对进程的内存映像的了解可以知道，程序中定义的局部变量是在进程的栈（stack）中分配空间的。其内存的分配与释放不需要用户关心，由系统自动完成。而如果在程序运行时需要动态分配的内存，将从系统可用内存中申请新的空间，并加入到进程的堆（heap）中。动态申请的内存在使用完毕后，应该由用户进行释放。

Linux 提供了专门的用于内存申请及释放的系统调用，分别是申请内存的 malloc、重新申请内存的 realloc 和释放内存的 free。这两个函数的声明位于头文件<stdlib.h>中，其原型如下所示。

```
#include <stdlib.h>
void *malloc (size_t __size);
void *realloc (void *__ptr, size_t __size);
void free (void *__ptr);
```

参数说明如下。

- __size：输入参数，内存缓存区的大小，以字节为单位。
- __ptr：realloc 的输入参数，已有的内存缓存区指针。
- __ptr：free 的输入参数，要释放的内存缓存区指针。

【范例 13-10】编程实现由键盘输入字符，保存于程序中动态分配的空间中。其实现过程如示例代码 13-10 所示。

示例代码 13-10

```
#include <stdio.h>                                          /*头文件*/
#include <stdlib.h>
main()                                                      /*主函数*/
{
    int i;                                                  /*循环变量*/
    char c,*p;                                              /*定义字符变量*/
    p = (char *)malloc(10);                                 /*分配 10 个字节的缓存区*/
    for(i=0;;i++)
    {
        c = getchar();                                      /*从键盘读入单个字符数据*/
        if(i>9)                    /*如果输入字符的个数大于分配的缓冲区，则重新申请内存*/
        {
            p = (char *)realloc(p,1);                       /*重新增加申请一个字节的空间*/
        }
        if(c == '\n')                                       /*输入<Enter>键，退出循环*/
        {
            p[i] = '\0';                                    /*终结字符串*/
            break;
        }
        else
        {
            p[i] = c;                                       /*将输入的字符保存到分配的缓存区*/
        }
```

```
24          }
25          printf("%s\n",p);                          /*输出缓存区中的内容*/
26          free(p);                                   /*释放动态分配的内存*/
27      }
```

【运行结果】经过编译链接，在 shell 下执行程序。通过键盘输入若干数据，输入<Enter>键，程序结束退出。

```
1       ./example13_10
2       1234567890abcde
3       1234567890abcde
```

【代码解析】在本例中，首先调用 malloc 初次分配长度为 10 字节的内存缓存区，然后循环从键盘接收输入的字符。如果输入字符数量超过已申请缓存区大小，则调用 realloc 增加申请更多的内存空间。最后，程序将在接收到回车符后退出。本例中，源代码的各行解释如下。

- 第 1～2 行：头文件信息。
- 第 7 行：调用 malloc 初次分配长度为 10 个字节的内存空间。
- 第 8～24 行：循环调用 getchar 从键盘获取输入数据，并保存至动态分配的空间中。
- 第 10 行：调用 getchar 从键盘获得输入的单个字符。
- 第 11～14 行：如果输入数据的长度已超过初始分配的空间，则调用 realloc 增加分配空间。
- 第 15～19 行：如果输入回车符号，则中断输入退出循环。注意：此处将最后一位字符置为空字符，以便于后续的字符串输出。
- 第 20～23 行：将输入的字符保存到分配的内存空间中。
- 第 25～26 行：将输入的数据输出并释放动态分配的空间。

本例中用于两处转义字符：“\0”表示空，其 ASCII 码值为 0；“\n”表示回车符号。所谓转义，是指用特定格式的字符串表示某个字符。转义主要用来输入不能直接输入的控制类字符。在 C 语言中，用“\”作为转义的前导符。在 Linux 下支持若干转义字符。

提示：如果知道要输入字符的 ASCII 码值，可以通过八进制转义或者十六进制转义方式直接使用。使用八进程表示时，用“\”作为前导，后面跟该字符的三位的八进制 ASCII 码值。使用十六进制时，则使用“\x”作为前导，后面跟两位的十六进制 ASCII 码值。如换行的 ASCII 码值为 10，可以使用\n 表示，也可以用\012（八进制）或者\x0a（十六进制）表示。

13.3 进程的创建

Linux 系统提供了多种创建新进程的方法。这些方法主要包括 fork 系统调用、exec 系列和 system 系统调用。在本节中将逐一进行介绍。

13.3.1 调用 fork 创建进程

创建进程的简单方法是调用 fork。调用完成后，将生成新的进程。此时，新生成的进程

称为子进程，而原来的调用进程称为父进程。fork 系统调用是非常特殊的一个系统调用。调用 fork 一次将返回两次，分别在父进程和子进程中返回。在父进程中，其返回值为子进程的进程标识符。在子进程中，其返回值为 0。其调用过程如图 13-6 所示。

fork 调用成功后，产生的子进程继承了父进程大部分的属性。这些属性主要包括以下几点。

- 进程的实际用户 ID、实际用户组 ID 和有效用户 ID、有效用户组 ID。
- 进程组 ID、会话 ID 及控制终端。
- 当前工作目录及根目录。
- 文件创建掩码 UMASK。
- 环境变量。

除此之外，也有一部分进程属性是不能直接从父进程继承的，主要包括以下几点。

- 进程号、子进程号不同于任何一个活动的进程组号。
- 子进程的用户时间和系统时间，这两个时间被初始化为 0。
- 子进程的超时时钟设置为 0，这个时钟是由 alarm 系统调用使用的。
- 子进程的信号处理函数指针组置为空。原来的父进程中的信号处理函数都将失效。
- 父进程的记录锁。

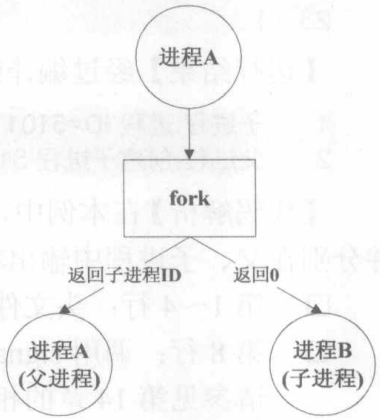


图 13-6　fork 调用过程

在调用 fork 时，应该注意以下几点。

- 父进程中已打开的文件描述符可以在子进程中直接使用。这些描述符不仅包括文件描述符，而且包括其他如套接口描述符等。在这种情况下，这些描述符的引用计数已经加一（每 fork 一次就加一）。因此，在关闭这些描述符时，要记住多次关闭直至描述符的引用计数为 0。
- 子进程复制了父进程的数据段，包括全局变量，但是父、子进程各有一份全局变量的拷贝。因此，不能通过全局变量在父子进程间通信，而要通过专门的进程间通信机制。

【范例 13-11】编程创建多个进程，每个进程输出当前时间。其实现过程如示例代码 13-11 所示。

示例代码 13-11

```
#include <stdio.h>                          /*头文件*/
#include <signal.h>
#include <sys/types.h>
#include <unistd.h>
int main()                                  /*主函数*/
{
    pid_t pid;                              /*进程 ID*/
    signal(SIGCLD, SIG_IGN);                /*信号处理，忽略 SIGCLD 信号，避免形成僵尸进程*/
    switch(pid=fork())                      /*创建子进程*/
    {
        case -1:                            /*创建子进程失败*/
            perror("fork");                 /*输出错误信息*/
```

```
13                  break;
14              case 0:                                 /*子进程*/
15                  printf("子进程:进程 ID=%d\n",getpid());      /*输出当前进程的进程 ID*/
16                  exit(0);
17                  break;
18              default:                                        /*父进程*/
19                  printf("父进程:创建子进程%d 成功.\n", pid); /*输出新创建的子进程的进程 ID*/
20                  sleep(5);                                   /*休眠 5 秒*/
21                  break;
22          }
23  }
```

【运行结果】经过编译链接，在 shell 下执行程序，程序的输出如下所示。

```
1   子进程:进程 ID=5101
2   父进程:创建子进程 5101 成功.
```

【代码解析】在本例中，首先调用 signal 忽略 SIGCLD 信号。然后调用 fork 创建子进程，并分别在父、子进程中输出相应提示信息。本例中，源代码的各行解释如下。

- 第 1～4 行：头文件信息。
- 第 8 行：调用 signal 忽略子进程的退出信号。关于 signal 及 SIGCLD 信号的详情，请参见第 14 章的相关内容。
- 第 9 行：调用 fork 创建子进程。
- 第 11～13 行：调用 fork 失败，输出错误信息后退出。
- 第 14～17 行：fork 的返回码为 0，表明是在子进程中。子进程输出提示信息后退出。
- 第 18～21 行：fork 的返回码非 0，表明是在父进程中，输出新创建的子进程 ID，休眠 5 秒后退出。

提示：Linux 系统中除 fork 外，还有另外一个系统调用 vfork。vfork 系统调用的目的是创建子进程，与 fork 不同的是，vfork 创建子进程的目的是调用 exec，并且 vfork 产生的子进程与父进程共享大多数进程空间。也就是说，vfork 调用成功后，父、子进程共享数据段。关于 vfork 的使用，有兴趣的读者可以查阅相关文档。

13.3.2 调用 exec 系列函数执行程序

exec 系统函数并不创建新进程，调用 exec 前后的进程 ID 是相同的。exec 系列函数的主要工作是清除父进程的可执行代码映像，用新程序的代码覆盖调用 exec 的进程代码。如果 exec 执行成功，进程将从新程序的 main 函数入口开始执行。调用 exec 函数后，除进程 ID 保持不变外，还有下列进程属性保持不变。

- 进程的父进程 ID。
- 实际用户 ID 和实际用户组 ID。
- 进程组 ID、会话 ID 和控制终端。
- 定时器剩余的时间。
- 当前工作目录及根目录。
- 文件创建掩码 UMASK。

- 进程的信号掩码。

exec 系列函数共有 6 种不同的形式，统称为 exec 函数。为讲解清晰，把这 6 个函数划分为两组：一组是 execl、execle、execlp；另一组是 execv、execve、execvp。这两组函数的不同在于 exec 后的第一个字符，第一组是 l，在这里称为 execl 系列；第二组是 v，在这里称为 execv 系列。这里的 l 是 list（列表）的意思，表示 execl 系列函数需要将每个命令行参数作为函数的参数进行传递。而 v 是 vector（矢量）的意思，表示 execv 系列函数将所有函数包装到一个矢量数组中传递即可。exec 函数的声明位于头文件<unistd.h>中，其原型如下所示。

```
#include <unistd.h>
int execv (__const char *__path, char *__const __argv[]);
int execve (__const char *__path, char *__const __argv[],char *__const __envp[]);
int execvp (__const char *__file, char *__const __argv[]);
int execl (__const char *__path, __const char *__arg, ...);
int execle (__const char *__path, __const char *__arg, ...);
int execlp (__const char *__file, __const char *__arg, ...);
```

（1）参数说明如下。

- __path：输入参数，要执行的程序路径。注意：这里是路径名，要求可以是绝对路径或者是相对路径。在 execv、execve、execl、execle 四个函数中，使用带路径名的文件名作为参数。
- __file：输入参数，要执行的程序名称。这里是指文件名。如果该参数中包含“/”字符，则视为路径名直接执行；否则视为单独的文件名，系统将根据 PATH 环境变量指定的路径顺序搜索指定的文件。
- __argv：输入参数，命令行参数的矢量数组。
- __envp：输入参数，带有该参数的 exec 函数，可以在调用 exec 系列函数时，指定一个环境变量数组。其他不带该参数的 exec 系列函数，则使用调用进程的环境变量。
- __arg：程序的第 0 个参数，即程序名自身，相当于__argv[0]。
- ...：输入参数，命令行参数列表。调用相应程序时有多少命令行参数，就需要有多少个输入参数项。注意：在使用此类函数时，在所有命令行参数的最后，应该增加一个空的参数项，表明命令行参数结束。

（2）返回值说明如下。

- -1：表明调用 exec 失败，可以查看 errno 获取详细的错误信息。
- 无返回：表明调用成功。由于调用成功后，当前进程的代码空间被新进程覆盖，所以无返回。

【范例 13-12】编程实现调用执行 ls 命令输出当前目录的文件列表。其实现过程如示例代码 13-12 所示。

示例代码 13-12

```
1   #include <stdio.h>                                  /*头文件*/
2   #include <sys/types.h>
3   #include <unistd.h>
4   main()                                              /*主函数*/
5   {
6       pid_t pid;                                      /*进程标识变量*/
7       char *para[]={"ls","-a",NULL};                  /*定义参数数组，为 execv 所使用*/
```

```
8       if((pid = fork()) < 0)                                  /*创建新的子进程*/
9       {
10              perror("fork");                                 /*错误处理*/
11              exit(0);
12      }
13      if(pid == 0)                                            /*子进程*/
14      {
15          if(execl("/bin/ls","ls","-l",(char *)0) == -1)      /*执行 ls -l 命令*/
16          {
17              perror("execl");                                /*错误处理*/
18              exit(0);
19          }
20      }
21      if((pid = fork()) < 0)                                  /*创建新的子进程*/
22      {
23          perror("fork");                                     /*错误处理*/
24          exit(0);
25      }
26      if(pid == 0)                                            /*子进程*/
27      {
28          if(execv("/bin/ls",para) == -1)                     /*执行 ls –a 命令*/
29          {
30              perror("execv");                                /*错误处理*/
31              exit(0);
32          }
33      }
34      return;
35  }
```

【运行结果】经过编译链接，在 shell 下执行程序，程序的输出如下所示。

```
1   .                   example13_2.c
2   ..                  example13_2
3   example13_1.c   example13_3.c
4   example13_1     example13_3
5   …
6   -rwxr-xr-x   1 develop users 9731 2008-09-25 02:03 example13_1
7   -rwxr-xr-x   1 develop users 9178 2008-09-28 10:07 example13_1.c
8   -rw-r--r--   1 develop users   407 2008-09-28 10:07 example13_2
9   -rwxr-xr-x   1 develop users 9393 2008-09-23 06:50 example13_2.c
10  …
```

【代码解析】在本例中，首先创建新的子进程，在子进程中调用 execl 执行程序 ls –l。然后创建子进程并调用 execv 执行/bin/ls –a。本例中，源代码的各行解释如下。

- 第 1～3 行：头文件信息。
- 第 7 行：定义并初始化调用 execv 所需要的矢量数组。数组中定义了调用 ls 的命令行参数。
- 第 8～12 行：调用 fork 创建新的子进程。
- 第 15～19 行：在子进程中调用 execl 执行程序 ls –l。
- 第 21～25 行：调用 fork 创建新的子进程。
- 第 28～32 行：在子进程中调用 execv 执行程序 ls –a。

13.3.3　调用 system 创建进程

为了方便地调用外部程序，Linux 提供了 system 系统调用。system 将加载外部的可执行程序，执行完毕后返回调用进程。system 的返回码就是加载的外部可执行程序的返回码。system 系统调用的声明位于头文件<stdlib.h>中，其原型如下所示。

```
#include <stdlib.h>
int system (__const char *__command);
```

（1）参数说明如下。

- __command：输入参数，要加载的外部程序的文件名。

（2）返回值说明如下。

- -1：执行 system 失败，可以从 errno 中获取详细的错误信息。
- 127：执行 system 失败。在 system 的内部实现中，system 首先 fork 子进程，然后调用 exec 执行新的 shell，在 shell 中执行要执行的程序。如果在调用 exec 时失败，system 将返回 127。由于要加载的外部程序也有可能返回 127，因此，在 system 返回 127 时，最好判断一下 errno。如果 errno 不为 0，表明调用 system 失败；否则，调用 system 成功，被加载的程序返回码是 127。
- 其他：执行 system 成功，返回值是调用的外部程序的返回码。

【范例 13-13】编程实现调用执行 ls 命令输出当前目录的文件列表。其实现过程如示例代码 13-13 所示。

示例代码 13-13

```
1    #include <stdio.h>                                          /*头文件*/
2    #include <stdlib.h>
3    main()                                                     /*主函数*/
4    {
5        printf("call ls return %d\n",system("ls -l"));         /*调用 system 执行 ls –l 并输出执行的
     返回值*/
6    }
```

【运行结果】经过编译链接，在 shell 下执行程序，程序的输出如下所示。

```
1    -rwxr-xr-x   1 develop users 9731 2008-09-25 02:03 example13_1
2    -rwxr-xr-x   1 develop users 9178 2008-09-28 10:07 example13_1.c
3    -rw-r--r--   1 develop users  407 2008-09-28 10:07 example13_2
4    -rwxr-xr-x   1 develop users 9393 2008-09-23 06:50 example13_2.c
5    …
6    call ls return 0
```

【代码解析】在本例中，直接调用 system 执行命令 ls –l，并输出 system 的返回值。本例中，源代码的各行解释如下。

- 第 1～2 行：头文件信息。
- 第 5 行：调用 system 执行 ls –l 并输出 system 返回值。

13.4 进程的终止

进程执行完毕后，应该合理的终止，释放进程占用的资源。终止进程的方式有多种，可以是接收到其他进程发送的信号而被动终止进程，也可以是进程自己执行完毕后主动退出进程。本节将介绍用于主动退出进程的 exit 函数。从外部发送信号退出进程的过程将在第 14 章中进行讲述。

13.4.1 调用 exit 退出进程

进程要执行的功能执行完毕，或者执行过程中出错，需要调用 exit 退出进程。在 Linux 系统中，除调用 exit 可以结束进程外，还有另外一个函数_exit 也可以实现类似的功能。但是，由于_exit 函数在退出时并不刷新带缓冲 I/O 的缓冲区。所以，在使用带缓冲的 I/O 操作时，应该调用 exit 函数，而不是_exit。exit 函数的声明位于头文件<stdlib.h>中，而_exit 位于<unistd.h>中。其原型如下所示。

```
#include <stdlib.h>
void exit (int __status);
#include <unistd.h>
void _exit (int __status);
```

参数说明如下。

- __status：输入参数，程序退出时的返回码。该返回码可以在 shell 中通过$?系统变量取得，也可以通过 system 系统调用的返回值取得，还可以在父进程中通过调用 wait 函数获得。

13.4.2 调用 wait 等待进程退出

一个进程结束运行时，将向其父进程发送 SIGCLD 信号。这一点将在第 14 章中进行详细讲解。父进程在收到 SIGCLD 信号后，可以忽略该信号或者安装信号处理函数处理该信号。而处理该信号需要调用 wait 系列函数。wait 系列函数的作用是等待子进程的退出，并获取子进程的返回码。

通常情况下，父进程调用 wait 等待其子进程的退出。如果没有任何子进程退出，则 wait 在缺省状态下将进入阻塞状态，直到调用进程的某个子进程退出。wait 系列函数主要有两个：一个是 wait，另一个是 waitpid。这两个函数的声明位于头文件<sys/wait.h>中，其原型如下所示。

```
#include <sys/wait.h>
__pid_t wait (__WAIT_STATUS __stat_loc);
__pid_t waitpid (__pid_t __pid, int *__stat_loc, int __options);
```

（1）参数说明如下。

- __stat_loc：输出参数，用于保存子进程的结束状态。

- __pid：输入参数，用于 waitpid。该参数可以有若干输入方式，每种方式有其独特的含义。
- __options：输入参数，用于 waitpid。该参数指定了调用 waitpid 时的选项。

（2）Linux 提供了多个宏以便从该结束状态中获取特定信息，具体信息如下。

- WIFEXITED(__stat_loc)：如果子进程正常结束则为非 0 值。
- WEXITSTATUS(__stat_loc)：取得子进程 exit()返回的结束代码。通常情况下，应先用 WIFEXITED 来判断是否正常结束才能使用此宏。
- WIFSIGNALED(__stat_loc)：如果子进程是因为信号而结束则返回真。
- WTERMSIG(__stat_loc)：返回子进程因信号而中止的信号代码。通常应先用 WIFSIGNALED 来判断后才使用此宏。
- WIFSTOPPED(__stat_loc)：如果子进程处于暂停执行情况则此宏返回真。只有使用 WUNTRACED 选项时才会有此情况。
- WSTOPSIG(__stat_loc)：返回引发子进程暂停的信号代码。通常应先调用 WIFSTOPPED 来判断后才使用此宏。

（3）返回值说明如下。

- -1：调用失败。
- 其他：调用成功，返回值为退出的子进程 ID。

【范例 13-14】编写代码实现子进程退出。其实现过程如示例代码 13-14 所示。

示例代码 13-14

```
#include <stdio.h>                                          /*头文件*/
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <signal.h>
void handle_sigcld(int signo)                               /*SIGCLD 信号处理函数*/
{
    pid_t pid;                                              /*保存退出进程的进程 ID*/
    int status;                                             /*保存进程的退出状态*/
    if((pid = wait(&status)) != -1)                         /*调用 wait 等待子进程退出*/
    {
        printf("子进程%d 退出\n",pid);                      /*输出提示信息*/
    }
    if(WIFEXITED(status))                                   /*判断子进程退出时是否有返回码*/
    {
        printf("子进程返回%d\n",WEXITSTATUS(status)); /*输出子进程的返回码*/
    }
    if(WIFSIGNALED(status))                                 /*判断子进程是否被信号中断而结束*/
    {
    printf("子进程被信号%d 结束\n",WTERMSIG(status));/*输出中断子进程的信号*/
    }
}
main()                                                      /*主函数*/
{
    pid_t pid;                                              /*定义 pid_t 类型变量，用于保存进程 ID*/
    signal(SIGCLD,handle_sigcld);                           /*安装 SIGCLD 信号*/
```

```
27      if((pid = fork()) < 0)                          /*创建子进程*/
28      {
29          perror("fork");                             /*错误处理*/
30          exit(0);
31      }
32      if(pid == 0)                                    /*子进程*/
33      {
34          exit(123);                                  /*子进程返回 123*/
35      }
36      sleep(5);                                       /*父进程休眠 5 秒,等待子进程退出*/
37  }
```

【运行结果】经过编译链接，在 shell 下执行程序，程序的输出如下所示。

```
1   子进程 5497 退出
2   子进程返回 123
```

【代码解析】在本例中，调用 signal 安装 SIGCLD 信号处理函数，以便在子进程退出时可以捕获信号。在信号处理函数中，调用 wait 等待子进程退出，并输出子进程的返回码。然后创建子进程，子进程未进行任何操作，直接返回。从输出可以看到，父进程通过捕获 SIGCLD 信号，成功获取到子进程的退出代码是 123。本例中，源代码的各行解释如下。

- 第 1～5 行：头文件信息。
- 第 6～21 行：信号 SIGCLD 的信号处理函数。在函数中，调用 wait 等待某个子进程退出。输出子进程的进程 ID 及退出代码。
- 第 26 行：调用 signal 安装 SIGCLD 信号。关于信号安装的详细过程，参见第 14 章的相关内容。
- 第 27～31 行：调用 fork 生成新的子进程。
- 第 34 行：子进程代码。在本例中，子进程未进行任何操作，直接退出。返回代码为 123。
- 第 36 行：父进程睡眠，等待子进程退出。

13.5 小结

在 Linux 系统中，进程是十分重要的概念。当可执行的程序被系统加载到内存空间运行时，就成为一个进程。本章首先讲解了关于进程的重要概念，然后讲解了进程的运行环境，最后重点讲解了如何创建和中止进程。灵活创建和中止进程是 Linux 程序开发的重要技能。本章的部分内容涉及信号的处理，请读者在学习第 14 章时能与本章的内容进行结合，以加深理解。

第 14 章　信　　号

信号是 Linux 系统下基本的进程间通信手段，它为进程提供了异步处理事件的方法。信号可以由系统内核发送至进程，也可以由其他进程发送至进程。系统为进程接收到的信号提供了缺省的处理方式，如忽略或者终止进程等。用户也可以自定义进程接收到信号的处理方式。本章向读者介绍了 Linux 下信号机制的基本原理，并就利用信号进行进程控制做了深入讲解。通过本章的学习，读者应重点掌握如下内容。

- 理解信号的基本概念，重点是信号的产生及处理机制。
- 掌握通过编程进行自定义信号处理的方法，特别是两个重要信号的处理 SIGCLD 和 SIGALRM。
- 了解信号的阻塞及未决信号等概念，在编程过程中灵活加以运用。

14.1　信号的基本概念

14.1.1　信号的定义

信号（signal）又称为软中断，用来通知进程发生了异步事件。信号何时产生、由什么来源产生，是由进程自身决定的，只有接收到信号后才能确定。信号的接收过程是异步的过程。也就是说，进程在正常运行过程中，随时可能被各种信号所中断。进程可以忽略该信号，也可以中断当前执行的程序转而调用相应的函数去处理信号。待信号处理完成后，继续执行被中断的程序。Linux 系统下信号的模型如图 14-1 所示。

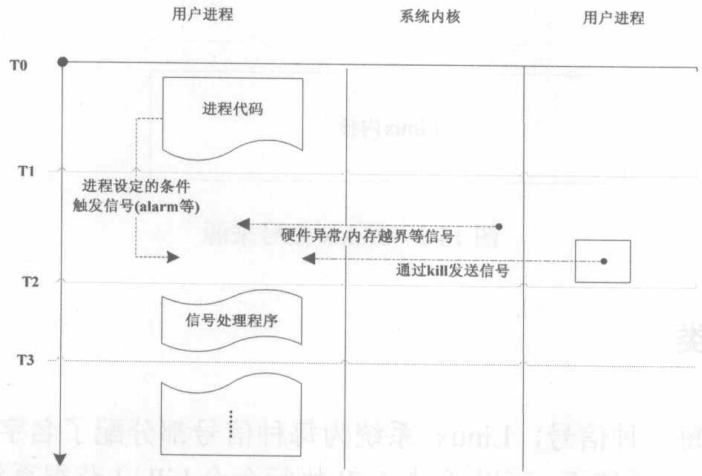


图 14-1　Linux 系统下信号的模型

提示：Linux 系统下的信号是异步产生的。而习惯了面向过程编程模式的用户，往往对信号的异步机制不容易理解。其实，Linux 系统下的信号机制与 Windows 下的消息机制是相似的。众所周知，Windows 系统下的编程模型中，消息驱动是很重要的一个机制。消息可能由系统产生，如重绘窗口的 WM_PAINT 消息，也可能由其他进程通过 SendMessage 发送过来。

14.1.2 信号的来源

只要具备相应权限，进程之间可以互相通过系统调用 kill 发送信号。操作系统内核也可能因为内部事件而给进程发送信号，通知进程有某个事件发生。因此，信号的来源可能是系统内核，也可能是其他的进程。引起信号产生的原因基本上包括以下几种。

- 程序中执行错误的代码。如内存访问越界、数学运算除零等。
- 其他进程发送来的信号。同一系统中的其他进程可以通过执行 kill 命令或者系统调用向当前进程发送信号。不过，要求发送信号的进程有向当前进程发送信号的权限。
- 用户通过控制终端发送来的信号。最常见的情况就是：一个程序在运行中，用户通过在键盘输入<Ctrl>+<C>键或者<Ctrl>+<\>键中止程序的执行。
- 子进程结束时向父进程发送的 SIGCLD 信号。
- 程序中设定的定时器生产的 SIGALRM 信号。

形象地进行描述 Linux 信号来源，如图 14-2 所示。

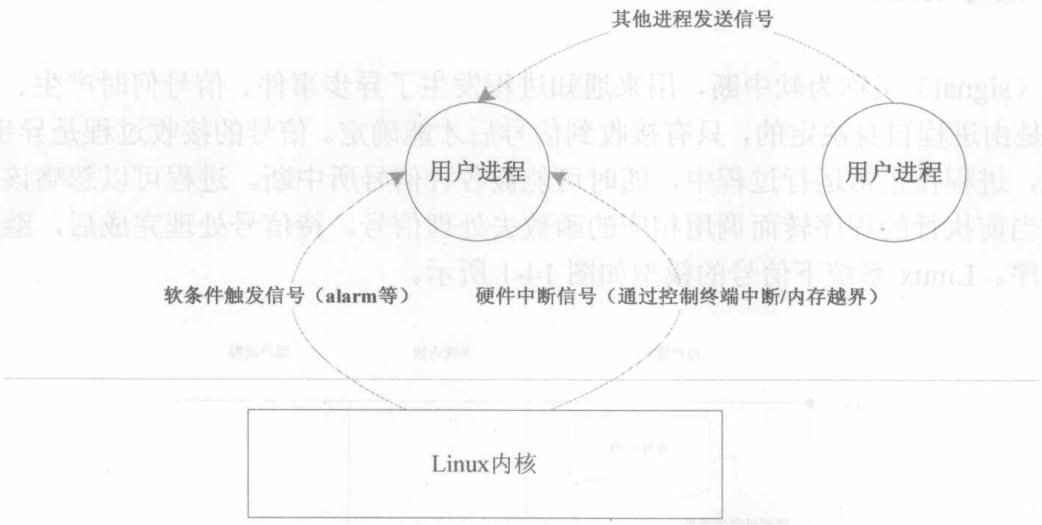


图 14-2 Linux 信号来源

14.1.3 信号的分类

为更方便地标识每一种信号，Linux 系统为每种信号都分配了名字。这些名字都以 SIG 开头进行定义。在 Linux 系统下，可以通过 shell 执行命令 kill –l 获得系统中全部信号的列表，如下所示。

```
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
.....
```

在 kill –l 的输出中，每个信号前都有一个数字标识，这就是信号的值。在程序中可以直接使用该整型值来代表某个信号。当然，为保证程序具有良好的可读性，使用信号的符号名称还是很有必要的。这些信号的名称及其值的定义在头文件<asm/signal.h>中。

Linux 系统中的信号可以划分为可靠的信号和不可靠的信号。可靠信号是指信号一旦发出，操作系统保证该信号不会丢失。而不可靠信号由于内核不对信号进行排队，造成的后果就是信号有可能丢失。在 Linux 的信号中，编号值是 1～31（SIGHUP～SIGSYS）的信号是不可靠信号，而编号为 32～63 的信号是可靠信号。需要注意的是，信号的可靠与否与信号是如何发送的是无关的，它只与信号的类型有关。

Linux 系统中常见的信号说明如下。

- SIGHUP：该信号是在进程的控制终端注销时产生的，由系统内核发往该终端上运行的所有进程。用户在登录 Linux 系统时，系统将会分配给该用户一个控制终端（/dev/tty1、/dev/pts/0 等，可以通过执行命令 tty 查看当前的控制终端）。所有在此终端上启动的用户进程，其控制终端就是该终端。如果此时用户从该终端注销，那么已启动的所有进程将收到 SIGHUP 信号。在系统缺省状态下对该信号的处理就是中止进程。
- SIGINT：程序终止信号。在程序运行过程中，用户通过键盘按下<Ctrl>+<C>键将产生该信号。
- SIGQUIT：程序退出信号。在程序运行过程中，用户通过键盘按下<Ctrl>+<\>键将产生该信号。与 SIGINT 不同的是，SIGQUIT 信号将产生系统核心转储 core 文件。core 文件是程序退出时的内存映像文件，包含了与程序相关的调试信息。
- SIGBUS 和 SIGSEGV：进程访问非法地址时，将引发该信号。如果地址是有效的，不过不属于当前进程的地址空间，则引发 SIGSEGV 信号。而如果地址本身是无效的，将引发 SIGBUS 信号。
- SIGFPE：进行算术运算中出现致命错误，如除零操作、数据溢出等。
- SIGKILL：终止用户进程的执行信号。该信号不能被忽略或者被用户捕获。一旦收到该信号，进程将立即中止。在 shell 下通过执行 kill -9 命令发送的就是该信号。
- SIGTERM：进程结束信号。与 SIGKILL 不同的是，该信号可以被用户捕获并处理。在 shell 下执行 kill 命令发送的就是该信号。
- SIGALRM：定时器信号。可以在程序中通过 alarm 系统调用设置一个超时时钟，在规定的时间到达后，进程将收到该信号。如在进程中调用 alarm(60)，则 60 秒后该进程将会收到 SIGALRM 信号。该信号常用作进行一些阻塞性操作的超时控制。
- SIGCLD：子进程退出信号。在一个进程创建一个或者若干子进程后，每个子进程退出时，父进程都将收到该信号。如果父进程没有忽略该信号，也没有处理该信号，则子进程退出后将形成僵尸进程。

14.2 信号的安装及处理

如果进程不进行任何信号控制操作，那么收到信号后将按照缺省规则进行处理。要使进程在接收到信号后进行自定义的处理，就要进行信号的安装工作。所谓信号的安装，就是在程序中指定某种信号的处理方式。

14.2.1 信号的处理方式

在 Linux 系统下，信号的处理有 3 种方式：一是忽略信号，在进程收到该种信号后，将不进行任何处理，好像没有收到信号一样；二是按照系统提供的缺省处理规则进行处理；三是捕捉信号，在程序中定义自己的信号处理函数，在信号处理函数中完成相应的功能。Linux 操作系统分别为前两种方式提供了相应的宏定义：SIG_IGN 和 SIG_DFN。可以在程序中通过系统调用 signal 来指定进程对某个信号的处理行为。在后面介绍信号安装时将详细介绍。

- 忽略信号：大多数信号都可以被忽略。但是 SIGKILL 和 SIGSTOP 除外，这两种信号是不能被忽略的。操作系统通过这两个信号的不可忽略机制，确保可以终止进程。
- 捕捉信号：要捕捉一个信号，首先要进行信号的安装，告诉进程在收到该信号后，执行自定义的函数，以完成特定的功能。
- 执行系统默认动作：在程序中如果不进行任何信号处理操作，则该程序运行过程中收到的信号将按照缺省处理方式进行。Linux 规定了缺省方式下每种信号的处理规则。主要信号的处理规则如表 14-1 所示。

表 14-1 主要信号的处理规则

信号名称	缺省处理方式	信号产生原因
SIGHUP	终止进程	控制终端挂起或者退出
SIGINT	终止进程	通过键盘中断（一般为<Ctrl>+<C>键）
SIGQUIT	终止进程并进行内核映像转储	终端输入退出键（一般为<Ctrl>+<\>键）
SIGKILL	终止进程	用于终止进程执行，不能阻塞、忽略、捕捉
SIGALRM	终止进程	定时器超时信号产生（alarm 系统调用）
SIGTERM	终止进程	用于终止进程执行，可以阻塞、捕捉
SIGCHLD	忽略信号	子进程退出时将向父进程发送此信号
SIGSTOP	暂停进程执行	用于暂停进程执行，不能阻塞、忽略、捕捉

14.2.2 用 signal 安装信号

在用户的应用程序中，可以自行安装信号，定义进程收到信号后的处理方法。这一过程是通过系统调用 signal 或者 sigaction 完成的。本节介绍 signal 系统调用，下一节将介绍 sigaction 系统调用。signal 系统调用用于安装信号，该函数的声明在头文件<signal.h>中，其原型如下。

```
#include <signal.h>
__sighandler_t signal (int __sig, __sighandler_t __handler);
```

（1）参数说明如下。

- __sig：输入参数，信号名称，指定要安装的信号名称。该参数是一个整型值。
- __handler：输入参数，指向信号处理函数的指针，用于指定收到信号后，调用的信号处理函数。除自定义处理函数指针外，该参数还可以有下述选择。SIG_DFN：采用缺省的信号处理方式；SIG_IGN：忽略该信号。

（2）返回值说明如下。

- SIG_ERR：安装信号失败。
- 其他值：原来的信号处理函数指针。

【范例 14-1】为进程安装 SIGINT 信号。其实现过程如示例代码 14-1 所示。

示例代码 14-1

```
1   #include <stdio.h>                                  /*头文件*/
2   #include <signal.h>
3   void HandleSigint(int signo)                        /*SIGINT 信号处理函数*/
4   {
5       printf("receive signal %d.\n",signo);           /*输出信号的编号*/
6   }
7   main()                                              /*主函数*/
8   {
9       if(signal(SIGINT,HandleSigint)  == SIG_ERR)     /*安装 SIGINT 信号*/
10      {
11          perror("signal");                           /*错误处理*/
12          exit(0);
13      }
14      pause();                                        /*暂停进程，等待信号*/
15  }
```

【运行结果】经过编译链接，在 shell 下运行上述程序，其结果如下所示。

```
16  ./example14_1
17                  ←输入<Ctrl>+<C>键
18  receive signal 2.
```

【代码解析】在本例中，首先通过系统调用 signal 安装信号 SIGINT。在进程接收到 SIGINT 信号后，将执行函数 HandleSigint。然后，调用 pause 等待信号的产生。本例中，源代码的各行解释如下。

- 第 1～2 行：头文件信息。
- 第 3～6 行：定义接收到信号 SIGINT 后的处理函数。在本例中，只是简单输出了信号的值就返回。
- 第 9～13 行：调用 signal 安装 SIGINT 的信号处理函数。
- 第 14 行：调用 pause 阻塞进程，直到某个信号的到来。

通过程序运行的输出可以看到，进程捕捉到了用户通过终端输入的<Ctrl>+<C>键信号。

提示：通过 signal 函数安装的信号，在发送信号时不能附带数据，且不能设置信号的一些选项，如系统调用被信号中断后是否自动重启系统调用等。如果需要在发送信号时附加数

据，或者进行更高级的信号控制，建议使用后面介绍到的 sigaction。

14.2.3 用 sigaction 安装信号

signal 可以实现基本的信号安装功能。但在有些情况下，可能需要对信号的安装进行更多的控制。此时，就需要使用另外一个信号安装的系统调用 sigaction。sigaction 的声明位于头文件<signal.h>中，其原型如下所示。

```
#include <signal.h>
int sigaction (int __sig, __const struct sigaction *__restrict __act,struct sigaction *__restrict __oact);
```

（1）参数说明如下。

- __sig：输入参数，指定要安装的信号。
- __act：输入参数，指定安装信号的数据结构。该参数是一个 struct sigaction 类型的结构指针。该结构的定义位于头文件<bits/sigaction.h>中，其定义如下。

```
struct sigaction
 {
#ifdef __USE_POSIX199309
 union
 {
 __sighandler_t sa_handler;
 void (*sa_sigaction) (int, siginfo_t *, void *);
 }__sigaction_handler;
# define sa_handler   __sigaction_handler.sa_handler
# define sa_sigaction    __sigaction_handler.sa_sigaction
#else
     __sighandler_t sa_handler;
#endif
     __sigset_t sa_mask;
     int sa_flags;
     void (*sa_restorer) (void);
  };
```

（2）该结构是 sigaction 调用的核心，其结构成员说明如下。

- __sigaction_handler：用于定义信号的处理函数，实际上是指向信号处理函数的指针。可以看到，该成员是一个联合类型。从联合的成员可以看出，该成员可以定义两种不同类型的信号处理函数。一种是与 signal 调用相同的处理函数，其类型为 __sighandler_t，这种类型的函数只有一个输入参数，即信号值；另一种则是只被 sigaction 系统调用所支持的函数，该函数的声明是：void (*sa_sigaction) (int, siginfo_t *, void *)。在后面介绍附带数据发送信号时将对该结构进行详细介绍。
- sa_mask：信号掩码。在信号处理函数执行的过程中，在该掩码中定义的信号将被阻塞。
- sa_flags：标志位。sigaction 支持若干标志位，主要的有 SA_SIGINFO、SA_RESTART 等。其中 SA_SIGINFO 标志表明该信号在发送时可以附带其他数据；SA_RESTART 参数表明如果信号中断了进程的某个系统调用，则由系统自动重启该系统调用。如果不指定该参数，则被中断的系统调用将返回失败，错误码为 EINTR。
- __oact：输出参数，指向 struct sigaction 结构的指针。调用 sigaction 成功后，该参数中将返回信号的原来处理方式。如果不需要获取原信号处理方式，则该参数可以传

入 NULL。

（3）返回值说明如下。

- 0：调用成功。
- -1：调用失败，可以查看 errno 获取错误信息。

用 signal 安装的信号，无法在发送信号时附加其他数据。而 sigaction 函数通过 SA_SIGINFO 标志安装的信号，则可能实现这一点。这一功能在有些情况下非常有用，读者应该尤其注意。发送信号时附加参数的过程如图 14-3 所示。

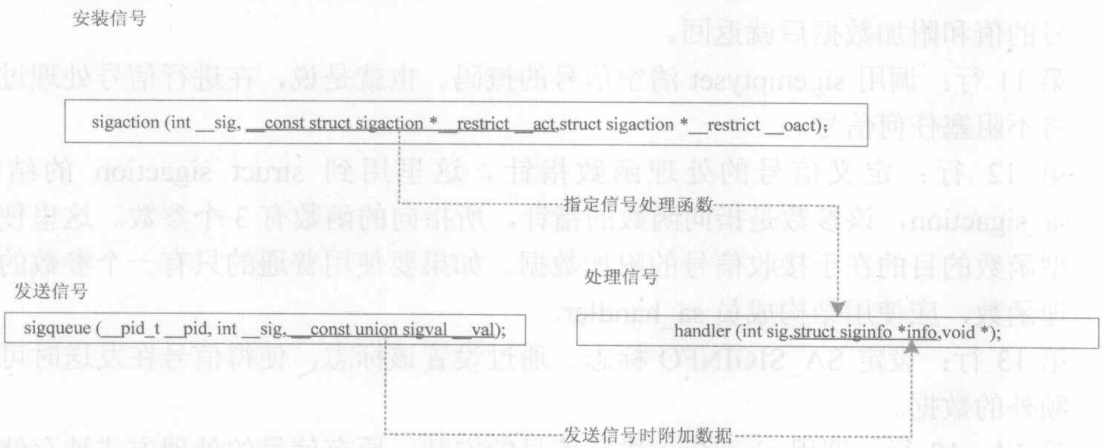


图 14-3　发送信号时附加参数的过程

【范例 14-2】用 sigaction 为进程安装 SIGINT 信号，要求在向进程发送信号时可以附加数据。其实现过程如示例代码 14-2 所示。

示例代码 14-2

```
#include <stdio.h>                                          /*头文件*/
#include <signal.h>
void HandleSigint(int signo,siginfo_t *info,void *none)     /*SIGINT 信号处理函数*/
{
    printf("receive signal %d,addtional data is %d\n",signo,info->si_value.sival_int);/*输出信号附加数据*/
}
main()
{
    struct sigaction act,oact;                              /*安装信号处理函数的结构*/
    memset(&act,0x00,sizeof(struct sigaction));             /*清空结构*/
    sigemptyset(&act.sa_mask);                              /*清空信号处理掩码*/
    act.sa_sigaction = HandleSigint;                        /*定义信号的处理函数*/
    act.sa_flags = SA_SIGINFO;                              /*指定发送信号时可以附加数据*/
    if(sigaction(SIGINT,&act,&oact) == -1)                  /*安装信号*/
    {
        perror("sigaction");                                /*错误处理*/
        exit(0);
    }
    pause();                                                /*暂停进程执行，等待信号*/
}
```

【运行结果】经过编译链接，在 shell 下运行上述程序，其结果如下所示。

```
./example14_2
                    ←输入<Ctrl>+<C>键
receive signal 2,addtional data is 0
```

【代码解析】在本例中，首先通过系统调用 sigaction 安装信号 SIGINT。在进程接收到 SIGINT 信号后，将执行函数 HandleSigint。然后，调用 pause 等待信号的产生。本例中，源代码的各行解释如下。

- 第 1～2 行：头文件信息。
- 第 3～6 行：定义接收到信号 SIGINT 后的处理函数。在本例中，只是简单输出了信号的值和附加数据后就返回。
- 第 11 行：调用 sigemptyset 清空信号的掩码。也就是说，在进行信号处理过程中，将不阻塞任何信号。
- 第 12 行：定义信号的处理函数指针。这里用到 struct sigaction 的结构成员 sa_sigaction，该参数是指向函数的指针，所指向的函数有 3 个参数。这里使用该类型函数的目的在于接收信号的附加数据。如果要使用普通的只有一个参数的信号处理函数，应使用结构成员 sa_handler。
- 第 13 行：设定 SA_SIGINFO 标志。通过设置该标志，使得信号在发送时可以附加额外的数据。
- 第 14～18 行：调用 sigaction 进行信号的安装。原有信号的处理方式被存储到 oact 结构中。
- 第 19 行：调用 pause 阻塞进程，直到某个信号的到来。

在本例中，定义了进程接收到的信号中可以附加其他数据。在程序执行时，只是简单地输入<Ctrl>+<C>键发送了 SIGINT 信号，并没有附加额外数据。在下一节介绍附加数据发送信号时，将使用该范例接收带附加数据的信号。

提示：在本例中，要特别注意对结构 struct sigaction 的赋值，这与 signal 有较大差异。特别是对信号处理函数的定义，本例中使用了可以附加额外数据的信号处理函数。在编程中要格外注意这种使用方法。

14.2.4 信号的阻塞处理

信号的阻塞就是通知系统内核暂时停止向进程发送指定的信号，而是由内核对进程接收到的相应信号进行缓存，直到进程解除对相应信号的阻塞为止。一旦进程解除对该信号的阻塞，则缓存的信号将发送到相应进程。信号阻塞的过程如图 14-4 所示。

信号在几种情况下会进入阻塞状态。一是在信号的处理函数执行过程中，该信号将被阻塞，直到信号函数执行完成，该阻塞将会解除。这种机制的作用主要是避免信号的嵌套；第二种情况是通过 sigaction 进行信号安装时，如果设置了 sa_mask 阻塞信号集，则该信号集中的信号在信号处理函数被执行期间将会阻塞；还有另外一种情况下信号将进入阻塞，这就是通过系统调用 sigprocmask，可以通过该系统调用指定阻塞某个或者某几个信号。

在引起信号阻塞的 3 种情况中，第一种可以认为是由系统自动阻塞的。后面两种情况是人为设置的信号阻塞。

第二种情况下进行信号阻塞的原因是：一个信号处理函数在执行过程中，可能会有其他信号到来。此时，当前的信号处理函数就会被中断。而这往往是不希望发生的。此时，可以通过 sigaction 系统调用的信号阻塞掩码对相关信号进行阻塞。通过这种方式阻塞的信号，在信号处理函数执行结束后就会解除。

第三种情况下进行阻塞的原因较多，一个典型的情况是：某个信号的信号处理函数与程序代码都要对某个共享数据区进行读写。如果在程序正在进行读写的过程中，进程收到了该信号，则程序的读写过程被中断转而执行信号处理函数，而信号处理函数也要对该共享数据区进行读写，此时共享数据区就会发生混乱。这种情况下，需要在程序读写共享数据区前阻塞信号，在读写完成后再解除信号阻塞。

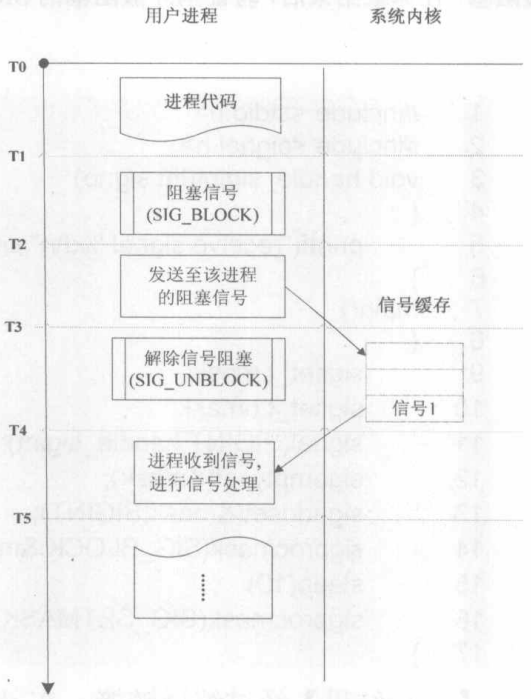


图 14-4　信号的阻塞

提示：实际上在信号的接收过程中可能存在这样的情况，若干个相同的信号同时到达。通过上面的介绍可以知道，信号处理函数执行时，同类信号将被阻塞。但是，如果用户定义的信号处理函数还没有来得及执行，此时该信号也就不会阻塞。在这种情况下，将会发生一种称为“信号合并”的现象。同时到达的这些信号将被合并处理，就像只有一个信号到达一样。

被阻塞的信号的集合称为当前进程的信号掩码。每一个进程都有一个惟一的信号掩码。为了对信号进行阻塞或者解除阻塞，Linux 提供了专门的系统调用 sigprocmask 完成这一任务。该函数的声明位于头文件<signal.h>中，其原型如下所示。

```
#include <signal.h>
int sigprocmask (int __how, __const sigset_t *__restrict __set,sigset_t *__restrict __oset);
```

（1）参数说明如下。

- __how：输入参数，设置信号阻塞掩码的方式。可以包括 3 种方式对信号的掩码进行设置，分别是阻塞信号的 SIG_BLOCK、解除阻塞的 SIG_UNBLOCK 和设置阻塞掩码的 SIG_SETMASK。
- __set：输入参数，阻塞信号集。当参数__how 为 SIG_BLOCK 时，该参数表明要阻塞的信号集；当__how 参数为 SIG_UNBLOCK 时，该参数表明要解除阻塞的信号集；当__how 参数为 SIG_SETMASK 时，该参数表明要阻塞的信号集。
- __oset：输出参数，原阻塞信号集。调用成功后，将返回该进程原阻塞信号集。

（2）返回值说明如下。

- 0：调用成功。
- -1：调用失败，可以通过 errno 获取详细错误信息。

【范例 14-3】通过编程实现下述功能：为进程安装 SIGINT 信号，先阻塞信号 SIGINT，休眠 10 秒，再

解除 SIGINT 的阻塞。程序运行时，在休眠期间，按下<Ctrl>+<C>键向进程发送 SIGINT 信号，看信号是否被阻塞。在休眠结束后，验证刚才被阻塞的 SIGINT 信号是否被重新发送。其实现过程如示例代码 14-3 所示。

示例代码 14-3

```
1   #include <stdio.h>                          /*头文件*/
2   #include <signal.h>
3   void handle_sigint(int signo)               /*SIGINT 信号处理函数*/
4   {
5       printf("receive signal %d\n",signo);    /*输出信号的值*/
6   }
7   main()
8   {
9       sigset_t mask;                          /*信号掩码结构变量，用于指定新的信号掩码*/
10      sigset_t omask;                         /*信号掩码结构变量，用于保存原来的信号处理掩码*/
11      signal(SIGINT,handle_sigint);           /*安装 SIGINT 信号*/
12      sigemptyset(&mask);                     /*清空信号处理掩码*/
13      sigaddset(&mask,SIGINT);                /*向掩码结构中增加信号 SIGINT*/
14      sigprocmask(SIG_BLOCK,&mask,&omask);/* 设置掩码，设置完成后 SIGINT 信号将被阻塞*/
15      sleep(10);                                      /*休眠 10 秒*/
16      sigprocmask(SIG_SETMASK,&omask,NULL);   /*恢复原有的信号处理掩码*/
17  }
```

【运行结果】经过编译链接，在 shell 下运行上述程序，其结果如下所示。

```
18  ./example14_3
19                      ←输入<Ctrl>+<C>键
20                      进程无返回，说明未收到信号（信号被阻塞）
21  receive signal 2    10 秒休眠结束后，进程收到信号（信号被重新发送）
```

【代码解析】在本例中，首先通过系统调用 signal 安装信号 SIGINT，然后调用 sigprocmask 阻塞 SIGINT 信号，接着进程进入 10 秒的休眠。最后解除 SIGINT 信号的阻塞。可以看到，在解除阻塞后，阻塞期间发出的 SIGINT 信号又被重新发送。本例中，源代码的各行解释如下。

- 第 1～2 行：头文件信息。
- 第 3～6 行：定义接收到信号 SIGINT 后的处理函数。
- 第 11 行：调用 signal 安装信号 SIGINT，信号的处理函数为 handle_sigint。
- 第 12～14 行：调用 sigprocmask，阻塞信号 SIGINT。
- 第 15 行：进程休眠 10 秒。此处进行休眠仅是为了测试信号是否被阻塞的需要。
- 第 16 行：调用 sigprocmask，解除信号 SIGINT 的阻塞。在解除阻塞后，原来被阻塞的信号将立即发送到进程。

提示：在创建新的子进程时，子进程将继承父进程的信号掩码。这一点在编程过程中应该注意。

14.2.5 信号集的操作

通过上节对信号阻塞的介绍可以知道，信号的阻塞实际上是对一个集合的操作。这个集合中可能包含多种信号，这就是信号集。信号集的数据类型为 sigset_t，实际上是一个结构，

结构中的惟一成员是一个 long 型数组，用于存储多个信号。

Linux 系统提供了多种函数对信号集进行操作。这些函数包括清空信号集的 sigemptyset，向信号集填充全部信号的 sigfillset，向信号集中增加信号的 sigaddset，从信号集中删除信号的 sigdelset 以及判断信号集是否包含某信号的 sigismember。这些函数的声明位于头文件<signal.h>中，其原型如下所示。

```
#include <signal.h>
int sigemptyset (sigset_t *__set) ;
int sigfillset (sigset_t *__set);
int sigaddset (sigset_t *__set, int __signo);
int sigdelset (sigset_t *__set, int __signo);
int sigismember (__const sigset_t *__set, int __signo);
```

（1）参数说明如下。

- __set：输入参数，信号集。
- __signo：输入参数，要增加或者删除的信号。

（2）返回值说明如下。

- 0：表示成功。
- -1：表示失败。

对于函数 sigismember 来说，返回 1 表明为真，信号属于信号集；返回 0 表明为假，信号不属于信号集。

14.2.6　未决信号的处理

信号的未决是信号产生后的一种状态，是指从信号产生（可能由硬件产生，也可能由其他进程发送）后，到信号被接收进程处理之前的一种过渡状态。

由于信号的未决状态时间非常短，所以通常情况下，处于未决状态的信号非常少。如果程序中使用了 sigprocmask 阻塞了某种信号，则向进程发送的这种信号将处于未决状态。Linux 系统提供了专门的函数 sigpending 获取当前进程中处于未决状态的信号。函数的声明位于头文件<signal.h>中，其原型如下所示。

```
#include <signal.h>
int sigpending (sigset_t *__set);
```

（1）参数说明如下。

- __set：输出参数，处于未决状态的信号集。

（2）返回值说明如下。

- 0：调用成功。
- -1：调用失败，查看 errno 可以获取到详细错误信息。

【范例 14-4】通过编程实现下述功能：为进程安装 SIGINT 信号，先阻塞信号 SIGINT，休眠 10 秒，最后查看当前进程未决的信号。程序运行时，在休眠期间，按下<Ctrl>+<C>键向进程发送 SIGINT 信号，验证该信号是否处于未决状态。其实现过程如示例代码 14-4 所示。

示例代码 14-4

```
1   #include <stdio.h>                                  /*头文件*/
2   #include <signal.h>
3   main()                                              /*主函数*/
4   {
5       sigset_t mask;                                  /*信号掩码结构变量定义*/
6       sigset_t omask;                                 /*定义 sigset_t 变量*/
7       sigset_t pmask;                                 /*定义 sigset_t 变量*/
8       sigemptyset(&mask);                             /*清空掩码结构*/
9       sigaddset(&mask,SIGINT);                        /*向掩码结构中增加 SIGINT 信号*/
10      sigprocmask(SIG_BLOCK,&mask,&omask);            /*阻塞信号 SIGINT*/
11      sleep(10);                                      /*休眠 10 秒*/
12      if(sigpending(&pmask) < 0)                      /*获得当前未决的信号集*/
13      {
14          perror("sigpending");                       /*错误处理*/
15          exit(0);
16      }
17      if(sigismember(&pmask,SIGINT))                  /*判断 SIGINT 是否在未决信号集中*/
18      {
19          printf("SIGINT signal is pending.\n");      /*输出提示信息*/
20      }
21  }
```

【运行结果】经过编译链接，在 shell 下运行上述程序，其结果如下所示。

```
1   ./example14_4
2                       ←输入<Ctrl>+<C>键
3                       进程无返回，说明未收到信号（信号被阻塞）
4   SIGINT signal is pending.   10 秒休眠结束后，当前进程的未决信号中有 SIGINT 信号
```

【代码解析】在本例中，首先通过调用 sigprocmask 阻塞 SIGINT 信号。接着进程进入 10 秒的休眠。最后调用 sigpending 获得当前进程未决的信号。可以看到，在休眠结束后，阻塞期间发出的 SIGINT 信号状态是未决的。本例中，源代码的各行解释如下。

- 第 1～2 行：头文件信息。
- 第 8～10 行：调用 sigprocmask，阻塞信号 SIGINT。
- 第 11 行：进程休眠 10 秒。进行休眠仅是为了使运行期间产生的 SIGINT 进入未决状态。
- 第 12 行：进程休眠结束后，调用 sigpenging 获得当前未决的信号集。
- 第 17～20 行：调用 sigismember 函数判断当前的未决信号集中是否包含 SIGINT 信号。

14.2.7 等待信号

一个阻塞式系统调用在执行过程中如果没有符合条件的数据，将进入休眠状态，直到有符合条件的数据到来。比较典型的例子是在从网络连接上读取数据，如果没有数据到来时，这个读操作将会阻塞。有两种情况可以中断该读操作的执行：一是网络上有数据到来，则读操作将获取到所需要的数据后返回；二是当前进程收到了某种信号，此时，读操作将被中断返回失败，错误码 errno 为 EINTR。

在有些情况下，程序需要暂时停止执行，进行休眠状态，以等待信号的到来。这时需要用到 pause 系统调用。pause 系统调用与上述阻塞式系统调用相同，一旦调用则进程将进入休眠状态。但是，只有在进程接收到信号后，pause 才会返回。这一点与阻塞式系统调用是不同的。

pause 系统调用的声明在头文件<unistd.h>中，其原型如下所示。

```
#include <unistd.h>
int pause (void);
```

pause 的返回值永远是-1，返回的错误码是 EINTR。

在 Linux 系统下，pause 可以实现暂停进程的执行等待某个信号的到来。但是，如果在 pause 调用被执行到之前，指定的信号到达进程。那么，在随后的 pause 调用中，假定不再有信号到来，则程序将进入无限期的等待中。为此 Linux 提供了功能更为强大的 sigsuspend 可以满足这种要求。sigsuspend 函数的作用是设置当前进程的信号掩码，然后进程进入休眠，直到某个信号到达。其工作的过程如下。

（1）设置进程的信号掩码并阻塞进程。

（2）收到信号，恢复原来的信号掩码。

（3）调用进程设置的信号处理函数。

（4）等待信号处理函数返回后，sigsuspend 返回。

上述 4 步操作是一次性完成的，操作系统保证操作过程的原子性。特别需要注意的是第三步调用信号处理函数是由 sigsuspend 完成的，利用 sigsuspend 可以更好的实现信号等待。该函数的声明位于头文件<signal.h>中，其原型如下所示。

```
#include <signal.h>
int sigsuspend (__const sigset_t *__set);
```

参数__set 表示输入参数，执行 sigsuspeng 过程中阻塞的信号集。sigsuspeng 返回值永远为-1。

【范例 14-5】用 sigsuspeng 实现等待 SIGINT 信号到来的功能。其实现过程如示例代码 14-5 所示。

示例代码 14-5

```
#include <stdio.h>                                          /*头文件*/
#include <signal.h>
int nInterrupt;                                             /*全局变量，用于控制是否已安装完信号处理*/
void handle_sigint(int signo) /*SIGINT 的信号处理函数,本例中该函数将在 sigsuspend 阻塞期间收到*/
{
    nInterrupt = 1;                                         /*设置中断标志为真*/
}
main()                                                      /*主函数*/
{
    sigset_t mask,omask;                                    /*掩码结构变量*/
    signal(SIGINT,handle_sigint);                           /*安装 SIGINT 的信号处理*/
    sigemptyset(&mask);                                     /*清空结构*/
    sigaddset(&mask,SIGINT);                                /*将 SIGINT 加入掩码结构*/
    sigprocmask(SIG_BLOCK,&mask,&omask);                    /*阻塞 SIGINT 信号*/
    nInterrupt = 0;                                         /*置中断标志为假*/
    while(!nInterrupt) /*循环调用 sigsuspend 等待信号，直到收到 SIGINT,ninterrupt 为 1,退出循环*/
```

```
17        sigsuspend(&omask);                          /*阻塞进程,直到有信号到达。以 omask 调
    用 suspend,所以首先将解除阻塞 SIGINT*/
18        printf("process return.\n");                 /*输出提示信息*/
19    }
```

【运行结果】经过编译链接，在 shell 下运行上述程序，其结果如下所示。

```
1    ./example14_5
2    process return.    ←输入<Ctrl>+<C>键
```

【代码解析】在本例中，首先通过调用 signal 安装 SIGINT 信号（该信号处理函数实现对全局变量 nInterrupt 赋值 1），然后阻塞 SIGINT 信号，最后循环调用 sigsuspend 等待 SIGINT 信号的到来。本例中，由于只有信号 SIGINT 的处理函数将 nInterrupt 的值置为 1，所以在进程接收到其他信号时，将正常处理（由 sigsuspend 调用信号处理函数进行处理）。在进程接收到 SIGINT 信号后，while 循环将退出。本例中，源代码的各行解释如下。

- 第 1～2 行：头文件信息。
- 第 3 行：定义全局变量 nInterrupt。该变量用于判断进程是否收到了 SIGINT 信号。
- 第 4～7 行：定义 SIGINT 的信号处理函数。函数中将全局变量 nInterrupt 的值置为 1。
- 第 11 行：安装 SIGINT 信号。
- 第 12～14 行：设置当前进程的信号掩码为阻塞 SIGINT 信号，并将原信号掩码存储至 omask。该操作的目的是保证在后面代码中对 nInterrupt 赋值时，不会接收到 SIGINT 信号，以免发生赋值冲突。
- 第 15 行：为 nInterrupt 赋初值。
- 第 16～17 行：循环调用 sigsuspend 等待信号 SIGINT。

14.2.8 信号处理函数的实现

信号处理函数是进程接收到信号后要执行的函数。该函数应该尽量简洁，一般不要执行过多代码。最好只是改变一个外部标志变量的值，而在另外的程序中不断的检测该变量，繁杂的工作都留给这些程序去做。在定义信号处理函数中，应该特别注意以下几点。

- 如果信号处理程序中需要存取某一个全局变量，则应该在程序中用关键字 volatile 声明此变量。通知编译器，在编译过程中不要对该变量进行优化。因为编译器在优化时，往往会将引用该变量的拷贝。而信号处理函数中的全局变量，往往是异步修改的，其值随时可能被信号处理函数更改。通过 volatile 关键字声明，使用编译器在引用该变量时，直接从该变量的内在地址中引用。
- 如果在信号处理函数中调用函数，那么该函数必须是可重入的，或者保证在信号处理函数执行期间不会有信号到达进程。Linux 系统下存在许多不可重入的函数，如分配内存的 malloc、获取机器名的 gethostbyname 等。

在信号处理函数里，有时需要用到长跳转的操作。所谓长跳转，就是从信号处理函数直接跳转到函数体外指定的代码位置继续运行。很多情况下，通过长跳转可以减少代码的复杂度。Linux 系统提供了两个函数实现该功能：设置跳转点的 sigsetjmp 和执行跳转的 siglongjmp。sigsetjmp 用来设置跳转点。在成功调用 sigsetjmp 后，sigsetjmp 语句所在的位置就是跳转点，

这个位置指针将被保存到 sigsetjmp 的第一个参数中。siglongjmp 用于跳转到指定的跳转点。这两个函数位于头文件<setjmp.h>中，其原型如下所示。

```
#include <setjmp.h>
int sigsetjmp (struct __jmp_buf_tag __env[1],int   __savemask);
void siglongjmp (sigjmp_buf __env, int __val);
```

（1）参数说明如下。

- __env：输出参数，该参数实际上是一个结构数据。结构中包含了长跳转指针，是否保存信号掩码及保存的信号掩码值等信息。对于应用编程人员来说，该结构是透明的。
- __savemask：输入参数，是否保存信号掩码。如果该参数非零，则在调用 sigsetjmp 后，当前进程的信号掩码将被保存。在调用 sigsetjmp 时，将恢复以前保存的信号掩码。

（2）返回值说明如下。

- 0：表明 sigsetjmp 在本进程中是首次调用。
- 1：表明当前的 sigsetjmp 调用是由 siglongjmp 跳转后的非首次调用。

【范例 14-6】编程实现捕捉 SIGINT 信号，在信号处理函数中用长跳转跳至主程序。其实现过程如示例代码 14-6 所示。

示例代码 14-6

```
#include <stdio.h>                                    /*头文件*/
#include <signal.h>
#include <setjmp.h>
static sigjmp_buf jmpbuf;                             /*静态全局变量，用于保存跳转点*/
static volatile int IsOK;     /*静态 volatile 变量，用于在信号处理函数中确定已调用了 sigsetjmp*/
void handle_sigint(int signo)
{
      if(!IsOK) return;       /*如果在未调用 sigsetjmp 的情况下收到信号，则不做任何处理返回*/
      printf("receive signal %d.\n",signo);           /*输出信号的相关信息*/
      IsOK = 0;               /*复位 IsOK，以确保下一次收到信号时，主函数已调用了 sigsetjmp*/
      siglongjmp(jmpbuf,1);                           /*长跳转至 jmpbuf */
}
main()
{
    int n;
    signal(SIGINT,handle_sigint);                     /*安装 SIGINT 信号*/
    n = sigsetjmp(jmpbuf,1);                          /*保存跳转点*/
    if(n == 0)                                        /*第一次调用 sigsetjmp 时将返回 0*/
        printf("first call sigsetjmp\n");             /*输出提示信息*/
    else                                  /*从信号处理函数跳转过来时，sigstejmp 将返回非 0 值*/
    {
        printf("call sigsetjmp after siglongjmp\n");  /*输出提示信息后返回*/
        return;
    }
    IsOK = 1;                             /*设置 IsOK 标志为真，以便于信号处理函数进行跳转*/
    for(;;)                               /*循环调用 pause 阻塞进程，直到某个信号到来*/
        pause();
}
```

【运行结果】经过编译链接，在 shell 下运行上述程序，其结果如下所示。

```
1    first call sigsetjmp
2    receive signal 2.
3    call sigsetjmp after siglongjmp
```

【代码解析】在本例中，首先通过调用 signal 安装 SIGINT 信号，在该信号处理函数内实现了长跳转，然后调用 sigsetjmp 设置长跳转点，最后循环调用 pause 等待信号的到来。本例中，在捕获到 SIGINT 信号后，在 SIGINT 的信号处理函数中长跳转到主函数内的跳转点。在主函数中判断 sigsetjmp 的返回，如果返回值为 0，表明是首次调用 sigsetjmp；否则，表明本次调用是经过 siglongjmp 跳转后的调用。本例中，源代码的各行解释如下。

- 第 1～3 行：头文件信息。
- 第 4 行：静态全局变量，用于保存跳转点信息。
- 第 5 行：静态变量，用于标志跳转点设置操作是否结束。只有在成功设置跳转点后，siglongjmp 才可以进行。在信号处理函数中可以见到，如果该标志为 0，则表示尚未设置跳转点，不进行长跳转直接返回。
- 第 6～12 行：SIGINT 的信号处理函数。在函数内部首先通过变量 IsOK 判断是否设置了跳转点。如果尚未设置，则直接返回；否则，长跳转至主函数内的跳转点。
- 第 16 行：调用 signal 安装信号 SIGINT。
- 第 17 行：调用 sigsetjmp 设置跳转点。
- 第 18～24 行：根据 sigsetjmp 返回值判断是否是首次调用 sigsetjmp。如果是首次，则继续执行；否则退出程序。
- 第 25 行：设置标志 IsOK 为 1。本语句执行成功后，进程如果收到 SIGINT 信号，将进行长跳转。
- 第 26～27 行：循环调用 pause 等待信号。

14.3 信号的发送

信号的来源可以多种多样，可能是由硬件产生的信号，也可能是由其他用户进程发送而来的信号。本节将介绍如何向进程发送信号。

14.3.1 使用 kill 发送信号

使用 kill 系统可以实现信号发送。kill 不仅可以向某个进程发送信号，而且可以实现向一组进程发送信号。kill 的声明位于头文件<sygnal.h>中，其原型如下所示。

```
#include <signal.h>
int kill(__pid_t __pid,int __sig);
```

参数说明如下。

- __pid：输入参数，进程 ID。kill 通过对进程 ID 赋予不同的值，实现多种信号发送方式。
- 0：发送信号成功。

❑ 1：发送失败，可以通过查看 errno 获取详细的错误信息。

提示：除了 kill 外，Linux 系统还提供了 raise 函数用于发送信号。不过，raise 只能用于向进程自身发送信号。raise 函数的功能可以通过 kill（getpid(),SIGXXX）替代实现。

14.3.2　使用 sigqueue 发送信号

通过 kill 可以实现信号的发送。有些情况下需要在发送信号的同时，需要附加其他数据。这种情况下 kill 无法满足要求，此时需要用到另外一个信号发送的系统调用 sigqueue。sigqueue 不仅可以实现 kill 的全部功能，而且还可以实现支持附加数据发送信号。因此，在程序设计过程中，建议读者使用 sigqueue 代替对 kill 的调用。sigqueue 的声明位于头文件<signal>中，其原型如下所示。

```
#include <signal.h>
int sigqueue(__pid_t __pid,int __sig,__const  union sigval __val);
```

（1）参数说明如下。

❑ __pid：输入参数，发送信号的目的进程 ID。与 kill 不同，通过 sigqueue 只能向一个进程发送。

❑ __sig：输入参数，要发送的信号名称。

❑ __val：输入参数，信号的附加数据。

该参数是一个联合，其结构如下所示。

```
typedef union sigval
{
int sigval_int;
void *sival_ptr;
}sigval_t;
```

在调用 sigqueue 时，该参数将会复制到信号处理函数的第二个参数中。在信号处理函数中可以获取到附加数据并进行处理。

（2）返回值说明如下。

❑ 0：发送成功。

❑ −1：发送失败，可以查看 errno 获取详细错误信息。

【范例 14-7】编程实现捕捉 SIGINT 信号，在信号处理函数中用长跳转跳至主程序。其实现过程如示例代码 14-7 所示。

示例代码 14-7

```
1   #include <stdio.h>                                          /*头文件*/
2   #include <signal.h>
3   void HandleSigint(int signo,siginfo_t *info,void *none)     /*SIGINT 信号处理函数*/
4   {
5       printf("receive addtional data is %d\n",info->si_value.sival_int);/*输出收到信号时的附加数据*/
6       exit(0);
7   }
8   main()                                                      /*主函数*/
9   {
```

```
10      int pid;                                        /*整型变量的定义*/
11      struct sigaction act;                           /*信号安装时需要用到的结构变量*/
12      union sigval sigvalPara;                        /*发送信号的附加数据时需要用到的联合变量*/
13      if((pid = fork()) == 0)                         /*创建新的子进程，下述为子进程执行模块*/
14      {
15          memset(&act,0x00,sizeof(struct sigaction));     /*清空结构变量*/
16          sigemptyset(&act.sa_mask);                      /*清除信号处理掩码*/
17          act.sa_sigaction = HandleSigint;                /*设置信号处理函数*/
18          act.sa_flags = SA_SIGINFO;                      /*设置发送信号时可以附加数据*/
19          if(sigaction(SIGINT,&act,NULL) == -1)           /*安装信号*/
20          {
21              perror("sigaction");                        /*错误处理*/
22              exit(0);
23          }
24          pause();                                        /*暂停子进程的执行，等待信号*/
25      }
26      else                                                /*父进程执行模块*/
27      {
28          sigvalPara.sival_int = 123;                     /*设置附加数据为 123*/
29          if(sigqueue(pid,SIGINT,sigvalPara) == -1)   /*向子进程发送信号 SIGINT，并附加数据*/
30          {
31              perror("sigqueue");                         /*错误处理*/
32              exit(0);
33          }
34      }
35  }
```

【运行结果】经过编译链接，在 shell 下运行上述程序，其结果如下所示。

```
1   receive addtional data is 123
```

【代码解析】在本例中，创建了新的子进程。子进程安装了 SIGINT 信号，父进程向子进程发送 SIGINT 信号。子进程收到 SIGINT 信号后，输出其附加数据信息。子进程的代码解释见示例代码 14-2 的解释。本例中，源代码的各行解释如下。

- 第 28 行：为联合体 sigvalPara 赋整型值 123。该值将随同信号一并发送至子进程。
- 第 29～32 行：调用 sigqueue 向子进程发送 SIGINT 信号。

14.4 SIGALRM 信号

在进行阻塞式系统调用时，为避免进程陷入无限期的等待，可以为这些阻塞式系统调用设置定时器。本节将要介绍 alarm 调用及 SIGALRM 信号的处理。

14.4.1 安装 SIGALRM 信号

要使用定时器，首先要安装 SIGALRM 信号。如果不安装 SIGALRM 信号，则进程收到 SIGALRM 信号后，缺省的动作是终止当前进程。因此，在实际应用中往往需要安装该信号。安装 SIGALRM 信号的过程与其他信号的安装完全相同。可以通过 signal 进行安装，也可以通过 sigqueue 安装。该信号一旦被安装，进程收到 SIGALRM 信号时，将触发信号处理函数。

SIGALRM 的信号处理函数应该做的操作往往是设置超时标志。由外部的程序去判断该超时标志，并输出相应的错误提示信息。

14.4.2　设置定时器

SIGALRM 信号安装成功后，在什么情况下进程会收到该信号呢？这就要依赖于 Linux 提供的定时器功能。在 Linux 系统下，每个进程都有一个惟一的定时器，该定时器提供了以秒为单位的定时功能。在定时器设置的超时时间到达后，调用 alarm 的进程将收到 SIGALRM 信号。设置定时器的系统调用是 alarm，该调用的声明位于头文件<unistd.h>中，其原型如下所示。

```
#include <unistd.h>
unsigned int alarm(unsigned int __seconds);
```

参数__seconds 表示定时器的定时时间，以秒为单位。在 alarm 调用成功后开始计时，超过该时间将触发 SIGALRM 信号。返回值为当前进程以前设置的定时器剩余秒数。

【范例 14-8】在程序中实现定时器。其实现过程如示例代码 14-8 所示。

示例代码 14-8

```
#include <stdio.h>                                  /*头文件*/
#include <signal.h>
int timeout;                                        /*全局变量，用于设置超时标志*/
void Handle_Alarm(int signo)                        /*信号处理函数*/
{
    timeout = 1;                                    /*收到 SIGALRM 信号，设置超时标志为 1*/
    printf("SIGALRM received.\n");                  /*输出提示信息*/
}
main()
{
    if(signal(SIGALRM,Handle_Alarm) ==SIG_ERR )     /*安装 SIGALRM 信号*/
    {
        perror("signal");                           /*输出错误信息*/
        exit(0);
    }
    timeout = 0;                                    /*设置超时标志为 0*/
    alarm(10);                                      /*启动定时器*/
    pause();                                        /*阻塞进程，等待信号*/
    if(timeout)                                     /*判断是否定时器超时，如果超时，输出提示*/
    {
        printf("Pause time out.\n");
    }
}
```

【运行结果】经过编译链接，在 shell 下运行上述程序，其结果如下所示。

```
SIGALRM received.
Pause time out.
```

【代码解析】在本例中，首先调用 signal 安装 SIGALRM 信号，然后调用 pause 等待信号到来。如果在定时器的超时时间内未收到任何信号，则 pause 将被 SIGALRM 信号中断。

本例中，源代码的各行解释如下。

- 第 1～2 行：头文件信息。
- 第 3 行：全局变量，用于标志是否超时。
- 第 4～8 行：SIGALRM 信号的处理函数。在函数中，只是简单地设置超时标志为 1 并输出相应信息后返回。
- 第 11～15 行：调用 signal 安装 SIGALRM 信号。
- 第 16 行：复位超时标志。
- 第 17 行：启动定时器，超时时间为 10 秒。
- 第 18 行：调用 pause 等待信号到达。
- 第 19 行：pause 返回后，判断超时标志，如果超时则提示相应信息。

提示：使用 alarm 设置定时器，可以实现秒级的超时控制。如果需要实现更细粒度的超时控制，建议使用 select 系统调用。通过 select 系统调用，可以实现微秒级的超时控制。

14.5 SIGCLD 信号

在 Linux 的多进程编程中，SIGCLD 是一个非常重要的信号。在子进程退出时，将向其父进程发送该信号。在编程过程中，可以捕获该信号，并获取子进程的退出状态。

14.5.1 子进程的退出过程

当一个子进程退出时，并不是立即释放其占用的资源，而是通知其父进程，由父进程进行后续的工作。在这一过程中，系统将依次产生下列事件。

（1）向父进程发送 SIGCLD 信号，子进程进入 zombie（也称为僵尸）状态。

（2）父进程接收到 SIGCLD 信号，进行处理。

如果在上述过程中，父进程既没有忽略 SIGCLD 信号，也未捕获信号进行处理，则该子进程将进入僵尸状态。僵尸状态的进程不能被操作系统调用，也没有任何可执行代码，它不过是占用了进程列表中的一个位置。如果仅有几个僵尸进程不会影响系统的运行，但是如果僵尸进程过多，进程列表空间被过多占用，则将严重影响系统的运行。因此，在编程过程中应避免产生僵尸进程。

【范例 14-9】编程实现僵尸进程。其实现过程如示例代码 14-9 所示。

示例代码 14-9

```
1   #include <stdio.h>                          /*头文件*/
2   #include <signal.h>
3   main()                                      /*主函数*/
4   {
5       int i,pid;                              /*定义整型变量*/
6       if((pid = fork()) == 0)                 /*创建子进程*/
7       {
```

```
8           exit(0);/*子进程立即退出，向父进程发送 SIGCLD 信号。注意：父进程未安装 SIGCLD 信号*/
9       }
10      else
11      {
12          for(i=0;i<10;i++)                    /*循环等待,用 ps 可以看到僵尸进程*/
13              sleep(5);                        /*休眠 5 秒*/
14      }
15  }
```

【运行结果】经过编译链接，在 shell 下运行上述程序。在另外的 shell 中运行 ps –alef|grep defunct 命令的输出，如下所示。

```
1   1 Z develop    2755   2754   0   76    0 -        0 exit    02:38 pts/2    00:00:00
    [example14_9] <defunct>
```

【代码解析】本例目的仅为示例僵尸进程，在编程过程中应避免出现该情况。在本例中，并未安装 SIGCLD 信号。首先调用 fork 创建子进程，在子进程中未做任何操作立即退出，在父进程中循环调用 sleep。通过 ps 的输出可以看到，子进程状态为 Z，已成为僵尸进程。本例中，源代码的各行解释如下。

- 第 6～9 行：子进程代码。本例中无任何操作，立即退出。
- 第 11～14 行：父进程代码。循环调用 sleep，以产生僵尸进程。

提示：通过执行 ps 命令可以查看到僵尸进程。ps 的命令输出中，状态为 Z 的进程即为僵尸进程。另外，在进程名称后面会附加 defunct 标志。

14.5.2　SIGCLD 信号的处理

通过上一节的介绍，读者应该了解僵尸进程是影响系统运行的。那么，在编程中如何避免出现僵尸进程呢？基本的处理方法有两种：一是父进程忽略 SIGCLD 信号；二是父进程捕获 SIGCLD 信号，在 SIGCLD 信号处理函数中获取子进程的退出状态。忽略信号的方式比较简单，只需要调用 signal（SIGCLD,SIG_IGN）语句即可完成。本节将主要介绍捕获 SIGCLD 信号的处理。

要捕获 SIGCLD 信号，首先要安装 SIGCLD 信号。在 SIGCLD 的信号处理函数中，应该调用 wait 或者 waitpid 等函数获取子进程的退出状态。详细的实现过程参见范例 10-10。

【范例 14-10】编程捕获 SIGCLD 信号，输出各子进程的退出状态。其实现过程如示例代码 14-10 所示。

示例代码 14-10

```
1   #include <stdio.h>                          /*头文件*/
2   #include <signal.h>
3   void Handle_Sigcld(int signo)               /*SIGCLD 信号处理函数*/
4   {
5       int pid,status;                         /*退出进程的进程 ID 退出进程的退出状态*/
6       pid = waitpid(-1,&status,0);            /*等待任何一个子进程退出*/
7       printf("Child process %d exit with code %d\n",pid,status);/*输出退出的子进程 ID 和退出代码*/
8   }
9   main()                                      /*主函数*/
10  {
```

```
11      int i,pid;                              /*整型变量*/
12      signal(SIGCLD,Handle_Sigcld);           /*安装 SIGCLD 信号*/
13      for(i=0;i<5;i++)                        /*循环创建子进程*/
14      {
15          if((pid = fork()) == 0)             /*创建子进程*/
16          {
17              srand(getpid());                /*产生随机数*/
18              exit((int)(rand()/1024));       /*退出子进程，退出码为上步产生的随机数*/
19          }
20          else
21          {
22              sleep(1);                       /*父进程，休眠一秒后继续循环*/
23              continue;
24          }
25      }
26  }
```

【运行结果】经过编译链接，在 shell 下运行上述程序输出，如下所示。

```
1   Child process 2995 exit with code 6656
2   Child process 2996 exit with code 58368
3   Child process 2997 exit with code 59392
4   Child process 2998 exit with code 32768
5   Child process 2999 exit with code 35840
```

【代码解析】在本例中，首先安装 SIGCLD 信号。在信号处理函数中，调用 waitpid 获取子进程的退出代码并输出。然后循环调用 fork 创建子进程，子进程调用 rand 生成随机数作为子进程的退出代码。本例中，源代码的各行解释如下。

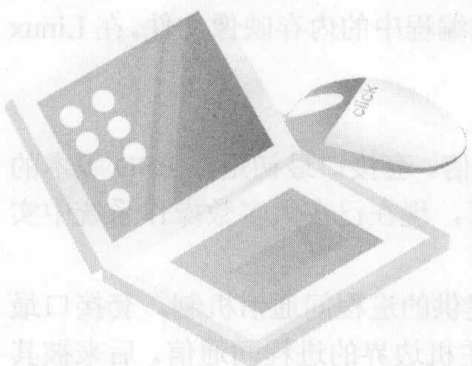
- 第 1～2 行：头文件。
- 第 3～8 行：SIGCLD 信号处理函数。在函数中，调用 waitpid 获取子进程的退出状态并输出。
- 第 12 行：安装 SIGCLD 信号。
- 第 13～25 行：for 循环创建子进程。
- 第 17～18 行：子进程代码。调用 srand 生成随机数种子，然后调用 rand 生成随机数。
- 第 22～23 行：父进程代码。循环调用 sleep，以产生子进程。

14.6 小结

信号处理是 Linux 编程中非常重要的内容。本章介绍了 Linux 系统下信号机制的原理，阐述了信号的产生机制及处理方法。其中，如何处理信号是 Linux 编程的核心内容。本章将两个比较重要的信号 SIGALRM 和 SIGCLD 进行了单独介绍，讲解如何安装和处理这两种信号。鉴于信号机制在 Linux 编程中的重要性，读者应加强对 Linux 信号机制的理解，并结合编程实践认真体会。

第四篇　进程间通信(IPC)及网络编程

第 15 章　进程间通信——管道
第 16 章　进程间通信——消息队列
第 17 章　进程间通信——共享内存
第 18 章　进程间通信——信号量
第 19 章　Linux 网络环境
第 20 章　基本套接口编程
第 21 章　综合实例——银行代理收费服务器



第 15 章　进程间通信——管道

管道是一种古老的进程间通信机制。在本章中，首先向读者介绍管道的基本概念及其分类，然后介绍了管道的编程模型。最后结合具体实例讲解了管道编程的方法。通过学习本章，读者应重点掌握如下内容。

- 了解进程间通信的概念及机制。
- 理解管道的概念、管道的类型，特别注意区分不同类型的管道间的区别。
- 掌握普通管道及命名管道的编程方法，对管道 I/O 编程应加深理解。

15.1　进程间通信概念

进程间通信是在不同进程间进行信息交换的机制。一个复杂的应用系统，往往需要多个进程协同完成。这就涉及多个进程间的通信问题，即进程间通信。Linux 进程间通信机制主要包括管道、信号、消息队列、共享共存、信号量、套接口。其中，套接口实现了跨越主机边界的通信，主要用于网络环境下的程序设计。

- 管道：管道可以划分为普通管道及命名管道。普通管道用于具有亲缘关系的进程（父子进程）间的通信。命名管道则在普通管道提供的功能基础上，通过给管道命名的方法，使管道变成文件系统中的管道文件，从而允许无亲缘关系进程间通过访问管道文件进行通信。
- 信号：信号类似于 Windows 系统下的消息，用于通知进程有某种事件发生。只要知道进程的进程号，就可以向进程发送信号。而进程可以自行定义对信息的处理方法。
- 消息队列：消息队列是消息的链表。进程可以向消息队列中发送某种类型的消息，也可以从消息队列中读取指定类型的消息。消息内容则可以根据需要自行定义，从而使消息队列克服了信号承载信息量过少的缺点，在实际编程中应用较广。
- 共享内存：共享内存是在系统内核分配的一块缓冲区，多个进程都可以访问该缓冲区。由于进程可以直接读写内存，避免了在内核空间与用户空间的切换，所以共享内存读写效率很高。共享内存类似于 Windows 环境编程中的内存映像文件。在 Linux 环境中，多用于存储应用程序的配置信息。
- 信号量：也称为信号灯，主要用于进程间的同步。
- 套接口：也称为套接字，可用于跨越主机边界的通信。套接口最初是在 BSD 版本的 UNIX 系统上实现的，由于其功能强大，接口友好，现在已在大多数操作系统中实现，成为事实上的网络编程接口。

在上述进程间通信机制中，管道和信号是系统早期就提供的进程间通信机制。套接口最初是由 BSD 版本的 UNIX 操作系统实现的，主要用于跨越主机边界的进程间通信。后来被其

他版本的 UNIX 系统及其他操作系统（包括 Linux）广泛支持。

消息队列、共享内存、信号量 3 种机制合并起来称为进程间通信，简称为 IPC（Inter Process Communication）。注意：这里的进程间通信是一个狭义的概念，只包括这 3 种机制。根据实现方式的不同，IPC 又可以分为 System V 进程间通信和 POSIX（可移植性操作系统界面）进程间通信。

Linux 操作系统完整地继承了 UNIX 的进程间通信机制，如图 15-1 所示。

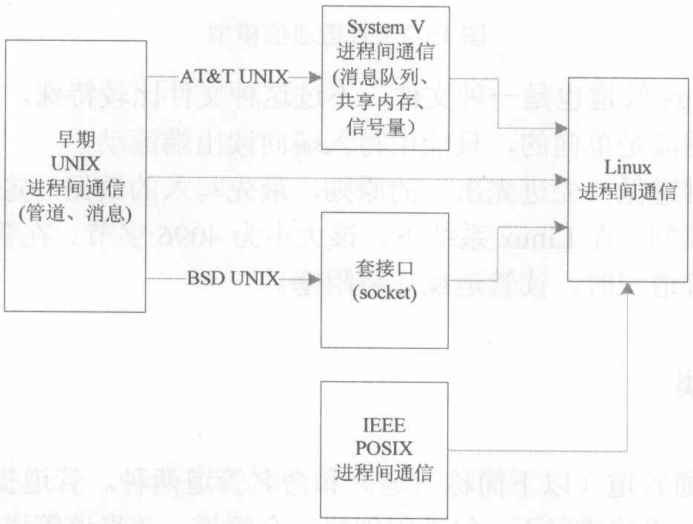


图 15-1　Linux 进程间通信

本章将首先介绍管道的概念及其编程。在第 16～18 章中将分别介绍 System V 的消息队列、共享内存和信号量。在第 20 章中向读者介绍套接口编程，即网络编程。

15.2　管道的概念及分类

管道是进程间通信机制中最古老的方式，从较早版本的 UNIX 系统一直到目前的 Linux 操作系统都得到较好的支持。了解管道的概念有助于更好的理解输入、输出重定向等 Linux 操作系统中重要的概念。

15.2.1　管道的概念及特点

管道，顾名思义就是一个连接两个进程的连接器。管道是单向的，一端只能用于输入，另一端只能用于输出。管道的输出遵循“先进先出（FIFO：First In First Out）”的原则。向管道中写入的数据附加至管道缓冲区的尾部，而读取数据则是从管道缓冲区的头部开始。另外，需要注意的是，普通管道只能用于有亲缘关系（父子进程）的进程间进行通信。

以统计当前目录下文件数量为例，该命令的语法为“ls|wc -l”。中间的“|”在 Linux 的 shell 中就是管道符号。这条命令的含义是 ls 的输出通过管道作为 wc –l 命令的输入。如此一

来，ls 的输出不再在输出设备上显示，而是作为 wc 命令的输入。在该命令中管道的通信模型如图 15-2 所示。



图 15-2　管道通信模型

从本质上看，Linux 管道也是一种文件。不过这种文件比较特殊，具有以下特点。

- 管道中的数据流是单向的，只能由写入端向读出端流动。
- 管道中的数据遵循“先进先出”的原则，最先写入的数据一定被最先读出。
- 管道有大小限制。在 Linux 系统下，该大小为 4096 字节。在管道满时，写管道操作将阻塞；在管道空时，读管道操作将阻塞。

15.2.2　管道的分类

管道可以分为普通管道（以下简称管道）和命名管道两种。管道提供了进程与其子进程进行通信的有效手段。通常情况下，父进程创建一个管道，并将该管道传递给子进程。这样，通过管道父进程与子进程便可以进行通信。命名管道在功能上与管道是相同的，只不过命名管道位于文件系统中，只要可以访问文件系统的进程都可以通过管道的名字进行访问。所以，命名管道可以实现无任何亲缘关系的两个进程间的通信处理。管道与命名管道存在如下不同，如图 15-3 所示。

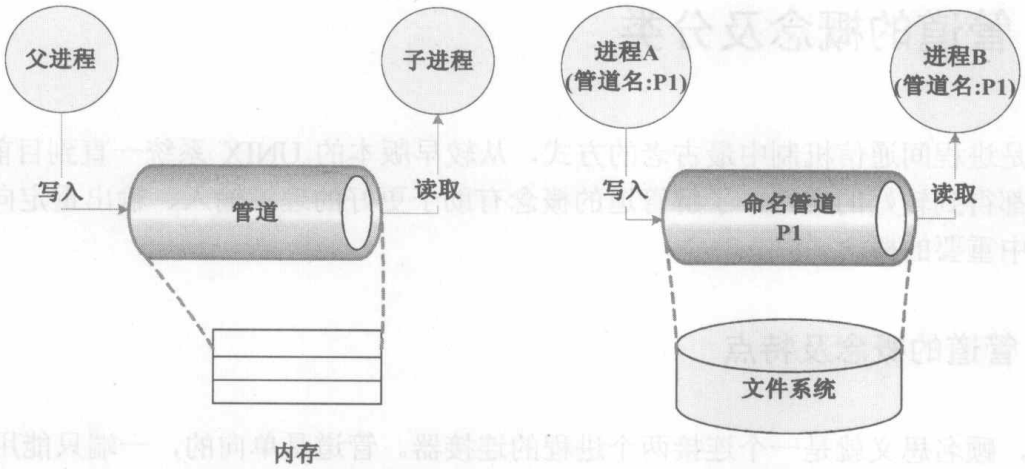


图 15-3　管道与命名管道

从上图可以发现，管道和命名管道不同之处如下。

- 普通管道位于内存，只能用于有亲缘关系的进程间通信。
- 命名管道位于文件系统中，可以实现不同进程间的通信。

15.3　管道编程

通过管道，具有亲缘关系的进程间可以进行通信。通常情况下，具有亲缘关系的父子进程，一个进程向管道写，另一个进程从管道读。从本质上看，管道是一对文件描述符的组合。因此，基本上对文件的操作都可以用在管道上。主要的管道操作包括创建管道、读写管道、关闭管道等。

15.3.1　创建管道

管道要创建后方可使用其进行数据交换。创建管道的函数是 pipe。该函数以一个两个成员的整型数据为参数。调用成功后，数组的第一个成员是用于读的文件描述符，而数组的第二个成员是用于写的文件描述符。pipe 的声明位于头文件<unistd.h>中，其原型如下所示。

```
#include <unistd.h>
int pipe (int __pipedes[2]);
```

（1）参数说明如下。

- __pipedes：整型数组。在函数执行后，该数组中将保存系统返回的管道描述符。其中__pipedes[0]用于读操作，而__pipedes[1]用于写操作。

（2）返回值说明如下。

- 0：表示函数执行成功。
- -1：表示执行失败。可以查询 errno 获取错误详细信息。

（3）使用该函数可能的错误原因包括以下几点。

- EFAULT：参数__pipedes 无效。
- EMFILE：超过进程可打开的最大文件数量。
- ENFILE：已达到系统对打开文件数量的总限制。

该函数通常情况下的应用场合为：一个进程通过 pipe 创建管道后，再 fork 一个子进程，然后通过管道实现父子进程间的通信。

【范例 15-1】创建一个管道，将命令行参数通过管道传输至新派生的子进程。其实现过程如示例代码 15-1 所示。

示例代码 15-1

```
1   #include <stdio.h>                          /*头文件*/
2   #include <stdlib.h>
3   #include <unistd.h>
4   #include <string.h>
5   #include <sys/wait.h>
6   int main(int argc, char *argv[])            /*主函数*/
7   {
8       int pfd[2];                             /*整型数组，用于保存打开管道后的两个文件描述符*/
9       pid_t cpid;                             /*用于保存进程标识符*/
10      char buf;                               /*定义 char 变量*/
```

```
11      if(argc != 2)                                   /*判断命令行参数是否符合要求*/
12      {
13         fprintf(stderr,"Usage: %s <string>\n",argv[0]); /*提示用法信息*/
14         exit(0);
15      }
16      if (pipe(pfd) == -1)                            /*创建管道*/
17      {
18          perror("pipe");                             /*错误处理*/
19          exit(EXIT_FAILURE);
20      }
21      cpid = fork();                                  /*创建进程*/
22      if (cpid == -1)                                 /*创建进程失败的处理*/
23      {
24          perror("fork");                             /*输出错误信息*/
25          exit(-1);
26      }
27      if (cpid == 0)                                  /*在子进程中*/
28      {
29          close(pfd[1]);                              /*关闭写管道描述符,实际上是将引用计数减 1*/
30          while (read(pfd[0], &buf, 1) > 0)                   /*从管道中循环读取数据*/
31              write(STDOUT_FILENO, &buf, 1);          /*输出读到的数据*/
32          write(STDOUT_FILENO, "\n", 1);/*输出从管道中读取的数据*/
33          close(pfd[0]);                              /*关闭管道的读描述符,实际上是将引用计数减 1*/
34          exit(0);                                    /*退出子进程*/
35      }
36      else
37      {
38          close(pfd[0]);                              /*关闭读管道描述符,实际上是将引用计数减 1*/
39          write(pfd[1], argv[1], strlen(argv[1]));            /*向管道中写入命令行参数 1*/
40          close(pfd[1]);                              /*关闭写管道描述符,实际上是将引用计数减 1*/
41          wait(NULL);                                 /*等待子进程退出*/
42          exit(0);
43      }
44  }
```

【运行结果】经过编译链接，在 shell 下运行上述程序，其结果如下所示。

```
1   ./example15_1 abc
2   abc
```

【代码解析】在本例中，首先创建管道，然后创建子进程。利用该管道从父进程向子进程发送信息。在本例中，消息是该程序的命令行参数。子进程收到由父进程传送的消息后输出。从本例的输出结果来看，子进程输出了命令行参数 1 的内容：abc。本例中，源代码的各行解释如下。

- 第 1～5 行：头文件信息。
- 第 11～15 行：判断命令行参数是否存在，不存在则提示本程序使用方法并退出。
- 第 16～20 行：调用 pipe 创建管道。
- 第 21～26 行：调用 fork 创建子进程。fork 成功后，如果返回值为 0，则表示当前位于子进程中，如果非 0，则表示位于父进程中。如果 fork 返回值为-1。表明创建子进程失败。
- 第 27～35 行：子进程处理模块。该模块中的语句将在子进程的环境下执行。注意

根据 fork 的返回值判断父子进程的规则。

- 第 29 行：关闭写管道描述符。此处的关闭并不是真正意义上的关闭。管道描述符与文件描述一样，系统内核在系统中保存着该描述符的引用计数。由于调用 fork 创建了子进程，所以每一个描述符的引用计数都将加 1。由此可见，这里的关闭只是将写管道描述符的引用计数减 1，使其引用计数变为 1。
- 第 30～32 行：循环从读管道描述符中读取数据，并输出至标准输出（缺省为计算机屏幕）。
- 第 33 行：关闭读管道描述符。执行成功后，读管道描述符的引用计数将为 1。
- 第 34 行：调用 exit 退出子进程。
- 第 38～42 行：父进程执行模块。在该模块中的语句将在父进程的环境中运行。
- 第 38 行：读管道描述符引用计数减 1。
- 第 39 行：向写管道描述符写入数据。本例中，数据是指从命令行参数 1 传递的字符串。
- 第 40 行：关闭写管道描述符。
- 第 41 行：调用 wait 等待子进程结束。如果不等待子进程退出而父进程直接退出，则子进程将形成僵尸进程，影响系统效率。
- 第 42 行：父进程退出。结束程序的执行。

提示：本例中每个管道描述符实际上都关闭了两次。这是由于在多进程的环境下，一个有效的管道描述符在并发子进程时将引用计数加 1。这里的每一次只是将其引用计数减 1。在一个描述符的引用计数变为 0 的情况下，系统才会真正关闭一个描述符。

15.3.2　读写管道

管道创建成功后，分别返回一个读管道描述符和一个写管道描述符。可以用文件操作的相关函数对描述符进行读写，如调用 read 从读描述符读取数据，调用 write 向写描述符写入数据。进行读写过程中，应注意以下几点。

- 如果管道的写描述符关闭，则读函数返回的读出字节数为 0。
- 如果管道的读描述符关闭，则写入操作将是无意义的。写操作将返回错误 SIGPIPE。
- 向管道中写入时，如果管道已满，则写操作将阻塞直至管道有可用空间。

【范例 15-2】利用管道编程实现 shell 的如下功能：ls –l|wc –l，即统计当前目录下文件的数量。其实现过程如示例代码 15-2 所示。

示例代码 15-2

```
1    #include <stdio.h>                    /*头文件*/
2    #include <stdlib.h>
3    #include <unistd.h>
4    #include <string.h>
5    #include <sys/wait.h>
6    int main()                            /*主函数*/
7    {
```

```
8   int pfds[2];                              /*整型数组，用于保存创建管道后的读写文件描述符*/
9   if ( pipe(pfds) == 0 )                    /*创建管道*/
10  {
11      if ( fork() == 0 )                    /*创建新进程*/
12      {
13          close(1);                         /*关闭标准输出*/
14          dup2( pfds[1], 1 );               /*将管道的写文件描述符复制到进程的标准输出*/
15          close( pfds[0] );                 /*关闭管道的读文件描述符，实际上是将引用计数减 1*/
16          execlp( "ls", "ls", "-1", NULL );          /*执行 ls –l 命令。执行后的输出将写入管道*/
17      }
18      else
19      {
20          close(0);                         /*关闭标准输入*/
21          dup2( pfds[0], 0 );               /*将管道的读文件描述符复制到进程的标准输入*/
22          close( pfds[1] );                 /*关闭管道的写文件描述符，实际上是将引用计数减 1*/
23          execlp( "wc", "wc", "-l", NULL );/*执行 wc –l 命令，将从管道中读取数据作为 wc 命令的输入*/
24      }
25  }
26  return 0;
27  }
```

【运行结果】经过编译链接，在 shell 下运行上述程序，其结果如下所示。

```
28  ./example15_2
29  5
```

【代码解析】在本例中，首先创建管道，然后创建子进程。在子进程中，调用 execlp 执行 ls 命令。利用函数 dup2 将子进程的标准输出重定向至写管道描述符。经过重定向后，本来应该向屏幕输出的 ls 命令的结果输出至管道的写描述符。而在父进程中，调用 execlp 执行 wc 命令。

同样，调用 dup2 将父进程的标准输入重定向至管道的读描述符。这样，父进程执行 wc 命令时，其输入就是从管道中读取的数据。而该数据正好是由子进程执行 ls 命令输出的数据。通过这一机制，巧妙的实现了程序要求的功能。本例中，源代码的各行解释如下。

- 第 1～5 行：头文件信息。
- 第 13～16 行：子进程执行模块。
- 第 13 行：关闭标准输出 STDOUT，这里用 1 表示。在前面章节中介绍过，每个进程缺省打开 3 个文件描述符，分别用于标准输入、标准输出、标准错误输出，其描述符的值为 0、1、2。
- 第 14 行：调用 dup2 将写管道描述符复制至标准输出。该函数执行成功后，子进程的标准输出已变为管道的写描述符。
- 第 15 行：将管道的读描述符的引用计数减 1。关于引用计数减 1 的详细内容请参见范例 15-1 的解释。
- 第 16 行：调用函数 execlp 执行程序 ls –l。execlp 将根据环境变量 PATH 的路径搜索要执行的目标程序 ls。找到后执行该程序，该函数的第 2 个及以后的参数作为要执行的程序的命令行参数。
- 第 20～23 行：父进程执行模块。
- 第 20 行：关闭父进程的标准输入。

- 第 21 行：调用 dup2 将父进程的标准输入重定向至管道的读描述符。
- 第 22 行：将管道的写描述符的引用计数减 1。
- 第 23 行：调用函数 execlp 执行程序 wc -l。注意：执行程序 wc 时，输入的数据将是由子进程通过管道传输来的 ls 命令的输出。

提示：在本例中用到了函数 execlp 加载一个外部程序。execlp 实际上是 exec 函数系列中的一个，其他的还包括 execl、execle、execv、execve 和 execvp。该函数执行时，将用新的程序直接覆盖调用进程的代码段地址空间，数据段及堆栈段也将被重建，惟一保留的只有该进程的进程 ID。

15.3.3　关闭管道

管道在使用完成后，应该调用 close 关闭。由于一个管道包含两个描述符，所以应该分别调用 close 进行关闭。另外，在多进程环境下，要注意管道描述符的引用计数，只有在引用计数为 0 的情况下，描述符才会真正被关闭。

15.3.4　管道 I/O

在应用管道进行编程时，往往需要首先创建一个管道，然后并发一个进程，父子进程通过管道进行数据交换。为更方便地进行编程，Linux 的标准 I/O 库提供了两个专用的 I/O 函数：popen 和 pclose。这两个函数的声明位于头文件<stdio.h>中，其原型如下所示。

```
#include <stdio.h>
FILE *popen (__const char *__command, __const char *__modes);
int pclose (FILE *__stream);
```

（1）参数说明如下。

- __command：要执行的命令串。
- __modes：方式。如果为 r，则返回的文件指针用于读；如果为 w，则返回的文件指针用于写。

（2）返回值说明如下。

- NULL：表示函数执行失败。可以查询 errno 获取错误详细信息。
- 其他：表示执行成功。

从 popen 函数底层的实现来看，其实现的功能是：首先，创建一个管道，然后 fork 一个子进程，在子进程中通过 exec 系列函数加载要执行的程序（即 popen 函数的第一个参数）。popen 调用成功后，将返回一个 FILE *数据结构指针，这是一个标准 I/O 文件指针。对该指针的解释要结合 popen 函数的第二个参数进行。如果调用 popen 时，指定的第二个参数是“r”，则该文件指针可用来读取命令（popen 的第一个参数）的输出；如果指定的第二个参数是“w”，则该文件指针可用来向命令输入数据。其实现原理如图 15-4 所示（以第二个参数是 w 为例）。

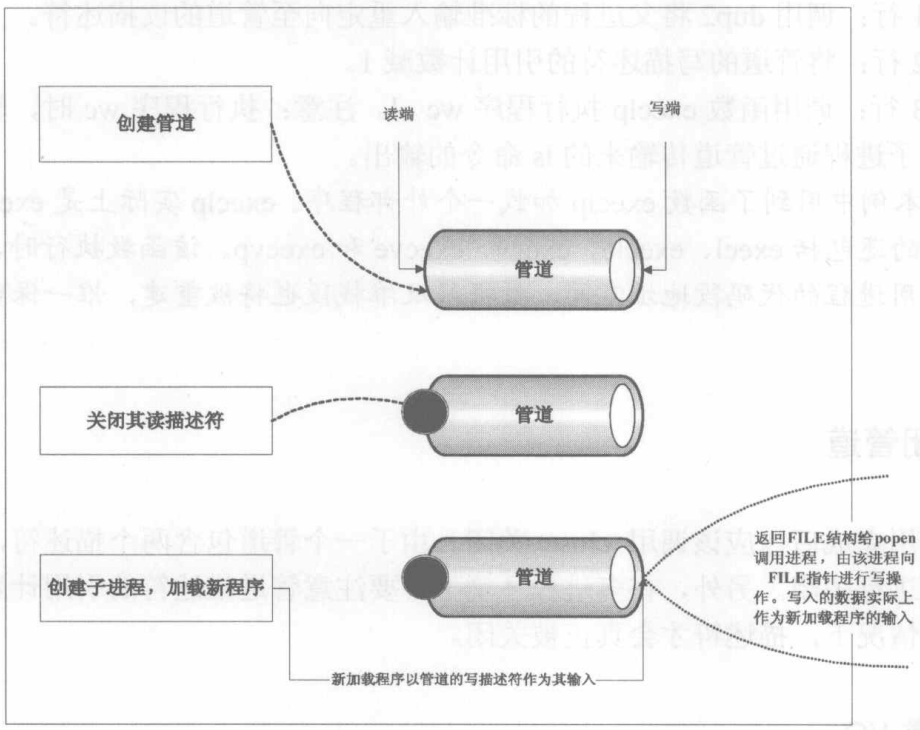


图 15-4　popen 实现原理图

【范例 15-3】通过 popen 编程统计输入数据的行数，输入行内容为 q 则结束。其实现过程如示例代码 15-3 所示。

示例代码 15-3

```
#include <stdlib.h>                                    /*头文件*/
#include <stdio.h>
#include <string.h>
#include <unistd.h>
int main( void )                                       /*主函数*/
{
    FILE    *stream;                                   /*流文件句柄*/
    char    buf[1024];                                 /*文件读写缓冲区*/
    memset( buf, '\0', sizeof(buf) );                  /*清空缓冲区*/
    stream = popen( "wc -l", "w" );   /*调用 popen 执行 wc –l 命令，wc-l 的输入来自 stream */
    for(;;)                           /*循环从键盘输入数据，直到输入某行数据为 q 退出*/
    {
        memset(buf,0x00,sizeof(buf));                  /*清空缓冲区*/
        scanf("%s",buf);                               /*从键盘接收输入的数据*/
        if(strcmp(buf,"q") == 0)      /*判断是否输入了 q，如果是则退出循环*/
        {
            break;                    /*中断循环*/
        }
        fprintf(stream,"%s\n",buf);   /*向 stream 中写入数据，实际上是作为 wc –l 的输入*/
    }
    pclose( stream );                 /*关闭流*/
    return 0;
}
```

【运行结果】经过编译链接，在 shell 下运行上述程序，其结果如下所示。

```
./example15_3
aaa
bbb
q
2
```

【代码解析】在本例中，以 w 方式调用 popen 执行命令 wc –l，即统计输入数据的行数。调用 popen 成功后，返回的 FILE 结构指针用于写。根据对 popen 实现机制的分析可知，向该 FILE 结构写入的数据将作为 wc 命令的输入。由此实现统计输入数据行数的功能。从运行结果看，输入了两行数据：aaa、bbb，统计结果为 2。本例中，源代码的各行解释如下。

- 第 1～4 行：头文件信息。
- 第 10 行：调用 popen 执行 wc 命令。
- 第 11～20 行：循环从键盘输入数据，每行数据为一个单位提交给 FILE 文件指针，输入行内容为 q 则结束。
- 第 21 行：调用 pclose 关闭 FILE 指针。调用该语句完成后，则执行的 wc 命令将输出要统计的行数信息。

15.4　命名管道编程

管道的缺点是只能在有亲缘关系的进程间通信。为解决这一问题，提供了命名管道的概念。命名管道提供了一个文件名与管道关联。在命名管道创建成功后，在文件系统中将产生一个物理的 FIFO 文件。对命名管道的操作，实际将通过该 FIFO 文件实现。通过这一机制，实现了无亲缘关系的进程也可以进行数据交换。与普通管道的操作相比，命名管道多了打开管道及删除管道的操作。

15.4.1　创建管道

在通过编程方式创建管道之前，先了解一下如何通过 shell 命令创建管道。在 Linux 下有两种命令可以创建一个命名管道，分别是 mknod 及 mkfifo。创建成功后，可以通过执行 ls 命令看到新创建的管道文件。使用 mknod 创建管道的语法如下所示。

```
mknod 管道名称 p
```

这里的管道名称即是命名管道的名称，而 p 是向 mknod 指明新创建的节点类型是管道。

使用命令 mkfifo 创建命名管道的语法如下所示。

```
mkfifo –m 权限 管道名称
```

这里的权限是指生成的管道文件的权限信息，如 0644 等。

【范例 15-4】利用命令创建命名管道 p1。其实现过程如示例代码 15-4 所示。

示例代码 15-4

```
6    mkfifo –m 0644 p1 或者 mknod p1 p
```

【运行结果】上述命令执行成功后，可以看到在 shell 下查看管道文件的信息，输出如下所示。

```
prw-r--r--    1 develop users        0 2008-08-29 19:23 p1|
```

【代码解析】本例中，mkfifo 和 mknod 两个命令是等价的。通过前面章节中对文件操作的了解可知，该文件类型为 p，表明这是一个管道文件。文件名为“p1|”，可以看到文件名后跟随一个管道符号“|”。

Linux 操作系统提供的创建管道的函数为 mkfifo。注意：这里的 mkfifo 是一个函数，与上述的 shell 命令是不同的。该函数的声明位于头文件<sys/stat.h>中，其原型如下所示。

```
#include <sys/stat.h>
int mkfifo (__const char *__path, __mode_t __mode);
```

（1）参数说明如下。

- __path：管道名称路径，类似于文件路径。可以使用绝对路径，也可以使用相对路径。
- __mode：新建管道的访问权限。与文件的访问权限相同，如 0644。请参见文件操作相关内容。

（2）返回值说明如下。

- 0：表示函数执行成功。
- -1：表示执行失败。可以查询 errno 获取错误详细信息。

【范例 15-5】通过编程创建命名管道 p1。其实现过程如示例代码 15-5 所示。

示例代码 15-5

```
1    #include <stdio.h>                  /*头文件*/
2    #include <sys/types.h>
3    #include <sys/stat.h>
4    main()                              /*主函数*/
5    {
6        if(mkfifo("p1",0644) < 0)       /*创建命名管道，访问权限为 644*/
7        {
8            perror("mkfifo");           /*错误处理*/
9            exit(-1);
10       }
11       return;
12   }
```

【运行结果】经过编译链接，在 shell 下执行。执行完毕后，查看创建的管道文件，如下所示。

```
prw-r--r--    1 develop users        0 2008-08-29 20:01 p1|
```

【代码解析】本例中，通过函数 mkfifo 在当前目录下创建管道文件 p1。源代码的各行解释如下。

- 第 1～3 行：头文件信息。

❑　第 6～10 行：调用 mkfifo 创建管道文件 p1，权限为 644。

提示：通过函数 mknod 也可以实现创建命名管道的功能。只要在调用 mknod 时指定新创建文件的 mode 信息为 010+“权限信息”即可实现管道的创建，如 mknod(“p1” ,010644,0)。

15.4.2　打开管道及读写

管道创建成功后，要想通过管道进行数据交换，还必须进行管道的打开操作。管道的打开与文件的打开是相同的，都是通过调用 open 进行。其函数原型参见文件操作的相关内容。对管道的打开包括两种方式：阻塞方式与非阻塞方式。在缺省状态下，open 打开是以阻塞方式打开的。如果在 open 时的标志中指定 O_NONBLOCK，则将以非阻塞方式打开。

在非阻塞方式下，对管道的只读打开将立即返回。而如果事先没有进程以只读方式打开管道，则对管道的写方式打开将返回失败，errno 中的错误信息是 ENXIO。

在阻塞方式下，当打开一个管道用于读时，如果没有打开管道用于写的进程，则该打开操作将阻塞，直到有一个进程用写的方式打开该管道为止。同样，当打开一个管道用于写时，如果没有打开管道用于读的进程，则该打开操作也将阻塞，直到有一个进程用读的方式打开该管道为止。如图 15-5 所示展示了阻塞方式下打开管道进行读写的时序图。

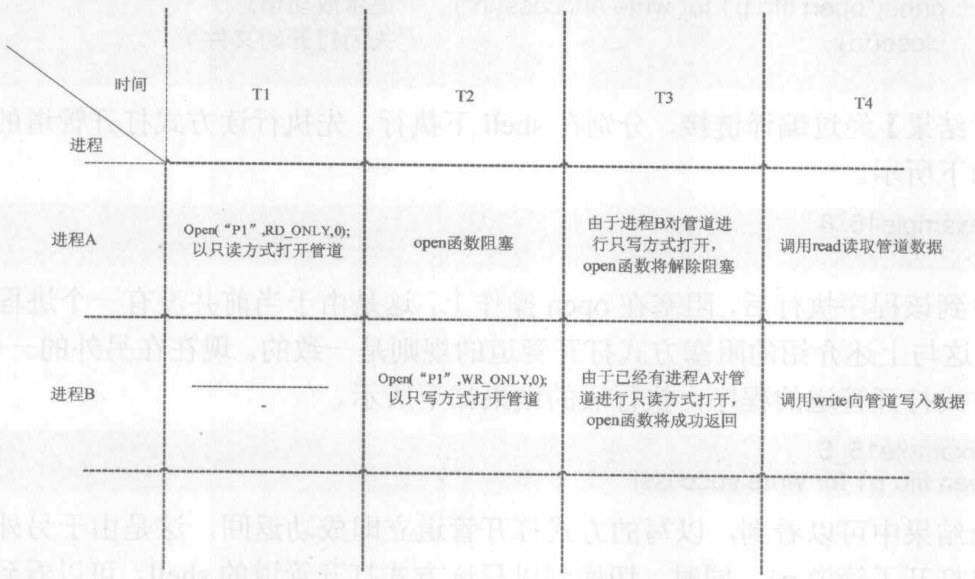


图 15-5　命名管道读写时序

【范例 15-6】通过编程分别以只读、只写的方式打开范例 15-5 中创建的管道 p1。检验阻塞方式打开管道的过程。其实现过程如示例代码 15-6 和示例代码 15-7 所示。

示例代码 15-6

```
1    #include <unistd.h>                          /*头文件*/
2    #include <sys/stat.h>
3    #include <sys/types.h>
4    #include <fcntl.h>
5    main()                                       /*主函数*/
6    {
```

```
7       int fd;                                          /*整型变量，用于保存打开的文件描述符*/
8       if((fd = open("p1",O_RDONLY,0)) < 0)             /*以只读的方式打开管道文件 p1*/
9       {
10          perror("open");                              /*错误处理*/
11          exit(-1);
12      }
13      printf("open fifo p1 for read success!\n");      /*提示成功信息*/
14      close(fd);                                       /*关闭打开的文件*/
15  }
```

示例代码 15-7

```
1   #include <unistd.h>                                  /*头文件*/
2   #include <sys/stat.h>
3   #include <sys/types.h>
4   #include <fcntl.h>
5   main()                                               /*主函数*/
6   {
7       int fd;                                          /*整型变量，用于保存打开的文件描述符*/
8       if((fd = open("p1",O_WRONLY,0)) < 0)             /*以只写的方式打开管道文件 p1*/
9       {
10          perror("open");                              /*提示错误信息*/
11          exit(-1);
12      }
13      printf("open fifo p1 for write success!\n");     /*提示成功信息*/
14      close(fd);                                       /*关闭打开的文件*/
15  }
```

【运行结果】经过编译链接，分别在 shell 下执行。先执行读方式打开管道的程序，执行后的结果如下所示。

```
1   ./example15_6
```

可以看到该程序执行后，阻塞在 open 操作上，这是由于当前并没有一个进程以写的方式打开管道。这与上述介绍的阻塞方式打开管道的规则是一致的。现在在另外的一个 shell 环境下执行写方式打开管道的程序，执行后的结果如下所示。

```
1   ./example15_6
2   open fifo p1 for write success!
```

从输出结果中可以看到，以写的方式打开管道立即成功返回，这是由于另外一个进程以只读方式已打开了管道 p1。同时，切换到以只读方式打开管道的 shell，可以看到，刚才被阻塞的只读方式打开管道的程序已正确返回，如下所示。

```
1   ./example15_6
2   open fifo p1 for read success!
```

【代码解析】本例中，示例代码 15-6 与示例代码 15-7 除打开管道时的打开方式不同外，其他都是相同的。源代码的各行解释如下。

- 第 1～4 行：头文件信息。
- 第 8～12 行：分别以只读和只写的方式打开管道。
- 第 13～14 行：输出提示信息并关闭管道描述符。

15.4.3　管道的删除

与普通管道不同，命名管道在创建完成后，将在文件系统产生一个管道类型的文件。如果该管道在其他场合不再使用，则需要删除该管道，这是通过对文件删除操作的 unlink 完成的。关于 unlink 函数的细节，请参见文件操作的相关内容。

15.5　小结

本章主要介绍了管道及命名管道编程的相关内容。管道虽然是一种比较古老的进程间通信机制，但在某些应用场合还是非常有作用的。因此，掌握管道编程的基础知识对于开发者来说十分重要。本章从创建、读写和关闭管道等方面，详细讲解了管道编程的内容。从下一章开始，将向读者介绍其他几种更为强大的进程间通信方式，主要包括 System V 进程间通信机制及套接口编程等。

第 16 章　进程间通信——消息队列

Linux 操作系统的 IPC 机制支持不同的标准，包括 System V 标准和 POSIX 标准。目前大量的应用都使用 System V 标准实现。尽管 POSIX 的 IPC 机制提供了更好的可移植性及效率，但是由于不同版本的 Linux 对 POSIX 的支持不尽相同，而对 System V 的进程间通信提供了良好的支持，因此，本书主要介绍 System V 标准的 IPC 机制。通过学习本章，读者应重点掌握如下内容。

- 掌握 System V 进程间通信的基本机制及相关数据结构。
- 理解消息队列的实现机制。
- 熟悉掌握使用消息队列接口 API 实现消息队列编程。

16.1　System V 进程间通信概述

System V 进程间通信主要包括消息队列、共享内存、信号量，简称 IPC。这 3 种方式的用途各不相同，实现机制则大同小异。IPC 对象可以通过编程接口进行访问控制，同时也可以从 shell 环境中通过命令直接访问。使用这些 IPC 的基本过程包括创建、读写、删除。在使用 IPC 对象前，需要指定一个关键字，并根据该关键字创建一个 IPC 对象。在应用程序的进程内，根据创建 IPC 对象返回的标识符对该 IPC 对象进行访问。

16.1.1　Shell 环境控制 IPC

IPC 对象一经创建，系统内核即为该对象分配相关数据结构。为方便对 IPC 对象的管理，Linux 提供专门的 IPC 控制命令，主要包括查看 IPC 对象信息的 ipcs 和删除 IPC 对象的 ipcrm。

1．查看 IPC 对象信息

```
ipcs [-asmq]
```

（1）参数说明如下。

- -a：查看全部 IPC 对象信息。
- -q：查看消息队列信息。
- -m：查看共享内存信息。
- -s：查看信号量信息。

（2）输出说明如下。

- key：键值，在系统中是全局惟一的，表明该对象的键值。不同的 IPC 机制，其 Key 值是可以重复的。
- ID：标识符，访问该 ipc 的标识符。对于同一键值的 IPC 对象，每重建（删除后再重新创建）一次，标识符都将加 1，到达系统约定的最大值后归 0 重新开始加 1。
- owner：对象属主。
- perms：对象权限。与文件权限类似，以 3 组共 9 位 8 进制数字表示，可在创建对象时指定访问权限。
- nattch：共享内存对象专用，表明有多少个进程对该共享内存进行了映射（shmat）。
- nsems：信号量专用，表明该信号量对象包含多少个信号量（一个信号量对象可以包含多个信号量）。
- used-bytes：消息队列专用，表明该消息队列中存储的数据量（以字节为单位）。
- messages：消息队列专用，表明该消息队列中有多少条消息。

2．删除 IPC 对象

```
ipcrm –[smq] ID 或者 ipcrm –[SMQ] Key
```

参数说明如下。

- -q、-Q：删除消息队列信息。
- -m、-M：删除共享内存信息。
- -s、-S：删除信号量信息。
- 如果指定了 smq，则用 IPC 的标识符（ID）作为输入，如果指定了 SMQ，则用 IPC 对象的键值（key）作为输入。

【范例 16-1】查看系统中的 IPC 对象信息，并删除其中的 ID 为 0x1234567 的消息队列。其实现过程如示例代码 16-1 所示。

示例代码 16-1

```
1    ipcs –a                          /*显示系统中所有的 IPC 对象*/
```

在 shell 下执行上述命令，输出结果如下所示。

```
2    ------ Shared Memory Segments --------
3    key        shmid      owner      perms      bytes      nattch     status
4    0x00000000 458752     root       644        151552     3          dest
5    ------ Semaphore Arrays --------
6    key        semid      owner      perms      nsems
7
8    ------ Message Queues --------
9    key        msqid      owner      perms      used-bytes   messages
10   0x01234567 98307      develop    666        0            0
```

删除消息队列的过程如下所示。

```
1    ipcrm –q 98307
```

【运行结果】执行上述命令后，再次执行 ipcs –a 查看系统中 IPC 对象的结果如下所示。

```
2    ------ Shared Memory Segments --------
```

```
key        shmid      owner      perms      bytes      nattch     status
0x00000000 458752     root       644        151552     3          dest
------ Semaphore Arrays --------
key        semid      owner      perms      nsems
------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
```

【代码解析】在本例中，首先通过执行 ipcs –a 命令查看系统中的 IPC 信息，然后根据消息队列的 ID 0x1234567 将消息队列删除。根据删除后的结果可以看到，消息队列已删除。

16.1.2 进程间通信关键字

与前面章节介绍过的文件一样，为区别不同的进程间通信的对象，Linux 的每个 IPC 对象都有一个名字，称为“键”（key）。这个关键字是全局惟一的，类似于文件系统中的文件名称。通过文件名，多个进程可以读写同一个文件。而通过“键”，不同的进程同样可以访问同一个 IPC 对象。

进程间通信关键字是一个 32 位的长整型数据。在使用过程中，不同的进程直接使用 key 去创建 IPC 对象容易引起混淆，并且不同的应用之间可能会因为使用同一个 key 而产生冲突。为此，Linux 操作系统提供了如下机制以产生惟一的关键字。

- 创建 IPC 对象时，指定关键字为 IPC_PRIVATE。以该参数创建 IPC 对象时，系统会自动分配一个可用的关键字。通过该参数创建的 IPC 对象的关键字值是 0，所以无法在其他进程中通过关键字对该对象进行访问，只能通过返回的标识符进行访问。
- 调用函数 ftok 产生一个惟一的关键字值。通过 IPC 进行通信的进程，只需要按照相同的参数调用 ftok 即可产生惟一的参数。通过该参数可有效解决关键字的产生及惟一性问题。

16.1.3 进程间通信标识符

虽然 IPC 机制提供了“键”来惟一标识一个 IPC 对象，但是对 IPC 对象的访问并不是通过“键”，而是通过标识符进行的。在 Linux 系统下，要通过编程方式访问 IPC 对象，首先必须得到 IPC 对象的标识符。

IPC 机制提供了相应的编程接口根据 IPC 键值获取标识符。与文件系统相比较，这个标识符类似于用 open 函数打开一个文件时返回的文件描述符。在文件系统中，不同的进程打开同一个文件时，返回的文件描述符可能是不同的。与文件描述符不同，不同的进程打开同一个 IPC 对象时返回的 IPC 标识符是相同的。IPC 对象标识符与文件描述符的区别如图 16-1 所示。

对于一个 IPC 对象，在打开时返回一个标识符，而关闭后再次打开同一 IPC 对象时，该标识符将顺序加 1。这就是下面将要介绍的 IPC 权限许可结构中的 seq 字段的作用。

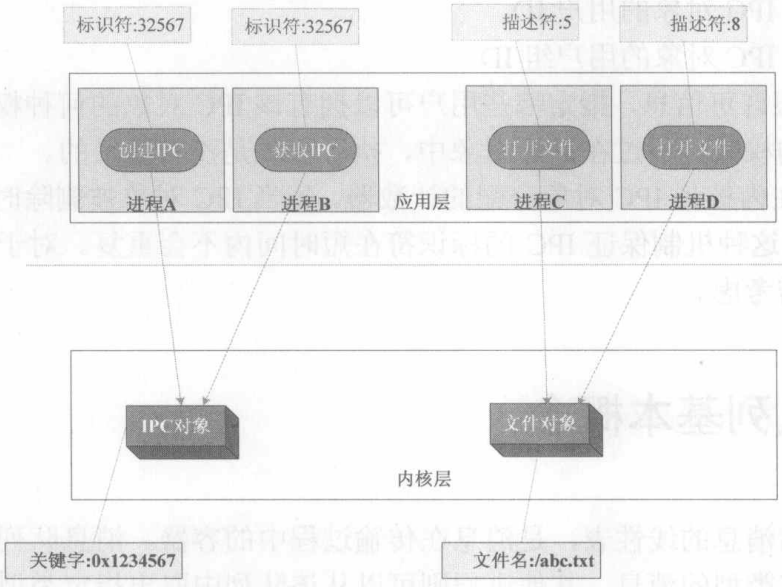


图 16-1　文件描述符与 IPC 标识符

16.1.4　IPC 权限许可结构

在 IPC 对象创建成功后，Linux 操作系统在内核为该对象分配了权限许可结构并初始化，该结构定义了对该 IPC 对象进行访问的许可权。IPC 对象创建成功后，可以通过相应的 IPC 控制函数（msgctl、shmctl、semctl 等）对权限许可及属主 ID 等进行修改。这一过程类似于文件系统中的 chmod 和 chown。该结构的定义位于头文件<bits/ipc.h>中，其结构如下所示。

```
struct ipc_perm
  {
    __key_t __key;                    /*关键字*/
    __uid_t uid;                      /*IPC 对象的属主用户 ID*/
    __gid_t gid;                      /*IPC 对象的属主组 ID*/
    __uid_t cuid;                     /*创建者的用户 ID*/
    __gid_t cgid;                     /*创建者的组 ID*/
    unsigned short int mode;          /*读写权限*/
    unsigned short int __pad1;        /*填充域*/
    unsigned short int __seq;         /*Sequence number*/
    unsigned short int __pad2;        /*填充域*/
    unsigned long int __unused1;      /*保留*/
    unsigned long int __unused2;      /*保留*/
  };
```

成员说明如下。

- __key：IPC 对象的关键字。
- uid：IPC 对象的属主用户 ID。在调用函数创建 IPC 对象时，该值是调用进程的有效用户 ID，可以通过相应的函数（msgctl）对该值进行修改。
- gid：IPC 对象的属主组 ID。与 uid 相似，在调用函数创建 IPC 对象时，该值是调用进程的有效组 ID，可以通过相应的函数（msgctl）对该值进行修改。

- cuid：创建 IPC 对象的用户 ID。
- cgid：创建 IPC 对象的用户组 ID。
- mode：权限许可信息，指定哪些用户可以拥有该 IPC 对象的何种权限。其类似于文件系统中的权限，不过在 IPC 对象中，执行权限是没有意义的。
- seq：系统在内核为 IPC 对象分配的计数器。每当 IPC 对象被删除时，该计数器值将加 1，通过这种机制保证 IPC 的标识符在短时间内不会重复。对于应用编程来说，该字段无需考虑。

16.2 消息队列基本概念

消息队列是存储消息的线性表，是消息在传输过程中的容器。消息队列一经创建，即可以向队列中写入指定类型的消息，其他进程则可以从该队列中取出指定类型的消息。

16.2.1 队列

在数据结构中，队列定义为一种运算受限的线性表。它只允许在队列的一端进行插入，而在队列的一端进行删除。队列的数据输入输出是按照“先进先出（FIFO）”的原则进行的。队列的模型如图 16-2 所示。

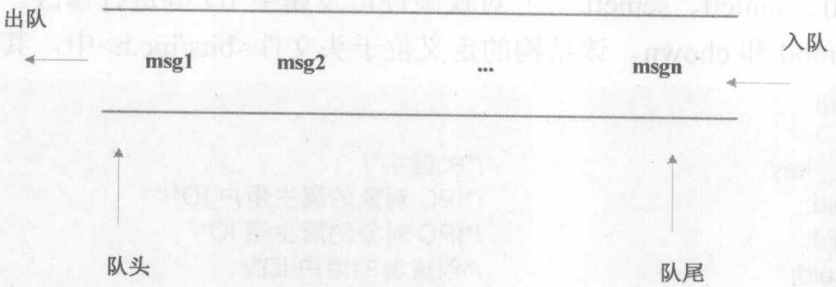


图 16-2 队列模型

16.2.2 消息

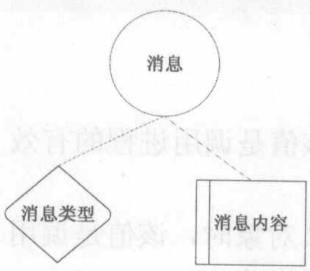


图 16-3 消息的组成

消息就是进程间传递的数据的内容。在消息队列机制中，消息是有类型的。也就是说，一条消息包含的内容不仅是数据，而且包括消息的类型信息。对于同一个消息队列来说，可以存储各种不同类型的消息，而进程可以根据需要从消息队列中取出自己所需要的类型的消息。同时，也可以向队列中发送不同类型的消息。消息的组成如图 16-3 所示。

Linux 的 IPC 机制为描述消息专门提供了一个结构模板，这个模

板的名称是 struct msgbuf，其定义位于<sys/msg.h>，结构定义如下所示。

```
struct msgbuf
  {
    long int mtype;
    char mtext[1];
  };
```

参数说明如下。

- mtype：消息类型，长整型变量。
- mtext：消息内容，字符串数组。注意：此处的长度定义为 1，由于该结构只是一个模板，在实际编程中，应根据该模板自行定义一个消息结构。这里的长度最大可以达到由系统内核规定的每条消息长度的最大值（kernel.msgmax）。

16.2.3　消息队列

消息队列具有队列的基本特征，也具有其独特的性质。在消息队列中，入队称为发送消息，出队称为接收消息，其模型如图 16-4 所示。

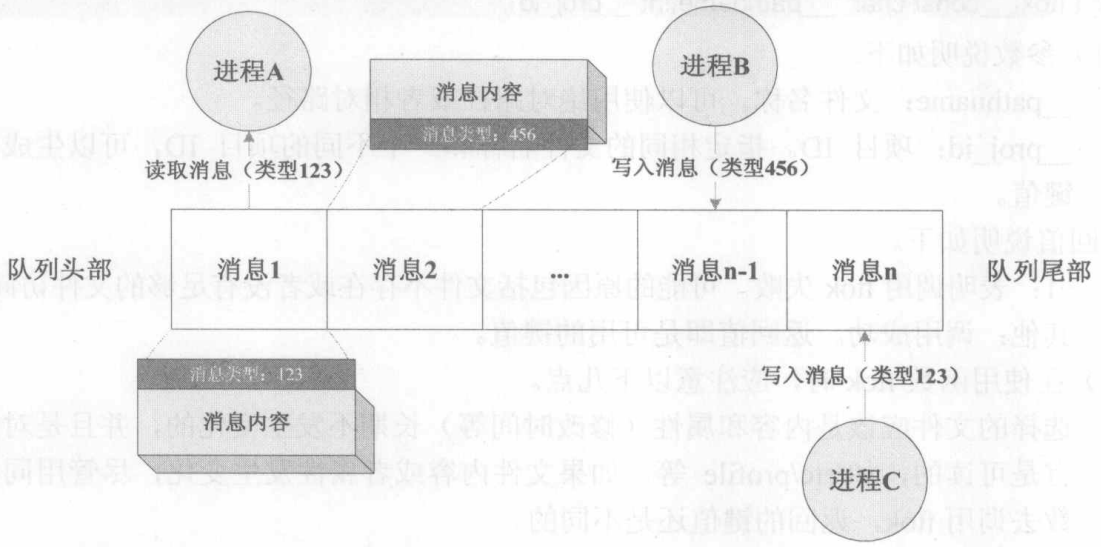


图 16-4　消息队列模型

除此之外，消息队列还具有以下特征。

- 消息队列中的消息是有类型的。在发送一条消息时，可以指定消息类型。在接收进程中，可以按照该消息类型从消息队列中获取数据。在实际应用中，可以多个应用共用一个队列，用消息类型区分不同的应用。
- 消息队列中的消息按照发送的顺序排队，对于相同类型的消息，先进入消息队列的消息先被接收。

提示：系统中对于消息队列等 IPC 机制是有极限值限制的。例如，在默认状态下，每条消息的最大字节数为 8192 字节；每个队列最多可容纳的字节数是 16384 字节；系统中允许最多建立 16 个消息队列等。这些参数可以通过执行命令 sysctl –a|grep msg 得到。

16.3 消息队列编程

System V 的 IPC 机制提供了对消息队列进行编程的接口，主要包括消息队列的创建、消息的读写、消息队列的控制及删除。另外，IPC 机制还提供了生成 IPC 键值的函数 ftok。本节将就上述内容进行详细介绍。

16.3.1 键值生成函数

键值生成函数 ftok 提供了一种键值生成方法，即根据文件名和一个整型的变量生成一个惟一的键值，并且保证每次以同样的参数调用 ftok 返回的键值是相同的。这里的整形变量称之为项目 ID。通过这种机制，使得编程人员不需要在程序中直接指定长整型的键值，而是通过定义一个文件和一个项目 ID 的方式即可。函数 ftok 的声明位于头文件<sys/ipc.h>中，其原型如下所示。

```
#include <sys/ipc.h>
key_t ftok(__const char *__pathname,int __proj_id);
```

（1）参数说明如下。

- __pathname：文件名称。可以使用绝对路径或者相对路径。
- __proj_id：项目 ID。指定相同的文件名称和一个不同的项目 ID，可以生成不同的键值。

返回值说明如下。

- -1：表明调用 ftok 失败。可能的原因包括文件不存在或者没有足够的文件访问权限。
- 其他：调用成功。返回值即是可用的键值。

（2）在使用函数 ftok 时，应注意以下几点。

- 选择的文件应该是内容和属性（修改时间等）长期不发生变化的，并且是对进程而言是可读的，如/etc/profile 等。如果文件内容或者属性发生变化，尽管用同样的参数去调用 ftok，返回的键值还是不同的。
- 可以通过指定同一个文件，而使用不同的项目 ID 的方式获取不同的键值。

【范例 16-2】使用 ftok 产生一个可用的键值。其实现过程如示例代码 16-2 所示。

示例代码 16-2

```
1   #include <stdio.h>                                          /*头文件*/
2   #include <stdlib.h>
3   #include <sys/types.h>
4   #include <sys/ipc.h>
5   main()                                                      /*主函数*/
6   {
7       key_t lKey1,lKey2,lKey3;                                /*键值变量*/
8       if((lKey1 = ftok("/etc/profile",1)) == -1)              /*调用 ftok 生成键值*/
9       {
10          perror("ftok");                                     /*提示错误信息*/
```

```
11              exit(1);
12          }
13          if((lKey2 = ftok("/etc/profile",2)) == -1)                    /*调用 ftok 生成键值*/
14          {
15              perror("ftok");                                           /*提示错误信息*/
16              exit(2);
17          }
18          if((lKey3 = ftok("/etc/profile",1)) == -1)                    /*调用 ftok 生成键值*/
19          {
20              perror("ftok");                                           /*提示错误信息*/
21              exit(3);
22          }
23          printf("key1=%ld key2=%ld key3=%ld\n",lKey1,lKey2,lKey3);     /*输出生成的键值*/
24          return 0;
25      }
```

【运行结果】经过编译链接，执行该程序的结果如下所示。

```
26  key1=16945329 key2=33722545 key3=16945329
```

【代码解析】在本例中，以/etc/profile 为文件名，分别用项目 ID 值 1、2、1 调用 ftok。根据输出的结果可以看到，用同样的参数调用 ftok，其返回值是相同的。而用同样的文件名称和不同的项目 ID 调用 ftok，其返回值是不同的。在本例中，源代码的各行解释如下。

- 第 1～4 行：头文件。注意：在 IPC 编程中，有两个头文件是必须要包含的，一个是<sys/types.h>，其中包含了类型定义信息；另一个是<sys/ipc.h>，其中定义了 IPC 机制的相关数据结构。
- 第 7 行：声明 3 个 key_t 类型的变量，用于保存 ftok 返回的键值。
- 第 8～12 行：以参数/etc/profile 和 1 调用 ftok。
- 第 13～17 行：以参数/etc/profile 和 2 调用 ftok，验证相同文件名和不同项目 ID 调用 ftok 返回的键值是不同的。
- 第 18～22 行：以参数/etc/profile 和 1 调用 ftok，验证相同文件名和相同项目 ID 调用 ftok 返回的键值是相同的。
- 第 23～24 行：输出返回的键值并退出。

16.3.2　创建消息队列

一个消息队列实际上是 Linux 在内核分配的一个数据结构。要使用消息队列进行消息传输，首先要创建一个消息队列。在 Linux 下，实现创建消息队列功能的是函数 msgget。该函数除了具有创建新的消息队列的功能外，还可以返回一个已存在的消息队列的标识符。其声明位于头文件<sys/msg.h>中，其原型如下所示。

```
#include <sys/msg.h>
int msgget (key_t __key, int __msgflg);
```

（1）参数说明如下。

- __key：输入参数，键值。

该参数的输入方式有 3 种，分别介绍如下。

①直接指定一个键值，如(key_t)0x123456。

②用 ftok 产生一个键值作为输入。

③用 IPC_PRIVATE 为参数输入，由系统创建一个可用的队列。注意：一旦使用 IPC_PRIVATE 调用 msgget，系统都将创建一个新的消息队列。并且，该消息队列的键值为 0。由此产生的问题是，在其他进程中不能用键值访问该消息队列，而必须用 msgget 返回的标识符。所以，进程间必须通过某种机制（子进程继承等）获取标识符，然后通过该标识符进行消息传送和接收。

- __msgflg：输入参数，标志和权限信息。该参数用于创建消息队列时，指定创建标志和队列的权限。通常情况下，在创建一个新的队列时，用参数 IPC_CREAT；而在获取一个已存在的消息队列的标识符时，则将该参数置为 0。该参数包括两部分内容，一是标志，二是权限。

权限可以输入的内容就是八进制的 3 组权限信息，这与文件的权限信息是相同的。例如 0666，表示该队列对于属主用户、同组用户、其他用户都具有读写权限。注意：执行权限对于消息队列是无意义的。

（2）返回值说明如下。

- -1：表明调用 msgget 失败。可以通过 errno 获取详细的错误信息。
- 其他：调用成功。返回值即是可用的标识符。

调用 msgget 成功后，操作系统在内核中分配了一个名称为 msgqid_ds 的数据结构用于管理该消息队列。IPC 机制的 3 种方式都有类似的数据结构，在后面介绍共享内存和信号量时可以看到。可以在程序中通过调用 msgctl 获取到该结构的内容，如何调用 msgctl 将在后续内容中介绍。这里首先介绍一下 msgqid_ds 的数据结构，该结构的定义位于头文件<bits/msq.h>中，其结构如下所示。

```
struct msqid_ds
{
  struct ipc_perm msg_perm;
  __time_t msg_stime;
  unsigned long int __unused1;
  __time_t msg_rtime;
  unsigned long int __unused2;
  __time_t msg_ctime;
  unsigned long int __unused3;
  unsigned long int __msg_cbytes;
  msgqnum_t msg_qnum;
  msglen_t msg_qbytes;
  __pid_t msg_lspid;
  __pid_t msg_lrpid;
  unsigned long int __unused4;
  unsigned long int __unused5;
};
```

（3）参数说明如下。

- msg_perm：许可权限结构。该结构定义不同用户对消息队列的访问权限。
- msg_stime：最后一次向该消息队列发送消息（msgsnd）的时间。
- msg_rtime：最后一次从该消息队列接收消息（msgrcv）的时间。
- msg_ctime：最后修改时间。
- __msg_cbytes：当前该消息队列中的消息长度（以字节为单位）。

- msg_qnum：当前该消息队列中的消息条数。
- msg_qbytes：该消息队列允许存储的最大长度（以字节为单位）。
- msg_lspid：最后一次向该消息队列发送消息（msgsnd）的进程 ID。
- msg_lrpid：最后一次从该消息队列接收消息（msgrcv）的进程 ID。

（4）在使用函数 msgget 时，应注意以下几点。

- 只要使用了 IPC_PRIVATE 作为参数调用 msgget，则 msgget 调用将总是返回成功，不过该消息队列是没有键值的。通过执行 ipcs –m 命令可以看到，这种队列的键值是 0x00000000。
- 要注意 msgget 不仅有创建消息队列的功能，而且在消息队列已存在的情况下，还可以返回该消息队列的标识符。要特别注意 IPC_CREAT 和 IPC_EXCL 合用时的含义。

【范例 16-3】创建一个新的消息队列，如果队列已存在，则返回其标识符。其实现过程如示例代码 16-3 所示。

示例代码 16-3

```
#include <stdio.h>                                          /*头文件*/
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>                                        /*消息队列编程需要包含此头文件*/
main()                                                      /*主函数*/
{
    key_t lKey;                                             /*键值*/
    int nMsgId;                                             /*消息队列标识符*/
    if((lKey = ftok("/etc/profile",1)) == -1)               /*生成键值*/
    {
        perror("ftok");                                     /*提示错误信息*/
        exit(1);
    }
    if((nMsgId = msgget(lKey,IPC_CREAT|IPC_EXCL|0666)) == -1)/*创建消息队列*/
    {
        if(errno != EEXIST)                                 /*创建失败并且不是由于队列已存在*/
        {
            perror("msgget");                               /*提示错误信息*/
            exit(2);
        }
        if((nMsgId = msgget(lKey,0)) == -1)                 /*消息队列已存在，返回队列描述符*/
        {
            perror("msgget");                               /*提示错误信息*/
            exit(3);
        }
    }
    printf("MsgID=%d\n",nMsgId);                            /*输出消息队列的描述符*/
    return 0;
}
```

【运行结果】经过编译链接，执行该程序的结果如下所示。

```
MsgID=32769
```

通过执行 ipcs 命令查看消息队列的情况，可以看到输出如下所示。

```
1  ------ Message Queues --------
2  key        msqid      owner      perms      used-bytes   messages
3  0x010290b1 32769      develop    666        0            0
```

【代码解析】通过输出可以看到队列已经创建成功。此时，再次运行本程序，可以看到同样的输出结果。这表明第一次程序执行的是第 16 行，即创建一个新的队列。而第二次调用执行的是第 23 行的程序，即获取已存在的消息队列标识符。在本例中，源代码的各行解释如下。

- 第 1～6 行：头文件。其中<sys/msg.h>是消息队列编程所需要的。
- 第 11～15 行：调用 ftok 生成键值。
- 第 16 行：带参数 IPC_CREAT 及 IPC_EXCL 调用 msgget。根据规则，在这两个参数作用下，如果队列不存在，则将创建队列；如果队列已存在，则返回 EEXIST。
- 第 18～22 行：如果 msgget 返回-1 并且错误码不是 EEXIST，则表示创建队列过程中出现了错误，调用 perror 格式化错误信息并输出。
- 第 23～26 行：以标志 0 调用 msgget。因为根据程序逻辑，此时已知道要创建的队列已存在，调用 msgget 只是为了返回队列的标识符。
- 第 29～30 行：输出消息队列的标识符并返回。

提示：IPC 机制（包括消息队列、共享内存、信号量）的作用范围是在整个系统范围内的，而不仅限于进程。也就是说，一旦 IPC 对象被创建，除非显示的删除该对象或者系统重启，否则该对象将一直存在。通过对文件系统的了解可以知道，进程内打开的文件在进程退出后将自动关闭，而 IPC 对象不存在关闭的概念。在进程退出后，IPC 对象将仍然存在，其中的数据也将保持原数据内容不变。

16.3.3 消息发送

消息队列创建成功后，便可以调用消息发送函数向队列中发送消息了。从前面的介绍可以知道，消息是具有类型的。因此，在发送消息时，要用到前面介绍过的消息结构。要向队列中发送消息需要调用函数 msgsnd 完成，该函数的声明位于头文件<sys/msg.h>中，其原型如下所示。

```
#include <sys/msg.h>
int msgsnd (int __msqid, __const void *__msgp, size_t __msgsz,int __msgflg);
```

（1）参数说明如下。

- __msqid：输入参数，消息队列的标识符，由 msgget 函数返回的标识符。
- __msgp：输入参数，消息结构指针。该参数的类型是 void *，实际上应该向该参数传递一个“消息结构”的指针。不过在传递参数的时候，强制类型转换为 void *。
- __msgsz：输入参数，消息长度。这里的长度是指真正的消息内容的长度，而不是消息结构的尺寸。
- __msgflg：输入参数，发送消息可选标志。如果不需要指定任何标志，在这里输入 0 即可。可以选择的标志只有 IPC_NOWAIT。在默认状态下，消息的发送操作是阻

塞的。一旦由于该消息队列中存储的消息已达到最大值（kernal.msgmnb）或者存储的消息条数已达最大值（kernel.msgmni），此时调用 msgsnd 将阻塞，直至有进程调用 msgrcv 从队列中读出消息。如果选择了 IPC_NOWAIT 标志，表明消息发送的过程是非阻塞的。如果队列已满，则 msgsnd 将立即返回-1。

（2）返回值说明如下。

- -1：表明调用 msgsnd 失败。可以通过 errno 获取详细的错误信息。
- 0：调用成功。

【范例 16-4】创建一个消息队列，并向其中发送一条消息。其实现过程如示例代码 16-4 所示。

示例代码 16-4

```
#include <stdio.h>                                  /*头文件*/
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
typedef struct                                      /*自定义数据类型，用于进行消息发送和接收*/
{
    long int nType;                                 /*消息类型*/
    char szText[256];                               /*消息内容*/
}MSG;                                               /*数据类型名称*/
main()
{
    key_t lKey;                                     /*键值变量*/
    int nMsgId;                                     /*消息队列标识符*/
    MSG msg;                                        /*消息结构*/
    if((lKey = ftok("/etc/profile",1)) == -1)       /*生成键值*/
    {
        perror("ftok");                             /*提示错误信息*/
        exit(1);
    }
    if((nMsgId = msgget(lKey,IPC_CREAT|IPC_EXCL|0666)) == -1)/*创建消息队列*/
    {
        if(errno != EEXIST)                         /*创建消息队列失败*/
        {
            perror("msgget");                       /*提示错误信息*/
            exit(2);
        }
        if((nMsgId = msgget(lKey,0)) == -1)         /*消息队列已存在，返回其标识符*/
        {
            perror("msgget");                       /*提示错误信息*/
            exit(3);
        }
    }
    memset(&msg,0x00,sizeof(MSG));                  /*清空队列结构*/
    msg.nType = 2;                                  /*指定消息类型为 2*/
    memcpy(msg.szText,"123456",6);                  /*指定消息内容*/
    if(msgsnd(nMsgId,(const void *)&msg,strlen(msg.szText),IPC_NOWAIT) < 0)    /* 非 阻 塞
发送消息*/
    {
```

```
40              perror("msgsnd");                    /*提示错误信息*/
41          }
42          return 0;
43      }
```

【运行结果】经过编译链接，执行该程序。执行完毕后，通过执行 ipcs –m 命令查看消息队列的输出，如下所示。

```
1    ------ Message Queues --------
2    key          msqid      owner      perms      used-bytes   messages
3    0x010290b1 32769      develop    666        6            1
```

【代码解析】通过输出可以看到队列中已经有一条消息，长度为 6 字节。在本例中，源代码的各行解释如下。

- 第 7～11 行：消息结构定义。此处用 typedef 声明了自定义数据类型 MSG，这就是用于发送消息的消息结构。包含了一个长整形的消息类型和 256 字节长的消息内容字段。
- 第 14～34 行：获取消息队列标识符。
- 第 35～37 行：初始化消息结构变量，并置消息类型为 2，消息内容为 6 个字节的字符串。
- 第 38 行：调用 msgsnd 向队列中发送消息。注意对消息结构变量的引用方式，是强制类型转换为 void *。另外，该调用采用了 IPC_NOWAIT 作为发送标志，这意味着该操作是非阻塞方式的。

关于消息发送，需要注意的还有以下几点。

- 在指定发送的消息类型时，类型值必须大于 0。
- 发送消息的目标队列必须对于进程而言有可写权限。
- 如果指定 IPC_NOWAIT 发送，而队列的消息数量已达最大值，此时 msgsnd 返回-1，错误码 errno 为 EAGAIN。

16.3.4 消息接收

消息已发送到消息队列中，其他进程如何获取该消息呢？这就需要用到消息接收的操作。利用消息接收操作可以从消息队列中读取指定类型的消息，也可以不指定类型，按照“先进先出”的原则读取队列头部的第一条消息。消息接收的函数是 msgrcv，其声明位于头文件<sys/msg.h>中，原型如下所示。

```
#include <sys/msg.h>
int msgrcv (int __msqid, void *__msgp, size_t __msgsz,long int __msgtyp, int __msgflg);
```

（1）参数说明如下。

- __msqid：输入参数，消息队列的标识符，由 msgget 函数返回的标识符。
- __msgp：输出参数，消息结构指针。该参数的类型是 void *，实际上应该向该参数传递一个“消息结构”的指针。注意，此处与 msgsnd 的该参数有所区别，msgsnd 是有 const 修饰的，也就是说，在 msgsnd 中，该参数是不能被修改的。而 msgrcv 则没有，因为读取到的消息要放到该结构中。

- __msgsz：输入参数，要读取的消息长度。在不确定消息长度的情况下，此处可以定义一个稍大的长度，系统将读取相应消息的全部内容，并在 msgrcv 的返回值中指明消息的正确长度。如果此处定义的要读取的长度比真正消息内容长度要小，则默认状态下将返回失败。不过，如果指定了 MSG_NOERROR 可选标志，也可以成功读取。
- __msgflg：输入参数，接收消息可选标志。

（2）返回值说明如下。

- -1：表明调用 msgrcv 失败。可以通过 errno 获取详细的错误信息。
- 其他：调用成功。返回值表示实际读取到消息内容的字节数。

【范例 16-5】从范例 16-4 创建的消息队列中读取刚刚发送的消息。其实现过程如示例代码 16-5 所示。

示例代码 16-5

```
#include <stdio.h>                                          /*头文件*/
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
typedef struct                                              /*自定义数据类型*/
{
    long int nType;                                         /*消息类型*/
    char szText[256];                                       /*消息内容*/
}MSG;                                                       /*自定义数据类型名称*/
main()                                                      /*主函数*/
{
    key_t lKey;                                             /*键值*/
    int n,nMsgId;                                           /*消息队列标识符*/
    MSG msg;                                                /*消息队列数据结构变量*/
    if((lKey = ftok("/etc/profile",1)) == -1)               /*生成键值*/
    {
        perror("ftok");                                     /*提示错误信息*/
        exit(1);
    }
    if((nMsgId = msgget(lKey,0)) == -1)                     /*获取消息队列标识符*/
    {
        perror("ftok");                                     /*提示错误信息*/
        exit(2);
    }
    memset(&msg,0x00,sizeof(MSG));                          /*清空结构变量*/
    if((n = msgrcv(nMsgId,(void *)&msg,sizeof(msg.szText),2L,0)) < 0) /*从队列中接收消息*/
    {
        perror("msgrcv");                                   /*提示错误信息*/
    }
    else
    {
        printf("msgrcv return length=[%d] text=[%s]\n",n,msg.szText); /*输出接收到的消息内容*/
    }
    return 0;
}
```

【运行结果】经过编译链接，执行该程序输出如下所示。

```
1    msgrcv return length=[6] text=[123456]
```

【代码解析】通过输出可以看到，消息已成功读出，长度为 6 字节。在本例中，源代码的各行解释如下。

- 第 28 行：调用 msgrcv 从消息队列中读取类型为 2 的消息。
- 第 34 行：msgrcv 调用成功，输出讲到的消息长度及内容。

16.3.5 控制消息队列

消息队列创建成功后，内核不仅创建了消息队列结构用于存储消息，而且创建了消息队列的控制结构，就是 16.3.2 节中介绍过的 msqid_ds。该结构中指定了该消息队列的权限信息、发送接收消息的进程、时间戳等信息。IPC 机制提供了专门的函数用于对该结构进行控制，这个函数就是 msgctl。其声明位于头文件<sys/msg.h>中，原型如下所示。

```
#include <sys/msg.h>
int msgctl (int __msqid, int __cmd, struct msqid_ds *__buf);
```

（1）参数说明如下。

- __msqid：输入参数，消息队列的标识符，由 msgget 函数返回的标识符。
- __cmd：输入参数，控制命令。
- __buf：输入或输出参数，指向 msqid_ds 结构的指针。在__cmd 参数为 IPC_STAT 时，该参数用作输出；在__cmd 参数为 IPC_SET 时，该参数用作输入。

（2）返回值说明如下。

- -1：表明调用 msgctl 失败。可以通过 errno 获取详细的错误信息。
- 0：调用成功。

【范例 16-6】获取范例 16-6 中创建的消息队列的读写权限，并将权限置为对所有用户只读，最后删除消息队列。其实现过程如示例代码 16-6 所示。

示例代码 16-6

```
1    #include <stdio.h>                                    /*头文件*/
2    #include <stdlib.h>
3    #include <errno.h>
4    #include <sys/types.h>
5    #include <sys/ipc.h>
6    #include <sys/msg.h>
7    typedef struct                                        /*自定义数据类型*/
8    {
9        long int nType;                                   /*消息类型*/
10       char szText[256];                                 /*消息内容*/
11   }MSG;                                                 /*自定义数据类型名称*/
12   main()                                                /主函数*/
13   {
14       key_t lKey;                                       /*键值变量*/
15       int n,nMsgId;                                     /*消息队列标识符变量*/
```

```
16      MSG msg;                                                    /*消息队列结构变量*/
17      struct msqid_ds qds;                                        /*消息队列控制结构变量*/
18      if((lKey = ftok("/etc/profile",1)) == -1)                   /*生成键值*/
19      {
20          perror("ftok");                                         /*提示错误信息*/
21          exit(1);
22      }
23      if((nMsgId = msgget(lKey,0)) == -1)                         /*获取消息队列标识符*/
24      {
25          perror("ftok");                                         /*提示错误信息*/
26          exit(2);
27      }
28      memset(&qds,0x00,sizeof(struct msqid_ds));                  /*清空控制结构变量*/
29      if(msgctl(nMsgId,IPC_STAT,&qds) < 0)                        /*获取消息队列属性*/
30      {
31          perror("msgctl IPC_STAT");                              /*提示错误信息*/
32          exit(3);
33      }
34      printf("msg_perm.mode=%d\n",qds.msg_perm.mode);             /*输出消息队列的访问权限*/
35      qds.msg_perm.mode &= (~0222);                               /*去除消息队列的写权限*/
36      if(msgctl(nMsgId,IPC_SET,&qds) < 0)                         /*设置消息队列的权限*/
37      {
38          perror("msgctl IPC_SET");                               /*提示错误信息*/
39          exit(4);
40      }
41      memset(&msg,0x00,sizeof(MSG));                              /*清空消息队列结构变量*/
42      msg.nType = 2;                          /*设置消息队列类型和内容，并向消息队列发送*/
43      memcpy(msg.szText,"12345",5);                                   /*设置消息内容*/
44      if(msgsnd(nMsgId,(void *)&msg,5,0) < 0)                         /*发送消息*/
45      {
46          perror("msgsnd");                                           /*提示错误信息*/
47      }
48      if(msgctl(nMsgId,IPC_RMID,NULL) < 0)                            /*删除消息队列*/
49      {
50          perror("msgctl IPC_RMID");                                  /*提示错误信息*/
51          exit(5);
52      }
53      return 0;
```

【运行结果】经过编译链接，执行该程序输出如下所示。

```
1   msg_perm.mode=438
2   msgsnd: Permission denied
```

【代码解析】程序中首先以 IPC_STAT 为参数调用 msgctl 获取消息队列的权限信息，然后调用 IPC_SET 将所有用户的写权限屏蔽，屏蔽后调用 msgsnd 向队列中发送消息，以验证是否成功设置只读。最后，以 IPC_RMID 为参数调用 msgctl 将队列删除。在本例中，源代码的各行解释如下。

- 第 1～11 行：头文件及类型定义。
- 第 18～27 行：获取消息队列的标识符。
- 第 28～33 行：以参数 IPC_STAT 调用 msgctl 以获取消息队列的控制结构。
- 第 35 行：将获取到的消息队列的权限信息与 0222“取反”后做“与”，以屏蔽写

权限。

- 第 36～40 行：以参数 IPC_SET 调用 msgctl 以设置上述修改的权限。
- 第 41～47 行：调用 msgsnd 向队列中发送消息以验证队列是否是只读的。
- 第 48～52 行：以参数 IPC_RMID 调用 msgctl 以删除该消息队列。

16.4 小结

消息队列是一种非常有用的进程间通信手段，在多进程编程的环境中经常用到。本章的重点内容是消息队列编程的内容，包括创建、发送、接收和控制消息队列。读者应该认真掌握消息队列的各项操作，仔细体会 IPC 机制与文件系统之间的相同及差异之处。在下面的章节中，将分别向读者介绍其他的两种 IPC 机制：共享内存和信号量。

第 17 章　进程间通信——共享内存

共享内存是 IPC 机制提供的另一种进程间通信机制。消息队列在实现消息的收发时，首先由发送进程从进程空间将数据复制至内核分配的数据缓冲区，接收进程再从内核的缓冲区复制到进程的虚拟地址空间。与消息队列的机制不同，共享内存是通过将内核分配的共享存储区映射到进程的地址空间实现的，没有数据的复制过程。因此，共享内存的访问速度比消息队列要快。

通常情况下，共享内存主要用于保存配置信息等相对静态的数据。应用程序启动时进行一次初始化，运行过程中各个进程从共享内存中读取信息。通过本章的学习，读者应重点掌握如下内容。

- 了解共享内存的内存映射机制。
- 熟练掌握使用共享内存接口 API 实现进程间通信编程。

17.1　共享内存基本概念

共享内存是 Linux 系统内核分配的一块存储区。两个进程通过共享内存通信是指，同一块物理内存被映射到多个进程各自的进程地址空间，各个进程都可以对共享内存中数据进行更新。使用共享内存可以更快地实现进程间通信。本节介绍共享内存的编程模型及进程运行时共享内存的地址映射机制。

17.1.1　共享内存编程模型

要使用共享内存，首先应该创建一个共享内存。在 Linux 系统下，这是由函数 shmget 实现的。共享内存创建后，需要调用 shmat 将该共享内存映射至调用进程的地址空间。映射完成后，通过返回的共享内存读写指针对共享内存进行读写。最后，调用 shmdt 关闭共享内存的映射。共享内存编程模型如图 17-1 所示。

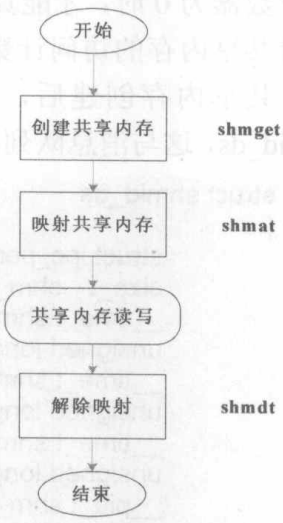


图 17-1　共享内存编程模型

17.1.2　共享内存的映射

共享内存创建后，用户程序在运行时，通过调用 IPC 接口函数将该存储区映射至应用程序的内存地址空间。成功映射后对共享内存的读写就像读写进程的变量一样快捷。映射后的进程地址空间如图 17-2 所示。

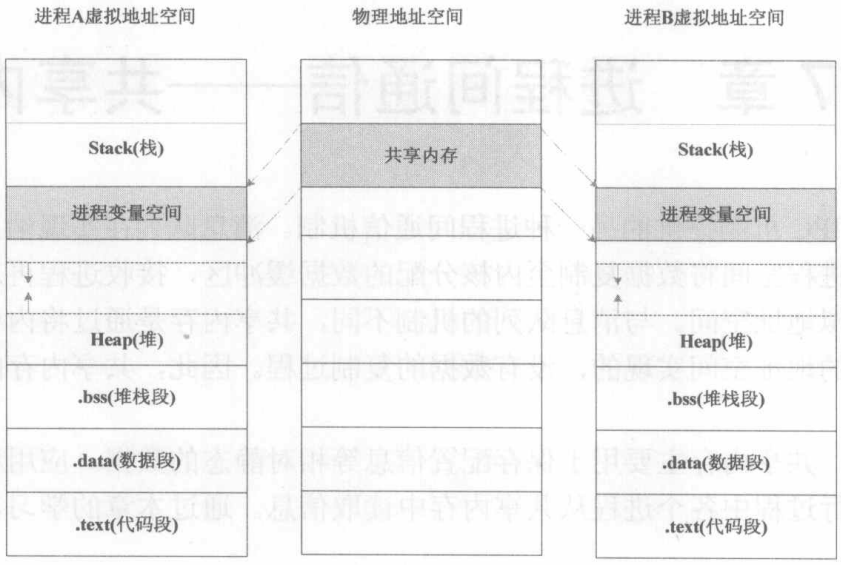


图 17-2　共享内存的映射

进行共享内存映射时，可以映射至任意的数据结构。例如，可以映射到程序中一个普通的字符数组，也可以映射到程序自定义的一个数据结构，这是由程序根据需要自行确定的。

17.1.3　共享内存数据结构

与其他 IPC 机制相同，共享内存也有键值与标识符。键值的生成使用与消息队列相同。与消息队列不同的是，共享内存有访问计数器机制。每有一个进程进行一次共享内存映射，共享内存计数器就会加 1。而每做一次解除映射操作，该计数器就会减 1。只有共享内存的访问计数器为 0 时，才能真正地删除一块共享内存。可以通过执行命令 ipcs –m 的输出项 nattch 查看共享内存的访问计数。

共享内存创建后，系统内核也为该共享内存分配了控制结构，该结构名称为 struct shmid_ds，这与消息队列是相似的。该结构的声明位于头文件<bits/shm.h>中，其结构如下所示。

```
struct shmid_ds
{
        struct ipc_perm shm_perm;
        size_t    shm_segsz;
        __time_t shm_atime;
        unsigned long int __unused1;
        __time_t shm_dtime;
        unsigned long int __unused2;
        __time_t shm_ctime;
        unsigned long int __unused3;
        __pid_t shm_cpid;
        __pid_t shm_lpid;
        shmatt_t shm_nattch;
        unsigned long int __unused4;
        unsigned long int __unused5;
}
```

成员说明如下。

- shm_perm：IPC 许可权限结构，与消息队列相同，是每个 IPC 对象都有的。
- shm_segsz：共享内存的大小，以字节为单位。
- shm_atime：最后一次调用 shmat 的时间。
- shm_dtime：最后一次调用 shmdt 的时间。
- shm_ctime：最后一次调用 shmctl 修改共享内存属性的时间。
- shm_cpid：创建共享内存的进程 ID。
- shm_lpid：最后一次操作共享内存的进程 ID。
- shm_nattch：当前共享内存的映射计数。

17.2　共享内存编程

Linux 系统提供了专门的接口 API 函数用于共享内存编程。这些函数分别是用于创建共享内存的 shmget，用于映射共享内存的 shmat，用于解除映射的 shmdt 等。本节将就这些函数逐一进行介绍。

17.2.1　创建共享内存

要使用共享内存，首先应该创建共享内存。在 Linux 下，该功能是通过函数 shmget 实现的。shmget 函数在头文件<sys/shm.h>中声明，其原型如下所示。

```
#include <sys/shm.h>
int shmget (key_t __key, size_t __size, int __shmflg);
```

（1）参数说明如下。

- __key：输入参数，键值。参见第 16 章中对键值的描述。
- __size：输入参数，要创建的共享内存的大小，单位为字节。
- __shmflg：输入参数，创建共享内存标志。

（2）在创建共享内存时，需要注意以下几点。

- 共享内存与消息队列一样，内核中对大小也有限制。可以通过执行 sysctl –a|grep shm 命令查看系统内核对大小的限制，如果有必要，可以通过修改/etc/sysctl.conf 修改内核配置。
- 共享内存创建后，除非将其删除或者重启系统，否则该共享内存将一直存在。

（3）返回值说明如下。

- -1：调用出错。详细的出错信息可从 errno 获取。
- 其他值：调用成功，返回的是共享内存的描述符。

【范例 17-1】调用 shmget 创建一块共享内存。其实现过程如示例代码 17-1 所示。

示例代码 17-1

```
1    #include <stdio.h>                                    /*头文件*/
```

```
2   #include <stdlib.h>
3   #include <sys/types.h>
4   #include <sys/ipc.h>
5   #include <sys/shm.h>                                    /*进行共享内存编程，需要包含此头文件*/
6   main()
7   {
8       key_t lKey;                                         /*键值变量*/
9       int nShmId;                                         /*共享内标识符*/
10      if((lKey = ftok("/etc/profile",1)) < 0)             /*生成键值*/
11      {
12          perror("ftok");                                 /*错误处理*/
13          exit(1);
14      }
15      if((nShmId = shmget(lKey,256,IPC_CREAT|0666)) == -1)  /*创建共享内存*/
16      {
17          perror("shmget");                               /*提示错误信息*/
18          exit(2);
19      }
20      printf("Shmid=%d\n",nShmId);                        /*输出共享内存标识符*/
21      return 0;
22  }
```

【运行结果】经过编译链接，执行该程序的结果如下所示。

```
1   Shmid=851971
```

通过 ipcs –m 查看系统中共享内存的输出如下所示。

```
2   ------ Shared Memory Segments --------
3   key        shmid      owner      perms      bytes      nattch     status
4   0x010290b1 851971     develop    666        256        0
```

【代码解析】在本例中，首先调用 ftok 生成键值，然后调用 shmget 创建一个大小是 256 字节的共享内存。根据输出的结果可以看到，共享内存已创建成功。在本例中，源代码的各行解释如下。

- 第 1～5 行：头文件。
- 第 10～14 行：调用 ftok 生成键值。
- 第 15～19 行：调用 shmget 创建一个新的共享内存。其中，权限为 666，大小是 256 字节。
- 第 20～21 行：输出生成的共享内存标识符并退出。

在本例中，输出信息的含义如下。

- 第 1 行：输出的共享内存标识符是 851971。这与用 ipcs 查看的结果是一致的。
- 第 4 行：键值（key）是 0x010290b1；共享内存标识符（shmid）是 851971；共享内存的属主（owner）是 develop；权限（perms）是 666，表明对于属主、属主的同组用户、其他用户均有读写权限；共享内存的大小（bytes）是 256 字节；共享内存被映射次数（nattch）为 0。

提示：在 Linux 系统中，共享内存物理上是位于 shm 文件系统中的文件。该文件系统挂载于/dev/shm 设备上，文件系统名称为 tmpfs。tmpfs 文件系统是完全驻留在系统内存中的，其访问速度非常快。

17.2.2　映射共享内存

共享内存创建成功后，在整个操作系统内是全局可见的。只要具备访问权限，应用程序都可以使用该共享内存。在使用共享内存时，要将共享内存映射到本地进程的地址空间。函数 shmat 可以实现这一映射功能。shmat 的声明位于头文件<sys/shm.h>中，其原型如下所示。

```
<#include sys/shm.h>
void *shmat (int __shmid, __const void *__shmaddr, int __shmflg);
```

（1）参数说明如下。

- __shmid：输入参数，共享内存标识符，由 shmget 返回。
- __ shmaddr：该参数是由调用 shmat 的用户进程指定的一个虚拟空间地址，新创建的共享内存将附在该地址后。该地址一般情况下直接指定为 NULL 即可。
- __shmflg：创建共享内存标志。

（2）返回值说明如下。

- -1：调用出错。详细的出错信息可从 errno 获取。
- 其他值：调用成功，返回值即是映射到本地进程的共享内存地址。

【范例 17-2】调用 shmget 创建一块共享内存，并将自定义的数据结构映射到该共享内存。其实现过程如示例代码 17-2 所示。

示例代码 17-2

```
#include <stdio.h>                                          /*头文件*/
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
typedef struct                                              /*自定义数据类型，用于定义共享内存结构*/
{
    int n;                                                  /*整型变量*/
    char str[256];                                          /*字符串数组变量*/
} ShmStru;                                                  /*自定义数据类型名称*/
main()                                                      /*主函数*/
{
    key_t lKey;                                             /*键值*/
    int nShmId;                                             /*共享内存标识符*/
    ShmStru *pstru;                                         /*共享内存结构指针*/
    if((lKey = ftok("/etc/profile",2)) < 0)                 /*生成键值*/
    {
        perror("ftok");                                     /*提示错误信息*/
        exit(1);
    }
    if((nShmId = shmget(lKey,sizeof(ShmStru),IPC_CREAT|0666)) == -1)/*创建共享内存*/
    {
        perror("shmget");                                   /*提示错误信息*/
        exit(2);
    }
    if((pstru = shmat(nShmId,NULL,0)) == (void *)-1)                    /*映射共享内存到本地*/
    {
```

```
28              perror("shmat");                                    /*提示错误信息*/
29              exit(3);
30          }
31          pstru->n = 1;                                           /*修改共享内存*/
32          strcpy(pstru->str,"123456");                            /*向共享内存写入数据*/
33          if( shmdt(nShmId,pstru) == -1)                          /*解除共享内存映射*/
34          {
35              perror("shmdt");                                    /*提示错误信息*/
36              exit(4);
37          }
38          return 0;
39      }
```

【运行结果】经过编译链接，执行该程序，在命令行下执行 ipcs –m 查看共享内存情况。共享内存的输出，如下所示。

```
40  ------ Shared Memory Segments --------
41  key          shmid      owner      perms      bytes      nattch     status
42  0x020290b1 884740     develop    666        260        0
```

【代码解析】在本例中，首先调用 ftok 生成键值，然后调用 shmget 创建一个大小与结构 ShmStru 相同的共享内存。根据输出的结果可以看到，共享内存已创建成功，大小是 260 字节。由于进程退出后，与共享内存的映射自动删除，所以，本例中的输出 nattch 仍然是 0。在本程序运行过程中，该值应该是 1。在本例中，源代码的各行含义解释如下。

- 第 1～5 行：头文件。
- 第 6～10 行：自定义数据结构，用于共享内存映射到本地进程。映射完成后，对该结构的查询修改即是对共享内存的操作。
- 第 21～25 行：调用 shmget 新建一块 ShmStru 大小的共享内存。
- 第 26～30 行：调用 shmat 映射共享内存到本地的结构指针 pstru 中。
- 第 31～32 行：对共享内存结构赋值。因为共享内存已映射至结构指针 pstru，所以此时对结构的操作就是对共享内存的操作。
- 第 33～37 行：调用 shmdt 删除与共享内存的映射。关于 shmdt 的细节将在下一节介绍。

在映射共享内存时，需要注意以下几点。

- 共享内存与消息队列一样，内核中对大小也有限制。可以通过执行 sysctl –a|grep shm 命令查看系统内核对大小的限制，如果有必要，可以通过修改/etc/sysctl.conf 修改内核配置。
- 共享内存创建后，除非将其删除或者重启系统，否则该共享内存将一直存在。

17.2.3 删除共享内存映射

在程序对共享内存访问结束后，应调用 shmdt 解除共享内存与进程地址空间的映射。该函数的声明位于头文件<sys/shm.h>中，其原型如下所示。

```
#include <sys/shm.h>
int shmdt (__const void *__shmaddr);
```

（1）参数说明如下。

- __shmaddr：输入参数，共享内存映射至本地的地址，即由 shmat 返回的地址。

（2）返回值说明如下。

- -1：删除映射失败。
- 0：删除映射成功。

（3）在删除共享内存映射时，需要注意以下几点。

- 在程序中使用共享内存结束后，要记住调用 shmdt 删除与进程的映射。虽然进行在退出时会自动删除与共享内存的映射，但是调用 shmdt 是一种更好的编程风格。
- 共享内存有引用计数器。调用 shmdt 只是将该共享内存的引用计数器减 1，只有引用计数值变为 0 的情况下，才可以调用 shmctl 将共享内存删除。删除共享内存映射的函数调用可以参照范例 17-2。

17.2.4　控制共享内存

与消息队列一样，共享内存创建后，系统在内核为其分配了一个类型为 struct shmid_ds 的控制结构。该控制结构的内容已在 17.1.3 节中做过介绍。IPC 提供了函数获取该结构并通过修改该结构控制共享内存，这个函数就是 shmctl。其声明位于头文件<sys/shm.h>中，原型如下所示。

```
#include <sys/shm.h>
int shmctl (int __shmid, int __cmd, struct shmid_ds *__buf);
```

（1）参数说明如下。

- __ shmid：输入参数，共享内存标识符，由 shmget 返回。
- __cmd：输入参数，控制命令。
- __buf：输入或输出参数，指向 shmid_ds 结构的指针。在__cmd 参数为 IPC_STAT 时，该参数用作输出；在__cmd 参数为 IPC_SET 时，该参数用作输入。

（2）返回值说明如下。

- -1：调用失败。
- 0：调用成功。

【范例 17-3】新建一共享内存，调用 msgctl 设置其访问权限为对非属主用户和非属主同组用户的其他用户只读，最后删除共享内存。其实现过程如示例代码 17-3 所示。

示例代码 17-3

```
1   #include <stdio.h>                          /*头文件*/
2   #include <stdlib.h>
3   #include <errno.h>
4   #include <sys/types.h>
5   #include <sys/ipc.h>
6   #include <sys/shm.h>
7   typedef struct                              /*用户自定义数据类型，用于映射共享内存*/
8   {
9    int n;                                     /*整型变量*/
```

```
10    char str[256];                                    /*字符串数组变量*/
11  } ShmStru;                                          /*数据类型名称*/
12  main()                                              /*主函数*/
13  {
14      key_t lKey;                                     /*键值变量*/
15      int nShmId;                                     /*共享内存标识符变量*/
16      struct shmid_ds sds;                            /*共享内存控制结构变量*/
17      if((lKey = ftok("/etc/profile",3)) == -1)       /*生成键值*/
18      {
19          perror("ftok");                             /*提示错误信息*/
20          exit(1);
21      }
22      if((nShmId = shmget(lKey,sizeof(ShmStru),IPC_CREAT|0666)) == -1) /*创建共享内存*/
23      {
24          perror("shmget");                                   /*提示错误信息*/
25          exit(2);
26      }
27      memset(&sds,0x00,sizeof(struct shmid_ds));              /*清空共享内存控制结构变量*/
28      if(shmctl(nShmId,IPC_STAT,&sds) < 0)                    /*获取共享内存属性*/
29      {
30          perror("shmctl IPC_STAT");                          /*提示错误信息*/
31          exit(3);
32      }
33      printf("First shm_perm.mode=0%o\n",sds.shm_perm.mode); /*输出共享内存的访问权限信息*/
34      sds.shm_perm.mode &= (~0002);                           /*去除其他用户的写权限*/
35      if(shmctl(nShmId,IPC_SET,&sds) < 0)                     /*设置共享内存的属性*/
36      {
37          perror("shmctl IPC_SET");                           /*提示错误信息*/
38          exit(4);
39      }
40      memset(&sds,0x00,sizeof(struct shmid_ds));              /*清空共享内存控制结构变量*/
41      if(shmctl(nShmId,IPC_STAT,&sds) < 0)                    /*得到共享内存属性*/
42      {
43          perror("shmctl IPC_STAT");                          /*提示错误信息*/
44          exit(5);
45      }
46      printf("Second shm_perm.mode=0%o\n",sds.shm_perm.mode); /*输出共享内存的访问权限信息*/
47      if(shmctl(nShmId,IPC_RMID,NULL) < 0)                    /*删除共享内存*/
48      {
49          perror("shmctl IPC_RMID");                          /*提示错误信息*/
50          exit(7);
51      }
52      return 0;
53  }
```

【运行结果】经过编译链接，执行该程序的输出如下所示。

```
1   First shm_perm.mode=0666
2   Second shm_perm.mode=0664
```

【代码解析】在本例中，首先调用 shmget 创建一个大小与结构 ShmStru 相同的共享内存；然后调用 shmctl 获取其读写权限；再调用 shmctl 设置其读写权限为其他用户禁止写；最后重新调用 shmctl 获取新的读写权限并删除共享内存。在本例中，源代码的各行含义解释如下。

- 第 1～5 行：头文件。

- 第 8～11 行：自定义数据结构类型，用于映射共享内存。在本例中未进行映射，只是使用了该结构的大小去分配共享内存。
- 第 17～26 行：调用 shmget 创建新的共享内存。
- 第 27～33 行：以参数 IPC_STAT 调用 shmctl，获取该共享内存的控制结构，并输出其权限信息。
- 第 34 行：将原有的权限与八进制权限 0002 的“取反”值进行“与”，实际上就是去除其他用户的写权限。注意：在 C 语言中，八进制常数用前缀表示，（具体用什么表示列举出来）十六进制常数用前缀 0x 表示。
- 第 35～39 行：以参数 IPC_STAT 调用 shmctl，设置新的访问权限信息。
- 第 40～46 行：第二次调用 shmctl 获取重新设置后的访问权限信息，验证 IPC_SET 是否成功。
- 第 47～51 行：以参数 IPC_RMID 为参数调用 shmctl，删除共享内存。

本例中，输出信息的含义如下。

- 第 1 行：第一次调用 shmctl 获取共享内存的访问权限是 0666。这与 shmget 时指定的权限是相符的。
- 第 2 行：重新设置权限后，再次调用 shmctl 获取的访问权限是 0664，并已将其他用户的写权限删除。

17.3　小结

本章向读者介绍了 Linux 共享内存的实现机制及编程方法。共享内存在实际应用中是非常有用的一种进程间通信手段，多用于保存进程的配置重要配置信息。由于共享内存不支持多进程同时读写，因此在实际应用中，如果存在多进程同时读写的情况，要采用进程间同步机制（下章将要介绍的信号量）进行读写的同步控制。

第 18 章　进程间通信——信号量

在多进程编程环境下，进程间往往需要共享某些资源，如第 17 章介绍过的共享内存。在多个进程同时对这些共享资源进行操作时，可能会出现访问冲突。为了更好地控制对共享资源的访问，Linux 系统提供了一种称作信号量的机制，这将是本章将要介绍的内容。通过本章的学习，读者应重点掌握如下内容。

- 了解 PV 操作的来源及基本原理。
- 理解 Linux 信号量的基本概念。
- 熟练掌握使用信号量接口 API 实现进程的互斥和同步。

18.1　PV 操作原理

PV 操作是信号量实现的基础。只有对 PV 操作原理深入了解，才可能真正地理解 Linux 信号量机制。本节将就 PV 操作的来源及原理进行介绍。

18.1.1　PV 操作的来源

PV 操作历史悠久，其来源可以追溯到 1962 年，是由荷兰的数学教授狄克斯特拉提出的。狄克斯特拉教授在设计 THE 操作系统时，为解决多进程、单处理机（CPU）条件下进程资源的分配问题，提出了用于进程间互斥访问的信号量机制。据说他的灵感来源于火车运行控制系统中的“信号灯”（semphore）。这也是目前大多数操作系统实现信号量时定义的名称，如 Windows 系统下的 semphore，Linux/unix 系统下的 sem 等。

为什么要使用信号量机制呢？假设进程 A 与进程 B 都对共享内存进行了映射。进程 A 负责向共享内存写入数据，进程 B 负责从共享内存读出数据。如果不对读写过程加以控制，可能出现这样的场景。进程 A 正在向共享内存写入数据并且尚未写完，而此时进程 B 正好也开始从共享内存读数据。那么，进程 B 读到的数据内容是不确定的。这一过程如图 18-1 所示，这也就是使用信号量的原因之一。

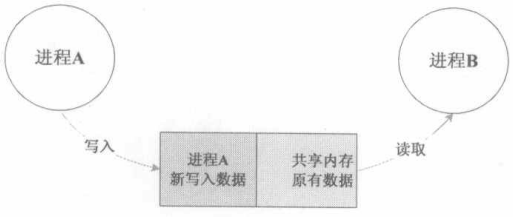


图 18-1　多进程并发访问的冲突

18.1.2　PV 操作的定义

PV 操作是两个操作的合称，这两个操作分别称为 P 操作和 V 操作。P 是荷兰语 passeren 的缩写，中文是通过的意思。V 是 Vrijgeven 的缩写，中文是释放的意思。

P 操作和 V 操作都是一种操作系统“原语”。原语是指不可中断的过程。可以这样理解为 P 和 V 操作都是原子操作，由操作系统来保证 P 操作和 V 操作的不可中断性。读者应该了解，操作系统内核的运行过程是由若干计算机指令完成的。这些指令的执行随时可能被各种硬件信号所“中断”。系统内核在接收到“中断”信号后，暂停当前指令的执行，进行“中断”处理。“中断”处理完成后，再继续进行原来暂停指令的处理。中断处理的过程如图 18-2 所示。

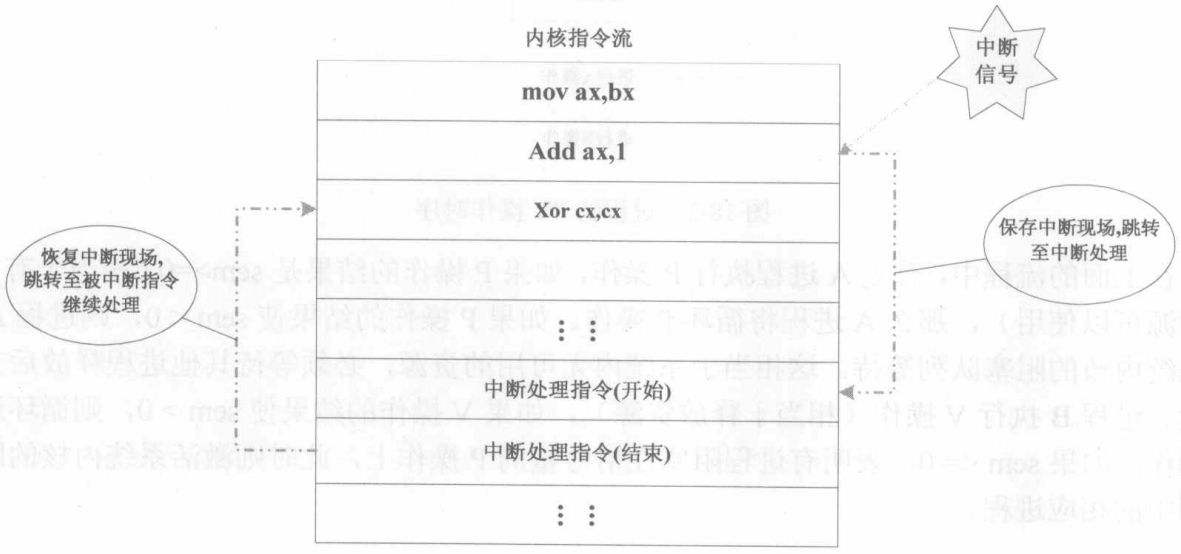


图 18-2　中断处理过程

所以一旦执行了 P 操作或者 V 操作，那么任何系统信号都不可能中断操作的执行，直到操作完成。PV 操作的前提是假定存在整型变量 sem，PV 操作的过程就是对该变量值进行加减的过程。

（1）P 操作的过程定义如下。

1）sem 减 1。

2）若 sem 减 1 后仍大于或等于零，则 P 操作返回，该进程继续执行。

3）若 sem 减 1 后小于零，则该进程被阻塞，进入操作系统的阻塞队列。

（2）V 操作的过程定义如下。

1）sem 加 1。

2）若加 1 后的结果大于零，则 V 操作返回，进程继续执行。

3）若相加结果小于或等于零，则从操作系统的阻塞队列中唤醒一个阻塞在该信号量上的进程，然后再返回原进程（调用 V 操作的进程）继续执行。

为更好地理解 PV 操作，假定系统中存在两个进程 A 和 B 对信号量进行操作。如图 18-3 所示展示了信号量操作的时序。

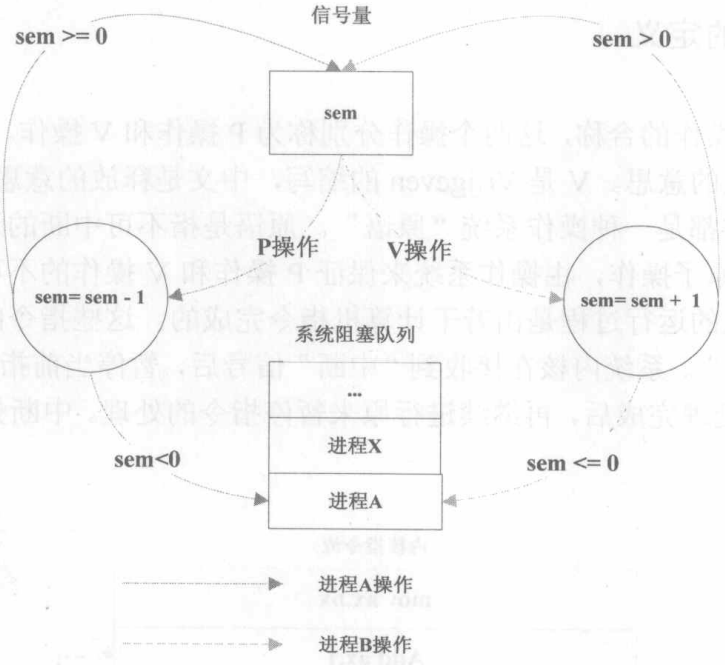


图 18-3　进程的 PV 操作时序

在上面的流程中，首先 A 进程执行 P 操作，如果 P 操作的结果是 sem>=0（相当于有 sem 个资源可以使用），那么 A 进程将循环 P 操作。如果 P 操作的结果使 sem < 0，则进程 A 进入系统内核的阻塞队列等待。这相当于系统内无可用的资源，必须等待其他进程释放后方可继续。进程 B 执行 V 操作（相当于释放资源），如果 V 操作的结果使 sem > 0，则循环进行 V 操作。如果 sem <= 0，表明有进程阻塞在信号量的 P 操作上，此时则激活系统内核的阻塞队列中的相应进程。

18.1.3　PV 操作的应用

PV 操作在进程间通信过程中应用广泛，不仅可以用来实现进程间对资源的互斥访问，而且可以实现进程间的同步。

1．进程互斥

假设系统中存在一个文件，两个进程 A 和 B 都要对其进行写操作。如果进程 A 或者进程 B 单独对文件进行写操作都不会存在问题。但是，一旦出现进程 A 和 B 同时对文件进行写，那么不可避免地会导致文件内容混乱。

可以使用 PV 操作解决该问题。首先定义一个信号量，其初始值假定为 1。在进程 A 向文件写之前，首先调用 P 操作原语锁定资源。此时资源是可用的，P 操作成功返回。进程 A 打开文件并向其中写入数据。写入数据完成后，关闭文件并调用 V 操作原语释放资源。进程 B 的操作过程与 A 相同。这个过程用伪代码描述如下所示。

```
P(sem);
WriteFile;
V(sem);
```

在上述代码中，WriteFile 的过程实际上被定义为同一时刻只能有一个进程调用。这种过程也称为“临界区”。临界区内的资源同一时刻只能被一个进程访问。上述对文件写的互斥访问过程如图 18-4 所示。

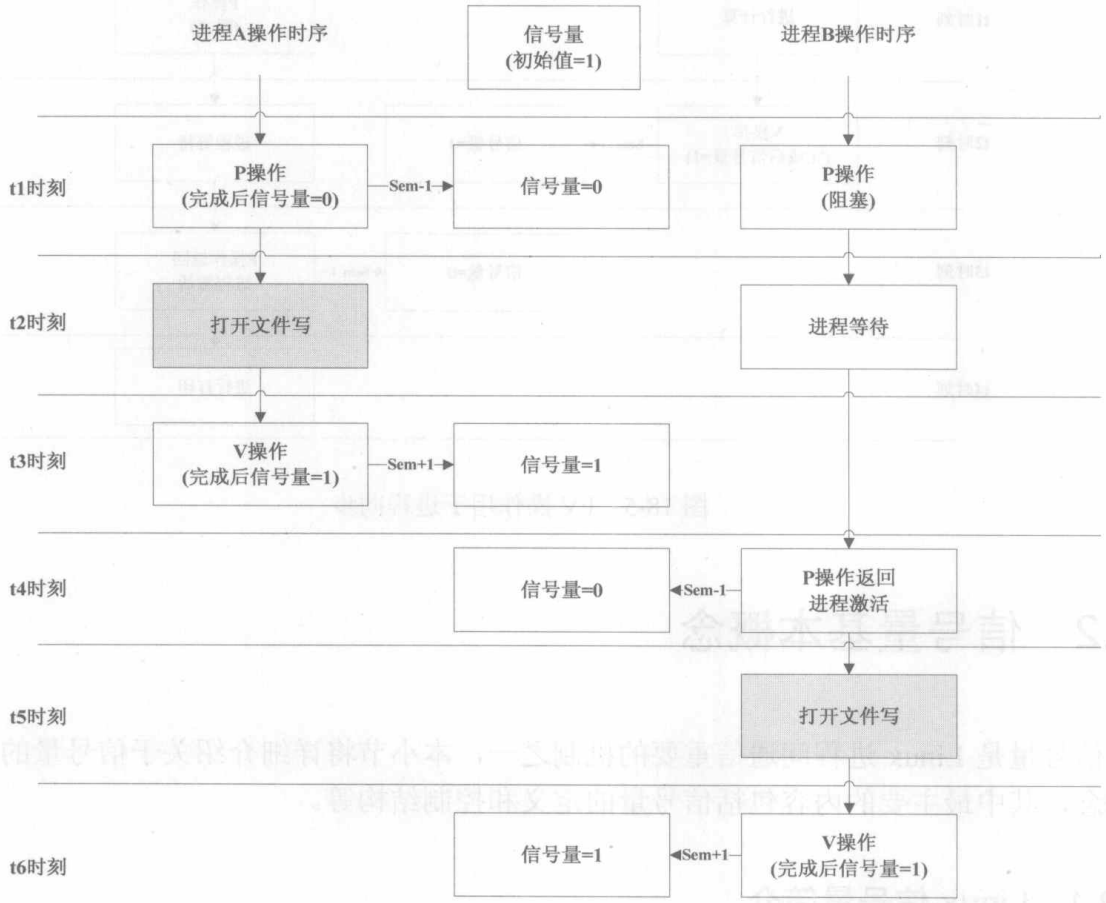


图 18-4　PV 操作用于进程互斥

2. 进程同步

在多进程编程环境下，有时一个进程（进程 A）需要暂时阻塞，等待另外的进程（进程 B）进行相应处理。进程 B 处理结束后，进程 A 再继续执行。这种情况下，需要用到进程同步。

以经典的计算——打印问题为例。假设其应用系统有两个进程 A 和 B，进程 A 负责进行问题的计算，进程 B 负责结果的打印。两个进程同时开始运行。进程 A 计算问题结果需要一定时间，而进程 B 必须要等待 A 执行完成后，方可继续。

以 PV 操作解决上述问题。在系统中定义一信号量，其初始值为 0。进程 A 执行计算任务，执行完成后调用 V 操作通知进程 B 打印。而进程 B 一开始就执行 P 操作，等待进程 A 计算结束。这个过程用伪代码描述如下所示。

```
进程 A                          进程 B
ComputeResult();                P(sem);
V(sem);                         PrintResult();
```

其时序如图 18-5 所示。

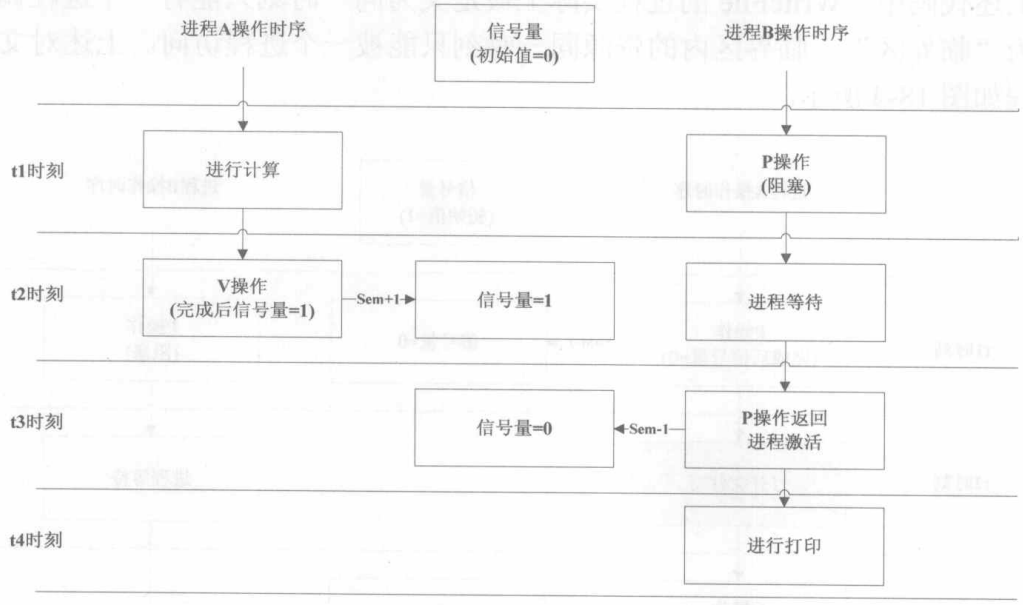


图 18-5　PV 操作用于进程同步

18.2　信号量基本概念

信号量是 Linux 进程间通信重要的机制之一，本小节将详细介绍关于信号量的基本内容和概念，其中最主要的内容包括信号量的定义和控制结构等。

18.2.1　Linux 信号量简介

与消息队列及共享内存不同，信号量实现的是一种类似计数器的功能，而不是用于存储进程间通信数据。这种机制由操作系统的 PV 操作发展而来，主要用于实现进程间的互斥及同步。

Linux 下的信号量机制在标准的 PV 操作基础上进行了扩充，主要体现在以下几点。

- Linux 支持信号量组。一次可以同时创建一组信号量，并可对组内的信号量进行不同操作。
- 每次信号量的 PV 操作不仅限于对信号量值加一或者减一，而是可以加减任意正整数。
- 支持进程退出时回滚对信号量的修改。

信号量的互斥功能用于这样一种情况：多个进程共享某项资源并且可能同时对该项资源进行操作。如果不加以控制，可能会破坏该资源数据的一致性。此时，就需要用到信号量进行进程对该资源的互斥访问。可以将访问该共享资源的代码定义为临界区，要访问临界区的代码，必须首先调用 P 操作获取对资源的访问权。而访问临界区结束后，要调用 V 操作释放信号量，以便于其他进程访问临界区。

信号量的同步功能则用于实现进程间同步。在多进程编程环境下，可能某个进程（进程

A）需要等待另一个的进程（进程 B）先执行一些操作后再执行。此时，可以用信号量实现两个进程的同步。建立一个信号量，其初始值为 0，这表示当前可用的资源数为 0。进程 A 执行时，首先调用操作量的 P 操作以获取资源，此时资源数为 0，A 进程将阻塞。进程 B 执行时，首先执行必须的功能代码，然后对信号量调用 V 操作。V 操作执行完毕后，信号量值将变为 1，此时被阻塞的 A 进程将激活继续执行。

18.2.2　信号量的控制结构

与消息队列和共享内存相同，信号量也有一个控制结构。这个控制结构在创建信号量成功后，由内核产生。其结构声明位于头文件<bits/sem.h>中，其结构如下所示。

```
struct semid_ds
{
  struct ipc_perm sem_perm;              /* operation permission struct */
  __time_t sem_otime;                    /* last semop() time */
  unsigned long int __unused1;
  __time_t sem_ctime;                    /* last time changed by semctl() */
  unsigned long int __unused2;
  unsigned long int sem_nsems;           /* number of semaphores in set */
  unsigned long int __unused3;
  unsigned long int __unused4;
};
```

成员说明如下。

- sem_perm：许可权限结构。
- sem_otime：最后一次 semop 操作的时间。
- sem_ctime：最后一次调用 semctl 改变信号量的时间。
- sem_nsems：信号量集中的信号量个数。Linux 系统可以对一个信号量集进行操作与控制。

18.3　信号量编程

Linux 系统提供了 3 个系统调用对信号量进行编程，分别是用于创建信号量的 semget、用于操作信号量的 semop 和用于控制信号量的 semctl。在本节中将就信号量的编程进行详细介绍。

18.3.1　创建信号量

要使用信号量，首先需要创建信号量。Linux 提供了系统调用 semget 用于创建信号量。与其他进程间通信机制相同，创建信号量时需要指定一个键值。该键值的生成方式可以参见第 16 章消息队列编程中相关内容的介绍。semget 的声明位于头文件<sys/sem.h>中，其原型如下所示。

```
#include <sys/sem.h>
int semget (key_t __key, int __nsems, int __semflg);
```

（1）参数说明如下。

- __key：键值，惟一标识一个信号量。其生成方式见消息队列函数 msgget 的介绍。
- __nsems：信号量集中信号量的个数。
- __semflg：创建信号量标志。主要包含两方面的信息：一是访问权限信息，该权限信息将初始化信号量控制结构中的 sem_perm 权限结构；二是创建标志，主要的创建标志包括 IPC_CREAT、IPC_EXCL 等。关于创建标志的详细说明，请参见第 16 章 msgget 的参数说明。

（2）返回值说明如下。

- -1：表示调用 semget 失败，可以通过 errno 查看错误信息。
- 其他值：调用成功，返回值为该信号量的描述符。在信号量的操作及控制中，需要用到该描述符为参数。

对于 semget 的调用，要注意以下几点。

- Linux 下 semget 创建的信号量，实际上是一个信号量集。semget 提供了一次性创建 n 个信号量的功能，其信号量编号为 0～n-1。
- semget 不仅用于生成新的信号量集，而且用于获取已存在的信号量集标识符。
- 在信号量已存在的情况下，如果调用 semget 只指定了 IPC_CREAT 标志，那么参数 __nsems 必须与创建时指定的值一致，否则 semget 将返回失败；如果调用 semget 时指定了 IPC_CREAT|IPC_EXCL 标志，semget 将返回 EEXIST 错误，即使是__nsems 与创建时不同也是如此。

信号量创建成功后，内核将为该信号量产生控制结构。根据__semflg 中的权限信息设置 sem_perm 的权限许可，根据__nsems 设置 sem_nsems 的值。同时，将根据调用进程的用户标识、调用时间等信息初始化其他相关内容。

【范例 18-1】调用 semget 创建包含一个信号量的信号量集。其实现过程如示例代码 18-1 所示。

示例代码 18-1

```
#include <stdio.h>                                          /*头文件*/
#include <stdlib.h>
#include <sys/types.h>
#include <sys/sem.h>                                        /*进行信号量编程应包含该头文件*/
main()                                                      /*主函数*/
{
    int semid;                                              /*信号量标识符变量*/
    key_t semkey;                                           /*信号量的键值变量*/
    if((semkey = ftok("/etc/profile",1)) < 0)               /*生成键值*/
    {
        perror("ftok");                                     /*错误处理*/
        exit(1);
    }
    if((semid = semget(semkey,1,IPC_CREAT|0666)) < 0) /*创建包含一个信号量的信号量集*/
    {
        perror("semget");                                   /*提示错误信息*/
        exit(1);
```

```
18        }
19        printf("semid=%d\n",semid);                    /*输出创建的信号量标识符*/
20    }
```

【运行结果】经过编译链接，执行该程序的结果如下所示。

```
1    semid=0
```

通过执行 ipcs –m 命令查看系统中信号量的输出如下所示。

```
2    ------ Semaphore Arrays --------
3    key          semid      owner      perms      nsems
4    0x010290b1 0            develop    666        1
```

【代码解析】在本例中，首先调用 ftok 生成键值，然后调用 semget 创建一个包含一个信号量的信号量集。根据输出的结果可以看到，信号量集已创建成功，其中包含了一个信号量。在本例中，源代码的各行含义解释如下。

- 第 1～4 行：头文件。
- 第 9～13 行：调用 ftok 生成键值。
- 第 14～18 行：调用 semget 创建信号量集。其中，权限为 666，信号量集中的信号量数目是 1。
- 第 19 行：输出生成的信号量集标识符。

在本例中，输出信息的含义如下。

- 第 1 行：输出的信号量集的标识符是 0。这与用 ipcs 查看的结果是一致的。
- 第 4 行：键值（key）是 0x010290b1；信号量集标识符（semid）是 0；信号量集的属主（owner）是 develop；权限（perms）是 666，表明对于属主、属主的同组用户、其他用户均有读写权限；信号量集中信号量的数目为 1（nsems）。

提示：Linux 系统下与信号量相关的极值可以通过执行 sysctl –a|grep sem 命令来获得。该命令的输出是以空格分隔的多个域，其输出格式为：SEMMSL SEMMNS SEMOPM SEMMNI。各输出项的含义如下。SEMMSL：每个信号量集的最大信号量数目；SEMMNS：整个系统可以创建的信号量最大数目；SEMOPM：每次 semop 对信号量操作的最大值；SEMMNI：系统中可以创建的信号量集的最大数目。

18.3.2　信号量操作

对资源的互斥访问及进程同步是通过信号量操作完成的。信号量操作其实就是 P 和 V 操作。在信号量创建成功后，就可以通过信号量操作实现进程的互斥及同步了。Linux 提供了系统调用 semop 对信号量进行操作。该系统调用的声明位于头文件<sys/sem.h>中，其原型如下所示。

```
#include <sys/sem.h>
int semop (int __semid, struct sembuf *__sops, size_t __nsops);
```

参数说明如下。

- __semid：输入参数，信号量集的标识符。该标识符是通过调用 semget 返回的。
- __sops：输入参数，信号量集的操作缓存。该参数表明了对该信号量集操作的详细

情况，其实际上是一个指向结构数组的指针。该结构数组的每一项定义了一个信号量的操作数据。该结构的定义如下。

```
struct sembuf
{
  unsigned short int sem_num;    /* semaphore number */
  short int sem_op;              /* semaphore operation */
  short int sem_flg;             /* operation flag */
};
```

（1）参数说明如下。

- sem_num：信号量编号。同一个信号量集中多个信号量的编号。该编号由 0 开始，指第一个信号量。
- sem_op：信号量的操作数。
- sem_flg：信号量的操作标志。
- __nsops：输入参数，操作的信号量个数。每次 semop 调用可以对一个信号量集中的多个信号量进行操作。

（2）信号量的操作标志主要包括 IPC_NOWAIT 和 SEM_UNDO，具体情况如下。

- IPC_NOWAIT：如果设置了该标志，则在 sem_op 中应该阻塞的调用都将立即返回失败。错误代码 errno 为 EAGAIN。
- SEM_UNDO：如果设置了 SEM_UNDO 标志，内核将分配一个 sem_undo 的数据结构记录对该信号量的操作。在进程结束时，在进程中对该信号量进行的 semop 操作将被撤销。设置该标志的意义在于，一旦进程没有释放信号量就退出（进程异常结束等），系统内核将根据 sem_undo 的记录主动代为释放。

（3）信号量的操作数可以分为若干情况，具体情况如下。

- sem_op > 0：表示要释放由该信号量控制的资源，将信号量的值增加 sem_op。
- sem_op = 0：表示要等待到信号量值变为 0。如果（信号量值为 0，则立即返回）；否则进入阻塞，直至信号量值为 0。
- sem_op < 0：表示要获取由该信号量控制的资源。如果当前信号量的值减去|sem_op|大于等于 0，则将信号量的值减去|sem_op|（这里|sem_op|是指 sem_op 的绝对值）。如果当前信号量的值减去|sem_op|小于 0，信号量值不变，调用进程进入阻塞状态，直至信号量的值大于等于|sem_op|。此时，被阻塞的调用进程将激活，并将信号量值减去|sem_op|。

【范例 18-2】利用范例 18-1 创建的信号量，设置初始值为 2，循环调用 semop 测试其功能。其实现过程如示例代码 18-2 所示。

示例代码 18-2

```
1   #include <stdio.h>                                    /*头文件*/
2   #include <stdlib.h>
3   #include <sys/types.h>
4   #include <sys/sem.h>
5   typedef union semun          /*自定义数据类型，该数据类型是进行 semctl 调用时所需要的*/
6   {
7       int val;                                          /*整型变量，用于保存信号量值*/
```

```
8       struct semid_ds *buf;                                   /*结构指针变量*/
9       ushort *array;                                          /*无符号短整型指针变量*/
10  } SEMCTL_UNION;                                             /*自定义数据类型名称*/
11  main()                                                      /*主函数*/
12  {
13      int n,semid;                                            /*信号量标识符变量*/
14      key_t semkey;                                           /*键值变量*/
15      SEMCTL_UNION semctl_arg;                                /*联合类型变量*/
16      struct sembuf buf;                                      /*进行 semop 调用所需要的结构变量*/
17      if((semkey = ftok("/etc/profile",1)) < 0)               /*生成键值*/
18      {
19          perror("ftok");                                     /*提示错误信息*/
20          exit(1);
21      }
22      if((semid = semget(semkey,1,0)) < 0)                    /*获取信号量标识符*/
23      {
24          perror("semget");                                   /*提示错误信息*/
25          exit(2);
26      }
27      semctl_arg.val = 2;                                     /*初始值为 2*/
28      if (semctl(semid,0,SETVAL,semctl_arg) < 0)              /*设置信号量的初始值*/
29      {
30          perror("semctl");                                   /*提示错误信息*/
31          exit(3);
32      }
33      memset(&buf,0x00,sizeof(struct sembuf));                /*清空结构*/
34      buf.sem_num = 0;          /*信号量在信号量集中的序号，由 0 开始，代表第一个信号量*/
35      buf.sem_op = -1;                                        /*信号量操作的值*/
36      buf.sem_flg = IPC_NOWAIT;                               /*非阻塞模式*/
37      for(n=0;;n++)                          /*循环调用 P 操作，直到信号量变为 0 则中断循环*/
38      {
39          if(semop(semid,&buf,1) == -1)                       /*调用 P 操作*/
40          {
41              perror("semop");                                /*提示错误信息*/
42              break;                                          /*退出循环*/
43          }
44                                                              /*获得并输出信号量的值*/
45          printf("semop[%d]:current                                                    semphore
    value=%d\n",n,semctl(semid,0,GETVAL,semctl_arg));
46      }
47  }
```

【运行结果】经过编译链接，执行该程序的结果，如下所示。

```
1   semop[0]:current semphore value=1
2   semop[1]:current semphore value=0
3   semop: Resource temporarily unavailable
```

【代码解析】在本例中，首先调用 ftok 生成键值，然后调用 semget 获取信号量的标识符并为该信号量设置值为 2。在此基础上，循环调用 semop 执行 P 操作，每次调用 P 操作结束都打印一次当前信号量的值。在本例中，源代码的各行含义解释如下。

- 第 1～4 行：头文件。
- 第 5～10 行：自定义数据类型 SEMCTL_UNION。该类型为一个联合，用于 semctl

调用。在下一节介绍 semctl 时将详细描述。在本例中，调用 semctl 设置和获取信号量的值。

- 第 9～13 行：调用 ftok 生成键值。
- 第 22～26 行：调用 semget 获取已存在的信号量的标识符。可以看到，在调用 semget 时，指定的标志位是 0，表明要获取已存在的信号量的描述符。另外需要注意的是，在获取描述符时，指定的信号量集中信号量数目应该与创建信号量集时一致。
- 第 27～32 行：调用 semctl 设置信号量的初始值为 2。
- 第 33 行：清空 sembuf 结构。
- 第 34 行：设置 semop 操作的信号量编号。该编号是指一个信号量集中信号量的编号，由 0 开始，代表第一个信号量。在本例中信号量集中只有一个信号量，所以指定该参数为 0。
- 第 35 行：设置信号量操作的值。本例中要进行 P 操作，所以指定该参数为-1。注意，在 Linux 系统中，信号量操作的数不仅可以为 1 或-1，还可以为系统极限值内任意的整数。
- 第 36 行：设置 semop 的标志。本例中，为避免进程最后阻塞，设置了 IPC_NOWAIT 参数。
- 第 37～45 行：循环调用 semop 函数，对信号量进行 P 操作。从输出可以看到，第一次调用和第二次调用是成功的。第三次调用时，由于信号量值已为 0，调用 semop 后值为-1。又因为设置了 IPC_NOWAIT，所以第三次调用立即返回失败。
- 第 45 行：输出当前信号量的值。此处调用了 semctl 来获取信号量的值。

在本例中，输出信息的含义如下。

- 第 1 行：第一次调用 semop 成功，调用结束后信号量的值是 1。
- 第 2 行：第二次调用 semop 成功，调用结束后信号量的值是 0。
- 第 3 行：第三次调用 semop 失败，由于设置了 IPC_NOWAIT 标志，调用立即返回失败。如果未设置 IPC_NOWAIT 标志，则该 semop 调用将阻塞。

在调用 semop 时，需要注意的有以下几点。

- semop 在对信号量集内的多个信号量操作时，该操作是原子性操作，这是由操作系统内核保证的。也就是说，对多个信号量的 semop 调用，要么全部成功，要么全部失败。
- 在 semop 被阻塞时，信号量的值符合 semop 要求时将会返回。除此之外，还有两种情况将导致 semop 返回；一是信号量被删除，此时 semop 将立即返回失败，错误码为 ERMID；二是进程收到信号，此时 semop 将被中断返回失败，错误码为 EINTR。

18.3.3 信号量控制

从上节的信号量操作中可以看到，在进行信号量操作时，经常需要用到对信号量进行控制，如设置信号量的值、获取信号量的值等。Linux 提供了系统调用 semctl 完成对信号量的控制。该函数的声明位于头文件<sys/sem.h>中，其原型如下所示。

```
#include <sys/sem.h>
int semctl (int __semid, int __semnum, int __cmd, ...);
```

（1）参数说明如下。

- __semid：输入参数，信号量集的标识符。该标识符是通过调用 semget 返回的。
- __semnum：输入参数，信号量的编号。信号量在信号量中集的索引。
- __cmd：输出参数，信号量控制命令。

（2）系统支持的控制命令包括以下几点。

- IPC_STAT：获取信号量集的控制结构。使用该命令时，其控制命令参数为指向 semid_ds 的结构指针。
- IPC_SET：设置信号量集的控制结构信息，同时更新 semid_ds 的最后调用 semctl 操作时间（ctime）结构成员。
- IPC_RMID：删除信号量，同时唤醒所有被该信号量集阻塞的进程。
- IPC_INFO：返回系统内核中定义的信号量集的限制及其他参数。执行该命令时，其控制命令参数为指向 seminfo 的结构指针。关于结构 seminfo 的组成，见后续内容中的描述。
- SEM_INFO：与 IPC_INFO 类似，都返回 seminfo 结构。
- SEM_STAT：与 IPC_STAT 类似，都返回 semid_ds 结构。不过，与 IPC_STAT 不同的是，使用该参数调用 semctl 时，semid 不是使用信号量标识符作为参数，而是系统内核中信号量集数组的索引。
- GETALL：返回信号量集中所包含的所有信号量的值。执行该控制命令时，控制命令参数为指向整型数组的指针。该整型数组用于保存返回的信号量值。
- GETNCNT：返回正在等待信号量值增加的进程数目。
- GETPID：返回最后一个调用 semop 对该信号量进行操作的进程 ID。
- GETVAL：返回信号量集中指定的信号量的值。
- GETZCNT：返回正在等待信号量变为 0 的进程数目。
- SETALL：根据整型数组 array 的值，设置信号量集中所有信号量的值。
- SETVAL：设置信号量集中指定信号量的值。
- ...：输入参数，控制命令参数。对于不同的控制命令，其参数格式是不同的。为满足这一需要，该参数使用了联合，其原型如下所示。

```
union semun
{
int      val;
struct semid_ds *buf;
unsigned short   *array;
struct seminfo   *__buf;
};
```

（3）在该联合中，各成员的作用说明如下。

- val：用于 SETVAL 控制命令。整型变量，用于设置信号量的值。
- buf：用于 IPC_STAT 和 IPC_SET 控制命令。指向 semid_ds 结构指针，用于获取或者设置信号量控制结构。
- array：用于 GETALL 和 SETALL 控制命令。指向短整型数组的指针，用于获取或

者设置信号量集的值。

- __buf：用于 IPC_INFO 控制命令。该控制命令是 Linux 系统下特有的，用于返回系统内核定义的信号量极值的定义信息。该成员为一个结构指针，结构类型 seminfo 的定义如下所示。

```
struct  seminfo
{
  int semmap;                    /*信号量映射入口，暂未用*/
  int semmni;                    /*系统中允许创建的信号量集的最大数目*/
  int semmns;                    /*系统中允许创建的信号量的最大数目*/
  int semmnu;                    /*暂未用*/
  int semmsl;                    /*一个信号量集中允许包含的最大信号量数目*/
  int semopm;                    /*一次 semop 可以同时操作的最大信号量数目*/
  int semume;                    /*每个进程允许 sem_undo 结构的最大数目，暂未用*/
  int semusz;                    /*sem_undo 结构的大小*/
  int semvmx;                    /*信号量允许设置的最大值*/
  int semaem;                    /*信号量的 sem_undo 机制可以记录的最大值*/

};
```

（4）返回值说明如下。

- -1：表示 semctl 调用失败，可以通过 errno 查看详细错误信息。
- 其他值：表示调用成功。

（5）程序可能的错误信息如下。

- EACCES：访问出错，权能不允许。
- EFAULT：控制命令指定的参数（联合）中，buf 或 array 所指的空间不可访问。
- EIDRM：信号集已被删除。
- EINVAL；参数无效。
- EPERM：权能不允许。
- ERANGE：给出的参数无效。

【范例 18-3】创建一个包含两个信号量的信号量集，用 SETALL 命令设置其初始值，然后调用 GETALL 获取信号量的值。其实现过程如示例代码 18-3 所示。

示例代码 18-3

```
1   #include <stdio.h>                                    /*头文件*/
2   #include <stdlib.h>
3   #include <sys/types.h>
4   #include <sys/sem.h>
5   main()                                                /*主函数*/
6   {
7       int n,semid;                                      /*信号量标识符变量*/
8       key_t semkey;                                     /*键值变量*/
9       short input_array[2];                             /*输入数组*/
10      short output_array[2];                            /*输出数组*/
11      if((semkey = ftok("/etc/profile",2)) < 0)         /*生成键值*/
12      {
13          perror("ftok");                               /*提示错误信息*/
```

```
14              exit(1);
15          }
16          if((semid = semget(semkey,2,IPC_CREAT|IPC_EXCL|0666)) < 0)    /*创建包含两个信号
    量的信号量集*/
17          {
18              perror("semget");                                          /*提示错误信息*/
19              exit(2);
20          }
21          input_array[0] = 2;                                            /*第一个信号量初始值为 2*/
22          input_array[1] = 3;                                            /*第二个信号量初始值为 3*/
23          if(semctl(semid,2,SETALL,input_array) < 0)                     /*设置信号量的初始值*/
24          {
25              perror("semctl");                                          /*提示错误信息*/
26              exit(3);
27          }
28          if(semctl(semid,2,GETALL,output_array) < 0)    /*获取信号量集中的两个信号量的值*/
29          {
30              perror("semctl");                                          /*提示错误信息*/
31              exit(3);
32          }
33          for(n=0;n<2;n++)                                               /*循环输出信号量的值*/
34              printf("semphore %d's value is %d\n",n,output_array[n]);    /*输出获得的信号量值*/
35      }
```

【运行结果】经过编译链接，执行该程序的结果，如下所示。

```
1    semphore 0's value is 2
2    semphore 1's value is 3
```

【代码解析】在本例中，首先调用 semget 创建一个包含两个信号量的信号量集，然后执行控制命令 SETALL 调用 semctl 设置信号量集的初始值，最后再执行控制命令 GETALL 调用 semctl 获取信号量集中所有信号量的值。根据输出的结果可以看到，信号量集的信号量的值分别为 2 和 3。在本例中，源代码各行含义解释如下。

- 第 21～27 行：以参数 SETALL 调用 semctl 设置信号量集中所有信号量的值。
- 第 28～32 行：以参数 GETALL 调用 semctl 获取信号量集中所有信号量的值。
- 第 33～34 行：输出信号量集中信号量的值并返回。

18.4　综合实例——利用信号量实现生产者-消费者模型

生产者-消费者模型是操作系统中典型的问题之一，模型包括生产者进程和消费者进程。生产者进程生产信息，例如，它可以是进行数学计算的进程。消费者进程使用信息，它可以是输出计算结果的进程。生产者和消费者彼此独立，且运行速度不确定，所以很可能出现生产者已产生了信息而消费者却没有来得及输出信息这种情况，这就需要一种同步机制在生产者和消费者间进行同步。本范例将利用信号量机制模拟实现这一过程。

18.4.1 需求

在系统中创建一个共享内存区，内存区结构长度为 1 的浮点型数组。由两个进程分别对共享内存区进行操作，进程一负责向循环共享内存区中写入产生的随机数，进程二负责循环从共享内存区中读出由进程一产生的随机数。

18.4.2 需求分析与设计

本例是一个典型的生产者-消费者问题。进程一作为生产者，不断产生随机数记入缓存；进程二作为消费者，不断消费缓存中的随机数。据此，设两个信号量分别用于控制进程一与进程二的读写。信号量一用于消费者（进程二）通知生产者（进程一）数据已取走，缓冲区已经可写入新数据。信号量二用于生产者（进程一）通知消费者（进程二）新数据已产生，消费者可以读取新产生的数据。

程序设定信号量一的初始值为 1，信号量二的初始值为 0。程序的实现逻辑如下所示。

```
进程一(生产者)                    进程二(消费者)
While(1)                          while(1)
{                                   {
P(信号量一);                        P(信号量二);
计算并写入数据;                       读取数据并显示输出;
V(信号量二);                        V(信号量一);
}                                   }
```

18.4.3 实现代码及分析

程序实现过程的主要代码如下。

（1）定义头文件以及对应的结构声明如下所示。

```
1   #include <stdio.h>                                    /*头文件*/
2   #include <stdlib.h>
3   #include <signal.h>
4   #include <sys/types.h>
5   #include <sys/sem.h>
6   #include <sys/shm.h>
7   typedef struct                                        /*自定义数据类型*/
8   {
9       double randnum;                                   /*双精度类型变量*/
10  } ShmStru;                                            /*自定义数据类型名称*/
11  int p(int semid,int semno)                            /*P 操作函数*/
12  {
13      struct sembuf buf;                                /*结构变量定义*/
14      memset(&buf,0x00,sizeof(struct sembuf));          /*清空结构*/
15      buf.sem_num = semno;                              /*指定要操作的信号量*/
16      buf.sem_op = -1;                                  /*P 操作(-1)*/
17      buf.sem_flg = 0;                                  /*无标志*/
18      return semop(semid,&buf,1);                       /*进行 P 操作*/
```

```
19  }
```

（2）操作函数的代码如下所示。

```
20  int v(int semid,int semno)                                  /*V 操作函数*/
21  {
22      struct sembuf buf;                                      /*结构变量定义*/
23      memset(&buf,0x00,sizeof(struct sembuf));                /*清空结构*/
24      buf.sem_num = semno;                                    /*指定要操作的信号量*/
25      buf.sem_op = 1;                                         /*V 操作(1)*/
26      buf.sem_flg = 0;                                        /*无标志*/
27      return semop(semid,&buf,1);                             /*进行 V 操作*/
28  }
```

（3）生成随机数的代码如下所示。

```
29  double randfunc()                                           /*生成随机数*/
30  {
31      srand(time(NULL));                          /*以当前时间值为产生随机数的种子*/
32      return (double)(rand()*0.01);                           /*生成随机数并返回*/
33  }
```

（4）主函数的代码如下所示。

```
34  main()
35  {
36      int n,pid;                                              /*定义整型变量*/
37      key_t key;                                              /*键值*/
38      int semid,shmid;                                        /*标识符*/
39      int semarray[2];                            /*整型数组，用于设置信号量值*/
40      ShmStru *pstru;                                         /*共享内存结构*/
41      if((key = ftok("/etc/profile",0)) < 0)                  /*生成键值*/
42      {
43          perror("ftok");                                     /*提示错误信息*/
44          exit(1);
45      }
46      if((semid = semget(key,2,IPC_CREAT|IPC_EXCL|0666)) < 0)  /*创建信号量*/
47      {
48          perror("semget");                                   /*提示错误信息*/
49          exit(2);
50      }
51      if((shmid = shmget(key,sizeof(ShmStru),IPC_CREAT|IPC_EXCL|0666)) < 0) /*创建共享内存*/
52      {
53          perror("shmget");                                   /*提示错误信息*/
54          exit(3);
55      }
56      if((pstru = shmat(shmid,NULL,0)) == (void *)-1)         /*将共享内存映射到本地变量*/
57      {
58          perror("shmat");                                    /*提示错误信息*/
59          exit(4);
60      }
61      semarray[0] = 1;                                        /*信号量 1 的初始值为 1*/
62      semarray[1] = 0;                                        /*信号量 2 的初始值为 0*/
63      if(semctl(semid,2,SETALL,semarray) < 0)                 /*设置信号量的初始值*/
64      {
65          perror("semctl");                                   /*提示错误信息*/
66          exit(5);
```

```
67          }
68          pstru->randnum = 0.00;                          /*复位随机数*/
69          if((pid = fork()) == 0)                         /*创建新进程*/
70          {
71              for(;;)                                     /*子进程中循环输出父进程创建的随机数*/
72              {
73                  p(semid,1);                             /*等待父进程通知可以读取数据*/
74                  printf("%.2f\n",pstru->randnum);        /*输出随机数*/
75                  v(semid,0);                             /*告诉父进程可以产生新数据了*/
76              }
77          }
78          else                                            /*父进程*/
79          {
80              for(n=0;n<10;n++)                           /*循环产生随机数*/
81              {
82                  p(semid,0);                             /*等待子进程通知，以便产生随机数*/
83                  pstru->randnum = randfunc();            /*产生随机数*/
84                  v(semid,1);                             /*通知子进程可以读取数据了*/
85                  sleep(1);                               /*休眠 1 秒*/
86              }
87              kill(pid,SIGKILL);                          /*杀死子进程*/
88          }
89          if( shmdt(pstru) == -1)                         /*解除与共享内存的映射*/
90          {
91              perror("shmdt");                            /*提示错误信息*/
92              exit(4);
93          }
94          shmctl(shmid,IPC_RMID,0);                       /*删除共享内存*/
95          semctl(semid,IPC_RMID,0);                       /*删除信号量*/
96          return 0;
97      }
```

【运行结果】经过编译链接，执行该程序的结果如下所示。

```
1   20780330.71
2   17818751.41
3   4100412.14
4   11679199.47
5   8698578.49
6   5642122.50
7   2613933.43
8   10212099.89
9   7248085.13
10  14967451.24
```

【代码解析】在本例中，分别创建了包含两个信号量的信号量组和一个共享内存。然后，创建子进程模拟消费者进程，父进程模拟生产者进程。由生产者进程产生随机数并写入共享内存，消费者进程从共享内存取出随机数并输出。在本例中，源代码的各行含义解释如下。

- 第 1～6 行：头文件。
- 第 7～10 行：自定义数据类型 ShmStru，用于映射共享内存。映射后，对共享内存的访问通过该结构即可进行。
- 第 11～19 行：P 操作函数，调用 semop 实现 P 操作。
- 第 20～28 行：V 操作函数，调用 semop 实现 V 操作。

- 第 29～33 行：随机数产生函数，调用 rand 产生 double 类型的随机数。
- 第 41～45 行：调用 ftok 产生键值，该键值用于创建共享内存和信号量。可以看到，不同的 IPC 机制是允许使用同一个键值的。
- 第 46～50 行：调用 semget 创建包含两个信号量的信号量集。
- 第 51～55 行：调用 shmget 创建共享内存。
- 第 56～60 行：调用 shmat 将新创建的共享内存映射至 ShmStru 结构指针。
- 第 61～67 行：以参数 SETALL 为控制命令调用 semctl，为信号量集中的两个信号量设置初始值。
- 第 68 行：为共享内存中的随机数赋初值。
- 第 69～77 行：创建子进程，在子进程中模拟消费者过程，从共享内存中取出随机数并输出。
- 第 78～88 行：父进程模拟生产者，产生新的随机数并写入共享内存。
- 第 89～93 行：调用 shmdt 解除对共享内存的映射。
- 第 94～96 行：分别删除共享内存及信号量并返回。

18.5　小结

信号量在 Linux 进程间通信机制中是非常重要的部分。通过信号量，可以实现进程间对资源的互斥访问和进程同步。在多进程编程环境下，这是非常有用的。本章的主要内容是信号量的编程内容，包括创建、操作和控制信号量等内容。希望读者在进行编程过程中，更好地利用这一机制，编写出功能强大的应用程序。

第 19 章　Linux 网络环境

Linux 本身是互联网发展的产物，并且依托互联网发展壮大。Linux 系统提供了功能强大的网络环境。在本章中，首先简要回顾计算机网络发展的历史，然后重点讲解了 Linux 环境下的 TCP/IP 协议。通过本章的学习，读者可以了解到如下内容。

- 了解计算机网络的基本概念。
- 理解 TCP/IP 协议的结构。
- 掌握网络编程具备的基础知识。

19.1　计算机网络基础

随着计算机应用的深入，特别是计算机的普及，一方面众多用户希望能共享信息资源，另一方面也希望各计算机之间能互相传递信息进行通信。个人计算机的硬件和软件配置一般都比较低，其功能也有限。因此，要求大型与巨型计算机的硬件和软件资源，以及它们所管理的信息资源应该为众多的微型计算机所共享，以便充分利用这些资源。基于这些原因，促使计算机向网络化发展，将分散的计算机连接成网，组成计算机网络。

计算机网络是现代通信技术与计算机技术相结合的产物。所谓计算机网络，就是把分布在不同地理区域的计算机与专门的外部设备用通信线路互联成一个规模大、功能强的网络系统，从而使众多的计算机可以方便地互相传递信息，共享硬件、软件、数据信息等资源。

19.1.1　计算机网络分类

计算机网络类型的划分标准有多种，从地理范围划分，是一种公认的通用网络划分标准。按计算机联网的区域大小，我们可以把网络分为局域网（LAN，Local Area Network）、城域网（MAN，Metropolitan Area Network）和广域网（WAN，Wide Area Network）。

局域网（LAN）是指在一个较小地理范围内的各种计算机网络设备互联在一起的通信网络，可以包含一个或多个子网，通常局限在几千米的范围之内，如在一个房间、一座大楼，或是在一个校园内的网络就称为局域网。局域网通常被用于连接公司办公室或工厂里的个人计算机和工作站，以便共享资源（打印机）和交换信息。IEEE 的 802 标准委员会定义了多种主要的局域网标准，即 Ethernet（以太网）、Token King（令牌环网）、FDDI（光纤分布式接口网络）、ATM（异步传输模式网）以及最新的 WLAN。

城域网（MAN）基本上是一种大型的 LAN，通常使用与 LAN 相似的技术，它采用 IEEE 802.6 标准。它可以覆盖一组邻近的公司办公室和一个城市，既可能是私有的也可能是公用的。MAN 与 LAN 相比，扩展的距离更长，连接的计算机数量更多。一个 MAN 网络通常连接着

多个 LAN 网。

广域网（WAN）连接地理范围较大，常常是一个国家或是一个洲，其目的是为了让分布较远的各局域网互联。我们平常所说的互联网（Internet）就是由不同的广域网、城域网、广域网互相连接构成的，是最大最典型的广域网。

19.1.2　网络拓扑结构

网络拓扑结构是指用传输媒体互联各种设备的物理布局。将参与工作的各种网络设备用媒体互联在一起有多种方法。网络的拓扑结构反映了网络中各个实体的结构关系，是建设计算机网络的第一步，是实现各种网络协议的基础，它对网络的性能、系统的可靠性都有重大影响。目前主要的网络拓扑结构有总线型拓扑、星型拓扑、环型拓扑。

1．总线型拓扑结构

将网络中的所有设备通过相应的硬件接口直接连接到公共总线上，结点之间按广播方式通信，一个结点发出的信息，总线上的其他结点均可“收听”到。总线型拓扑结构适用于网络节点数目相对较少的局域网，传输速率一般在 100Mbps，通常使用同轴电缆作为通信介质。由于所有结点共享同一传输链路，每次只能有一个设备使用传输链路。因此，需要采用某种形式的访问控制策略，来决定下一个使用链路的结点。

一般情况下，总线型网络使用载波监听多路访问或冲突检测控制策略。总线型拓扑的优点是结构简单、布线容易、可靠性较高，易于扩充，是局域网常采用的拓扑结构；其缺点是所有的数据都需经过总线传送，总线成为整个网络传输的瓶颈，出现故障诊断较为困难。典型的总线型网络拓扑结构如图 19-1 所示。

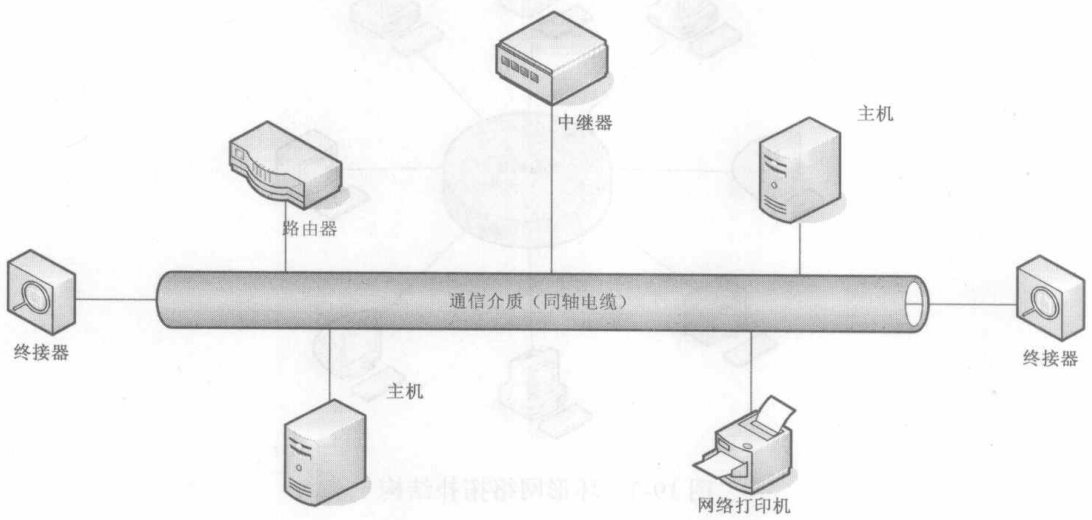


图 19-1　总线型网络拓扑结构

2．星形拓扑结构

每个结点都由一条单独的通信线路与中央节点连接，任何两个节点要进行通信都必须经过中央节点控制。现有的数据处理和声音通信的信息网大多采用星形拓扑，如电话交换机就是星形网拓扑结构的典型实例。星形拓扑结构的优点包是结构简单、容易实现、便于管理，

连接点的故障容易监测和排除等；其缺点是中央节点负载过重，可靠性低，通信线路利用率低。中央节点是网络的可靠瓶颈，一旦出现故障将导致网络的瘫痪。星形网络拓扑结构如图 19-2 所示。

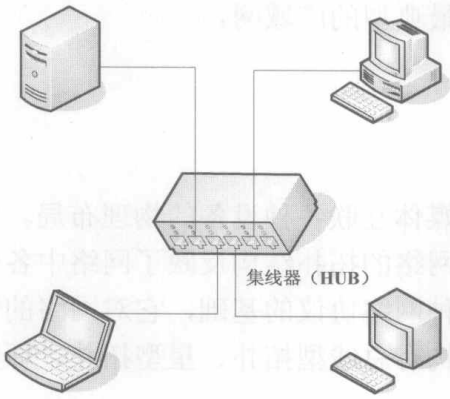


图 19-2　星形网络拓扑结构

3．环形拓扑结构

各节点通过通信线路组成闭合回路，环中数据只能单向传输，信息按固定方向流动，或顺时针方向，或逆时针方向。该结构的优点是结构简单，适合使用光纤，传输距离远，传输延迟确定。但是，传输效率低，可扩展性差，只支持较少的网络节点。环网中的每个节点均成为网络可靠性的瓶颈，任意节点出现故障都会造成网络瘫痪，另外故障诊断也较困难。环形网络的一个典型代表是采用同轴电缆作为传输介质的 IEEE802.5 的令牌环网。环形网络拓扑结构如图 19-3 所示。

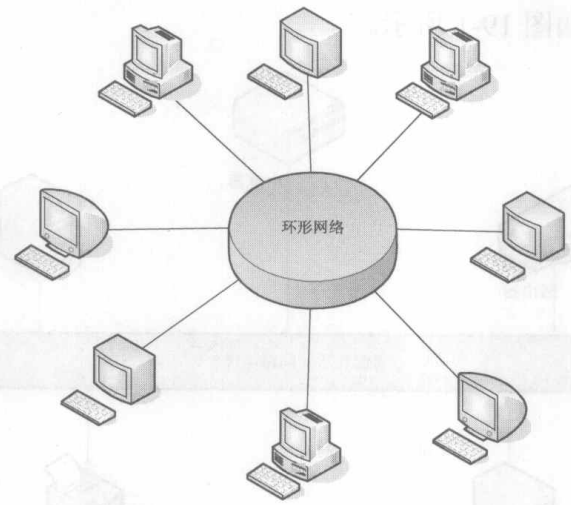


图 19-3　环形网络拓扑结构

19.1.3　网络通信协议

网络上的计算机之间是如何交换信息的呢？这就像我们说话用某种语言一样，在网络上的各台计算机之间也有一种语言，这就是网络通信协议。网络通信协议是计算机网络和分布

系统中，互相通信的对等实体间交换信息时所必须遵守的规则的集合。通信协议定义了网络上的各种计算机和设备之间相互通信、数据管理、数据交换的整套规则。

通过这些规则，网络上的节点才有了彼此通信的“共同语言”。不同的计算机之间必须使用相同的网络协议才能进行通信。当然，网络协议也有很多种，具体选择哪一种协议则要看情况而定。主要的网络通信协议有 OSI 协议、TCP/IP 协议、NetBEUI 协议、IPX/SPX 协议、SNA 协议等。下面将为读者简要介绍 OSI 通信协议模型，在下一节中将详细介绍 TCP/IP 协议。

在介绍 OSI 协议之前，有必要向读者介绍一下网络分层的基本概念。两个系统中实体间通信是一个十分复杂的过程。为减少协议设计过程的复杂性，多数网络的实现都按分层次的方式来组织。每一层完成其特定的功能，同时，每一层又建立在它的下层之上。不同的网络标准，其分层的数量、层次的功能及内容都不尽相同。但是，对于每层来说，都是通过层间的接口向上一层提供特定的服务，而这种服务是以何种方式实现的对上层屏蔽，网络协议的层次结构如图 19-4 所示。

网络的分层模型遵循以下规则。

- 第 n 层的通信实体在实现其功能时，只能使用第 n-1 层提供的服务。
- 第 n 层向第 n+1 层提供服务，这里的服务不仅仅包括第 n 层实现的服务，而是第 n 层（含 n 层）以下层所实现的全部服务的总和。
- 相邻层间通过接口提供服务，下层所提供服务的实现细节对上层完全屏蔽。
- 最底层只向上提供服务，是服务的基础；最高层是用户，只使用服务；中间层次既是下一层的用户，同时又是上一层服务的提供者。

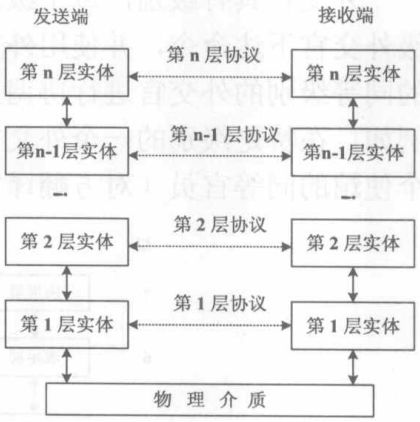


图 19-4　网络协议的层次结构

19.1.4　OSI 参考模型

为了使这些隶属于不同网络体系结构的网络产品实现互联，在国际标准化组织（ISO）和国际电报电话咨询委员会（CCITT）的共同努力下制定了开放系统互联参考模型（Open System Interconnection），简称 OSI 参考模型。各厂商均按照 OSI 参考模型制定网络协议和生成网络设备，从而统一了不同厂商、不同网络产品的开发标准，使不同产品的互联不再困难。

OSI 参考模型采用了分层的结构化技术，共分 7 层，即物理层、数据链路层、网络层、传输层、会话层、表示层、应用层。每一层所提供的功能均有详细的定义，如图 19-5 所示。

在 OSI 模型中，系统通信是由协议栈的每一层与通信对端的协议栈协调完成的。例如，一个系统的传输层将根据另一个系统的传输层的情况协调它的活动。为更好地理解网络通信模型，以外交活动为例进行说明。设想在两个大使馆之间需要安排一次正式会议。表面上看来，由两位大使签署正式协议，而在背后，外交官和官员们整理文件，制定日程，并进行其他活动。

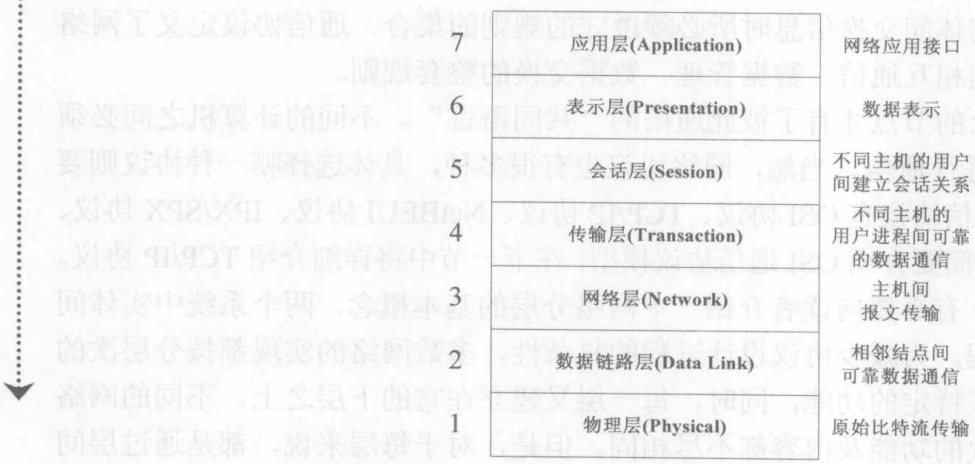


图 19-5　OSI 参考模型

外交官具有级别，每个级别的外交官为更高级的官员做一些服务。最高级别的大使向低级外交官下达命令，并使用外交官提供的服务。同时，大使级以下的外交官会与另一个使馆的同等级别的外交官进行协调工作。每个外交官都按照为他们这个级别制定的外交惯例执行。例如，在特定级别的一个外交官员可能提供语言翻译服务。根据翻译过程，这个官员与另一个使馆的同等官员（对方翻译官员）进行通信。OSI 的网络通信模型如图 19-6 所示。

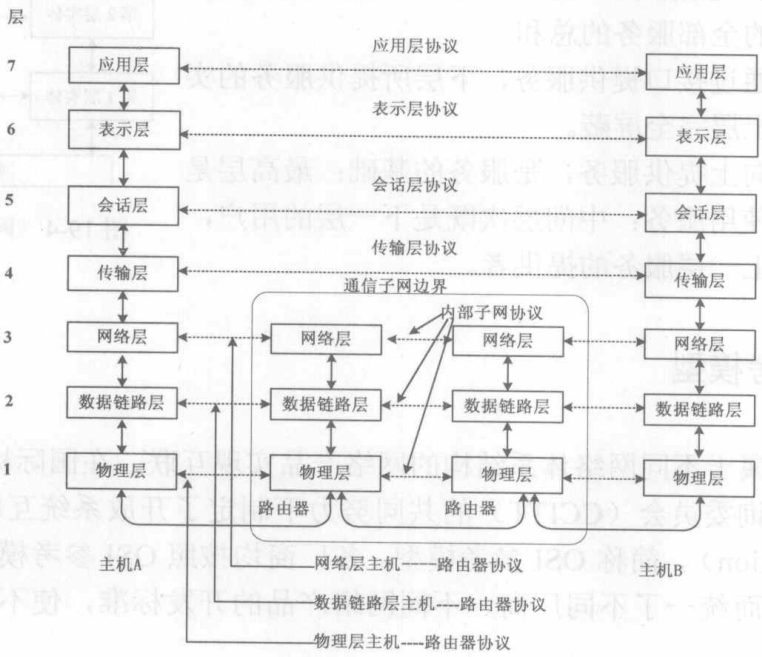


图 19-6　OSI 网络通信模型

在外交领域，一个使馆的官员只需简单地拿起电话找另一个使馆的同等级的官员。与此类似，在网络通信领域，尽管物理数据流是从上至下（或者从下至上）按照垂直方向传输的，但在逻辑上，可以看成是每个协议层的实体，通过对信息打包与对等实体通信。

在实际通信过程中，数据最终将抵达最低的物理层，变成二进制的数据比特流，在物理连接介质间传递。当信息传递过协议层，它们形成一个称为协议数据单元的分组。每一层的实体按照约定向协议数据单元中加入自己的信息头部。当数据通过物理层抵达对端系统，它

向上通过每一层协议栈，相应的信息头部被取下，并传送给层对应的实体。

以前面介绍过的外交活动来说，设想 A 大使希望送给另一使馆的 B 大使一个消息。A 大使写了一封信，并交于下一级别的助手。这个助手将信装入信封，并在信封上写明另一使馆和他同等级别的外交人员的地址。这个信件包被传给下一级别的外交人员，他再对这个信件打包，并在信封上写明另一使馆和他同等级别的外交人员的地址。

这个过程根据级别进行下去直到到达“物理层”为止。在“物理层”，信件被邮差送往另一使馆。在另一使馆，每个外交人员只阅读给他的信息地址，并将里面的信封送给上一级别的外交官，直至最终信件送达 B 大使。在 OSI 参考模型中的数据流如图 19-7 所示。

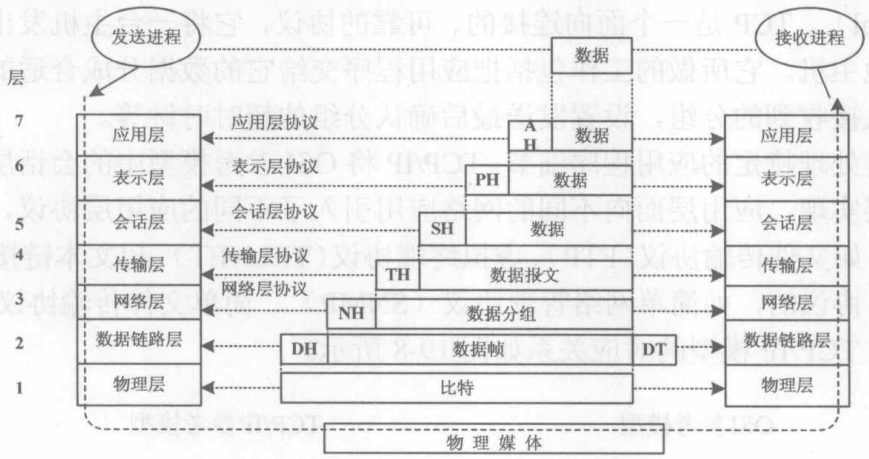


图 19-7　OSI 数据流模型

OSI 是一个理想的网络通信模型，因此一般的网络系统只涉及其中的几层，很少有系统能够具有所有的 7 层，并完全遵循它的规定。由于 OSI 的设计划分过细，过于复杂，在实际应用过程中逐渐被 TCP/IP 协议所取代。TCP/IP 协议已成为事实上的网络通信标准。

19.2　TCP/IP 协议概述

美国国防部高级研究计划局从 20 世纪 60 年代开始致力于研究不同类型计算机网络之间的互相连接问题，最终成功地开发出著名的 TCP/IP 协议，支持不同厂家计算机主机的通信，并应用 TCP/IP 组建了 APRA 网。最初，ARPA 网只连接了美国境内的 4 所大学。随后的几年中，它通过租用的电话线连接了数百所大学和政府部门。最终 ARPANET 发展成为全球规模最大的互联网络——因特网。TCP/IP 协议也成为是事实上的工业标准。

19.2.1　TCP/IP 分层模型

与 OSI 参考模型相似，TCP/IP 也是一种分层模型。它是基于硬件层次上的 4 个概念性层次构成，即网络接口层、IP 层、传输层、应用层。

网络接口层为 TCP/IP 协议的最底层，也被称为数据链路层或主机-网络层。网络接口层

通常包括操作系统中的设备驱动程序和计算机中对应的网络接口卡，它们一起处理与传输媒介的物理接口细节。它与 OSI 模型的下两层相对立，即物理层与数据链路层。

网络层，有时也称作网络互连层，是整个 TCP/IP 协议栈的核心。它的功能是把分组发往目标网络或主机。TCP/IP 协议族中，网络层协议包括 IP 协议（网际协议）、ICMP 协议（Interne 互联网控制报文协议）以 IGMP 协议（Internet 组管理协议）。除分组发送功能外，网络层还完成路由及异构网络互连任务。

传输层主要为两台主机上的应用程序提供端到端的通信。在传输层定义了两种服务质量不同的协议。即传输控制协议 TCP（transmission control protocol）和用户数据报协议 UDP（user datagram protocol）。TCP 是一个面向连接的、可靠的协议，它将一台主机发出的数据流无差错地发往到其他主机。它所做的工作包括把应用程序交给它的数据分成合适的小块交给下面的网络层，确认接收到的分组，设置发送最后确认分组的超时时钟等。

应用层负责处理特定的应用程序细节。TCP/IP 将 OSI 参考模型中的会话层和表示层的功能合并到应用层实现。应用层面向不同的网络应用引入了不同的应用层协议，其中，既有基于 TCP 协议的，如文件传输协议（FTP）、虚拟终端协议（TELNET）、超文本链接协议（HTTP），也有基于 UDP 协议的，如简单网络管理协议（SNMP）、简单文件传输协议（TFTP）等。OSI 参考模型与 TCP/IP 模型的对应关系如图 19-8 所示。

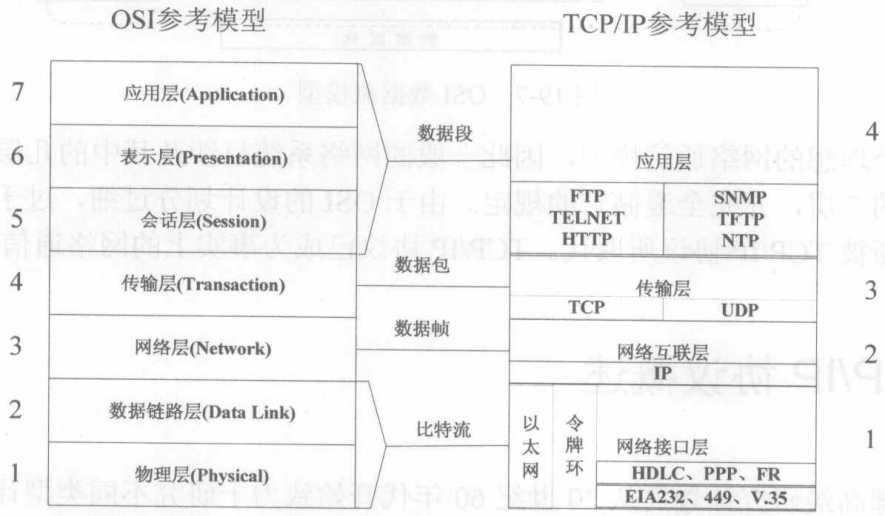


图 19-8　OSI 参考模型与 TCP/IP 参考模型的对应关系

19.2.2　TCP/IP 协议族

TCP/IP 协议实际上并不仅仅包含 TCP 及 IP 协议，它是包括 TCP 及 IP 协议在内的若干协议的总称，这些协议分布在不同的协议层次中，如图 19-9 所示。

- IP：网际协议，可以分为版本 4（IPv4）及版本 6（IPv6）。IPv4（通常就称之为 IP）自 80 年代早期以来一直是网际协议族的主力协议，它使用 32 位的地址，给 TCP、UDP、ICMP 提供递送分组的服务。IPv6 在 90 年代中期设计出来，用以替代 IPv4，主要变化是使用 128 位的大地址以处理 90 年代因特网爆发性的增长。

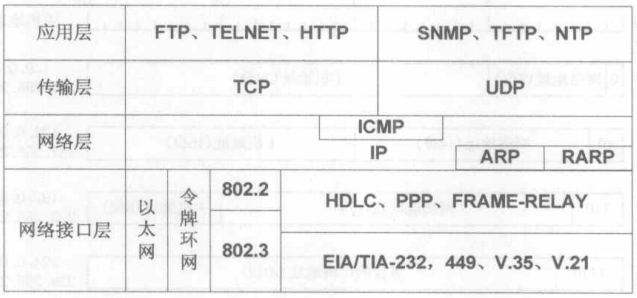


图 19-9　TCP/IP 协议族

- TCP：传输控制协议。TCP 是一种面向连接的协议，它给用户进程提供可靠的全双工的字节流。TCP 关心确认、超时和重传等具体细节。大多数网络应用程序使用 TCP。
- UDP：用户数据报协议。UDP 是一种无连接协议。与 TCP 不同，UDP 不能保证每一 UDP 数据报可以到达目的地。
- ICMP：网际控制消息协议。ICMP 处理路由器和主机间的错误和控制消息。这些消息一般由 TCP/IP 网络软件自身产生和处理。
- ARP：地址解析协议。ARP 把 IP 地址映射到硬件地址(以太网地址)。ARP 一般用于广播网络，如以太网、令牌环网等，但不用于点对点网络。
- RARP：反向地址解析协议。RARP 把硬件地址映射到 IP 地址。它有时用于无盘节点，如引导时的 x 终端。
- FTP、TELNET、HTTP 等协议都是应用层协议，用于实现文件传输、超文本传输等高层功能。

19.2.3　网络地址

无论采用何种通信协议，都需要给网络设备分配网络地址。网络地址在整个网络中应是惟一的。在 TCP/IP 协议中，这个地址也称为 IP 地址。参与网络通信的每台主机都至少应该分配一个 IP 地址。IP 协议使用这个地址在主机间传递信息。IP 地址的长度是 32 位（4 个字节）。为便于记忆，将 32 位地址从逻辑上划分为 4 段，每段 8 位。

每一段用一个十进制数据表示，其取值范围为 0～255。在 4 个段之间用句点分隔，称为点分十进制表示法。例如，IP 地址 00001010000000000000000000000001 用点分十进制表示就是 10.0.0.1。IP 地址就像是我们的家庭住址一样，如果要写信给一个人，你就要知道他的地址，这样邮递员才能把信送到。计算机发送信息就好比是邮递员，它必须知道惟一的“家庭地址”才能不至于把信送错。只不过现实环境中的地址使用文字来表示的，而计算机网络地址用十进制数字表示。

最初设计互联网络时，为了便于寻址以及层次化构造网络，每个 IP 地址包括两个标识码，即网络标识和主机标识。同一个物理网络上的所有主机都使用同一个网络标识，网络上的一个主机（包括网络上工作站，服务器和路由器等）有一个主机标识与其对应。IP 地址根据网络标识的不同划分为 5 种类型，A 类地址、B 类地址、C 类地址、D 类地址和 E 类地址，如图 19-10 所示。

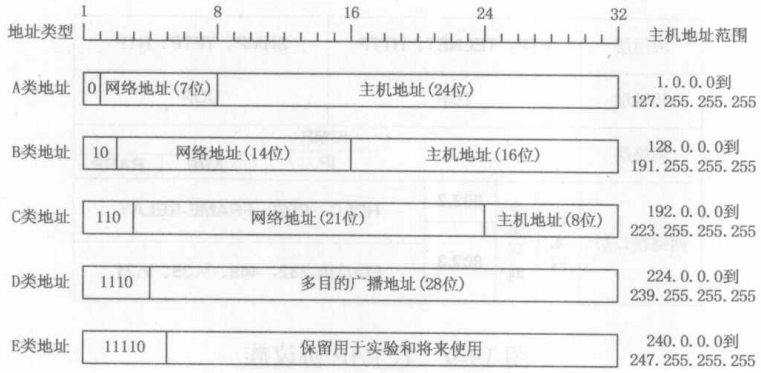


图 19-10　IP 地址分类

1．A 类 IP 地址

一个 A 类 IP 地址由 1 字节的网络地址和 3 字节主机地址组成，网络地址的最高位必须是“0”，地址范围 1.0.0.1～126.255.255.254。可用的 A 类网络有 126 个，每个网络能容纳 1 亿多个主机。

2．B 类 IP 地址

一个 B 类 IP 地址由 2 个字节的网络地址和 2 个字节的主机地址组成，网络地址的最高位必须是“10”，地址范围 128.0.0.1～191.255.255.254。可用的 B 类网络有 16382 个，每个网络能容纳 6 万多个主机。

3．C 类 IP 地址

一个 C 类 IP 地址由 3 字节的网络地址和 1 字节的主机地址组成，网络地址的最高位必须是“110”，地址范围 192.0.0.1～223.255.255.254。C 类网络可达 209 万余个，每个网络能容纳 254 个主机。

4．D 类 IP 地址

D 类 IP 地址第一个字节以“1110”开始，它是一个专门保留的地址，用于多点广播（Multicast）。它并不指向特定的网络，目前这一类地址被用在多点广播（Multicast）中。多点广播地址用来一次寻址一组计算机，它标识共享同一协议的一组计算机。地址范围 224.0.0.1～239.255.255.254。

5．E 类 IP 地址

以“11110”开始，为将来使用保留。全零（“0.0.0.0”）地址对应于当前主机，全“1”的 IP 地址（“255.255.255.255”）是当前子网的广播地址。

【范例 19-1】查看系统的当前主机 IP 地址。其实现过程如示例代码 19-1 所示。

示例代码 19-1

```
1    ifconfig                    #运行 ifconfig 命令
```

【运行结果】在 shell 下运行上述脚本（需要超级用户权限），其结果如下所示。

```
2    eth0      Link encap:Ethernet  HWaddr 00:0C:29:F0:DC:95
3              inet addr:169.254.215.193  Bcast:169.254.255.255  Mask:255.255.0.0
4              inet6 addr: fe80::20c:29ff:fef0:dc95/64 Scope:Link
```

```
5                UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
6                RX packets:988 errors:0 dropped:0 overruns:0 frame:0
7                TX packets:231 errors:0 dropped:0 overruns:0 carrier:0
8                collisions:0 txqueuelen:1000
9                Interrupt:10 Base address:0x1400
10   lo          Link encap:Local Loopback
11               inet addr:127.0.0.1  Mask:255.0.0.0
12               inet6 addr: ::1/128 Scope:Host
13               UP LOOPBACK RUNNING  MTU:16436  Metric:1
14               RX packets:342 errors:0 dropped:0 overruns:0 frame:0
15               TX packets:342 errors:0 dropped:0 overruns:0 carrier:0
```

【代码解析】在本例中，ifconfig 是 Linux 系统下用来查看网络接口参数的命令，也可以直接在命令行上指定要查看的接口，如 ifconfig eth0。

在本例中，第一列是指接口名称，本例中有两个网络接口，分别是 eth0 及 lo；第二列表示接口详细信息。其输出信息的含义具体如下。

- 第 1 行：网络接口 eth0。网络类型是以太网，MAC 地址是 00:0C:29:F0:DC:95。MAC 地址是网络接口设备的物理地址，是设备生产过程中固化在设备 ROM 中的。eth0 中的“eth”指网络接口是以太网，“0”表示第一个以太网接口。如果系统中存在多个以太网接口设备，则依次为 eth1、eth2 等。
- 第 2 行：网络接口的 IP 地址是 169.254.215.193，网络广播地址是 169.254.255.255，子网掩码是 255.255.0.0。
- 第 3 行：网络接口的 IPv6 接口信息。
- 第 4 行：网络接口状态。UP 表示接口是启动状态，最大传输单元（MTU）为 1500。各种网络对一次传输数据帧的最大长度都有一个限制，这个特性一般体现在网络接口层的数据链路子层。如果网络层有一个数据报要传，而且数据的长度比数据链路层的 MTU 还大，那么 IP 层就需要进行分片，把数据分成若干片，这样每一片都小于 MTU。Metric 是度量值，供某些操作系统所用，用于计算一条路由的成本。
- 第 5 行：接收数据包统计值，主要包括总包数、错误包数（errors）、冲突包数（collisions）。共接收到 988 个数据包，无错误。
- 第 6 行：发送数据包统计值。共发送出 231 个数据包，无错误。
- 第 7 行：数据包无冲突。该网络接口的发送队列长度为 1000。
- 第 8 行：网络设备的中断调用号和基地址。
- 第 9 行：网络接口 lo。在 Linux 系统下，用接口名称 lo 表示环回接口（Loopback Interface）。环回接口允许运行在同一台主机上的客户程序和服务器程序通过 TCP/IP 进行通信。A 类网络 127 就是为环回接口预留的。根据惯例，大多数系统把 I P 地址 127.0.0.1 分配给这个接口，并命名为 localhost。
- 第 10 行～第 14 行：环回接口的详细参数，参数的意义请见 eth0 的解释。

19.2.4　端口

IP 地址分配后，要实现网络节点间通信，还需要端口号。端口号用于标识同一主机上不同的应用程序。一台拥有 IP 地址的主机可以提供许多网络服务，如 Web 服务、ftp 服务、telnet

服务等，这些服务完全可以通过 1 个 IP 地址来实现。

那么，主机怎样区分不同的网络服务呢？显然不能只靠 IP 地址，因为 IP 地址与网络服务的关系是一对多的关系。实际上是通过“IP 地址+端口号”来区分不同的网络服务的。如果把 IP 地址比作一间房子，端口就是出入房子的门。真正的房子只有几个门，但是一个 IP 地址可以有 65535 个端口。

如读者熟悉的 ftp、telnet 分别是在端口 21 及 23 提供网络服务。端口号是 16 个二进制位的整形数，其取值范围为 1～65535。为实现系统提供的公用网络服务，1～1023 间的端口号由系统保留，这部分端口号称为知名端口号。而用户自己编写的网络应用程序应该在 1024 以后的端口号中选择使用。

在 TCP/IP 协议中，与 TCP 协议和 UDP 协议相对应，端口主要有两种：TCP 端口和 UDP 端口。由于 TCP 和 UDP 两个协议是独立的，因此各自的端口号也相互独立，如 TCP 有 235 端口，UDP 也可以有 235 端口，两者并不冲突。

提示：Linux 系统下与网络相关的配置文件主要有两个：/etc/hosts 及/etc/services。/etc/hosts 是主机名文件，该文件中存放的是一组 IP 地址与主机名的列表，如果在该列表中指出某台主机的 IP 地址，那么访问该主机时将无需进行 DNS 解析。这个文件的作用是进行域名解析，在下一章中我们将介绍通过 gethostbyname 根据机器名称获取 IP 地址。

19.3 客户机/服务器模型

客户机/服务器结构是一种基于信息的组合式结构，意在共享大型机的资源。客户机被定义为一种服务请求器，服务器被定义为一种服务供应商。单个机器既可以作为客户机也可以作为服务器，这主要取决于软件配置及执行功能类型。

在网络环境中，客户机/服务器模型提供了一种便利途径，以实现有效分布在不同位置的程序之间的相互连接。客户机/服务器模型已成为网络计算机的主要观念之一，目前所使用的大多数商业应用程序使用的都是客户机/服务器模型。

Internet 中的核心技术 TCP/IP 为客户机和服务器间的通信提供了组织结构。目前在商业应用中，采用这种结构的系统应用非常广泛。当然，随着技术的发展，目前正逐渐向三层次甚至多层次方向发展。典型的客户机/服务器模型如图 19-11 所示。

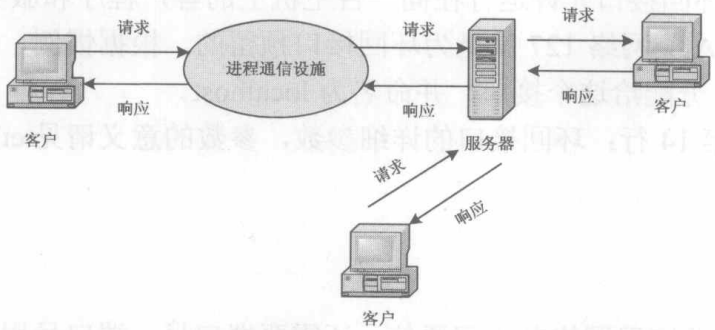


图 19-11 客户机/服务器模型

在通常客户机/服务器模型中，服务器是活性的，处于等待客户机请求状态。典型的多客户机程序共享通用服务器程序服务。客户机程序和服务器程序都是大型程序或应用的一部分。例如，相对于 Internet 而言，Web 浏览器是一种客户机程序，向 Web 服务器请求网络中其他计算机上的某种服务。

（1）重复服务器的工作流程如下。

1）等待一个客户请求的到来。

2）处理客户请求。

3）发送响应给发送请求的客户。

4）返回步骤（1）。

重复服务器主要的问题发生在 I2 状态。这时，它不能为其他客户机提供服务。也就是说，重复服务器在同一时刻只能处理一个客户的请求。典型的重复服务器模型如图 19-12 所示。

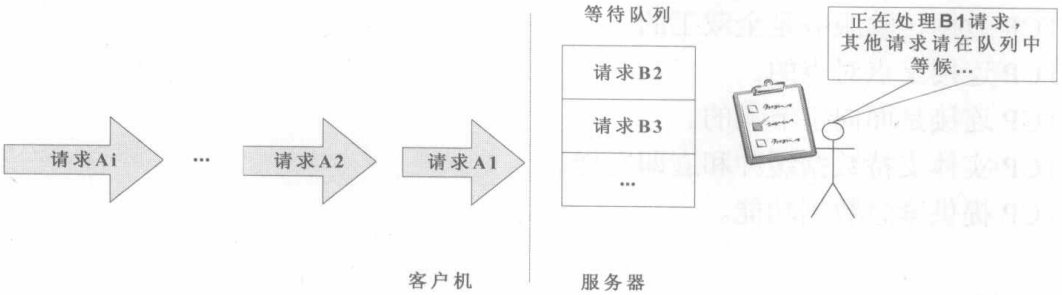


图 19-12　重复服务器模型

（2）相应地，并发型服务器的工作流程如下。

1）创建主服务器（Master Server），并等待一个客户请求的到来。

2）启动一个新的服务器来处理这个客户的请求。在这期间可能生成一个新的进程、任务或线程，并依赖底层操作系统的支持。新生成的服务器也称为从服务器（Slave Server）。生成的从服务器对客户的全部请求进行处理。处理结束后，终止这个新服务器。

3）返回步骤（1）。

并发服务器的优点在于它是利用生成其他服务器的方法来处理客户的请求。也就是说，每个客户都有它自己对应的服务器处理进程。如果操作系统允许多任务，那么就可以同时为多个客户服务。典型的并发服务器模型如图 19-13 所示。

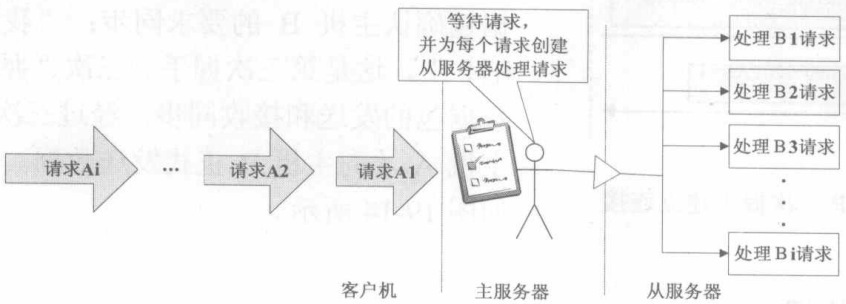


图 19-13　并发服务器模型

重复服务器的优点是消耗资源较低，但也有明显的缺点，那就是同一时刻只能处理一个请求，实时性差。而并发服务器优点在于实时性强，缺点是在高并发的情况下，对系统的资

源消耗较大。在实际应用过程中，采用哪一种服务器模型取决于应用的需要。一般来说，UDP 协议的应用可以采用重复服务器，而 TCP 协议的应用由于对于实时性要求较高，大多采用并发服务器模型。

19.4 传输控制协议

传输控制协议（Transmission Control Protocol，TCP）的主要作用是在不可靠的网络服务上为应用层提供面向连接的、端到端的可靠字节流服务。TCP 是一种面向连接的运输协议，在进行数据传输时首先必须建立一条运输连接，数据传输完成之后释放连接。TCP 协议具有以下特点。

- TCP 数据传输服务是全双工的。
- TCP 连接是点对点的。
- TCP 连接是面向字节流的。
- TCP 实体支持数据缓冲和立即发送。
- TCP 提供紧急数据功能。

19.4.1 连接建立

TCP 是基于连接的协议，在正式收发数据前，必须和对方建立可靠的连接。这一过程与平时打电话很相似，先拨号，等待对方摘机说“喂”，然后才说明是谁。一个 TCP 连接必须要经过三次“握手”才能建立起来，其中的过程非常复杂，这里只做简单、形象地介绍，读者只要做到能够理解这个过程即可。

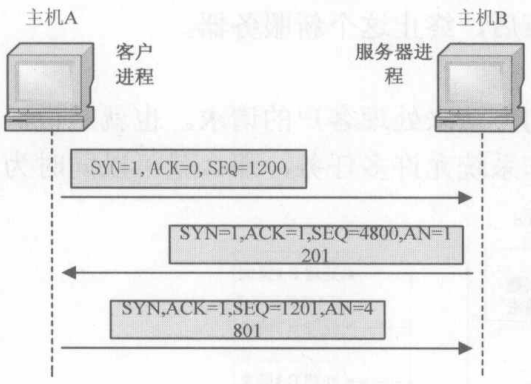


图 19-14 TCP 三次握手建立连接

用户来看看这三次对话的简单过程。主机 A 向主机 B 发出连接请求数据包：“我想给你发数据，不知道是否可以？”，这是第一次握手。主机 B 向主机 A 发送同意连接和要求同步的数据包：“可以。你什么时候发？”，这是第二次握手。主机 A 再发出一个数据包确认主机 B 的要求同步：“我现在就发，你接吧！”，这是第三次握手。三次“握手”的目的是使数据包的发送和接收同步，经过三次“握手”之后，主机 A 才向主机 B 正式发送数据。三次握手的过程如图 19-14 所示。

19.4.2 连接关闭

数据传输完成后，任何一方都可以关闭一个 TCP 连接。关闭连接的过程可以形象地称之为四次“握手”。这是由于 TC 连接是全双工（即数据在两个方向上能同时传递），因此每

个方向必须单独地进行关闭。

还是用建立连接的例子，假设主机 A 要关闭当前连接，下面是这四次握手的过程。主机 A 向主机 B 发出关闭连接请求数据包:“我想关闭连接，不知道是否可以？”，这是第一次握手。此时，主机 A 将不能再向 B 发送数据，但是可以从 B 接收数据。主机 B 向主机 A 发送同意连接和要求同步的数据包:“可以。”，这是第二次握手。主机 B 再向主机 A 发出一个关闭连接请求:“我想关闭连接，不知道是否可以？”，这是第三次握手；此时，主机 B 不能再向 A 发送数据。主机 A 再发出一个数据包确认主机 B 的关闭要求:“可以。”，这是第四次握手。

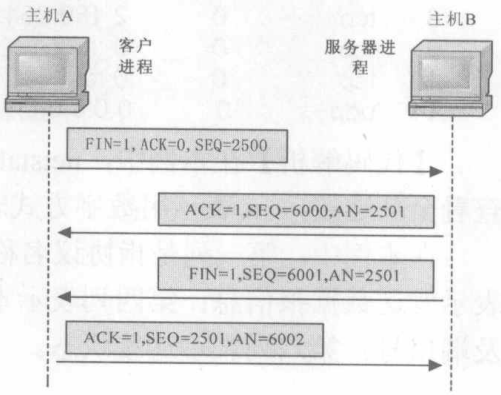


图 19-15　TCP 四次握手关闭连接

四次握手后，连接进入 TIME_WAIT 状态，连接正式关闭。四次握手的过程如图 19-15 所示。

在 TCP 建立连接及关闭连接的过程中，TCP 连接的状态是不断发生变化的。如图 19-16 所示展示了 TCP 连接状态变化的详细情况。

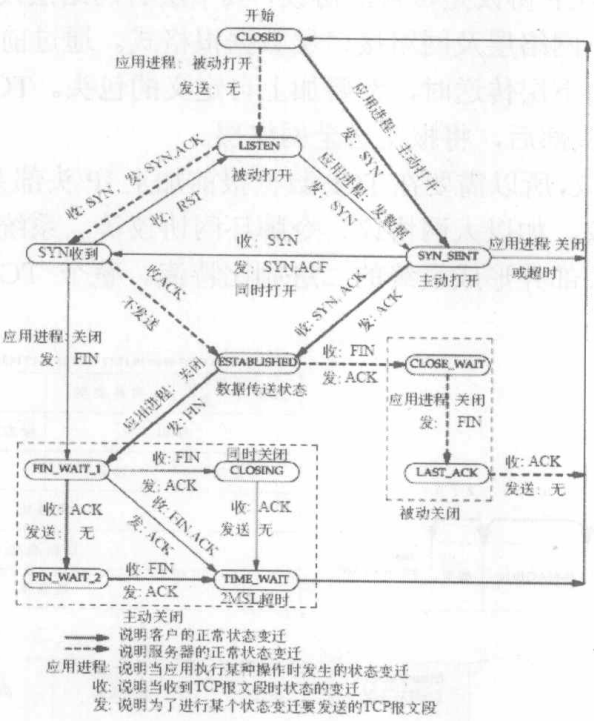


图 19-16　TCP 连接状态

【范例 19-2】查看系统的当前网络连接情况。其实现过程如示例代码 19-2 所示。

示例代码 19-2

```
1    netstat –an                                    #运行 netstat 命令
```

【运行结果】在 shell 下运行上述脚本（需要超级用户权限），其结果如下所示。

```
1    Active Internet connections (servers and established)
2    Proto Recv-Q Send-Q Local Address              Foreign Address          State
```

```
3    tcp        0      0 0.0.0.0:111             0.0.0.0:*                   LISTEN
4    tcp        0      0 0.0.0.0:23              0.0.0.0:*                   LISTEN
5    tcp        0      0 127.0.0.1:25            0.0.0.0:*                   LISTEN
6    tcp        0      0 169.254.215.193:23      169.254.215.193:32770       ESTABLISHED
7    tcp        0      0 169.254.215.193:32770   169.254.215.193:23          ESTABLISHED
8    tcp        0      2 169.254.215.193:23      169.254.215.192:16697       ESTABLISHED
9    tcp        0      0 :::22                   :::*                        LISTEN
10   tcp        0      0 ::1:25                  :::*                        LISTEN
11   udp        0      0 0.0.0.0:111             0.0.0.0:*
```

【代码解析】在本例中，netstat 是 linux 系统下用来查看网络连接信息的命令。-a 表明查看全部连接，-n 表示用数字方式表示（地址用 IP 地址表示而不用主机名称）。

在本例中，第一列是指协议名称：TCP 或者 UDP；第二列表示接收数据报信息；第三列表示发送数据报信息；第四列表示本地绑定 IP 地址及端口号；第五列表示对方绑定 IP 地址及端口号；第六列表示连接状态。

19.4.3 TCP 数据报格式

在 TCP/IP 协议中，TCP 协议是传输层协议，其下层有网络层及网络接口层。在了解 TCP 报文格式前，先介绍一下网络层及网络接口层数据报格式。通过前面的学习可以知道，协议是分层的。每层协议在向下层传送时，都要加上自定义的包头。TCP 接收到来自应用层的数据包，在前面填充 TCP 头部后，将报文送至网络层。

网络层协议是 IP 协议，所以需要在 TCP 数据报前加上 IP 头部并向下传输至网络接口层。网络接口层存在多种协议，如以太网协议、令牌环网协议等。系统将根据不同的协议类型，在 IP 报文前填充相应头部并形成最终的二进制比特流。整个 TCP 协议的数据报格式如图 19-17 所示。

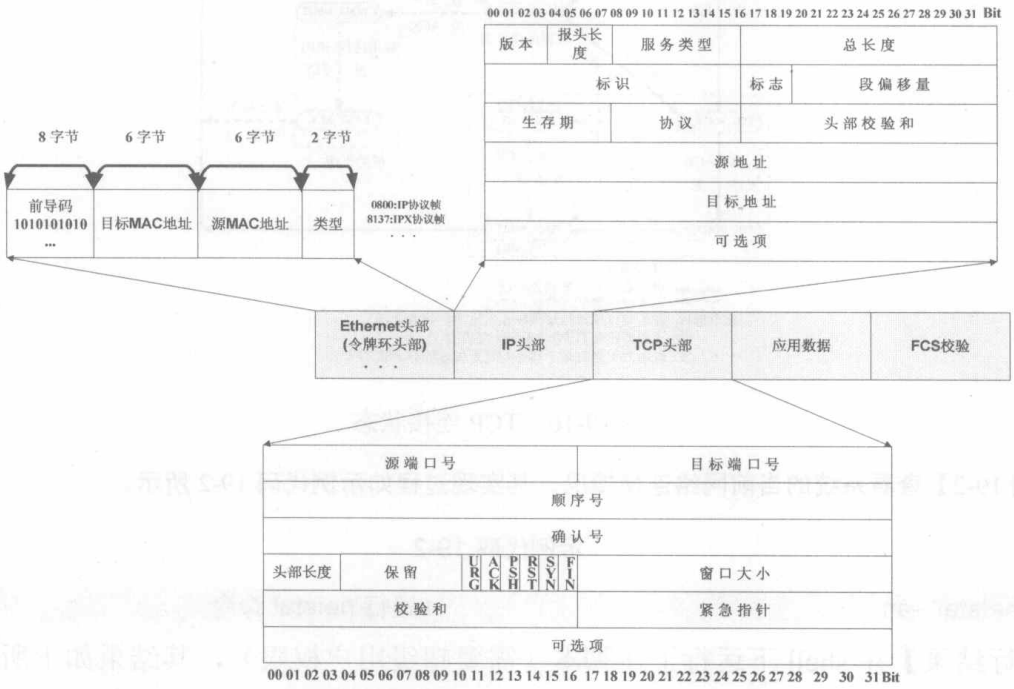


图 19-17 TCP 协议的数据报格式

对于 IP 头部各字段的解释，在此不进行详细描述。通过对 TCP 数据报格式的分析可以知道，在以太网中，最大传输单元 MTU 为 1500 个字节，在一个 IP 包中，去除 IP 包头的 20 个字节，可以传输的最大数据长度为 1480 个字节。在 TCP 包中，去除 20 个字节的 TCP 包头，可以传输的最大数据段为 1460 个字节。因此，当应用层数据超过最大数据长度时，将对该数据进行分片处理，在 IP 包头中会看到有多个片在传输，但标识号是相同的，表示是同一个数据包。

19.5　用户数据报协议

UDP 协议是一种面向无连接的协议。“面向非连接”就是在正式通信前不必与对方先建立连接，不管对方状态就直接发送。这与发送手机短信或者打电报非常相似，你在发短信的时候，只需要输入对方手机号就可以。

UDP 适用于一次只传送少量数据、对可靠性要求不高的应用环境。例如用户经常通过执行“ping”命令来测试两台主机之间通信是否正常，其实“ping”命令的原理就是向对方主机发送 UDP 数据包，然后对方主机确认收到数据包，如果数据包是否到达的消息及时反馈回来，那么网络就是通的。

因为 UDP 协议没有连接的过程，所以它的通信效率高，但也正因为如此，它的可靠性不如 TCP 协议。要保证可靠的数据传输，就需要在上层（应用层）进行可靠性处理。UDP 数据报的格式如图 19-18 所示。

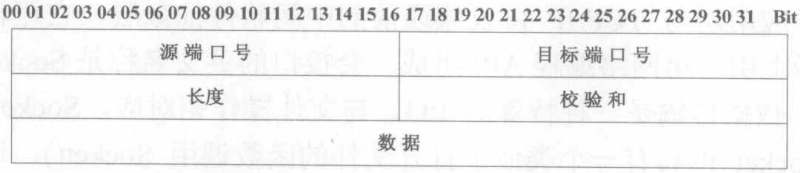


图 19-18　UDP 数据报格式

虽然 UDP 是一种不可靠的传输协议，在有些情况下 UDP 协议可能会变得非常有用。因为 UDP 具有 TCP 无法比拟的速度优势。TCP 协议中植入了各种安全保障功能，但是在实际执行的过程中会占用大量的系统开销，无疑使速度受到严重的影响。

19.6　小结

Linux 提供了功能强大的网络环境，其中 TCP/IP 是最为重要的通讯协议。本章就 TCP/IP 的基础知识进行了介绍，目的是为下一章进行网络编程打好基础。根据 Linux 网络编程的要求，本章主要讲解了该协议的分层模型和端口知识，同时，还对 TCP 协议的内容进行了详细的讲解，具体分析了在 TCP 协议下如何创建连接和关闭连接等。这些内容都是 Linux 网络编程的重要基础知识。

第 20 章　基本套接口编程

套接口是 Linux 系统下主要的网络编程接口。套接口最初是在 BSD 版本的 UNIX 上实现的，随着采用 BSD UNIX 计算机厂商的日益增多，套接口被广泛认可并逐渐成为事实上的工业标准。目前，几乎所有的操作系统都提供了对套接口的支持。Linux 继承实现了套接口编程接口，支持 BSD 套接口。本章首先介绍了套接口编程的基本概念，并对套接口编程接口函数进行了说明。通过本章的学习，读者应该掌握如下内容。

- 理解套接口的基本概念。特别是对半相关/全相关、地址族/协议族、主机字节序/网络字节序等概念的理解。
- 掌握套接口的数据结构，并能够根据不同的应用环境对结构进行正确赋值。
- 掌握套接口编程接口函数，能够熟练应用接口函数进行网络程序设计。
- 了解套接口选项的作用，并能够在实际应用中合理运用。
- 熟练掌握面向连接及面向无连接的网络编程模式。

20.1　套接口编程简述

套接口是位于应用层与 TCP/IP 协议族通信的中间软件抽象层，它逻辑上位于传输层与应用层之间，实际上由一组网络编程 API 组成。套接口的英文名称是 Socket，也称为插口或者套接字。Socket 数据传输是一种特殊的 I/O。与文件操作相对应，Socket 也可以看作是一种文件描述符。Socket 也具有一个类似于打开文件的函数调用 Socket()，该函数返回一个整型的 Socket 描述符，随后的连接建立、数据传输等操作都是通过该 Socket 实现的。套接口在网络层次中的逻辑位置如图 20-1 所示。

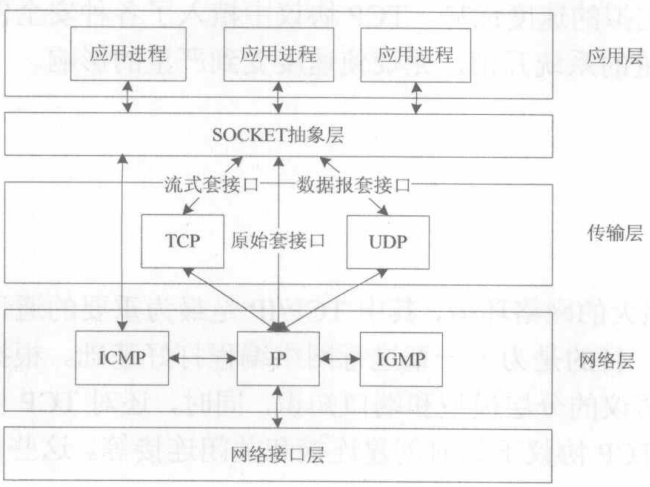


图 20-1　套接口在网络层次中的位置

20.1.1　半相关与全相关

在网络通信模型中，一个连接一旦建立，则必然包括以下要素：协议、本地地址、本地端口号、远端地址、远端端口号。这里的协议是指通讯协议，如 TCP 协议或者 UDP 协议等。本地地址及本地端口号定义网络连接的本地参数，而远端地址及远端端口号定义了通讯对端的连接参数。这样的一组要素，称其为“五元组”，又称为“全相关”。

在上述五元组中，协议、本地地址和本地端口号，这三项要素惟一地标识了网络连接的本地进程，而协议、远端地址、远端端口号则惟一地标识了网络连接的对端进程。这三项要素又称为“三元组”。由于三元组指定了一个完整的网络连接的半部分，所以称为“半相关”。全相关与半相关如图 20-2 所示。

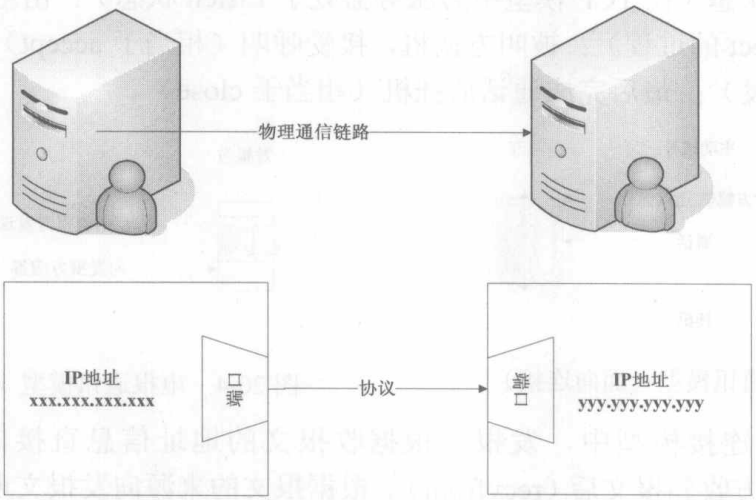


图 20-2　全相关与半相关

20.1.2　地址族与协议族

Linux 套接口支持多种协议族。协议族也称为域，不同的协议族定义了不同的通信环境。Linux 支持的协议族及协议族的定义包含在头文件<bits/socket.h>中，本书中介绍的网络编程使用 AF_INET 协议族。常见的协议族及其作用的环境如下所示。

```
PF_UNIX/PF_LOCAL/PF_FILE：用于主机内进程间通信
PF_INET:Ipv4 网络通信协议，用于远程主机间通信
PF_INET6:用于 Ipv6 网络通信
PF_IPX:用于 Novell IPX 网络通信
PF_X25:用于 ITU-T X.25/ISO-8208 网络通信
```

根据套接口规范的定义，一个协议族可以支持多个地址族。但是，到目前为止的套接口实现中，全部都是一个协议族支持一个地址族。因此，在各个版本的 Linux 系统中，都把协议族与地址族定义为同一个值。在实际的编程过程中，考虑到将来的可移植性，在调用创建套接口的函数时，应该使用 PF 系列宏定义。而在类似地址绑定的系统调用中，应该使用 AP 系列宏定义。

20.1.3 面向连接与面向无连接

在套接口编程模型中，存在面向连接的服务与面向无连接的服务。面向连接的服务类似于电话系统，如图 20-3 所示。在拨打电话的过程中，首先需要拨号，电话拨通后双方进行通话，通话完毕后挂断电话。与此相似，在面向连接的服务模式下，每一次完整的数据传输都要经过建立连接、使用连接、终止连接的过程。无连接服务则类似于电报系统，如图 20-4 所示。在发送电报的过程中，并不需要先与目的地建立连接，而是根据目标地址直接发送报文即可。同样，在面向无连接的服务模式下，只需要知道通信对端的信息，即可直接向目标发送数据。

在上述面向连接的模型中，被叫方相当于 TCP 服务器，主叫方相当于客户机。首先，被叫方处于等待呼叫状态（在 TCP 模型中为服务器处于 Listen 状态）；由主叫方拨号，呼叫被叫方（相当于 connect 的过程）；被叫方摘机，接受呼叫（相当于 accept）过程，然后双方通话（相当于数据收发）；最后完成通话后挂机（相当于 close）。

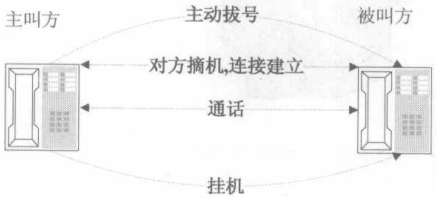


图 20-3 电话通讯模型（面向连接）

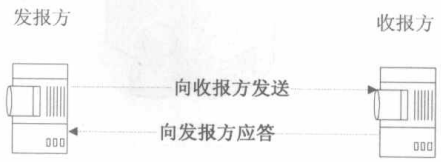


图 20-4 电报通讯模型（面向无连接）

在上述面向无连接模型中，发报文根据收报文的地址信息直接向收报文发送报文（sendto）。而收报方收到报文后（recvfrom），根据报文的来源向发报文返回报文（sendto），发报文则接收该响应报文（recvfrom）。

20.1.4 套接口类型

在创建套接口时，除需要指定协议族外，还需要指定套接口的类型。Linux 支持多种套接口类型，主要包括流式套接口、数据报套接口和原始套接口。

- 流式套接口（SOCK_STREAM）：提供面向连接、可靠的全双工数据传输服务。流式套接口通过 TCP 协议实现。而 TCP 的差错控制机制可以保证数据无差错、无重复地发送，且按发送顺序接收。另外，流式套接口有流量控制机制，可以有效避免数据流超限。telnet、ftp 等应用采用的就是流式套接口。
- 数据报式套接口（SOCK_DGRAM）：提供无连接服务。数据报套接口通过 UDP 协议实现。数据发送时并不经过建立连接的过程，而是直接以数据包形式被发送。另外，数据报套接口不能保证传输过程中的正确性，并且数据接收顺序可能与发送顺序不一致。传输内容的正确性及数据顺序要靠应用层编码进行控制。
- 原始式套接口（SOCK_RAW）：该接口允许对较低层协议（如 IP、ICMP）进行直接访问。在某些应用中，使用原始套接口可以构建自定义头部信息的 IP 报文。创建

原始套接口需要超级用户权限。

20.1.5　字节序

字节序是指占内存大于一个字节的类型的数据在内存中的存储顺序，按照不同的顺序可以划分为小端字节序（Little edian）、大端字节序（Big edian）两种，统称为主机字节序。

小端字节序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；大端字节序是高字节数据存放在低地址处，低字节数据存放在高地址处。字节序是与计算机硬件的种类相关的，基于 Intel X86 体系结构的 PC 机是小端字节序的，而有些平台则是大端字节序的。如图 20-5 所示，展示了 int 型数据 0x0001 在大字节序主机和小字节序主机的内存中的存储映像。

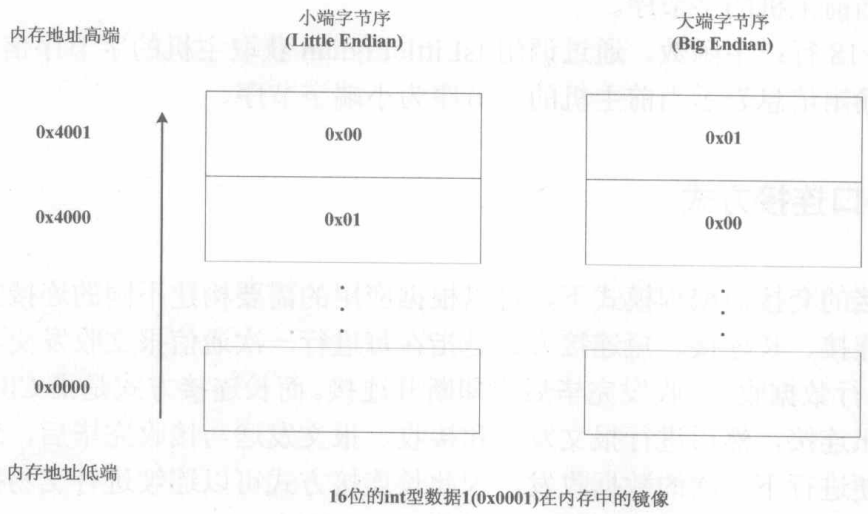


图 20-5　小字节序和大字节序存储映像

【范例 20-1】查看当前系统的字节序的过程。其实现过程如示例代码 20-1 所示。

示例代码 20-1

```
#include <stdio.h>                                          /*头文件*/
#include <stdlib.h>
int IsLittleEndian()                                        /*定义函数，判断字节序*/
{
    unsigned short i=1;                                     /*定义无符号短整型变量*/
    return (1 == *((char *)&i));                            /*取变量 i 的低 8 位，与 1
比较,如果相等，则为小字节序；如果不等，则为大字节序*/
}
main()                                                      /*主函数*/
{
    if(IsLittleEndian())                                    /*调用函数判断字节序*/
    {
        printf("Little Endian.\n");                         /*输出字节序信息*/
    }
    else
```

```
15      {
16          printf("Big Endian.\n");                    /*输出字节序信息*/
17      }
18  }
```

【运行结果】经过编译链接，在 shell 下运行上述程序，其结果如下所示。

```
19  Little Endian.
```

【代码解析】在本例中，通过函数 IsLittleEndian 判断当前主机是否采用小端字节序。根据图 20-1 可以知道，如果是小端字节序，在将 int 数据强制转换为 char 时，将取内存低端的第一个字节的值，而这个字节的值为 1 表示当前主机的字节序为小字节序，反之则为大字节序。源代码的各行解释如下。

- 第 1～2 行：头文件信息。
- 第 3～7 行：IsLittleEndian 函数体。使用强制类型转换，根据地址低端存储的数据值，判断当前主机的字节序。
- 第 8～18 行：主函数。通过调用 IsLittleEndian 获取主机的字节序信息。

该范例的输出信息表示当前主机的字节序为小端字节序。

20.1.6 套接口连接方式

在面向连接的套接口编程模式下，可以根据应用的需要构建不同的连接方式。主要的连接方式包括短连接、长连接。短连接方式是指在每进行一次通信报文收发交易时都需要先建立连接，然后进行数据收发，收发完毕后立即断开连接。而长连接方式是指 Client 方与 Server 方先建立好通讯连接，然后进行报文发送和接收。报文发送与接收完毕后，连接并不断开而继续存在，以便进行下一次的数据收发，因此长连接方式可以连续进行交易报文的发送与接收。短连接与长接连的模型分别如图 20-6 和图 20-7 所示。

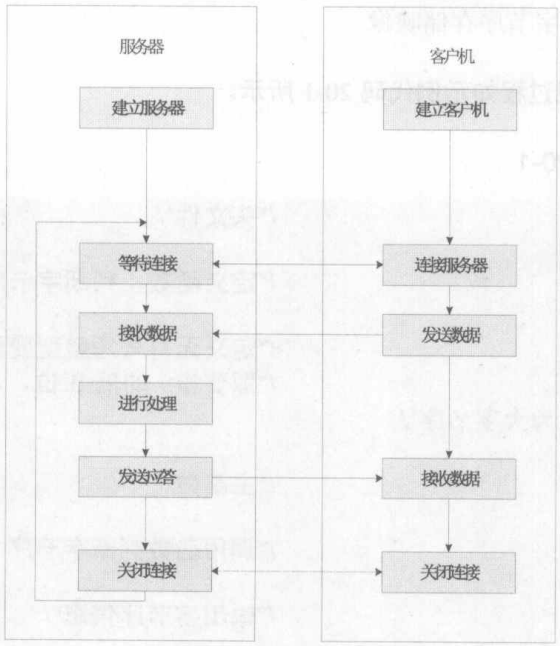


图 20-6 短连接方式

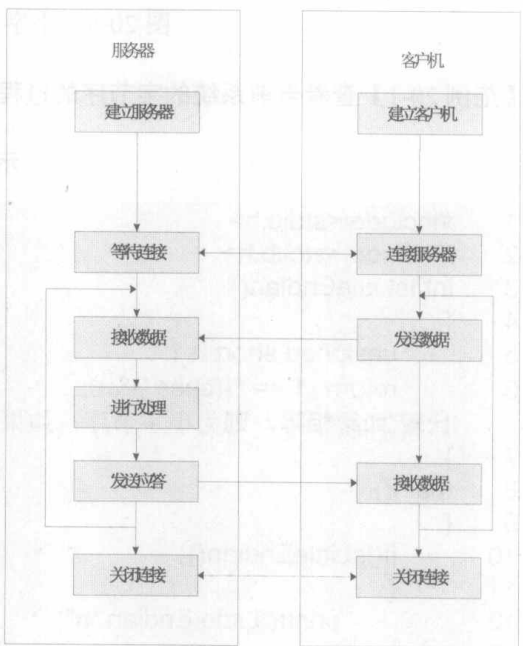


图 20-7 长连接方式

短连接与长连接都有各自适合应用的场合。与长连接相比，短连接方式的优点是方式简单，不需要过多考虑长连接方式下的网络故障异常处理，其缺点是每次通信都要有建立连接的过程，处理效率上不如长连接。而长接连的优点是处理效率高，但是需要考虑各种网络异常的处理，程序逻辑比短连接相对要复杂。在实际应用中，具体采用何种连接方式应视应用环境确定。典型的并发服务器采用的是短连接方式。如果客户机与服务器之间通信频繁，并且对传输的时间要求较高，可以考虑采用长连接方式。

20.1.7　数据传输方式

连接建立完成后，在数据发送与接收过程中也存在不同的方式，主要包括同步和异步两种方式。这里的同步与异步是指从程序的逻辑结构上看数据是同步发送或接收的。

对于同步方式来说，报文发送和接收是同步进行的，即报文发送后，发送方等待接收方处理完成并返回应答报文。同步方式需要考虑超时问题，报文发出后发送方需要设定超时时间，超时后发送方不再继续等待，而直接返回。对于异步方式来说，发送方只负责发送数据，不需要等待接收任何返回数据；而接收方只负责接收数据。通常情况下，异步方式在客户端和服务器端各有两个进程专门负责数据收发。这两个进程相互独立，互不影响。

在实际网络编程过程中，建立网络通信模型需要综合考虑连接方式及数据传输方式。比较有价值的网络通信模型包括同步短连接、同步长连接、异步长连接等。异步长连接的通信模型如图 20-8 所示。

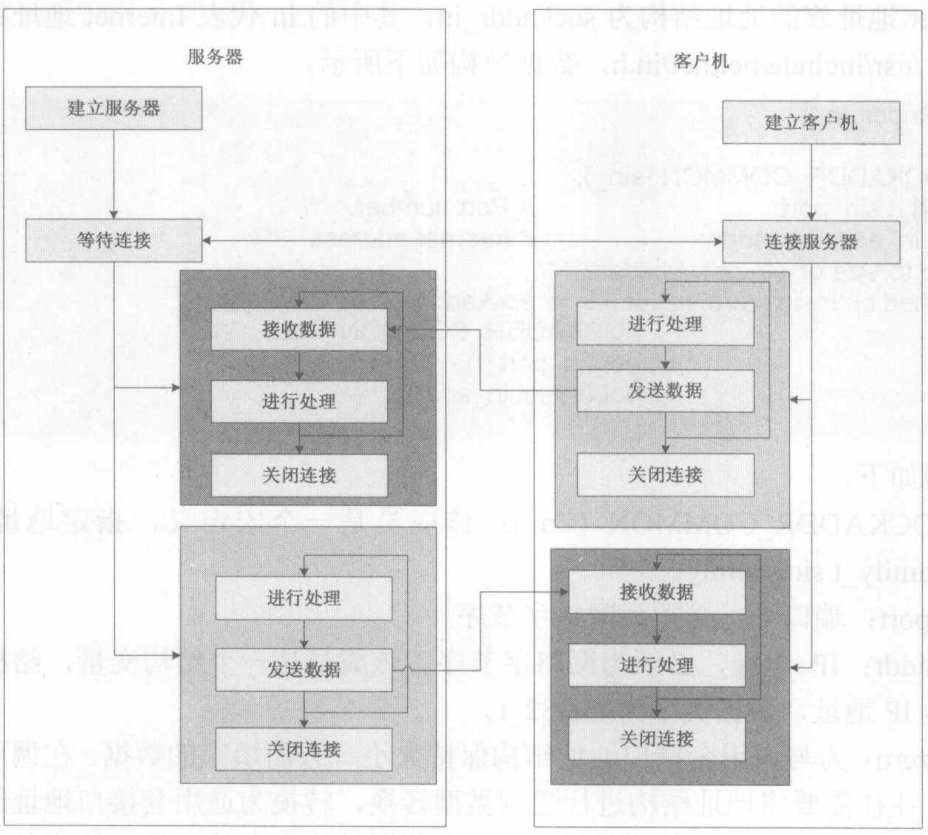


图 20-8　异步长连接通信模型

从异步长连接通信模型可以看到，异步长连接包括两条连接。通常情况下，一条连接负责发送数据，另外一条连接负责接收数据。每个通信节点实际是由两个子进程组成。每个子进程负责维持一条通信链路。异步长连接是相对较为复杂的通信模型，接收和发送分别由不同的进程完成。但是，较高的通信效率也增加了控制的复杂度。首先，数据的接收和发送是异步完成的，这就存在数据同步的问题。

20.2 套接口数据结构

在套接口编程接口函数中，定义了若干数据结构。这些数据结构大多为结构类型，基本上所有的套接口函数都会用到这些结构的内容。本节对这些结构的内容进行了详细阐述，读者可以结合后续章节中对接口函数的介绍等内容加深对这些结构的理解。

20.2.1 套接口地址结构

在套接口编程模式下，大多接口函数都需要用到套接口地址结构。套接口地址结构就是包含了套接口的地址及端口等信息的结构体。该结构在调用大多数套接口函数时是必需的。Linux 操作系统支持多种地址族的套接口，如 UNIX、INET、IPX、X25 等，而每种套接口又有其特有的地址结构。在 TCP/IP 协议下进行网络编程使用的地址族是 Internet 地址族，即 INET。Internet 地址族的地址结构为 sockaddr_in，其中的 in 代表 Internet 地址族。该地址结构的定义位于/usr/include/netinet/in.h，数据结构如下所示。

```
struct sockaddr_in
  {
    __SOCKADDR_COMMON (sin_);
    in_port_t sin_port;                    /* Port number.   */
    struct in_addr sin_addr;               /* Internet address.   */
    /* Pad to size of 'struct sockaddr'.   */
    unsigned char sin_zero[sizeof (struct sockaddr) -
                           __SOCKADDR_COMMON_SIZE -
                           sizeof (in_port_t) -
                           sizeof (struct in_addr)];
  };
```

参数说明如下。

- __SOCKADDR_COMMON (sin_)：该成员是一个宏定义，指定地址族，等价于 sa_family_t sin_family。
- sin_port：端口号，必须为网络字节序。
- sin_addr：IP 地址，必须为网络字节序。该成员是一个结构变量，结构中惟一的成员是 IP 地址，数据类型为 uint32_t。
- sin_zero：为与通用套接口地址结构保持大小一致而填充的数据。在调用套接口编程时，往往需要将地址结构进行强制类型转换，转换为通用套接口地址结构进行传递参数。为满足这一要求，需要保持两个数据结构大小的一致。该成员内容应设置为

空字符“\0”。

20.2.2　通用套接口地址结构

Linux 支持多种不同的地址族，每种地址族的结构内容是各不相同的，例如，internet 地址族的地址结构是 sockaddr_in，而 UNIX 地址族的地址结构是 sockaddr_un 等。在向套接口的编程接口函数传递地址结构指针时，需要将各不相同的地址结构转换为一个通用的数据结构，这就是通用套接口地址结构。结构在<bits/socket.h>中定义，声明如下所示。

```
struct sockaddr
  {
    __SOCKADDR_COMMON (sa_);      /* Common data: address family and length.  */
    char sa_data[14];             /* Address data.  */
  };
```

参数说明如下。

- __SOCKADDR_COMMON (sa_)：该成员是一个宏定义，指定地址族，等价于 sa_family_t sa_family。
- sa_data：地址数据。对于 Internet 地址族来说，就是包括 sin_port、sin_addr、sin_zero 在内的全部地址数据。

20.2.3　主机名称数据结构

在进行网络编程时，通常需要用到主机名称（域名）与 IP 地址转换。在这种情况下，就需要使用主机名称数据结构 hostent，该数据结构定义了主机名与 IP 地址的对应关系。在套接口编程模型中，与地址绑定相关的操作都需要使用该结构。该结构定义包括在头文件 netdb.h 中，其声明如下所示。

```
struct hostent
{
  char *h_name;                 /* Official name of host.  */
  char **h_aliases;             /* Alias list.  */
  int h_addrtype;               /* Host address type.  */
  int h_length;                 /* Length of address.  */
  char **h_addr_list;           /* List of addresses from name server.  */
#define h_addr  h_addr_list[0]  /* Address,  for backward compatibility.  */
};
```

参数说明如下。

- h_name：主机名称。
- h_aliases：主机别名列表。主机别名可能存在多个，该成员为指向别名列表的指针。
- h_addrtype：主机地址类型。在 Internet 地址族下，该值一般为 AF_INET。
- h_length：地址的字节长度。
- h_addr_list ：一个以 0 结尾的数组，包含该主机的所有地址。
- h_addr：在 h_addr_list 中的第一个地址。

用于获取该结构的接口函数有 gethostbyname、gethostbyaddr 及 gethostent 等。其中，

gethostbyname 用于把主机名映射成 IP 地址，而 gethostbyaddr 系统调用的作用则相反。

【范例 20-2】查看当前系统的主机名称及 IP 地址。其实现过程如示例代码 20-2 所示。

示例代码 20-2

```
#include <stdio.h>                                                    /*头文件*/
#include <stdlib.h>
#include <netdb.h>
#include <unistd.h>
#include <errno.h>
main()                                                                /*主函数*/
{
    int n;                                                            /*循环变量*/
    struct hostent *h;                                                /*结构指针变量定义*/
    char **p;                                                         /*定义指向字符串的指针*/
    char hostname[PATH_MAX];                                          /*定义字符串数组*/
    if(gethostname(hostname,PATH_MAX) < 0)                            /*得到机器名*/
    {
        fprintf(stderr,"gethostname failed:%s\n", strerror(errno));   /*输出错误信息*/
        exit(0);
    }
    if((h = gethostbyname(hostname)) == NULL)                         /*根据机器名解析地址*/
    {
        fprintf(stderr," gethostbyname failed:%s\n",strerror(errno)); /*输出错误信息*/
        exit(0);
    }
    fprintf(stderr,"Host name:%s\n",h->h_name);                       /*输出主机名*/
    for(n=0,p=h->h_aliases; *p != NULL; p++,n++)                      /*循环输出所有主机别名*/
    {
        fprintf(stderr,"Alias name %d:%s\n",n+1,*p);                  /*输出别名*/
    }
    for(n=0; n< h->h_length/sizeof(int); n++)                         /*循环输出所有的 IP 地址*/
    {
        fprintf(stderr,"Ip address %d:%s\n",n+1,inet_ntoa(*((struct in_addr *)(h
->h_addr_list[n])));                                                  /*解析 IP 地址并输出*/
    }
    return;
}
```

【运行结果】经过编译链接，在 shell 下运行上述程序，其结果如下所示。

```
Host name:Hubery.site
Alias name 1:Hubery
Ip address 1:192.168.1.8
```

【代码解析】在本例中，首先通过系统调用 gethostname 得到当前主机的主机名，然后根据主机名调用 gethostbyname 解析/etc/hosts，得到当前主机的详细信息。源代码的各行解释如下。

- 第 1～5 行：头文件信息。其中 netdb.h 中包含了 gethostbyname 系统调用的原型声明。
- 第 12 行：调用 gethostname 获取当前主机的主机名，并保存至局部变量 hostname 中。
- 第 17 行：调用 gethostbyname 获取主机名的详细信息，返回数据存放于 struct hostent 类型的结构指针中。

- 第 23～26 行：循环输出主机的别名信息。
- 第 27～31 行：循环输出主机的 IP 地址信息。其中系统调用 inet_ntoa 的作用是将地址数据转换为可读性更好的点分十进制字符串。

其输出信息的含义具体如下。

- 第 1 行：主机名是 Hubery.site。
- 第 2 行：主机有一个别名 Hubery。
- 第 3 行：主机定义了一个 IP 地址 192.168.1.8。

提示：在 Linux 系统中，主机名称与 IP 地址的对照关系是在文件/etc/hosts 中体现的。系统调用 gethostbyname 及 gethostbyaddr 正是对该文件进行解析获取 hostent 结构的。一般情况下 hosts 文件的每行为一个主机，每行由三节组成，每个节之间由空格隔开。在 Internet 系统下，该文件还起到域名解析的作用。域名解析通过需要通过 DNS 服务器进行，但如果在该文件中对域名进行了定义，则将不进行 DNS 解析，直接使用该文件中指定的 IP 进行访问。

20.2.4　服务名称数据结构

在 Linux 系统中存在一个网络服务配置文件/etc/services，该文件中定义了当前主机提供的网络服务名称、对应的端口号和协议。文件的每一行定义了一种网络服务，每一行由 4 节组成，节间由空格分隔。其中，第一节指定了服务的名称，如 telnet；第二节指定了服务的端口及协议，端口与协议用“/”分隔；第三节定义了服务的别名。

这个文件的作用主要有两个：一是为 Linux 超级服务所使用；二是在进行网络编程使用。在进行服务器网络编程中，需要定义服务绑定的端口号。如果直接将端口号写到程序中，那么一旦端口号发生变化，则需要修改程序并重新编译。此时，可以在/etc/services 中定义服务名称。在编程时，只需通过相应系统调用根据服务名称获取对应的端口号。在端口号需要变更时，只需要修改/etc/services 文件即可。

Linux 系统提供了对/etc/services 文件进行操作的一系列函数，如 getservbyname、getservbyport 等。这些函数都需要使用服务名称数据结构，即 struct servent。该结构的定义在头文件 netdb.h 中，声明如下所示。

```
struct servent
{
  char *s_name;                 /*Official service name*/
  char **s_aliases;             /*Alias list*/
  int s_port;                   /*Port number*/
  char *s_proto;                /*Protocol to use*/
};
```

参数说明如下。

- s_name：服务名称，如通常的 telnet、ftp 等。
- s_aliases：服务的别名列表。
- s_port：服务的端口号，如 telnet 定义在 23 端口，而 ftp 定义在 21 端口。
- s_proto：协议名称，如 TCP、UDP 等。

【范例 20-3】查看当前系统中 telnet 服务绑定的端口号。其实现过程如示例代码 20-3 所示。

示例代码 20-3

```
01  #include <stdio.h>                                          /*头文件*/
02  #include <stdlib.h>
03  #include <netdb.h>
04  #include <unistd.h>
05  #include <errno.h>
06  main()                                                      /*主函数*/
07  {
08      struct servent *s;                                      /*结构指针变量定义*/
09      if((s = getservbyname("telnet","tcp")) == NULL)         /*根据服务名称得到服务结构*/
10      {
11          fprintf(stderr,"getservbyname failed:%s\n",strerror(errno));   /*输出错误信息*/
12          exit(0);
13      }
14      fprintf(stderr,"Port:%d\n",ntohs(s->s_port));           /*输出该服务所对应的端口*/
15      return;
16  }
```

【运行结果】经过编译链接，在 shell 下运行上述程序，其结果如下所示。

```
01  Port:23
```

【代码解析】在本例中，首先通过系统调用 getservbyname 得到 telnet 服务的服务名称数据结构，然后根据调用 ntohs 进行网络字节序至主机字节序的转换，并输出端口号。源代码的各行解释如下。

- 第 1～5 行：头文件信息，其中 netdb.h 中包含了 getservbyname 系统调用的原型声明。
- 第 9 行：调用 getservbyname，根据服务名称及协议获取服务名称数据结构，返回指向 struct servent 结构的指针。
- 第 14 行：进行网络字节序至主机字节序的转换，并输出转换后的端口号。

输出信息的第 1 行表示 telnet 绑定的服务端口是 23。

20.2.5 通用数据收发结构

套接口编程环境下，数据的收发可以通过函数 send、recv 进行，也可以通过 sendto、recvfrom 进行。除此之外，套接口还提供了两个通用数据收发函数 sendmsg 和 recvmsg。这两个函数可以完成前面提到的函数的全部功能。在调用通用数据收发函数 sendmsg 和 recvmsg 时，需要用到一个类型为 struct msghdr 的结构，本书中称其为通用数据收发结构。该结构的定义位于头文件<bits/socket.h>中，其结构如下所示。

```
struct msghdr
  {
    void *msg_name;              /*套接口名*/
    socklen_t msg_namelen;       /*套接口地址结构（即 msg_name 参数）的尺寸*/
    struct iovec *msg_iov;       /*读取或者接收数据缓冲区结构*/
    size_t msg_iovlen;           /*msg_iov 结构数组的元素个数*/
    void *msg_control;           /*辅助数据*/
    size_t msg_controllen;       /*辅助数据的尺寸*/
    int msg_flags;               /*接收数据的标志*/
  };
```

参数说明如下。

- msg _name：套接口名。在 PF_INET 协议族中，该参数为指向套接口地址结构（struct sockaddr_in）的指针。该参数对于 sendmsg 来说是一个传入参数，对于 recvmsg 来说则是一个值-结果参数。
- msg_namelen：套接口地址结构（即 msg_name 参数）的尺寸。
- msg_iov：读取或者接收数据缓冲区结构。

其结构定义如下所示。

```
struct iovec
  {
    void *iov_base;        /*输入/输出缓冲区的指针*/
    size_t iov_len;        /*要读取或者接收的数据的长度*/
  };
```

- msg_iovlen：msg_iov 结构数组的元素个数。
- msg_control：辅助数据。
- msg_controllen：辅助数据的尺寸。该参数对于 recvmsg 函数来说，是一个值-结果参数。
- msg_flags：标志。该参数只对 recvmsg 函数有效。如果在调用 sendmsg 时要使用标志值，需要使用 sendmsg 函数的参数 flags，而不是 msghdr 结构中的 msg_flags。

20.3　基本套接口函数

套接口是 TCP/IP 协议在传输层之上向应用编程人员提供的网络接口。该接口面向开发人员提供了大量接口函数。这些函数大致可以划分为字节操作函数、字节序操作函数、地址转换函数及基本套接口编程函数等。本节将结合若干实例就这部分函数的内容向读者进行详细介绍。

20.3.1　字节操作函数

在网络编程中，经常使用各种不同的数据类型。为方便对这些数据进行操作，Linux 提供了两组函数。一组是以字母 b 开头的 bzero、bcopy，bcmp 等函数；另外一组是以 mem 开头的内存操作函数。这两组函数在头文件<string.h>中进行声明，其函数原型如下所示。

```
#include <string.h>
void bzero (void *__s,size_t __n);
void bcopy (__const void *__src, void *__dest, size_t __n);
int bcmp (__const void *__s1, __const void *__s2, size_t __n);
void *memset (void *__s, int __c, size_t __n);
void *memcpy (void *__restrict __dest,__const void *__restrict __src, size_t __n);
int memcmp (__const void *__s1, __const void *__s2, size_t __n);
```

在上述函数中，bzero 用于将缓存区中指定长度的内容清空。通常该函数被用来初始化套接口相关的地址结构。该功能也可以用 memset 替代实现。不过，memset 除将缓存区内容清空外，还可以将缓存区内容设置为其他值，而不仅仅是“\0”。bcopy 用于从源参数向目标参

数复制指定长度的数据。该功能同样可由 memcpy 替代实现。不过，bcopy 在源参数与目标参数相同时，可以很好地工作，而 memcpy 则将出现不可预知的结果。

在实际使用过程中，请读者根据需要合理选择使用。bcmp 和 memcmp 是用于比较两个缓存区内容的差异。如果二个缓存区内容相同，则返回 0；如果不相同，则返回值根据两个参数不相同的第一个字节的内容进行确定。如果参数 s1 的相应字节值大于参数 s2 的相应字节值，则返回值大于 0；反之则小于 0。

另外，请读者注意 bcopy、memcpy 和 strcpy 函数的区别。strcpy 是用于字符串拷贝的函数。注意：字符串是以空字符“\0”为结束符的。这就是说，如果拷贝的源数据中存在空字符，则 strcpy 将只能拷贝至空字符前的位置。而 bcopy 和 memcpy 则可以忽略空字符，根据指定的长度进行拷贝。

提示：在字节操作的相关函数声明中，出现了限定词 const。该限定词的含义是指该变量的值不能在函数中被修改。另外，C 语言中还有另外一个常用的限定词 volatile。volatile 限定词的含义是告诉编译器，该变量的值可能被外部事件所更改。经过 volatile 限定声明的变量，在运行过程中取值时，一定会从该变量的地址中获取。

【范例 20-4】从命令行输入两个字符串，比较两个字符串的大小。其实现过程如示例代码 20-4 所示。

示例代码 20-4

```
#include <stdio.h>                                          /*头文件*/
#include <string.h>
main(int argc,char *argv[])                                 /*主程序*/
{
    int n;                                                  /*整型变量*/
    if(argc != 3)                                           /*判断命令行参数个数*/
    {
        fprintf(stderr,"Usage: %s src1 src2\n",argv[0]);    /*输出用法信息*/
        exit(0);
    }
    if(strlen(argv[1]) > strlen(argv[2]))                   /*参数 1 的长度大于参数 2*/
    {
        printf("[%s]>[%s]\n",argv[1] ,argv[2]);             /*输出提示信息*/
    }
    else if(strlen(argv[1]) < strlen(argv[2]))              /*参数 1 的长度小于参数 2*/
    {
        printf("[%s]<[%s]\n",argv[1] ,argv[2]);             /*输出提示信息*/
    }
    else                                                    /*参数 1 的长度等于参数 2*/
    {
        n = bcmp(argv[1] ,argv[2] ,strlen(argv[1]));        /*比较两个参数的内容*/
        if(n > 0)                                           /*n>0 表明参数 1 大于参数 2*/
        {
            printf("[%s]>[%s]\n",argv[1] ,argv[2]);         /*输出提示信息*/
        }
        else if(n < 0)                                      /*n<0 表明参数 1 小于参数 2*/
        {
            printf("[%s]<[%s]\n",argv[1] ,argv[2]);         /*输出提示信息*/
        }
        else                                                /*n==0 表明参数 1 等于参数 2*/
```

```
31              {
32                  printf("[%s]=[%s]\n",argv[1] ,argv[2]);          /*输出提示信息*/
33              }
34          }
35          return;
36      }
```

【运行结果】经过编译链接，在 shell 下运行上述程序，其结果如下所示。

```
01  ./example20_4 abc abd
02  [abc]<[abd]
```

【代码解析】在本例中，首先判断输入的两个字符串长度是否相等。如果长度不等，则长度较长的字符串较大；如果长度相等，则调用 bcmp 判断两个字符串是否相等。源代码的各行解释如下。

- 第 1～2 行：头文件信息。其中 string.h 中包含了 bcmp 系统调用的原型声明。
- 第 3 行：主函数。注意：该主函数定义了命令行参数。其中，argc 表示命令行参数的个数，argv 是命令行参数的数组。argc 中参数的个数包含了程序名本身，即 argv 数组的第一个成员就是程序本身的名称。
- 第 6～10 行：判断命令行参数是否符合要求。如果不符合，则进行语法提示后退出。
- 第 11～18 行：在输入的字符串长度不相等时情况下的处理。
- 第 21 行表示：调用 bcmp 对两个字符串进行比较。
- 第 22～25 行：bcmp 返回值大于 0，表明第一个字符串大于第二个字符串。
- 第 26～29 行：bcmp 返回值小于 0，表明第一个字符串小于第二个字符串。
- 第 30～33 行：bcmp 返回值等于 0，表明两个字符串相等。

20.3.2　字节序操作函数

在 TCP/IP 协议中，规定统一采用大端字节序。这就要求在填充套接口数据结构时，要进行字节序转换。由于主机的字节序是与硬件体系结构相关的，为保证程序良好的可移植性及处理的简单化，在发送端统一将主机字节序转化为网络字节序。而在接收端，则将网络字节序转换化主机字节序。至于主机字节序到底是大端字节序还是小端字节序，在应用程序中完全可以不必关心。

为实现上述的功能，Linux 提供了一系列字节序转换函数，其中由主机字节序向网络字节序转换的包括 htons、htonl。这里的 h 就是主机（host）的意思，而 n 就是网络（network）的意思，最后的 s 表示函数转换的对象是 short 数据类型，l 表示是 long 类型。同样，由网络字节序向主机字节序转换的函数包括 ntohs 和 ntohl。这些函数的声明位于头文件<netinet/in.h>中，其原型定义如下所示。

```
#include <netinet/in.h>
uint16_t htons (uint16_t __hostshort);
uint32_t htonl (uint32_t __hostlong);
uint16_t ntohs (uint16_t __netshort);
uint32_t ntohl (uint32_t __netlong);
```

在实际编程过程中，在什么场合下使用哪种转换函数呢？读者只需要记住这样一个规

则：向套接口数据结构中赋值时，用主机字节序转换至网络字节序函数；从套接口数据结构中获取值时，用网络字节序转换为主机字节序函数。至于 short 和 long 的区别，二者分别用于 16 位和 32 位数据的类型转换。通常情况下，short 用于网络端口的转换，而 long 用于网络地址的转换。

20.3.3 地址转换函数

IP 地址是进行网络编程时必不可少的要素。在编程过程中，往往需要进行 IP 地址格式的转换。在向套接口地址结构赋值时，往往需要从可读性好的点分十进制方式转换为套接口能够识别的数据类型。而在有些情况下，又需要把套接口内部的 IP 地址数据转换为可读性好的点分十进制的形式。Linux 套接口编程接口提供了相关的转换函数供编程使用。

其中，从点分十进行制形式转换为套接口内部数据类型的有 inet_addr、inet_aton、inet_atop。inet_aton 和 inet_atop 这两个函数的区别是：inet_atop 不仅支持 IPv4 版本，而且支持 IPv6 版本。为保证程序良好的可移植性，建议使用 inet_atop 函数。inet_addr 实现了与 inet_aton 相同的功能。但是，inet_addr 在调用出错时的返回值是 INADDR_NONE，这个值恰好与 IP 地址“255.255.255.255”的进行转换后的内部数据值是相同的。

此时，程序将无法判断到底是返回失败还是进行了“255.255.255.255”地址的转换。因此，建议在应用过程中使用 inet_aton 或者 inet_atop 函数代替 inet_addr 的调用。从套接口内部数据类型转换为点分十进制的函数有 inet_ntoa、inet_ntop。这些函数的声明位于头文件<apra/inet.h>中，其原型如下所示。

（1）
```
#include <arpa/inet.h>
in_addr_t inet_aton (__const char *__cp, struct in_addr *__inp);
```

1）输入/输出参数说明如下。

- __cp：输入数据，点分十进制表示的 IP 地址或者主机名称（域名）。如果解析主机名称（/etc/hosts）或者进行域名解析失败，将返回错误。
- __inp：输出数据，指向 in_addr 结构的指针。调用成功后，该结构中存储地址数据。请注意，该地址数据是网络字节序的数据。

2）返回值说明如下。

- 返回 0：表示调用失败。
- 返回 1：表示调用成功。

3）数据结构说明如下。

- struct in_addr
  {
  unsigned long int s_addr;
  };

（2）
```
#include <arpa/inet.h>
int inet_pton (int __af,__const char *__restrict __cp,void *__restrict __buf);
```

1）输入/输出参数说明如下。

- __af：输入数据，地址族名称。在 Internet 地址族中，使用宏定义 AF_INET。
- __cp：输入数据，点分十进制表示的 IP 地址或者主机名称（域名）。如果解析主机名称（/etc/hosts）或者进行域名解析失败，将返回错误。
- __buf：输出数据，指向无类型的指针。调用功能后，该指针所指向的内容将存储地址数据。注意：该地址数据是网络字节序的数据。在 IPv4 版本下，可以用指向 struct in_addr 的结构指针作为该参数。

2）返回值说明如下。

- 0：表示输入参数有误。
- -1：表示调用出错。
- 1：表示调用成功。

```
（3）#include <arpa/inet.h>
in_addr_t inet_addr (__const char *__cp);
```

1）输入/输出参数说明如下。

__cp：输入数据，点分十进制表示的 IP 地址或者主机名称（域名）。如果解析主机名称（/etc/hosts）或者进行域名解析失败，将返回错误。

2）返回值说明如下。

- INADDR_NONE：表示调用失败。
- 其他：表示调用成功，返回网络字节序的地址。

```
（4）#include <arpa/inet.h>
char *inet_ntoa (struct in_addr __in)
```

1）输入/输出参数说明如下。

- __in：输入数据，类型为 struct in_addr 结构。

2）返回值说明如下。

返回值为点分十进制表示的字符串指针。

```
（5）#include <arpa/inet.h>
__const char *inet_ntop (int __af, __const void *__restrict __cp,char *__restrict __buf, socklen_t
__len);
```

1）输入/输出参数说明如下。

- __af：输入数据，地址族名称。在 Internet 地址族中，使用宏定义 AF_INET。
- __cp：输入数据，指向地址数据的指针。
- __buf：输出数据，字符串指针。调用成功后，该字符串中将保存点分十进制 IP 地址。
- __len：输入数据，标明__buf 缓冲区的尺寸。在<netinet/in.h>中分别提供了宏定义对 IPv4 和 IPv6 版本的地址长度进行了限定。在程序中根据 IP 版本的不同直接引用宏定义即可。

2）返回值说明如下。

- NULL：表示调用失败，errno 中将保存失败的错误码。
- 非 NULL：表示调用成功，返回值是指向__buf 的字符串指针。

【范例 20-5】根据主机名称解析其 IP 地址。其实现过程如示例代码 20-5 所示。

示例代码 20-5

```
01  #include <stdio.h>                                                    /*头文件*/
02  #include <stdlib.h>
03  #include <netdb.h>
04  #include <arpa/inet.h>
05  #include <errno.h>
06  main(int argc,char *argv[])                                  /*主函数*/
07  {
08      struct hostent *h;                                       /*结构指针变量*/
09      char buf[INET_ADDRSTRLEN +1];                            /*字符串数组*/
10      if(argc != 2)                                            /*判断命令行参数是否符合要求*/
11      {
12          fprintf(stderr,"Usage: %s hostname\n",argv[0]);/*输出用法提示信息*/
13          exit(0);
14      }
15      if((h = gethostbyname(argv[1])) == NULL)                 /*根据主机名解析主机*/
16      {
17          fprintf(stderr,"gethostbyname failed:%s\n",strerror(errno));  /*提示错误信息*/
18          exit(0);
19      }
20      bzero(buf，sizeof(buf));                                 /*清空缓冲区*/
21      if(inet_ntop(AF_INET, (const void *)(h->h_addr) ,&buf[0] ,INET_ADDRSTRLEN) != NULL)
    /*根据 22  主信息转换为 IP 地址*/
23      {
24          fprintf(stderr,"Ip address:%s\n",buf);          /*输出 IP 地址信息*/
25      }
26      else
27      {
28          fprintf(stderr,"inet_ntop failed:%s\n",strerror(errno));   /*输出错误提示信息*/
29      }
30      return;
31  }
```

【运行结果】经过编译链接，在 shell 下运行上述程序，其结果如下所示。

```
01  ./example20_5 Hubery.site
02  Ip address:192.168.1.8
```

【代码解析】在本例中，首先判断输入的两个字符串长度是否相等。如果长度不等，则长度较长的字符串较大；如果长度相等，则调用 bcmp 判断两个字符串是否相等。源代码的各行解释如下。

- 第 1～5 行：头文件信息。其中<arpa/inet.h>中包含了 inet_ntop 系统调用的原型声明。
- 第 15～19 行：调用 gethostbyname 根据主机名解析/etc/hosts 文件获取 hostent 结构。
- 第 21 行：调用 inet_ntop 对 hostent 中包含的地址数据进行解析，获取点分十进制 IP 地址。

提示：在套接口编程中，有几个比较特殊的 IP 地址。一个是 INADDR_ANY，另一个是广播地址 INADDR_BROADCAST，还有一个是环回地址 INADDR_LOOPBACK。INADDR_ANY 被定义为任意地址，其对应的点分十进制表示为 0.0.0.0。通常情况下，在建立套接口服务器前，都要在某个 IP 地址的某个端口建立监听，以便允许客户端向这个地址发起连接。

20.3.4　套接口函数

在本小节中，将详细介绍 Linux 的套接口函数。熟悉这些套接口函数，会给用户的套口编程带来极大的便利。本小节将详细介绍这些函数的功能和使用方法。

1．socket 函数

在进行网络通信前，必须首先建立套接口。套接口创建是通过系统调用 socket 实现的，其返回值是一个 int 型的描述符。在后续的套接口编程中，将以该描述符为参数。在 Linux 系统中，一切皆是文件。同样，套接口也可以看作是一个文件。对文件操作的某些系统调用（read、write 等）可以直接应用在套接口中。从 Linux 内核来看，建立套接口的操作实际上是在内核中分配了一块数据结构，并根据套接口的类型等参数对其进行初始化。socket 函数在头文件<sys/socket.h>中进行声明，其原型如下所示。

```
#include <sys/socket.h>
int socket (int __domain, int __type, int __protocol);
```

（1）输入/输出参数说明如下。

- __domain：输入数据，域（协议族）名称。在 Internet 协议族中，使用宏定义 PF_INET。
- __type：输入数据，套接口类型。常用的套接口类型有 SOCK_STREAM、SOCK_DGRAM、SOCK_RAW。
- __protocol：输入数据，协议类型，用于指定在当前协议族中采用的具体协议类型。由于 domain 和 type 参数已经给出，此处可以直接输入 0 作为参数，表示采用符合 domain 和 type 类型的默认协议。例如，对于 domain 为 PF_INET，type 为 SOCK_STREAM 的调用来说，默认采用的 protocol 就是 TCP。如图 20-9 所示，展示了 domain 和 type 组合所对应的默认协议名称。

type \ domain	PF_INET	PF_INET6	PF_LOCAL
SOCK_STREAM	TCP（IPPROTO_TCP）	TCP（IPPROTO_TCP）	支持
SOCK_DGRAM	UDP（IPPROTO_UDP）	UDP（IPPROTO_UDP）	支持
SOCK_RAW	IPv4（IPPROTO_IP）	IPv6（IPPROTO_IPV6）	不支持

图 20-9　协议族、套接口类型支持的协议

（2）返回值说明如下。

- 非负值：表示调用成功，返回套接口描述符。
- -1：表示调用出错，查看 errno 获取出错原因。

2．bind 函数

bind 函数的作用是为套接口绑定 IP 地址和端口。这里的 IP 地址必须是本机的某一个 IP

地址。对于多穴主机来说，也可以用参数 INADDR_ANY 指定本机的任意地址。函数 bind 的声明位于头文件<sys./socket.h>中，其原型如下所示。

```
#include <sys/socket.h>
int bind (int __fd, __CONST_SOCKADDR_ARG __addr, socklen_t __len)
```

（1）输入/输出参数说明如下。

- __fd：输入数据，套接口描述符。该参数由 socket 调用后返回。
- __addr：输入数据，指向通用套接口地址结构的指针。通过前面的内容可以知道，在调用套接口相关函数时，需要将某种（INET）类型的套接口地址结构进行强制类型转换，转换为通用套接口地址结构。这里就需要进行这种转换。__CONST_SOCKADDR_ARG 是一个宏，其实际定义为 const struct sockaddr *。对于 PF_INET 协议族来说，该参数变量的类型应该为 struct sockaddr_in 类型的指针。在调用 bind 传递参数时，强制类型转换为 struct sockaddr *类型。
- __len：输入数据，__addr 地址结构的长度。

（2）返回值说明如下。

- 0：表示调用成功。
- -1：表示调用出错，查看 errno 获取出错原因。

在调用 bind 函数时，很重要的一个参数就是__addr。对于 PF_INET 协议族（或者说是 AF_INET 地址族）来说，这个结构类型是 struct sockaddr_in。在前面的章节中曾介绍过该结构的内容。在进行 bind 系统调用时，对该结构进行赋值应该注意以下几点。

- 通常情况下，直接将 sin_addr 赋值 INADDR_ANY 即可。在此情况下，系统将选择本机上的所有的 IP 地址进行绑定。除非有特殊需要的例外。例如，在多穴主机中，只想在某个IP地址上监听，此时，应该指定具体的IP地址，而不能用INADDR_ANY。
- 如果调用 bind 时 sin_port 指定为 0，则将由系统随机分配端口号。通常情况下，只有在建立服务器端应用时需要调用 bind，而服务器程序大多需要明确指定监听的端口，所以，一般不会对该值赋 0。
- 在套接口编程模型中，作为客户端的套接口一般不需要调用 bind 绑定地址和端口，而是由系统自动分配一个随机的可用端口进行绑定。如果明确要求客户端必须在某个端口上与服务器发起连接，则客户端也可调用 bind。
- 在 sockaddr_in 结构中，要注意 sin_addr 及 sin_port 应该都是网络字节序。

【范例 20-6】建立一个套接口并绑定到本机的 5678 端口。其实现过程如示例代码 20-6 所示。

示例代码 20-6

```
01  #include <stdio.h>                                    /*头文件*/
02  #include <sys/socket.h>
03  #include <errno.h>
04  #include <netdb.h>
05  #include <netinet/in.h>
06  #include <arpa/inet.h>
07  main()                                                /*主函数*/
08  {
09      int nSocket;                                      /*整型变量，保存套接口描述符*/
```

```
10      struct sockaddr_in sin;                                    /*结构变量*/
11      if((nSocket = socket(PF_INET,SOCK_STREAM,0)) < 0) /*创建套接口*/
12      {
13          fprintf(stderr,"socket error:%s\n",strerror(errno)); /*输出错误提示信息*/
14          exit(0);
15      }
16      bzero(&sin,sizeof(struct sockaddr_in));                    /*清空结构*/
17      sin.sin_family = AF_INET;                                  /*Internet 地址族*/
18      sin.sin_addr.s_addr = htonl(INADDR_ANY);                   /*绑定到本地所有主机*/
19      sin.sin_port = htons(5678);                                /*绑定到 5678 端口*/
20      if(bind(nSocket,(struct sockaddr *)&sin,sizeof(struct sockaddr_in)) < 0)/*绑定套接口到上述
地址和端口*/
21      {
22          fprintf(stderr,"bind error:%s\n",strerror(errno));    /*输出错误提示信息*/
23          close(nSocket);                                        /*关闭套接口*/
24          exit(0);
25      }
26      printf("Bind success!\n");                                 /*提示信息*/
27      close(nSocket);                                            /*关闭套接口*/
28      return;
29  }
```

【运行结果】经过编译链接，在 shell 下运行上述程序。其结果如下所示，表明绑定成功。

```
01  Bind success!
```

【代码解析】在本例中，首先创建一个流式套接口，然后对套接口地址结构进行赋值，最后调用 bind 将其绑定至套接口。源代码的各行解释如下。

- 第 1～6 行：头文件信息。
- 第 10 行：声明一个套接口地址结构（struct sockaddr_in）变量 sin。
- 第 11～15 行：调用 socket 创建一个流式套接口。
- 第 16 行：调用 bzero 对套接口地址结构变量 sin 清空。
- 第 17～19 行：对地址结构赋值。地址族采用 AF_INET，绑定地址为全部地址，绑定端口为 5678。其中对于地址和端口进行了主机字节序至网络字节序的转换。
- 第 20～25 行：绑定套接口至本机的 5678 端口。
- 第 27～28 行：关闭套接口并返回。

3．connect 函数

connect 函数用于建立与 TCP 服务器的连接。该函数一般用于客户端编程中。通过调用 connect，实际上在 TCP 协议中发起了三次握手的过程。该函数声明位于头文件<sys/socket.h>中，其原型如下所示。

```
#include <sys/socket.h>
int connect (int __fd, __CONST_SOCKADDR_ARG __addr, socklen_t __len);
```

（1）输入/输出参数说明如下。

- __fd：输入数据，套接口描述符。该参数由 socket 调用后返回。
- __addr：输入数据，指向通用套接口地址结构的指针。该变量中存储的值是要连接的服务器的配置信息，如服务器的 IP 地址，服务器的端口号等。通过前面的内容可以知道，在调用套接口相关函数时，需要将某种（INET）类型的套接口地址结构进

行强制类型转换，转换为通用套接口地址结构。这里就需要进行这种转换。__CONST_SOCKADDR_ARG 是一个宏，其实际定义为 const struct sockaddr *。

- __len：输入数据，__addr 地址结构的长度。

（2）返回值说明如下。

- 0：表示调用成功。
- -1：表示调用出错，可以查看 errno 获取出错原因。一个最常见的错误就是“连接拒绝（connect refused）”，出现这个错误的原因是在相应主机的指定端口上没有建立 TCP 服务器。

提示：要检查在某个端口上是否已启动服务器，可以通过 Linux 执行命令 netstat 查看连接状态完成。在输出信息中，可以分为 6 列。其中，第 1 列是协议类型，此处是 TCP 协议；第 2 列和第 3 列分别是在当前连接上数据传输的字节数；第 4 列是本地地址及端口号，中间用“:”分隔；第 5 列是远程地址及端口号。第 6 列是连接状态。如果此处的状态是 LISTEN，表明在本地地址指定的端口上相应服务已启动。从本行输出来看，在本机所有地址上，都提供 telnet（端口 23）服务。

【范例 20-7】连接指定的主机的 DAYTIME（端口 13）服务，获取该机器时间。其实现过程如示例代码 20-7 所示。

示例代码 20-7

```
#include <stdio.h>                                                    /*头文件*/
#include <sys/socket.h>
#include <errno.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>
main()                                                                /*主函数*/
{
    int n;                                                            /*整型变量定义*/
    int nSocket;                                                      /*整型变量定义*/
    char buf[21];                                                     /*字符串数组，用作缓冲区*/
    struct sockaddr_in sin;                                           /*结构变量*/
    if((nSocket = socket(PF_INET,SOCK_STREAM,0)) < 0)/*创建套接口*/
    {
        fprintf(stderr,"socket error:%s\n",strerror(errno)); /*输出错误提示信息*/
        exit(0);
    }
    bzero(&sin,sizeof(struct sockaddr_in));                           /*清空结构*/
    if(inet_pton(AF_INET,"10.232.70.81",&(sin.sin_addr)) != 1) /*转换 IP 地址为内部数据结构*/
    {
        fprintf(stderr,"inet_pton error:%s\n",strerror(errno));       /*输出错误信息*/
        exit(0);
    }
    sin.sin_family = AF_INET;                                         /*Internet 地址族*/
    sin.sin_port = htons(13);                                         /*13 号网络端口*/
    if(connect(nSocket, (struct sockaddr *)&sin,sizeof(struct sockaddr_in)) < 0)/*连接相应的地
址和端口*/
    {
        fprintf(stderr,"connect error:%s\n",strerror(errno));         /*输出错误提示*/
```

```
29              close(nSocket);                                    /*关闭套接口*/
30              exit(0);
31          }
32          memset(buf,0x00,sizeof(buf));                          /*清空缓冲区*/
33          n = recv(nSocket,buf,20);                              /*从套接口读取数据*/
34          printf("%s\n",n,buf);                                  /*输出读到的数据*/
35          close(nSocket);                                            /*关闭套接口*/
36          return;
37      }
```

【运行结果】经过编译链接，在 shell 下运行上述程序，其结果如下所示，表明绑定成功。

```
01    25 AUG 2008 06:09:09
```

【代码解析】在本例中，首先创建一个流式套接口；然后对套接口地址结构进行赋值，设置要连接的服务器地址及端口并调用 connect 连接至服务器；最后，从服务器中读取时间数据并关闭套接口。源代码的各行解释如下。

- 第 1～6 行：头文件信息。
- 第 13～17 行：建立流式套接口。
- 第 18 行：调用 bzero 清空套接口地址结构。
- 第 19～25 行：对套接口地址结构赋值，包括地址及端口，其中调用了 inet_pton 把点分十进制的 IP 地址转换为内部数据。注意在调用了 inet_pton 后，并未对地址进行 htonl 操作以变更为网络字节序。这是由于调用 inet_pton 后，结构中的数据已经是网络字节序了。
- 第 26～30 行：调用 connect 与服务器建立连接。
- 第 32～35 行：从服务器读取时间数据并输出，最后关闭套接口。

4．listen 函数

listen 函数用于在指定套接口启动监听。该函数只在服务器端编程中调用。在一个套接口上调用该函数后，该套接口的状态将变为 LISTEN。当调用 socket 函数建立套接口时，该套接口默认为主动套接口。主动套接口的含义是指该套接口是用于主动发起连接的客户端套接口。一旦调用 listen，则套接口变为被动套接口。就是说，套接口变为被动接受连接的服务器端套接口。该函数在<sys/socket.h>中声明，其原型如下所示。

```
#include <sys/socket.h>
int listen (int __fd,int __n);
```

（1）输入/输出参数说明如下。

- __fd：输入数据，套接口描述符。该参数由 socket 调用后返回。
- __n：backlog 值，指定排队等待接受连接的最大值。该值的最大可取值由<bits/socket.h>中的宏定义 SOMAXCONN 指定。

（2）返回值说明如下。

- 0：表示调用成功。
- -1：表示调用出错，可以查看 errno 获取出错原因。

（3）关于 Listen 函数，应该注意以下几点。

- listen 应该在调用 socket 创建套接口，并且通过 bind 进行地址、端口绑定后调用。

- backlog 也称为积压值。在套接口底层的实现中，该参数是规定了 TCP 层接收链接的缓冲池（或者称为队列）的最大个数。当服务器端的 TCP 层接受客户端连接后，该连接就进入队列排队。如果此时应用层调用 accept 接受连接，TCP 层将把该队列中相应的连接移除。

【范例 20-8】建立一个服务器套接口，在 5678 端口上进行监听。其实现过程如示例代码 20-8 所示。

示例代码 20-8

```
#include <stdio.h>                                                    /*头文件*/
#include <sys/socket.h>
#include <errno.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>
main()                                                                /*主函数*/
{
    int nSocket;                                                      /*整型变量*/
    struct sockaddr_in sin;                                           /*结构变量*/
    if((nSocket = socket(PF_INET,SOCK_STREAM,0)) < 0)                 /*创建套接口*/
    {
        fprintf(stderr,"socket error:%s\n",strerror(errno));          /*输出错误信息*/
        exit(0);
    }
    bzero(&sin,sizeof(struct sockaddr_in));                           /*清空结构*/
    sin.sin_family = AF_INET;                                         /*Internet 地址族*/
    sin.sin_addr.s_addr = htonl(INADDR_ANY);                          /*绑定到本机任何地址*/
    sin.sin_port = htons(5678);                                       /*绑定到 5678 号端口*/
    if(bind(nSocket, (struct sockaddr *)&sin,sizeof(struct sockaddr_in)) < 0)     /*绑定指定的主机端口*/
    {
        fprintf(stderr,"bind error:%s\n",strerror(errno));            /*提示错误信息*/
        close(nSocket);                                               /*关闭套接口*/
        exit(0);
    }
    listen(nSocket, SOMAXCONN);                                       /*变为被动套接口，监听*/
    sleep(20);                                                        /*休眠 20 秒，以便查看网络 28 状态*/
    close(nSocket);                                                   /*关闭套接口*/
    return;
}
```

【运行结果】经过编译链接，在 shell 下运行上述程序。在本机的另外一个 shell 中执行 netstat-an|grep 5678 命令查看连接状态，其结果如下所示，表明建立服务器套接口成功。

```
tcp        0      0 0.0.0.0:5678            0.0.0.0:*               LISTEN
```

【代码解析】在本例中，首先创建一个流式套接口，然后对套接口地址结构进行赋值，设置要监听的 IP 地址及端口并调用 listen 启动监听。休眠 20 秒后，关闭套接口退出。源代码的各行解释如下。

- 第 1～25 行：参见 bind 函数的范例解释。
- 第 26 行：调用 listen，在指定的套接口上监听。
- 第 27 行：休眠 20 秒。实际应用中，应该是循环调用 accept 接收连接并进行处理。

此处仅为方便查看套接口状态。

- 第 28～29 行：关闭套接口并退出。

5．accept 函数

与 listen 函数相同，accept 也是在服务器编程中调用。在服务器调用 listen 后，客户端调用 connect 连接至服务器。从 listen 函数的说明中可以看到，服务器在 TCP 层为这些客户端连接维持了一个队列。accept 函数就是由从这个队列中取出一个已完成的连接。如果 accept 函数调用成功，则返回一个由内核产生的新的套接口。该套接口用于与客户端进行通信。accept 函数声明位于头文件<sys/socket.h>中，其原型如下所示。

```
#include <sys/socket.h>
int accept (int __fd, __SOCKADDR_ARG __addr,socklen_t *__restrict __addr_len);
```

（1）输入/输出参数说明如下。

- __fd：输入数据，套接口描述符。该参数由 socket 调用后返回。
- __addr：输出数据，指向通用套接口地址结构的指针。在 accept 函数调用成功后，该变量中将返回与客户端连接的配置信息，如客户端的 IP 地址、客户端的端口号等。__CONST_SOCKADDR_ARG 是一个宏，其实际定义为 const struct sockaddr *。对于 PF_INET 协议族来说，该参数变量的类型应该为 struct sockaddr_in 类型的指针。在调用 accept 传参时，强制类型转换为 struct sockaddr *类型。
- __addr_len：输入/输出数据，在调用 accept 时，该值为结构__addr 的尺寸。在函数返回时，该值为返回的_addr 结构的尺寸。

（2）返回值说明如下。

- 负值：表示调用出错，可以查看 errno 获取出错原因。
- 其他值：表示调用成功，返回值为新的套接口描述符，与客户端收发数据将使用新的套接口描述符。

（3）在使用 accept 函数中，应该注意以下问题。

- 如果应用层对于返回的客户端信息不关心，可以将参数__addr 及__addr_len 置为 NULL。
- 如果当前连接队列中没有准备好的连接，accept 调用将阻塞，直至有连接请求到达。
- 在并发服务器程序设计中，为保证服务器可以同时接受并处理多个客户端连接，通常采用多进程或者多线程技术。每次调用 accept 接受到一个客户端连接后，派生出一个新的子进程（线程），由新进程负责处理与客户端的数据收发处理。而主进程则继续调用 accept 接受下一个连接。

提示：在 accept 函数的参数中，__addr_len 参数既用来向 accept 传入__addr 结构的尺寸（从进程传入内核），又用来返回值（从内核返回至进程）。这种参数又称为值-结果参数。在套接口编程过程中，需要用到值-结果参数的函数有若干个，如 accept、recvfrom、getsockname、getpeername 等。在编程过程中，要注意值-结果参数传递的指向变量的指针。

【范例 20-9】建立服务器套接口，在 5678 端口上进行监听。接受端连接，获取客户端 IP 地址和端口号，并向客户端返回“Hello”及客户端 IP。其实现过程如示例代码 20-9 所示。

示例代码 20-9

```
01  #include <stdio.h>                                          /*头文件*/
02  #include <sys/socket.h>
03  #include <errno.h>
04  #include <netdb.h>
05  #include <netinet/in.h>
06  #include <arpa/inet.h>
07  main()                                                      /*主函数*/
08  {
09      int nCliLen;                                            /*整型变量*/
10      int nSocket，nCliSocket;                    /*定义用于保存套接口描述 11  符的型变量*
12      char buf[INET_ADDRSTRLEN+1];                            /*字符数组，用作缓冲区*/
13      char retbuf[101];                                       /*字符数组，用作缓冲区*/
14      struct sockaddr_in sin，cli_sin;                        /*结构变量*/
15      if((nSocket = socket(PF_INET,SOCK_STREAM，0)) < 0)  /*创建套接口*/
16      {
17          fprintf(stderr,"socket error:%s\n",strerror(errno));    /*输出错误信息*/
18          exit(0);
19      }
20      bzero(&sin,sizeof(struct sockaddr_in));                 /*清空结构*/
21      sin.sin_family = AF_INET;                               /*Internet 地址族*/
22      sin.sin_addr.s_addr = htonl(INADDR_ANY);                /*绑定本机任何地址*/
23      sin.sin_port = htons(5678);                             /*绑定至 5678 端口*/
24      if(bind(nSocket, (struct sockaddr *)&sin,sizeof(struct sockaddr_in)) < 0)     /*进行绑定*/
25      {
26          fprintf(stderr,"bind error:%s\n",strerror(errno));      /*提示错误信息*/
27          close(nSocket);                                     /*关闭套接口*/
28          exit(0);
29      }
30      listen(nSocket, SOMAXCONN);                             /*变为被动套接口，监听*/
31      for(;;)                                                 /*循环接收客户端连接*/
32      {
33          nCliLen = sizeof(cli_sin);                          /*取变量尺寸*/
34          if((nCliSocket = accept(nSocket, (struct sockaddr *)&cli_sin,&nCliLen)) <0)  /* 接 受 客
户连接*/
35          {
36              fprintf(stderr,"accept error:%s\n",strerror(errno));  /*提示错误信息*/
37              continue;                                       /*继续循环，接受下一个连接*/
38          }
39          memset(buf,0x00，sizeof(buf));                      /*清空缓冲区*/
40          printf("accept connection from\                     /*输出接入的客户端信息*/
41                  %s:%d\n",inet_ntop(AF_INET,&cli_sin.sin_addr,buf,sizeof(buf)) ,ntohs
(cli_sin.sin_port));
42          memset(retbuf,0x00,sizeof(retbuf));                 /*清空缓冲区*/
43          sprintf(retbuf,"Hello,%s.",buf);                    /*填充发送缓冲区*/
44          send(nCliSocket,retbuf,strlen(retbuf));             /*向客户端发送响应*/
45          close(nCliSocket);                                  /*关闭新套接口*/
46          continue;                                           /*继续循环，接受下一个连接*/
47      }
48      close(nSocket);                                         /*关闭服务器套接口*/
49      return;
50  }
```

【运行结果】经过编译链接，在 shell 下运行上述程序。在本机的另外一个 shell 中执行“telnet 服务器地址 5678”命令与服务器建立连接，服务器端提示信息如下所示。

```
01  accept connection from 192.168.1.3:1596
```

客户端提示信息如下所示。

```
01  Hello,192.168.1.3。
```

【代码解析】在本例中，首先创建一个流式套接口；然后对套接口地址结构进行赋值，设置要监听的 IP 地址及端口并调用 listen 启动监听循环接收来自客户端的连接，获取接入的客户端地址等信息，并调用 write 向客户端回写数据；最后关闭与客户端连接并继续接受下一个连接。源代码的各行解释如下。

- 第 1～29 行：参见 bind 函数的范例解释。
- 第 30 行：建立循环体，循环接受客户端连接。
- 第 32～37 行：调用 accept 函数接受来自客户的一个连接。
- 第 38～40 行：调用 inet_ntop 获取接收的客户端 IP 等信息。
- 第 41～43 行：填充返回数据，并调用 send 向客户端写入数据。
- 第 44～45 行：关闭客户端套接口，并继续接受下一个连接。

6．send 函数和 recv 函数

无论该套接口是客户端套接口还是服务器套接口，这两个函数都用于在流式套接口中发送和接收数据。另外，在前面的章节中介绍过，套接口描述符可以视同文件描述符，通过 read、write 系统调用完成数据收发。通过 read 和 write 系统调用时，不能指定标志参数，其他功能与 send 和 recv 相同。

send 函数并不是真正地向网络对端发送数据，发送数据的过程是由 TCP 协议完成的。send 只是检查套接口的发送缓存区是否有足够的空间，如果空间足够放入本次发送的数据中，则直接将要发送的数据拷贝至发送缓存区，send 就成功返回；如果发送缓存区空间不足，则等待 TCP 协议发送完缓存中原有的数据后，再将要发送的数据拷贝至发送缓存区。

同样，recv 函数也不正在从网络中接收数据。该函数只是从套接口的接收缓存区中读取由 TCP 协议接收到的数据。在调用 recv 时，如果接收缓存区为空，则 recv 将阻塞直至有数据到达。这两个函数的声明位于头文件<sys/socket.h>中，其原型如下所示。

```
#include <sys/socket.h>
ssize_t send (int __fd, __const void *__buf, size_t __n, int __flags);
ssize_t recv (int __fd, void *__buf, size_t __n, int __flags);
```

（1）输入/输出参数说明如下。

- __fd：输入数据，套接口描述符。该参数由 socket 调用后返回。
- __buf：输入/输出数据。对于 send 函数来说，该参数是输入参数，其中存储了要发送的数据，而对于 recv 来说，该参数是输出参数，用于缓存接收到的数据。
- __n：输入数据，表明要发送或者接收的数据长度。
- __flags：输入数据，标志位，表明 send 或者 recv 调用的标志。可选值主要包括 MSG_DONOTWAIT、MSG_OOB、MSG_PEEK 等。大多数情况下，可以将参数赋值 0，表示不采用任何标志。MSG_DONOTWAIT 标志可以使用本次数据收发操作

采用不阻塞模式。

（2）返回值说明如下。

- 负值：表示调用出错，可以查看 errno 获取出错原因。
- 其他值：表示调用成功，返回值为新的套接口描述符，与客户端收发数据将使用新的套接口描述符。

对于 send 和 recv 函数的使用可以参见范例 20-7 和范例 20-9。

提示：在套接口编程的数据收发模式中，可以分为阻塞模式及非阻塞模式，默认采用的是阻塞模式。在阻塞模式下，如果 send 或者 recv 调用时，发送缓存区已满或者接收缓存区无数据，则 send 和 recv 将阻塞。而在非阻塞模式下，则 send 和 recv 将立即返回负值，并设置 errno 为 WSAWOULDBLOCK。为了设置一个套接口为非阻塞模式，可以通过系统调用 fcntl。首先通过 F_GETFL 参数获取到当前的配置，然后将该配置与 O_NONBLOCK 进行或（|）操作，最后通过 fcntl 的 F_SETFL 参数设置到套接口。

7．sendto 函数和 recvfrom 函数

前面介绍的 send 和 recv 通常用于流式套接口数据的收发。sendto 和 recvfrom 则用于数据报套接口的数据收发。sendto 和 recvfrom 相比较于 send 和 recv 函数，其参数中多了套接口数据结构。根据这个参数，可以方便的指定要发送的报文的目的地或者获取从哪个地址接收的数据。这个特征恰好符合数据报套接口的要求。所以，sendto 和 recvfrom 更多的应用于数据报套接口的编程环境中。其函数原型如下所示。

```
#include <sys/socket.h>
ssize_t sendto (int __fd, __const void *__buf, size_t __n，int __flags, __CONST_SOCKADDR_ARG
__addr,socklen_t __addr_len);
ssize_t recvfrom (int __fd, void *__restrict __buf, size_t __n,int __flags, __SOCKADDR_ARG
__addr,socklen_t *__restrict __addr_len);
```

（1）输入/输出参数说明如下。

- __fd：输入数据，套接口描述符。该参数由 socket 调用后返回。
- __buf：输入/输出数据。对于 send 函数来说，该参数是输入参数，其中存储了要发送的数据。而对于 recv 来说，该参数是输出参数，用于缓存接收到的数据。
- __n：输入数据，表明要发送或者接收的数据长度。
- __flags：输入数据，标志位。请参见 send 函数中对标志的说明。
- __addr：输入/输出数据，指向 struct sockaddr 结构的指针。对于 sendto 来说，这是一个输入参数，存储了发送数据的目标地址数据。对于 recvfrom 来说，该参数是一个值-结果参数。
- __addr_len：输入/输出数据，表示_addr 结构的长度。对于 recvfrom 来说，该参数是一个值-结果参数。注意：在 recvfrom 函数中要传入的参数是一个指针。

（2）返回值说明如下。

- 负值：表示调用出错，可以查看 errno 获取出错原因。
- 其他值：表示调用成功，返回值为发送或者接收的数据长度。

【范例 20-10】假定当前主机在 5000 端口提供 UDP 服务，编写程序向该端口发送 UDP 报文。其实现过程如示例代码 20-10 所示。

示例代码 20-10

```
01  #include <stdio.h>                                                    /*头文件*/
02  #include <errno.h>
03  #include <sys/socket.h>
04  #include <netinet/in.h>
05  main()                                                               /*主函数*/
06  {
07      int sock_fd，n;                                                  /*整型变量*/
08      size_t size;                                                     /*定义 size_t 变量*/
09      struct sockaddr_in sin;                                          /*结构变量定义*/
10      char buf[15] = "Hello World!";                                   /*定义字符数组*/
11      sock_fd = socket(PF_INET,SOCK_DGRAM,IPPROTO_UDP);                /创建套接口*/
12      sin.sin_family = AF_INET;                                        /*Internet 地址族*/
13      sin.sin_addr.s_addr = htonl(INADDR_LOOPBACK);                    /*绑定到本机任何地址*/
14      sin.sin_port = htons(5000);                                      /*绑定到 5000 端口*/
15      size = sizeof(sin);                                              /*求得结构的尺寸*/
16      if((n = sendto(sock_fd,buf,13,0, (struct sockaddr*)&sin,size)) < 0) /*发送 UDP 数据报*/
17      {
18          fprintf(stderr,"sendto failed:%s\n",strerror(errno));        /*输出错误提示*/
19      }
20      else
21      printf("sendto success!\n",n);                                   /*提示信息*/
22      close(sock_fd);                                                  /*关闭套接口*/
23  }
```

【运行结果】经过编译链接，在 shell 下运行上述程序，其结果如下所示，表明绑定成功。

```
01  sendto success!
```

【代码解析】在本例中，首先创建一个流式套接口；然后对套接口地址结构进行赋值，设置要连接的服务器地址及端口并调用 connect 连接至服务器；最后，调用 recvmsg 读取时间数据并关闭套接口。源代码的各行解释如下。

- 第 1～4 行：头文件。
- 第 11 行：调用 socket 创建数据报套接口；socket 的第三个参数是 IPPROTO_UDP，该参数也可以输入 0，表示由系统自动选择协议。
- 第 12～14 行：为 sendto 调用准备套接口地址结构，定义数据报发送的目标地址和端口号。
- 第 15～21 行：调用 sendto 向目标发送 UDP 数据报。
- 第 22 行：关闭套接口。

提示：recvfrom 不仅可以应用于面向无连接模式的数据报接收，而且也可以用于面向连接的数据接收。如果调用 recvfrom 时__addr 参数传入 NULL，则等同于调用 recv。sendto 和 send 的关系也是如此。

8．sendmsg 函数和 recvmsg 函数

这两个函数是通用的数据收发函数，仅可以用于在流式套接口中收发数据（代替 send 和 recv），还可以用于在数据报套接口中发送和接收数据（代替 sendto 和 recvfrom）。

```
#include <sys/socket.h>
ssize_t sendmsg (int __fd, __const struct msghdr *__message,int __flags);
ssize_t recvmsg (int __fd, struct msghdr *__message, int __flags);
```

（1）输入/输出参数说明如下。

- __fd：输入数据，套接口描述符。该参数由 socket 调用后返回。
- __message：输入/输出数据，指向 msghdr 结构的指针。对于 sendmsg 该参数指定发送的目标及数据内容，对于 recvmsg 该参数是一个值-结果参数。调用成功时，该参数中保存由内核返回的相关数据。
- __flags：标志，用于指定发送或者接收的标志。请参见 send 和 recv 对标志的说明。

（2）返回值说明如下。

- -1：表明调用出错，可以查看 errno 获取出错详细信息。
- 其他：调用成功，返回值为发送/接收的字节数。

【范例 20-11】用 recvmsg 实现范例 20-7。连接指定的主机的 DAYTIME（端口 13）服务，获取该机器时间。其实现过程如示例代码 20-11 所示。

示例代码 20-11

```
#include <stdio.h>                                          /*头文件*/
#include <sys/socket.h>
#include <errno.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>
main()                                                      /*主函数*/
{
    int n;                                                  /*定义整型变量*/
    int nSocket;                                            /*定义整型变量*/
    char buf[21];                                           /*字符数组，用作缓存区*/
    struct sockaddr_in sin;                                 /*结构变量*/
    struct iovec iov[1];                                    /*结构数组变量*/
    struct msghdr mh;                                       /*结构变量*/
    if((nSocket = socket(PF_INET,SOCK_STREAM,0)) < 0)       /*创建套接口*/
    {
        fprintf(stderr,"socket error:%s\n",strerror(errno));/*输出错误提示信息*/
        exit(0);
    }
    bzero(&sin，sizeof(struct sockaddr_in));                /*清空结构*/
    if(inet_pton(AF_INET,"10.232.70.81",&(sin.sin_addr)) != 1)/*转换字符串形式 IP*/
    {
        fprintf(stderr,"inet_pton error:%s\n",strerror(errno));/*输出错误提示信息*/
        exit(0);
    }
    sin.sin_family = AF_INET;                               /*Internet 地址族*/
    sin.sin_port = htons(13);                               /*13 号网络端口*/
    if(connect(nSocket,(struct sockaddr *)&sin,sizeof(struct sockaddr_in)) < 0) /*连接至相应端口*/
    {
        fprintf(stderr,"connect error:%s\n",strerror(errno));/*输出错误提示信息*/
        close(nSocket);                                     /*关闭套接口*/
        exit(0);
    }
    bzero(&mh,sizeof(struct msghdr));                       /*清空结构*/
```

```
35        mh.msg_name=NULL;                                    /*为结构赋值*/
36        mh.msg_namelen = 0;
37        mh.msg_iov = iov;
38        mh.msg_iovlen = 1;
39        iov[0].iov_base = (char *)buf;                       /*设置接收数据缓存区*/
40        iov[0].iov_len = 20;                                 /*接收数据长度*/
41        memset(buf,0x00,sizeof(buf));
42        recvmsg(nSocket,&mh,0);                              /*接收数据*/
43        printf("%s\n",buf);                                  /*输出接收到的数据*/
44        close(nSocket);                                      /*关闭套接口*/
45        return;
46    }
```

【运行结果】经过编译链接，在 shell 下运行上述程序，其结果如下所示，表明绑定成功。

```
01    25 AUG 2008 22:00:49
```

【代码解析】在本例中，首先创建一个流式套接口；然后对套接口地址结构进行赋值，设置要连接的服务器地址及端口并调用 connect 连接至服务器；最后，调用 recvmsg 读取时间数据并关闭套接口。源代码的各行解释如下。

- 第 1～33 行：参见范例 20-7 的相关解释。
- 第 34～40 行：为结构 msghdr 赋值。在本例中，不需要使用套接口地址结构 msg_name，所以将该参数设置为 NULL。程序中定义的输出缓冲区 buf 绑定到 iov 结构数组，并将 iov 赋值给 msghdr 结构中的 msg_iov 参数。请注意，该参数实际是一个指向 struct iov 的结构数组。在本例中，只需要用到一个缓冲区，所以数组长度为 1。
- 第 42 行：调用 recvmsg 从服务器读取数据。读取的长度是在 iov 结构的 iov_len 变量中指定的。
- 第 43～45 行：输出读取的数据并关闭套接口。

提示：sendmsg 和 recvmsg 除可以正常发送/接收数据外，还有一个重要的功能，就是发送/接收辅助数据（或者称为控制信息）。关于辅助数据的内容本书不作讨论，读者有兴趣可以参考其他相关资料。

9．close 函数和 shutdown 函数

这两个函数都用于关闭套接口。不过，close 是与文件操作的 close 共用的函数，而 shutdown 则是套接口专用的函数。

TCP 连接是全双工的连接，即在同一个连接上既可以发送数据，又可以接收数据。在调用 close 函数后，该套接口既不能用于读取数据，也不能用于发送数据。而调用 shutdown 函数可以在关闭套接口时做更多的选择。shutdown 函数允许指定只关闭一个方向（读或者写）的通信或者关闭双向通信。函数 close 的声明位于头文件<unistd.h>中，其原型如下所示。

```
#include <unistd.h>
int close (int __fd);
```

函数 shutdown 的声明头文件<sys/socket.h>中，其原型如下所示。

```
#include <sys/socket.h>
int shutdown (int __fd, int __how);
```

（1）输入/输出参数说明如下。

- __fd：输入数据，套接口描述符。该参数由 socket 调用后返回。

- __how：输入数据，表明套接口关闭的方式。
- __flags：标志，用于指定发送或者接收的标志。请参见 send 和 recv 对标志的说明。

（2）返回值说明如下。

- 0：表明调用成功。
- 其他：表明调用失败。

（3）在调用 shutdown 函数中，应该注意如下几点。

- shutdown 函数关闭套接口时，不受套接口引用计数问题的限制，它将直接关闭套接口。
- shutdown 函数只对面向连接的套接口有效，对于面向无连接的套接口，应使用 close。

提示：在多进程编程环境（并发服务器）下，要注意套接口的引用计数问题。Linux 文件系统对进程的每个打开的文件描述符都在内核保存一个引用计数，对于套接口同样也是如此。如果该套接口描述符被多个进程所引用，则该套接口的引用计数会大于 1。此时，关闭套接口的操作只是对其引用计数减 1。只有在套接口的引用计数变为 0 的情况下，内核才会真正关闭一个套接口。

10．getsockname 函数和 getpeername 函数

这两个函数分别用于获取与套接口关联的本地连接信息及远程信息。之所以提供这两个函数，是由于在某些情况下，需要通过套接口描述符获取连接的地址信息。例如，在调用 connect 与服务器建立连接后，可以调用 getsockname 获取 TCP 分配给该连接的本地端口号；在调用 accept 接受客户端连接后，可以调用 getsockname 获取分配给该连接的本地地址及端口号等。这两个函数的声明位于头文件<sys/socket.h>中，其原型如下所示。

```
#include <sys/socket.h>
int getpeername (int __fd, __SOCKADDR_ARG __addr,socklen_t *__restrict __len);
int getsockname (int __fd, __SOCKADDR_ARG __addr,socklen_t *__restrict __len);
```

（1）输入/输出参数说明如下。

- __fd：输入数据，套接口描述符。该参数由 socket 调用后返回。
- __addr：输入/输出数据，指向 struct sockaddr 结构的指针。调用成功后，该参数中保存本地或者远程地址信息。
- __len：该参数是一个值-结果参数。

（2）返回值说明如下。

- 0：表明调用成功。
- -1：表明调用失败。可以查看 errno 获取错误详细信息。

提示：调用 getpeername 或者 getsockname 获取地址结构中的数据均为网络字节序，在解析至本地时要注意进行主机字节序转换。

【范例 20-12】以范例 20-7 为例，从指定服务器的 13 端口获取服务器时间。在连接服务器成功后，获取本地连接所绑定的地址及端口号。其实现过程如示例代码 20-12 所示。

示例代码 20-12

```
01  #include <stdio.h>                                          /*头文件*/
02  #include <sys/socket.h>
03  #include <errno.h>
04  #include <netdb.h>
```

```
#include <netinet/in.h>
#include <arpa/inet.h>
main()                                                          /*主函数*/
{
    int n，len;                                                 /*定义整型变量*/
    int nSocket;                                                /*定义整型变量*/
    char buf[21];                                               /*字符数组*/
    struct sockaddr_in sin;                                     /*定义结构变量*/
    struct sockaddr_in cin;                                     /*定义结构变量*/
    if((nSocket = socket(PF_INET,SOCK_STREAM,0)) < 0)           /*创建套接口*/
    {
        fprintf(stderr,"socket error:%s\n",strerror(errno));    /*提示错误信息*/
        exit(0);
    }
    bzero(&sin,sizeof(struct sockaddr_in));                     /*清空结构*/
    if(inet_pton(AF_INET,"10.232.70.81",&(sin.sin_addr)) != 1) /*转换字符串形式的 IP 地址*/
    {
        fprintf(stderr,"inet_pton error:%s\n",strerror(errno)); /*提示错误信息*/
        exit(0);    }
    sin.sin_family = AF_INET;                                   /*Internet 地址族*/
    sin.sin_port = htons(13);                                   /*连接到 13 号端口*/
if(connect(nSocket,(struct sockaddr *)&sin,sizeof(struct sockaddr_in)) < 0)/*执行连接操作*/
    {
        fprintf(stderr,"connect error:%s\n",strerror(errno));   /*提示错误信息*/
        close(nSocket);                                         /*关闭套接口*/
        exit(0);
    }
    len = sizeof(cin);                                          /*计算结构的尺寸*/
    bzero(&cin,len);                                            /*清空结构*/
    if(getsockname(nSocket, (struct sockaddr *)&cin,&len) < 0)/*得到本地连接的信息*/
    {
        fprintf(stderr,"getsockname error:%s\n",strerror(errno));/*提示错误信息*/
    }
    memset(buf,0x00,sizeof(buf));                               /*清空缓存区*/
    printf("local bind %s:%d\n",                                /*输出本地连接的信息*/
    inet_ntop(AF_INET, (void *)&cin.sin_addr.s_addr,buf,INET_ADDRSTRLEN),
ntohs(cin.sin_port));
    memset(buf,0x00,sizeof(buf));                               /*清空接收缓存区*/
    n = recv(nSocket,buf,20,0);                                 /*从套接口接收数据*/
    printf("%s\n",buf);                                         /*输出接收到的数据*/
    close(nSocket);                                             /*关闭套接口*/
    return;
}
```

【运行结果】经过编译链接，在 shell 下运行上述程序，其结果如下所示，表明绑定成功。

```
local bind 10.232.70.81:32781
26 AUG 2008 00:03:46
```

【代码解析】在本例中，在连接至服务器后，调用 getsockname 获取该连接绑定的地址及端口。从程序的输出可以看到，该连接绑定的地址是 10.232.70.81，端口是 32781。源代码的各行解释如下（参见范例 20-7）。

- 第 32～37 行：调用 getsockname 获取连接绑定的地址数据。
- 第 38～40 行：调用 inet_ntop 把获取到的地址数据转换为点分十进制的字符串形式。

20.4 套接口选项

套接口功能强大，接口复杂，能够满足多种情况下的网络编程的需要，由此产生了套接口大量的可编程配置项，这就是套接口选项。在本节中，将首先介绍获取、设置套接口选项的相关函数，然后介绍几种常用的套接口选项。

20.4.1 套接口选项函数

在 Linux 下，可以通过两个函数获取或者设置套接口选项。这两个函数是 getsockopt 和 setsockopt，其声明位于头文件<sys/socket.h>中，原型如下所示。

```
#include <sys/socket.h>
int getsockopt (int __fd, int __level, int __optname,void *__restrict __optval,socklen_t *__restrict __optlen);
setsockopt (int __fd, int __level, int __optname,__const void *__optval, socklen_t __optlen);
```

（1）输入/输出参数说明如下。

- __fd：输入数据，套接口描述符。该参数由 socket 调用后返回。
- __level：输入数据，选项级别。该参数指定了在何种级别上对套接口选项进行操作，这里的级别可以理解为协议的层次。主要的级别包括 SOL_SOCKET、IPPROTO_IP、IPPROTO_TCP 等。本书只对 SOL_SOCKET 级别的选项进行讲解，其他级别的选项请读者自行参考相关资料。
- __optname：输入数据，选项名称。在 SOK_SOCKET 选项级别下，主要的选项名称包括 SO_KEEPALIVE、SO_LINGER、SO_RCVBUF、SO_SNDBUF、SO_RCVTIMEO、SO_SNDTIMEO、SO_REUSEADDR、SO_REUSRPORT 等。在下面的内容中，将逐一进行介绍。
- __optval：输入/输出数据，指向选项值的指针。对于 getsockopt 来说，该值是值-结果参数，调用成功后，该参数中保存套接口的当前选项值。对于 setsockopt 来说，该值则是输入参数，用于设置新的套接口选项值。该参数多数情况下为整形变量，但在某些选项中，该参数也可能为结构变量，如 SO_LINGER 选项。
- __optlen：该参数是一个值-结果参数，表示 opeval 的尺寸。

（2）返回值说明如下。

- 0：表明调用成功。
- -1：表明调用失败。可以查看 errno 获取错误详细信息。

（3）关于套接口选项函数，应该注意：在调用 socket 创建套接口成功后，才能在套接口上获取或者设置选项。

20.4.2 SO_KEEPALIVE 选项

在网络环境下的应用中，可能存在这样的情况：客户机与服务器连接成功后，很长时间

没有数据的收发。此时，服务器无法知道客户机是否还在活动状态。如果客户机已经宕机或者进程已经异常终止，会导致服务器端产生一个半开状态的连接。一旦这种连接过多，将大量耗费服务器资源，严重影响服务器效率。

为避免上述情况的产生，套接口提供了 SO_KEEPALIVE 选项，用于探测对方主机是否崩溃。一旦在套接口上打开该选项，如果在 2 个小时内没有任何数据收发，系统将自动向对方发送一个探测包。根据 TCP/IP 协议的要求，对方的 TCP 协议在收到这个探测包后，必须进行响应。响应可以分为以下 3 种情况。

- 对方回复以明确的响应。此时，表明该连接一切正常。在下一个 2 小时内继续按照该规则发送探测包。这种情况对应用层没有任何影响。
- 对方回复系统已崩溃或者重启的信息。此时，本地应用层的套接口将返回 ECONNRESET 错误。如果在套接口上调用 select 函数或者 read 函数等待读取数据，则很容易捕获到该错误信息。应用层根据应用的要求做出适当处理。
- 对方未对探测包做出任何响应。在这种情况下，不同版本的套接口在实现上略微有差异。多数的做法是每隔 75 秒发送一个探测包，连续发送 8 个。如果在第一个包发出后超过 11 分 15 秒仍然无响应，则应用层套接口将捕获到错误信息 ETIMEOUT。应用层需要根据应用的要求做出适当处理。

20.4.3　SO_LINGER 选项

通常情况下，在数据收发完成后，应用程序将调用 close 关闭套接口。TCP/IP 协议在默认状态下，会将底层协议中发送缓冲区的数据发送完成后，再关闭连接。在某些情况下，可能需要对该默认处理方式进行修改，这时就需要用到 SO_LINGER 选项。SO_LINGER 选项的获取和设置需要用到一个特殊的数据结构 struct linger，该结构中头文件<bits/socket.h>中声明，其定义如下所示。

```
struct linger
  {
    int l_onoff;                /* Nonzero to linger on close.*/
    int l_linger;               /* Time to linger.*/
  };
```

对于 SO_LINGER 选项的设置可以分为以下 3 种情况。

- l_onoff 值设置为 0。在这种情况下，close 函数将按照前面所说的默认方式进行工作，将底层协议中发送缓冲区中的数据发送完成后，关闭连接。
- l_onoff 值设置为非 0 值，而且 l_linger 设置为 0。在这种情况下，一旦调用 close 关闭套接口，则该套接口将立即关闭，底层协议中发送缓冲区中的数据将会丢失。
- l_onoff 值设置为非 0 值，而且 l_linger 设置为非 0 值。在这种情况下，在调用 close 后，如果发送缓冲区中还有数据尚未发送完成，则 TCP/IP 协议将继续进行发送，直至数据发送完成或者 l_linger 设置的时间已超时。如果出现超时的情况，则 close 将返回失败，同时错误代码 errno 被置为 EWOULDBLOCK，底层协议中发送缓冲区中未发送完成的数据将全部丢失。

【范例 20-13】以范例 20-7 为例，从指定服务器的 13 端口获取服务器时间。在建立套接口后，调用 setsockopt 设置该套接口在调用 close 函数 60 秒的发送超时时间。其实现过程如示例代码 20-13 所示。

示例代码 20-13

```
#include <stdio.h>                                          /*头文件*/
#include <sys/socket.h>
#include <errno.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>
main()                                                      /*主函数*/
{
    int n;                                                  /*定义整型变量*/
    int nSocket;                                            /*定义整型变量*/
    char buf[21];                                           /*字符数组*/
    struct sockaddr_in sin;                                 /*结构变量*/
    struct linger ling;                                     /*结构变量*/
    if((nSocket = socket(PF_INET,SOCK_STREAM,0)) < 0)       /*创建套接口*/
    {
        fprintf(stderr,"socket error:%s\n",strerror(errno));    /*输出错误提示信息*/
        exit(0);
    }
    ling.l_onoff = 1;                                       /*打开 Linger 选项*/
    ling.l_linger = 60;                                     /*延时 60 秒关闭*/
    if(setsockopt(nSocket,SOL_SOCKET,SO_LINGER,(char*)&ling,sizeof(ling))<0)/*设置选项*/
    {
        fprintf(stderr,"setsockopt error:%s\n",strerror(errno));  /*输出错误提示信息*/
    }
    bzero(&sin,sizeof(struct sockaddr_in));                 /*清空缓存区*/
    if(inet_pton(AF_INET,"10.232.70.81",&(sin.sin_addr)) != 1) /*转换 IP 地址*/
    {
        fprintf(stderr,"inet_pton error:%s\n",strerror(errno));  /*输出错误提示*/
        exit(0);
    }
    sin.sin_family = AF_INET;                               /*Internet 地址族*/
    sin.sin_port = htons(13);                               /*要连接 13 号端口*/
    if(connect(nSocket, (struct sockaddr *)&sin,sizeof(struct sockaddr_in)) < 0)/*执行连接*/
    {
        fprintf(stderr,"connect error:%s\n",strerror(errno));  /*输出错误提示*/
        close(nSocket);                                     /*关闭套接口*/
        exit(0);
    }
    memset(buf,0x00,sizeof(buf));                           /*清空缓冲区*/
    n = recv(nSocket,buf,20,0);                             /*从套接口接收数据*/
    printf("%s\n",buf);                                     /*输出接收到的数据*/
    close(nSocket);                                         /*关闭套接口*/
    return;
}
```

【运行结果】经过编译链接，在 shell 下运行上述程序，其结果如下所示。调用 setsockopt 时无错误信息输出，表明设置 SO_LINGER 参数成功。

```
26 AUG 2008 04:09:33
```

【代码解析】在本例中，在创建套接口后，调用 setsockopt 设置 SO_LINGER 选项。源码的各行解释如下（参见范例 20-7）。

- 第 13 行：定义 struct linger 结构变量。
- 第 19 行：设置 linger 结构成员 l_onoff 为非 0 值，表明 SO_LINGER 选项在设置完成后将有效。
- 第 20 行：设置 linger 结构成员 l_linger 为 60，表示 close 后系统将等待 60 秒，以便底层协议完成发送缓冲区内的数据。
- 第 21 行：调用 setsockopt 设置 SO_LINGER 选项。

20.4.4　SO_RCVBUF 和 SO_SNDBUF 选项

通过前面对 send 和 recv 函数的介绍，读者应当了解到：TCP/IP 协议的底层为每一个套接口建立了两个缓冲区，分别是发送缓冲区及接收缓冲区。在应用程序中调用 send 或者 recv 收发数据，实际上是向这两个缓冲区中拷贝数据，真正的数据收发由 TCP/IP 协议去完成。

默认情况下，这个缓冲区的内容大致为 4096～8192 字节（不同的系统实现可能有所不同）。在每次收发数据量比较大的情况下，可以适当调整该参数的大小，以避免不断的循环收发，提高效率。

对于该选项的设置，应该注意以下几点。

- 对于客户端套接口，应该在调用 connect 之前进行设置；对于服务器端套接口，应该在调用 listen 之前进行设置。这是由于在底层协议的实现中，该选项是在建立连接的过程中设置的。所以，必须在套接口建立连接之前进行设置。
- 由于 TCP 协议存在滑动窗口机制，发送方在发送数据前会先查看接收方的接收缓存区大小，根据该大小发送数据。因此，接口缓存区永远不会溢出。

【范例 20-14】以范例 20-7 为例，从指定服务器的 13 端口获取服务器时间。在连接服务器成功后，获取当前连接的发送缓存区尺寸。其实现过程如示例代码 20-14 所示。

示例代码 20-14

```
01  #include <stdio.h>                                          /*头文件*/
02  #include <sys/socket.h>
03  #include <errno.h>
04  #include <netdb.h>
05  #include <netinet/in.h>
06  #include <arpa/inet.h>
07  main()                                                      /*主函数*/
08  {
09      int n，nLen，nSndBuf;                                   /*定义整型变量*/
10      int nSocket;                                            /*定义整型变量*/
11      char buf[21];                                           /*定义字符数组变量*/
12      struct sockaddr_in sin;                                 /*定义结构变量*/
13      if((nSocket = socket(PF_INET,SOCK_STREAM,0)) < 0)       /*创建套接口*/
14      {
15          fprintf(stderr,"socket error:%s\n",strerror(errno));    /*输出错误提示*/
16          exit(0);
```

```
17        }
18        bzero(&sin，sizeof(struct sockaddr_in));                    /*清空结构*/
19        if(inet_pton(AF_INET,"10.232.70.81",&(sin.sin_addr)) != 1) /*转换 IP 地址*/
20        {
21            fprintf(stderr,"inet_pton error:%s\n",strerror(errno));    /*输出错误提示*/
22            exit(0);
23        }
24        sin.sin_family = AF_INET;                                   /*Internet 地址族*/
25        sin.sin_port = htons(13);                                   /*要连接端口 13*/
26        if(connect(nSocket, (struct sockaddr *)&sin,sizeof(struct sockaddr_in)) < 0) /*执行连接操作*/
27        {
28            fprintf(stderr,"connect error:%s\n",strerror(errno));   /*输出错误信息*/
29            close(nSocket);                                         /*关闭套接口*/
30            exit(0);
31        }
32        nSndBuf = 0;                                                /*设置变量值为 0*/
33        nLen = sizeof(int);                                         /*取整型变量的尺寸*/
34        if(getsockopt(nSocket,SOL_SOCKET,SO_SNDBUF, (void *)&nSndBuf,&nLen) < 0)   /* 得
到选 35    SO_SNDBUF 的值*/
36        {
37            fprintf(stderr,"getsockopt error:%s\n",strerror(errno));   /*输出错误信息*/
38        }
39        printf("SO_SNDBUF=%d\n",nSndBuf);                           /*输出选项的值*/
40        close(nSocket);                                             /*关闭套接口*/
41        return;
42    }
```

【运行结果】经过编译链接，在 shell 下运行上述程序，其结果如下所示，表明当前连接的发送缓冲区为 50592 字节。

```
SO_SNDBUF=50592
```

【代码解析】在本例中，在连接至服务器后，调用 getsockopt 获取该连接的发送缓存区尺寸。源代码的各行解释参见范例 20-7，其中第 32～38 行表示调用 getsockopt 获取连接的发送缓存区大小，此处的 nLen 是一个值-结果参数。

20.4.5 SO_RCVTIMEO 和 SO_SNDTIMEO 选项

在阻塞模式下，可以为套接口设置读写操作的超时时间。在默认状态下，这两个参数都是禁止的，即不设置超时时间。这个超时时间影响的函数包括 read、recv、recvfrom、recvmsg 及对应的发送函数 write、send、sendto、sendmsg。该超时时间的设置可以达到微秒级的精确度。设置该参数时，需要用到一个结构 timeval。该结构定义了超时的时间参数，其声明位于<bits/time.h>中，定义如下所示。

```
struct timeval
  {
    __time_t tv_sec;            /* Seconds*/
    __suseconds_t tv_usec;      /* Microseconds*/
  };
```

其中，tv_sec 定义以秒为单位的超时时间；tv_usec 定义以微秒为单位的超时时间。

20.4.6　SO_REUSEADDR 和 SO_REUSEPORT 选项

在默认情况下，如果一个套接口绑定了某个端口，则该端口将无法再绑定到其他套接口上。在 Linux 系统中，在该大接口正常关闭或程序退出后，在一段时间内该端口将仍然保持被绑定的状态，其他程序（或者重新启动原程序）无法绑定该端口。为避免出现该情况，可以通过设置 SO_REUSEADDR 和 SO_REUSEPORT 选项解决。

对于该选项，比较常见的应用场景是服务器出现故障后停止服务器并重新启动。此时，如果不设置该选项，由于刚刚停止的服务器所持有的 TCP 连接仍然处在 TIME_WAIT 状态，在进行绑定（bind）操作时将报错。而通过设置该选项，则可以保证服务器立即启动成功。

提示：在某些版本的 Linux 下，SO_REUSEPORT 套接口选项不支持。

20.5　TCP 套接口编程

本节将向读者介绍通用的 TCP 套接口编程模型。在上一章介绍客户机/服务器模式时，曾经介绍过服务器可以分为重复服务器和并发服务器。在本节中，将向读者介绍套接口编程这两种服务器模型的实现，并通过范例讲解实现过程。

20.5.1　重复服务器编程

重复服务器指的是服务器在接收客户机的连接请求后即与之建立连接，在处理完与客户机的通信任务后断开连接，再去接收另一客户机的连接请求。其优点是不必生成通信子进程，对系统资源要求较低；缺点是每次只能有一个客户机与服务器建立连接。TCP 重复服务器一次只能处理一个客户端的请求，不能满足大多数 TCP 服务器的要求。因此，TCP 服务器一般较少用重复服务器模型。TCP 重复服务器编程模型如图 20-10 所示。

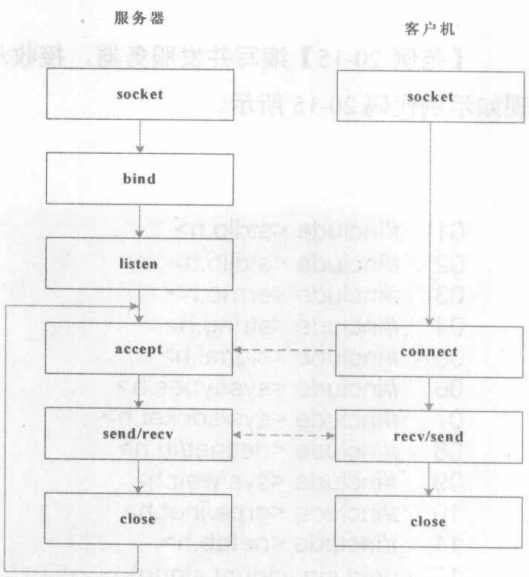


图 20-10　TCP 重复服务器编程模型

20.5.2　并发服务器编程

并发服务器编程在服务器编程中最为常见。主要的并发服务器实现方式有两种：一是通过子进程并发的方式实现并发服务器；二是通过 I/O 多路复用实现并发服务器。在这里向读者介绍第一种基于子进程并发的并发服务器。读者如果感兴趣，可以自行研究一下基于 I/O 多路复用方式的并发服务器。并发服务器编程模型如图 20-11

所示。

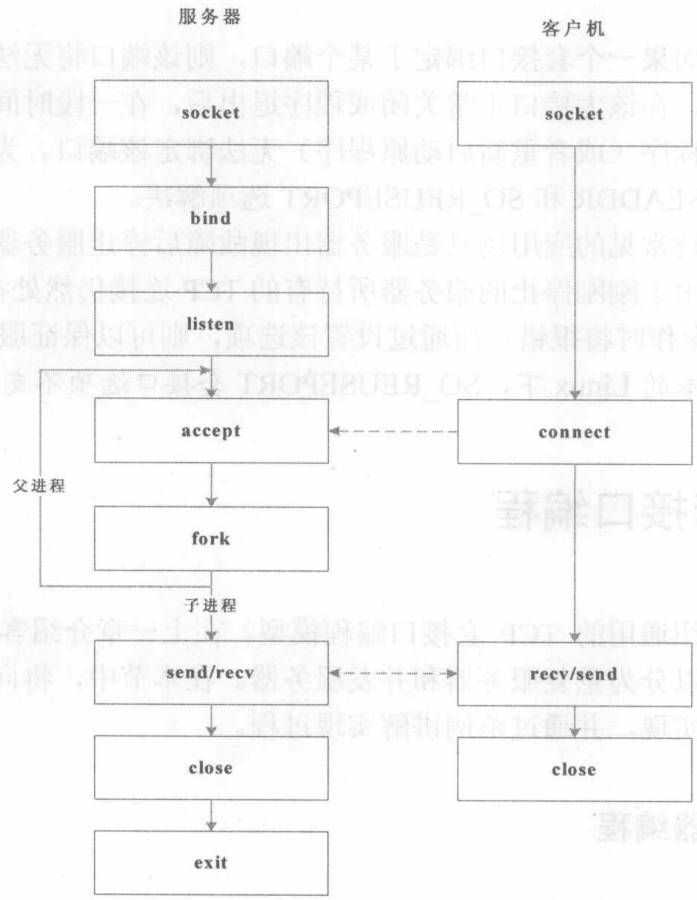


图 20-11　TCP 并发服务器编程模型

【范例 20-15】编写并发服务器，接收从客户端传入的正整数，计算其阶乘后向客户端返回。服务器实现如示例代码 20-15 所示。

示例代码 20-15

```
#include <stdio.h>                                          /*头文件*/
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/wait.h>
#include <arpa/inet.h>
#include <netdb.h>
void sig_cld(int signo)                                     /*信号处理函数*/
{
    int status;                                             /*定义整型变量*/
    while(waitpid(-1,&status,WNOHANG)>0);                   /*等待子进程退出*/
}
long ComputeFact(int nFact)                                 /*定义函数*/
```

```
{
    int n;                                                    /*整型变量*/
    long lResult = 1;                                         /*长整型变量*/
    if(nFact == 0) return 1;                                  /*0 的阶乘为 1 */
    for(n=1;n<=nFact;n++)                                     /*循环计算*/
    {
        lResult *= n;                                         /*累乘*/
    }
    return lResult;                                           /*返回阶乘结果*/
}
int main()                                                    /*主函数*/
{
    long lFact;                                               /*定义长整型变量*/
    int sockfd,new_fd;                                        /*定义整型变量*/
    int len,size,nReadVar;                                    /*定义整型变量*/
    struct sockaddr_in sin,cin;                               /*定义结构变量*/
    struct sigaction act;                                     /*定义结构变量*/
    unsigned int sin_size;                                    /*无符号整型变量*/
    if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1) /*创建套接口*/
    {
        perror("socket");                                     /*输出错误提示信息*/
        exit(1);
    }
    size=1;                                                   /*赋值*/
    len=sizeof(size);                                         /*求变量的尺寸*/
    if(setsockopt(sockfd,SOL_SOCKET,SO_REUSEADDR, (char*)&size,len) < 0)   /* 设 置 套
接口选项*/
    {
        perror("setsockopt REUSEADDR");                       /*输出错误提示信息*/
        exit(1);
    }
    bzero(&sin,sizeof(sin));                                  /*清空结构*/
    sin.sin_family = AF_INET;                                 /*Internet 地址族*/
    sin.sin_port = htons(5000);                               /*要绑定到网络端口 5000*/
    sin.sin_addr.s_addr = htonl(INADDR_ANY);                  /*要绑定到本机任何地址*/
    if(bind(sockfd, (struct sockaddr *)&sin, sizeof(struct sockaddr_in)) == -1)/*执行绑定*/
    {
        perror("bind");                                       /*输出错误提示信息*/
        exit(1);
    }
    act.sa_handler=sig_cld;                                   /*设置要安装的信号处理函数*/
    sigemptyset(&act.sa_mask);                                /*清空信号掩码*/
    act.sa_flags=0;                                           /*清理标志*/
    sigaction(SIGCLD,&act,NULL);                              /*安装信号*/
    if (listen(sockfd, 5) == -1) {                            /*启动监听*/
        perror("listen");                                     /*输出错误信息*/
        exit(1);
    }
    while(1) {                                                /*循环接受客户连接*/
        sin_size = sizeof(struct sockaddr_in);                /*取结构的尺寸*/
    if ((new_fd = accept(sockfd, (struct sockaddr *)&cin, &sin_size))==-1) /*接受客户连接*/
        {
            perror("accept");                                 /*输出错误提示信息*/
```

```
70                  continue;                                          /*继续接受下一个客户连接*/
71              }
72              if (!fork())                                           /*创建子进程，处理客户连接*/
73              {
74                  close(sockfd);                                     /*关闭监听套接口，引用计数减 1*/
75                  if(recv(new_fd, (void *)&nReadVar,sizeof(int) ,0) < 0)   /*从新套接口接收数据*/
76                  {
77                      perror("send");                                /*输出错误提示信息*/
78                      close(new_fd);                                 /*关闭新的与客户端的套接口*/
79                      exit(0);
80                  }
81                  lFact = ComputeFact(nReadVar);                     /*计算阶乘*/
82                  if (send(new_fd, (void *)&lFact,sizeof(long) , 0) < 0)     /*发送计算阶乘的结果*/
83                  {
84                      perror("send");                                /*输出错误提示信息*/
85                      close(new_fd);                                 /*关闭新的与客户端的套接口*/
86                      exit(0);
87                  }
88                  close(new_fd);                                     /*关闭新的与客户端的套接口*/
89                  exit(0);
90              }
91              else
92              {
93                  close(new_fd);                                 /*父进程关闭与客户端连接的套接 94 口*/
95                  continue;                                      /*父进程继续循环，接收下一连接*/
96              }
97          }
98      }
```

【运行结果】经过编译链接，在 shell 下运行服务器程序，没有错误信息输出，表明服务器启动成功，等待客户端连接，如下所示。

```
01  ./example20_15
```

【代码解析】在本例中，首先创建一个流式套接口，然后设置套接口选项 SO_REUSEADDR，并将该套接口绑定到本机的 5000 端口进行监听，并发接收客户端连接，在子进程中计算正整数的阶乘并写回客户端。在本例中，使用了多进程编程。在多进程编程环境中，注意对信号 SIGCLD 的处理。源代码的各行解释如下。

- 第 1～1 行：头文件。
- 第 12～16 行：函数 sig_cld 定义。该函数是程序设置的信号 SIGCLD 的处理函数。每当子进程退出时，进程都会收到该信号，并调用 sig_cld 进行处理。如果不进行 SIGCLD 信号的处理，系统中将产生僵尸进程。
- 第 17～27 行：计算阶乘的函数。输出参数为正整数，返回值为 long 型的阶乘结果。
- 第 26～40 行：建立流式套接口。
- 第 41～46 行：设置套接口的 SO_REUSEADDR 选项。
- 第 47～55 行：绑定套接口到本机的 5000 端口。
- 第 56～59 行：设置信号 SIGCLD 的处理方式。
- 第 60～63 行：调用 listen 在套接口上开始监听。调用 listen 后，客户端即可以调用 connect 进行连接。

- 第 64～95 行：接受并处理客户连接的循环，主要的业务逻辑都在该循环中完成。
- 第 65～70 行：调用 accept 接受一个客户端连接。注意：accept 调用成功后，将返回由内核产生的新的套接口。此后与客户端的数据收发都在该套接口上进行。
- 第 71 行：接受到客户端连接后，立即派生一个子进程，对该连接进行处理。
- 第 72～89 行：子进程处理的模块。
- 第 92～93 行：父进程处理的模块。父子进程的判断是依据 fork 调用返回值进行的。如果 fork 调用返回 0，表明是在子进程中，如果大于 0，表明是在父进程中；如果小于 0，表明 fork 调用出错。
- 第 73 行：关闭监听套接口。注意：此时是在多进程的环境中，也就是说监听套接口的引用计数是 2（调用 fork 后，父进程中的套接口资源被子进程复制，套接口的引用计数加 1），这里关闭监听套接口的意义在于减少套接口的引用计数为 1。否则，在子进程并发到一定程度后，监听套接口的引用计数将累计到一定值。此时，将不能产生新的套接口（系统为每个进程设置了打开套接口的最大数量）。所以，此处关闭监听套接口并不会真正意义上关闭该套接口。
- 第 74～79 行：接收客户端发送的数据。从该程序的应用需求看，该值是一个正整数。此处调用 recv 时，用到一个整型变量。注意：此处传递的是整型变量的指针。
- 第 80 行表示：调用函数计算从客户端传入的正整数的阶乘。
- 第 81～86 行：调用 send 向客户端返回 long 型的阶乘结果。
- 第 87 行：关闭与客户端连接的套接口。在第 73 的解释中介绍过，目前是处于多进程并发的环境下。监听套接口及与客户端连接的新的套接口的引用计数都是 2，此处关闭套接口只是将引用计数减为 1。要真正关闭套接口，必须还要对该套接口调用一次 close。这另外的一次关闭可以看到在第 92 行由父进程进行了一次关闭。
- 第 92～93 行：父进程处理过程。关闭客户端套接口（此处关闭的意义请参见第 87 行的解释），并进入下一个循环，继续接受客户端连接。

编写与范例 20-15 对应的客户端程序，输入正整数，并发送至服务器。由服务器计算阶乘并返回客户端。其实现过程如代码 20-16 所示。

示例代码 20-16

```
01  #include <stdio.h>                                    /*头文件*/
02  #include <sys/socket.h>
03  #include <errno.h>
04  #include <netdb.h>
05  #include <netinet/in.h>
06  #include <arpa/inet.h>
07  main(int argc,char *argv[])                           /*主函数*/
08  {
09      int n,nSend;                                      /*定义整型变量*/
10      int nSocket;                                      /*定义整型变量*/
11      char buf[21];                                     /*字符数组*/
12      long lResult;                                     /*长整型变量*/
13      struct sockaddr_in sin;                           /*结构变量*/
14      if(argc != 2)                                     /*判断命令行参数是否符合要求*/
15      {
16          fprintf(stderr,"Usage:%s number\n",argv[0]);/*输出错误提示*/
```

```
17          exit(0);
18      }
19      nSend = atoi(argv[1]);                                          /*转换为整型*/
20      if(nSend < 0)                                                   /*判断是否大于 0*/
21      {
22          fprintf(stderr,"The number must great than zero.\n");       /*输出错误提示*/
23          exit(0);
24      }
25      if((nSocket = socket(PF_INET,SOCK_STREAM,0)) < 0)               /*创建套接口*/
26      {
27          fprintf(stderr,"socket error:%s\n",strerror(errno));        /*输出错误提示*/
28          exit(0);
29      }
30      bzero(&sin,sizeof(struct sockaddr_in));                         /*清空结构*/
31      if(inet_pton(AF_INET,"10.232.70.81",&(sin.sin_addr)) != 1)      /*转换 IP 地址*/
32      {
33          fprintf(stderr,"inet_pton error:%s\n",strerror(errno));     /*输出错误提示信息*/
34          exit(0);
35      }
36      sin.sin_family = AF_INET;                                       /*Interne 地址族*/
37      sin.sin_port = htons(5000);                                     /*要连接 5000 端口*/
38      if(connect(nSocket, (struct sockaddr *)&sin,sizeof(struct sockaddr_in)) < 0)/*执行连接*/
39      {
40          fprintf(stderr,"connect error:%s\n",strerror(errno));       /*输出错误提示信息*/
41          close(nSocket);                                             /*关闭套接口*/
42          exit(0);
43      }
44      send(nSocket, (void *)&nSend,sizeof(int) ,0);                   /*发送数据*/
45      recv(nSocket, (void *)&lResult,sizeof(long) ,0);                /*接收数据*/
46      printf("%ld\n",lResult);                                        /*输出接收到的数据*/
47      close(nSocket);                                                 /关闭套接口*/
48      return;
49  }
```

【运行结果】经过编译链接，在 shell 下运行程序，计算 5 的阶乘，其结果如下所示。

```
01  ./example20_16 5
02  120
```

【代码解析】在本例中，首先，创建一个流式套接口并与指定的服务器建立连接；然后，调用 send 向服务器发送数据，并调用 recv 接收服务器的响应；最后，将服务器返回数据输出。源代码中相关行解释如下。

- 第 44 行：向服务器端发送数据。
- 第 45 行：从服务器端获取计算结果。

20.6 UDP 套接口编程

UDP 也是一种传输层协议。与 TCP 协议相比较，UDP 协议是一种非连接的、不可靠的数据报文协议。虽然 UDP 有不足，但是还是有很多网络应用使用它，如 DNS、SNMP 等。通常情况下，UDP 客户端程序不需要与服务器程序建立连接，直接使用 sendto 来发送数据。

同样，UDP 服务器也不需要接收客户端程序的连接，而是直接用 recvfrom 来接收客户端发送的数据。在本节中，首先介绍 UDP 编程模型，然后结合具体实例讲解实现过程。

20.6.1　UDP 编程模型

从理论上讲，UDP 服务器是面向非连接的。所以，不会出现类似于 TCP 服务器的一个客户端占用连接后，其他客户端无法连接的情况。因此，在 UDP 编程模式中，尽管也可以通过多进程实现并发服务器，但没有必要。因为普通的循环服务器基本上已能满足并发应用的需要。在 UDP 模式下的编程模型如图 20-12 所示。

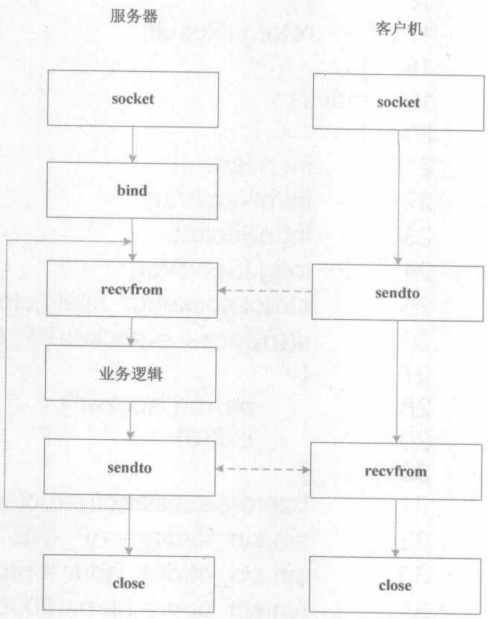


图 20-12　UDP 服务器编程模型

20.6.2　UDP 客户/服务器编程

UDP 编程也可以分为服务器与客户端两部分。其中服务器端编程的基本流程为以下几点。

（1）创建数据报套接口，并绑定地址及端口信息。

（2）接收来自客户端的数据报文。

（3）进行交易逻辑处理。

（4）向客户端发送应答。

（5）而客户端编程的基本流程为：①创建数据报套接口；②向指定地址及端口发送数据报文；③读取服务器的响应并输出。

下面的范例展示了这一编程的实现过程。

【范例 20-16】编写 UDP 服务器，接收从 UDP 客户端传入的正整数，计算其阶乘后向客户端返回。服务器实现如示例代码 20-17 所示。

示例代码 20-17

```
#include <stdio.h>                                          /*头文件*/
#include <sys/socket.h>
#include <errno.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>
long ComputeFact(int nFact)                                 /*计算阶乘*/
{
    int n;                                                  /*定义整型变量*/
    long lResult = 1;                                       /*定义长整型变量*/
    if(nFact < 0) return nFact;                             /*如果为负数，返回原值*/
    if(nFact == 0) return 1;                                /*0 的阶乘为 1*/
```

```
13      for(n=1;n<=nFact;n++)                                   /*循环进行累乘*/
14      {
15          lResult *= n;                                       /*累乘*/
16      }
17      return lResult;                                         /*返回结果*/
18  }
19  main()                                                      /*主函数*/
20  {
21      int n,len;                                              /*定义整型变量*/
22      int nRecvVar;                                           /*定义整型变量*/
23      int nSocket;                                            /*定义整型变量*/
24      long lSendVar;                                          /*定义整型变量*/
25      struct sockaddr_in sin,cin;                             /*定义结构变量*/
26      if((nSocket = socket(PF_INET,SOCK_DGRAM,0)) < 0)        /*创建套接口*/
27      {
28          perror("socket");                                   /*输出错误信息*/
29          exit(0);
30      }
31      bzero(&sin,sizeof(struct sockaddr_in));                 /*清空结构*/
32      sin.sin_family = AF_INET;                               /*Internet 地址族*/
33      sin.sin_addr.s_addr = htonl(INADDR_ANY);                /*绑定本机任何地址*/
34      sin.sin_port = htons(6000);                             /*绑定至 6000 端口*/
35      if(bind(nSocket, (struct sockaddr *)&sin,sizeof(struct sockaddr_in)) < 0)/*执行绑定*/
36      {
37          perror("bind");                                     /*输出错误提示信息*/
38          exit(0);
39      }
40      for(;;)                                                 /*循环接收数据*/
41      {
42          len=sizeof(struct sockaddr_in);                     /*计算结构的尺寸*/
43          bzero(&cin,sizeof(struct sockaddr_in));             /*清空结构*/
44          n = recvfrom(nSocket, (void *)&nRecvVar,sizeof(int) ,0,\   /*从缓冲区接收数据*/
45                       (struct sockaddr *)&cin,&len);
46          lSendVar = ComputeFact(nRecvVar);                   /*计算阶乘*/
47          len=sizeof(struct sockaddr_in);                     /*计算结构的尺寸*/
48          n = sendto(nSocket, (void *)&lSendVar,sizeof(long),0,\/*将阶乘的结果发送回客户端*/
49                     (struct sockaddr *)&cin,len);
50          if(nRecvVar < 0) break;                             /*收到的数据小于 0，退出循环*/
51      }
52      close(nSocket);                                         /*关闭循环*/
53      return;
54  }
```

【运行结果】经过编译链接，在 shell 下运行服务器程序，运行过程中未提示错误，表明服务器启动成功，其结果如下所示。

```
01  ./example20_17
```

【代码解析】在本例中，首先创建一个数据报套接口并绑定到指定的端口；然后调用 recvfrom 从客户端接收数据，计算该数据的阶乘并返回结果给客户端。源代码中相关行解释如下。

- 第 7～8 行：计算阶乘的函数。
- 第 26～30 行：建立数据报套接口。

- 第 31～39 行：绑定套接口到指定的端口号 6000。
- 第 40～51 行：循环接收来自客户端的请求，计算阶乘并向客户端发送应答。如果收到数据小于 0，则退出循环。
- 第 52 行：关闭套接口。

编写与范例 20-17 对应的 UDP 客户端程序，输入正整数，并发送至服务器。由服务器计算阶乘并返回客户端。其实现过程如代码 20-18 所示。

示例代码 20-18

```
#include <stdio.h>                                          /*头文件*/
#include <sys/socket.h>
#include <errno.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>
main(int argc,char *argv[])                                 /*主函数*/
{
    int n,nSend,len;                                        /*定义整型变量*/
    int nSocket;                                            /*定义整型变量*/
    long lResult;                                           /*定义长整型变量*/
    struct sockaddr_in sin;                                 /*定义结构变量*/
    if(argc != 2)                                   /*判断命令行参数是否符合要求*/
    {
        fprintf(stderr,"Usage:%s number\n",argv[0]);        /*输出用法提示信息*/
        exit(0);
    }
    nSend = atoi(argv[1]);                                  /*字符串转换为整型*/
    if((nSocket = socket(PF_INET,SOCK_DGRAM,0)) < 0)        /*创建套接口*/
    {
        fprintf(stderr,"socket error:%s\n",strerror(errno));/*输出错误提示信息*/
        exit(0);
    }
    bzero(&sin,sizeof(struct sockaddr_in));                 /*清空结构*/
    if(inet_pton(AF_INET,"10.232.70.81",&(sin.sin_addr)) != 1)  /*转换字符串 IP*/
    {
        fprintf(stderr,"inet_pton error:%s\n",strerror(errno));/*输出错误提示信息*/
        exit(0);
    }
    sin.sin_family = AF_INET;                               /*Internet 地址族*/
    sin.sin_port = htons(6000);                             /*要发送至 6000 端口*/
    if(sendto(nSocket, (void *)&nSend,sizeof(int) ,0,\      /*发送数据*/
            (struct sockaddr *)&sin,sizeof(struct sockaddr_in)) < 0)
    {
        fprintf(stderr,"sendto error:%s\n",strerror(errno));/*输出错误提示*/
        close(nSocket);                                     /*关闭套接口*/
        exit(0);
    }
    len = sizeof(struct sockaddr_in);                       /*计算结构的尺寸*/
    recvfrom(nSocket, (void *)&lResult,sizeof(long) ,0,\    /*从套接口接收数据*/
            (struct sockaddr *)&sin,&len);
    printf("%ld\n",lResult);                                /*输出接收到的数据*/
    close(nSocket);                                         /*关闭套接口*/
    return;
```

```
45  }
```

【运行结果】经过编译链接，在 shell 下运行程序，计算 5 的阶乘，其结果如下所示。

```
01  ./example20_18 5
02  120
```

【代码解析】在本例中，首先创建一个数据报套接口，然后调用 sendto 向服务器发送数据，并调用 recvfrom 接收服务器的响应，最后将服务器返回数据输出。源代码中相关行解释如下。

- 第 32～38 行：调用 sendto 向服务器发送数据。
- 第 39～41 行：调用 recvfrom 从服务器接收应答。
- 第 42～44 行：输出服务器的应答并关闭套接口返回。

20.7 小结

网络编程是 Linux 编程中非常重要的一部分。在本章中，重点介绍了套接口编程的相关知识，其中对于 TCP 并发服务器的设计尤为重要，读者可以在实际应用中加强对该部分内容的理解。另外，受篇幅所限，网络编程的一部分高级内容在本书中未进行深入讨论，包括 I/O 多路复用实现并发服务器以及 Linux 下的 EPOLL 编程模型，感兴趣的读者可以进一步研究学习。

第21章　综合实例——银行代理收费服务器

通过前面章节的学习，读者对 Linux 系统下编程的各个环节都有了一定了解。在本章中，将以一个银行代理收费的服务器程序为例，综合介绍 Linux 系统下编程的过程。通过本章的学习，读者应重点掌握如下内容。

- 加深对前面章节内容的理解。
- 具有编写初步 Linux 应用程序的能力。

21.1　程序需求

所谓代理收费，就是指银行代理收取各种费用，如水费、电费、电话费等。以代收电话费为例，通常情况下，在银行与通信公司之间会建立 DDN 专用通信线路，然后根据双方制订的通讯协议，编写通讯程序，实现数据的收发。

本例中要实现的功能是银行端的服务器模块。它负责从银行网点接收客户的收费报文，然后向通讯公司转发报文。通信公司进行账务处理后将处理结果返回银行服务器，银行服务器再向其网点返回缴费信息并打印各种发票。系统的架构如图 21-1 所示。

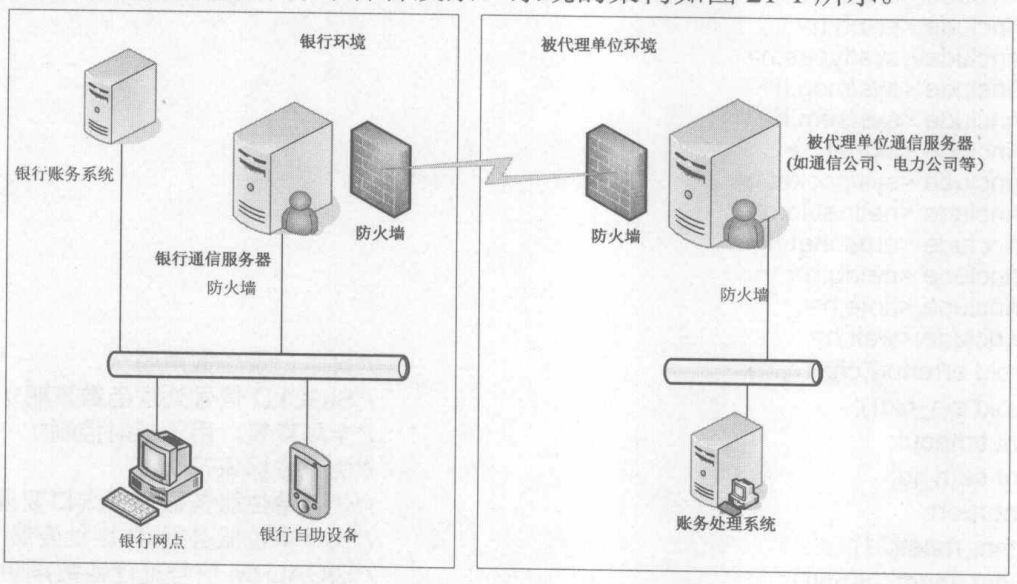


图 21-1　系统架构

21.2　程序实现

在本小节中，将根据前面小节的程序需要，来分别实现程序代码。由于整个程序代码会

比较庞大，下面将分步骤详细介绍。

1．配置文件

为实现参数的可配置化管理，将程序相关的参数保存到配置文件中。根据程序的需要，将本地服务器的地址和监听端口、被代理单位的服务器地址和监听端口保存到配置文件中。配置文件格式如下所示。

```
本地服务器 IP 本地监听端口 被代理单位服务器 IP 被代理单位服务器监听端口
```

2．命令行管理

程序的命令行支持 3 种选项，分别是“-s”启动系统、“-x”停止系统、“-c”指定配置文件名称。其中“-s”和“-x”二者必选其一，“-c”选项必须输入并且要指定配置文件。

3．通讯报文格式

在进行通讯时，规定报文内容前面使用 4 个字节的报文长度字段。该长度字符所指定的长度不包括长度字段本身的长度。

4．程序代码

下面将详细介绍本案例的程序代码。

（1）头文件: commserv.h 的代码如下所示。

```
#include <stdio.h>                          /*头文件*/
#include <stdarg.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/msg.h>
#include <sys/sem.h>
#include <signal.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <time.h>
#include <wait.h>
void errorlog(char *,...);                  /*日志记录函数声明*/
void sig_cld();                             /*SIGCLD 信号处理函数声明*/
int timeout;                                /*全局变量，用于超时控制*/
int sem_id;                                 /*信号量标识符*/
int rport;                                  /*代理单位服务器监听端口变量*/
char rhost[21];                             /*代理单位服务器 IP 地址变量*/
void handtimeout();                         /*SIGALRM 信号处理函数声明*/
char *handletime();                         /*格式化时间处理函数声明*/
#define MSG_PERM          0777              /*宏定义 消息队列访问权限*/
#define FTOKSEQ           66                /*宏定义 用于 ftok*/
#define MAXQLEN           100               /*宏定义 最大消息长度*/
#define MAXPROC           20                /*宏定义 最大并发度*/
#define PACKLEN           1024              /*宏定义最大通讯报文长度*/
#define SERVER_STARTUP    1                 /*宏定义 启动服务器标志*/
#define SERVER_SHUTDOWN 0                   /*宏定义 停止服务器标志*/
```

```
typedef struct {                                    /*自定义数据类型，用于保存配置信息*/
        char mhost[21];                             /*本地服务器地址*/
        int mport;                                  /*本地服务器端口*/
        char rhost[21];                             /*代理单位服务器地址*/
        int rport;                                  /*代理单位服务器端口号*/
} XMGL;
```

（2）函数源程序文件 commtools.c 的代码是本例的主要代码。其中去除字符串的左右两侧的函数代码如下所示。

```
#include "commserv.h"                               /*自定义头文件*/
#include <getopt.h>                                 /*获取命令行参数需要的头文件*/
char *lrtrim(char *buf)                             /*去除字符串的左右两侧空格*/
{
        int len;                                    /*定义整型变量*/
        char *p;                                    /*定义字符串变量*/
        while(isspace(*buf++));                     /*循环检查每个字符是否为空格*/
        p=--buf;                                    /*指向字符串中的下一个字符*/
        len=strlen(p);                              /*计算剩下的字符串长度*/
        while(isspace(p[--len])) p[len]=0;          /*循环处理字符串，将左侧空格去除*/
        return(p);                                  /*返回处理后的字符串*/
}
```

（3）记录日志定义的代码如下所示。

```
void errorlog(char *fmt, ...)                       /*记录日志定义，支持可变参数*/
{
        FILE    *fp;                                /*流文件*/
        char    *rq,yr[6],fname[11];                /*字符串变量*/
        va_list para;                               /*定义 va_list 类型变量*/
        va_start(para,fmt);                         /*固定调用方式*/
        rq=(char*)handletime(0);                    /*获得当前时间*/
        strncpy(yr,rq,5);                           /*取年月*/
        yr[5]=0;
        sprintf(fname,"./log%s",yr);                /*根据日期组装日志文件名*/
        fp=fopen(fname,"a");                        /*打开日志文件*/
        fprintf(fp,"[commserv][%s]:",rq);
        vfprintf(fp,fmt,para);                      /*向日志文件中输出日志*/
        fclose(fp);                                 /*关闭日志文件*/
        va_end(para);                               /*固定调用方式*/
}
```

（4）格式化输出时间的函数代码如下所示。

```
char *handletime(int bz)                            /*格式化输出时间*/
{
        struct tm *tp;
        long    secs;
        char    *buff;                              /*字符串变量*/
        buff=(char *)malloc(30);                    /*动态分配内存*/
        secs=time((long *)0);                       /*得到当前时间*/
        tp=localtime(&secs);                        /*格式化时间*/
        switch(bz){                                 /*判断需要的时间格式*/
                case 0:                             /*MM.DD HH:mm:SS*/
                sprintf(buff,"%02d.%02d %02d:%02d:%02d",tp->tm_mon+1,tp->tm_mday
,tp->tm_hour,tp->tm_min,tp->tm_sec);
```

```
41                      break;
42                      case 1:                                 /*YYYYMMDD*/
43                      sprintf(buff,"%04d%02d%02d",tp->tm_year+1900,tp->tm_mon+1,tp->tm
44  _mday);
45                      break;
46                      case 2:                                 /*YYYYMMDDHHmmSS*/
47                      sprintf(buff,"%04d%02d%02d%02d%02d%02d",tp->tm_year+1900,tp->tm_
48  mon+1,tp->tm_mday,tp->tm_hour,tp->tm_min,tp->tm_sec);
49                      break;
50                      default:                                /*YYYYMMDD*/
51                      sprintf(buff,"%04d%02d%02d",tp->tm_year+1900,tp->tm_mon+1,tp->tm
52  _mday);
53                      break;
54          }
55          return(buff);                                       /*返回格式化后的时间*/
56  }
```

（5）获取配置信息的代码如下所示。

```
57  int readpara(char *filename,XMGL *xs)                       /*从配置文件中获取配置信息*/
58  {
59          int       i;                                        /*定义整型变量*/
60          FILE      *fp;                                      /*流文件指针*/
61          char      buf[201];                                 /*字符数组变量*/
62          if((fp=fopen(filename,"r"))==NULL)                  /*打开配置文件读*/
63          {
64              errorlog("Open commserv.cfg error.");           /*输出错误提示*/
65              return(-1);
66          }
67          if(fgets(buf,100,fp) == NULL) return -1;            /*读取一行数据*/
68          sscanf(buf,"%s %d %s %d",xs->mhost,&xs->mport,\
69                  xs->rhost,&xs->rport);                      /*格式化输入到结构变量中*/
70          return(0);
71  }
```

（6）获取命令行参数的代码如下所示。

```
72  int CheckCmdLine(int gc,char *gv[],XMGL *xs)                /*获取命令行参数*/
73  {
74          char      c;                                        /*定义字符变量*/
75          int       nFlag=-1;                                 /*定义整型变量*/
76          char *opt_arg_value = NULL;                         /*定义字符串变量*/
77          struct option longopts[] = {                        /*长命令行结构参数*/
78              { "startup", no_argument, NULL,'s'},            /*长格式：--startup  短格式:-s*/
79              { "shutdown", no_argument, NULL,'x'},           /*长格式：--shutdown  短格式:-x*/
80              { "config", required_argument, NULL,'c'},/*长格式：--config  短格式:-c  需要参数值*/
81              {       NULL,       0,       NULL,       0},
82          };
83          while((c = getopt_long(gc, gv, ":sxc:", longopts, NULL)) != -1)/*循环解析命令行参数*/
84          {
85              switch (c)
86              {
87                  case 's':                                   /*-s 参数，启动系统*/
88                      nFlag = SERVER_STARTUP;                 /*设置标志*/
89                      break;
90                  case 'x':                                   /*-x 参数，停止系统*/
91                      nFlag = SERVER_SHUTDOWN;                /*设置标志*/
```

```
92                          break;
93                      case 'c':                                   /*-c 参数，指定配置文件*/
94                          opt_arg_value = optarg;                 /*复制-c 选项附带的数据*/
95                          break;
96                      case ':':                                   /*参数需要输入参数值，实际未输入*/
97                          fprintf(stderr,"argument %c need value.\n",optopt); /*输出错误提示*/
98                          break;
99                      case '?':                                   /*无效的参数*/
100                         fprintf(stderr,"Invalidate argument %c .\n",optopt); /*输出错误提示*/
101                         break;
102                     default:                                    /*其他情况*/
103                         break;
104                 }
105         }
106         if(nFlag == -1 || opt_arg_value == NULL)                /*如果未指定-c 的参
    数或者没有指定-s –x 其中之一*/
107         {
108             fprintf(stderr,"Usage:%s [--startup{-s}|--shutdown{-x}] \
109                                 -config{-c} filename\n",gv[0]);  /*提示用法*/
110             return -1;
111         }
112         if(opt_arg_value != NULL)
113         {
114             readpara(opt_arg_value,xs);                         /*从配置文件读取配置信息*/
115         }
116         return nFlag;
117 }
```

（7）消息发送函数的代码如下所示。

```
118 msg_snd(int mid,void *msg_text,int len)                        /*消息发送函数定义*/
119 {
120         int i;                                                  /*整型变量定义*/
121         for(i=0;i<5;i++)                                        /*循环发送，直到发送成功*/
122         if(!msgsnd(mid,(void *)msg_text,len,IPC_NOWAIT))        /*非阻塞发送消息*/
123         return;
124 }
125 /*信号量创建函数,如果信号量已存在，则返回信号量描述符*/
126 int     sem_create(key_t key,int num_sem)                       /*函数定义*/
127 {
128         int     id,i;                                           /*整型变量*/
129
130         if (key == IPC_PRIVATE)                                 /*不允许由系统生成键值*/
131                 return (-1);
132         else if (key == (key_t)-1)                              /*键值==-1*/
133                 return (-1);
134       if ((id = semget(key,num_sem,0666|IPC_CREAT|IPC_EXCL)) < 0) /*创建信号量*/
135       {
136                 if(errno==EEXIST)                               /*信号量已存在的处理*/
137                 {
138                         if((id=semget(key,num_sem,0))<0)        /*返回信号量的描述符*/
139                                 return (-1);
140                         else return(id);
141                 }
142                 else return (-1);
143         }
```

```
144             return id;                                              /*返回信号量 ID*/
145 }
```

（8）设置信号量数值的代码如下所示。

```
146 int set_sem(int id,int sem_no,int init_val)                           /*设置信号量的值*/
147 {
148             union    semun    {
149                                 int        val;
150                                 struct semid_ds *buf;
151                                 ushort              *array;
152                                 } semctl_arg;                       /*联合变量定义*/
153                     semctl_arg.val = init_val;                      /*要设置的值*/
154                     if (semctl(id,sem_no,SETVAL,semctl_arg) < 0)    /*设置信号量的值*/
155                     {
156                         return(-1);
157                     }
158                     else
159                         return(0);
160 }
161 int get_sem(int id,int sem_no)                                      /*得到信号量的值*/
162 {
            union    semun    {
163                                 int        val;
164                                 struct semid_ds *buf;
165                                 ushort              *array;
166                                 } semctl_arg;                       /*联合变量定义*/
167                 return(semctl(id,sem_no,GETVAL,semctl_arg)); /*调用 semctl 返回信号量的值*/
168 }
```

（9）操作信号量的代码如下所示。

```
169 /*信号操作函数  id:信号量标识符  num_sem 信号量编号  amount:信号量操作值*/
170 int sem_op(int id,int num_sem,int amount)
171 {
172             int fh=0;                                               /*定义整型变量*/
173             struct sembuf op_op[1];                                 /*结构数组变量*/
174             op_op[0].sem_num=num_sem;                               /*设置要操作的信号量序号*/
175             op_op[0].sem_op=amount;                                 /*设置信号量的值*/
176             op_op[0].sem_flg|=IPC_NOWAIT;                           /*非阻塞方式操作*/
177             fh=semop(id,&op_op[0],1);                               /*操作信号量*/
178             return fh;
179 }
180
181 /*P 操作函数  id:信号量标识符  num_sem:信号量编号*/
182 int sem_wait(int id,int num_sem)
183 {
184             int fh=0;                                               /*定义整型变量*/
185             fh=sem_op(id,num_sem,-1);                               /*调用 sem_op 设置信号量*/
186             return fh;
187 }
188 /*V 操作函数  id:信号量标识符  num_sem:信号量编号*/
189 int sem_signal(int id,int num_sem)
190 {
191             int fh=0;                                               /*定义整型变量*/
192             fh=sem_op(id,num_sem,1);                                /*调用 sem_op 设置信号量*/
193             return fh;
```

```
194 }
195 /*删除信号量*/
196 void free_sem(int semid)
197 {
198             sem_signal(semid,0);                                    /*释放信号量*/
199             semctl(semid,0,IPC_RMID);                               /*删除信号量*/
200 }
201 int get_pid(long key,int semnum,int sem_no)                  /*得到上次对信号量操作进程号*/
202 {
203             int       semid;
204             int       id;
205             union semun {
206                                 int       val;
207                                 struct semid_ds *buf;
208                                ushort *array;
209                                 } arg;                              /*定义联合变量*/
210             if((semid=semget((key_t)key,semnum,0))<0)               /*得到信号量标识符*/
211                         return(-1);
212             if((id=semctl(semid,sem_no,GETPID,arg))<0) /*调用 semctl 得到上次访问的进程标识*/
213             return(-1);
214             else
215             return(id);                                             /*返回得到的进程 ID*/
216 }
```

（10）从本步骤开始，用户需要编写主函数源程序文件:commserv.c 的代码，其中声明和定义代码段如下所示。

```
1     #include "commserv.h"                                           /*包含自定义的头文件*/
2     main(argc,argv)                                                 /*主函数*/
3     int       argc;
4     char      *argv[];
5     {
6               key_t   semkey;                                       /*定义 key_t 变量*/
7               int       nCmdFlag;                                   /*定义整型变量*/
8               int       size,chid,len,i,pid,cpid,spid;              /*定义整型变量*/
9               int       hsockfd,csockfd,n,c_len,mport;              /*定义整型变量*/
10              char      lhost[21],ip[21],mhost[21];                 /*定义字符数组变量*/
11              struct sockaddr_in sin,c_add,s_add;                   /*套接口地址结构*/
12              struct msqid_ds mqst;                                 /*消息队列控制结构变量*/
13              struct sigaction act,oact;                          /*定义结构变量，用于信号处理*/
14              struct hostent *hp;                                   /*主机地址结构指针变量*/
15              XMGL      *xmgl;                                      /*自定义数据类型的指针*/
16              xmgl=(XMGL *)malloc(sizeof(XMGL));                    /*分配缓存区*/
17              bzero(xmgl,sizeof(XMGL));                             /*清空缓存区*/
```

（11）检测获取信息的代码如下所示。

```
18              if((nCmdFlag = CheckCmdLine(argc,argv,xmgl)) < 0)           /*检查命令行参数*/
19              {
20                   free(xmgl);                                      /*错误处理,释放内存并返回*/
21                   return 1;
22              }
23              semkey=ftok("/etc/profile",FTOKSEQ);                  /*生成键值*/
24              for(i=0;i<32;i++)                                     /*忽略绝大部分信号*/
25              if(i!=SIGKILL&&i!=SIGBUS&&i!=SIGALRM)                 /*循环设置忽略信号*/
26              signal(i,SIG_IGN);
```

```
27          if(nCmdFlag == SERVER_STARTUP)                    /*启动服务器*/
28          {
29              strcpy(mhost,xmgl->mhost);                    /*获取本地地址*/
30              mport=xmgl->mport;                            /*获取本地端口*/
31              gethostname(lhost,20);                        /*得到当前主机名*/
32              hp=gethostbyname(lhost);                      /*根据主机名进行地址解析*/
33              memcpy((char *)&sin.sin_addr,(char *)hp->h_addr,hp->h_length);/*得到当前主机
    的 IP*/
34              strcpy(ip,inet_ntoa(sin.sin_addr));           /*转换 IP 为字符串形式*/
35              if(strcmp(mhost,lhost)&&strcmp(mhost,ip))     /*检查与文件中指定地址一致*/
36              {
37                  fprintf(stderr,"服务未在本机定义.[%s].\n", mhost);  /*输出错误提示信息*/
38                  free(xmgl);                               /*释放动态申请的内存*/
39                  exit(0);
40              }
41              strcpy(rhost,xmgl->rhost);                    /*得到对方主机地址*/
42              rport=xmgl->rport;                            /*得到对方主机端口*/
43              free(xmgl);                                   /*释放缓存*/
44              pid=setpgrp();                                /*设置新进程组*/
45              errorlog("Starting commserv...\n");           /*记录错误日志*/
46              if((sem_id=sem_create(semkey,1))<0)           /*创建信号量*/
47              {
48                  errorlog("Sem_create failed.\nStart failed.\n");  /*记录错误日志*/
49                  exit(-1);
50              }
51              spid=get_pid(semkey,1,0);                     /*得到上次访问信号量的进程*/
52              if(spid>0&&spid!=getpid()){  /*上次访问的进程不是进程，表明服务器已启动*/
53                  fprintf(stderr,"The system has been started.\n");  /*输出错误提示信息*/
54                  exit(-1);
55              }
56              set_sem(sem_id,0,MAXPROC);                    /*设置信号量的值为最大
    流量，该信号量用于控制流量*/
57              errorlog("Startup Done.\n");                  /*记录错误日志*/
58          }
59          else                                              /*关闭服务器*/
60          {
61              free(xmgl);                                   /*释放内存*/
62              if((sem_id=sem_create(semkey,1))<0)           /*得到信号量标识符*/
63              {
64                  errorlog("Sem_create failed.\nStart failed.\n");  /*记录错误日志*/
65                  exit(-1);
66              }
67              errorlog("Stopping commserv...\n");           /*记录错误日志*/
68              pid=get_pid(semkey,1,0);                  /*得到上次访问信号量的进程 ID*/
69              pid=(-1)*pid;                                 /*将进程 ID 乘以-1*/
70              free_sem(sem_id);                             /*删除信号量*/
71              errorlog("Shutdown Done.\n");                 /*记录错误日志*/
72              kill(pid,SIGKILL);                        /*向服务器进程组发送 KILL 信号*/
73              exit(0);
74          }
75          if((cpid=fork())<0){                              /*创建新进程*/
76              errorlog("Fork:%s\n",strerror(errno));        /*记录错误日志*/
77              exit(-1);
78          }
```

```
79          if(cpid!=0)                                                    /*父进程退出*/
80          exit(0);
81          signal(SIGALRM,handtimeout);                                   /*安装 SIGALRM 信号*/
82          for(;;)                                                /*循环创建套接口，直到创建成功*/
83          {
84              if((hsockfd=socket(PF_INET,SOCK_STREAM,0))<0)   /*创建流式套接口*/
85              {
86                  errorlog("Socket:%s\n",strerror(errno));           /*记录错误日志*/
87                  sleep(2);                                          /*休眠 2 秒后继续创建*/
88              }
89              else break;
90          }
91          bzero((char *)&s_add,sizeof(s_add));                       /*清空结构*/
92          s_add.sin_family            = AF_INET;                     /*AF_INET 地址族*/
93          s_add.sin_addr.s_addr     = htonl(INADDR_ANY);             /*绑定任何主机地址*/
94          s_add.sin_port              = htons(mport);   /*设置要绑定的端口,注意转换字节序*/
95          size=1;
96          len=sizeof(size);                                          /*得到变量的尺寸*/
```

（12）设置 SO_REUSEADDR 选项的代码段如下所示。

```
97          /*设置 SO_REUSEADDR 选项*/
98          setsockopt(hsockfd,SOL_SOCKET,SO_REUSEADDR,(char *)&size,len);
99          for(;;)                                                /*循环绑定，直到绑定成功*/
100         {
101             /*绑定套接口到指定的地址和端口*/
102             if(bind(hsockfd,(struct sockaddr *)&s_add,sizeof(s_add))<0)
103             {
104                 errorlog("Bind:%s\n",strerror(errno));         /*记录错误日志*/
105                 sleep(2);                                      /*休眠 2 秒，继续循环绑定*/
106             }
107             else break;                                        /*绑定成功，退出循环*/
108         }
109
110         listen(hsockfd,5);                                     /*开始监听*/
111
112         act.sa_handler=sig_cld;                                /*设置信号处理函数*/
113         sigemptyset(&act.sa_mask);                             /*清空信号处理掩码*/
114         act.sa_flags=0;                                        /*清空信号处理标志位*/
115         act.sa_flags=0;
116         sigaction(SIGCLD,&act,&oact);                          /*安装 SIGCLD 信号*/
117         for(;;){                                               /*循环接受客户连接*/
118             c_len=sizeof(c_add)
119             csockfd=accept(hsockfd,(struct sockaddr *)&c_add,&c_len);/*接受客户端连接*/
120             if(csockfd<0)                                      /*接受客户连接失败的处理*/
121             {
122                 if(errno==EINTR) continue;                     /*被中断，继续循环*/
123                 errorlog("Accept:%s\n",strerror(errno));       /*记录错误日志*/
124                 close(csockfd);                                /*关闭套接口*/
125                 continue;                                      /*继续循环*/
126             }
127             if(sem_wait(sem_id,0))                             /*进行可用的流量减少操作*/
128             {
129                 errorlog("Can't fork too many process.");      /*记录错误日志*/
130                 close(csockfd);                                /*关闭套接口*/
```

```
131                         continue;                                   /*继续循环*/
132                     }
133                     chid=0;
134                     if((chid=fork())!=0)                            /*创建子进程处理接入的连接*/
135                     {
136                         if(chid==-1)                                /*创建子进程失败的处理*/
137                         {
138                             errorlog("Fork:%s\n",strerror(errno));  /*记录错误日志*/
139                             sem_signal(sem_id,0);                   /*进行信号的V操作*/
140                         }
141                         close(csockfd);   /*关闭 csockfd。实际上是将 csockfd 的引用计数减 1*/
142                         continue;
143                     }
144                     close(hsockfd); /*子进程，关闭 hsockfd。实际上是将 hsockfd 的引用计数减 1*/
145                     handle_request(csockfd);                        /*调用函数处理请求*/
146                     /*关闭与客户端的套接口，在父进程中已进行过关闭，因此该客户端套接口将
    真正关闭*/
147                     close(csockfd);
148                     exit(0);                                        /*子进程退出*/
149             }
150 }
151 void handtimeout()                                                  /*SIGALRM 信号处理函数*/
152 {
153             timeout=1;                                              /*设置超时标志为 1*/
154 }
155 void sig_cld(int signo)                                             /*SIGCLD 信号处理函数*/
156 {
157             int status;
158             while(waitpid(-1,&status,WNOHANG)>0){                   /*等待子进程退出*/
159                 sem_signal(sem_id,0);                   /*释放流量，将流量控制信号量的值加 1*/
160             }
161 }
```

（13）处理新接入的客户连接的代码段如下所示。

```
162 int handle_request(csockfd)
163 int csockfd;                                                        /"与客户端连接的套接口*/
164 {
165             int   n,buflen,sockfd;                                  /*定义整型变量*/
166             char buf[PACKLEN+1],sendbuf[PACKLEN+1],retbuf[PACKLEN+1],lenbuf[5];
167             struct hostent              *hp;                        /*结构指针变量*/
168             struct sockaddr_in          sin;                        /*结构变量*/
169             memset((void *)buf,0,sizeof(buf));                      /*清空缓存区*/
170             n=readsocket(csockfd,buf,4);                            /*读取 4 位的报文长度*/
171             if(n!=4){                                               /*读取数据不足 4 位的处理*/
172                     errorlog("Read error");                         /*记录错误日志*/
173                     return(-1);
174             }
175             buf[4]=0;                                               /*字符串结束*/
176             buflen=atoi(buf);                                       /*取得报文内容长度*/
177             if(readsocket(csockfd,buf,buflen)!=buflen){             /*读取报文内容*/
178                     errorlog("Read error");                         /*记录错误日志*/
179                     return(-2);
180             }
181             errorlog("接收到网点报文:%s\n",buf);                    /*记录错误日志*/
```

```
182         memset(sendbuf,0x00,sizeof(sendbuf));                    /*清空缓存区*/
183         sprintf(sendbuf,"%04d%s",buflen,buf);
184         errorlog("向代理单位发送请求报文:%s\n",sendbuf);          /*记录错误日志*/
185         if((sockfd=socket(PF_INET,SOCK_STREAM,0))<0){            /*创建新的套接口*/
186                 errorlog("Socket:%s\n",strerror(errno));         /*记录错误日志*/
187                 return(-4);
188         }
189         if((hp=gethostbyname(rhost))==NULL){                     /*获得代理单位主机地址结构*/
190                 errorlog("Gethostbyname:%s\n",strerror(errno));          /*记录错误日志*/
191                 close(sockfd);                                           /*关闭套接口*/
192                 return(-5);
193         }
194         bzero((char *)&sin,sizeof(sin));                                 /*清空结构*/
195         sin.sin_family              = AF_INET;                           /*AF_INET 地址族*/
196         memcpy((char *)&sin.sin_addr,(char *)hp->h_addr,hp->h_length);       /*拷贝地址结构
    中的地址数据*/
197         sin.sin_port                = htons(rport);                      /*设置要连接的端口*/
198         timeout=0;                                                       /*复位超时标志*/
199         alarm(20);                                                       /*设置定时器*/
200         if(connect(sockfd,(struct sockaddr *)&sin,sizeof(struct sockaddr))<0){ /*与代理单位主
    机进行连接*/
201                 alarm(0);                                                /*取消定时器*/
202                 errorlog("Connect:%s\n",strerror(errno));                /*记录错误日志*/
203                 close(sockfd);                                           /*关闭套接口*/
204                 return(-6);
205         }
206         alarm(0);                                                        /*取消定时器*/
207         if(timeout==1){                                                  /*超时*/
208                 errorlog("Connect [%s][%d] timeout.\n",rhost,rport);/*记录错误日志*/
209                 timeout=0;                                               /*复位超时标志*/
210                 close(sockfd);                                           /*关闭套接口*/
211                 return(-7);
212         }
213         buflen +=4;                                                      /*发送长度加上长度本身*/
214         n=writesocket(sockfd,sendbuf,buflen);                        /*向代理单位发送数据报文*/
215         if(n!=buflen){                                                   /*发送的长度不符*/
216                 errorlog("Write:[%s]failed\n",buf);                      /*记录错误日志*/
217                 close(sockfd);                                           /*关闭套接口*/
218                 return(-8);
219         }
220         bzero(retbuf,sizeof(retbuf));                                    /*清空缓存区*/
221         n=readsocket(sockfd,retbuf,4);                           /*从代理单位接收应答报文长度*/
222         if(n!=4){                                                        /*接收到的长度不符*/
223                 close(sockfd);                                           /*关闭套接口*/
224                 return(-9);
225         }
226         retbuf[4]=0;                                                     /*字符串结束*/
227         buflen=atoi(retbuf);                                             /*转换为整型*/
228         bzero(retbuf,sizeof(retbuf));                                    /*清空缓存区*/
229         n=readsocket(sockfd,retbuf,buflen);                          /*从代理单位接收应答报文*/
230         if(n!=buflen)                                                    /*接收到的长度不符*/
231         {
232                 close(sockfd);                                       /*关闭套接口*/
```

```
233                     return(-11);
234             }
235             retbuf[buflen]=0;                                   /*字符串结束*/
236             errorlog("从代理单位接收应答报文:%s\n",retbuf);      /*记录错误日志*/
237             bzero(sendbuf,sizeof(sendbuf));                     /*清空发送缓存区*/
238             sprintf(sendbuf,"%04d%s",buflen,retbuf);            /*向发送缓存区写入数据*/
239             buflen += 4;                                        /*发送长度加上长度本身*/
240             errorlog("向网点返回应答报文:%s\n",sendbuf);         /*记录错误日志*/
241             if(writesocket(csockfd,retbuf,buflen)!=buflen){     /*向网点返回应答报文*/
242                     errorlog("Write [%s] failed\n",retbuf);     /*记录错误日志*/
243                     return(-14);
244             }
245             return(0);
246     }
```

（14）向套接口读取数据的函数的代码段如下所示。

```
247     int readsocket(int sockfd,char *buf,int len)                /*从套接口读取数据的函数*/
248     {
249             int n,inum;                                         /*定义整型变量*/
250             char errbuf[201];                                   /*错误信息缓存区*/
251             for(inum=0;inum<len;inum+=n){ /*循环读取指定长度的报文，直到全部读取完成或者超时*/
252                     timeout=0;                                  /*复位超时标志*/
253                     alarm(10+(len-inum)/100);                   /*设置定时器*/
254                     n=read(sockfd,&buf[inum],len-inum);         /*读取数据*/
255                     alarm(0);                                   /*取消定时器*/
256                     if(n<0){                                    /*读取失败的处理*/
257                             if(timeout==1){                     /*超时导致的失败*/
258                                     timeout=0;                  /*复位超时标志*/
259                                     errorlog("Read timeout");   /*记录错误日志*/
260                             }
261                             else{                               /*其他原因引起的失败*/
262                                     errorlog("Read:%s\n",strerror(errno)); /*记录错误日志*/
263                             }
264                             return(-1);
265                     }
266             }
267             return(inum);                                       /*返回读取到的字节数*/
268     }
```

（15）向套接口发送数据的函数的代码段如下所示。

```
269     int writesocket(int sockfd,char *buf,int len)               /*向套接口发送数据的函数*/
270     {
271             int n;                                              /*定义整型变量*/
272             char errbuf[201];                                   /错误输出缓存区*/
273             timeout=0;                                          /*复位超时标志*/
274             alarm(10+len/100);                                  /*设置定时器*/
275             n=write(sockfd,buf,len);                            /*发送数据*/
276             alarm(0);                                           /*取消定时器*/
277             if(n<0){                                            /*发送数据失败*/
278                     if(timeout==1){                             /*超时引起的失败*/
279                             timeout=0;                          /*复位超时标志*/
280                             errorlog("Write timeout");          /*记录错误日志*/
281                     }
```

```
282                     else{                                                   /*其他原因引起的失败*/
283                             errorlog("Write:%s\n",strerror(errno));         /*记录错误日志*/
284                     }
285                     return(-1);
286             }
287             return(n);                                                      /*返回发送的字节数*/
288 }
```

（16）Makefile 文件的代码如下所示。

```
1   # Make system version 1.0 programs.
2   #
3   # makefile 1.100 10/01/2008
4   #
5   # Change the following definitions as appropriate for your site:
6   INCDIR      = ./include
7   LIBDIR      = ./lib
8   INCLUDE= -I$(INCDIR)
9   CC=cc
10  all: commserv
11
12  OBJ=commserv.o commtools.o
13
14  commserv.o: commserv.h
15
16  commserv:$(OBJ)
17          cc $(OBJ) -lnsl -o $@
18
19  $(OBJ) :$(HEADERS)
20  .c.o:
21          cc -c -O $(INCLUDE) $*.c
```

5．程序运行结果

运行程序代码后，日志文件的内容如下所示。

```
1   [commserv][10.12 01:03:05]:Starting commserv...
2   [commserv][10.12 01:03:50]:Startup Done.
3   [commserv][10.12 01:04:00]:接收到网点报文:abcd
4   [commserv][10.12 01:04:00]:向代理单位发送请求报文:0004abcd
5   [commserv][10.12 01:04:00]:从代理单位接收应答报文:OKabcd
6   [commserv][10.12 01:04:00]:向网点返回应答报文:0006OKabcd
7   [commserv][10.12 01:05:00]:Stopping commserv...
8   [commserv][10.12 01:05:00]:Shutdown Done.
```

21.3　小结

本章讲解了一个银行代理收费服务器的实现方法，其主要功能是代理收取各种费用。主要的编程流程是：在银行与通信公司之间会建立 DDN 专用通信线路，根据通讯协议，编写通讯程序，实现数据的发送和接收。其中，数据的收发其中用到了网络编程的各种技术，具体的内容请用户查看前面章节的知识。读者应该从这个案例体会 Linux 编程的流程，从而为真正的项目开发做好准备。

附录　Linux 下常见 C 函数字母索引

函　数	说　明
A	
abort	进程异常终止
abs	计算整数的绝对值
accept	处理 socket 请求
access	判断进程是否具有访问文件的权限
acos	计算反余弦的值
alarm	内核定时器
alphasort	按照字母顺序排序目录结构
asctime	字符串时间操作
asin	计算反正弦的值
atan2	计算反正切的值
atan	计算反正切的值
atexit	进程终止处理函数
atof	将字符串转换成浮点数
atoi	将字符串转换成整数
atol	将字符串转换成长整数
B	
bcmp	比较两块内存中的内容
bcopy	复制内存中的内容
bind	将 socket 描述符与一个套接口绑定
brk	改变数据段大小
bsearch	在有序数组中二分查找数据
bzero	将一块内存内容全清零
C	
calloc	内存分配
ceil	计算大于或等于给定参数的最小整数
chdir	改变当前的工作目录
chmod	改变文件的权限
chown	改变文件的所有者
chroot	改变进程的根目录
clearenv	删除所有环境变量的值
clearerr	清除错误标志

续表

函　数	说　明
clock	计算运行时间
closedir	关闭目录文件
closelog	关闭信息记录
close	关闭已经打开的文件
connect	与远程主机连接
cosh	计算双曲线的余弦值
cos	计算余弦的值
creat	创建一个文件
crypt	对字符串使用 DES 算法编码
ctime	字符串时间
cuserid	获取用户名
D	
difftime	计算时间差
div	计算两整数相除后的商和余数
dlclose	关闭动态函数库文件
dlerror	动态函数出错处理
dlopen	打开动态库文件
dlsym	在动态库文件中查找函数
drand48	产生一个正的随机浮点数
dup2	复制文件描述符到指定的位置
dup	复制文件描述符
E	
ecvt	将浮点数转换成字符串
endgrent	关闭系统组文件函数
endprotoent	结束网络协议的读取函数
endpwent	关闭系统口令函数
endservent	关闭文件/etc/services
endutent	关闭 utmp 文件
erand48	产生一个正的随机浮点数
execle	执行文件
execlp	从 PATH 环境变量中查找文件并执行
execl	执行文件函数
execve	执行文件
execvp	执行文件
execv	执行文件

续表

函　数	说　明
exit	结束进程运行的函数
exp	计算浮点数的指数
F	
fabs	计算浮点数的绝对值
fchdir	使用文件描述符改变当前的工作目录
fchmod	改变一个已经打开的文件的权限
fclose	关闭流文件描述符
fcntl	改变文件的状态
fcvt	将浮点数转换成字符串
fdopen	转换文件描述符到文件指针
feof	文件结束判断函数
ferror	文件错误判断函数
fflush	刷新流文件读写缓存
ffs	在一个整数中查找第一个值为真的位
fgetc	从流文件中读取一个字符
fgetgrent	从文件中读取组数据
fgetpos	返回流文件当前读写位置
fgetpwent	从文件中读取密码格式数据
fgets	从流文件中读出一行数据
fileno	获取流文件的文件描述符
fopen	打开文件
fork	建立一个新的进程
fprintf	格式化输出到文件
fputc	将字符写入流文件
fputs	将字符串写入流文件
fread	从流文件中读出数据
free	释放动态分配内存函数
freopen	打开文件
frexp	计算浮点数的尾数和指数
fscanf	格式化从文件中读出数据
fseek	移动流文件的读写位置
fsetpos	设置流文件当前读写位置
fstat	得到一个打开文件的文件状态
fsync	将缓冲区数据回写到磁盘文件
ftell	得到流文件的读取位置

续表

函　数	说　明
ftime	取得系统时间
ftok	获取项目相关的 IPC 键值
ftruncate	改变一个打开文件的文件大小
ftw	遍历目录树
fwrite	向流文件中写入数据
G	
gcvt	将浮点数转换成字符串
get_current_dir_name	取得当前的工作目录
getchar	从标准输入中读入一个字符
getcwd	取得进程的当前工作目录
getc	从文件中读出一个字符
getegid	获取有效（effectivegid）组识别符
getegid	取得有效进程组 ID
getenv	取得环境变量的值
geteuid	获取有效（effectiveuid）用户识别符
getgid	获取组识别符
getgid	取得实际进程组 ID
getgrent	从文件/etc/group 中读取组信息
getgrgid	从文件/etc/group 中查找组信息
getgrnam	从文件/etc/group 中查找组信息
getgroups	返回当前用户所属的组列表
gethostbyaddr	由 IP 获取主机信息
gethostbyname	由主机名获取主机信息
getlogin	获取当前用户的账号名称
getopt	分析命令参数
getpagesize	获取操作系统内存页大小函数
getpass	取得用户输入密码字符串
getpid	取得有效组 ID
getppid	取得父进程的进程识别码
getpriority	取得进程执行优先级
getprotobyname	由协议名获取协议数据
getprotobynumber	由协议编号获取协议数据
getprotoent	读取文件/etc/protoent 中的网络协议数据
getpwent	读取/etc/passwd 文件中的数据
getpwnam	以用户名获取用户信息

续表

函　数	说　明
getpwuid	以用户 ID 获取用户信息
getpw	获取指定 uid 的用户信息
getservbyname	获取网络服务协议数据
getservbyport	获取网络服务协议数据
getservent	读取主机网络服务数据
getsockopt	获取指定 socket 描述符的状态
gets	从标准输入中读入一行数据
gettimeofday	取得系统当前时间
getuid	获取当前进程所属的用户 ID
getutent	获取登陆信息
getutid	获取指定用户 ID 的登录信息
getutline	查找指定的登陆信息
gmtime	日历时间格式函数
H	
hcreate	建立哈希表
hdestroy	删除哈希表
herror	错误原因输出函数
hsearch	在哈希表查找关键数据
hstrerror	错误代码查询
htonl	地址字节顺序转换
htons	地址字节顺序转换
hypot	计算直角三角形斜边长度
I	
index	查找字符串中第一个出现的指定字符
inet_addr	网络地址类型转换
inet_aton	网络地址类型转换
inet_ntoa	网络地址类型转换
initgroups	初始化组信息
initstate	建立随机数状态数组
isalnum	判断字符是否为英文字母或数字
isalpha	判断字符是否为英文字母
isascii	判断字符是否为 ASCII 字符
isatty	判断文件描述符是否为终端机
isblank	判断字符是否为空白字符
iscntrl	判断字符是否为 ASCII 的控制字符

续表

函　数	说　明
isdigit	判断字符是否为阿拉伯数字
isgraph	判断字符是否为可打印字符
islower	判断字符是否为小写英文字母
isprint	判断字符是否为可打印字符
ispunct	判断字符是否为标点符号
isspace	判断字符是否为空格字符
isupper	判断字符是否为大写英文字母
isxdigit	判断字符是否为十六进制数字
J	
jrand48	产生一个长整型的随机数
K	
kill	信号发送函数
L	
labs	计算长整数的绝对值
lchown	改变符号链接文件的拥有者
lcong48	设置 48 位运算的随机数种子
ldexp	计算浮点数与 2 的次方值的积
ldiv	计算两长整数相除后的商和余数
lfind	线性搜索
link	建立一个硬链接
listen	网络 socket 监听
localtime	获取当前系统时间
log10	计算以 10 为底的对数值
logwtmp	添加登陆记录信息
log	计算以 e 为底的对数值
longjmp	跳到原先 setjmp 存储的堆栈环境
lrand48	产生一个正的长整型随机数
lsearch	线性搜索
lseek	移动文件的读写位置
lstat	得到符号链接的文件状态
M	
malloc	动态内存分配
memccpy	复制内存中的内容
memchr	在一块内存指定范围查找一个指定字符
memcmp	比较内存中存放的内容

续表

函　数	说　明
memcpy	复制一块内存内容到另一块中
memfrob	对某个内存区重新编码
memmove	复制内存内容
memset	将某值填入到一块内存区域
mkfifo	创建 fifo 管道函数
mktemp	产生临时文件的文件名
mktime	系统时间转换函数
mmap	文件映射
modf	将浮点数分解为小数和整数两部分
mrand48	产生一个长整型随机数
msgctl	消息队列操作
msgget	创建或打开消息队列
msgrcv	读取消息队列
msgsnd	向消息队列中写入消息
munmap	释放映射的内存地址
N	
nftw	遍历目录树
nrand48	产生一个正的长整数随机数
ntohl	地址字节顺序转换
ntohs	地址字节顺序转换
O	
on_exit	设置程序正常结束前调用的函数
opendir	打开目录文件
openlog	打开记录信息
open	打开一个文件
P	
pause	进程暂停执行函数
pclose	关闭管道文件
perror	错误原因输出函数
pipe	创建匿名管道
popen	创建管道文件
pow	计算次方值
printf	格式化输出
pselect	多路转接 I/O
putchar	将字符输出到标准输出中

续表

函　数	说　明
putc	将字符写入文件
putenv	改变环境变量的值
puts	将字符串输出在标准输出中
pututline	添加登录记录信息
Q	
qsort	使用快速排序法排列数组
R	
raise	向进程自身发送信号函数
random	产生随机数
rand	产生伪随机数
readdir	读取目录文件的内容
readlink	读取符号链接文件本身的内容
read	读取文件的数据
realloc	内存截取函数
realpath	将相对目录路径转换成绝对路径
recvfrom	接收消息
recvmsg	多缓冲读取数据
recv	接收消息
regcomp	编译正则表达式
regerror	编译错误判断函数
regexec	执行正则匹配
regfree	释放正则模式串资源
remove	删除文件或者目录
rename	更改文件的名称或位置
rewinddir	回卷目录文件
rewind	将流文件的读写位置移动到文件的起始位置
rindex	查找字符串中最后一个出现的指定字符
S	
sbrk	增加数据空间
scanf	格式化输入
seed48	设置 48 位运算的随机数种子
seekdir	文件定位函数
select	多路 I/O 转接
semctl	信号量操作函数
semget	创建或打开信号量

续表

函　数	说　明
semop	释放或获取信号量
sendto	发送数据到远程主机
send	向远程主机发送数据
setbuffer	设置文件缓冲区
setbuf	设置文件缓冲区
setegid	设置有效组 ID
setenv	改变环境变量的值
seteuid	设置有效用户 ID
setfsgid	设置进程的文件系统的组 ID
setfsuid	设置进程的文件系统的用户 ID
setgid	设置进程的进程组 ID
setgid	设置进程的组 ID
setgrent	将文件 etc/group 的读写位置移动到起始位置
setgroups	设置当前进程的组识别符
setjmp	存储当前堆栈环境
setlinebuf	设置行缓冲模式
setpriority	设置程序进程优先级
setprotoent	打开网络协议文件
setpwent	将文件 etc/passwd 的读写位置移动到起始位置
setregid	设置当前进程的真实以及有效组 ID
setreuid	设置当前进程的真实以及有效用户 ID
setservent	打开网络服务文件
setsockopt	设置 socket 描述符的状态
setstate	建立随机数状态数组
settimeofday	系统时间设置函数
setuid	设置进程的用户 ID
setuid	设置进程组 ID
setutent	将文件 utmp 的读写位置移动到起始位置
setvbuf	设置自定义文件缓冲区
shmat	导入共享内存
shmctl	共享内存操作函数
shmdt	将共享内存从程序中脱离
shmget	创建共享内存
shutdown	结束 socket 套接字
sigaction	高级信号处理方式设置函数

续表

函　数	说　明
sigaddset	添加信号到信号集函数
sigdelset	删除信号集中对应信号函数
sigemptyse	清空信号集函数
sigfillset	填充信号集函数
sigismember	测试某个信号是否在信号集函数
siglongjmp	跳到原先 sigsetjmp 存储的堆栈环境
signal	信号处理方式设置函数
sigpause	暂停进程直到信号到来
sigpending	查询未决信号
sigprocmask	信号屏蔽函数
sigsetjmp	存储当前堆栈环境和屏蔽的信号集
sigsuspend	暂停直到信号到来
sinh	计算双曲线的正弦值
sin	取正弦的值
sleep	让进程休眠一段时间
snprintf	格式化字符输入
socket	创建网络套接字
sprintf	格式化字符串打印
sqrt	计算浮点数的平方根
srand48	设置 48 位运算的随机数种子
srandom	设置随机数种子
srand	设置随机数种子
sscanf	格式化字符串取值
stat	取得文件状态
strcasecmp	忽略大小写比较字符串
strcat	将一个字符串连接另一字符串的尾部
strchr	查找字符串中指定字符
strcmp	比较两个字符串
strcoll 的	根据当前环境信息来比较字符串
strcpy	复制一个字符串的内容到另一个字符串中。
strcspn	计算字符串中由非指定字符集字符组成的子字符串的长度
strdup	复制字符串内容
strerror	错误代码查询
strfry	随机重组一个字符串
strftime	时间格式化输出函数

续表

函　数	说　明
strlen	返回字符串长度
strncasecmp	忽略大小写比较两个字符串
strncat	将一个字符串的前 n 个字符连接另一字符串的尾部
strncmp	比较两个字符串
strncpy	复制字符串
strpbrk	查找字符串中第一个出现的指定字符
strrchr	查找字符串中最后一个出现的指定字符
strspn	计算字符串中由指定字符集字符组成的子字符串的长度
strtod	将字符串转换成浮点数
strtol	将字符串转换成长整数
strtoul	将字符串转换成无符号长整数
symlink	创建一个符号链接
sync	将所有打开的文件写回磁盘
syslog	输出记录信息
system	执行 shell 命令
T	
tanh	计算双曲线的正切函数值
tan	计算角度的正切值
tdelete	从二叉树中删除数据
telldir	取得目录文件的读取位置
tfind	搜索二叉树
time	获取系统当前时间
tmpfile	建立临时的二进制文件
toascii	将整数转换成合法的 ASCII 字符
tolower	将大写字母转换成小写字母
toupper	将小写字母转换成大写字母
truncate	改变文件大小
tsearch	搜索二叉树
ttyname	返回终端机的名称
twalk	遍历二叉树
tzset	设置系统时区变量 tzname
U	
umask	设置创建文件时使用的权限掩码
ungetc	将字符写回文件
unlink	删除文件的一个硬链接

续表

函　数	说　明
unsetenv	清除环境变量的值
updwtmp	将登陆数据记录写入 utmp 文件
utmpname	设置文件 utmp 的绝对位置
V	
vfork	建立一个新的进程
vfprintf	函数
vfscanf	格式化输入
vprintf	格式化输出函数
vscanf	格式化输入函数
vsnprintf	格式化字符串复制
vsprintf	格式化字符串复制
vsscanf	格式化字符输入
W	
wait3	等待进程结束并且输出子进程统计信息
waitpid	等待子进程中断或结束
wait	等待子进程中断或结束
write	输出内容到文件
_exit	结束进程运行

Document generated by Anna’s Archive around 2023-2024 as part of the DuXiu collection (https://annas-blog.org/duxiu-exclusive.html).

Images have been losslessly embedded. Information about the original file can be found in PDF attachments. Some stats (more in the PDF attachments):

{
  "filename": "MTIyOTI3Njguemlw",
  "filename_decoded": "12292768.zip",
  "filesize": 120733837,
  "md5": "d76f6d0343ceeb035cc7fe7f164157c4",
  "header_md5": "37f5ab4ddcd9f9c13daa7c16d7a8436d",
  "sha1": "0ed81b1d116a0f60180cd5e3ae3c736632bfe9c5",
  "sha256": "2c47750b32eb10f2ab9619d1f1bd412bbbde4e9bc980801437dc71045967b136",
  "crc32": 3083290717,
  "zip_password": "52gv",
  "uncompressed_size": 136323730,
  "pdg_dir_name": "12292768",
  "pdg_main_pages_found": 391,
  "pdg_main_pages_max": 391,
  "total_pages": 412,
  "total_pixels": 2654140167,
  "pdf_generation_missing_pages": false
}