嵌入式开发直通车

嵌入式 Linux 从入门到精通

陆桂来　梁　芳　张　波　编著

電子工業出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

本书的设计思路是让读者从了解基于 ARM 处理器的嵌入式系统的结构组成、硬件系统和软件操作系统入手，一步步地学习在嵌入式硬件系统中定制和移植 Linux 操作系统及在 Linux 操作系统下进行应用开发的过程。

本书共 12 章，分为 4 部分，分别是嵌入式系统基础、在 ARM 处理器系统上移植 Linux 操作系统、在 Linux 操作系统上进行软件开发及综合应用。

本书既有嵌入式系统硬件结构、ARM 处理器基础、操作系统基础等内容的介绍，也有一步步将 Linux 操作系统移植到 ARM 处理器上的过程，还有在嵌入式 Linux 上进行软件开发的过程，并且提供了大量应用实例，适合有一定计算机硬件基础、C 语言基础和 Linux 操作系统基础的工程师学习，亦适合高等院校计算机相关专业的学生和爱好者阅读，也可作为工程设计的参考手册。

图书在版编目（CIP）数据

嵌入式 Linux 从入门到精通 / 陆桂来，梁芳，张波编著. -- 北京：电子工业出版社，2015.4
（嵌入式开发直通车）
ISBN 978-7-121-25688-2

Ⅰ. ①嵌… Ⅱ. ①陆… ②梁… ③张… Ⅲ. ①Linux 操作系统－程序设计 Ⅳ. ①TP316.89

中国版本图书馆 CIP 数据核字（2015）第 048729 号

策划编辑：王敬栋（wangjd@phei.com.cn）
责任编辑：张　京
印　　刷：
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编：100036
开　　本：787×1092　1/16　印张：23.5　字数：601.6 千字
版　　次：2015 年 4 月第 1 版
印　　次：2015 年 4 月第 1 次印刷
印　　数：3 000 册　　定价：59.80 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前　言

行业背景

嵌入式系统是以应用为中心，以计算机技术为基础，采用可裁剪软/硬件，适用于对功能、可靠性、成本、体积、功耗等要求严格的专用计算机系统。随着 ARM 处理器的出现，嵌入式系统应用技术得到了长足的发展。

Linux 是在 UNIX 基础上发展起来的一套可以免费使用和自由传播的操作系统，从 1991 年问世到现在走过了 20 多年的历程，已从一个简单架构的系统内核发展到了现在结构完整、功能丰富的多版本用户系统。Linux 已经成为现今世界上最流行的操作系统之一；不仅能在 PC 和服务器上运行，而且随着嵌入式系统的发展，Linux 操作系统已成为 ARM 处理器最好的搭配。

嵌入式 Linux+ARM 已经广泛应用于机顶盒、智能手机、平板电脑、MPC（多媒体个人计算机）、网络设备、工业控制等领域，并且具有良好的市场前景。

关于本书

本书的设计思路是基于 S3C2440 处理器及 Ubuntu 操作系统，让读者从了解基于 ARM 处理器的嵌入式系统的结构组成、硬件系统和软件操作系统入手，一步步地学习在嵌入式硬件系统中定制和移植 Linux 操作系统及在 Linux 操作系统下进行应用开发的过程。

本书共 12 章，分为 4 部分，分别是嵌入式系统基础、在 ARM 处理器系统上移植 Linux 操作系统、在 Linux 操作系统上进行软件开发及综合应用。

- 第一部分：包括第 1～3 章，分别介绍了嵌入式系统的组成，Linux 操作系统的基础结构和命令，包括 ARM 在内的嵌入式处理器和常用的外围硬件结构，还介绍了一个基于 S3C2440 处理器的硬件开发板。
- 第二部分：包括第 4～6 章，用“step by step”的方法介绍在嵌入式系统硬件上移植 Linux 操作系统的过程，包括系统引导软件（Bootloader）、交叉编译环境的使用方法及文件系统的结构和移植方法等。
- 第三部分：包括第 7～11 章，介绍了在移植好 Linux 操作系统的嵌入式系统上使用 C 语言进行开发的方法，包括 C 语言开发环境介绍、文件和流操作方法、进程和线程操作方法、网络编程方法。
- 第四部分：包括第 12 章，介绍了 5 个嵌入式系统下的应用实例，包括守护进程的设计、串口双机通信等。

本书特色

- 基础内容丰富，涉及了嵌入式系统从软件到硬件各个方面的知识。
- 循序渐进，由浅入深，一步步地介绍了在嵌入式硬件系统上移植 Linux 操作系统的方法，并基于移植好的 Linux 操作系统介绍了使用 C 语言进行 Linux 编程开发的方法。
- 实例丰富，对于所介绍的相应知识，都基于 S3C2440 处理器的硬件系统和 Linux 操作系统给出了相当数量的实例。

作者介绍

本书由陆桂来、梁芳、张波编著，同时参与本书编写的还有严雨、刘艳伟、韩敏、徐慧超、刘洋洋、王闯、严安国、何世兰、汤嘉立、姚宗旭、葛祥磊、张玉梅等人。在此，对以上人员致以诚挚的谢意。由于时间仓促，程序较多，受学识水平所限，错误之处在所难免，恳请广大读者批评指正。

编　者

目　录

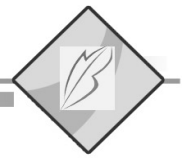
第一部分　嵌入式系统基础

第二部分 在 ARM 处理器系统上移植 Linux 操作系统

第三部分　在 Linux 操作系统上进行软件开发

第四部分　综 合 应 用

第一部分



嵌入式系统基础

第1章

嵌入式系统概述

嵌入式系统（Embedded System）是一种“完全嵌入受控器件内部，为特定应用而设计的专用计算机系统”，根据英国电气工程师协会（U. K. Institution of Electrical Engineer）的定义，嵌入式系统为控制、监视或辅助设备、机器或用于工厂运作的设备。与大型机、台式机、笔记本通用计算机系统不同，嵌入式系统通常执行的是带有特定要求的预先定义的任务。本章将对嵌入式系统进行一个基础介绍，涉及的内容如下：

- 嵌入式系统的发展；
- 嵌入式系统的构成；
- 嵌入式系统的处理器；
- 嵌入式系统的操作系统；
- 嵌入式系统的开发流程。

1.1 嵌入式系统的发展

嵌入式系统从 20 世纪 70 年代发展到如今已经迈过了 40 多个年头，随着电子和计算机技术的飞速发展，嵌入式系统也逐步得到了极大的成熟，总体来说其可以分为单片机时代、专用处理器时代和 ARM 时代三大阶段。

1.1.1 单片机时代（20 世纪 70 ~ 80 年代）

单片机时代起始于 1976 年，Intel 发布了世界上最早的单片机 8048；随后，Motorola 推出了 68HC05，Zilog 推出了 Z80 等一系列单片机。随着电子技术的发展，Intel 发布了著名的 MCS-51 单片机内核，ATMEL、NXP（前飞利浦）等公司在该内核的基础上生产了几百款不同的单片机产品，如图 1.1 所示是 Z80 和 51 系列单片机实物。

这些早期单片机系统的出现，使得汽车、家电、工业机器、通信装置及成千上万种产品可以通过内嵌电子装置来获得更佳的使用性能，而且更容易使用、处理速度更快、价格更便宜。正是由于电子装置是“内嵌式的”，因此也就使得“嵌入式系统”这个初级概念深入人心。今天看来，当时这些装置已经初步具备了嵌入式的应用特点，但是这时的应用

只是使用 8 位的芯片，硬件技术相对落后，如只能执行一些单线程的程序，还谈不上“多核”的概念。但是它标志着“嵌入式系统”出现了硬件雏形。在开创嵌入式系统独立发展的道路上。

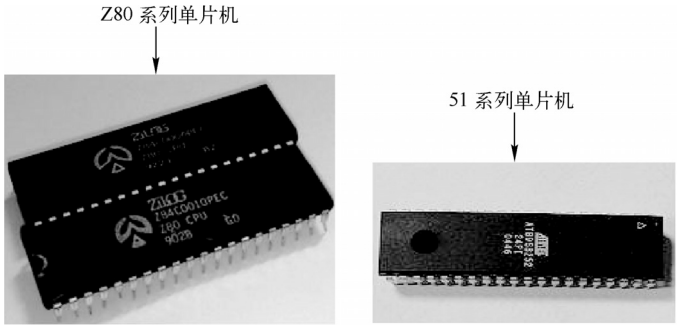


图 1.1　Z80 和 51 系列单片机实物

从 20 世纪 80 年代早期开始，嵌入式系统的程序员开始用商业级的“操作系统”编写嵌入式应用软件，这使得可以获得更短的开发周期、更少的开发资金和更高的开发效率，“嵌入式系统”真正出现了。确切地说，这个时候的操作系统是一个实时核，这个实时核包含了许多传统操作系统的特征，包括任务管理、任务间通信、同步与相互排斥、中断支持、内存管理等功能。其中比较著名的有 Integrated System Incorporation（ISI）的 PSOS、Ready System 公司的 VRTX、IMG 的 VxWorks 和 QNX 公司的 QNX 等。这些嵌入式操作系统都具有嵌入式的典型特征。

- 它们的系统内核很小，具有可裁剪、可扩充和可移植性，可以移植到各种各样的处理器芯片上。
- 它们均采用占先式的调度，响应时间很短，任务执行的时间可以确定。
- 较强的实时性和可靠性，适合嵌入式应用。
- 这些嵌入式实时多任务操作系统的出现，使得应用开发人员得以从小范围的开发中解放出来，同时也促使嵌入式有了更为广阔的应用空间。

但是从整体来说单片机时代的软件都具有“无操作系统”直接运行在处理器上，根据实际用途编写，相对简单，及硬件耦合性极大而不便于移植的特点。

1.1.2　专用处理器时代（20 世纪 90 年代 ~ 21 世纪）

进入 20 世纪 90 年代以后，随着计算机技术、微电子技术、IC 设计和 EDA 工具的发展，嵌入式处理器开始向片上系统（System-on-Chip，SoC）发展，出现了包括 51 单片机、AVR 单片机、MSP430 单片机、DSP、CPLD/FPGA 在内的一系列处理器，如图 1.2 所示，而 ARM 处理器也在此时初露头角。

此时出现了众多嵌入式操作系统，它们大多具有跨平台的移植技术，并且在同一个系统之下也可以通过选择开发工具来使用 Java、C 或汇编语言等开发者熟悉的语言来开发，该阶段比较常用的有 WinCE、Palm、WM、Linux、VxWorks、μC/OS-II，Symbian 等。

图 1.2　专用处理器时代的嵌入式处理器

1.1.3　ARM 时代（21 世纪至今）

进入 21 世纪之后，随着相关电子工业技术的发展，嵌入式处理器相关技术得到了突飞猛进的发展，出现了 64 位嵌入式处理器（如 Cortex-A50 系列），其处理器内核也已经实现了 8 核（目前正计划实现 16 核）。

到目前为止，嵌入式处理器可以分为三个大类：以 MTK、高通、三星为代表支持的 ARM 架构处理器、以 Intel 为代表支持的 x86 架构处理器及其他以 FPGA 为代表的特殊/专用处理器，如图 1.3 所示。

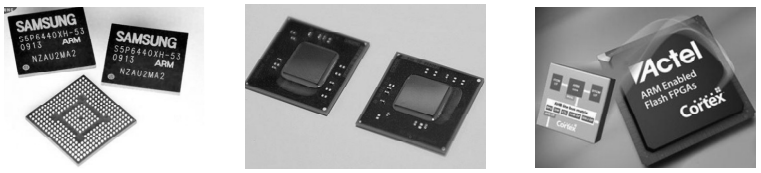


图 1.3　进入 21 世纪之后的嵌入式处理器

随着嵌入式处理器的发展，嵌入式系统的硬件性能得到了极大的提升，此时嵌入式操作系统也开始出现一些新的面孔，Android 和 IOS 则是其中的典型代表，它们从 2007 年出现开始（Android 于 2007 年 11 月正式发布，IOS 则在 2007 年 1 月正式发布）就风卷残云般地占领了绝大多数嵌入式消费电子产品（主要是平板电脑、手机和数字播放器）的市场；而微软公司（Microsoft Corporation）不甘落后，从 2010 年开始连续发布了 WP（Windows Phone）和 Windows RT（RunTime）操作系统，用于抢占消费电子产品市场。而在工业控制等领域上，嵌入式操作系统本着稳定可靠的原则，则依然是 winCE、VxWorks 和 Linux 当道。

1.2　嵌入式系统的构成

1.2.1　嵌入式系统的层次模型

嵌入式系统的层次模型如图 1.4 所示，由包括嵌入式处理器在内的硬件系统层、中间驱

动层、操作系统层和应用软件层组成。

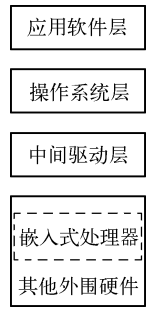


图 1.4　嵌入式系统的结构

1．硬件系统层

硬件系统层主要包含了嵌入式系统中必要的硬件设备：嵌入式处理器和协处理器、存储器、其他外围 I/O 设备。

- 处理器：是嵌入式系统硬件层的核心，主要负责对信息的运算处理，相当于通用计算机的中央处理器；协处理器则是协助处理器完成相应工作的部件辅助处理器。
- 存储器：其用来存储数据和代码，嵌入式系统的存储器一般包括嵌入式处理器内部和外部存储器，此外还有大容量存储器。
- 其他外围 I/O 设备：其用于提供嵌入式系统和其他系统的数据接口及一些特定的工作，如 RS-232 接口、CAN 总线接口、A/D 和 D/A 模块、电动机和继电器驱动等执行机构等，此外包括输入设备、显示设备等在内的人机交互部件也包含在内。

2．中间驱动层

中间驱动层为硬件层与系统软件层之间的部分，有时也称为硬件抽象层（Hardware Abstract Layer，HAL）或板级支持包（Board Support Package，BSP）。对于上层的操作系统，中间驱动层提供了操作和控制硬件的方法和规则。而对于底层硬件，中间驱动层主要负责相关硬件设备的驱动等。

中间驱动层将系统上层软件与底层硬件分离开来，使系统的底层驱动程序与硬件无关，上层软件开发人员无须关心底层硬件的具体情况，根据中间驱动层提供的接口即可进行开发。

中间驱动层主要包含以下几个功能。

- 底层硬件初始化操作按照自底而上、从硬件到软件的次序分为三个环节，依次是：片级初始化、板级初始化和系统级初始化。
- 硬件设备配置对相关系统的硬件参数进行合理的控制以正常工作。另一个主要功能是硬件相关的设备驱动。
- 硬件相关的设备驱动程序的初始化通常是一个从高到低的过程。尽管中间层中包含硬件相关的设备驱动程序，但是这些设备驱动程序通常不直接由中间层使用，而是在系统初始化过程中由中间层将它们与操作系统中通用的设备驱动程序关联起来，并在随后的应用中由通用的设备驱动程序调用，实现对硬件设备的操作。

3．操作系统层

操作系统层由实时多任务操作系统（Real-time Operation System，RTOS）及其实现辅助功能的文件系统、图形用户界面接口（Graphic User Interface，GUI）、网络系统及通用组件模块组成，其中实时多任务操作系统（RTOS）是整个嵌入式系统开发的软件基础和平台。

4．应用软件层

应用软件层是开发设计人员在系统软件层的基础之上，根据需要实现的功能，结合系统的硬件环境所开发的软件。

1.2.2 嵌入式系统的处理器

嵌入式系统的核心就是各种类型的嵌入式处理器。目前几乎每个半导体制造商都生产嵌入式处理器，越来越多的公司拥有自己的处理器设计部门。嵌入式微处理器的体系结构经历了从 CISC 到 RISC 和 Compact RISC 的转变；位数由 4 位、8 位、16 位、32 位到 64 位；寻址空间一般为 64KB～16MB，处理速度为 0.1MIPS～2000MIPS；常用的封装为 8～144 个引脚。根据其现状，嵌入式处理器可以分为嵌入式微控制器（Embedded Microcontroller Unit，EMCU）、嵌入式微处理器（Embedded Microprocessor Unit，EMPU）、嵌入式数字信号处理器（Embedded Digital Signal Processor，EDSP）和嵌入式片上系统（Embedded System on Chip，ESOC）四类。

1．嵌入式微控制器（EMCU）

嵌入式微控制器以各种单片机为代表，也就是在一块芯片中集成了整个计算机系统。嵌入式微控制器一般以某种微处理器内核作为核心，芯片内部集成 ROM/EPROM、E^2PROM、Flash、RAM、总线、总线逻辑、定时/计数器、WatchDog、I/O 口、脉宽调制输出、A/D 和 D/A 等各种必要功能和外设。微控制器比微处理器体积小，功耗和成本低，可靠性高，因而是目前嵌入式工业的主流，品种和数量都很多。

2．嵌入式微处理器（EMPU）

嵌入式微处理器相对嵌入式微控制器而言因为其内置的外围接口较少所以控制能力稍弱，但是计算能力得到了极大提高，并且相当多型号还内置了内存管理单元（MMU）以方便运行操作系统。它一般装配在专门设计的电路板上，只保留与嵌入式应用有关的母板功能，但是电路板上必须包括 ROM、RAM、总线接口、各种外设等器件，目前常见的嵌入式处理器有 ARM、MIPS、Power PC、Intel ATOM 等。

- ARM：ARM（Advanced RISC Machines）公司是全球领先的 16/32 位 RISC（精简指令集计算机）微处理器知识产权设计供应商。ARM 公司通过转让高性能、低成本、低功耗的 RISC 微处理器、外围和系统芯片设计技术给合作伙伴，使他们能用这些技术来生产各具特色的芯片。ARM 已成为移动通信、手持设备和多媒体数字设备嵌入式解决方案的 RISC 标准。ARM 处理器有三大特点：体积小、低功耗、低成本和高性能，16/32 位双指令集，全球的合作伙伴众多。
- MIPS：MIPS（Microprocessor without Interlocked Pipeline Stages）是一种处理器内核标准，它是由 MIPS 技术公司开发的。MIPS 技术公司是一家设计制造高性能、高档次的嵌入式 32 位和 64 位处理器的厂商，在 RISC 处理器方面占有重要地位。2000 年，MIPS 公司发布了针对 MIPS 32 4Kc 处理器的新版本及未来 64 位 MIPS 64 20Kc 处理器内核。MIPS 技术公司既开发 MIPS 处理器结构，又自己生产基于 MIPS 的 32 位/64 位芯片。为了使用户更加方便地应用 MIPS 处理器，MIPS 公司推出了一套集成的开发工具，称为 MIPS IDF（Integrated Development Framework），特别适用于嵌入式系统的开发。
- Power PC：Power PC 架构的特点是可伸缩性好，方便灵活。Power PC 处理器品种很多，既有通用的处理器，又有嵌入式控制器和内核，应用范围从高端的工作站、服

务器到桌面计算机系统，从消费类电子产品到大型通信设备等各个方面，非常广泛。目前 Power PC 独立微处理器与嵌入式微处理器的主频从 25MHz～700MHz 不等，它们的能量消耗、大小、整合程度和价格差异悬殊，主要产品模块有主频 350～700MHz 的 Power PC 750CX 和 750CXe 及主频 400MHz 的 Power PC 440GP 等。嵌入式的 Power PC 405（主频最高为 266MHz）和 Power PC 440（主频最高为 550MHz）处理器内核可以用在各种集成的系统芯片（SoC）设备上，在电信、金融和其他许多行业具有广泛的应用。

- Intel ATOM：Intel 公司出品的 ATOM（凌动处理器）是英特尔历史上体积最小和功耗最小的处理器，其基于新的微处理架构，专门为小型和嵌入式系统所设计，和其他嵌入式处理器相比其最大的优势是采用了 X86 体系结构，可以运行 Windows 操作系统（也可以运行 Android 操作系统），能提供更好的通用性，所以在平板电脑等消费电子产品中得到了广泛的应用。

3．嵌入式数字信号处理器（EDSP）

数字信号处理器对系统结构和指令进行了特殊设计，使其适合于执行 DSP 算法，编译效率较高，指令执行速度也快。DSP 应用正从在通用单片机中以普通指令实现 DSP 功能，发展到采用嵌入式数字信号处理器。嵌入式数字信号处理器的长处在于能够进行向量运算、指针线性寻址等运算量较大的数据处理。比较有代表性的产品是 Motorola 的 DSP56000 系列、Texas Instruments 的 TMS320 系列，以及 Philips 公司基于可重置嵌入式 DSP 结构制造的低成本、低功耗的 REAL DSP 处理器。

4．嵌入式片上系统（ESOC）

片上系统 SOC 则是在一个硅片上实现一个更为复杂的系统。各种处理器内核将作为 SOC 设计公司的标准库，成为 VLSI 设计中的一种标准器件，用标准的 VHDL 语言描述，存储在器件库中。SOC 可以分为通用 SOC 和专用 SOC 两类。通用系列包括 Infineon（Siemens）的 TriCore、Motorola 的 M-Core、某些 ARM 系列器件等。而专用的 SOC 专用于某个或某类系统中，不为一般用户所知。例如 Philips 的 Smart XA，它将 XA 单片机内核和支持超过 2048 位复杂 RSA 算法的 CCU 单元制作在一块硅片上，形成一个可以加载 Java 或 C 语言的专用的片上系统。

1.2.3　嵌入式系统的操作系统

单片机时代的嵌入式系统都不用操作系统，它们只是为了实现某些特定功能，使用一个简单的循环控制对外界的控制请求进行处理，不具备现代操作系统的基本特征，这对一些简单的系统而言是足够的。但是当系统越来越复杂、利用的范围越来越大时，缺少操作系统就成为了最大的一个缺点，因为每添加一项新功能都可能需要重新开始设计，否则只能增加开发成本和系统复杂度。

从 20 世纪 80 年代开始，出现了各种各样的商业用嵌入式操作系统，如 VxWorks、Linux 和 Android 等。

1．VxWorks

VxWorks 操作系统是美国风河（WindRiver）公司于 1983 年设计开发的一种实时操作系

统，以其良好的可靠性和卓越的实时性被广泛应用在通信、军事、航空、航天等高精尖技术领域及实时性要求极高的领域中，如卫星通信、军事演习、导弹制导、飞机导航等，其最大的缺点是价格高昂，主要特点说明如下。

高性能实时：VxWorks 的微内核 Wind 是一个具有较高性能的、标准的嵌入式实时操作系统内核，其支持抢占式的基于优先级的任务调度，支持任务间同步和通信，还支持中断处理、看门狗（WatchDog）定时器和内存管理。其任务切换时间短、中断延迟小、网络流量大的特点使得 VxWorks 的性能得到很大的提高，与其他嵌入式系统相比具有很大的优势。

POSIX 兼容：POSIX（the Portable Operating System Interface）是工作在 ISO/IEEE 标准下的一系列有关操作系统的软件标准。制定这个标准的目的就是在源代码层次上支持应用程序的可移植性。这个标准产生了一系列适用于实时操作系统服务的标准集合 1003.1b（过去是 1003.4）。

可配置性好：VxWorks 提供良好的可配置能力，可配置的组件超过 80 个，用户可以根据自己系统的功能需求通过交叉开发环境方便地进行配置。

友好的开发调试环境：VxWorks 提供的开发调试环境便于进行操作和配置，开发系统 Tornado 更是受到了广大嵌入式系统开发人员的欢迎。

支持的处理器种类多，如 x86、i960、Sun Sparc、Motorola MC68000、MIPS RX000、Power PC、StrongARM、XScale 等。大多数 VxWorks API 是专用的，VxWorks 提供的板级支持包（Board Support Package，BSP）支持多种硬件板，包括硬件初始化、中断设置、定时器和内存映射等例程。

2．Linux

嵌入式 Linux 现在已经有许多版本，包括强实时的嵌入式 Linux（如新墨西哥工学院的 RT-Linux 和堪萨斯大学的 KURT-Linux 等）和一般的嵌入式 Linux 版本（如 uCLinux 和 PocketLinux 等）。其中，RT-Linux 把通常的 Linux 任务优先级设为最低，而所有的实时任务的优先级都高于它，以达到既兼容通常的 Linux 任务又保证强实时性能的目的。另一种常用的嵌入式 Linux 是 uCLinux，它是针对没有 MMU（内存管理单元）的处理器而设计的。它不能使用处理器的虚拟内存管理技术，对内存的访问是直接的，所有程序中访问的地址都是实际的物理地址。它专门为嵌入式系统做了许多小型化的工作。

3．μC/OS-II

μC/OS 是“MicroController Operating System”的缩写，它是源码公开的实时嵌入式操作系统，其主要特点说明如下。

- 源代码公开，系统透明，很容易把操作系统移植到各个不同的硬件平台上。
- 可移植性强，μC/OS-II 绝大部分源码是用 ANSI C 写的，可移植性（Portable）较强。而与微处理器硬件相关的那部分是用汇编语言写的，已经压缩到最低限度，使 μC/OS-II 便于移植到其他微处理器上。
- 可固化，μC/OS-II 是为嵌入式应用而设计的，这就意味着，只要开发者有固化（ROMable）手段（C 编译、连接、下载和固化），μC/OS-II 即可嵌入到开发者的产品中成为产品的一部分。
- 可裁剪，通过条件编译可以只使用 μC/OS-II 中应用程序需要的那些系统服务程序，

以减少产品中的 μC/OS-II 所需的存储器空间（RAM 和 ROM）。

- 占先式，μC/OS-II 完全是占先式（Preemptive）的实时内核，这意味着 μC/OS-II 总是运行就绪条件下优先级最高的任务。大多数商业内核也是占先式的，μC/OS-II 在性能上和它们类似。
- 实时多任务，μC/OS-II 不支持时间片轮转调度法（Round-roblin Scheduling），该调度法适用于调度优先级平等的任务。
- 可确定性，全部 μC/OS-II 的函数调用与服务的执行时间具有可确定性。

由于 μC/OS-II 仅是一个实时内核，这就意味着它不像其他实时操作系统那样，它提供给用户的只是一些 API 函数接口，有很多工作往往需要用户自己去完成。把 μC/OS-II 移植到目标硬件平台上也只是系统设计工作的开始，后面还需要针对实际的应用需求对 μC/OS-II 进行功能扩展，包括底层的硬件驱动、文件系统和用户图形接口（GUI）等，从而建立一个实用的 RTOS。

4．Android（安卓）和 IOS

Android（安卓）是一种基于 Linux 的自由及开放源代码的操作系统，主要使用于移动设备，如智能手机和平板电脑，由 Google 公司和开放手机联盟领导及开发。该操作系统于 2007 年 11 月正式发布，2008 年 10 月第一部采用其技术的智能手机（G1）上市，截止到 2013 年年末，Android 的版本号已经发展到了 4.4，采用该操作系统的智能设备数量已经超过了 10 亿台。

Android 可以在 ARM 和 x86 体系结构的处理器上运行，其采用了分层的架构，可以从高层到低层分为应用程序层、应用程序框架层、系统运行库层和 Linux 内核层。

Android 具有开放性、不受束缚、硬件丰富、开发方便等优势，当然，其最大的优势还是 Google 公司和众多硬件服务商的支持，其缺点是采用了虚拟机制，导致效率略低，且碎片化严重。

iOS 是苹果公司于 2007 年发布的系统，是以 Darwin（一种类 UNIX 操作系统）为基础的操作系统，最开始命名为 iPhone OS，后来更名为 iOS，是 iPod touch、iPad 及 Apple TV 等产品的操作系统，其版本号目前已经更新到了 iOS 7.0.4。

iOS 拥有在嵌入式操作系统中最多的应用程序及最好的 App（application）库，支持 ARM 体系架构的处理器，具有高安全、支持多语言、流畅、美观等特点，其缺点是封闭性较强。

5．WP 和 Windows RT

WP 是 Windows Phone 的简称，是微软在 2010 年发布的一款智能手机操作系统，将微软旗下的 Skype、必应、Xbox Live 游戏、Xbox Music 音乐与独特的视频体验整合至手机；到 2013 年年底其已经更新到了 8.1 版本，其支持 ARM 体系架构的处理器。

Windows RT（RunTime）则是微软为实时嵌入式系统发布的 Windows 版本，其此采用了 Metro 风格的用户界面，支持 ARM 体系架构处理器，但是其无法兼容普通 x86 处理器结构上 Windows 的软件。“RT”代表“Runtime”，也就是 Windows Runtime Library。它是一项非常重要的技术，因为它允许开发人员写一个 App，但是可以同时在利用英特尔处理器的 Windows 8 上运行，还可以在利用 ARM 处理器的 Windows RT 上运行。

1.3 嵌入式系统和通用计算机系统的简单比较

通用计算机是类似个人台机、笔记本等具有普通计算机基本形态，通过安装不同的应用软件，以基本类同的“外形面貌”在社会的各行业、各种工作环境中都能使用的计算机，其和嵌入式系统的比较如表 1.1～表 1.3 所示。

表 1.1 嵌入式系统和通用计算机系统的比较

特　　点	嵌入式系统	通用计算机
组成	采用 51 单片机、ARM 等集成了部分外部设备和总线的嵌入式处理器，硬件和软件耦合性较强	采用 Intel 和 AMD 的标准处理器，采用标准通用总线和外部设备，硬件和软件相对独立
外形特征	多“嵌入”到应用系统内部，用户不能直接观察到	用户可以直接观察和使用
开发方式	采用交叉开发方式，在通用计算机上开发，在嵌入式系统上运行	开发和运行都在通用计算机上进行
二次开发性	较高	较差

表 1.2 嵌入式系统和通用计算机系统的硬件比较

部　　件	嵌入式系统	通用计算机
处理器	ARM、单片机等	Intel 和 AMD 的通用处理器
内存	多使用 SDRAM 芯片	已经发展到了 DDR 芯片
存储设备	FLASH	硬盘等
输入设备	按键、触摸屏等定制设备	鼠标、键盘等通用设备
显示设备	LED、数码管、定制液晶屏等	显示器
发声器件	音频芯片、蜂鸣器等	声卡
接口	RS232、RS485、CAN 总线、USB 等	串口、USB 口等
其他	特定的驱动器件如电机驱动芯片等	外部扩展卡如 HDMI 等

表 1.3 嵌入式系统和通用计算机系统的软件比较

特　　点	嵌入式系统	通用计算机
引导代码	多为 Bootloader 如 U-boot	主板 BIOS 和硬盘引导区结合
操作系统	WinCE、Linux、Vxworks、Android 等	Windows、Linux 等
驱动程序	根据硬件和操作系统自行裁剪	操作系统或者厂商提供通用的
协议栈	根据需求自行定义	操作系统或者第三方提供
开发环境	交叉编译环境	本机调试
仿真环境	需要 JTAG 仿真器等	直接本机调试

综上所述，嵌入式系统和通用计算机系统相比较的特点如下。

- 专用性强：嵌入式系统通常面向特定应用，将任务集成在嵌入式处理器内部完成，体积通常都较小。
- 技术融合性高：嵌入式系统通常将计算机、通信和电子、自动化控制等多个方面的

技术融合到一起。

- 软/硬件耦合性高：嵌入式系统以硬件为基础，以软件为核心，通常都会去除冗余，量体裁衣，同时带来较高的耦合性。
- 资源较少：面向特定应用的嵌入式系统考虑到经济型等因素，通常会涉及较少的硬件资源，从而带来更低的成本和更简单的结构。
- 需要专门的开发工具和环境。
- 技术较为先进、性价比高，对系统配置要求低，实时性强。

1.4　嵌入式系统的开发流程

嵌入式系统开发是一个系统性的工程，一个完整的开发过程总体来说包括了硬件系统设计、操作系统移植和应用软件设计三个部分。

1.4.1　硬件系统设计

硬件设计包括了硬件体系架构、基于 Protel 或其他电路板设计软件的电路图设计、电路板厂商的电路板制作、焊接和测试、电路板调试等步骤，其根本是得到一个在电气连接上没有错误、满足设计需求的硬件电路板。

1.4.2　操作系统移植

操作系统的移植是指当嵌入式硬件开发已经完成且保证没有硬件错误之后将一个目标操作系统移植到硬件系统上并且运行的过程，其目标是在硬件系统上运行一个操作系统。

以 Linux 为例来介绍操作系统的移植过程，大概可以分为以下 4 个步骤。

（1）配置和编译 Bootloader，然后将 Bootloader 下载到开发板，其可以初始化硬件设备，建立内存空间的映射表，对操作系统进行引导。

（2）下载操作系统的源代码，建立交叉编译环境，配置和编译操作系统内核，并且根据硬件系统的特点对其进行相应裁剪和配置，然后将通过 Bootloader 将完成的操作系统下载到目标板上。

（3）为 NAND FLASH 移植文件系统，通常来说是 YAFFS2 文件系统，这样才能形成完整的操作系统应用环境。

（4）建立嵌入式系统和开发环境的数据交互通道，可以是 FTP，也可以是根文件映射等。

1.4.3　应用软件设计

应用软件设计是指在已经移植完成操作系统的嵌入式系统上根据系统的特定需求进行软件设计的过程，此时通常也需要交叉编译环境。

注意：本书主要介绍第二步操作系统移植和第三步应用软件设计。

1.5 嵌入式系统的应用

随着电子技术的发展，嵌入式系统已经从最开始偏重工业控制应用逐步向包括消费电子产品在内的日常生活用品普及，可以说嵌入式系统的应用已经深入了社会生活的方方面面，如图 1.5 所示是嵌入式系统的常见应用。

图 1.5 嵌入式系统的常见应用

常见的嵌入式系统应用列举如下。

- 银行业经常使用的自动柜员机（ATM）。
- 用于工业自动化和监测的可编程逻辑控制器（PLCs）。
- 航空电子，如惯性导航系统、飞行控制硬件和软件及其他飞机和导弹中的集成系统。
- 计算机网络设备，包括路由器、时间服务器和防火墙。
- 办公设备，包括打印机、复印机、传真机、多功能打印机（MFPs）。
- 磁盘驱动器（软盘驱动器和硬盘驱动器）。
- 汽车发动机控制器和防锁死刹车系统。
- 家庭自动化产品，如恒温器、冷气机、洒水装置和安全监视系统。
- 家用电器，包括微波炉、洗衣机、电视机、DVD 播放器和录制器。
- 医疗设备，如 X 光机、核磁共振成像仪。
- 测试设备，如数字存储示波器、逻辑分析仪、频谱分析仪。
- 智能手表和包括智能手环在内的各种可穿戴设备。
- 多媒体电器，互联网无线接收机、电视机顶盒、数字卫星接收器、网络播放器、智能电视等。
- 影像记录设备，如卡片相机、DV、单反。
- 平板电脑，如 iPad、kindle fire。
- 智能手机，如 HTC one、iphone。
- 固定游戏机和便携式游戏机，如 XBOX。

第2章

嵌入式系统的硬件

嵌入式系统的硬件架构是嵌入式系统硬件设计的基础，一个完整的嵌入式系统的硬件结构如图 2.1 所示，包括了嵌入式处理器、存储器和其他外围设备。本章将重点介绍广泛应用的处理器 ARM 及嵌入式系统中的存储器件，然后介绍本书基于的 S3C2440 嵌入式开发板 GT2440 的硬件特点和资源，涉及的内容如下：

- 嵌入式系统的硬件架构；
- ARM 处理器和 ARM 公司的发展历程；
- ARM 处理器的分类和特点；
- SDRAM、FLASH 等嵌入式系统常用存储芯片介绍；
- GT2440 嵌入式开发板的硬件特点和资源。

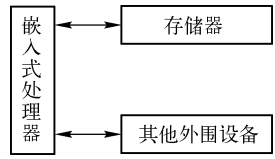


图 2.1　嵌入式系统的硬件结构

2.1　嵌入式系统的 ARM 处理器

嵌入式系统中的处理器有多种类型，小到 51 单片机，大到 i7 处理器和某些基于可编程逻辑器件的专用芯片，都可以作为核心处理器，但是在实际应用尤其是消费数码产品和工业控制系统中，目前 ARM 一枝独秀，占据了半壁江山，其优点是功耗低，缺点是计算能力较为低下且不能支持普通的 Windows 系统（Window 7、8 非 RT 系统）。ARM（Advanced RISC Machines）是 ARM（安谋国际科技）公司设计的处理器核心，是目前广为使用的嵌入式系统处理器。

2.1.1　ARM 处理器的发展历程

ARM 公司（安谋国际科技）是业界领先的微处理器技术提供商，其提供最广泛的微处理器内核，可满足几乎所有应用市场对性能、功耗及成本的要求。再加上一个富有活力的生态系统（拥有 1000 多家可提供芯片、开发工具和软件的合作伙伴），ARM 已售出超过 300 亿个处理器，每天的销量超过 1600 万，是真正意义上的“The Architecture for the Digital World”（面向数字世界的体系结构）。

安谋国际科技公司前身为艾康电脑，于 1978 年在英国剑桥（Cambridge）创立，在 20

世纪 80 年代晚期，苹果电脑开始与艾康电脑合作开发新版的 ARM 核心。1985 年，艾康电脑研发出采用精简指令集的新处理器，名为 ARM（Acorn RISC Machine），又称 ARM 1。因为艾康电脑的财务出现状况，1990 年 11 月 27 日，获得 Apple 与 VLSI 科技的资助，分割出安谋国际，成为独立子公司，其运作模式主要是涉及 IP 的设计和许可，并不进行生产和销售实际的半导体芯片，也就是说，其只提供一些对应的 ARM 核心架构，而由不同的半导体公司根据这些核心架构再加上其他外围部件来形成具体型号的芯片。ARM 公司的特点如下：

- 全球领先的半导体 IP 公司；
- 成立于 1990 年；
- 目前为止已销售超过 200 亿个基于 ARM 的芯片；
- 向 250 多家公司出售了 800 个处理器许可证；
- 获得了所有基于 ARM 的芯片的版税；
- 赢得了长期成长型市场的市场份额；
- ARM 的收益增速通常要比整个半导体行业快。

除了处理器内核之外，ARM 公司还提供了一系列用于优化片上系统设计的工具、物理和系统 IP，如图 2.2 所示。

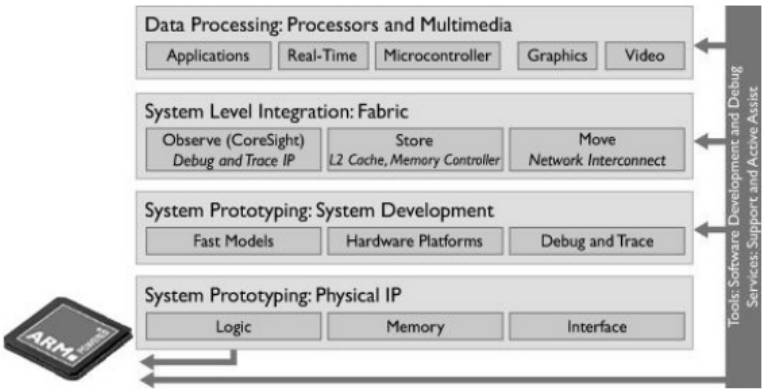


图 2.2　ARM 公司提供的其他产品

安谋国际科技的发展里程碑如下，可以看到几乎每一年都有创新性产品推出。

- 1985 年：Acorn Computer Group 开发出全球第一款商业 RISC（Reduced Instruction Set Computer，精简指令集）处理器。
- 1987 年：Acorn 的 ARM 处理器作为低成本 PC 的第一款 RISC 处理器亮相。
- 1990 年：Advanced RISC Machines（ARM)）无须 Acorn 和 Apple Computer 提供合作，即可独立制定新的微处理器标准章程，VLSI Technology 成为投资商和第一个授权使用方。
- 1991 年：ARM 推出第一款 RISC 核心，即 ARM6 解决方案。
- 1993 年：推出 ARM7 解决方案。
- 1995 年：发布 Thumb 架构扩展，以在 16 位系统成本的基础上提供 32 位的 RISC 性

能，并且提供业界领先的代码；StrongARM 核心发布。

- 1996 年：推出 ARM810 微处理器；Windows CE 被扩展到 ARM 架构上。
- 1997 年：发布 ARM9TDMI 系列处理器；JavaOS 被扩展到 ARM 架构上。
- 1998 年：发布 ARM7TDMI 核心。
- 1999 年：发布提高了信号处理能力的 ARM9E 核心。
- 2000 年：发布 SecurCore 智能卡系列核心。
- 2001 年：发布 ARMv6 架构。
- 2002 年：发布 ARM11 微架构；发布 RealView 开发工具系列。
- 2003 年：发布针对多核心的 CoreSight 实时调试和跟踪解决方案；和 Nokia、STM、TI 成立 MIPI 联盟，为移动应用处理器指定开放性标准；发布为 ARM 核心提供了安全平台的 TrustZone 技术。
- 2004 年：发布了基于 ARMv7 架构的 ARM Cortex 系列处理器，并且发布了首款产品 Cortex-M3；发布了第一款集成多处理器 MPCore；发布了具有开创性的嵌入式信号处理核心 OptimeoDe。
- 2005 年：收购了 Keil Software 公司；发布了 Cortex-A8 处理器。
- 2007 年：推出了针对智能卡应用的 SecurCore SC300 处理器；推出了实现可扩展性和低功耗设计的 Cortex-A9 处理器。
- 2008 年：发布了全球第一个在 1080 HDTV 分辨率下符合 Khronos Open GL ES 2.0 标准的 Mali-200 GPU；同年，ARM 处理器销售量已经达到了 100 亿台。
- 2009 年：推出体积最小、功耗最低和能效最高的处理器 Cortex-M0；宣布实现具有 2GHz 频率的 Coretx-A9 双核处理器。
- 2010 年：推出符合 AMBA 4 协议的系统 IP Corelink 400 系列；推出 ARM Mali-T604 图形处理单元，同时 ARM Mali 成为被最广泛授权的嵌入式 GPU 价格；推出 Cortex-A15 MPCore 处理器；微软公司（Microsoft）成为 ARM 架构授权使用方；推出 Cortex-M4 处理器。
- 2011 年：和 Cadence、TSMC 合力推出第一款 20nm Cortex-A15 多核处理器；发布了嵌入式软件库 ESS；发布了 ARM Mali-T658 GPU；推出了 ARMv8 架构；发布了 Cortex-A7 处理器；微软公司提出了基于 ARM 的 Windows 产品 Windows RT。
- 2012 年：第一代 Windows RT 产品问世。

2.1.2　ARM 处理器的架构、类型和型号及一些专用术语

ARM 处理器的分类是一个比较复杂的过程，需要区分不同的架构、类型（系列）和具体的芯片型号，其中架构和类型（系列）是 ARM 公司提供的，而芯片型号则是由生产厂商所提供的。

ARM 公司提供了 ARMv1、ARMv2、ARMv3、ARMv4、ARMv5、ARMv6、ARMv7 和 ARMv8 共八种不同的架构，其中 ARMv1 和 ARMv2 都没有太大的实际使用价值，从 ARMv3 开始才逐步开始正式商用。

从 ARMv3 架构开始，ARM 推出了对应的 ARM6、7 处理器类型（系列），目前常见的

ARM 处理器类型（系列）有 ARM7、ARM9、ARM10、ARM11 和 Cortex。而每个系列处理器中又有许多不同的类型，如 ARM9 系列就有 ARM9E-S、ARM966E-S 等类型。

表 2.1 给出了 ARM 体系架构和具体的产品类型（系列）对应关系列表，其中后缀-E 表明支持增强型 DSP 指令集、-J 表明支持新的 Java。

表 2.1　ARM 体系架构和对应的类型（系列）

体 系 架 构	具体处理器类型（系列）
ARMv1	ARM1
ARMv2	ARM2、ARM3
ARMv3	ARM6、ARM7
ARMv4	SrongARM、ARM7TDMI、ARM9TDMI、ARM940T、ARM920T、ARM720T
ARMv5	ARM9E-S 、ARM966E-S 、ARM1020E 、ARM 1022E 、XScale 、ARM9EJ-S 、ARM926EJ-S 、ARM7EJ-S、ARM1026EJ-S、ARM10
ARMv6	ARM11 系列（ARM1136J(F)-S、ARM1156T2(F)-S、ARM1176JZ(F)-S 和 ARM11 MPCore）、ARM Cortex-M
ARMv7	ARM Cortex-A、ARM Cortex-M、ARM Cortex-R
ARMv8	Cortex-A50

具体的产品型号则是得到授权的生产厂商根据 ARM 公司提供的具体处理器类型生产的芯片，如高通公司（Qualcomm）的处理器 MSM7201A 即是 ARM11 系列处理器，其对应的具体 ARM 类型是 ARM1136J(F)-S，使用的是 ARMv6 架构，这块处理器应用在 HTC 公司的 Dream、Magic 等手机上。

目前在消费数码产品中应用得最为广泛的 ARM 处理器是 Cortex-A 系列，采用了 ARMv7 架构，包括了 Cortex-A8 和 Cortex-A9 等子系列。

以下是一些 ARM 处理器所涉及的专用技术和术语。

- ARM 32-bit ISA：基于 RISC 原理的 32 位 ARM 指令集。
- Thumb 16-Bit ISA：Thumb 技术是对 32 位 ARM 体系结构的扩展，Thumb 指令集是已压缩至 16 位宽操作码的、最常用 32 位 ARM 指令的子集。在执行时，这些 16 位指令实时、透明地解压缩为完整 32 位 ARM 指令，且无性能损失。卓越的代码密度，以尽量减小系统内存大小和降低成本。
- Thumb-2：以 ARM Cortex 体系结构为基础的指令集，其提升了众多嵌入式应用的性能、能效和代码密度，可以提供最佳代码大小和性能。其是以获得成功的 Thumb（ARM 微处理器内核的创新型高代码密度指令集）为基础进行构建，用于增强 ARM 微处理器内核的功能，从而可以使开发人员能够开发出低成本且高性能的系统。
- VFP：为浮点体系结构（Vector Floating Point，VFP）为半精度、单精度和双精度浮点运算中的浮点操作提供硬件支持。可以为汽车动力系统、车身控制应用和图像应用（如打印中的缩放、转换和字体生成及图形中的 3D 转换、FFT 和过滤）中使用的浮点运算提供增强的性能。
- Jazelle 技术：可以用于提高执行环境（如 Java、.Net、MSIL、Python 和 Perl）速度。Jazelle 技术是 ARM 提供的组合型硬件和软件解决方案。ARM Jazelle 技术软件

是功能丰富的多任务 Java 虚拟机（JVM），经过高度优化，可利用许多 ARM 处理器内核中提供的 Jazelle 技术体系结构扩展。还包括功能丰富的多任务虚拟机（MVM），领先的手机供应商和 Java 平台软件供应商提供的许多 Java 平台中均集成了此类虚拟机。通过利用基础 Jazelle 技术体系结构扩展，ARM MVM 软件解决方案可提供高性能应用程序和游戏，快速启动和应用程序切换，并且使用的内存和功耗预算非常低。

- TrustZone：安全扩展，提供可信计算，是系统范围的安全方法，针对高性能计算平台上的大量应用，包括安全支付、数字版权管理（DRM）和基于 Web 的服务。TrustZone 技术与 Cortex-A 处理器紧密集成，并通过 AMBA AXI 总线和特定 TrustZone 系统 IP 块在系统中进行扩展。此系统方法意味着，现在可保护外设（包括处理器旁边的键盘和屏幕），以确保恶意软件无法记录安全域中的个人数据、安全密钥或应用程序，或与其进行交互。用例包括：实现安全 PIN 输入，在移动支付和银行业务中加强用户身份验证，安全 NFC 通信通道，数字版权管理，数字版权管理，基于忠诚度的应用，基于云的文档的访问控制，电子售票移动电视。
- SIMD：单指令多数据，当前的智能手机和 Internet 设备必须提供高级媒体和图形性能，才具有竞争力。ARMv6 和 ARMv7 体系结构中的 SIMD 扩展改进了此类性能。SIMD 扩展已经过优化，可适用于众多软件应用领域，包括视频和音频编解码器，这些扩展将性能提高了 75%或更多。
- NEON：通用 SIMD 引擎可有效处理当前和将来的多媒体格式，从而改善用户体验。可加速多媒体和信号处理算法（如视频编码/解码、2D/3D 图形、游戏、音频和语音处理、图像处理技术、电话和声音合成）。可增强许多多媒体用户体验（观看任意格式的任意视频、编辑和强化捕获的视频-视频稳定性、游戏处理、快速处理几百万像素的照片、语音识别）。
- Virtualization：随着软件复杂性的提高，对在同一个物理处理器上提供多种软件环境的要求也同时增多。因为隔离、可靠性或不同实时特征而要求分隔的软件应用程序需要一个具备所需功能的虚拟处理器。通过高能效方式提供虚拟处理器要求组合利用硬件加速和高效的软件虚拟机监控程序。云计算和其他面向数据或内容的解决方案增加了对每个虚拟机的物理内存系统的需求。
- 多核技术 ARM MPCore：除了 Cortex-A8 外，其他（A5、A9、A15）都支持 ARM 的第二代多核技术：单核到四核实现，支持面向性能的应用领域，支持对称和非对称的操作系统实现。技术允许设计时可配置的处理器支持一个、两个、三个或四个处理器一起运行，同时保持集成的高速缓存一致性。这些多核处理器群集在 1 级高速缓存边界内完全一致，而且可通过加速器一致性端口（ACP）配置将有限的一致性扩展到其余的芯片上系统（SoC）中。ACP 允许系统主外设和带有未经缓存的内存视图的加速器（如 DMA 引擎或加密加速器内核）共享处理器的高速缓存，同时保持高速缓存完全一致。多核群集包括一个与全局中断控制器（GIC）体系结构兼容的带专用外设的集成中断和通信系统，因此可提高性能和简化软件可移植性。此 GIC 可配置为支持 0（旧版 Bypass 模式）至 224 个独立中断源，以此为大量设备提

供低延迟中断途径。该处理器可支持单核或双核 64 位 AMBA 3AXI 互连接口，以及 SoC 内不同地址空间之间的全速过滤选项。

2.1.3 ARM 处理器的分类

对于 ARM 的处理器而言，其目前有 Classic（传统）系列、Cortex-M 系列、Cortex-R 系列、Cortex-A 系列和 Cortex-A50 系列 5 个大类。

1．Classic（传统）系列

Classic（传统）系列处理器上市已经超过 15 年，其中的 ARM7TDMI 依然是市场占有率最高的 32 位处理器，该系列处理器由三个子系列九种处理器组成。

- ARM7 系列：基于 ARMv3 或 ARMv4 架构，包括 ARM7TDMI-S 和 ARM7EJ-S 处理器。
- ARM9 系列：基于 ARMv5 架构，包括 ARM926EJ-S、ARM946E-S 和 ARM968E-S 处理器。
- ARM11 系列：基于 ARMv 架构，包括 ARM1136J(F)-S、ARM1156T2(F)-S、ARM1176JZ(F)-S 和 ARM11MPCore 处理器。

注意：Classic 系列处理器在很大程度上已经逐步被 Cortex 系列所取代，所以在最新的设计中并不推荐使用该系列处理器，尤其是 ARM7 系列。

ARM 的 Classic（传统）系列内核主要基于三种主要技术产品而构建，可以用于各种应用领域，表 2.2 给出了它们的技术说明。

表 2.2 ARM 的 Classic 系列内核

技　术	ARM7 ARMv4T 架构	ARM9 ARMv5TE 架构	ARM11 ARMv6 架构
ARM ISA（指令集架构）	√	√	√
Thumb ISA	√	√	√
Thumb-2 ISA	×		√（仅 ARM1156T2-S）
DSP 扩展	×	√	√
SIMD 扩展	×	×	√
Jazelle 字节码支持	×	√（仅 ARM926EJ-S）	√（ARM1156T2-S 除外）
浮点支持	×	√（VFP9）	√（VFP11）
TrustZone 安全扩展	×	×	√（仅 ARM1176JZ(F)-S）
cache 支持	×	√	√
TCM（紧密耦合内存）支持	×	√	√

ARM 的 Classic 系列处理器具有经济实惠的特点，其可以以多种形式进行授权，可以提供单次使用许可、多年期许可和永久使用许可，并且其中多种处理器可用作硬核从而可以降低设计风险并且缩短上市时间。在 ARM 的 Connected Community（合作伙伴联盟）中有 650 多家成员支持 ARM 处理器，提供了专门针对 ARM 指令集架构的关键开放源项目；提供了行业中最广泛的编译器、调试器和 RTOS 工具体系，以及大量可以和处理器集成的第三方 IP。

ARM 的 Classic 系列处理器的比较说明如表 2.3 所示，需要说明的是，在最新的设计中都推荐使用 Cortex 产品进行升级替换。

表 2.3　ARM 的 Classic 系列处理器的比较说明

系　列	型　号	说　明	Cortex 系列替代产品
ARM11	ARM11MPCore	率先采用了多核技术，主要应用于手机、导航设备及智能本	Cortex-A9 Cortex-A5
	ARM1176JZ(F)-S	是 Classic 系列中性能最高的单核处理器，引入了 TrustZone 技术，主要应用于手机、机顶盒、数字电视等	Cortex-A9 Cortex-A8 Cortex-A5
	ARM1156T2(F)-S	是 Classic 系列中性能最高的实时处理器，首次引入了 Thumb-2 指令集架构，主要应用于高性能确定性控制系统，如汽车、工业控制和机器人等	Cortex-R4
	ARM1136J(F)-S	其与 ARM926EJ-S 类似，但是扩展了流水线，提升了频率和性能，还引入了基本 SIMD（单指令多数据），可以提高编码、解码的性能，并且可以提供可选浮点支持	Cortex-A5
ARM9	ARM968E-S	是面积最小、功耗最低的 ARM9 处理器，并且可以轻松地通过标准接口集成紧密耦合内存	Cortex-R4
	ARM946E-S	是包含了可选 cache 接口并且有完整的内存保护单元的实时处理器	Cortex-R4
	ARM926EJ-S	是入门级处理器，但是可以支持包括 Linux、Windows CE 和 Symbian 在内的完全版操作系统，所以是需要完整图形用户界面的应用的理想选择	Cortex-A5
ARM7	ARM7TDMI-S	是出色的重负荷处理器，曾经被用于手机，现在广泛地用于移动和非移动领域	Cortex-M3 Cortex-M0

2．Cortex-M 系列

Cortex-M 系列处理器包括 Cortex-M0、Cortex-M0+、Cortex-M1、Cortex-M3、Cortex-M4 共 5 个子系列，该系列主要针对成本和功耗敏感的应用，如智能测量、人机接口设备、汽车和工业控制系统、家用电器、消费性产品和医疗器械等。

整体来说，Cortex-M 系列处理器偏重于工业控制，其提供了更低的功耗和更长的电池寿命，提供了更少的代码和更高的性能，并且提供了兼容性的代码、统一的工具和操作系统支持，具有如下优点。

- Cortex-M 系列处理器为 8 位和 16 位体系结构提供了极佳的代码密度，在具有对内存大小要求苛刻的应用中具有很大的优势。
- Cortex-M 系列处理器虽然使用 32 位的指令，但是其使用了可提供极佳代码密度的 ARM Thumb-2 技术，也可以支持 16 位的 Thumb 指令，其对应的 C 编译器也会使用 16 位版本的指令，可以更加有效地执行运算。
- Cortex-M 系列处理器采用了 8 位和 16 位的数据传输，从而可以高效地利用数据内存，同时开发者可以使用其在面向 8/16 位系统的应用代码中的相同的数据类型。
- Cortex-M 系列处理器提供了较大的能效优势，面对如 USB、蓝牙、WiFi 等连接及如加速计和触摸屏等复杂模拟传感器且成本日益降低的产品需求有极大地优势。
- Cortex-M 处理器完全可以通过 C 语言编程，并且附带了各种高级调试功能，能帮助定位软件中的问题，同时网上具有大量的应用实例可以参考。

表 2.4 给出了 Cortex-M 系列处理器的对比，其升级关系如图 2.3 所示，所有的 Cortex-M 系列处理器都是二进制向上兼容的，可以很方便地重用软件及从一个 Cortex 处理器无缝

发展到另外一个。

表 2.4　Cortex-M 系列处理器的对比

Cortex-M0	Cortex-M0+	Cortex-M3	Cortex-M4
8/16 位应用	8/16 位应用	16/32 位应用	32 位/DSC 应用
低成本和简单性	低成本，最佳性能	高性能，通用	有效的数字信号控制

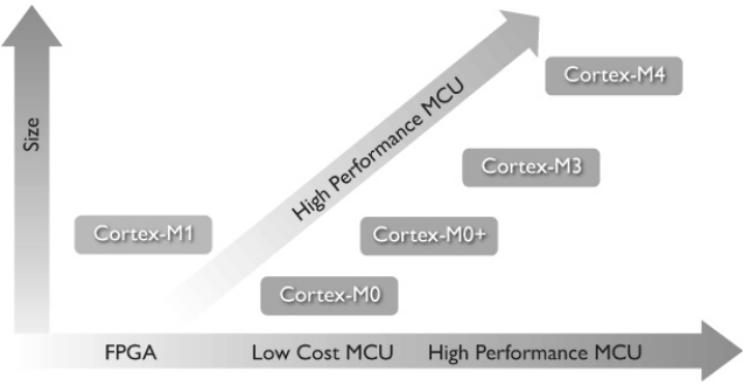


图 2.3　Cortex-M 系列处理器的升级关系

- Cortex-M0：Cortex-M0 是最小的 ARM 处理器，体积极小、能耗很低且编程所需要的代码占用量极少，其具有低功耗（90LP 工艺的最低配置下门数低于 12KB 的时候能耗只有 16μW/MHz）、简单（只有 56 个指令且架构对 C 语言友好，提供了可供选择的具有完全确定性的指令和中断计时，使得计算响应时间十分容易）和优化的连接性（支持实现低能耗网络互连设备）等特点，常见的 Cortex-M0 处理器有 NXP 的 LPC1100 系列、意法半导体的 STM32F0 系列。
- Cortex-M0+：Cortex-M0+是在 Cortex-M0 基础上开发的能效极高的处理器，其保留了 Cortex-M 的全部指令集和数据兼容性，同时进一步降低了能耗，其和 Cortex-M 一样，芯片面积很小、功耗极低，所需的代码量极少，使得开发人员可以直接跳过 16 位系统以接近 8 位系统的成本开销获取 32 位系统的性能。其具有一个只有 2 级的流水线，具有低功耗（90LP 工艺的最低配置下门数低于 12KB 的时候能耗只有 11.2μW/MHz）、简单（保留了 Cortex-M0 的 56 个指令）和多功能性（如内存保护单元、可重定位的矢量表、用于提高控制速度的单周期 I/O 接口和用于增强调试的 Micro Trace Buffer）等特点，常见的 Cortex-M0+处理器有 NXP 的 LPC1100 系列和 Atmel 的 SAM D20 系列。

注意：在当前的实际产品中基本都使用 Cortex-M0+来替代 Cortex-M0。

- Cortex-M1：Cortex-M1 类型的 ARM 处理器是为了在 FPGA 中应用而设计的，其支持包括 Actel、Altera 和 Xilinx 公司的 FPGA 设备，可以满足 FPGA 应用的高质量、标准处理器架构的需要，开发人员可以在受行业中最大体系支持的单个架构上进行

标准设计以降低其硬件和软件工程成本，所以在通信、广播、汽车等行业得到了广泛应用。其可以实现常用高密度 Thumb-2 指令集的最新型三阶段 32 位 RISC 处理器，其可以使处理器和软件占用空间都满足最小 FPGA 设备的面积预算，同时保留与 ARM7TDMI 处理器上任何 ARM 处理器 Thumb 代码的兼容性，其可以提供 0.8DMIPS/MHz。

- Cortex-M3：Cortex-M3 处理器是行业领先的 32 位处理器，适用于具有较高确定性的实施应用，如汽车车身系统、工业控制系统、无线网络和传感器等，其具有出色的计算性能及对事件的优异系统响应能力。其具有较高的性能和较低的动态功耗，支持硬件除法、单周期乘法和位字段操作在内的 Thumb-2 指令集，最多可以提供 240 个具有单独优先级、动态重设优先级功能和集成系统时钟的系统中断。Cortex-M3 相比 Cortex-M0 来说提供了更高的性能和更丰富的功能，于 2004 年推出，其将集成的睡眠模式与可选的状态保留功能相结合，具有较高的性能和较低的动态功耗，所以可以提供领先的能效。其提供了包括了硬件除法、单周期乘法和位字段操作在内的 Thumb-2 指令集以获取最佳的性能和代码大小；其还可以高效地处理多个 I/O 通道和类似 USB OTG 的协议标准。其常见的型号有 Atmel 的 SAM3N（无与伦比的性能和易用性）、SAM3S（低功耗和简化的 PCB 应用）、SAM3U（带高速 USB 接口）、SAM3A（CAN 总线应用）、SAM3X（增强型网络应用）；NXP 的 LPC1300 系列和 LPC1700 系列；德州仪器（TI）的 TMS470M 系列、Stellaris 系列、C2000 Concerto 28x 系列；意法半导体的 STM32F1 和 STM32F2 系列。
- Cortex-M4：Cortex-M4 是 Cortex-M3 的升级版，其提供了无可比拟的功能，将 32 位控制与领先的数字信号处理技术集成来满足需要很高能效级别的市场，曾经在 Elektra2010（European Electronics Industry Awards 2012）上获得了大奖， 其主要实际应用型号包括 Atmel 的 SAM4L、SAM4S、德州仪器（TI）的 TM4C 系列和意法半导体的 STM32F3 系列。

3．Cortex-R 系列

Cortex-R 系列处理器包括 Cortex-R4、Cortex-R5、Cortex-R7 共 3 个子系列，其对低功耗、良好的中断行为、卓越性能及与现有平台的高兼容性这些需求进行了平衡考虑，具有高性能、实时、安全和经济实惠的特点，面向如汽车制动系统、动力传动解决方案、大容量存储控制器等深层嵌入式实时应用。

Cortex-R 系列处理器使用了深度流水化微架构及指令预取、分支预测和超标量执行等性能增强技术，提供了硬件除法、浮点单元（FPU）选项和硬件 SIM DSP，采用了可以在不牺牲性能的前提下实现高密度代码的带 Thumb-2 指令的 ARMv7-R 架构和带指令、指令 cache 控制器的哈佛架构，并且拥有获得快速响应代码和数据处理器本地的紧密耦合内存（TCM）和高性能 AMBA3 的 AXI 总线接口，其具有如下特点。

- 高性能：可以快速地执行复杂代码和 DSP 功能，其使用了高性能、高时钟频率、深度流水化的微架构；使用了双核多处理（AMP/SMP）配置；使用了可以用于超高性能 DSP 和媒体功能的硬件 SIMD 指令。
- 实时性：可以保证响应速度和高吞吐量的确定性操作；其有快速、有界且确定性的

中断响应；有用于获得快速响应代码/数据的处理器本地的紧密耦合内存（TCM）；有可加快终端进入速度的低延迟中断模式（LLIM）。

- 安全性：可以检测错误并保证可靠的系统运行，其具有内存保护单元（MPU）的用户和授权软件操作模式；有由于 1 级内存系统及总线的 ECC 和奇偶校验错误检测/更正；有双核锁步（DCLS）冗余内核配置。
- 经济实惠。

Cortex-R 系列处理器和 Cortex-M 和 Cortex-A 系列处理器都不同，其提供的性能比 Cortex-M 系列要高得多，可以作为 Classic 系列处理器中的 ARM9、ARM11 系列处理器的升级产品；但是其又不像 Cortex-A 系列处理器更偏重于面向必须使用虚拟内存管理技术的复杂软件操作系统，从图 2.4 中可以看出它们的升级关系。

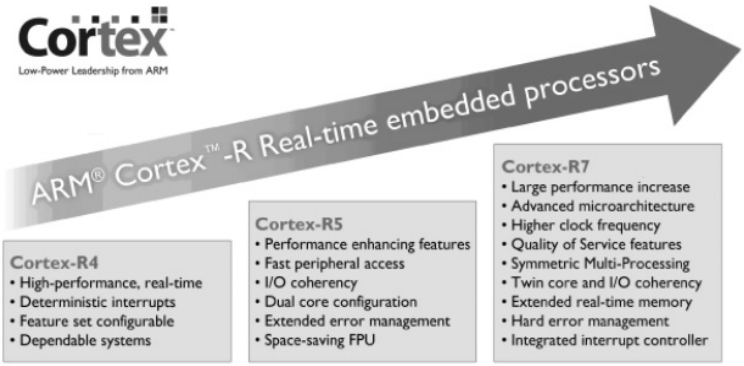


图 2.4　Cortex-R 系列处理器的升级关系

- Cortex-R4：Cortex-R4 处理器是第一款基于 ARMv7-R 架构的深度嵌入式实时处理器，主要用于高产量、深入嵌入式的片上系统应用，如硬盘驱动控制器、无线基带处理器、消费类产品和汽车系统的电子控单元等；其能提供更高的性能、实时的响应速度、可靠性和高容错性。Cortex-R4 在使用主流低功耗工艺技术（40nm LP）进行实现的时候，可以做到 600MHz 以上的最大时钟频率，性能可以达到 1.66 DMIPS/MHz，效率到达 24 DMIPS/mW 以上。
- Cortex-R5：Cortex-R5 处理器是在 Cortex-R4 基础上扩展功能集得到的，支持在可靠的实时系统中获得更高级别的系统性能、提高效率和可靠性并加强错误管理，提供了一种从 Cortex-R4 处理器向上迁移到更高性能的 Cortex-R7 处理器的简单迁移途径；通常用于市场上的实时应用提供高性能解决方案，包括移动基带、汽车、大容量存储、工业和医疗市场。Cortex-R5 是为了实现高级芯片工艺而设计的，其重点是更高的能效、实时的响应速度、高级功能和简单的系统设计，其提供了高度灵活且有效的双周期本地内存接口，并且集成了包括 LLPP（低延迟外设端口）在内的许多高级系统级功能以帮助进行软件开发。Cortex-R5 在使用主流低功耗工艺技术（40nm LP）进行实现的时候，可以做到 600MHz 以上的最大时钟频率，性能可以达

到 1.66 DMIPS/MHz，效率达到 24 DMIPS/mW 以上。

- Cortex-R7：Cortex-R7 同样是为实现高级芯片工艺而设计的，其重点是更高的能效、实时的响应速度、高级功能和简单的系统设计，提供了支持紧耦合内存（TCM）本地共享内存和外设端口的灵活本地内存系统，使 SoC 设计人员可以在受到限制的芯片资源内达到高标准的硬实时要求。Cortex-R7 在使用主流高性能移动工艺技术（28nm HPM）进行实现的时候，可以做到 1GHz 以上的最大时钟频率，性能可以达到 2.5 DMIPS/MHz，效率达到 27 DMIPS/mW 以上。

4．Cortex-A 系列

Cortex-A 处理器包括 Cortex-A5、Cortex-A7、Cortex-A8、Cortex-A9、Cortex-A12 和 Cortex-A15 共 6 个子系列，用于具有高计算要求、运行丰富操作系统及提供交互媒体和图形体验的应用领域，如智能手机、平板电脑、汽车娱乐系统、数字电视等。

Cortex-A 系列处理器具有以下特点。

- 是移动互联网的理想选择，其为 Adobe Flash10.1 以上版本提供了原生支持；提供了高性能 NEON 引擎，广泛支持媒体解码器；采用了低功耗设计，支持全天浏览和连接。
- 其具有高性能，可以为其目标应用领域提供各种可伸缩的能效性能点。
- 都支持 ARM 的第二代多核技术；支持面向性能的应用领域的单核到四核的实现；支持对称和非对称的操作系统实现；并且可以通过加速器一致性端口（ACP）在系统的整个处理器中保持一致性。并且 Cortex-A5 和 Cortex-A15 将多核一致性扩展至 AMBA ACE 的 1～4 核群集以上，支持 big LITTLE（ARM big.LITTLE™ 处理是一项节能技术，它将最高性能的 ARM 处理器与最高效的 ARM 处理器结合到一个处理器子系统中，与当今业内最优秀的系统相比，不仅性能更高，能耗也更低）处理。
- 除了具有与上一代经典 ARM 和 Thumb 架构的二进制兼容性之外，Cortex-A 类处理器还可以通过 Thumb-2、TrustZone 安全扩展、Jazelle 技术扩展来提供更多的优势。

Cortex-A 系列处理器均适用于各种不同的性能应用领域，其共享 ARMv7-A 的架构和功能集，成为开放式平台设计的最佳解决方案并且可以为不同设计之间的软件提供兼容性和可移植性，其提供了 ARM、Thumb-2、Thumb、Jazelle、DSP 的指令集支持、TrustZone 安全扩展、高级单精度和双精度浮点支持、NEON 媒体处理引擎及对包括 Linux 全部分发版本（Android、Chrome、Ubuntu 和 Debian）、Linux 第三方（MontaVista、QNX、Wind River、Symbian、Windows CE）、需要使用内存管理单元的其他操作系统支持。Cortex-A 也提供了各种显著不同的特点，如表 2.5 所示。

表 2.5　Cortex-A 系列处理器比较

内核	Cortex -A5	Cortex-A5 多核	Cortex-A8	Cortex-A9	Cortex-A9 多核	Cortex-A9 硬核	Cortex-A15 多核	Cortex-A7 多核
架构	ARMv7	ARMv7 + MP	ARMv7	ARMv7	ARMv7 + MP	ARMv7 +MP	ARMv7 +MP +LPAE	ARMv7 +MP +LPAE
中断控制器	GIC-390	已集成-GIC	GIC-390	GIC-390	已集成-GIC	已集成-GIC	已集成-GIG	GIC-400

续表

L2 cache 控制器	L2C-310	L2C-310	已集成	L2C-310	L2C-310	L2C-310	已集成	已集成
预期实现	300～800MHz	300～800MHz	600～1000MHz	600～1000MHz	600～1000MHz	800～2000MHz	1000～2500MHz	800～1500MHz
DMIPS/MHz	1.6	1.6/处理器	2.0	2.5	2.5/处理器	5.0/双核	TBC	1.9/处理器

- Cortex-A5：Cortex-A5 处理器是体积最小、功耗最低的应用型处理器，并且可以带来完整的网络体验，可为现有的 ARM926EJ-S 和 ARM1176JZ-S 处理器设计提供高价值的迁移途径。它可实现比 ARM1176JZ-S 更好的性能、比 ARM926EJ-S 更好的功效和能效，以及 100%的 Cortex-A 兼容性。
- Cortex-A7：Cortex-A7 处理器是一种高能效应用处理器，除了低功耗应用外，还支持低成本、全功能入门级智能手机，该处理器与其他 Cortex-A 系列处理器完全兼容并整合了高性能 Cortex-A15 处理器的所有功能，包括虚拟化、大物理地址扩展（LPAE）NEON 高级 SIMD 和 AMBA 4 ACE 一致性。单个 Cortex-A7 处理器的能效是 ARM Cortex-A8 处理器的 5 倍，性能提升了 50%，而尺寸仅为后者的五分之一，支持如今的许多主流智能手机。目前提供 Cortex-A7 的厂商包括德州仪器（TI）、三星（SAMSUNG）、飞思卡尔（Freescale）、博通（Broadcom）、海思半导体（HISILICON）和 LG。
- Cortex-A8：Cortex-A8 处理器基于 ARMv7 架构，支持 1GHz 以上的工作频率，采用了高性能、超标量微架构及用于多媒体和 SIMD 处理的 NEOD 技术，可以满足 300mW 以下运行的移动设备的低功耗要求，并且与 ARM926、ARM1136 和 ARM1176 处理器的二进制兼容，目前提供 Cortex-A8 的厂商有德州仪器（TI）、三星（SAMSUNG）、飞思卡尔（Freescale）、博通（Broadcom）和 ST（意法半导体）。
- Cortex-A9：Cortex-A9 处理器是低功耗或散热受限的成本敏感型设备的首选处理器，其支持多核，在用作单核心的时候性能比 Cortex-A8 提升了 50%以上，其主要用于主流智能手机、平板电脑、多媒体播放器等。还可以提供多达 4 个处理器集成，在必要的时候能实现轻量级的工作量及峰值性能。
- Cortex-A12 处理器：其是 Cortex-A9 的升级版，专注应用于智能手机和平板电脑，提供了对 1TB 存储空间的支持，在同功耗下其比 Cortex-A9 性能提升了大约 40%，通常来说该芯片使用较少，故在此不多再赘叙。
- Cortex-A15：Cortex-A15 处理器是 Cortex-A 系列处理器的最新产品，也是最高性能产品，和其他处理器系列兼容，具有无序超标量流水线，带有紧密耦合的大小可以达到 4MB 的低延迟 2 级 cache；改进后的浮点和 NEON 媒体性能可以给用户提供下一代的体验并为 Web 基础结构应用提供高性能计算。其通常应用于移动计算、高端数码家电、服务器和无线基础架构。Cortex-A15 是基于 ARMv7 架构的处理器，其和 Cortex-A 系列的其他处理器完全兼容，所以支持相当多成熟的开发平台和软件体系，包括 Android、AFP（Adobe Flash Player）、Java Platform Standard Edition（Java

SE）、JavaFX、Linux、Microsoft Windows Embedded、Symbian 和 Ubuntu 等。Cortex-A15 还支持 big.LITTLE 技术，其可以和 Cortex-A7 处理器配对并且确保合适的工作分配给合适的处理器，以达到合适的性能和功耗的平衡。

- Cortex-A50 系列：Cortex-A50 系列处理器基于 ARMv8 架构，可以在 AArch32 执行状态下为 ARMv7 的 32 位代码提供更好的性能，也可以在 AArch64 执行状态下支持 64 位数据和更大的虚拟寻址空间，其允许在 32 位和 64 位之间进行完全的交互操作，因此可以从运行 32 位 ARMv7 应用程序的 64 位操作系统开始，迁移到在同一系统中混合运行 32 位应用程序和 64 位应用程序，最终一步步迁移到 64 位系统。其提供了 A53 和 A57 两种型号的处理器，由于该系列处理器刚刚发展起来，还没有大规模实际应用，在此不多做赘述。

5．ARM 处理器选择总结

通常来说，作为工业控制处理器，可以选择 Cortex-M 系列处理器，其中 M0 比较简单便宜，适合用于替代 51 单片机，其缺点是不带 MMU，需要直接编写/运行控制代码；Cortex-R 系列处理器可以取代 ARM9 作为具有带操作系统的控制系统；Cortex-A 系列处理器更加常用的场合是消费电子，但是其中的 Cortex-A5 处理器也经常用于工业控制场合。

2.2　嵌入式系统的存储器件

存储器是用于存放系统软件和用户软件的载体。前者包括了启动引导软件、操作系统和硬件驱动；后者是用户用于实现应用目的而设计的软件。在嵌入式系统设计中需要考虑如何安排这些存储器的地址，尤其是 NOR Flash 和 SDRAM，因为它们涉及了系统的启动步骤等。嵌入式硬件中常用的存储器件包括 SDRAM、FLASH、E^2PROM、大容量存储系统（SD 卡、U 盘、硬盘）等。

2.2.1　SDRAM

SDRAM 是同步动态随机存储器（Synchronous Dynamic Random Access Memory），通常其作为嵌入式系统的内存，等同于普通 PC 的内存。

SDRAM 从发展到现在已经经历了 4 代，分别为 SDR SDRAM、DDR SDRAM、DDR2 SDRAM 和 DDR3 SDRAM。

第 1 代 SDR SDRAM 采用单端（Single-Ended）时钟信号，而从第 2 代～第 4 代则由于工作频率比较快，所以采用了可降低干扰的差分时钟信号作为时钟信号，该时钟信号即为数据存储的频率。第 1 代 SDRAM 采用时钟频率来命名，如 PC100、PC133 则表示时钟频率为 100MHz/133MHz，数据读/写速率也为 100MHz/133MHz。从第 2 代开始的 DDR SDRAM 则采用数据读/写速率命名，并且在前面加上表示其 DDR 代数编码的数字，如 PC2700 是 DDR333，其工作频率是 333MHz/2=160MHz，2700 则表示带宽为 2.7GB。

DDR 的读/写频率为 DDR200～DDR400；DDR2 从 DDR2-400～DDR2-800；DDR3 从 DDR3-800～DDR3-1600。

注意：DDR4 内存规范已经完成，其对应的服务器、消费级产品已经逐步普及，而 DDR4 SO-DIMM 笔记本内存也已经开始产品化（如美光的 Crucial）。

1．SDRAM 的主要参数

SDRAM 的两个主要参数说明如下。

- 容量：SDRAM 的容量通常用“存储单元×体×每个存储单元的位数”来表示，例如，某 SDRAM 芯片的容量为 4M×4×8bit，则表明该存储器芯片的容量为 16MB（字节），或者 128Mb。
- 时钟周期：SDRAM 能运行的最大频率，如对应 PC100 的 SDRAM 则表示其时钟周期为 10ns，工作频率则为 100MHz。

此外，SDRAM 还有存取时间、CAS 延迟时间、综合性能评价等参数，在此不再赘述。在 SDRAM 中有两个很重要的概念：物理 Bank 和芯片位宽。

- 物理 Bank：处理器和 SDRAM 进行数据交互所需要的数据总线的位宽。
- 芯片位宽：SDRAM 芯片的数据总线的位宽。

通常来说，物理 Bank 会大于等于芯片的位宽，在这个时候 SDRAM 就需要将多块芯片组合到一起以满足物理 Bank 的需求。

2．SDRAM 的分类特点

SDRAM 的分类和特点如表 2.6 所示。

表 2.6　SDRAM 的分类和特点

参　数	SDR SDRAM	DDR SDRAM	DDR2 SDRAM	DDR3 SDRAM
核心频率（MHz）	66～166	100～200	100～200	100～250
时钟频率（MHz）	66～166	100～200	200～400	400～1000
数据传输率（Mbps）	66～166	200～400	400～800	800～2000
预取设计	1bit	2bit	4bit	8bit
突发长度	1/2/4/8full page	2/4/8	4/8	8
CL 值	2/3	2/2.5/3	3/4/5/6	5/6/7/8/9
Bank 数量	2/4	2/4	4/8	8/16
工作电压	3.3V	2.5/2.6V	1.8V	1.5V
封装	TSOP II-54	TSOP II-54/66	FBGA60/68/84	FBGA78/96
生产工艺（nm）	90/110/150	沿用 SDR 生产线 70/80/90	53/65/70/90	45/50/65
容量标准（Byte）	2M～32M	8M～128M	32M～512M	64M～1G
新增特性		查分时钟，DQS	ODT、OCD、AL、POSTED CAS	异步重置 Reset
优点	制造工艺简单，TSOP 封装焊接拆卸方便，成品率高	数据传输率有所提高，生产设备简单	数据传输速率高、更好的电气性能与散热性、体积小、功耗大、无需上拉终结电阻、成本相对较低	工作频率进一步提高，功耗和发热量跟小，容量更大
缺点	速度低、焊盘与 PCB 接触面积小、散热差、高频阻抗和寄生电容影响稳定性和频率提升	容量受限、高频时稳定性和散热性差、需要大量终结电阻	CL 延迟增加、成品率较低	价格较高

3．SDRAM 的典型芯片

SDRAM 的典型结构如图 2.5 所示，这是一块 4M×4×8bit 的 HYB25L35610，可以看到其共有 54 个引脚，可以分为如下的几大部分。

- 地址输入引脚：执行 ACTIVE 命令和 READ/WRITE 命令时用于决定使用的 bank。
- 时钟输入引脚：高电平有效，当该引脚处于低电平期间，提供给所有 bank 预充电和刷新的操作。
- 片选信号引脚：在多存储芯片架构中选择进行存取的芯片。
- 行地址选通和列地址选通引脚。
- bank 地址输入信号引脚：决定使用激活的 bank，如果引脚数目为 n，2^n 则为 bank 的数量。
- 输入/输出屏蔽引脚。
- 电源相关引脚。

常见的 SDRAM 实体都是以模组形式存在的，即内存条，如图 2.6 所示。

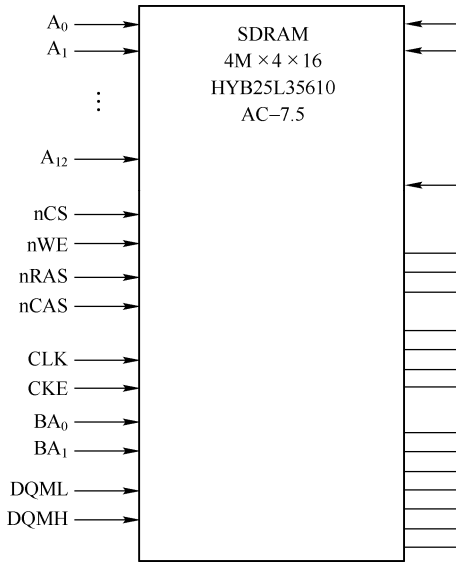


图 2.5　SDRAM 的典型结构

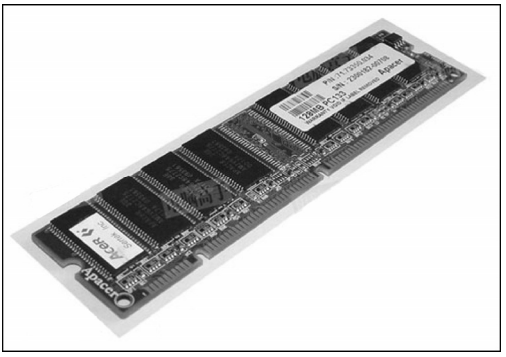


图 2.6　常见的 SDARM 实体

而在嵌入式系统中，最常用的是 Hynix（海力士，原现代）、Micron（美光）、Spectek（镁光）、Elpida（必尔达）生产的 8 位/16 位数据宽度、工作在 3.3V 电压下的单个 SDRAM 芯片，它们的典型产品如表 2.7 所示。

表 2.7　典型的 SDRAM 产品

厂　商	SDRAM	DDR SDRAM	DDR2 SDRAM
Hynix	HY57V64820HGT-H（8M×8 PC133）、HY57V561620FTP-H-A（16M×16 PC133）、HY5S7B2ALFP-6E-C（16M×32 PC133）	HY5DU281622FTP-D43-C（8M×16 PC400）、HY5DU12822 CTP-J-C（64M×8 PC333）、HY5DU12822CTP-D43（64M×8 PC400）	HY5PS12821EFP-Y5（64M×8 PC667）、HY5PS1G1631CFP-Y5-C（64M×16 PC667）、HY5PS12821CFP-S5（64M×8 PC800）、HY5PS1G831CFP-S6（128M×8 PC800）

续表

厂　商	SDRAM	DDR SDRAM	DDR2 SDRAM
Micron	MT48LC16M16A2P-7（16M×16 PC143）、MT48LC4 M32B2P-6（4M ×32 PC166）	MT46V64M8P-5B（64M×8 PC 400）、MT46V128M4P-5B（128M×8 PC400）	MT47H32M16HQ-25（32M×16 PC800）、MT47H128M 8HQ-25:E（32M×8 PC800）、MT47H 64M16 HR-3:E（64M×16 PC667）
Spectek	S16008LK6TKF-75A（8M×16 PC133）、S16004LK6TKF-75A（4M×16 PC133）	—	—
Elpida	EDS2516CDTA-75-E（16M×16 PC133）、EDS6432AFTA-6B-E（2M×32 PC166）	EDD5108ADTA-5C（64M×8 PC400）、EDD5108AGTA-5B-E（64M×8 PC533）	EDE5108ABSE-5C-E（64M×8 PC533）、EDE5104AESK-4A-E（128M×4 PC400）

单个 SDRAM 芯片如图 2.7 所示，这是一片 HY5DU 12822BT-D43。

图 2.7　单个 SDRAM 芯片

2.2.2　FLASH

SDRAM 是嵌入式系统的内存，而 FLASH 则可以看作嵌入式系统的硬盘，主要用于存放嵌入式系统运行所必需的数据，如操作系统和应用程序等。

1．FLASH 的分类

目前市场上的 FLASH 可以分为由 Intel 公司在 1988 年发布的 NOR 和东芝公司在 1989 年发布的 NAND 两大类，它们因为其内部结构分别与“或非门”和“与非门”相似而得名，它们的内部结构、外部特性和应用均有较大差异。

通常来说 NOR FLASH 的容量不大，常见的只有几 MB，可以重复擦写的次数较多，可以达到 10 万次～100 万次，遵循 CFI 标准可以通过 CFI 命令查询制造商、型号、容量、内部扇区布局等参数，从而通过软件实现自动配置，并且其可以保证无坏块，每个数据位都是有效的，并且其寻址采用了线性的完整数据和地址线编码，所以 NOR FLASH 通常用于充当嵌入式系统的启动存储器、刻录 U-BOOT 等。NOR FLASH 支持芯片内执行（eXecute In Place，XIP），程序可以直接在 NOR FLASH 中执行，其使用方法和普通的 SDRAM 几乎没有区别。

NAND FLASH 则可以做得很大，从几十 MB～几 GB，其可重复擦写的次数不如 NOR FLASH，但是也可以达到 10 万次，其没有数据线和地址线概念，只有复用的 I/O 引脚和控制引脚，所以必须通过特定的逻辑来操作，不能作为嵌入式系统的启动 ROM（某些在内部固化了对 NADN FLASH 支持代码的处理器例外，如 AT91SAM926x）。NAND FLASH 在出厂时候除了内部第一块外允许有坏块的存在并且可以进行相应的标记处理。

注意：目前 NOR FLASH 和 NAND FLASH 都出现了使用 SPI 接口的串行接口产品，其可以显著地减小电路板的设计难度和面积，容量也和普通的芯片相当，只是有可能不能充当启动 ROM。

NOR FLASH 和 NOR FLASH 的比较如表 2.8 所示。

表 2.8　NOR FLASH 和 NAND FLASH 的比较

特　　点	NOR FLASH	NAND FLASH
传输速率	很高	较低
写入和擦除速度	较低	较高
读速度	较高	较低
写入/擦除操作速度	以 64～128KB 的块进行，时间为 5s	8～32KB 的块，只需要 4ms
外部接口	带有 SRAM 接口，有足够的地址引脚，可以对内部每个字节进行寻址操作	使用复杂的 I/O 引脚来串行的存取数据
价格	较高	较低
单片容量	1～16MB	8～128MB
用途	代码存储	数据存储，如 CF 卡、MMC 卡等
写操作	以字节或字为单位	以页面为基础单位

2．NAND FLASH 的内部结构

NAND FLASH 由 BLOCK（块）构成，块的基本组成单元是 PAGE（页），一个块通常由 16、32 或 64 个页组成。页的大小有两种，其中 small page 的大小为 528 字节，包含了 512 字节的数据存储区和 16 字节的备用区域；对应的 large page 的页则由 2048 字节的数据存储区（Data area）和 64 字节的备用区域（Spare area）组成，图 2.8 给出了两种页面组成的 NAND FLASH 结构示意。

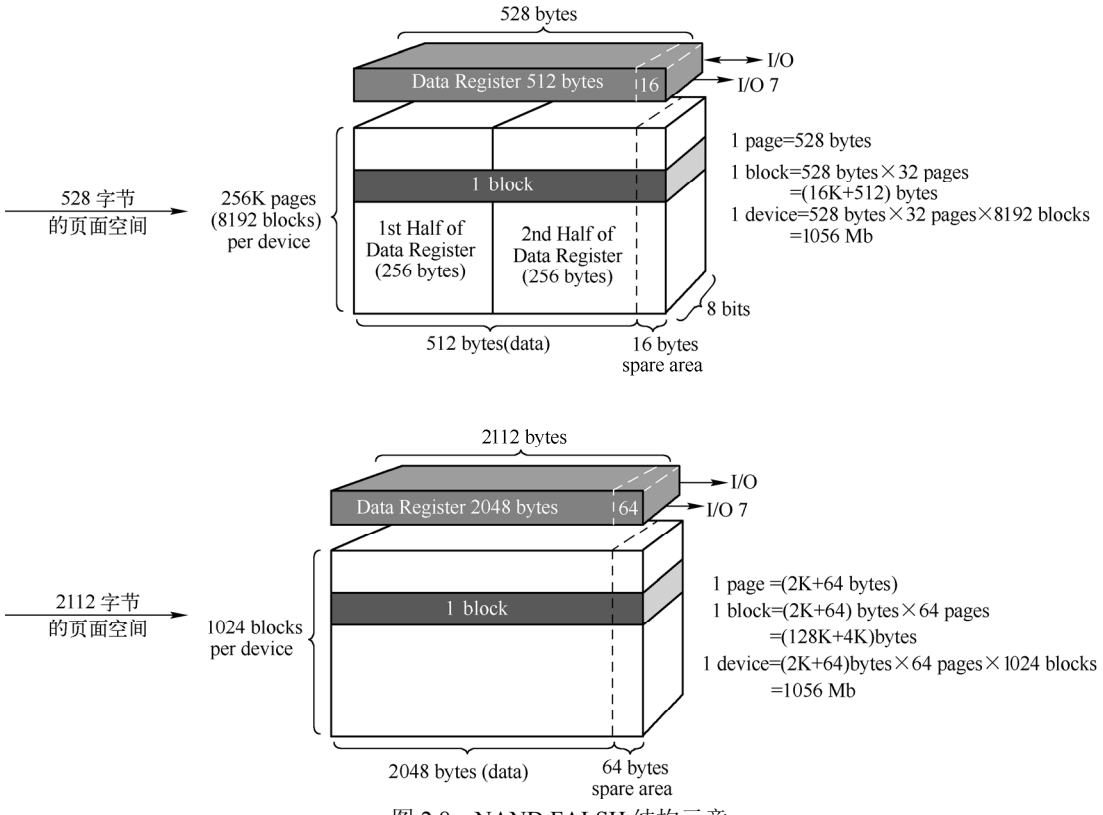


图 2.8　NAND FALSH 结构示意

NAND FLASH 有三种基本操作，页面读、页面写和块擦除：

- 在页面读操作中，该页内的数据首先被读入数据寄存器，然后输出；
- 在页面写操作中，该页内的数据首先被写入数据寄存器，然后写入存储空间中；
- 在块擦除操作中，一组连续的页在单独操作下被擦除。

备用区域（Spare Area）可以用于标识 NAND FLASH 中的坏块（BAD BLOCK），也可以用于和数据存储区（Data Area）一样存储数据，其使用方法可以参考图 2.9，这是三星公司提出的一种标准，包括 8 位的 small page、16 位的 small page、8 位的 large page 和 16 位的 large page 四种格式。

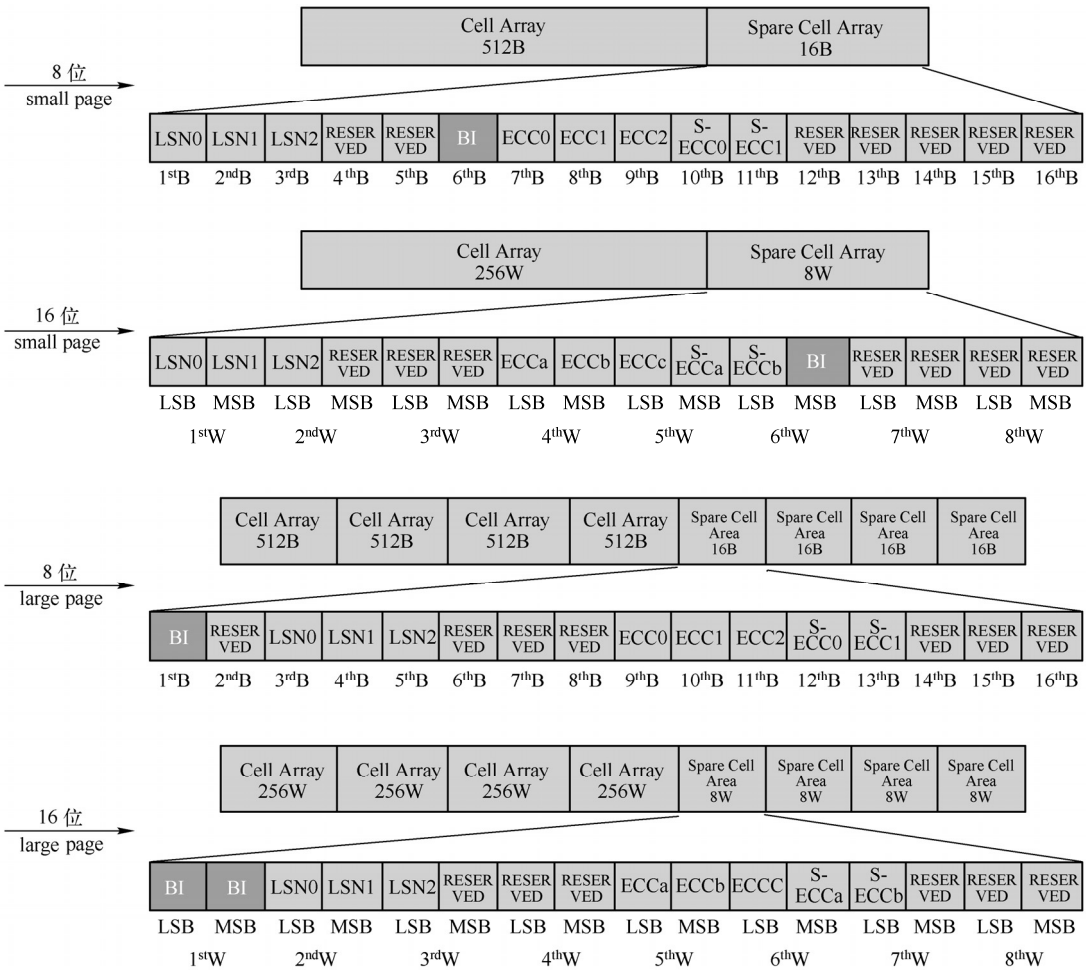


图 2.9　NAND FLASH 的备用区域使用标准

3．常见的 FLASH 芯片型号

常见的 NOR FLASH 和 NAND FLASH 芯片如表 2.9 和表 2.10 所示。

表 2.9　常见 NOR FLASH 型号

厂　商	型　号	容量（B）	结　构	电　压
ST	M29F512B	512K	8 位	5V
	M29F010B	1M		
	M29F002B	2M		
	M29F040B	4M		
	M29080A	8M		
	M29F016B	16M		
	M29F100B M29F102B	1M	8 位/16 位	
	M29F200B	2M		
	M29F400B	4M		
	M29F800A	8M		
	M29F160B	16M		
	M29W512B	512K	8 位	3V
	M29W010B	1M		
	M29W002B M29W022B	2M		
	M29W040B M29W004B	4M		
	M29W008A	8M		
	M29W116B	16M		
	M29W102B	1M	8 位/16 位	
	M29W200B	2M		
	M29W400B	4M		
	M29W800A	8M		
	M29W160B	16M		
ATMEL	AT49F512 AT29C512 AT29C010A	512K	8 位	5V
	AT49F010 AT49HF010A AT49F001	1M		
	AT29C020 AT49F020 AT49F002	2M		
	AT29C040 AT49F040	4M		
	AT49C008 AT49F080	8M		
	AT29F1614 AT49F1614T	16M		
	AT29C1024 AT49F102A AT49F1045	1M	8 位/16 位	5V
	AT49F2048	2M		
	AT49F4096A	4M		
	AT49F8192	8M		
	AT49F1604 AT49F1604T	16M		

续表

厂　商	型　号	容量（B）	结　构	电　压
ATMEL	AT49BV512 AT29LV512	512K	8 位	3V
	AT29BV010A AT49BV010 AT49HBV010 AT49BV001	1M		
	AT29BV020 AT29LV020 AT49BV020 AT49BV002 AT49LV020 AT49LV002	2M		
	AT29BV040A AT29LV040A AT49BV040 AT49LV040	4M		
	AT49BV080 AT49LV080	8M		
	AT29LV1204	1M	8 位/16 位	3V
	AT29BV2048 AT49LV2048	2M		
	AT49BV4096A AT49LV4096A	4M		
	AT49BV8192 AT49LV8192	8M		
hynix	HY29F002T/C	2M	8 位	5V
	HY29F040 HY29F040A	4M		
	HY29F080T/C	8M		
	HY29F400T/G	4M	8 位/16 位	5V
	HY29F800T/G	8M		

注意：此外 SST、Intel、AMD、Micron 等公司也都有对应的产品，用户可以自行查阅相应的手册，从表 2.9 中可以看到常用的 NOR FLASH 最大容量也就是 16MB。

表 2.10　常见 NAND FLASH 型号介绍

厂　商	型　号	容　量	备　注
Samsung	K9F2808	16M	
	K9F5608	32M	
	K9F1208	64M	
	K9K1G08	128M	
	K9E2G08	256M	
	K9W4G08	512M	
	K9K8G08	1G	
	K9WAG08	2G	
	K9NBG08	4G	
	K9NCG08	8G	
	K9MDG08	16G	
	K9PFG08	32G	

续表

厂　商	型　号	容　量	备　注
Micron	MT29F1G08	128M	1.8V
	MT29F2G08	256M	
	MT29F4G08	512M	
	MT29F8G08	1G	
	MT29F8G08A	1G	3.3V
	MT29F16G08	2G	
	MT29F32G08A	4G	
	MT29F64G08	8G	
	MT29F128G08	16G	
Intel	JS29F04G08	512M	
	JS29F08G08	1G	
	JS29F16G08	2G	
	JS29F32G08	4G	
	JS29F64G08	8G	
	JS29F16B08	16G	
	JS29F32B08	32G	
Hynix	HY27US08281A	16M	
	HY27US08561M	32M	
	HY27US08121M	64M	
	HY27UA081G4M	128M	
	HY27UF082G2M	256M	
	HY27UG084G2M	512M	
	HY27UG088G5M	1G	
	HY27UH08AG5B	2G	
	HY27UK08BGFB	4G	
	HY27UW08CGFM	8G	
	H27UAG8M2MYR	16G	

注意：此外 Spectek、Renesas、ST、Toshiba、SanDisk 等公司也都有相应的产品，用户可以自行查阅相应的手册，从表 2.10 中可以看到常用的 NAND FLASH 的容量通常在 16MB～32GB 之间。

2.2.3 E^2PROM

E^2PROM（Electrically Erasable Programmable Read-Only Memory）主要用于在嵌入式系统中保存一些小量数据或特殊用途数据，通常使用串行通信接口和嵌入式处理器进行数据交互，如 I^2C 总线接口、SPI 总线接口和 1-wire 总线接口等，其容量通常在几百字节到几百万字节不等。

2.2.4 大容量存储系统

嵌入式系统中的大容量存储系统包括 SD 卡、U 盘和普通硬盘，通常用于保存大容量数据，和嵌入式处理器通过对应的接口芯片或时序进行数据交互。

2.3 嵌入式系统的外围器件

嵌入式系统的大部分功能都需要通过这些外围设备来实现，常见的外围设备包括：人体输入设备，如独立按键、行列扫描键盘、拨码开关等；显示设备，包括发光二极管、发光二极管、液晶显示屏等；驱动和执行设备，包括三极管、达林顿管、电动机、继电器、蜂鸣器等；通信接口设备，包括串口、网络接口、USB 口、无线设备等。

2.4 S3C2440 处理器和 GT2440 嵌入式开发板

从市场和实际应用普及度等情况出发，本章选择了 SumSung 公司生产的 ARM9 处理器 S3C2440 作为嵌入式系统的核心处理器，选择了 GT2440 作为嵌入式系统的开发板。

2.4.1 S3C2440 处理器的特点和内部资源

S3C2440 是 SumSung 公司开发的一款基于 ARM920T 内核和 0.18μm CMOS 工艺的 16/32 位 RISC 微处理器，适用于低成本、低功耗、高性能的手持设备或其他电子产品，其结构和各个模块的典型应用如图 2.10 所示。

S3C2440 中集成了以下一些通用的系统外设和接口，其硬件特点说明如下。

- 采用 1.2V 内核供电，1.8V/2.5V/3.3V 存储器供电，3.3V 外部 I/O 供电，具备 16KB 的 I-Cache 和 16KB 的 D-Cache/MMU 微处理器。
- 可以扩展外部存储控制器（SDRAM 控制和片选逻辑）。
- 内置了最大支持 4K 色 STN 和 256K 色 TFT 的 LCD 控制器，并且提供 1 通道 LCD 专用 DMA。
- 内置 4 通道 DMA 并有外部请求引脚。
- 内置 3 通道 UART，支持 IrDA1.0、64 字节 Tx FIFO 和 64 字节 Rx FIFO。
- 提供 2 通道 SPI 接口。
- 提供 1 通道 I^2C 总线接口。
- 提供 1 通道 I^2S 总线音频编解码器接口。
- 提供 AC’97 解码器接口。
- 兼容 SD 主接口协议 1.0 版和 MMC 卡协议 2.11 兼容版。
- 提供了两个 1.1 版本的 USB 主机端口和 1 个 USB 设备端口。
- 内置了 4 通道 PWM 定时器和 1 通道内部定时器/看门狗定时器。
- 提供了 8 通道 10 比特 ADC 和触摸屏接口。

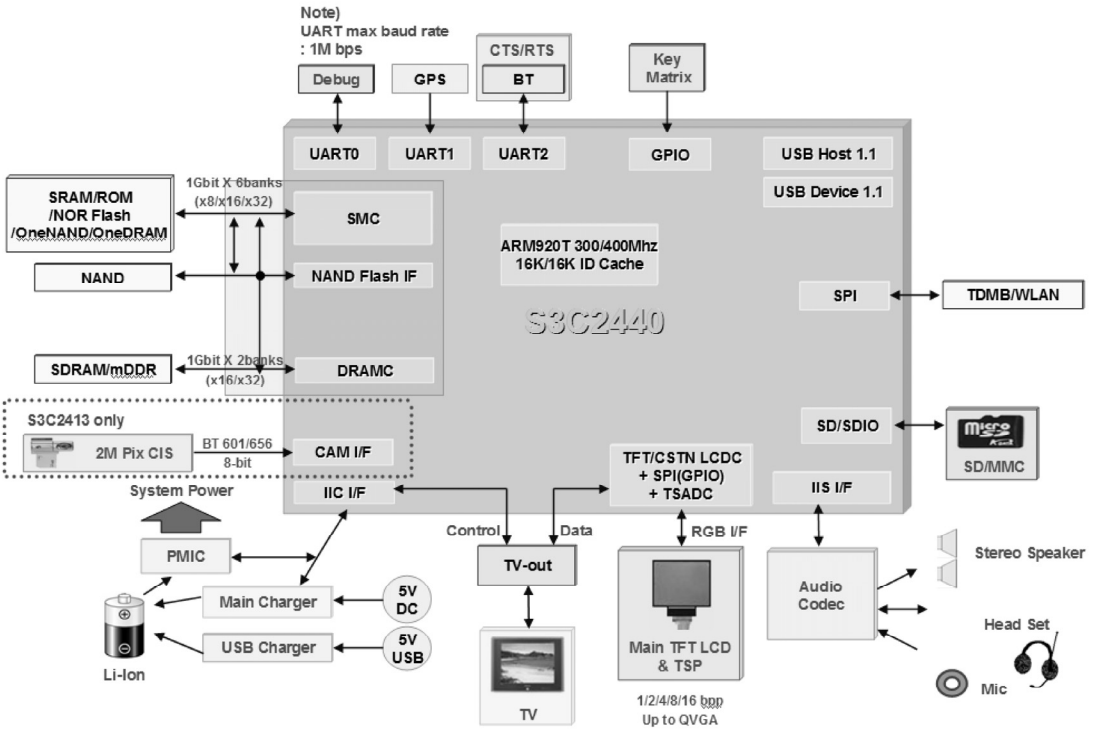


图 2.10　S3C2440 的硬件结构和典型应用

- 提供了具有日历功能的实时时钟（RTC）。
- 提供了摄像头接口，支持最大 4096×4096 像素。
- 提供了 130 个通用 I/O 口和 24 通道外部中断源。
- 具有普通、慢速、空闲和掉电模式。
- 具有 PLL 片上时钟发生器。
- 为手持设备和通用嵌入式应用提供片上集成系统解决方案。
- 采用了 16/32 位 RISC 体系结构和 ARM920T 内核强大的指令集。
- 采用 ARM920T 处理器内核，支持 ARM 调试体系结构。
- 采用了加强的 ARM 体系结构 MMU，可以支持 WinCE、EPOC 32 和 Linux 操作系统。
- 内置了指令高速存储缓冲器 I-Cache、数据高速存储缓冲器 D-Cache、写缓冲器和物理地址 TAG RAM 以减少主存带宽和响应性带来的影响。
- 内部高级微控制总线 AMBA 体系结构 AMBA2.0、AHB/APB。

1．S3C2440 的系统管理器

S3C2440 的系统管理器具有如下特点。

- 支持大/小端方式。
- 支持高速总线模式和异步总线模式。
- 支持大寻址空间，每 bank 128MB，总共 1GB。

- 支持可编程的每 bank 8/16/32 位数据总线带宽。
- 共有 8 个存储器 bank，其中 6 个适用于 ROM、SRAM，其他另外两个适用于 ROM、SRAM 和同步 DRAM。所有的存储器 bank 都具有可编程的操作周期，从 bank0～bank6 都采用固定的 bank 起始寻址，bank7 具有可编程的 bank 的起始地址和大小。
- 支持外部等待信号延长总线周期。
- 支持掉电时的 SDRAM 自刷新模式。
- 支持包括 NOR/NAND Flash、E^2PROM 等各种型号的 ROM 引导。

2．S3C2440 的 NAND Flash 启动引导

S3C2440 的 NAND Flash 具有如下特点。

- 支持从 NAND Flash 存储器的启动。
- 采用 4KB 内部缓冲器进行启动引导。
- 支持启动之后 NAND 存储器仍然作为外部存储器使用。
- 支持先进的 NAND Flash。

3．S3C2440 的 Cache 存储器

S3C2440 的 Cache 存储器具有以下特点。

- 64 项全相连模式，采用 I-Cache（16KB）和 D-Cache（16KB）。
- 每行 8 字长度，其中每行带有一个有效位和两个 dirty 位。
- 伪随机数或轮转循环替换算法位。
- 采用写穿式（write-through）或写回式（write-back）Cache 操作来更新主存储器。
- 写缓冲器可以保存 16 字的数据和 4 个地址。

4．S3C2440 的时钟和电源管理

S3C2440 拥有片上 MPLL 和 UPLL，采用 UPLL 产生操作 USB 主机/设备的时钟，MPLL 可以在 1.3V 下产生最大输出为 400MHz 的处理器工作时钟，其时钟和电源管理的主要特点说明如下。

- 通过软件可以有选择性地为每个功能模块提供时钟。
- 电源具有正常、慢速、空闲和掉电模式。
- 可以通过 EINT[15:0]或 RTC 报警中断来从掉电模式中唤醒处理器。

5．S3C2440 的中断

S3C2440 内部提供了 60 个中断源，包括 1 个看门狗定时器、5 个定时器、9 个 UARTs、24 个外部中断、4 个 DMA、2 个 RTC、2 个 ADC、1 个 I^2C 总线接口、2 个 SPI 总线接口，1 个 SDI、2 个 USB、1 个 LCD、1 个电池故障、1 个 NAND Flash、2 个 Camera 接口和 1 个 AC97 音频，其中断具有如下特点。

- 提供了电平/边沿触发模式的外部中断源。
- 提供了可编程的边沿/电平触发极性。
- 支持为紧急中断请求提供快速中断服务。

6．S3C2440 的 PWM 模块

S3C2440 提供了 4 通道 16 位具有 PWM 功能的定时器，1 通道 16 位内部定时器，可基

于 DMA 或中断工作，其具有如下特性。

- 提供了可编程的占空比周期，频率和极性。
- 能产生死区。
- 支持外部时钟源。

7．S3C2440 的实时时钟

S3C2440 的实时时钟提供了包括秒、分、时、日期，星期、月和年在内的完整时钟特性，具有如下特性。

- 采用 32.768kHz 工作。
- 提供报警中断。
- 提供节拍中断。

8．S3C2440 的通用 I/O 端口

S3C2440 提供了 24 个外部中断端口和 130 个多功能输入/输出端口。

9．S3C2440 的 DMA 控制器

S3C2440 内置了 4 通道的 DMA 控制器，具有以下特点。

- 支持存储器到存储器、I/O 到存储器、存储器到 I/O 和 I/O 到 I/O 的数据传输。
- 可以采用触发传输模式来加快传输速率。

10．S3C2440 的 LCD 控制器

S3C2440 的 LCD 控制器提供了对 STN LCD 和 TFT LCD 的显示支持。

S3C2440 支持 4 位双扫描、4 位单扫描、8 位单扫描共 3 种显示类型的 STN LCD 显示屏；支持单色模式、4 级、16 级灰度 STN LCD、256 色和 4096 色 STN LCD。其还支持多种不同尺寸的液晶屏，包括 640×480、320×240、160×160 等，其最大虚拟屏幕大小为 4MB，在 256 色模式下支持的最大虚拟屏可以达到 4096×1024、2048×2048、1024×4096 等。

S3C2440 还支持 TFT 格式的 LCD，其特点说明如下。

- 支持 1、2、4 或 8bbp（像素每位）调色显示。
- 支持 16、24bbp 无调色真彩显示 TFT。
- 在 24bbp 模式下支持最大 16M 色 TFT。
- 支持包括屏幕大小为 640×480、320×240、160×160 的实际屏幕，提供 4MB 的虚拟屏，在 64K 色彩模式下可以达到最大尺寸为 2048×1024 的虚拟屏。

11．S3C2440 的 UART

S3C2440 内置 3 通道 UART，可以基于 DMA 模式或中断模式工作，其支持 5 位、6 位、7 位或 8 位串行数据发送/接收支持外部时钟作为 UART 的运行时钟，具有以下特点。

- 可编程的波特率。
- 支持 IrDA1.0。
- 具有测试用的还回模式。
- 每个通道都具有内部 64 字节的发送 FIFO 和 64 字节的接收 FIFO。

12．S3C2440 的 A/D 转换和触摸屏接口

S3C2440 内置了一个 8 通道多路复用 ADC 模块，可以提供最大 500KSPS/10 位精度，

并且提供了一个内部 TFT 直接触摸屏接口。

13．S3C2440 的看门狗定时器

S3C2440 提供了 16 位的看门狗定时器，在定时器溢出时发生中断请求或系统复位。

14．S3C2440 的 I^2C 总线接口

S3C2440 提供了 1 通道多主 I^2C 总线接口，可进行串行、8 位、双向数据传输，标准模式下数据传输速度可达 100Kb/s，快速模式下可达到 400Kb/s。

15．S3C2440 的 I^2S 总线接口

S3C2440 提供了 1 通道音频 I^2S 总线接口，可基于 DMA 方式工作；其采用每通道 8/16 位串行数据传输，发送和接收具备 128 字节（64 字节加 64 字节）FIFO 缓冲，支持 I^2S 格式和 MSB-justified 数据格式。

16．S3C2440 的 AC97 音频解码器接口

S3C2440 支持 16 位采样，支持如下功能：

- 1-ch 立体声 PCM 输入；
- 1-ch 立体声 PCM 输出；
- 1-ch MIC 输入。

17．S3C2440 的 USB 总线接口

S3C2440 提供了遵从 OHCI Rev.1.0 标准、兼容 USB version 1.1 标准的两个 USB 主设备接口和 1 个兼容 USB version 1.1 标准，具有 5 个 Endpoint 的 USB 从设备接口。

18．S3C2440 的 SPI 总线接口

S3C2440 提供了兼容 2.11 版 SPI 协议的 2 通道 SPI 接口，发送和接收具有 2×8 位的移位寄存器，可以基于 DMA 或中断模式工作。

19．S3C2440 的摄像头接口

S3C2440 内置了 ITU-R BT 601/656 8-bit 模式的摄像头接口，其特点说明如下：

- 具有 DZI（数字变焦）能力；
- 具有极性可编程视频同步信号；
- 最大值支持 4096 ×4096 像素输入，支持 2048×2048 像素输入缩放；
- 支持 *X* 轴、*Y* 轴和 180° 镜头旋转；
- 摄像头输出格式为 16/24-bit 的 RGB 与 YCBCR 4:2:0/4:2:2 格式。

20．S3C2440 的 SD 卡接口

S3C2440 提供了兼容 2.11 版 MMC 卡协议的 SD 卡接口，提供了正常、中断和 DMA 数据传输模式，支持字节、半字节传输，其具有如下特点：

- 只支持字传输的 DMA burst4 接入支持；
- 兼容 SD 存储卡协议 1.0 版；
- 兼容 SDIO 卡协议 1.0 版；
- 发送和接收具有 64 字节 FIFO。

21．S3C2440 的工作电压、操作频率和封装

S3C2440 采用了 289-FBGA 封装，其 Pclk 最高可达 68MHz，Hclk 最高可达 136MHz，Fclk 最高可以达到 400MHz。

在不同的工作频率下，S3C2440 需要不同的核心工作电压，300MHz 时工作电压为 1.2V；400MHz 时工作电压为 1.3V。

S3C2440 的 I/O 端口为 3.3V 电平，其内存支持 1.8V、2.5V、3.0V 和 3.3V 工作电压。

2.4.2　S3C2440 处理器的内部结构和工作模式

S3C2440 是采用 ARMv4 架构的 ARM920T 处理器，到其内核包括以下几部分：

- ARM9TDMI（32 位 RISC）处理器；
- 数据缓存器（Data Cache）；
- 指令缓存器（Instruction Cache）；
- 指令存储管理单元（Instruction MMU）；
- 数据存储管理单元（Data MMU）；
- 写缓冲（Write Buffer）和回写存储单元（Write Back PA TAG RAM）。

以上部件通过 AMBA 总线（AMBA Bus）相互传输数据以实现指令和数据的并行处理，除此之外，内核还包括四个与外界进行数据交换的接口：

- 总线接口（AMBA Bus Interface）；
- 扩展协处理器接口（External Coprocessor Interface）；
- 跟踪接口（Trace Interface）；
- JTAG 接口。

这些接口可以连接 DMA 控制器、UART、USB、中断控制器和电源管理器等。核心通过与外围部件共同工作完成整个嵌入式系统的正常数据处理任务。

1．S3C2440 的工作模式

S3C2440 有 7 种运行模式，说明如下。

- 用户模式（User，usr）：正常程序执行时 S3C2440 所处的状态。
- 快速中断模式（FIQ，fiq）：用于快速的数据传输和通道处理。
- 外部中断模式（IRQ，irq）：用于通常的中断处理。
- 特权模式（Supervisor，sve）：供操作系统使用的一种保护模式。
- 数据访问中止模式（Abort，abt）：当数据或指令预期终止时进入该模式，用于虚拟存储及存储保护。
- 未定义指令终止模式（Undefined，und）：用于支持硬件协处理器软件仿真。
- 系统模式（System，sys）：用于运行特权级的操作系统任务。

通常情况下，应用程序运行在用户模式下，这时应用程序不能访问一些受操作系统保护的系统资源，同时应用程序也不能直接进行处理器模式的切换。

当应用程序发生异常中断时，处理器进入相应的异常模式。在每一种异常模式中都有一组属于自己的寄存器，供相应的异常处理程序使用，这样可以保证异常模式时用户程序下的寄存器值不被破坏。

系统模式属于特权模式，它和用户模式具有完全一样的寄存器，在该模式下，可以访问所有的系统资源，也可以直接进行处理器模式切换。注意，从用户模式进入到系统模式，并不是通过异常过程进入的。

2．S3C2440 的寄存器

S3C2440 处理器共有 37 个寄存器，其中有 31 个通用寄存器、6 个状态寄存器，这些寄存器都是 32 位的。

S3C2440 处理器运行在每一种模式下时，都会使用属于自己的一组寄存器组，通常包括 15 个通用寄存器（R0～R14）、一个或两个状态寄存器及程序计数器（PC）。每一种模式下的寄存器组是部分重叠的，图 2.11 列出了各处理器模式下可见的寄存器情况。

图 2.11　各种处理器模式下的寄存器

通用寄存器中 R0～R7 是所有处理器模式公用的一组寄存器，也就是说，在从一种模式切换到另一种模式时，必须保存它们的值。R8～R14 为备份寄存器，其中对于 R8～R12 来说，每一个寄存器对应两个不同的物理寄存器，R13 和 R14 对应 6 个不同的物理寄存器，其中 R13 通常用作堆栈指针，采用下面的记号来区分各个物理寄存器：

```
R13_<MODE>
```

其中<MODE>通常可以使用下列几个值：usr、svc、abt、und、irq 及 fiq。

R14 寄存器有两个特殊的作用：

- 用户模式下，R14 用作链接寄存器（LR），存放子程序被调用时的返回地址；

● 异常处理模式下，R14 用来保存异常的返回地址。

R15 为程序计数器，又记作 PC。由于 ARM 采用了流水线机制，因此 PC 的值为当前指令地址的值加 8 字节，也就是说，PC 指向当前指令的下两条指令的地址。

在 ARM 处理器中，程序状态寄存器用来保存程序执行时的各种状态值，包括条件标志位、中断禁止位、当前处理器模式标志和其他一些位。程序状态寄存器分为 CPSR 和 SPSR 两种类型。在任何一种处理器模式下，都会有一个公用的 CPSR，另外异常模式下还会有一个专用的 SPSR（备份程序状态寄存器）。当异常发生时，这个寄存器用于存放当前程序状态寄存器的内容，当退出异常处理时，再把 SPSR 中的值恢复到 CPSR 中。CPSR 和 SPSR 格式相同，如图 2.12 所示。

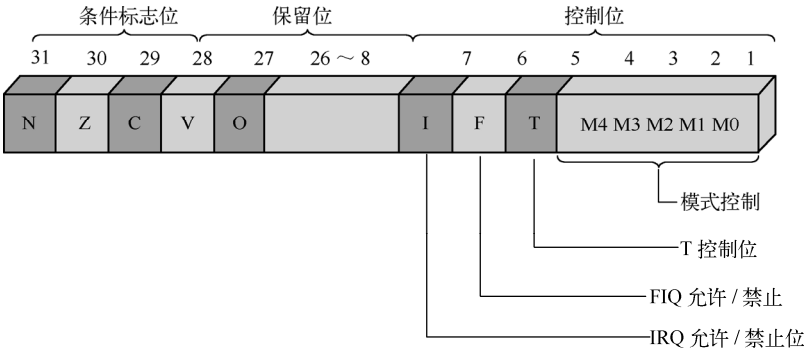


图 2.12　CPSR 和 SPSR 的格式

3．S3C2440 的异常处理

S3C2440 的异常是 ARM 体系结构中的异常，其与 8 位/16 位体系结构的中断有很大的相似之处，但异常与中断的概念并不完全等同。在 ARM 体系架构中，将正常的程序执行流程发生暂时的停止的情况都称为异常，中断不过是其中最为常见的一种，而且被区分为外部普通中断（IRQ）和外部快速中断（FIQ）两种。在处理异常之前，当前处理器的状态必须保留，这样当异常处理完成之后，被异常所“打断的”程序可以继续执行。ARM 处理器允许多个异常同时发生，它们将会按固定的优先级进行处理。

中断是外界和嵌入式系统交换信息的最重要的一种方式，中断处理也是 ARM 编程模型中需要详细了解的一部分知识。理解异常处理是理解 ARM 体系结构的一个重要途径，因为异常处理中涉及相当多的 ARM 体系结构知识。

S3C2440 所支持的异常及其对应的具体含义如表 2.11 所示。

表 2.11　S3C2440 所支持的异常

异 常 种 类	说　　明
复位	当处理器的复位电平有效时，产生复位异常，程序跳转到复位异常处理程序处执行
未定义指令	当处理器或协处理器遇到不能处理的指令时，产生未定义指令异常。可使用该异常机制进行软件仿真
软件中断	该异常由执行 SWI 指令产生，可用于用户模式下的程序调用特权操作指令。可使用该异常机制实现系统功能调用

续表

异 常 种 类	说　　明
指令预取中止	若处理器预取指令的地址不存在或该地址不允许当前指令访问，存储器会向处理器发出中止信号，但当预取的指令被执行时，才会产生指令预取中止异常
数据中止	若处理器数据访问指令的地址不存在或该地址不允许当前指令访问时，产生数据中止异常
IRQ（外部中断请求）	当处理器的外部中断请求引脚有效且 CPSR 中的 I 位为 0 时，产生 IRQ 异常。系统的外设可通过该异常请求中断服务
FIQ（快速中断请求）	当处理器的快速中断请求引脚有效且 CPSR 中的 F 位为 0 时，产生 FIQ 异常

由表 2.11 可以看出，S3C2440 实际上是将可能遇到的各类“异常”情况在体系架构层次上做了比较详细的划分。在实际运行中，如果某种异常出现，ARM 内核可以由硬件来判断异常的类型，然后自动跳转到对应的某个特定地址上（这些地址位于从某个地址开始的一段连续内存上），从这个特定地址再次跳转到对应的异常处理函数中去，从而进行快速处理。在异常处理的细节过程中，涉及寄存器值的保存和恢复、ARM 内核工作模式的切换、工作模式堆栈的维护等一系列 ARM 底层知识。

图 2.13 所示的异常向量表是由一组跳转指令构成的连续地址上的指令集合，“表”中指定了各异常模式及其处理程序的对应关系，它通常存放在存储器地址的低端起始地址上（有的内核型号支持将该表放置在特定的高地址）。在 ARM 体系中，异常向量表的大小为 32 字节。其中，每个异常占据 4 字节，保留了 4 字节空间。每 4 字节空间存放一个跳转指令或一个向 PC 寄存器中赋值的数据访问指令。通过这两种指令，程序将跳转到相应的异常处理程序处执行，形式通常如下：

```
B    FIQHandler;                    处理函数名称，代表其存放地址
```

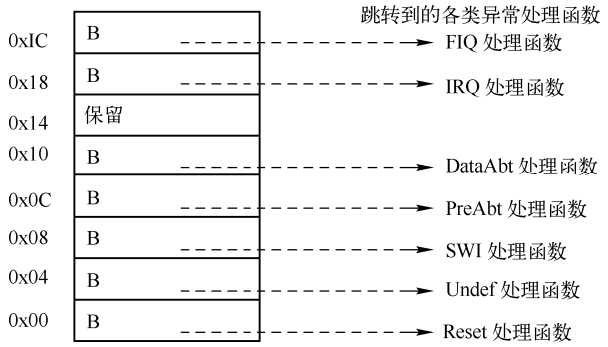


图 2.13　ARM 异常向量表

需要说明的是，当 ARM 内核工作在 16 位数据宽度的“Thumb”模式下时，如果发生异常，内核模式会自动切换回 ARM 工作模式，然后才跳转到对应的地址上去，否则对每次取指令到底是取 32 位数据还是 16 位数据就会产生疑惑。

可以把“异常向量表”理解为硬件对于发生某种特定情况时，用软件接管处理的入口地址列表。这张表占据了一块特殊的地址空间，使得硬件只要判断出异常类型，然后跳到对应地址就可以了，接下来的工作交由程序员来完成。异常向量表的内容和建立过程也需要程

序员设计好，也就是说，这些特殊的地址上原本并没有这张向量表，需要复制或写入。最常见的建立异常向量表的方法是利用 ARM 公司的软件工具（ADS/MDK）所提供的“Scatter Loading”（分散装载）的方式，可以很轻松地实现异常向量表在某个地址的复制。

当多个异常同时发生时，系统根据固定的优先级决定异常的处理次序。当然有些异常是不可能同时发生的，如指令预取中止异常和软件中断（SWI）异常是由同一条指令的执行触发的，它们是不可能同时发生的。处理器执行某个特定的异常的过程称为处理器处于特定的异常模式。各异常的向量地址及异常的处理优先级如表 2.12 所示。

表 2.12　异常向量的优先级

向 量 地 址	异 常 说 明	异常发生后内核进入的模式	异常的优先级（6 最低）
0x0000	复位	管理模式	1
0x0004	未定义指令	未定义模式	6
0x0008	软件中断	管理模式	6
0x000C	中止（预取指令）	中止模式	5
0x0010	中止（数据）	中止模式	2
0x0014	保留	保留	保留
0x0018	IRQ	IRQ 模式	4
0x001C	FIQ	FIQ 模式	3

注意异常类型（模式）和工作模式之间的区别和对应关系，工作模式是寄存器分组的依据，每种工作模式都有自己特定的寄存器和自己的堆栈，而异常类型是 ARM 体系结构对各种异常情况的细分，发生一种异常时会进入唯一对应的工作模式对异常进行处理，这一点需要在学习的时候加以注意。

当一个异常出现以后，S3C2440 通常会执行以下几步操作。

（1）将下一条指令的地址存入相应的链接寄存器 LR，以便程序在处理异常返回时能从正确的位置重新开始执行。

（2）将 CPSR 复制到相应的 SPSR 中。

（3）根据发生的异常类型，强制设置 CPSR 的运行模式位，此后 ARM 内核进入对应的异常工作模式，对异常情况进行处理；屏蔽中断，暂时禁止新的中断发生；处理器原先处于 Thumb 状态，之后自动切换到 ARM 状态，使得在接下来的软件处理中总能以字符为单位取到新执行的指令，而不至于因为状态未知而导致取值宽度不确定。

（4）强制 PC 值为相关的异常向量地址，在该地址上，存放着跳转到软件处理异常函数的入口地址，从而跳转到相应的异常处理程序。硬件对异常的自动响应到此结束，后继由软件接管，处理异常并返回。

可以说，异常处理是硬件和软件协同工作完成对外界信号的反应，以上 4 个步骤由硬件完成，实现了对异常发生时刻的部分现场保护，其他需要保护的寄存器值则由后继的软件处理进行保存。原则上说，需要保护的寄存器就是在后面异常处理中用到的寄存器，它们原先的值需要保存到存储器中。通常采用“栈”的方式，用压栈汇编指令依次保存到存储器中去。由于不知道需要保存哪些寄存器，在一般的现场保护过程中，采用保护所有要处理的异

常模式对应的通用寄存器，如果涉及工作模式的再次切换或重入，那么状态寄存器、连接寄存器也要保护。

软件部分还要对异常的具体情况进行处理（如中断处理），处理完成以后进行中断返回，即现场恢复，回到异常发生时刻的状态。

异常处理完毕之后，S3C2440 微处理器会执行以下几步操作，以从异常返回。

（1）将连接寄存器 LR 的值减去相应的偏移量后送到 PC 中。

（2）将 SPSR 复制到 CPSR 中。

（3）若在进入异常处理时设置了中断禁止位，要在此清除。

特别需要注意的是，各种异常返回时需要减去的偏移量是不同的，需要根据不同的异常种类加以区别。可以认为应用程序总是从复位异常处理程序开始执行的，因此复位异常处理程序不需要返回。

当一个异常处理返回时，一共有 3 件事情需要处理。

- 通用寄存器的恢复。
- 状态寄存器的恢复。
- PC 指针的恢复。

通用寄存器的恢复采用一般的堆栈操作指令，而 PC 和 CPSR 的恢复可以通过一条指令来实现，下面是三个例子：

```
MOVS pc，lr 或 SUBS pc，lr，#4 或 LDMFD sp!，{pc}^
```

这几条指令都是普通的数据处理指令，特殊之处就是把 PC 寄存器作为目标寄存器，并且带了特殊的后缀“S”或“^”。在特权模式下，“S”或“^”的作用就是使指令在执行时，同时完成从 SPSR 到 CPSR 的复制，达到恢复状态寄存器的目的。

异常返回时另一个非常重要的问题是返回地址的确定。在前面章节中提到进入异常时处理器会有一个保存 LR 的动作，但是该保存值并不一定是正确中断的返回地址。下面以一个简单的三级流水线的情况下，指令执行流水状态图来对此加以说明，如图 2.14 所示。

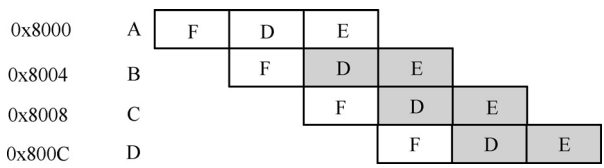


图 2.14　状态下三级指令流水线执行示例

系统运行时，异常可能会随时发生，为保证在 S3C2440 发生异常时不至于影响程序的运行，在应用程序的设计中，必须进行异常处理，采用的方式是在异常向量表中的特定位置放置一条跳转指令，跳转到异常处理程序，当 S3C2440 发生异常时，程序计数器 PC 会被强制设置为对应的异常向量地址，从而跳转到软件人员编写的异常处理函数，当异常处理完成以后，返回主程序继续执行。

下面以最常见的中断为例，详细地看一下从中断发生到处理、返回的具体过程。假设在 A+4 地址处发生了一次中断异常，那么 S3C2440 会按照图 2.15 所示的顺序进行处理。

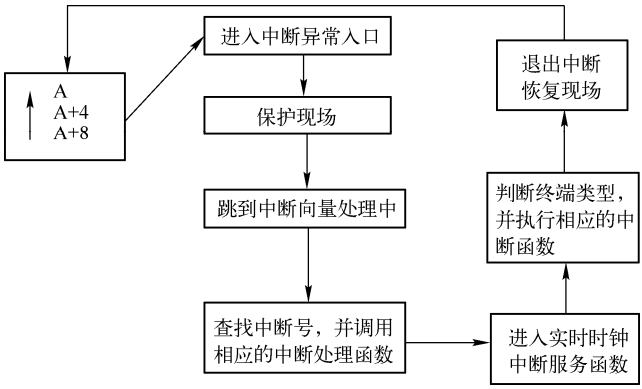


图 2.15　S3C2440 中断处理流程

4．S3C2440 的存储器组织

S3C2440 以虚拟地址的方式对存储器进行组织。与其他架构如 x86 一样，ARM 体系架构的处理器也存在大小端的问题，不同的寻址方式会有不同的结果。大小端寻址方式可以通过硬件引脚由用户控制，ARM9 及其以上架构都由 MMU 单元来管理存储器。

ARM 储存器的组织主要有两大类型，分别为小端格式和大端格式，也称为小端次序（Little Endian）的字节序和大端次序（Big Endian）的字节序（Byte Order）。两种储存类的区别在于一个 32 位的数据存放到储存器中时，高位字节是放在高地址还是放在低地址，如图 2.16 所示。

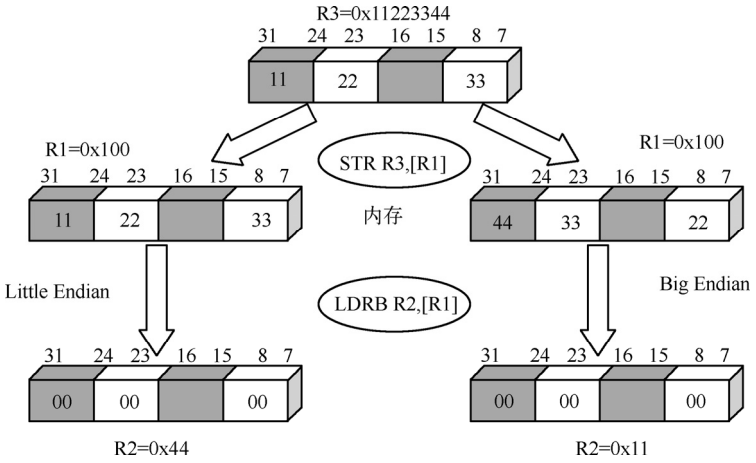


图 2.16　ARM 大小端存储系统

当用户储存一个 32 位数据 0x11223344 到地址 0x100 上时，如果是小端字节序，那么存储在 0x100 地址上的字节应该为 0x44 这个数据、0x101 为 0x33、0x102 为 0x22、0x103 为 0x11，也就是高位数据放高位地址，低位数据放低位地址。而大端字节序则正好相反。

字节序确定了储存的基本方式，特别是在对半字和字节为宽度的数据操作时，需要特别注意。如图 2.16 中所示，使用 LDRB（以字节为单位装载寄存器）时，不同的字节序所

获得的数据结果是不一样的。

注意：S3C2440 默认的字节序方式为小端字节序。

不同的嵌入式应用系统中，其存储体系也会差别很大。例如在 ARM7TDMI 核中，存储体系使用最简单的平板式地址映射机制。在该方式下，对地址空间的分配是固定的，系统使用物理地址，就像单片机系统一样，这种方式会带来以下几个问题。

- 程序员必须自己管理物理内存的分配、使用和回收，增加了编程的难度。
- 应用程序出错可能会带来整个内核的崩溃。

为此，在很多 ARM 微处理器内核中，都使用虚拟内存映射机制，整个内存由内存管理单元（即 MMU）进行管理，整个系统使用虚拟地址，再由 MMU 将其映射为实际的物理地址，图 2.17 为 MMU 转换示意图。

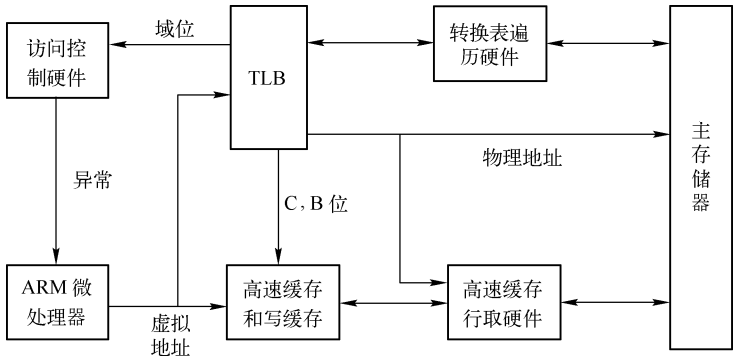


图 2.17　高速缓存的 MMU 存储器系统

这种映射机制对于嵌入式系统非常重要，在通常情况下，MMU 主要完成以下工作。

- 虚拟存储空间到物理存储空间的映射。在 ARM 中，无论是物理地址还是虚拟地址都使用分页机制，即把空间分为一个个大小固定的块，每一块称为一页。物理空间的页和虚拟地址的页大小相同。
- 存储器访问权限的控制。

2.4.3　GT2440 嵌入式开发板的硬件资源

本章采用的嵌入式开发板是基于三星 S3C2440 的 GT2440 平台，其硬件实物外观如图 2.18 所示。目前市面上的 S3C2440 开发板大都基于该平台开发，尽管配置与芯片的选用上稍有差别，但大体性能并没有多大的改变。因此基于嵌入式 ARM 的 Linux 开发在该平台上几乎通用，并不局限于某一种或某一款型号的开发板。用户如果想在不同厂商的目标板上移植，只需修改有限的几个参数，就能很方便地实现，这也正是嵌入式的优势之一。

GT2440 平台的硬件的主要特性如下。

- 采用的 CPU 处理器是 Samsung S3C2440 ARM9 系列，其主频为 400MHz，最高可达 533MHz。
- 板载 32bit 数据总线的 64M 时钟频率为 100MHz 的 SDRAM。

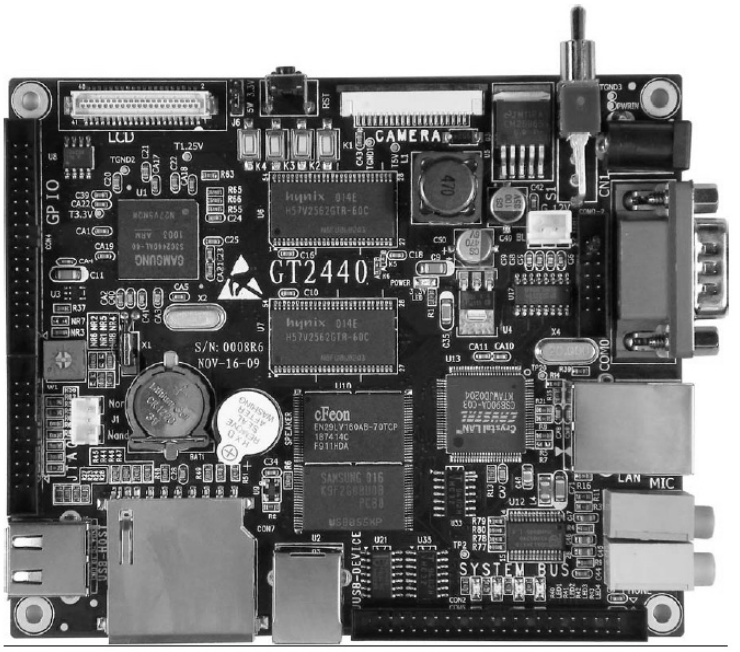


图 2.18　GT2440 平台的实物示意

- 板载 256M Nand Flash 和 2M Nor Flash，均为掉电非易失，且后者已经安装 BIOS。
- 供电采用专业 1.25V 核心电压供电，能耗较低。

GT2440 的接口和硬件资源说明如下，其对应实物的具体分布如图 2.19 所示。

- 提供了 1 个采用 CS8900 网络芯片的 10MB 以太网 RJ-45 接口。
- 提供了 3 个串口，其中两个需要自行引出。
- 提供了 1 个 USB HOST 接口和 1 个 USB Slave B 型接口。
- 提供了 1 个 SD 卡接口。
- 提供了 1 路麦克输入和 1 路立体音频输出接口。
- 提供了 1 个 2.0mm 间距 20 针标准 JTAG 接口。
- 提供了 4 个用户可编程发光二极管和 4 个用户可编程独立按键。
- 提供了 1 个 PWM 驱动蜂鸣器。
- 提供了 1 个用于 A/D 转换模块测试的可调电阻。
- 提供了 1 个 I^2C 总线接口的 E^2PROM 芯片 AT24C08。
- 提供了 1 个 20pin 摄像头接口。
- 提供了带板载电池的实时时钟。
- 提供了带电源开关和指示灯的 12V 电源接口。

GT2440 目标板支持两种启动模式：一种是从 Nand Flash 启动；另一种是从 Nor Flash 启动。它们可以使用跳线进行切换，在这两种启动模式下 GT2440 的存储器地址空间分布是不同的，如图 2.20 所示，其中左边是 nGCS0 片选的 Nor Flash 启动模式下存储器空间分布，右边是 Nand Flash 启动模式下的存储分布。

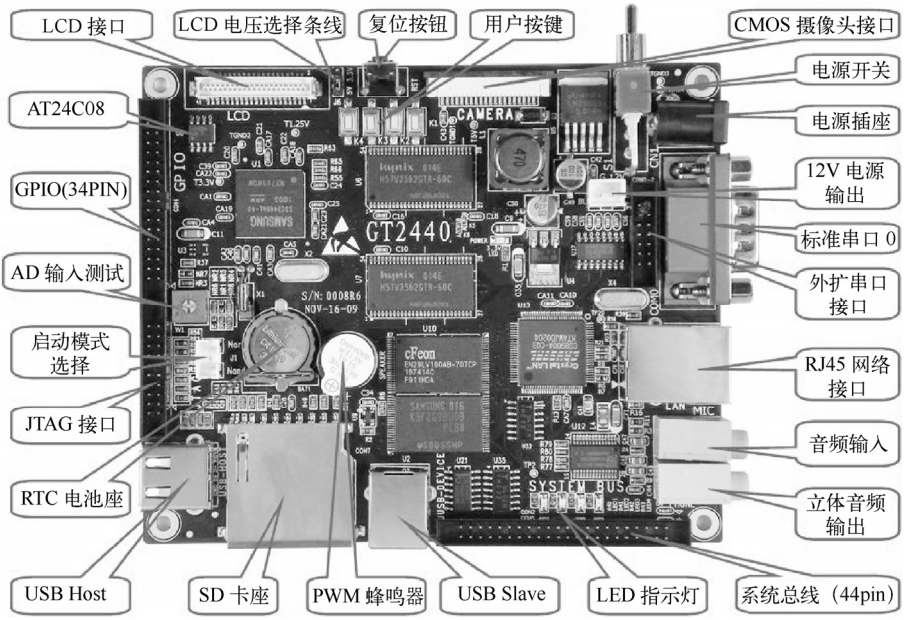


图 2.19　GT2440 的接口和硬件资源分布

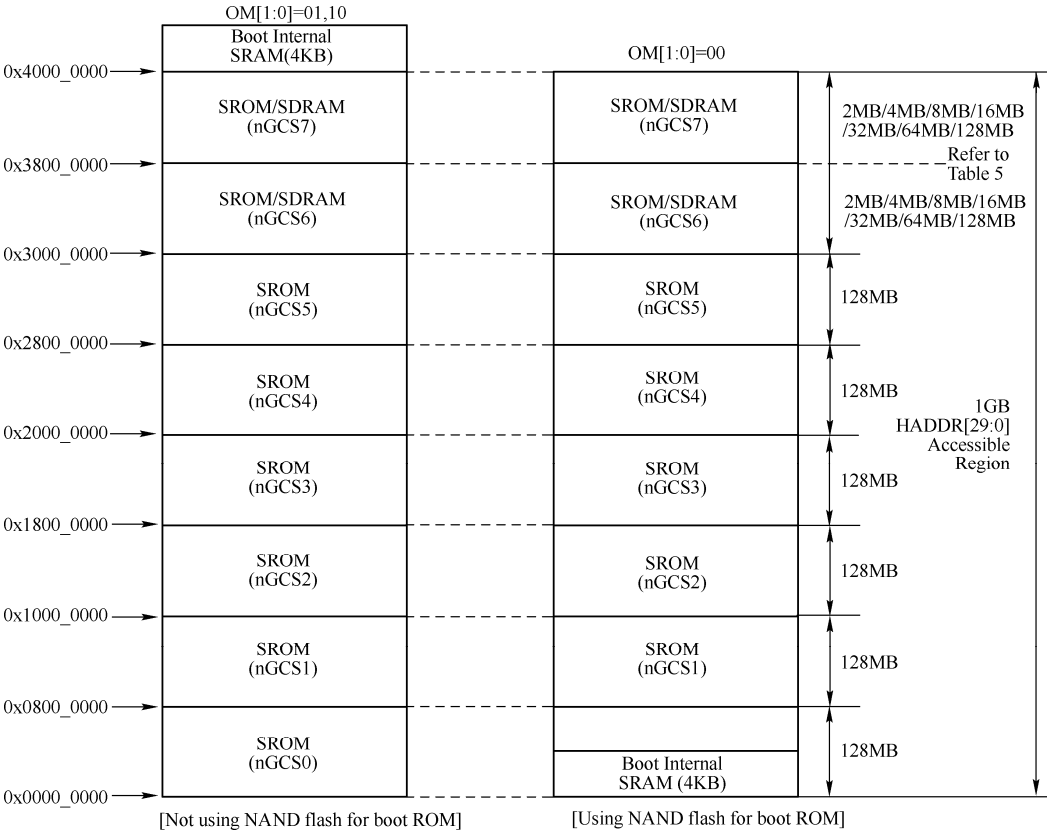


图 2.20　GT2440 的启动模式和存储器空间分布

第3章

嵌入式系统的 Linux 操作系统

Linux 是一套免费使用和自由传播的类 UNIX 操作系统，已发展成为现今世界上最流行的操作系统之一，其同时也是嵌入式系统中最常用的操作系统之一，具有代码公开性、可裁剪性、自由度高、免费等一系列优点，本章将详细介绍其基础知识，涉及的内容如下：

- Linux 的发展历史、特点、组成结构和常见的发行版；
- Linux 的人机交互方法，包括图形界面和 Shell；
- Linux 中常见的操作命令说明。

3.1 Linux 操作系统基础

3.1.1 Linux 操作系统的发展

Linux 操作系统从 1991 年问世到现在已经有 20 多年的历史，其从一个简单架构的系统内核发展到现在结构完整、功能丰富的多版本用户系统。Linux 操作系统是一种类 UNIX 操作系统，它最早是由芬兰人 Linus Torvalds 设计的。目前 Linux 操作系统可以运行在 x86、MIPS、m68K、M32R、Power PC、s390、ARM 等类型的计算机上。从功能来看，它既可以作为普通的桌面操作系统，也可以作为中小型的网络操作系统，甚至作为大型网络的操作系统。

Linux 的内核是系统的心脏，内核包括了几百万行代码，是运行程序和管理硬件设备的核心程序。没有内核，就不能运行程序，但内核不是操作系统的全部。Linux 初学者常会把内核版本与发行套件版本弄混了， 实际上内核版本指的是在 Linus 领导下的开发小组开发出的系统内核的版本号。Linux 的每个内核版本使用形式为 x.y.zz-www 的一组数字来表示。其中，x.y 为 Linux 的主版本号，zz 为次版本号，www 代表发行号（注意，它与发行版本号无关）。当内核功能有一个飞跃时，主版本号升级，如 Kernel2.2、2.4、2.6 等。内核增加了少量补丁时，常常会升级次版本号，如 Kernle2.6.15、2.6.20 等。当然还有更复杂的版本号系统，如 2.6.20-32 等。通常 *y* 若为奇数，表示此版本为测试版，系统会有较多漏洞，主要用途是提供给用户测试。随着每一次系统的小漏洞的修正，zz 会增加。

注意：截止到 2014 年 1 月 15 日，Linux 的稳定版本是 3.12.8，测试版本是 3.13-rc8。

3.1.2 Linux 操作系统的特点

Linux 具有 UNIX 的所有特性，并且发展出来自身的一些特性。

- 具有良好的开放性：开放性是指系统遵循世界标准规范，特别是遵循开放系统互连（OSI）国际标准。凡遵循国际标准所开发的硬件和软件，都能彼此兼容，可方便地实现互连。
- 支持多用户：多用户是指系统资源可以被不同的用户各自拥有并使用，即使每个用户对自己的资源（如文件、设备）有特定权限，也互不影响，Linux 和 UNIX 都具有多用户特性。
- 支持多任务：多任务是现代计算机的一个最主要特点，它是指计算机同时执行多个程序，而且各个程序的运行相互独立。Linux 操作系统调试每一个进程平等地访问处理器。由于处理器的处理速度非常快，其结果是启动的应用程序看起来好像是在并行运行。事实上，从处理器执行的一个应用程序中的一组指令到 Linux 调试处理器，与再次运行这个程序之间只有很短的时间延迟，用户是感觉不出来的。
- 提供友好的用户界面：Linux 向用户提供了用户界面和系统调用界面。Linux 的传统用户界面基于文本的命令行界面，即 Shell。它既可以联机使用，又可以存储在文件上脱机使用。Shell 有很强的程序设计能力，用户可方便地用它编写程序，从而为用户扩充系统功能提供了更高级的手段。Linux 还提供了图形用户界面，它利用鼠标、菜单和窗口等设施，给用户呈现一个直观、易操作、交互性强的友好图形化界面。
- 具有良好的设备独立性：设备独立性是指操作系统把所有外部设备统一当作文件来看，只要安装它们的驱动程序，任何用户都可以像使用文件那样操作并使用这些设备，而不必知道它们的具体存在形式。设备独立性的关键在于内核的适应能力，其他的操作系统只允许一定数量或一定种类的外部设备连接，因为每一个设备都是通过其与内核的专用连接独立地进行访问的。Linux 是具有设备独立的操作系统，它的内核具有高度的适应能力，随着更多程序员加入 Linux 编程，相信以后会有更多硬件设备加入到各种 Linux 内核和发行版本中。
- 提供了丰富的网络功能：完善的内置网络是 Linux 的一大特点，Linux 在通信和网络功能方面优于其他操作系统。其他操作系统不包含如此紧密的内核结合在一起的连接网络能力，也没有内置这些连网特性的灵活性。而 Linux 为用户提供了完善的、强大的网络功能。Linux 免费提供了大量支持 Internet 的软件，Internet 是在 UNIX 领域中建立并发展起来的，在这方面使用 Linux 是相当方便的，用户能用 Linux 与世界上其他人通过 Internet 网络进行通信。
- 支持文件传输和远程访问：用户能通过一些 Linux 命令完成内部信息或文件的传输；同时 Linux 为系统管理员和技术人员提供了访问其他系统的窗口。通过这种远程访问的功能，一位技术人员能够有效地为多个系统服务，即使那些系统位于很远的地方。
- 具有可靠的安全性：Linux 操作系统采取了许多安全措施，包括对读、写操作进行权限控制，带保护的子系统，审计跟踪和内核授权，这为用户提供了必要的安全保障。

- 具有良好的可移植性：可移植性是指将操作系统从一个平台转移到另一个平台，使它仍然能按其自身的方式运行的能力。Linux 是一款具有良好可移植性的操作系统，能够在微型计算机到大型计算机的任何环境中和平台上运行。该特性为 Linux 操作系统的不同计算机平台与其他任何机器进行准确而有效的通信提供了保障，不需要另外增加特殊的通信接口。X-Windows 系统是用于 UNIX 机器的一个图形系统，该系统拥有强大的界面系统，并支持许多应用程序，是业界标准界面。
- 提供了内存保护模式：Linux 使用处理器的内存保护模式来避免进程访问分配给系统内核或其他进程的内存。对于系统安全来说，这是一个主要的贡献，一个不正确的程序因不能再使用系统而崩溃（在理论上）。
- 提供了共享程序库：共享程序库是一个程序工作所需要的例程的集合，有许多同时被多于一个进程使用的标准库，因此使用户觉得需要将这些库的程序载入内存一次，而不是一个进程一次，通过共享程序库使这些成为可能，因为这些程序库只有当进程运行的时候才被载入，所以它们被称为动态链接库。

3.1.3　Linux 操作系统的组成结构

Linux 既是一个操作系统的名称，也是一个操作系统内核的名称。一个完整的 Linux 操作系统由 Linux 内核、命令解释器、文件系统和实用工具组成，如图 3.1 所示。

1．Linux 内核

内核（Kernel）是 Linux 操作系统的心脏，是运行程序和管理像磁盘和打印机等硬件设备的核心程序，其从用户那里接受命令并把命令送给内核去执行。

Linux 内核主要由五个子系统组成：进程调度、内存管理、虚拟文件系统、网络接口和进程间通信，如图 3.2 所示。

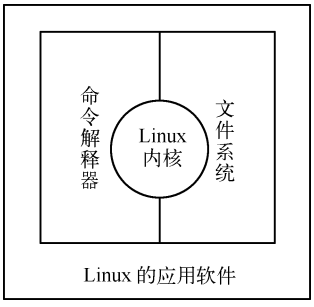


图 3.1　Linux 的组织结构

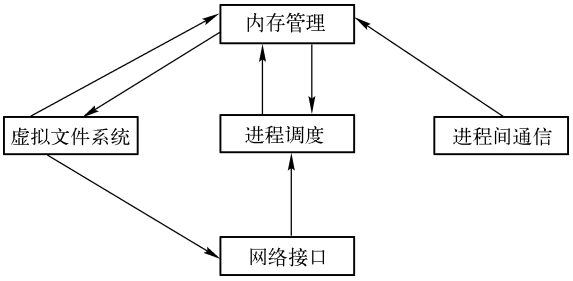


图 3.2　Linux 的内核结构

进程调度（SCHED）控制进程对处理器的访问。当需要选择下一个进程运行时，由调度程序选择最值得运行的进程。可运行进程实际上是仅等待处理器资源的进程，如果某个进程在等待其他资源，则该进程是不可运行进程。Linux 使用了比较简单的基于优先级的进程调度算法选择新的进程。

内存管理（MM）允许多个进程安全地共享主内存区域。Linux 的内存管理支持虚拟内存，即在计算机中运行的程序，其代码、数据、堆栈的总量可以超过实际内存的大小，操作

系统只是把当前使用的程序块保留在内存中，其余的程序块则保留在磁盘中。必要时，操作系统负责在磁盘和内存间交换程序块。内存管理从逻辑上分为硬件无关部分和硬件有关部分。硬件无关部分提供了进程的映射和逻辑内存的对换；硬件相关部分为内存管理硬件提供了虚拟接口。

虚拟文件系统（Virtual File System，VFS）隐藏了各种硬件的具体细节，为所有的设备提供了统一的接口，VFS 提供了多达数十种不同的文件系统。虚拟文件系统可以分为逻辑文件系统和设备驱动程序。逻辑文件系统指 Linux 所支持的文件系统，如 ext2、Fat 等，设备驱动程序指为每一种硬件控制器所编写的设备驱动程序模块。

网络接口（NET）提供了对各种网络标准的存取和各种网络硬件的支持。网络接口可分为网络协议和网络驱动程序。网络协议部分负责实现每一种可能的网络传输协议。网络设备驱动程序负责与硬件设备通信，每一种可能的硬件设备都有相应的设备驱动程序。

进程间通信（IPC） 支持进程间的各种通信机制。处于中心位置的进程调度，所有其他的子系统都依赖它，因为每个子系统都需要挂起或恢复进程。一般情况下，当一个进程等待硬件操作完成时，它被挂起；当操作真正完成时，进程被恢复执行。例如，当一个进程通过网络发送一条消息时，网络接口需要挂起发送进程，直到硬件成功地完成消息的发送，当消息被成功发送出去以后，网络接口给进程返回一个代码，表示操作的成功或失败。其他子系统以相似的理由依赖于进程调度。

2．命令解释器

命令解释器（Shell）是系统的用户界面，其提供了用户与内核进行交互操作的一种接口。它接收用户输入的命令并把它送入内核去执行。

命令解释器解释由用户输入的命令并把它们送到内核。不仅如此，Shell 有自己的编程语言用于对命令的编辑，它允许用户编写由 Shell 命令组成的程序。Shell 编程语言具有普通编程语言的很多特点，如它也有循环结构和分支控制结构等，用这种编程语言编写的 Shell 程序与其他应用程序具有同样的效果。

除了 Shell，Linux 同样提供了像 Windows 那样的可视的命令输入界面——X Window 的图形用户界面（GUI）。它提供了很多窗口管理器，其操作就像 Windows 一样，有窗口、图标和菜单，所有的管理都是通过鼠标控制的。现在比较流行的窗口管理器是 KDE 和 GNOME。

注意：X Window 其实质是 Linux 实用工具的一种。

每个 Linux 系统的用户可以拥有他自己的用户界面或 Shell，用以满足他们自己专门的 Shell 需要。

同 Linux 本身一样，Shell 也有多种不同的版本，目前主流的 Shell 说明如下：

- Bourne Shell：是贝尔实验室开发的。
- BASH：是 GNU 的 Bourne Again Shell，是 GNU 操作系统上默认的 shell。
- Korn Shell：是对 Bourne SHell 的发展，在大部分内容上与 Bourne Shell 兼容。
- C Shell：是 SUN 公司 Shell 的 BSD 版本。

3．文件系统

文件系统（File System）是文件存放在磁盘等存储设备上的组织方法，其主要体现在对

文件和目录的组织上。

目录提供了管理文件的一个方便而有效的途径，用户可以从一个目录切换到另一个目录，而且可以设置目录和文件的权限，设置文件的共享程度。在 Linux 系统下，用户可以设置目录和文件的权限，以便允许或拒绝其他人对其进行访问。Linux 目录采用多级树形结构，用户可以浏览整个系统，可以进入任何一个已授权进入的目录，访问那里的文件。

文件结构的相互关联性使共享数据变得容易，几个用户可以访问同一个文件。Linux 是一个多用户系统，系统本身的驻留程序存放在以根目录开始的专用目录中，有时被指定为系统目录。

4．实用工具

Linux 操作系统通常都提供一系列叫作实用工具的应用程序，这些实用工具包括和用户进行人机交互的 X Window、计算器、浏览器等，主要用于增加系统可用性，和 Windows 把这些工具（主要是 X Windows）集合到一起不能分离不同，Linux 的实用工具都可以让用户自定义。整体来说，Linux 的实用工具可分为如下三类。

- 编辑器：用于编辑文件，Linux 常见的编辑器主要有 Ed、Ex、Vi 和 Emacs。Ed 和 Ex 是行编辑器，Vi 和 Emacs 是全屏幕编辑器。
- 过滤器：用于接收数据并过滤数据，Linux 的过滤器（Filter）读取从用户文件或其他地方的输入，检查和处理数据，然后输出结果。从这个意义上说，它们过滤了经过它们的数据。Linux 有不同类型的过滤器，一些过滤器用行编辑命令输出一个被编辑的文件。另外一些过滤器是按模式寻找文件并以这种模式输出部分数据。还有一些执行字处理操作，检测一个文件中的格式，输出一个格式化的文件。过滤器的输入可以是一个文件，也可以是用户从键盘输入的数据，还可以是另一个过滤器的输出。过滤器可以相互连接，因此，一个过滤器的输出可能是另一个过滤器的输入。在有些情况下，用户可以编写自己的过滤器程序。
- 交互程序：允许用户发送信息或接收来自其他用户的信息，交互程序是用户与机器的信息接口。Linux 是一个多用户系统，它必须和所有用户保持联系。信息可以由系统上的不同用户发送或接收。信息的发送有两种方式：一种方式是与其他用户一对一地链接进行对话；另一种方式是一个用户对多个用户同时链接进行通信，即所谓广播式通信。

3.1.4　Linux 操作系统的发行版

一般而言，一个基本的 Linux 只是包含了 Linux 核心（Kernel）和 GNU 软件的一些基本的系统软件和实用工具（Utilities），这样一个操作系统仅仅能够让那些 Linux 专家完成一些基本的系统管理任务，若要满足普通用户的办公或基于视窗的应用开发等需要，则还需要在系统中加入 GNOME、KDE 等桌面环境及相应的办公应用软件（如 Office）等。因此一些组织或厂家将 Linux 系统内核与 GNU 软件（系统软件和工具）整合起来，并提供一些安装界面和系统设定与管理工具，这样就构成了一个发行套件，如最常见的 Ubuntu、Fedora 等。实际上发行套件就是 Linux 的一个大软件包而已，通常包括 C 语言及 C++的编译器、Perl 脚本解释程序、Shell 命令解释器、图形用户界面及众多的应用程序等。相对于内核版本，发行套件的版本号随发布者的不同而不同，与系统内核的版本号是相对独立的。因此把 Ubuntu、Fedora

等直接说成是 Linux 是不确切的，它们是 Linux 的发行版本，更确切地说，应该叫作“以 Linux 为核心的操作系统软件包”。根据 GPL 准则，这些发行版本虽然都源自一个内核，并都有自己各自的贡献，但都没有自己的版权。Linux 的各个发行版本都是使用 Linux 主导开发并发布的同一个 Linux 内核，因此在内核层不存在兼容性问题。至于每个版本都有不一样的感觉，只是在发行版本的最外层才有所体现，而绝不是本身，也不是内核不统一或不兼容。

目前 Linux 的发行版很多，其中比较流行的有 Ubuntu、Fedora、Slackware、Debian、OpenSUSE 和 Mandriva 等。

3.2 Linux 操作系统的人机交互方法

Linux 通常使用图形操作界面或 Shell 和用户进行交易，前者是一个类似 Windows 的操作界面，而后者则为类似 DOS 的命令行输入反馈界面。

3.2.1 Linux 的图形界面

几乎所有的 Linux 发行版本中都包含了 GNOME 和 KDE 两种图形操作环境，许多 Linux 操作系统默认的图形操作界面为 GNOME，它除了具有出色的图形环境功能外，还提供了编程接口，允许开发人员按照自己的爱好和需要来设置窗口管理器。2013 年 9 月，GNOME 3.10 发布，该版本带来了包括 Wayland（新一代显示技术）、经过重新设计的全新的系统状态区、标头列、新的应用程序等一系列新的特性和功能。

KDE 桌面环境是一个具有强大网络功能的桌面环境，它功能强大，除了窗口管理器和文件管理器外，基本覆盖了大部分 Linux 任务的应用程序组，同时还结合了 UNIX 操作系统的灵活性，2013 年 2 月 6 日，KDE SC 4.10.0 发布。

Linux 内核本身并不提供图形界面，这些图形界面的实现只是 Linux 下的应用程序实现的，也就是说，不管是 KDE 还是 GNOME，它们只是一个应用软件，并不是类似于 Windows 操作系统的 GUI（图形用户界面），图形界面并不是 Linux 操作系统的一部分。大部分发行版本的 Linux 操作系统中集成了 KDE 和 GNOME 两种图形环境，对一个习惯 Windows 的用户来说，要正确理解 Linux 的图形环境可能颇为困难，因为它与纯图形化 Windows 并没有多少共同点，并且用户在使用过程中与 Windows 并没有多少区别，这里有必要先介绍 UNIX/Linux 图形环境的概念，且要从 UNIX 操作系统说起，并将它们与 Windows 操作系统进行对比。

3.2.2 Linux 的 Shell

Shell 俗称壳（用来区别于核），是指“提供使用者使用界面”的软件（命令解析器），其类似于 DOS 下的 command.com。它接收用户命令，然后调用相应的应用程序。同时它又是一种程序设计语言。作为命令语言，它交互式解释和执行用户输入的命令或自动地解释和执行预先设定好的一连串命令；作为程序设计语言，它定义了各种变量和参数，并提供了许多在高阶语言中才具有的控制结构，包括循环和分支。

Shell 并不是 Linux 独有的，Windows 下也同样有；Shell 也不仅仅是以命令行形式出现的，其实 X Windows 也是 Shell 的一种，不过在本小节中所特指的 Shell 是在 Linux 下以命令行形式提供的。

Shell 基本上是一个命令解释器，其接收用户命令，然后调用相应的应用程序来执行这些命令。

1．常见的 Shell

常见的 Shell 包括 ash、bash、ksh、csh 和 zsh 五种，其简单的介绍如下。

- ash：ash Shell 是由 Kenneth Almquist 编写的，是 Linux 中占用系统资源最少的一个小 Shell，它只包含 24 个内部命令，因而使用起来很不方便。
- bash：bash 是 Linux 系统默认使用的 Shell，它由 Brian Fox 和 Chet Ramey 共同完成，是 Bourne Again Shell 的缩写，内部命令一共有 40 个。Linux 使用它作为默认的 Shell 是因为它有以下的特色：可以使用类似 DOS 下面的 doskey 的功能，用上下方向键查阅和快速输入并修改命令；自动通过查找匹配的方式，给出以某字串开头的命令；包含了自身的帮助功能，只要在提示符下面输入 help 就可以得到相关的帮助。
- ksh：是 Korn Shell 的缩写，由 Eric Gisin 编写，共有 42 条内部命令。该 Shell 最大的优点是几乎和商业发行版的 ksh 完全相容，这样就可以在不用花钱购买商业版本的情况下尝试商业版本的性能了。
- csh：其是在 Linux 操作系统中应用比较多的 Shell，它由以 William Joy 为代表的共 47 位作者编成，共有 52 个内部命令，该 Shell 其实是指向/bin/tcsh 这样的一个 Shell，也就是说，csh 其实就是 tcsh。
- zch：这是 Linux 最大的 Shell 之一，由 Paul Falstad 完成，共有 84 个内部命令。如果只是一般的用途，是没有必要安装这样的 Shell 的。

可以在终端下使用相应的命令来查看当前 Linux 操作系统中使用的 Shell，ubuntu 中默认使用的 Shell 是 bash。

```
alloeat@ubuntu:/$ echo $SHELL
/bin/bash
```

2．Shell 和终端

和 Linux 内核类似，Shell 仅提供了一个计算机和用户进行交互的内核，而其具体的命令行输入输出交流要通过终端来完成，Linux 操作系统中用户也可以自定义终端来完成相应的工作，如 Ubuntu 12.04 发行版自带的终端是 Terminal，其运行界面如图 3.3 所示。

图 3.3　终端运行界面

3．Shell 的工作方式

Shell 既可以作为命令行提供给用户控制内核完成相应的任务，也可以作为一种编程语言供开发者使用。

命令行工作方式：在命令行工作方式下，Shell 识别并且对用户的输入字符串进行响应以完成相应的工作，这种工作方式通常也称为“交互式”的工作方式，当用户有输入的时候 Shell 才对其做出相应的响应。

编程语言工作方式：Shell 同样可以用作编程语言。在 Linux 中存在一种特殊的可执行文件，其内容是一系列由各种命令组成的纯文本文件（脚本文件），其通常用于完成某些步骤比较多的复杂工作或重复性比较强的工作，Shell 可以对这些文件进行识别，并且按照设定自动执行相应的动作，这种工作方式通常也称为“非交互式”的工作方式，不需要用户输入，Shell 会自动做出相应的动作。

注意：Shell 还可以对用户的环境进行配置，这通常会在 Shell 的初始化文件中完成，这些配置包括设置窗口属性、快捷键等。

4．Shell 的启动

Shell 在启动的时候，先读取/etc/bash.bashrc 文件对整个 Linux 操作系统进行配置，然后读取$HOME/.bashrc 文件对当前用户进行配置，如果这两个文件有冲突，则以后者为准，这些文件包括以下方面的内容。

- .bash_profile 文件：该文件只被登录用户对应的 Shell 所读取，而操作系统内未登录的 Shell 只读取.bashrc 文件。
- .bashrc 文件：该文件被启动的所有 Shell 所读取。
- .bash_logout 文件：bash 退出时候执行该文件。

如果用户安装了多个 Shell，则可以在用户管理的相关目录文件中进行设置。

图 3.3 所示的终端实际上是一个虚拟终端，其是在 X Window 中运行的，如果想要进入完整的“真实终端”，可以使用“Ctrl+Alt+Fn”（Fn = F1～F6）。

3.3 Linux 操作系统的命令

在 Linux 中，用户经常需要在 Shell 下使用适当的命令来完成相应的操作，本节将介绍 Linux 中的部分常用命令，熟练掌握这些命令是使用 Linux 和在嵌入式系统中对 Linux 进行开发的必要基础。

注意：本节内容基于 Ubuntu 12.04，不同的 Linux 发行版可能会略有区别。

3.3.1 Linux 操作系统的命令基础

本小节将简要介绍基于 Bash 的 Shell 的基础使用方法，终端的运行界面可以参考图 3.3，以下仅给出在其中进行的相应字符串操作。

1．Shell 命令的标准格式

Shell 和用户交互是以字符串形式存在的命令和命令输出反馈存在的，在 Linux 命令行中输入的第一个字必须是一个命令的名字，第二个字是命令的选项或参数，命令行中的每个字必须由空格或 Tab 隔开，格式如下：

```
$ 命令 选项 参数
```

或者

```
# 命令 选项 参数
```

提示符“$”和“#”区分了用户的不同权限，“$”表示普通用户权限，而“#”代表的是 root 用户（超级用户）权限；选项是包括一个或多个字母的代码，它前面有一个减号（减号是必要的，Linux 用它来区别选项和参数），选项可用于改变命令执行的动作的类型。

注意：在 Ubuntu 操作系统中，用户不能直接使用 root 权限，只能通过 sudo 命令来暂时获得 root 权限。

命令行实际上是一个可以编辑的文本缓冲区，在按回车键之前，可以对输入的文本进行编辑。例如利用“BackSpace”键可以删除刚输入的字符，可以进行整行删除，还可以插入字符，使得用户在输入命令（尤其是复杂命令）时，若出现输入错误，无须重新输入整个命令，只要利用编辑操作即可改正错误。

利用上箭头可以重新显示刚执行的命令，利用这一功能可以重复执行以前执行过的命令，而无须重新输入该命令。

一个标准的 Shell 命令和命令的反馈输出如下，这是一个“ls”查看当前文件夹下文件列表的命令：

```
alloeat@ubuntu:/$ ls
bin   cdrom etc   host   initrd.img.old   lost+found   mnt   proc   run    selinux   sys   usr   vmlinuz
boot dev   home   initrd.img   lib   media      opt   root   sbin   srv      tmp   var   vmlinuz.old
```

2．Shell 的通配符

在 Shell 中除使用普通字符外，还可以使用一些具有特殊含义和功能的字符，称为通配符，在使用它们时应注意其特殊的含义和作用范围。

Shell 的通配符主要用于模式匹配，如文件名匹配、路径名搜索、字串查找等。常用的通配符有“*”、“?”和括在方括号“[ ]”中的字符序列等，用户可以在作为命令参数的文件名中包含这些通配符，构成一个所谓的“模式串”，以在执行过程中进行模式匹配。这三个通配符的含义如下。

- “*”代表任意长度的字串，如“L*”匹配以 L 开头的任意字串。但应注意，文件名中的圆点（.）和路径名中的斜线（/）必须是显式的，即不能用通配符替代它们。例如“*”不能匹配.c，而“.*”才可以匹配.c。
- “?”代表任何单个字符。
- “[ ]”指定了模式串匹配的字符范围，只要文件名中“[ ]”处的字符在指定的范围之内，那么这个文件名就与该模式串匹配。方括号中的字符范围可以由字符串组成，也可以由表示限定范围的起始字符、终止字符及中间连字符（-）组成。例如，f [a-d]与 f [abcd]的作用相同。

Shell 将把与命令行中指定的模式串相匹配的所有文件名都作为命令的参数，形成最终的命令，然后执行这个命令。如果目录中没有与指定的模式串相匹配的文件名，那么 Shell 将使用此模式串本身作为参数传给命令（这正是命令中出现特殊字符的原因所在）。

表 3.1 列举了这些通配符的具体实例及含义。

表 3.1　Shell 的通配符

模式串举例	含　义
*	当前目录下所有文件的名称
Text	当前目录下所有文件名中含有 Text 字串的文件的名称
[ab-dm]*	当前目录下所有以 a，b，c，d，m 开头的文件的名称
[ab-dm]?	当前目录下所有以 a，b，c，d，m 开头且后面只跟一个字符的文件的名称
/usr/bin/??	目录/usr/bin/下所有名称长度为 2 个字符的文件的名称

需要注意的是，中间连字符（-）仅在方括号内有效，表示字符范围，若在方括号外面，就成为普通字符了。而“*”和“?”则只在方括号外有效，若出现在方括号之内，它们也失去通配符的能力，成为普通字符了。例如，模式 L[*?]abc 中只有一对方括号是通配符，而“*”和“?”均为普通字符，因此，它匹配的字串只能是 L*abc 和 L?abc。

以下是一个使用“*”通配符来让“ls”命令只显示当前文件夹中带“exam”的“.c”文件的命令和对应输出。

```
alloeat@ubuntu:～/chapter4Exam$ ls
copytest.txt  exam3write.c～  examaccess.c～  examcopy  examfcntl.c  exammkdir.c～  examtest  examvim.c
exam1open  exam4read.c  examchdirgetcwd  examcp.c  examlseek  examopendir  examumask  foo
exam1open.c  exam4read.c～  examchdirgetcwd.c  examcp.c～  examlstat  examopendir.c  examumask.c  lseektest.txt
exam1open.c～  exam5lseek.c  examchdirgetcwd.c～  examcpoy  examlstat.c  examopendir.c～  examumask.c～  renamebar
exam2create.c  exam5lseek.c～  examchmod  examdup  examlstat.c～  examrename  examutime  testdir
exam2create.c～  examaccess  examchmod.c  examdup.c  exammkdir  examrename.c  examutime.c  thpic.c
exam3write.c  examaccess.c  examchmod.c～  examfcntl  exammkdir.c  examrename.c～  examutime.c～
alloeat@ubuntu:～/chapter4Exam$ ls exam*.c
exam1open.c  exam4read.c  examchdirgetcwd.c  examdup.c  exammkdir.c  examumask.c
exam2create.c  exam5lseek.c  examchmod.c  examfcntl.c  examopendir.c  examutime.c
exam3write.c  examaccess.c  examcp.c  examlstat.c  examrename.c  examvim.c
```

3．Shell 中的引号

在 Shell 中可以使用的引号包括单引号、双引号和反引号三种。

由单引号括起来的字符都作为普通字符出现。特殊字符用单引号括起来以后，也会失去原有意义，而只作为普通字符解释。例如下面的一系列命令：

```
alloeat@ubuntu:～/chapter4Exam$ string='$PATH'
alloeat@ubuntu:～/chapter4Exam$ echo $string
$PATH
```

可见，单引号中的“$”保持了其本身的含义，作为普通字符出现。而在一般情形下，“$”符号的含义是引用变量的值，PATH 本身是一个 Linux 下的环境变量，其值是一系列的目录，当用户运行某个程序时，Linux 在这些目录下进行搜寻。可以使用下面的命令查看变量 PATH 的值：

```
#echo $PATH
```

双引号的作用与单引号类似，区别在于它没有那么严格。单引号告诉 Shell 忽略所有的特殊字符，而双引号只要求忽略大多数字符。具体来说，括在双引号中的三种特殊字符不被忽略：$，\和`，即双引号会解释字符串的特别意义，而单引号则直接使用字符串。如果使用双引号将字符串赋给变量并反馈它，实际上与直接反馈变量并无差别。如果要查询包含空格的字符串，经常会用到双引号。

```
alloeat@ubuntu:/$ x=*                          //定义字符变量 x
alloeat@ubuntu:/$ echo $x                      //显示 x 的值
bin boot cdrom dev etc home host initrd.img initrd.img.old lib lost+found media mnt opt proc root run
sbin selinux srv sys tmp usr var vmlinuz vmlinuz.old
alloeat@ubuntu:/$ echo '$x'                    //单引号
$x
alloeat@ubuntu:/$ echo "$x"                    //双引号
*
```

从以上实例中可以清楚地看出无引号、单引号和双引号之间的区别。

- 第一种情况，显示变量 x 的值。由于 x 的值，即字符*匹配了当前目录（root 目录）下的所有文件名，故显示变量 x 的值时，即显示了当前目录的所有文件名。
- 第二种情况，使用了单引号。单引号中的字符保持其本身的含义，这种情况最简单。
- 最后一种情况，使用了双引号。双引号告诉 Shell 在引号内照样进行变量名替换，所以 Shell 把$x 替换为*，因为双引号中不做文件名替换（忽略掉了非特殊字符），所以就把*作为要显示的值传递给 echo 命令，作为 echo 命令的参数。

另外，从该实例中还可以看到 Shell 赋值的先后次序：Shell 先作变量替换，然后作文件名替换，最后把这些替换值作为参数传递给命令。

反引号“`”字符所对应的键一般位于键盘的左上角，不要将其同单引号“'”混淆。反引号括起来的字串被 Shell 解释为命令行，在执行时，Shell 首先执行该命令行，并以它的标准输出结果取代整个反引号（包括两个反引号）部分。例如：

```
alloeat@ubuntu:/$ pwd
/
alloeat@ubuntu:/$ string="current directory is `pwd`"
alloeat@ubuntu:/$ echo $string
current directory is /
```

Shell 执行 echo 命令时，首先执行`pwd`中的命令 pwd，并将输出结果“/”取代`pwd`部分，最后输出替换后的整个结果。

利用反引号的这种功能可以进行命令置换，即把反引号括起来的执行结果赋值给指定变量。再例如：

```
alloeat@ubuntu:/$ today=`date`
alloeat@ubuntu:/$ echo today is $today
today is 2012 年 08 月 03 日 星期五 16:58:54 CST
```

另外，反引号还可以嵌套使用。但需要注意的是，嵌套使用时内层的反引号必须用反斜线（\）转义。

4．Shell 中的注释符

在 Shell 编程或 Linux 的配置文档中，经常要对某些正文行进行注释，以增强程序的可读性。在 Shell 中以字符 # 开头的正文行表示注释行。

3.3.2 目录操作命令

Linux 系统以文件目录的方式来组织和管理系统中的所有文件。所谓文件目录就是将所有文件的说明信息采用树形结构组织起来，即常说的目录。

1．创建目录命令（mkdir）

在 Linux 系统中建立新目录的命令是 mkdir。该命令的使用方式如下：

```
mkdir [选项] 目录
```

mkdir 命令选项说明如表 3.2 所示。

表 3.2 mkdir 命令选项说明

选　项	说　明
-m	在建立目录时把按模式指定设置为目录权限。该目录的权限分为：目录所有者的权限、组中其他人对目录的权限和系统中其他人对目录的权限。这三个权限分别用三个数字之和来表示：对目录的读权限是 4、写权限是 2、执行权限是 1
-p	可以是一个路径名称。此时若路径中的某些目录尚不存在，加上此选项后，系统将自动建立好那些尚不存在的目录，即一次可以建立多个目录

如下是一个使用 mkdir 命令在当前目录中建立一个 dir3 目录并且将其权限设置为只有文件拥有者才能读写和执行。

```
alloeat@ubuntu:~/Exammkdir$ ls
dir1  dir2
alloeat@ubuntu:~/Exammkdir$ mkdir -p -m 700 dir3
alloeat@ubuntu:~/Exammkdir$ ls -l
总用量 12
drwxrwxr-x 2 alloeat alloeat 4096  8 月  4 10:42 dir1
drwxrwxr-x 2 alloeat alloeat 4096  8 月  4 10:42 dir2
```

2．删除目录命令（rmdir）

与创建目录对应的是删除目录，rmdir 命令用来删除目录，一般情况下要删除的目录必须为空目录，如果所给的目录不为空，系统会报告错误。该命令的使用方式如下：

```
rmdir [选项] 目录列表
```

rmdir 命令选项说明如表 3.3 所示。

表 3.3　rmdir 命令选项说明

选　项	说　明
-p	在删除目录表指定的目录后，若父目录为空，则 rmdir 也删除父目录。状态信息显示什么被删除、什么没被删除

3. 显示当前工作目录命令（pwd）

显示当前工作目录的命令是 pwd 命令，该命令的使用方式如下：

```
pwd
```

如下是一个使用 pwd 命令来显示当前工作目录的应用实例。

```
alloeat@ubuntu:~/Exammkdir$ pwd
/home/alloeat/Exammkdir
```

4. 改变当前工作目录命令（cd）

改变当前工作目录在 Linux 系统中使用的是 cd 命令。该命令的使用方式如下：

```
cd    [directory]
```

该命令将当前目录改变至 directory 所指定的目录。若没有指定 directory，则回到用户的主目录。为了改变到指定目录，用户必须拥有对指定目录的执行和读权限。该命令可以使用通配符。

如下是一个使用 cd 命令在多个目录路径下切换的应用实例。

```
alloeat@ubuntu:~/Exammkdir$ ls
dir1  dir2  dir3
alloeat@ubuntu:~/Exammkdir$ cd dir1
alloeat@ubuntu:~/Exammkdir/dir1$ cd ../dir2
alloeat@ubuntu:~/Exammkdir/dir2$ cd ..
alloeat@ubuntu:~/Exammkdir$ cd /
alloeat@ubuntu:/$ ls
bin    dev    host              lib          mnt    root   selinux  test   var
boot   etc    initrd.img        lost+found   opt    run    srv      tmp    vmlinuz
cdrom  home   initrd.img.old    media        proc   sbin   sys      usr    vmlinuz.old
```

5. 改变目录权限命令（chmod）

在 Linux 系统中，用户设定文件权限控制其他用户不能访问、修改。但在系统应用中，有时需要让其他用户使用某个原来其不能访问的文件或目录，这时就需要重新设置文件的权限，使用的命令是 chmod 命令。并不是谁都可改变文件和目录的访问权限，只有文件和目录的所有者才有权限修改其权限，另外，超级用户可对所有文件或目录进行权限设置。该命令的使用方式如下：

```
chmod [who] [+ | - | =] [mode]  文件名
```

chmod 命令中的操作对象 who 可以是表 3.4 字母中的任一个或它们的组合。

表 3.4　chmod 命令 who 选项说明

选　项	说　明
u	表示“用户（user）”，即文件或目录的所有者
g	表示“同组（group）用户”，即与文件属主有相同组 ID 的所有用户
o	表示“其他（others）用户”
a	表示“所有（all）用户”。它是系统默认值

chmod 命令操作符号说明如表 3.5 所示。

表 3.5　chmod 命令操作符号说明

选　项	说　明
+	添加某个权限
-	取消某个权限
=	赋予给定权限并取消其他所有权限（如果有的话）

mode 所表示的权限可以是表 3.6 中字母的任意组合。

表 3.6　chmod 命令 mode 选项说明

选　项	说　明
r	可读
w	可写
x	可执行
X	只有目标文件对某些用户是可执行的或该目标文件是目录时才追加 x 属性
s	在文件执行时把进程的属主或组 ID 置为该文件的文件属主。方式“u+s”设置文件的用户 ID 位，“g+s”设置组 ID 位
t	保存程序的文本到交换设备上
u	与文件属主拥有一样的权限
g	与和文件属主同组的用户拥有一样的权限
o	与其他用户拥有一样的权限

在一个命令行中可给出多个权限方式，其间用逗号隔开。例如 chmod g+r，o+r example，这个命令将使同组和其他用户对文件 example 有读权限。

文件和目录的权限还可用八进制数字模式来表示。首先了解用数字表示的属性的含义：0 表示没有权限，1 表示可执行权限， 2 表示可写权限，4 表示可读权限，然后将其相加。所以数字属性的格式应为 3 个从 0 到 7 的八进制数，其顺序是（u）（g）（o）。例如，如果想让某个文件的属主有“读/写”两种权限，需要把 4（可读）+2（可写）＝6（读/写）。

数字设定法的一般形式为：

```
chmod  [mod]  文件名
```

如下是一个使用 chmod 命令来修改文件 chmodtest.txt 的权限的实例，在其中设定文件 chmodtest.txt 的属性为：文件属主（u）增加执行权限，与文件属主同组用户（g）和其他用户（o）增加读权限。

```
alloeat@ubuntu:~/Exammkdir$ ls chmodtest.txt -l
-rw-rw-r-- 1 alloeat alloeat 16　8 月　4 10:59 chmodtest.txt
alloeat@ubuntu:~/Exammkdir$ chmod u+x chmodtest.txt
alloeat@ubuntu:~/Exammkdir$ ls chmodtest.txt -l
-rwxrw-r-- 1 alloeat alloeat 16　8 月　4 10:59 chmodtest.txt
alloeat@ubuntu:~/Exammkdir$ chmod go+w chmodtest.txt
alloeat@ubuntu:~/Exammkdir$ ls chmodtest.txt -l
-rwxrw-rw- 1 alloeat alloeat 16　8 月　4 10:59 chmodtest.txt
```

6. **改变目录所属命令（chown）**

chown 命令用来更改某个文件或目录的属主和属组。这个命令也很常用。例如 root 用户把自己的一个文件复制给用户 alloeat，为了让用户 alloeat 能够存取这个文件，root 用户应该把这个文件的属主设为 alloeat，否则，用户 alloeat 无法存取这个文件。该命令的使用方式如下：

```
chown [选项] 用户或组 文件
```

chown 将指定文件的拥有者改为指定的用户或组。用户可以是用户名或用户 ID。组可以是组名或组 ID。文件是以空格分开的要改变权限的文件列表，支持通配符。chown 命令选项说明如表 3.7 所示。

表 3.7　chown 命令选项说明

选　项	说　明
-c	若该文件拥有者确实已经更改，才显示其更改动作
-f	若该文件拥有者无法被更改也不要显示错误信息
-i	在删除与链接名同名的文件时先进行询问
-h	只对链接进行变更，而非该链接真正指向的文件
-v	显示拥有者变更的详细资料
-R	递归式地改变指定目录及其下的所有子目录和文件的拥有者

3.3.3　文件操作命令

文件操作是 Linux 系统里最基本也是最常用的操作。

1. **列举文件命令（ls）**

ls 用于显示指定工作目录中所包含的文件，该命令的使用方法如下：

```
ls [选项]　[文件目录列表]
```

ls 命令中的常用选项如表 3.8 所示。

表 3.8　ls 命令选项说明

选　项	说　明
-a	列出目录下的所有文件，包括以“.” 开头的隐含文件
-b	把文件名中不可输出的字符用反斜杠加字符编号（就像在 C 语言里一样）的形式列出
-c	输出文件的 i 节点的修改时间，并以此排序

续表

选　　项	说　　明
-d	将目录像文件一样显示，而不是显示其下的文件
-e	输出时间的全部信息，而不是输出简略信息
-f-U	对输出的文件不排序
-i	输出文件的 i 节点的索引信息
-k	以 k 字节的形式表示文件的大小
-l	列出文件的详细信息
-m	横向输出文件名，并以“,”作为分隔符
-n	用数字的 UID,GID 代替名称
-o	显示文件的除组信息外的详细信息
-p -F	在每个文件名后附上一个字符以说明该文件的类型，“*”表示可执行的普通文件；“/”表示目录；“@”表示符号链接；“|”表示 FIFOs；“=”表示套接字（sockets）
-q	用?代替不可输出的字符
-r	对目录反向排序
-s	在每个文件名后输出该文件的大小
-t	以时间排序
-u	以文件上次被访问的时间排序
-x	按列输出，横向排序
-A	显示除 “.”和“..”外的所有文件
-B	不输出以 “~”结尾的备份文件
-C	按列输出，纵向排序
-G	输出文件的组的信息
-L	列出链接文件名而不是链接到的文件
-N	不限制文件长度
-Q	把输出的文件名用双引号括起来
-R	列出所有子目录下的文件
-S	以文件大小排序
-X	以文件的扩展名（最后一个“.”后的字符）排序
-1	一行只输出一个文件
--color=no	不显示彩色文件名

由于 Linux 支持多种文件类型，每一类用一个字符来表示，其说明如表 3.9 所示。

表 3.9　Linux 的文件类型说明

文 件 类 型	说　　明
-	常规文件
d	目录
b	块特殊设备
c	字符特殊设备
p	有名管道
s	信号灯
m	共享存储器

文件类型的字符表示文件的权限，权限由三个字符串组成，这三个字符串分别表示：该文件所有者的权限、组中其他人的权限和系统中其他人的权限；每个字符串又由三个字符

组成，依次表示对文件的读（用字符 r 表示）、写（用字符 W 表示）和执行权限（用字符 x 表示）。当用户没有相应的权限时，该权限的对应位置用短线“-”来表示。 例如：

```
drwxr-x---
```

表示的含义是：d 表示该文件是目录；目录拥有者的权限是 rwx（表示有读、写和执行权限）；组中其他人对该目录的权限是 r-x（表示有读和执行权限，没有写权限），系统中其他人对该目录的权限是---（表示读、写和执行权限都没有）。

如下是一个使用“ls”命令来显示当前根目录下文件列表的应用实例。

```
alloeat@ubuntu:/$ ls
bin    dev    host          lib          mnt    root   selinux  tmp   vmlinuz
boot   etc    initrd.img    lost+found   opt    run    srv      usr   vmlinuz.old
cdrom  home   initrd.img.old media       proc   sbin   sys      var
```

2. 查找文件命令（find）

在 Linux 系统中，可以使用 find 命令来查找文件，其标准使用格式如下：

```
find [目录列表]  [匹配标准]
```

find 命令有两个目录列表和匹配标准两个参数，其说明如下。

- 目录列表：希望查询文件或文件集的目录列表，目录间用空格分隔。
- 匹配标准：希望查询文件的匹配标准或说明，其详细说明如表 3.10 所示。

表 3.10　find 命令的匹配标准参数说明

选　项	说　明
-amin n	查找系统中最后 n 分钟访问的文件
-atime n	查找系统中最后 n*24 小时访问的文件
-cmin n	查找系统中最后 n 分钟被改变状态的文件
-ctime n	查找系统中最后 n*24 小时被改变状态的文件
-empty	查找系统中空白的文件，或空白的文件目录，或目录中没有子目录的文件夹
-false	查找系统中总是错误的文件
-fstype type	查找系统中存在于指定文件系统的文件，如 ext2
-gid n	查找系统中文件数字组 ID 为 n 的文件
-group gname	查找系统中文件属于 gnam 文件组，并且指定组和 ID 的文件
-daystart	测试系统从今天开始 24 小时以内的文件，用法类似-amin
-depth	使用深度级别的查找过程方式，在某层指定目录中优先查找文件内容
-follow	遵循通配符链接方式查找；另外，也可忽略通配符链接方式查询
-maxdepth levels	在某个层次的目录中按照递减方法查找
-mount	不在文件系统目录中查找

如下是在目录“home/alloeat/chapter4Exam/”下查找 exam5lseek.c 文件的命令和对应的响应输出：

```
alloeat@ubuntu:/$ find home/alloeat/chapter4Exam/exam5lseek.c
home/alloeat/chapter4Exam/exam5lseek.c
```

3. 查找文件内容命令（grep）

查找文件内容的命令是 grep 命令。该命令的使用方式如下：

```
grep  [选项]  [查找模式]  [文件名 1，文件名 2，……]
```

grep 命令选项说明如表 3.11 所示。

表 3.11　grep 命令选项说明

选　项	说　明
-E	每个模式作为一个扩展的正则表达式对待
-F	每个模式作为一组固定字符串对待（以新行分隔），而不作为正则表达式
-b	在输出的每一行前显示包含匹配字符串的行在文件中的字节偏移量
-c	只显示匹配行的数量
-i	比较时不区分大小写
-h	在查找多个文件时，指示 grep 不要将文件名加入到输出之前
-l	显示首次匹配串所在的文件名并用换行符将其隔开。当在某文件中多次出现匹配串时，不重复显示此文件名
-n	在输出前加上匹配串所在行的行号（文件首行行号为 1）
-v	只显示不包含匹配串的行
-x	只显示整行严格匹配的行
-e expression	指定检索使用的模式。用于防止以“-”开头的模式被解释为命令选项
-f expfile	从 expfile 文件中获取要搜索的模式，一个模式占一行

如下是在一个 C 语言文件中查找 printf 字符串的实例。

```
alloeat@ubuntu:~/chapter4Exam$ grep printf -n exam5lseek.c
10:          printf("run error!\n");
```

4. 显示文件内容命令（cat）

显示文本文件内容的命令是 cat 命令，用来将文件的内容显示到终端上。该命令的使用方式如下：

```
cat  [选项]  文件列表
```

cat 命令选项说明如表 3.12 所示。

表 3.12　cat 命令选项说明

选　项	说　明
-v	用一种特殊形式显示控制字符，LFD 与 TAB 除外。加了-v 选项后，-T 和 -E 选项将起作用。其中：-T 将 TAB 显示为“ù I”。该选项需要与 -v 选项一起使用。即如果没有使用 -v 选项，则这个选项将被忽略。-E 在每行的末尾显示一个 $ 符。该选项需要与 -v 选项一起使用
-u	输出不经过缓冲区
-A	等于 -vET
-t	等于 -vT
-e	等于 -vE
-n	在文件的每行前面显示行号

如下是使用 cat 命令来显示某个 C 语言文件的应用实例，在每一行之前加上了行编号。

```
alloeat@ubuntu:~/chapter4Exam$ cat -n exam5lseek.c
     1  #include <stdio.h>
     2
     3  int main(int argc,char *argv[])
     4  {
     5     int temp,seektemp,i,j;
     6     FILE *fp;                    //文件指针
     7     char wbuf[17] = "this is a test!\r\n";
     8     if(argc!= 2)
     9     {
    10            printf("run error!\n");
    11            return 1;                                    //如果参数不正确则退出
    12     }
    13     fp = fopen(*(argv+1),w);        //打开文件
    14  //  temp = fputs(wbuf,fp);          //写入数据
    15  //   seektemp = lseek(fileID,0,SEEK_CUR);          //获得当前的偏移量
    16     for(i=0;i<10;i++)
    17     {
    18        j = sizeof(wbuf) * (i+1);                  //计算下一次的偏移量
    19        fseek(fp,j,SEEK_SET);
    20        temp = fputs(wbuf,fputs);     //写入数据
    21     }
    22     fclose(fp);
    23     return 0;
    24  }
    25
```

5. 文件排序命令（sort）

sort 命令的功能是对文件中的各行进行排序。sort 命令有许多非常实用的选项，这些选项最初是用来对数据库格式的文件内容进行各种排序操作的。实际上，sort 命令可以被认为是一个非常强大的数据管理工具，用来管理内容类似数据库记录的文件。sort 命令将逐行对文件中的内容进行排序，如果两行的首字符相同，该命令将继续比较这两行的下一字符，如果还相同，将继续进行比较。该命令的使用方式如下：

```
sort  [选项]  文件
```

sort 命令对指定文件中所有的行进行排序，并将结果显示在标准输出上。如果不指定输入文件或使用“-”，则表示排序内容来自标准输入。

sort 排序是根据从输入行抽取的一个或多个关键字进行比较来完成的。排序关键字定义了用来排序的最小的字符序列。默认情况下以整行为关键字按 ASCII 字符顺序进行排序。

sort 命令选项说明如表 3.13 所示。

表 3.13　sort 命令选项说明

选　项	说　明
-m	若给定文件已排好序，合并文件
-c	检查给定文件是否已排好序，如果它们没有都排好序，则打印一个出错信息，并以状态值 1 退出
-u	对排序后认为相同的行只留其中一行
-o	输出文件将排序输出写到输出文件中而不是标准输出，如果输出文件是输入文件之一，sort 先将该文件的内容写入一个临时文件，然后排序和写输出结果
-d	按字典顺序排序，比较时仅字母、数字、空格和制表符有意义
-f	将小写字母与大写字母同等对待
-l	显示首次匹配串所在的文件名并用换行符将其隔开。当在某文件中多次出现匹配串时，不重复显示此文件名
-I	忽略非打印字符
-M	作为月份比较：“JAN” < “FEB” <… < “DEC”
-r	按逆序输出排序结果
+posl -pos2	指定一个或几个字段作为排序关键字，字段位置从 posl 开始，到 pos2 为止（包括 posl，不包括 pos2）。如果不指定 pos2，则关键字为从 posl 到行尾。字段和字符的位置从 0 开始
-b	在每行中寻找排序关键字时忽略前导的空白（空格和制表符）
-t separator	指定字符 separator 作为字段分隔符

6. 文件比较命令（comm）

如果想对两个有序的文件进行比较，可以使用 comm 命令。该命令的使用方式如下：

```
comm [-123 ] file1 file2
```

该命令是对两个已经排好序的文件进行比较。其中 file1 和 file2 是已排序的文件。comm 读取这两个文件，然后生成三列输出：仅在 file1 中出现的行；仅在 file2 中出现的行；在两个文件中都存在的行。如果文件名用“- ”，则表示从标准输入读取。选项 1、2 或 3 抑制相应的列显示。

```
comm -12   只显示在两个文件中都存在的行。
comm -23   只显示在第一个文件中出现而未在第二个文件中出现的行。
comm -123  则什么也不显示。
```

7. 文件内容比较命令（diff）

diff 命令的功能为逐行比较两个文本文件，列出其不同之处。它对给出的文件进行系统的检查，并显示出两个文件中不同的行，不要求事先对文件进行排序。该命令的使用方式如下：

```
diff  [选项]  file1  file2
```

该命令告诉用户，为了使两个文件 file1 和 file2 一致，需要修改它们的哪些行。如果用“-”表示 file1 或 file2，则表示标准输入。如果 file1 或 file2 是目录，那么 diff 将使用该目录中的同名文件进行比较。 通常输出由下述形式的行组成：

```
n1 a n3，n4
n1，n2 d n3
```

```
n1，n2 c n3，n4
```

字母（a、d 和 c）之前的行号（n1，n2）是针对 file1 的，其后面的行号（n3，n4）是针对 file2 的。字母 a、d 和 c 分别表示附加、删除和修改操作。

在上述形式的每一行的后面跟随受到影响的若干行，以“<”打头的行属于第一个文件，以“>”打头的行属于第二个文件。

diff 能区别块和字符设备文件及 FIFO（管道文件），不会把它们与普通文件进行比较。

如果 file1 和 file2 都是目录，则 diff 会产生很多信息。如果一个目录中只有一个文件，则产生一条信息，指出该目录路径名和其中的文件名。

diff 命令选项说明如表 3.14 所示。

表 3.14　diff 命令选项说明

选　项	说　明
-b	忽略行尾的空格，而字符串中的一个或多个空格符都视为相等
-c	采用上下文输出格式（提供三行上下文）
-C n	采用上下文输出格式（提供 n 行上下文）
-e	产生一个合法的 ed 脚本作为输出
-r	当 file1 和 file2 是目录时，递归作用到各文件和目录上

注意：diff 命令常常用于对比经过修改的文件的前后异同。

8. 复制文件命令（cp）

Linux 下的 cp 命令用于复制文件或目录，该命令的使用方式如下：

```
cp [选项] 源文件或目录 目标文件或目录
```

该命令把指定的源文件复制到目标文件或把多个源文件复制到目标目录中。

cp 命令选项说明如表 3.15 所示。

表 3.15　cp 命令选项说明

选　项	说　明
-a	该选项通常在复制目录时使用。它保留链接、文件属性，并递归地复制目录，其作用等于 dpR 选项的组合
-d	复制时保留链接
-f	删除已经存在的目标文件而不提示
-i	和 f 选项相反，在覆盖目标文件之前将给出提示要求用户确认。回答 y 时目标文件将被覆盖，是交互式复制
-p	此时 cp 除复制源文件的内容外，还将把其修改时间和访问权限也复制到新文件中
-r	若给出的源文件是一目录文件，此时 cp 将递归复制该目录下所有的子目录和文件。此时目标文件必须为一个目录名
-l	不作复制，只是链接文件

注意：为防止用户在不经意的情况下用 cp 命令破坏另一个文件，如果指定的目标文件名是一个已存在的文件名，用 cp 命令复制文件后，这个文件就会被新复制的源文件覆盖，

因此，在使用 cp 命令复制文件时，最好使用-i 选项。

9. 移动和重命名文件命令（mv）

在 Linux 系统中，移动文件可使用 mv 命令。mv 命令还可为文件改名，即把源文件以一个新文件名移动到另一个新的目录中去。该命令的使用方式如下：

```
mv [选项] 源文件名 目标文件名
mv [选项] 源目录名 目标目录名 2
mv [选项] 文件列表 目录
```

mv 命令选项说明如表 3.16 所示。

表 3.16 mv 命令选项说明

选　项	说　明
-b	当遇到要覆盖其他文件或目录时，将自动备份，备份文件名为原文件名加上 -S 参数指定的字符串，若未设置则加上“~”
-i	交互模式，当移动的目录已存在同名的目标文件名时，用覆盖方式写文件，但在写入之前给出提示
-f	通常情况下，目标文件存在但用户没有写权限时，mv 会给出提示。本选项会使 mv 命令执行移动而不给出提示
-u	当要覆盖的文件或目录比源文件新时，不覆盖目标文件
-S <字符串>	指定备份文件名后要加上的字符串

10. 文件内容统计命令（wc）

wc 命令的功能为统计指定文件中的字节数、字数、行数，并将统计结果显示输出。该命令使用方式如下：

```
wc [选项] 文件列表
```

该命令统计给定文件中的字节数、字数、行数。如果没有给出文件名，则从标准输入读取。wc 同时也给出所有指定文件的总统计数。字是由空格字符区分开的最大字符串。

wc 命令选项说明如表 3.17 所示。

表 3.17 wc 命令选项说明

选　项	说　明
-c	统计字节数
-l	统计行数
-w	统计字数

3.3.4 磁盘管理命令

磁盘管理命令包括对文件进行压缩的命令和对磁盘管理的命令。

1. 压缩命令（tar）

文件的压缩、解压与打包在嵌入式 Linux 中经常使用，特别是解压。因为通常 Linux 下的源码都是以 gz、bz2 等打包的形式提供的，常用的压缩、解压命令都是基于 tar 命令的。

tar 命令将用户所指定的文件或目录打包成一个文件，不过它并不做压缩。一般 UNIX

上常用的压缩方式是先用 tar 命令将许多文件打包成一个文件，再用 gzip 等压缩命令压缩文件，该命令使用方法如下：

```
tar [选项] 压缩后的文件名 要被压缩的文件
```

tar 命令选项说明如表 3.18 所示。

表 3.18　tar 命令选项说明

选　项	说　明
-c	创建一个新的 tar 文件
-v	显示运作过程信息
-f	指定文件名称
-z	调用 gzip 压缩命令执行压缩
-j	调用 bzip2 压缩命令执行压缩
-t	参看压缩文件内容
-x	解开 tar 文件

tar 命令本身没有压缩能力，但是可以在产生的 tar 文件后立即使用其他压缩命令来压缩，省去需要输入两次命令的麻烦。使用-z 参数来解开最常见的.tar.gz 文件，如将文件解开至当前目录下的命令为：

```
$ tar -zxvf foo.tar.gz
```

使用-j 参数解开 tar.bz2 压缩文件，如将文件解开至当前目录下，则可使用命令：

```
$ tar –jxvf linux-2.6.25.tar.bz2
```

使用-Z（大写 Z）参数指定以 compress 命令压缩。例如，欲将当前用户所在的目录下所有后缀名为.tif 的文件打包，并压缩成.tar.Z 文件，压缩后的文件名为“picture.tar.Z”，应该使用命令：

```
$ tar –cZvf picture.tar.Z *.tif
```

2. 查看分区命令（fdisk）

fdisk 可以查看硬盘分区情况，并可对硬盘进行分区管理。此外 fdisk 也是一个非常好的硬盘分区工具，该命令的使用方法如下：

```
fdisk [-l]
```

说明：使用 fdisk 命令时必须拥有 root 权限。

IDE 硬盘对应的设备名称分别为 hda、hdb、hdc 和 hdd，SCSI 硬盘对应的设备名称则为 sda、sdb 等。此外，hda1 代表 hda 的第一个硬盘分区，hda2 代表 hda 的第二个分区，依此类推。

通过查看/var/log/messages 文件，可以找到 Linux 系统已辨认出来的设备代号，如果如下使用 fdisk 命令：

```
# fdisk -l
```

则将在屏幕上显示输出：

```
Disk /dev/hda: 40.0 GB， 40007761920 bytes
240 heads， 63 sectors/track， 5168 cylinders
Units = cylinders of 15120 * 512 = 7741440 bytes
   Device Boot      Start         End      Blocks   Id  System
/dev/hda1   *           1        1084     8195008+   c  W95 FAT32 (LBA)
/dev/hda2            1085        5167    30867480    f  W95 Ext'd (LBA)
/dev/hda5            1085        2439    10243768+   b  W95 FAT32
/dev/hda6            2440        4064    12284968+   b  W95 FAT32
/dev/hda7            4065        5096     7799526   83  Linux
/dev/hda8            5096        5165      522081   82  Linux swap
```

3. **挂载命令（mount）**

mount 用于挂载文件系统，其使用权限是超级用户或/etc/fstab 中允许的使用者。挂载是指把分区和目录对应的过程，而挂载点是指挂载在文件树中的位置。mount 命令可以把文件系统挂载到相应的目录下，并且由于 Linux 中把设备都当作文件一样使用，因此，mount 命令也可以挂载不同的设备。

在 Linux 下“/mnt”目录是专门用于挂载不同的文件系统的，它可以在该目录下新建不同的子目录来挂载不同的设备文件系统，其使用方法说明如下：

```
mount [选项] [类型]  设备文件名 挂载点目录
```

其中的类型是指设备文件的类型，mount 的各个参数的含义如表 3.19 所示。

表 3.19　mount 参数表

选　项	说　明
-a	依照/etc/fstab 的内容装载所有相关的硬盘
-l	列出当前已挂载的设备、文件系统名称和挂载点
-t	将后面的设备以指定类型的文件格式装载到挂载点上。常见的类型有前面介绍过的几种：vfat、ext3、ext2、iso9660、nfs 等
-f	通常用于除错。它会使 mount 不执行实际挂上的动作，而是模拟整个挂上的过程，通常会和-v 一起使用

使用 mount 命令时的主要几个步骤如下。

（1）确认是否为 Linux 可以识别的文件系统。Linux 可识别的文件系统只有以下几种：Windows 95/98 常用的 FAT32 文件系统 vfat；Windows NT/2000 用的文件系统 ntfs；OXS 的文件系统 hpfs；Linux 用的文件系统：ext2、ext3、nfs 和 CD-ROM 光盘用的文件系统 iso9660。

（2）确定设备的名称，确定设备名称可通过使用命令“fdisk -l”查看。

（3）查找挂接点，需要注意的是在此之前必须确定挂载点已经存在，也就是“/mnt”下的相应子目录已经存在，一般建议在“/mnt”下新建几个如“/mnt/windows”、“/mnt/usb”的子目录，现在有些新版本的 Linux（如 Ubuntu、红旗 Linux、中软 Linux、Mandrake Linux）都可自动挂载文件系统，Red Hat 仅可自动挂载光驱。

（4）挂载文件系统，使用如下命令：

```
# mount -t vfat /dev/hda1 /mnt/c
```

（5）在使用完该设备文件后可使用命令 umount 将其卸载，如在终端输入：

```
# umount /mnt/c
```

此时已经完成磁盘的挂载。

3.3.5　用户管理命令

用户管理命令用于在 Linux 中实现对登录用户和进程的管理。

1. 用户切换命令（su）

Linux 是一种多用户操作系统，如果所有用户共享一个账号，会造成许多麻烦。因此在 Linux 中每个用户都有自己的账号，各个用户的账号可以根据需要分配不同的权限。Linux 提供了与之相关的用户操作命令。su 命令可以用来切换用户身份，该命令的使用方式如下：

```
su　[选项]　user
```

除 root 外，其他用户切换身份时，需输入密码，su 命令选项说明如表 3.20 所示。

表 3.20　su 命令选项说明

选　项	说　明
-p	执行 su 时不改变环境参数
-c	切换到 user 用户并执行指令（command），然后切换回原来的用户
-s	Shell 指定要执行的 Shell，默认在/etc/passwd 文件中已设置完成，若用户需改 Shell，可采用此参数

sudo 命令用来以系统管理员的身份执行指令，该命令的使用方式如下：

```
sudo [选项] 命令
```

以系统管理者的身份执行指令，也就是说，经由 sudo 所执行的指令就好像是 root 亲自执行的一样。sudo 命令选项说明如表 3.21 所示。

表 3.21　sudo 命令选项说明

选　项	说　明
-l	显示出执行 sudo 的用户的权限
-v	sudo 在第一次执行时或是在 N 分钟内没有执行（N 预设为 5）会问密码，这个参数将重新做一次确认，如果超过 N 分钟，也会问密码
-k	强迫用户在下一次执行 sudo 时间密码（不论有没有超过 N 分钟）
-b	执行的指令放在后台执行
-p prompt	更改问密码的提示语，其中 %u 会代换为使用者的账号名称， %h 会显示主机名称
-u username/#uid	不加此参数，代表要以 root 的身份执行指令，而加了此参数，可以以 username 的身份执行指令（#uid 为该 username 的使用者账号）
-s	执行环境变数中的 Shell 所指定的 Shell，或/etc/passwd 里所指定的 Shell
-H	将环境变数中的 home 目录指定为要变更身份的使用者 home 目录（如不加-u 参数就是系统管理者 root）

2. 进程管理命令（ps 和 kill）

ps 命令用于显示当前系统中由该用户运行的进程列表，而 kill 命令用于输出特定的信号给指定进程号（PID）的进程并根据该信号完成指定的行为，其中可能的信号有进程挂起、

进程等待、进程终止等，它们的使用方法说明如下：

```
ps：ps [选项]
kill：kill [选项]  进程号(PID)
```

ps 命令选项说明如表 3.22 所示；kill 命令中的进程号为信号输出的指定进程的进程号，当选项默认时为输出终止信号给该进程。

表 3.22 ps 命令选项说明

选　项	说　明
-ef	查看所有进程及其 PID（进程号）、系统时间、命令的详细目录、执行者等
-aux	除可显示-ef 所有内容外，还可显示 CPU 及内存占用率、进程状态
-w	以加宽方式显示，这样可以显示较多的信息

Kill 命令选项说明如表 3.23 所示。

表 3.23 kill 命令选项说明

选　项	说　明
-s	将指定信号发送给进程

例如，在命令行中，输入以下命令：

```
# ps –ef
```

系统将会显示所有的进程，如下所示：

```
UID        PID  PPID  C  STIME      TTY      TIME      CMD
root         1     0  0  2005       ?        00:00:05  init
root         2     1  0  2005  ?       00:00:00  [keventd]
root         3     0  0  2005  ?       00:00:00  [ksoftirqd_CPU0]
root         4     0  0  2005  ?       00:00:00  [ksoftirqd_CPU1]
root      7421     1  0  2005  ?       00:00:00  /usr/local/bin/ntpd -c /etc/ntp
root     21787 21739  0  17:16 pts/1   00:00:00  grep ntp
```

该例中，先查看所有进程，接下来要终止进程号为 7421 的 ntp 进程。输入如下命令：

```
# kill 7421
```

之后再次查看，使用命令如下：

```
# ps –ef | grep ntp
```

系统输出：

```
root     21789 21739    0 17:16 pts/1       00:00:00 grep ntp
```

可以看出，已经没有该进程号的进程，说明该进程已经被删除。

注意：ps 命令通常可以与其他一些命令结合起来使用，主要作用是提高效率。ps 选项

中的参数 w 可以写多次，通常最多写 3 次，它的含义表示加宽 3 次，这足以显示很长的命令行了。例如：ps －auxwww。

3.3.6　网络管理命令

网络管理相关的命令很多，常用的是 ifconfig、ftp。一般用来设置网络或使用网络服务等。

1．IP 地址管理命令（ifconfig）

ipconfig 命令用于查看和配置网络接口的地址和参数，包括 IP 地址、网络掩码、广播地址，它的使用权限是超级用户，其有两种使用格式，分别用于查看和更改网络接口。

ifconfig [选项] [网络接口]：用来查看当前系统的网络配置情况。

ifconfig 网络接口 [选项] 地址：用来配置指定接口(如 eth0、eth1)的 IP 地址、网络掩码、广播地址等。

ifconfig 第二种使用方式的常见选项说明如表 3.24 所示。

表 3.24　ifconfig 命令选项说明

选　项	说　明
-interface	指定的网络接口名，如 eth0 和 eth1
up	激活指定的网络接口卡
down	关闭指定的网络接口卡
broadcast address	设置接口的广播地址
poin to point	启用点对点方式
address	设置指定接口设备的 IP 地址
netmask address	设置接口的子网掩码地址

2．ftp 协议操作命令（ftp）

ftp 命令允许用户利用 ftp 协议上传和下载文件，其使用方法说明如下：

ftp [选项] [主机名/IP]

ftp 的相关命令包括使用命令和内部命令，其中使用命令的格式如上所列，主要用于登录 ftp 服务器的过程中。内部命令是指成功登录后进行的一系列操作，下面会详细列出。若用户默认“主机名/IP”，则可在转入到 ftp 内部命令后继续选择登录，ftp 命令选项说明如表 3.25 所示。

表 3.25　ftp 命令选项说明

选　项	说　明
-v	显示远程服务器的所有响应信息
-n	限制 ftp 的自动登录
-d	使用调试方式
-g	取消全局文件名

ftp 常见内部命令选项说明如表 3.26 所示。

表 3.26　ftp 常见内部命令选项说明

选　项	说　明
account[password]	提供成功登录远程系统后访问系统资源所需的补充口令

续表

选　项	说　明
ascii	使用 ASCII 类型传输方式，为默认传输模式
bin/ type binary	使用二进制文件传输方式（嵌入式开发中的常见方式）
bye	退出 ftp 会话过程
cd remote-dir	进入远程主机目录
cdup	进入远程主机目录的父目录
chmod mode file-name	将远程主机文件 file-name 的存取方式设置为 mode
close	中断与远程服务器的 ftp 会话（与 open 对应）
delete remote-file	删除远程主机文件
debug[debug-value]	设置调试方式，显示发送至远程主机的每条命令
dir/ls[remote-dir][local-file]	显示远程主机目录，并将结果存入本地文件 local-file
disconnection	同 close
get remote-file[local-file]	将远程主机的文件 remote-file 传至本地硬盘的 local-file
lcd[dir]	将本地工作目录切换至 dir
mdelete[remote-file]	删除远程主机文件
mget remote-files	传输多个远程文件
mkdir dir-name	在远程主机中建一目录
mput local-file	将多个文件传输至远程主机
open host[port]	建立指定 ftp 服务器连接，可指定连接端口
passive	进入被动传输方式（在这种模式下，数据连接是由客户端程序发起的）
put local-file[remote-file]	将本地文件 local-file 传送至远程主机
reget remote-file[local-file]	类似于 get，但若 local-file 存在，则从上次传输中断处续传
size file-name	显示远程主机文件的大小
system	显示远程主机的操作系统类型

例如，使用 ftp 命令访问“ftp://ftp.kernel.org”站点，可以使用命令：

```
# ftp ftp.kernel.org
```

注意：若需要匿名登录，则在“Name (**.**.**.**):”处输入 anonymous，在“Password:”处输入自己的 E-mail 地址。若要传送二进制文件，务必要把模式改为 bin。

3.3.7　其他命令

1. **帮助命令（man）**

对于绝大部分 Linux 终端用户和 C 语言程序员而言，经常需要查询一些命令或函数的具体使用方法，此时可以使用 Linux 自带的 man 帮助文件命令。

只要在命令 man 后输入想要获取的命令的名称（如 ls），man 就会列出一份完整的说明，其内容包括命令语法、各选项的意义及相关命令等。该命令使用方式如下：

```
man　[选项]　命令名称
```

man 命令选项说明如表 3.27 所示。

表 3.27　man 命令选项说明

选　项	说　明
-f	只显示出命令的功能而不显示其中详细的说明文件
-w	不显示手册页，只显示将被格式化和显示的文件所在位置
-a	显示所有的手册页，而不是只显示第一个
-E	在每行的末尾显示$符号

2．shell 帮助命令（help）

help 命令用于查看所有 shell 命令。用户可以通过该命令寻求 shell 命令的用法，只需在所查找的命令后输入 help 命令，就可以看到所查命令的内容了。例如，输入 cd –help 便可查看 cd 命令的使用方法。

info 命令用来获取相关命令的详细使用方法，例如，info ls 可以获取使用 info 的详细信息。

3．文件查找命令（whereis）

和 find 命令不同，whereis 命令用来定位可执行文件、源代码文件、帮助文件在文件系统中的位置。例如，最常用的 ls 命令是在/bin 这个目录下的。如果希望知道某个命令存在哪一个目录下，可以用 whereis 命令来查询。该命令的使用方式如下：

```
whereis　[选项]　命令名
```

whereis 命令选项说明如表 3.28 所示。

表 3.28　whereis 命令选项说明

选　项	说　明
-b	定位可执行文件
-m	定位帮助文件
-s	定位源代码文件
-u	搜索默认路径下除可执行文件、源代码文件、帮助文件以外的其他文件
-B	指定搜索可执行文件的路径
-M	指定搜索帮助文件的路径

4．关机和重启命令

由于 Linux 是一种多用户、多任务操作系统，因此在切断计算机电源之前，必须先关闭 Linux 系统。决不能不执行关机进程就切断计算机电源，这样做会导致保存在内存缓冲区的磁盘数据来不及写回磁盘，从而破坏文件系统。下面介绍与关机和重启计算机有关的命令。

shutdown 命令可以安全地关闭或重启 Linux 系统，它在系统关闭之前给系统上的所有登录用户提示一条警告信息。该命令还允许用户指定一个时间参数，可以是一个精确的时间，也可以是从现在开始的一个时间段。精确时间的格式是 hh:mm，表示小时和分钟；时间段由“+”和分钟数表示。系统执行该命令后，会自动进行数据同步的工作。该命令使用方式如下：

```
shutdown [选项] [时间] [警告信息]
```

shutdown 命令选项说明如表 3.29 所示。

表 3.29　shutdown 命令选项说明

选　项	说　明
-k	并不真正关机，而只是发出警告信息给所有用户
-r	关机后立即重新启动
-h	关机后不重新启动
-c	取消一个已经运行的 shutdown

注意：关机命令需要 root 权限。

halt 是最简单的关机命令，其实际上是调用 shutdown -h 命令。halt 执行时，杀死应用进程，文件系统写操作完成后就会停止内核。该命令的使用方式如下：

```
halt [选项]
```

halt 命令选项说明如表 3.30 所示。

表 3.30　halt 命令选项说明

选　项	说　明
-n	在关机前不做将内存资料写回硬盘的动作
-w	并不会真的关机，只是把记录写到 /var/log/wtmp 文件里
-d	不把记录写到 /var/log/wtmp 档案里（-n 这个参数包含了-d）
-f	强迫关机，不调用 shutdown 这个指令
-i	在关机之前先把所有网络相关的装置停止
-p	当关机的时候，顺便做关闭电源（poweroff）的动作。取消一个已经运行的 shutdown

注意：halt 命令同样需要超级用户权限。

reboot 命令用来重新启动计算机。该命令的使用方式如下：

```
reboot [选项]
```

reboot 命令选项说明如表 3.31 所示。

表 3.31　reboot 命令选项说明

选　项	说　明
-n	在关机前不做将内存资料写回硬盘的动作
-w	并不会真的关机，只是把记录写到 /var/log/wtmp 文件里
-d	不把记录写到 /var/log/wtmp 档案里（-n 这个参数包含了-d）
-f	强迫关机，不调用 shutdown 这个指令
-i	在关机之前先把所有网络相关的装置停止

第二部分

在 ARM 处理器系统上移植 Linux 操作系统

第4章

移植和使用嵌入式系统的引导软件（Bootloader）

当嵌入式系统的硬件开发完成之后，接下来的工作就是代码开发，此时面临的一个任务就是如何将编写好的代码下载到嵌入式硬件系统上让其能正常地开始运行，本章将介绍这个过程，涉及的内容有：

- 嵌入式系统的裸机开发；
- 系统引导软件（Bootloader）基础；
- 最常用的系统引导软件 U-Boot 的介绍；
- 如何在安装好 U-Boot 的硬件系统上进行程序下载。

4.1 嵌入式系统的软件开发

嵌入式系统的软件开发环境由几个部分组成：裸机开发环境、嵌入式操作系统移植环境、嵌入式操作系统下的开发环境。

- 裸机开发环境：直接在嵌入式硬件系统进行软件开发所需要的软件环境，用户所编写的代码将会直接被编译生成嵌入式处理器可以直接运行的指令，然后通过对应的编程工具刻录进处理器直接运行。在嵌入式系统硬件调试中常常需要使用这种开发环境来确定硬件系统是否存在问题，此外类似 Cortex-M0、Cortex-M3 等不能/通常不运行操作系统处理器也会直接使用此种裸机开发环境。
- 嵌入式操作系统移植环境：将某个操作系统移植到嵌入式系统上所需要的软件，通常包括引导程序的开发工具和环境、嵌入式系统的裁剪编译环境、文件系统的裁剪编译环境等。
- 嵌入式操作系统下的开发环境：编写嵌入式操作系统可执行代码的开发环境。

4.1.1 进行裸机开发

所谓裸机开发是指在没有安装操作系统的嵌入式硬件系统上直接编写并下载用户代码

的过程。用于裸机开发的 IDE 通常有 ADS、IAP 和 MDK，配合 IDE 使用的硬件有 JTAG 调试器和 J-link 等，后者和 MDK 具有良好的接口。

1. **裸机开发的流程**

裸机开发的流程通常包括编码、编译、下载和调试几部分。

- 编码：用代码编辑器（可以是 IDE 自带的也可以是普通文本编辑器）编写代码的过程。
- 编译：使用代码对应的编译器将用户编写的代码生成嵌入式处理器可执行指令的过程。
- 下载：把可执行的代码下载到嵌入式系统的 Flash 中的过程，如果此时 Flash 是空白的（没有 Bootloader）则可能需要使用一定的工具，如可以使用 JTAG 刻录，也可以使用 SD 卡启动或使用处理器内部自带的 Bootloader 进行下载；如果此时 Flash 里已经有了 Bootloader 则通常可以使用该 Bootloader 通过 USB/串口等方式直接下载。
- 调试：通过 JTAG/串口等硬件接口对当前嵌入式系统的工作状态进行测试并且反馈的过程。

2. **裸机开发环境 MDK 介绍**

嵌入式系统下最常用的裸机开发环境是 MDK，其是 Keil 公司（2007 年被 ARM 公司收购）推出的针对各种嵌入式处理器的软件开发工具，可以用于开发 ARM7、ARM9 和最新的 Cortex-M、Cortex-R 系列处理器，目前最新版的 MDK 是 4.0。

如图 4.1 所示的 J-Link 是 SEGGER 公司为支持仿真 ARM 内核芯片推出的 JTAG 仿真器，和 MDK 集成开发环境连接方便，可以支持 ARM7、ARM9、ARM11、Cortex-M0～Cortex-M4 及 Cortex-A4～Cortex-A8 等内核芯片的仿真，需要注意的是在 MDK 下使用时需要安装相应的驱动，之后系统就可以自动识别，支持 64 位的 Windows 7 操作系统。

图 4.1　J-Link 硬件调试器

注意：Keil μVision 是一个 IDE 开发环境，类似水杯，而且这个水杯可以装不同的饮料；编译器和链接器则是饮料，不同的饮料有不同的口味，而 MDK 就是装了 ARM 饮料的水杯。

3. **【应用实例】——在 MDK 中进行裸机开发**

以下用一个实例来介绍在 MDK 中建立应用代码并且进行刻录调试的过程。

（1）在 MDK 中新建一个工程，如图 4.2 所示。

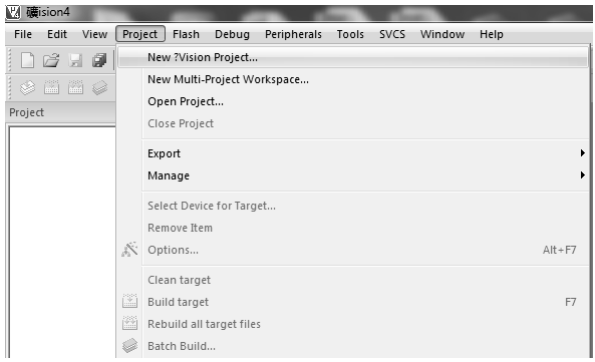


图 4.2　在 MDK 中新建一个工程

（2）建立一个名称为“test”的目录，并且新建一个名称为“test”的工程文件，如图 4.3 所示。

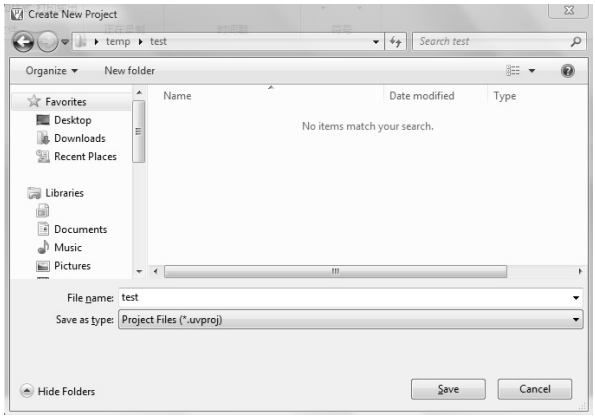


图 4.3　新建一个工程文件

（3）选择项目对应的处理器，此时选择 Samsung 公司的 S3C2440A 后单击 OK 按钮，并且将“S3C2440.s”启动代码加入工程文件中，如图 4.4 所示。

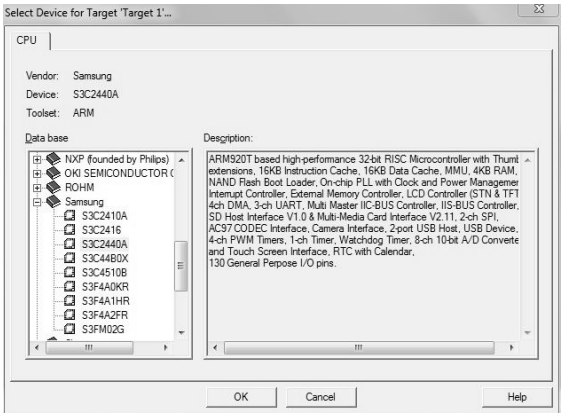


图 4.4　选择工程文件对应的处理器

（4）根据嵌入式系统的实际硬件情况对“S3C2440.s”启动文件进行一定的配置，打开文件后选择编辑界面下方的“Configuration wizard”进行配置，如图 4.5 所示。

（5）新建一个 C 语言源文件，在该文件中进行编码输入，然后将该源文件加入项目的工程文件，如图 4.6 所示。

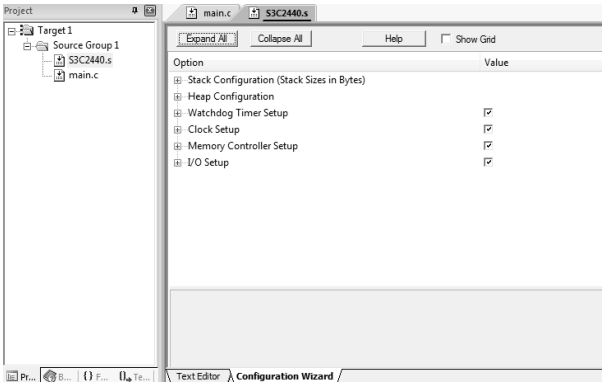


图 4.5　对启动文件进行配置

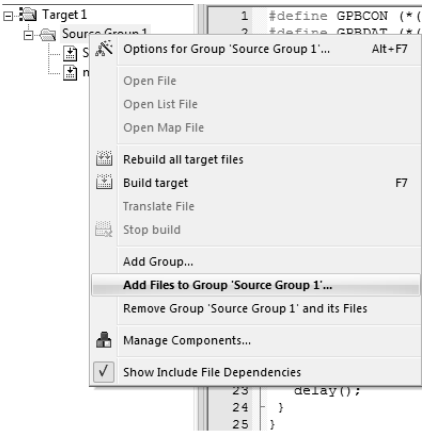


图 4.6　新建 C 语言源文件并且加入工程项目

（6）对工程项目的目标文件进行配置以供生成.Hex 可执行文件，此时需要修改如图 4.7～4.9 所示的部分，包括 Output，不选择 Utilities 选项卡中的 Update Target before Debugging 复选框，选择编程工具为 J-LINK/J-TRACE ARM，然后选择要刻录到的 Flash 型号的编程算法。

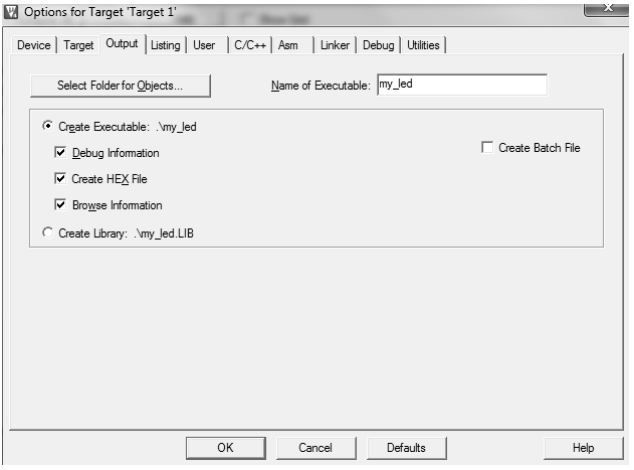


图 4.7　选择 Output 输出文件

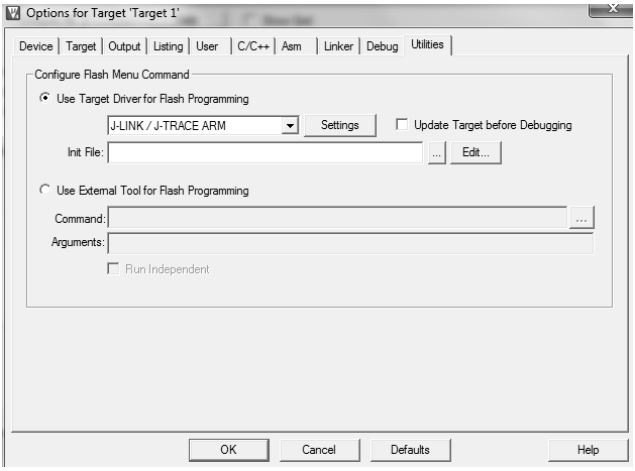


图 4.8　选择编程工具

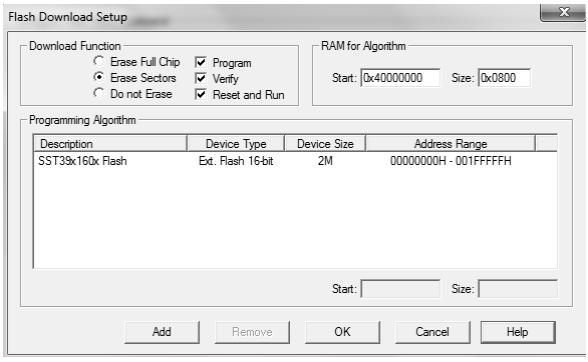


图 4.9　选择 Flash 器件

（7）此时可以进行重建，然后生成对应的.Hex 文件，然后连接好 J-LINK，单击 Download 按钮即可进行下载操作。

注意：在实际应用中，常常将可执行文件放入 SDRAM 中运行和调试，MDK 同样支持通过 J-LINK 进行相应的调试操作。

例 4.1 是使用 MDK 编译的按键扫描的 C 语言代码的实例片段。

【例 4.1】 按键扫描的应用代码。

```
/*************************************************************
4 Key Scan
*************************************************************/
#include "def.h"
#include "option.h"
#include "2440addr.h"
#include "2440lib.h"
#include "2440slib.h"
```

```
#define LED1                (1<<5)          // rGPB[5] =1 ;
#define LED2                (1<<6)          // rGPB[5] =1 ;
#define LED3                (1<<7)          // rGPB[5] =1 ;
#define LED4                (1<<8)          // rGPB[5] =1 ;
/*****************************************************************************
4 个用户按键
四个输入引脚：
                            EINT0   -----( GPF0   )----INPUT---K1
                            EINT2   -----( GPF2   )----INPUT---K2
                            EINT11 -----( GPG3   )----INPUT---K3
                            EINT19 -----( GPG11 )----INPUT---K4
*****************************************************************************/
U8 Key_Scan( void )
{
    Delay( 80 ) ;
    if(         (rGPFDAT&(1<< 0)) == 0 )
    {
            rGPBDAT = rGPBDAT &～(LED1);                    //亮 LED1
            return 4;
    }
    else if( (rGPFDAT&(1<< 2)) == 0 )
    {
            rGPBDAT = rGPBDAT &～(LED2);                    //亮 LED2
            return 3;
    }
    else if( (rGPGDAT&(1<< 3)) == 0 )
    {
            rGPBDAT = rGPBDAT &～(LED3);                    //亮 LED3
            return 2 ;
    }
    else if( (rGPGDAT&(1<< 11)) == 0 )
    {
            rGPBDAT = rGPBDAT &～(LED4);                    //亮 LED4
            return 1 ;
    }
    else
    {
            rGPBDAT = rGPBDAT &～0x1e0|0x1e0;               //LED[8:5] => 1;
            return 0xff;
    }
}
static void __irq Key_ISR(void)
{
    U8 key;
    U32 r;
    EnterCritical(&r);
```

```
    if(rINTPND==BIT_EINT8_23) {
        ClearPending(BIT_EINT8_23);
        if(rEINTPEND&(1<<11)) {
        //Uart_Printf("eint11\n");
            rEINTPEND |= 1<< 11;
        }
        if(rEINTPEND&(1<<19)) {
        //      Uart_Printf("eint19\n");
            rEINTPEND |= 1<< 19;
        }
    }
    if(rINTPND==BIT_EINT0) {
        //Uart_Printf("eint0\n");
        ClearPending(BIT_EINT0);
    }
    if(rINTPND==BIT_EINT2) {
        //Uart_Printf("eint2\n");
        ClearPending(BIT_EINT2);
    }
    key=Key_Scan();
    if( key == 0xff )
        Uart_Printf( "Interrupt occur... Key is released!\n") ;
    else
        Uart_Printf( "Interrupt occur... K%d is pressed!\n", key) ;
    ExitCritical(&r);
}
void KeyScan_Test(void)
{
    Uart_Printf("\nKey Scan Test, press ESC key to exit !\n");
    rGPGCON = rGPGCON & (~((3<<22)|(3<<6))) | ((2<<22)|(2<<6)) ;          //GPG11,3 set EINT
    rGPFCON = rGPFCON & (~((3<<4)|(3<<0))) | ((2<<4)|(2<<0)) ;           //GPF2,0 set EINT
    rEXTINT0 &=~(7|(7<<8));
    rEXTINT0 |= (0|(0<<8));         //set eint0,2 falling edge int
    rEXTINT1 &=~(7<<12);
    rEXTINT1 |= (0<<12); //set eint11 falling edge int
    rEXTINT2 &=~(0xf<<12);
    rEXTINT2 |= (0<<12); //set eint19 falling edge int
    rEINTPEND |= (1<<11)|(1<<19);               //clear eint 11,19
    rEINTMASK &=~((1<<11)|(1<<19));         //enable eint11,19
    ClearPending(BIT_EINT0|BIT_EINT2|BIT_EINT8_23);
    pISR_EINT0 = pISR_EINT2 = pISR_EINT8_23 = (U32)Key_ISR;
    EnableIrq(BIT_EINT0|BIT_EINT2|BIT_EINT8_23);
     while( Uart_GetKey() != ESC_KEY ) ;
     DisableIrq(BIT_EINT0|BIT_EINT2|BIT_EINT8_23);
}
```

4.1.2　在嵌入式操作系统下进行开发

本书介绍的重点是在嵌入式系统（Linux）下进行开发，和裸机开发不同，在进行开发之前必须先在嵌入式硬件系统上移植相应的操作系统。嵌入式操作系统下的软件开发通常会使用交叉编译环境，在 PC 上完成相应的代码开发，然后通过相应的下载方式下载到嵌入式操作系统下。

操作系统的移植是指当嵌入式硬件开发已经完成且保证没有硬件错误之后将一个目标操作系统移植到硬件系统上并且运行的过程，其目标是在硬件系统上运行一个操作系统。

以 Linux 操作系统为例来介绍在嵌入式操作系统中进行开发的方法，大概可以分为以下 5 个步骤，其详细描述及对应本书的章节说明如下。

（1）配置和编译 Bootloader，然后将 Bootloader 下载到开发板，其可以初始化硬件设备，建立内存空间的映射表，对操作系统进行引导——在第 4.2～4.4 节中进行详细介绍。

（2）下载操作系统的源代码，建立交叉编译环境，配置和编译操作系统内核，并且根据硬件系统的特点对其进行相应的裁剪和配置，然后通过 Bootloader 将完成的操作系统下载到目标板上——在本书第 5 章进行详细介绍。

（3）为 NAND FLASH 移植文件系统，通常来说是 YAFFS2 文件系统，这样才能形成完整的操作系统应用环境——在第 6 章中进行详细介绍。

（4）建立嵌入式系统和开发环境的数据交互通道（可以是 ftp，可以是根文件映射）以便于将在 PC 上编译好的代码下载到嵌入式系统上——在第 7.5 小节中进行介绍。

（5）在 PC 上根据嵌入式操作系统的特点编写相应的代码——在第 7～12 章中进行介绍。

4.2　嵌入式系统的引导软件基础

如果嵌入式系统不需要运行操作系统，可以直接使用 MDK 集成开发环境来进行可执行文件的开发，但是如果希望进行操作系统的移植或更加方便的下载，此时应该使用嵌入式系统引导软件（Bootloader），其亦称引导加载程序，是嵌入式系统加电后运行的第一段软件代码，是整个系统执行的第一步。

4.2.1　Bootloader 介绍

Bootloader 不仅应用于嵌入式系统中，在通常的 PC 系统中，其引导加载的程序是由 BIOS 和位于硬盘 MBR 中的 OS Bootloader 完成的，这里常见的 OS Bootloader 有 LILO 和 GRUB 等。BIOS 在完成硬件检测和资源分配后，将硬盘 MBR 中的 Bootloader 读到系统的 RAM 中，然后将控制权交给 OS Bootloader。Bootloader 的主要运行任务就是将内核映像从硬盘上读到 RAM 中，然后跳转到内核的入口点去运行，即开始启动操作系统。

出于经济性、价格方面的考虑，虽然一些嵌入式处理器中会嵌入一段短小的启动程序，但是通常并没有像 BIOS 那样的固件程序，所以相对于 PC 上的 OS Bootloader 所做的工作，嵌入式系统的 Bootloader 不仅要将内核映像从硬盘上读到 RAM 中，然后引导启动操作

系统内核，还需要完成 BIOS 所做的硬件检测和资源分配工作。可见，嵌入式系统中的 Bootloader 比 PC 中的 Bootloader 更强大、功能更多。例如，在一个基于 ARM920T Core 的嵌入式系统中（如 S3C2440），系统在上电或复位时通常都从地址 0x00000000 处开始执行，而以处理器为核心的嵌入式系统，通常都有某种类型的固态存储设备（如 EEPROM、FLASH 等）被映射到这个预先设置好的地址上。在系统加电复位后，处理器将首先处理 Bootloader 程序。因此 Bootloader 是系统加电后、操作系统内核或用户应用程序运行之前，首先运行的一段代码，通过这段代码，可以初始化硬件设备，建立内存空间的映射图（有的处理器没有内存映射功能），从而将系统的软/硬件环境设定在一个合适的状态，为最终调用操作系统内核、运行用户程序准备好正确的环境。对于嵌入式系统，尽管有的使用操作系统，有得不使用操作系统（如功能简单、仅包括应用程序的系统），但在系统启动时，都必须运行 Bootloader，为系统运行准备好软/硬件环境。

4.2.2 基于 Bootloader 的嵌入式架构

由于嵌入式系统平台是一种软/硬件结合的平台。它和普通的单片机系统最大的区别就是嵌入式系统平台上具有专用的嵌入式操作系统，如前面介绍的 Linux 操作系统、Wince 操作系统、Vxworks 操作系统等。由于具有专用的操作系统支持，所以在嵌入式系统平台上开发应用软件和在普通 PC 上一样方便、快捷。但是所有这些软件，包括应用软件和操作系统软件，它们都是离不开 Bootloader 的，从图 4.10 中可以看出 Bootloader 在嵌入式软件架构中的作用。通常一个嵌入式系统软件架构可以分为四个层次：用户应用程序、文件系统、嵌入式操作系统内核和引导加载程序（即 Bootloader）。

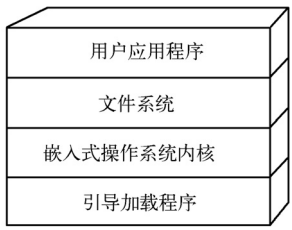


图 4.10　嵌入式系统软件架构

从下往上，各层次完成的主要功能如下。

- 引导加载程序：固化在硬件 Flash 上的一段引导代码，用于完成硬件的一些基本配置，引导嵌入式操作系统内核启动。
- 嵌入式操作系统内核：包括特定于某嵌入式硬件平台的定制操作系统内核及内核的启动参数等。
- 文件系统：包括根文件系统和建立在 Flash 内存设备上的文件系统。通常用 ram disk 或 yaffs 来作为文件系统，包括固化在固件（firmware）中的 boot 代码（可选）和 Bootloader 两大部分。
- 用户应用程序：特定于用户的应用程序，有时在用户应用程序和内核层之间可能还会包括一个嵌入式图形用户界面。常用的嵌入式 GUI 有 MicroWindows、MiniGUI、QT/ Embeded 等。

4.2.3　Bootloader 的工作模式

一般 Bootloader 包含两种不同的工作模式：启动加载模式（Bootloading）和下载模式（Downloading）。其实对于开发人员，这种区分是非常重要的，但是对于最终的用户来说，就不需要区分了，只需知道 Bootloader 的作用是用来加载操作系统就可以了，当然就不存在所谓的启动加载模式和下载工作模式的区别了。

- 启动加载模式（Bootloading）又称自主模式，是指 Bootloader 从目标机上的某个固件存储设备上将操作系统加载到 RAM 中运行，整个过程没有用户的介入。这种模式是 Bootloader 的正常工作模式。当嵌入式产品最终发布时，Bootloader 就被默认在这种模式下。
- 下载模式（Downloading）下，目标机上的 Bootloader 将通过串口、网络或 USB 等其他通信手段从主机下载文件，如下载内核镜像、根文件系统镜像等，从主机下载的文件通常首先被 Bootloader 保存到目标机的 RAM 中，然后被 Bootloader 写到目标机的 Flash 内固态存储设备中。Bootloader 的这种模式通常在第一次安装内核与根文件系统时使用。此外，以后的系统更新也会使用 Bootloader 的这种工作模式。工作在这种模式下的 Bootloader 通常都会向它的中端用户提供一个简单的命令接口。

4.2.4　Bootloader 的启动方式

Bootloader 的主要功能是引导操作系统启动，它的启动方式一般有网络启动、磁盘启动和 Flash 启动三种启动方式，具体描述如下。

1．网络启动

在网络启动方式下，Bootloader 通过以太网接口远程下载 Linux 内核映像或文件系统，如图 4.11 所示。这种方式的开发板不需要配置较大的存储介质，与无盘工作站有点类似。但是使用这种启动方式之前，需要把 Bootloader 安装到板上的 EPROM 或 Flash 中。交叉开发环境就是以网络启动方式建立的。这种方式对于嵌入式系统开发来说非常重要。当然，采用这种启动方式也有一定的前提条件。

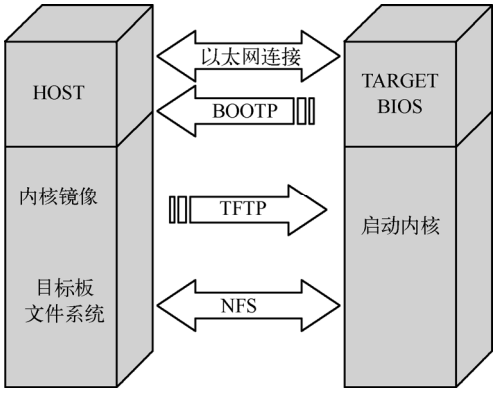


图 4.11　网络启动方式

- 目标板有串口、以太网接口或其他连接方式。串口一般可以作为控制台，同时可以用来下载内核影像和 RAMDISK 文件系统。但是，串口通信传输速率过低，不适合用来挂接 NFS 文件系统。所以以太网接口成为通用的互连设备，一般的开发板都可以配置 10Mb/s 以太网接口。对于 PDA 等手持设备来说，以太网的 RJ-45 接口显得大了些，而 USB 接口，特别是 USB 的迷你接口，尺寸非常小。对于开发的嵌入式系统，可以把 USB 接口虚拟成以太网接口来通信。
- 开发主机和开发板两端都需要相应接口的驱动程序。如果采用串口，则两端要安装串口的驱动程序；如果采用 USB 等高速的接口形式，则开发主机和开发板两端同样需要先安装驱动程序。
- 另外，还要在服务器上配置启动相关网络服务。Bootloader 下载文件一般都使用 TFTP 网络协议，还可以通过 DHCP 的方式动态配置 IP 地址。DHCP/BOOTP 服务为 Bootloader 分配 IP 地址，配置网络参数，然后才能够支持网络传输功能。如果 Bootloader 可以直接设置网络参数，就可以不使用 DHCP，而是用 TFTP 服务，TFTP 服务同样为 Bootloader 客户端提供文件下载功能，把内核映像和其他文件放在指定目录下。这样 Bootloader 可以通过简单的 TFTP 协议远程下载内核映像到内存。

大部分引导程序都支持网络启动方式。例如，BIOS 的 PXE（Preboot Execution Environment）功能就是网络启动方式。

2．磁盘启动

磁盘启动方式主要用在 PC 的 BIOS 中，如传统的 Linux 系统运行在台式机或服务器上，这些计算机一般都使用 BIOS 引导，并且使用磁盘作为存储介质。如果进入 BIOS 设置菜单，可以探测处理器、内存、硬盘等设备，可以设置 BIOS 从软盘、光盘或某块硬盘启动。但 BIOS 并不直接引导操作系统。这样在硬盘的主引导区还需要一个 Bootloader。这个 Bootloader 可以通过磁盘启动方式从磁盘文件系统中把操作系统引导起来。

Linux 传统上是通过 LILO（Linux Loader）引导的，后来又出现了 GNU 的软件 GRUB（GRand Unified Bootloader）。GRUB 是 GNU 计划的主要 Bootloader。GRUB 最初是由 Erich Boleyn 为 GNU Mach 操作系统撰写的引导程序。后来由 Gordon Matzigkeit 和 Okuji Yoshinori 接替 Erich 的工作，继续维护和开发 GRUB。这两种 Bootloader 广泛应用在 X86 的 Linux 系统上，熟悉它们有助于配置多种系统引导功能。另外，GRUB 能够使用 TFTP 和 BOOTP 或 DHCP 通过网络启动，这种功能对于系统开发过程很有用。

除了传统的 Linux 系统上的引导程序以外，还有其他一些引导程序，也可以支持磁盘引导启动。例如，LoadLin 可以从 DOS 下启动 Linux，ROLO、LinuxBIOS、U-Boot 也支持这种功能。

3．Flash 启动

Flash 启动方式通常有两种：一种是可以直接从 Flash 启动；另一种是可以将压缩的内存映像文件从 Flash（为节省 Flash 资源、提高速度）中复制、解压到 RAM，再从 RAM 启

动。Flash 存储介质有很多类型，包括 NOR Flash、NAND Flash 等。其中 NOR Flash 使用最为普遍。

因为 NOR Flash 支持随机访问，所以代码可以直接在 Flash 上执行。Bootloader 一般是存储在 Flash 芯片上的，Linux 内核映像和 RAMDISK 也是存储在 Flash 上的。通常需要把 Flash 分区使用，每个区的大小应该是 Flash 擦除大小的整数倍。如图 4.12 所示是 Bootloader 和内核映像及文件系统的分区表。

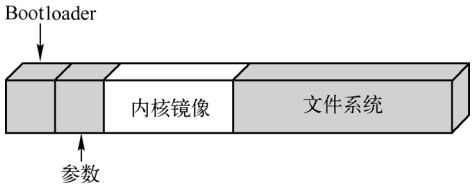


图 4.12　Flash 存储示意图

Bootloader 一般放在 Flash 的底端或顶端，这是根据处理器的复位向量设置的，要使 Bootloader 的入口位于处理器上电执行第一条指令的位置。接下来是分配参数区，这里可以作为 Bootloader 的参数保存区域。再下来是内核映像区。Bootloader 引导 Linux 内核，就是要从这个地方把内核映像解压到 RAM 中去，然后跳转到内核映像入口执行。最后是文件系统区，如果使用 Ramdisk 文件系统，则需要 Bootloader 把它解压到 RAM 中；如果使用 YAFFS2 文件系统，将直接挂接为根文件系统。

最后还可以分出一些数据区，这要根据实际需要和 Flash 的大小来考虑。这些分区是开发者定义的，Bootloader 一般直接读/写对应的偏移地址。到了 Linux 内核空间，可以配置成 MTD 设备来访问 Flash 分区。但是，有的 Bootloader 也支持分区的功能，如 Redboot 可以创建 Flash 分区表，并且内核 MTD 驱动可以解析出 Redboot 的分区表。

除了 NOR Flash，还有 NAND Flash、Compact Flash、DiskOnChip 等。这些 Flash 具有芯片价格低、存储容量大的特点。但是这些芯片一般通过专用控制器的 I/O 方式来访问，不能随机访问，因此引导方式与 NOR Flash 也不同。在这些芯片上，需要配置专用的引导程序。通常，这种引导程序起始的一段代码就把整个引导程序复制到 RAM 中运行，从而实现自举启动，这与从磁盘上启动有些相似。

4.2.5　Bootloader 的启动流程

当电源打开时，系统会执行 ROM（较多的是 Flash）里面的 Bootloader 启动代码。启动代码是用来初始化电路及用来为高级语言编写的软件做好运行前准备的一小段汇编语言，在商业实时操作系统中，启动代码部分一般被称为板级支持包，英文缩写为 BSP，其主要功能就是将电路初始化并为高级语言编写的软件运行做准备。Bootloader 启动的具体流程如图 4.13 所示，主要的过程如下。

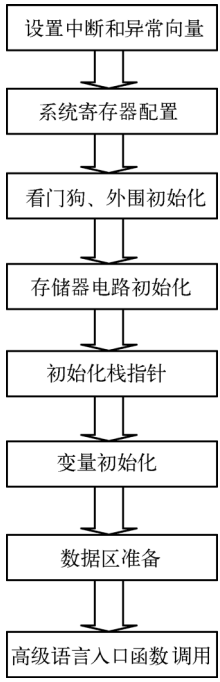


图 4.13　Bootloader 启动流程

（1）启动代码的第一步是设置中断和异常向量。

（2）完成系统启动所必需的最小配置。某些处理器芯片包含一个或几个全局寄存器，这些寄存器必须在系统启动的最初进行配置。

（3）设置看门狗。用户设计的部分外围电路如果必须在系统启动时初始化，就可以放在这一步。

（4）配置系统所使用的存储器，包括 Flash、SRAM 和 DRAM 等，并为它们分配地址空间。如果系统使用了 DRAM 或其他外设，就需要设置相关的寄存器，以确定其刷新频率、数据总线宽度等信息，并初始化存储器系统。有些芯片可通过寄存器编程初始化存储器系统，而对于较复杂的系统，通常集成有 MMU 来管理内存空间。

（5）为处理器的每个工作模式设置栈指针。ARM 处理器有多种工作模式，每种工作模式都需要设置单独的栈空间。

（6）变量初始化。这里的变量指的是在软件中定义的已经赋好初值的全局变量，启动过程中需要将这部分变量从只读区域（也就是 Flash）复制到读写区域中，因为这部分变量的值在软件运行时有可能重新赋值。还有一种变量不需要处理，就是已经赋好初值的静态全局变量，这部分变量在软件运行过程中不会改变，因此可以直接固化在只读的 Flash 或 E^2PROM 中。

（7）数据区准备。对于软件中所有未赋初值的全局变量，启动过程中需要将这部分变量所在区域全部清零。

（8）最后一步是调用高级语言入口函数，如 main 函数等。

系统启动代码完成基本软/硬件环境初始化后，在有操作系统的情况下，启动操作系统、启动内存管理、设置任务调度、加载驱动程序等，最后执行应用程序或等待用户命令；对于没有操作系统的系统，直接执行应用程序或等待用户命令。

4.2.6　常见的 Bootloader

Bootloader 不仅和嵌入式系统的处理器架构相关，还和具体的嵌入式系统硬件结构及需要运行的操作系统相关。在 Linux 操作系统下，有许多现成的 Bootloader 可用，如表 4.1 所示。

表 4.1　Linux 操作系统下的常见 Bootloader

Bootloader	监控程序	说明	架构					
			X86	ARM	PowerPC	MIPS	M68k	SuperH
LILO	否	Linux 主要的磁盘引导加载程序	*					
GRUB	否	LILO 的 GNU 版后继者	*					
ROLO	否	不需要 BIOS，可直接从 ROM 加载 Linux	*					
Loadlin	否	从 DOS 加载 Linux	*					
Etherboot	否	从 ETHERNET 卡启动系统的 Romable loader	*					
LinuxBIOS	否	以 Linux 为基础的 BIOS 替代品	*					
Compaq 的 bootldr	是	主要用于 Compaq iPAQ 的多功能加载程序		*				
blob	否	来自 LART 硬件计划的加载程序		*				
PMON	是	Agenda VR3 中所使用的加载程序				*		
sh-boot	否	LinuxSH 计划的主要加载程序						*
U-Boot	是	以 PPCBoot 和 ARMBoot 为基础的通用加载程序	*	*	*	*		*
RedBoot	是	以 eCos 为基础的加载程序	*	*	*	*	*	*
Vivi	是	适用于 SAMSUNG 公司 ARM9 微处理器		*				

4.3　【应用实例】——移植 Bootloader 软件 U-Boot

U-Boot（Universal Boot Loader）是遵循 GPL 条款的开放源码项目，本节将介绍其特点和在嵌入式系统上对其进行部署的方法。

4.3.1　U-Boot 的特点和功能

U-Boot 是从 FADSROM、8xxROM、PPCBOOT 逐步发展演化而来的。其源码目录、编译形式与 Linux 内核很相似，事实上，不少 U-Boot 源码就是相应的 Linux 内核源程序的简化，尤其是一些设备的驱动程序，从 U-Boot 源码的注释中能体现这一点。但是 U-Boot 不仅支持嵌入式 Linux 系统的引导，还支持 NetBSD、VxWorks、QNX、RTEMS、ARTOS、LynxOS 等嵌入式操作系统。其目前要支持的目标操作系统是 OpenBSD、NetBSD、FreeBSD、4.4BSD、Linux、SVR4、Esix、Solaris、Irix、SCO、Dell、NCR、VxWorks、LynxOS、pSOS、QNX、RTEMS、ARTOS。这是 U-Boot 中 Universal 的一层含义，另外一层含义则是 U-Boot 除了支持 PowerPC 系列的处理器外，还能支持 MIPS、x86、ARM、

NIOS、XScale 等诸多常用系列的处理器。这两个特点正是 U-Boot 项目的开发目标，即支持尽可能多的嵌入式处理器和嵌入式操作系统。到目前为止，U-Boot 对 PowerPC 系列处理器的支持最为丰富，对 Linux 的支持最完善。

U-Boot 具有以下特点。

- 源代码开放。
- 支持包括 Linux、NetBSD、VxWorks、QNX、RTEMS、ARTOS、LynxOS 在内的多种嵌入式操作系统。
- 支持包括 PowerPC、ARM、x86、MIPS、XScale 多种处理器在内的架构。
- 具有较高的可靠性和稳定性。
- 具有高度灵活的功能设置，适合调试、操作系统不同引导要求和产品发布等需求。
- 提供了包括串口、以太网、SDRAM、Flash、LCD、NVRAM、E^2PROM、RTC、键盘等在内的丰富设备驱动源代码。
- 提供了较为丰富的开发调试文档与强大的网络技术支持。

U-Boot 提供了如下功能。

- 系统引导：支持 NFS 挂载、RAMDISK（压缩或非压缩）形式的根文件系统。支持 NFS 挂载，并从 Flash 中引导压缩或非压缩系统内核。
- 基本辅助功能：强大的操作系统接口功能；可灵活设置、传递多个关键参数给操作系统，适合系统在不同开发阶段的调试要求与产品发布，尤其对 Linux 支持最为强劲；支持目标板环境参数多种存储方式，如 Flash、NVRAM、EEPROM；CRC32 校验，可校验 Flash 中内核、RAMDISK 映像文件是否完好。
- 设备驱动功能：串口、SDRAM、Flash、以太网、LCD、NVRAM、EEPROM、键盘、USB、PCMCIA、PCI、RTC 等驱动支持。
- 上电自检功能：SDRAM、Flash 大小自动检测；SDRAM 故障检测；CPU 型号。

此外，U-Boot 还支持 XIP 内核引导等特殊功能。

4.3.2 U-Boot 的源代码结构分析

U-Boot 是 Bootloader 的一种，其使用和嵌入式处理器及嵌入式系统硬件关联非常紧密，所以用户应该对其源代码有一定的了解以便根据自己的硬件特点进行定制。

1．U-Boot 的结构

U-Boot（基于 1.1.2 版本）的内部结构如图 4.14 所示，其包括了如下一些模块。

- board：和一些已有开发板有关的代码，如 makefile 和 U-Boot.lds 等都和具体开发板的硬件和地址分配有关。
- common：与体系结构无关的代码，用来实现各种命令的 C 程序。
- cpu：包含 CPU 相关代码，其中的子目录都以 U-BOOT 所支持的 CPU 为名，如有子目录 arm926ejs、

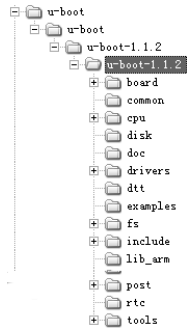


图 4.14　U-Boot 的源代码结构示意图

mips、mpc8260 和 nios 等，每个特定的子目录中都包括 cpu.c 和 interrupt.c，start.S 等。其中 cpu.c 初始化 CPU、设置指令 Cache 和数据 Cache 等；interrupt.c 设置系统的各种中断和异常，如快速中断、开关中断、时钟中断、软件中断、预取中止和未定义指令等；汇编代码文件 start.S 是 U-BOOT 启动时执行的第一个文件，它主要是设置系统堆栈和工作方式，为进入 C 程序奠定基础。

- disk：disk 驱动的分区相关代码。
- doc：文档。
- drivers：通用设备驱动程序，如各种网卡、支持 CFI 的 Flash、串口和 USB 总线等。
- fs：支持文件系统的文件，U-BOOT 现在支持 cramfs、fat、fdos、jffs2 和 registerfs 等。
- include：头文件，还有对各种硬件平台支持的汇编文件，系统的配置文件和对文件系统支持的文件。
- net：与网络有关的代码，BOOTP 协议、TFTP 协议、RARP 协议和 NFS 文件系统的实现。
- lib_arm：与 ARM 体系结构相关的代码。
- tools：创建 S-Record 格式文件和 U-BOOT images 的工具。

2．U-Boot 需要修改的代码分析

U-Boot 中的重要代码包括启动文件 start.S、中断处理文件 interrupts.c、处理器操作文件 cpu.c 和嵌入式系统配置文件 memsetup.S。

- 启动文件：该文件位于 cpu/arm920t/start.S，这是 U-Boot 的起始位置。在这个文件中设置了处理器的状态、初始化中断向量和内存时序等，从 Flash 中跳转到定位好的内存位置执行，其对应的代码说明如例 4.2 所示。

【例 4.2】 U-Boot 的启动文件。

```
.globl_start （起始位置：中断向量设置）
_start:        b               reset
        ldr      pc, _undefined_instruction
        ldr      pc, _software_interrupt
        ldr      pc, _prefetch_abort
        ldr      pc, _data_abort
        ldr      pc, _not_used
        ldr      pc, _irq
        ldr      pc, _fiq
_undefined_instruction:     .word undefined_instruction
_software_interrupt:     .word software_interrupt
_prefetch_abort:     .word prefetch_abort
_data_abort:          .word data_abort
_not_used:          .word not_used
_irq:                .word irq
_fiq:                .word fiq
_TEXT_BASE: （代码段起始位置）
.word     TEXT_BASE
```

```
.globl _armboot_start
_armboot_start:
      .word _start
/*
 * These are defined in the board-specific linker script.
 */
.globl _bss_start （BSS 段起始位置）
_bss_start:
        .word __bss_start
.globl _bss_end
_bss_end:
       .word _end
reset: （执行入口）
        /*
        * set the cpu to SVC32 mode;使处理器进入特权模式
        */
        mrs      r0,cpsr
        bic      r0,r0,#0x1f
        orr      r0,r0,#0xd3
        msr      cpsr,r0
relocate:      （代码的重置）                  /* relocate U-Boot to RAM        */
        adr      r0, _start          /* r0 <- current position of code     */
        ldr      r1, _TEXT_BASE          /* test if we run from flash or RAM */
        cmp        r0, r1                    /* don't reloc during debug              */
        beq        stack_setup
        ldr      r2, _armboot_start
        ldr      r3, _bss_start
        sub      r2, r3, r2        /* r2 <- size of armboot                 */
        add      r2, r0, r2        /* r2 <- source end address              */
copy_loop: （复制过程）
        ldmia r0!, {r3-r10}         /* copy from source address [r0]      */
        stmia r1!, {r3-r10}         /* copy to      target address [r1]      */
        cmp     r0, r2                /* until source end addreee [r2]      */
        ble     copy_loop
        /* Set up the stack；设置堆栈   */
stack_setup:
        ldr     r0, _TEXT_BASE          /* upper 128 KiB: relocated uboot     */
        sub     r0, r0, #CFG_MALLOC_LEN      /* malloc area                                    */
        sub     r0, r0, #CFG_GBL_DATA_SIZE /* bdinfo                                        */
clear_bss: （清空 BSS 段）
        ldr      r0, _bss_start       /* find start of bss segment             */
        ldr      r1, _bss_end              /* stop here                                    */
        mov        r2, #0x00000000         /* clear                                        */
clbss_l:str      r2, [r0]             /* clear loop...                         */
        add     r0, r0, #4
        cmp     r0, r1
```

```
        bne     clbss_l
        ldr     pc, _start_armboot
_start_armboot:.word start_armboot
```

- 中断处理文件：interrupts.c 用于中断处理，如打开和关闭中断等，其对应的代码说明如例 4.3 所示。

【例 4.3】 U-Boot 的中断处理文件。

```
#ifdef CONFIG_USE_IRQ
/* enable IRQ interrupts；中断使能函数 */
void enable_interrupts (void)
{
        unsigned long temp;
        __asm__ __volatile__("mrs %0, cpsr\n"
                             "bic %0, %0, #0x80\n"
                             "msr cpsr_c, %0"
                             : "=r" (temp)
                             :
                             : "memory");
}
/ * disable IRQ/FIQ interrupts；中断屏蔽函数
 * returns true if interrupts had been enabled before we disabled them */
int disable_interrupts (void)
{
        unsigned long old,temp;
        __asm__ __volatile__("mrs %0, cpsr\n"
                             "orr %1, %0, #0xc0\n"
                             "msr cpsr_c, %1"
                             : "=r" (old), "=r" (temp)
                             :
                             : "memory");
        return (old & 0x80) == 0;
}
#endif
void show_regs (struct pt_regs *regs)
{
        unsigned long flags;
        const char *processor_modes[] = {
        "USER_26", "FIQ_26",   "IRQ_26",   "SVC_26",
        "UK4_26",   "UK5_26",   "UK6_26",   "UK7_26",
        "UK8_26",   "UK9_26",   "UK10_26", "UK11_26",
        "UK12_26", "UK13_26", "UK14_26", "UK15_26",
        "USER_32", "FIQ_32",   "IRQ_32",   "SVC_32",
        "UK4_32",   "UK5_32",   "UK6_32",     "ABT_32",
        "UK8_32",   "UK9_32",   "UK10_32", "UND_32",
        "UK12_32", "UK13_32", "UK14_32", "SYS_32",
```

```
        };
…
}
/* 在 U-Boot 启动模式下，在原则上要禁止中断处理，所以如果发生中断，当作出错处理 */
void do_fiq (struct pt_regs *pt_regs)
{
        printf ("fast interrupt request\n");
        show_regs (pt_regs);
        bad_mode ();
}
void do_irq (struct pt_regs *pt_regs)
{
        printf ("interrupt request\n");
        show_regs (pt_regs);
        bad_mode ();
}
```

● 处理器操作文件：cpu.c 文件是对处理器进行操作，其对应的代码说明如例 4.4 所示。

【例 4.4】 U-Boot 的处理器操作文件。

```
int cpu_init (void)
{
        /* setup up stacks if necessary；设置需要的堆栈*/
#ifdef CONFIG_USE_IRQ
        DECLARE_GLOBAL_DATA_PTR;
        IRQ_STACK_START=_armboot_start - CFG_MALLOC_LEN - CFG_GBL_DATA_SIZE - 4;
        FIQ_STACK_START = IRQ_STACK_START - CONFIG_STACKSIZE_IRQ;
#endif
        return 0;
}
int cleanup_before_linux (void) /* 准备加载 linux */
{
        /*
          * this function is called just before we call linux
          * it prepares the processor for linux
          *
          * we turn off caches etc ...
          */
        unsigned long i;
        disable_interrupts ();
       /* turn off I/D-cache：关闭 cache */
        asm ("mrc p15, 0, %0, c1, c0, 0":"=r" (i));
        i &=~(C1_DC | C1_IC);
        asm ("mcr p15, 0, %0, c1, c0, 0": :"r" (i));
        /* flush I/D-cache */
        i = 0;
```

```
        asm ("mcr p15, 0, %0, c7, c7, 0": :"r" (i));
        return (0);
}
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
        . = 0x00000000;
              . = ALIGN(4);
        .text           :
        {
          cpu/arm920t/start.o (.text)
          *(.text)
        }
        . = ALIGN(4);
        .rodata : { *(.rodata) }
        . = ALIGN(4);
        .data : { *(.data) }
        . = ALIGN(4);
        .got : { *(.got) }
        __u_boot_cmd_start = .;
        .u_boot_cmd : { *(.u_boot_cmd) }
        __u_boot_cmd_end = .;
        . = ALIGN(4);
        __bss_start = .;
        .bss : { *(.bss) }
        _end = .;
}
```

- 嵌入式配置文件：memsetup.S 文件是用于配置开发板参数的，其对应的代码说明如例 4.5 所示。

【例 4.5】 U-Boot 的嵌入式配置文件。

```
/* memsetup.s */
        /* memory control configuration */
        /* make r0 relative the current location so that it */
        /* reads SMRDATA out of FLASH rather than memory ! */
        ldr         r0, =SMRDATA
        ldr r1, _TEXT_BASE
        sub r0, r0, r1
        ldr r1, =BWSCON    /* Bus Width Status Controller */
        add         r2, r0, #52
0:
        ldr         r3, [r0], #4
        str         r3, [r1], #4
        cmp         r2, r0
```

```
    bne     0b

    /* everything is fine now */
    mov pc, lr

    .ltorg
```

4.3.3 移植 U-Boot

U-Boot 的移植需要根据当前的嵌入式系统的硬件环境来完成，其主要的步骤说明如下，最新的 U-Boot 可以从 http://ftp.denx.de/pub/u-boot/中获得。

1. 修改当前嵌入式系统的硬件类型

阅读 makefile 文件，在 makefile 文件中添加两行，如下所示：

```
GT2440_config: unconfig
    @./mkconfig  $(@:_config=) arm arm920t GT2440
```

其中“arm”表示处理器体系结构的种类，“arm920t”表示处理器体系结构的名称，“GT2440”为主板名称。

在 board 目录中建立 GT2440 目录，并将 U-Boot 目录中 smdk2410 字目录中的内容复制到该目录中。在 include/configs/目录下将 smdk2410.h 复制到该目录中。修改 ARM 编译器的目录名及前缀（都要改成以“GT2440”开头）。

完成之后，可以测试配置。

```
$ make GT2440_config;make
```

2. 修改程序链接地址

在 board/S3C2440 中有一个 config.mk 文件，它用于设置程序链接的起始地址，因为会在 U-Boot 中增加功能，所以留下 6MB 的空间，修改 33F80000 为 33A00000 是为了以后能用 U-Boot 的 go 命令执行修改过的用 loadb 或 tftp 下载的 U-Boot，需要在 board/ S3C2440 的 memsetup.S 中标记符“0:”上加入 5 句：

```
mov r3, pc
ldr r4, =0x3FFF0000
and r3, r3, r4  （以上 3 句得到实际代码启动的内存地址）
aad r0, r0, r3  （用 go 命令调试 u-boot 时，启动地址在 RAM）
add r2, r2, r3  （把初始化内存信息的地址加上实际启动地址）
```

3. 修改禁止处理

中断禁止的部分代码应该改为如下代码。

```
# if defined(CONFIG_S3C2440)
    ldr    r1, =0x7ff
    ldr    r0, =INTSUBMSK
    str    r1, [r0]
```

```
# endif
```

4. 修改启动代码

因为在 GT2440 开发板启动时直接从 Nand Flash 加载代码，所以启动代码应该改成如下代码（/cpu/arm920t/start.S）。

```
#ifdef CONFIG_S3C2440_NAND_BOOT     @START
@ reset NAND
    mov r1, #NAND_CTL_BASE
    ldr     r2, =0xf830                 @ initial value
    str     r2, [r1, #oNFCONF]
    ldr     r2, [r1, #oNFCONF]
    bic    r2, r2, #0x800                   @ enable chip
    str     r2, [r1, #oNFCONF]
    mov r2, #0xff               @ RESET command
    strb r2, [r1, #oNFCMD]
    mov r3, #0                              @ wait
    nand1:
    add    r3, r3, #0x1
    cmp r3, #0xa
    blt     nand1
nand2:
    ldr     r2, [r1, #oNFSTAT]          @ wait ready
    tst      r2, #0x1
    beq    nand2
    ldr     r2, [r1, #oNFCONF]
    orr    r2, r2, #0x800                   @ disable chip
    str     r2, [r1, #oNFCONF]
    @ get read to call C functions (for nand_read())
    ldr     sp, DW_STACK_START            @ setup stack pointer
    mov fp, #0                              @ no previous frame, so fp=0
@ copy U-Boot to RAM
    ldr     r0, =TEXT_BASE
    mov         r1, #0x0
    mov r2, #0x20000
    bl       nand_read_ll
    tst      r0, #0x0
    beq    ok_nand_read
bad_nand_read:
    loop2:      b       loop2               @ infinite loop
ok_nand_read:
@ verify
    mov r0, #0
    ldr     r1, =TEXT_BASE
    mov r2, #0x400          @ 4 bytes * 1024 = 4K-bytes
go_next:
```

```
        ldr     r3, [r0], #4
        ldr     r4, [r1], #4
        teq     r3, r4
        bne     notmatch
        subs r2, r2, #4
        beq     stack_setup
        bne     go_next
    notmatch:
    loop3:      b       loop3           @ infinite loop
    #endif @ CONFIG_S3C2440_NAND_BOOT   @END
```

在“ _start_armboot: .word start_armboot ”后加入：

```
    .align  2
    DW_STACK_START:  .word  STACK_BASE+STACK_SIZE-4
```

5. 修改内存配置

将 board/GT2440/lowlevel_init.S 文件修改如下：

```
    #define BWSCON          0x48000000
    #define PLD_BASE        0x2C000000
    #define SDRAM_REG       0x2C000106
    /* BWSCON */
    #define DW8                 (0x0)
    #define DW16                (0x1)
    #define DW32                (0x2)
    #define WAIT                (0x1<<2)
    #define UBLB                (0x1<<3)
    /* BANKSIZE */
    #define BURST_EN            (0x1<<7)
    #define B1_BWSCON           (DW16 + WAIT)
    #define B2_BWSCON           (DW32)
    #define B3_BWSCON           (DW32)
    #define B4_BWSCON           (DW16 + WAIT + UBLB)
    #define B5_BWSCON           (DW8 + UBLB)
    #define B6_BWSCON           (DW32)
    #define B7_BWSCON           (DW32)
    /* BANK0CON */
    #define B0_Tacs             0x0 /*   0clk */
    #define B0_Tcos             0x1 /*   1clk */
    #define B0_Tacc             0x7 /*   14clk */
    #define B0_Tcoh             0x0 /*   0clk */
    #define B0_Tah              0x0     /*   0clk */
    #define B0_Tacp             0x0         /* page mode is not used */
    #define B0_PMC              0x0 /* page mode disabled */
    /* BANK1CON */
    #define B1_Tacs             0x0 /*   0clk */
```

```
#define B1_Tcos                 0x1 /*    1clk */
#define B1_Tacc                 0x7 /*    14clk */
#define B1_Tcoh                 0x0 /*    0clk */
#define B1_Tah                  0x0 /*    0clk */
#define B1_Tacp                 0x0         /* page mode is not used */
#define B1_PMC                    0x0 /* page mode disabled */
/* REFRESH parameter */
#define REFEN                     0x1 /* Refresh enable */
#define TREFMD                    0x0 /* CBR(CAS before RAS)/Auto refresh */
#define Trp                 0x0 /* 2clk */
#define Trc                 0x3 /* 7clk */
#define Tchr                    0x2 /* 3clk */
#define REFCNT              1113 /*period=14.6us,HCLK=60Mhz, (2048+1-14.6*60)*/
    .word ((B6_MT<<15)+(B6_Trcd<<2)+(B6_SCAN))
    .word ((B7_MT<<15)+(B7_Trcd<<2)+(B7_SCAN))
    .word ((REFEN<<23)+(TREFMD<<22)+(Trp<<20)+(Trc<<18)+(Tchr<<16)+REFCNT)
    .word 0x32
    .word 0x30
    .word 0x30
```

6. 加入 **Nand Flash** 读操作代码

创建 board/GT2440/nand_read.c 文件并加入 Nand Flash 读函数。

```
#include <config.h>
#define __REGb(x) (*(volatile unsigned char *)(x))
#define __REGi(x) (*(volatile unsigned int *)(x))
#define NF_BASE    0x4e000000
#define NFCONF    __REGi(NF_BASE + 0x0)
#define NFCMD    __REGb(NF_BASE + 0x4)
#define NFADDR    __REGb(NF_BASE + 0x8)
#define NFDATA    __REGb(NF_BASE + 0xc)
#define NFSTAT    __REGb(NF_BASE + 0x10)
#define BUSY 1
inline void wait_idle(void)
{
    Int i;
    while(!(NFSTAT & BUSY))
    {
        for (i = 0; i < 10; i++);
    }
}

/* low level nand read function */
int nand_read_ll(unsigned char *buf, unsigned long start_addr, int size)
{
    int i, j;
```

```
    if ((start_addr & NAND_BLOCK_MASK) || (size & NAND_BLOCK_MASK))
    {
        return -1; /* invalid alignment */
    }
    /* chip Enable */
    NFCONF &=~0x800;
    for (i = 0; i < 10; i++);
    for (i = start_addr; i < (start_addr + size);)
    {
        /* READ0 */
        NFCMD = 0;
        /* Write Address */
        NFADDR = i & 0xff;
        NFADDR = (i >> 9) & 0xff;
        NFADDR = (i >> 17) & 0xff;
        NFADDR = (i >> 25) & 0xff;
        wait_idle();
        for (j = 0; j < NAND_SECTOR_SIZE; j++, i++)
        {
            *buf = (NFDATA & 0xff);
             buf++;
        }
    }
    /* chip Disable */
    NFCONF |= 0x800; /* chip disable */
    return 0;
}
```

然后修改 board/GT2440/makefile 文件，以增加 nand_read()函数。

```
OBJS := GT2440.o   flash.o    nand_read.o
```

7. 对 Nand Flash 进行初始化

在 board/GT2440/GT2440.c 文件中加入 Nand Flash 的初始化函数。

```
#if (CONFIG_COMMANDS & CFG_CMD_NAND)
typedef enum
{
    NFCE_LOW,
    NFCE_HIGH
} NFCE_STATE;
static inline void NF_Conf(u16 conf)
{
    S3C2440_NAND * const nand = S3C2440_GetBase_NAND();
    nand->NFCONF = conf;
}
static inline void NF_Cmd(u8 cmd)
```

```
{
    S3C2440_NAND * const nand = S3C2440_GetBase_NAND();
    nand->NFCMD = cmd;
}
static inline void NF_CmdW(u8 cmd)
{
    NF_Cmd(cmd);
    udelay(1);
}
static inline void NF_Addr(u8 addr)
{
    S3C2440_NAND * const nand = S3C2440_GetBase_NAND();
    nand->NFADDR = addr;
}
static inline void NF_SetCE(NFCE_STATE s)
{
    S3C2440_NAND * const nand = S3C2440_GetBase_NAND();
    switch (s)
    {
        case NFCE_LOW:
            nand->NFCONF &= ~(1<<11);
            break;
        case NFCE_HIGH:
            nand->NFCONF |= (1<<11);
            break;
    }
}
static inline void NF_WaitRB(void)
{
    S3C2440_NAND * const nand = S3C2440_GetBase_NAND();
    while (!(nand->NFSTAT & (1<<0)));
}
static inline void NF_Write(u8 data)
{
    S3C2440_NAND * const nand = S3C2440_GetBase_NAND();
    nand->NFDATA = data;
}
static inline u8 NF_Read(void)
{
    S3C2440_NAND * const nand = S3C2440_GetBase_NAND();
    return(nand->NFDATA);
}
static inline void NF_Init_ECC(void)
{
    S3C2440_NAND * const nand = S3C2440_GetBase_NAND();
    nand->NFCONF |= (1<<12);
```

```
}
static inline u32 NF_Read_ECC(void)
{
        S3C2440_NAND * const nand = S3C2440_GetBase_NAND();
        return(nand->NFECC);
}
#endif
/*
* NAND flash initialization.
*/
#if (CONFIG_COMMANDS & CFG_CMD_NAND)
extern ulong nand_probe(ulong physadr);
static inline void NF_Reset(void)
{
        int i;
        NF_SetCE(NFCE_LOW);
        NF_Cmd(0xFF); /* reset command */
        for (i = 0; i < 10; i++); /* tWB = 100ns. */
        NF_WaitRB(); /* wait 200～500us; */
        NF_SetCE(NFCE_HIGH);
}
static inline void NF_Init(void)
{
        #define TACLS 0
        #define TWRPH0 4
        #define TWRPH1 2
        NF_Conf((1<<15)|(0<<14)|(0<<13)|(1<<12)
                        |(1<<11)|(TACLS<<8)|(TWRPH0<<4)|(TWRPH1<<0));
        /* 1 1 1 1, 1 xxx, r xxx, r xxx */
        /* En 512B 4step ECCR nFCE=H tACLS tWRPH0 tWRPH1 */
        NF_Reset();
}
void nand_init(void)
{
        S3C2440_NAND * const nand = S3C2440_GetBase_NAND();
        NF_Init();
        #ifdef DEBUG
              printf("NAND flash probing at 0x%.8lX\n", (ulong)nand);
        #endif
        printf ("%4lu MB\n", nand_probe((ulong)nand) >> 20);
}
#endif
```

8. 修改 **I/O** 引脚配置

在 board/GT2440/GT2440.c 中修改 GPIO 的配置。

```
/* set up the I/O ports */
gpio->GPACON = 0x007FFFFF;
gpio->GPBCON = 0x002AAAAA;
gpio->GPBUP = 0x000002BF;
gpio->GPCCON = 0xAAAAAAAA;
gpio->GPCUP = 0x0000FFFF;
gpio->GPDCON = 0xAAAAAAAA;
gpio->GPDUP = 0x0000FFFF;
gpio->GPECON = 0xAAAAAAAA;
gpio->GPEUP = 0x000037F7;
gpio->GPFCON = 0x00000000;
gpio->GPFUP = 0x00000000;
gpio->GPGCON = 0xFFEAFF5A;
gpio->GPGUP = 0x0000F0DC;
gpio->GPHCON = 0x0018AAAA;
gpio->GPHDAT = 0x000001FF;
gpio->GPHUP = 0x00000656
```

9. 在头文件中加入 **Nand Flash** 设备

在 include/linux/mtd/nand_ids.h 文件中加入 Nand Flash 设备及对应的宏定义。

```
static struct nand_flash_dev nand_flash_ids[] =
{
        ......
       {"Samsung KM29N16000",NAND MFR SAMSUNG, 0x64, 21, 1, 2, 0x1000, 0},
       {"Samsung K9F1208U0M",    NAND_MFR_SAMSUNG, 0x76, 26, 0, 3, 0x4000, 0},
       {"Samsung unknown 4Mb", NAND_MFR_SAMSUNG, 0x6b, 22, 0, 2, 0x2000, 0},
        ......
        {NULL,}
};
```

10. 设置 **Nand Flash** 环境

对 common/env_nand.c 文件进行相关的 Nand Flash 设置，将代码修改如下。

```
int nand_legacy_rw (struct nand_chip* nand, int cmd,
            size_t start, size_t len,
            size_t * retlen, u_char * buf);
extern struct nand_chip nand_dev_desc[CFG_MAX_NAND_DEVICE];
extern int nand_legacy_erase(struct nand_chip *nand,
                                        size_t ofs, size_t len, int clean);
/* info for NAND chips, defined in drivers/nand/nand.c */
extern nand_info_t nand_info[CFG_MAX_NAND_DEVICE];
......
#else /* ! CFG_ENV_OFFSET_REDUND */
int saveenv(void)
{
      ulong total;
      int ret = 0;
```

```
        puts ("Erasing Nand...");
        if (nand_legacy_erase(nand_dev_desc + 0,
                        CFG_ENV_OFFSET, CFG_ENV_SIZE, 0))
        {
                return 1;
        }
        puts ("Writing to Nand... ");
        total = CFG_ENV_SIZE;
        ret = nand_legacy_rw(nand_dev_desc + 0, 0x00 | 0x02, CFG_ENV_OFFSET,
                        CFG_ENV_SIZE, &total, (u_char*)env_ptr);
        if (ret || total != CFG_ENV_SIZE)
        {
                return 1;
        }
        puts ("done\n");
        return ret;
        ......
#else /* ! CFG_ENV_OFFSET_REDUND */
void env_relocate_spec (void)
{
#if !defined(ENV_IS_EMBEDDED)
        ulong total;
        int ret;
        total = CFG_ENV_SIZE;
        ret = nand_legacy_rw(nand_dev_desc + 0, 0x01 | 0x02, CFG_ENV_OFFSET,
                        CFG_ENV_SIZE, &total, (u_char*)env_ptr);
        ......
```

4.3.4 刻录 U-Boot

在 U-Boot 修改完成之后，可以使用 JTAG 软件 SJF2440 来将 U-Boot 刻录进嵌入式系统中。SJF2440 是由三星公司提供的用来刻录嵌入式系统的 Flash 的工具程序，其可以通过并口连接一个 JTAG 板，用来刻录 K9F1208、AMD29LV800BB 等型号的 Flash。

注意：因为 SJF2440 是一种十分慢速的刻录工具，并且没有校验功能，它一般适用于刻录比较小的 Bootloader。

此外也可以使用 4.1 节中介绍的 J-link 来对 U-Boot 进行刻录，其详细操作步骤说明如下。

（1）安装好驱动之后会在桌面上出现如图 4.15 所示的 J-Link ARM 启动图标（2 个）。

图 4.15　J-Link ARM 启动图标

（2）连接好硬件系统后启动 J-Link ARM，可以看到如图 4.16 所示的界面。

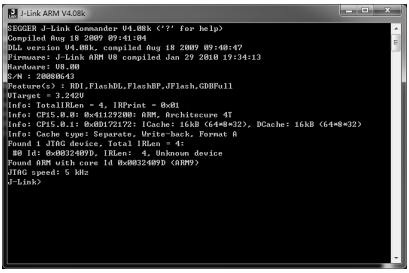


图 4.16　J-Link ARM 的运行界面

（3）按照如图 4.17 所示对 J-Link 进行相应的配置。

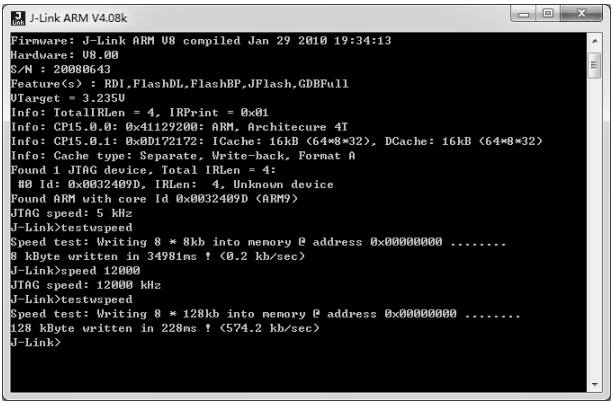


图 4.17　对 J-Link 进行配置

（4）双击打开 J-Flash ARM，在如图 4.18 所示的 File 菜单中选择 New Project 子菜单。

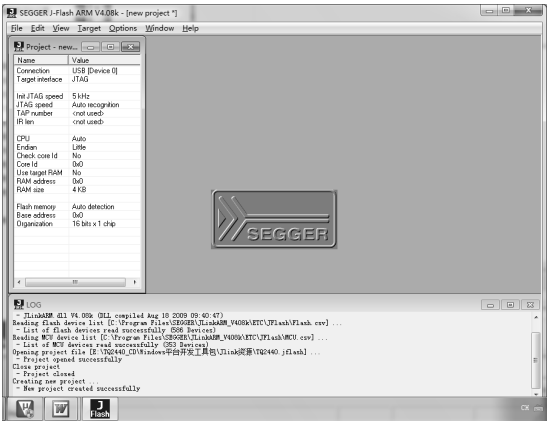


图 4.18　新建刻录项目

（5）然后在 Option 菜单中选择 Project settings 子菜单对项目进行设置，如图 4.19 所示。

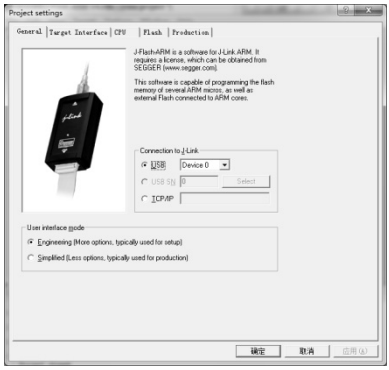


图 4.19　对项目进行设置

（6）对目标系统接口（Target Interface）、处理器（CPU）和 Flash 进行设置，如图 4.20～图 4.22 所示。

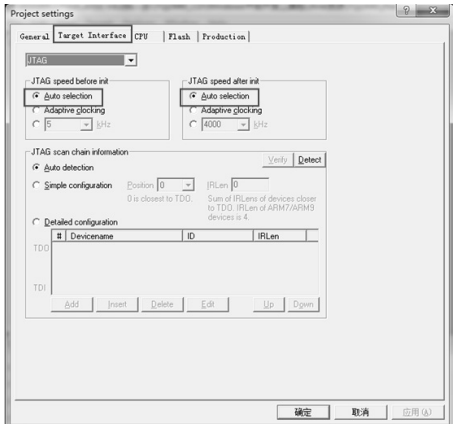


图 4.20　设置目标系统接口

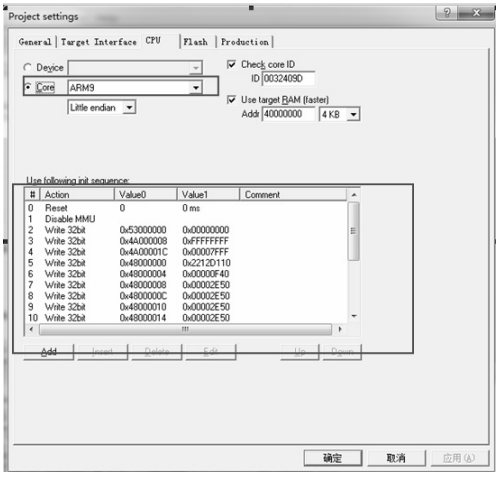


图 4.21　设置处理器

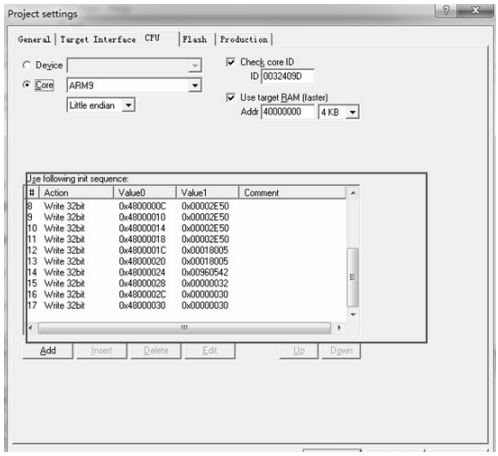


图 4.22　设置 Flash

注意：Flash 的具体型号应该根据用户嵌入式系统的实际情况进行选择。

（7）在 File 菜单中选择 open program 并选择对应的 U-Boot 文件，如图 4.23 所示，然后单击 OK 按钮。

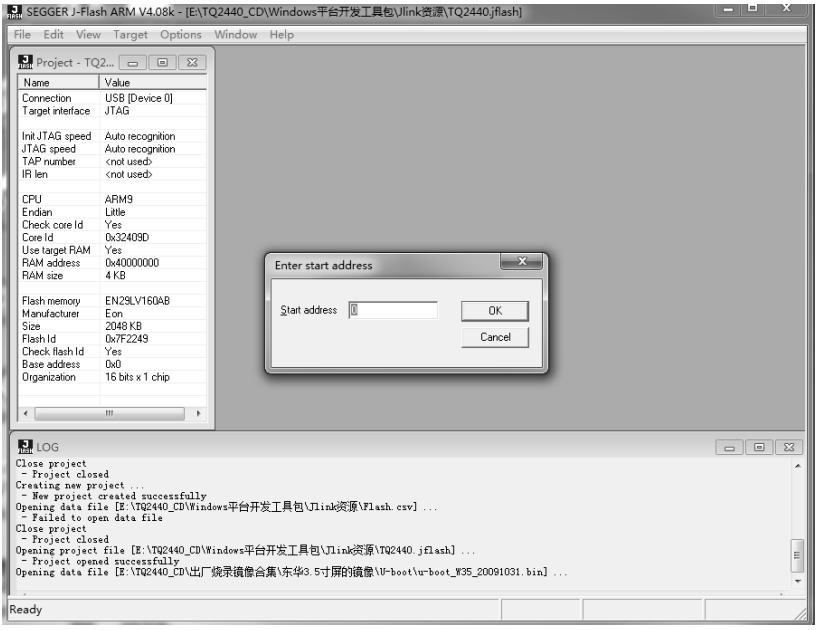


图 4.23　选择对应的 U-Boot 文件

（8）此时可以看到如图 4.24 所示的文件编码，选择 Target 菜单中的 Program 或使用 F5 快捷键进行刻录。

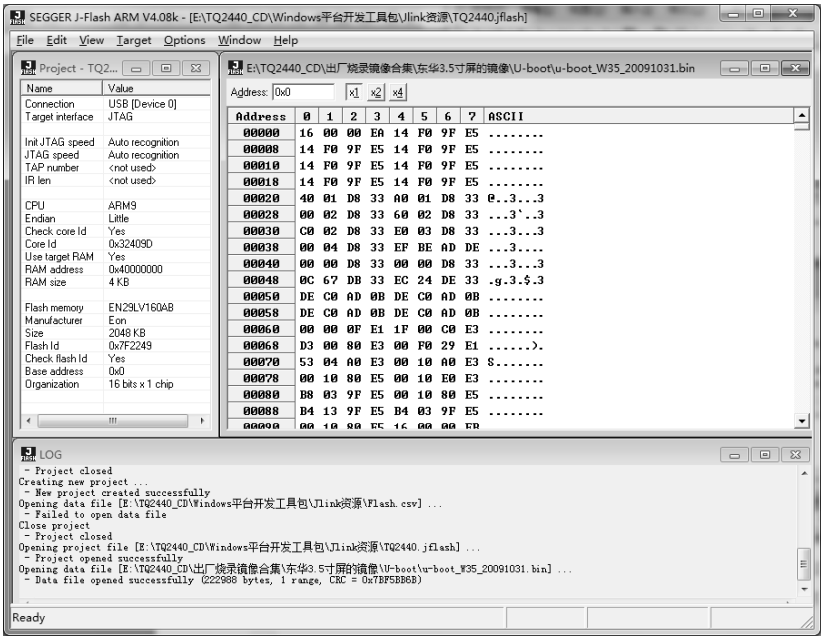


图 4.24　刻录 U-Boot

4.4　【应用实例】——使用 U-Boot

U-Boot 被刻录进嵌入式系统之后即可以使用一些辅助工具进行相应的调试观察、刻录裸机代码及操作系统、文件系统，这些辅助工具包括超级终端和 DNW。

4.4.1　使用超级终端和嵌入式系统进行通信

超级终端是一个在 Windows（在 Linux 下则使用 Minicom）下 PC 和与嵌入式系统通过串口进行数据交互的工具，这里以 Windows 7 为例着重介绍超级终端的设置。

（1）由于在 Windows 7 的附件中已经不自带超级终端了，所以用户需要从网络上自行下载超级终端程序，这是一个绿色程序，不需要安装，直接运行即可。此时，会出现“默认 Telnet 程序”界面，询问是否将它设为默认的 telnet 程序，这里可以单击“否（N）”按钮，如图 4.25 所示。

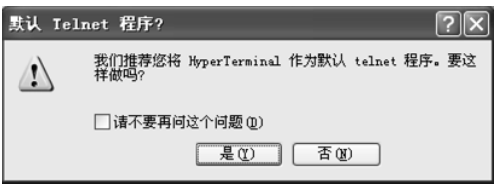


图 4.25　设置默认的 Telnet 程序

（2）接着会弹出如图 4.26 所示的“位置信息”对话框，在“您的区号（或城市号）是什么？”这个地方可以直接跳过或随意填写，然后单击“确定”按钮。

图 4.26　设置“位置信息”界面

（3）接下会弹出“电话和调制解调器选项”对话框，如图 4.27 所示，此时单击“确定”按钮即可。

图 4.27　“电话和调制解调器选项”对话框设置

（4）在“连接描述”对话框中输入用户设置的超级终端的名称，可以根据用户自己的选择输入一个字符串，然后单击“确定”按钮。

图 4.28　“连接描述”对话框

（5）在“连接到”对话框中选择对应的串口编号，可以根据用户的实际情况选择。

图 4.29 “连接到”对话框

（6）在如图 4.30 所示的“COM 属性”对话框中设置串口信息，“每秒位数”选择“115200”，“数据位”选择“8”，“奇偶校验”选择“无”，“停止位”选择“1”，“数据流控制”选择“无”，然后单击“确定”按钮。

图 4.30 COM 属性设置

需要注意的是，在这一步中数据流控制必须选择“无”，也就是无数据流控制，否则只能看到输出而看不到输入，另外波特率的值也必须是 115200bps。

（7）在如图 4.31 所示的超级终端窗口中选择“文件”菜单中的“保存”子菜单，用于保存超级终端设置，此后就可以直接使用该快捷方式了。

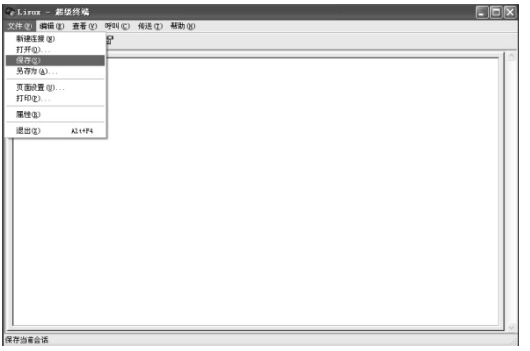


图 4.31 保存超级终端的快捷方式

当完成超级终端设置之后，就可以使用串口和嵌入式系统进行数据交互，图 4.32 是交互界面。

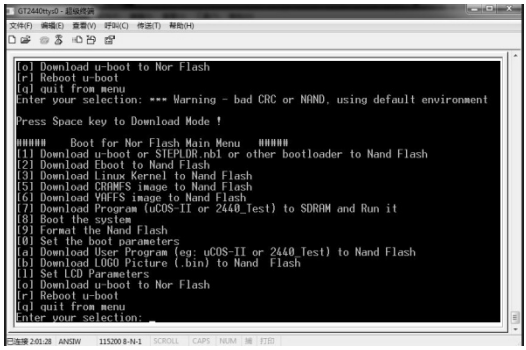


图 4.32　使用超级终端和嵌入式系统进行数据交互

4.4.2　使用 DNW 下载工具和嵌入式系统进行通信

在嵌入式系统上已经移植完成 U-Boot 之后，在超级终端上可以看到如图 4.32 所示的交互界面，此时将嵌入式系统 GT2440 的 USB 端口（主 USB）通过 USB 线连接到 PC 的 Windows 系统，将会自动识别出新设备，然后弹出一个“找到新的硬件向导”的对话框，此时可以安装对应的 DNW 工具驱动。

安装完毕 USB 下载驱动后，打开 DNW 软件，就可以在 DNW 软件的顶上看到 USB 连接 OK 的字样“[USB:OK]”，如果驱动未安装成功则会显示“USB:x”，如图 4.33 所示。同时可以在“设备管理器”中看到刚刚安装的 USB 驱动，如图 4.34 所示。

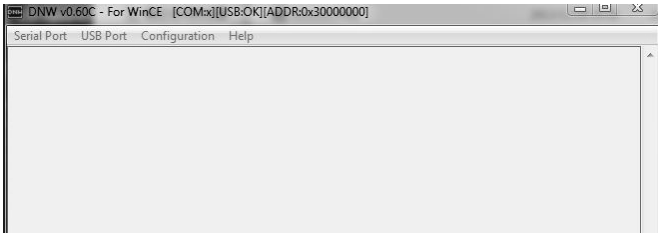


图 4.33　DNW 中 USB 安装成功提示

图 4.34 “设备管理器”中 USB 下载驱动安装成功

此时就可以使用 USB 下载 U-Boot、操作系统和文件系统了。DNW 具体设置步骤如下。

（1）打开 DNW 软件，选择 Configuration→Options 命令，弹出 UART/USB Options 对话框，如图 4.35 所示。

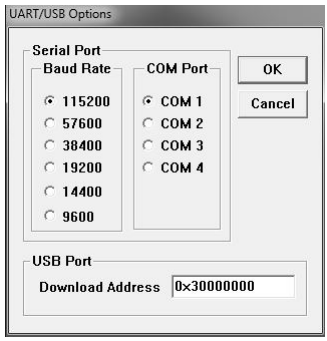


图 4.35　UART/USB Options 对话框

（2）按照如图 4.35 的参数设置：Baud Rate 选择“115200”，COM Port 选择 COM1（根据 PC 的实际情况选择此选项），USB Port 的 Download Address 填入“0x30000000”，然后单击 OK 按钮。

此时即可使用 DNW 和超级终端和嵌入式系统进行数据交互和下载。

第5章

建立和使用嵌入式系统的交叉编译环境

在第 4 章的 4.1.2 节中已经介绍过嵌入式系统的软件开发需要在 PC 上进行，其使用运行在 PC 上的编辑和编译环境，即交叉编译环境，本章将介绍交叉编译环境的建立和使用方法，涉及的知识点说明如下：

- 在 Linux 环境下建立交叉编译环境的方法；
- VIM 编辑器的使用方法；
- GCC 的使用方法；
- GDB 的使用方法。

5.1 建立交叉编译环境

嵌入式系统中的编译过程和普通开发平台的编译过程完全不同：后者有时称为本地编译，即在当前平台编译，编译得到的程序也在本地执行；而前者称为交叉编译，即在一种平台上编译，并能够运行在另一种体系结构完全不同的平台上。通常来说，用户需要在 X86 系列的处理器平台上编译出能运行在 ARM 架构的处理器平台上的程序，这种可以跨平台的编译工具一般被称为交叉编译工具，由于它是由多个程序连接构成的，所以又称为交叉编译工具链。它在不同平台的移植和嵌入式开发时非常有用。如果要得到在目标机上运行的程序，就必须使用交叉编译工具来完成。

5.1.1 交叉编译环境的工具链

交叉开发工具链就是编译、链接、处理和调试跨平台体系结构的程序代码。每次执行工具链软件时，通过带有不同的参数，可以实现编译、链接、处理或调试等不同的功能。从工具链的组成上来说，它一般由多个程序构成，分别对应着各个功能。

工具链一般由编译器、连接器、解释器和调试器组成。在 X86 的 Linux 主机上，交叉开发工具链除了能够编译生成在 ARM、MIPS、PowerPC 等硬件架构上运行的程序，还可以为 X86 平台上不同版本的 Linux 提供编译开发的程序功能。所以可以通过在同一台 Linux 主

机上使用交叉编译工具的方式来维护不同版本的 X86 目标机。

Linux 经常使用的工具链软件包括：Binutils、GCC、Glibc 和 Gdb。

- Binutils 是二进制程序处理工具，包括链接器、汇编器等目标程序处理的工具。
- GCC（GNU Compiler Collection）是编译器，不但支持 C/C++语言的编译，而且支持 FORTRAN JAVA ADA 等编程语言。不过，一般不需要配置其他语言的选项，也可以避免编译其他语言功能而导致的错误。对于 C/C++语言的完整支持，需要支持 Glibc 库。
- Glibc 是应用程序编程的函数库软件包，可以编译生成静态库和共享库，完整的 GCC 需要支持 Glibc。
- Gdb 是调试工具，可以读取可执行程序中的符号表，对程序进行源码调试。

通过这些软件包，可以生成 gcc、g++、ar、as、ld 等编译链接工具，还可以生成 glibc 库和 gdb 调试器。在生成交叉开发的工具链时，可以在文件名字上加一个前缀，用来区别本地的工具链，如 arm-linux-gcc，表示这个编译器用于编译在 Linux 系统下 ARM 目标平台上运行的程序。

在裁剪用于嵌入式系统的 Linux 内核时，由于嵌入式系统的存储大小有限，所以需要的链接工具也可根据嵌入式系统的特性进行制作，建立自己的交叉编译工具链。例如，有时为了减小 Glibc 库的大小，可以考虑用 uclibc、dietlibc 或 newlib 库来代替 Glibc 库，这时就需要自己动手进行交叉编译工具链的构建。由于 Linux 交叉编译工具链使用和 GNU 一样的工具链，而 GNU 的工具和软件都是开放源代码的，所以读者只需要从 GNU 网站 http://www .gnu.org 或镜像网站下载源代码后，根据需要进行裁剪，然后编译即可。当然，构建交叉编译工具链是一个相当复杂的过程，如果读者不想经历这一过程，可以在网上下载一些编译好的工具链。不过，为了学习，还是建议读者自己动手制作一个交叉编译工具链。

构建交叉编译器的第一个步骤就是确定目标平台。在 GNU 系统中，每个目标平台都有一个明确的格式，这些信息用于在构建过程中识别要使用的不同工具的正确版本。因此，当在一个特定目标机下运行 GCC 时，GCC 便在目录路径中查找包含该目标规范的应用程序路径。GNU 的目标规范格式为 CPU-PLATFORM-OS。例如，x86/i386 目标机名为 i686-pc-linux。

基于 ARM 平台的交叉工具链目标平台的名称为 arm-linux，通常可以采用如下三种方法来获得或构建交叉编译链。

- 分步编译和安装交叉编译工具链所需要的库和源代码，最终生成交叉编译工具链。该方法相对比较困难，适合想深入学习构建交叉工具链的读者。如果只是想使用交叉工具链，建议使用方法二构建交叉工具链。
- 直接通过网上下载已经制作好的交叉编译工具链，相对来说该方法非常简便且稳定性高，缺点是灵活性较差，且不一定适合用户的目标开发系统。
- 通过 Crosstool 脚本工具来实现一次编译，生成交叉编译工具链，该方法相对于方法一要简单许多，并且出错的机会也非常少，建议在大多数情况下使用该方法构建交叉编译工具链。

5.1.2 【应用实例】——安装交叉编译环境

在 Linux 里可以直接安装并使用已经制作好的交叉编译环境，其详细安装步骤说明如下（本书的操作步骤基于 ubuntu 12.04LTS，但是在其他 Linux 发行版中差别不大）。

（1）从 http://www.kegel.com/crosstool/中下载 arm-linux-gcc-4.3.3.tar.gz 包，这是一个可以对 Linux 内核和应用程序进行编译的工具。

注意：如果要对 U-Boot 进行编译，则需要下载 arm-linux-gcc-3.4.1.tar.gz 包。

（2）对下载包执行 tar xvzf arm-linux-gcc-4.3.3.tar.gz –C 命令，此时 arm-linux-gcc-4.3.3 已经被安装到了/usr/local/arm/4.3.3 目录，此时可以用 ls 命令查看该文件夹，如图 5.1 所示。

```
alloy@ubuntu:/usr/local/arm/4.3.3$ ls
arm-none-linux-gnueabi  bin  lib  libexec  share
alloy@ubuntu:/usr/local/arm/4.3.3$
```

图 5.1　安装好的 arm-linux-gcc 目录

（3）使用 vim 编辑器（具体使用方法参考 5.3.1 节），或者在 Linux 的图形编辑界面下使用 gedit 等工具对/root/.bashrc 文件进行编辑，给系统添加环境变量，在最后一行添加如下的语句，这是为了在任何目录下都可以调用 arm-linux-gcc 命令，如图 5.2 所示。

```
export PATH=/usr/local/arm/4.3.3/bin:$PATH
```

```
# Alias definitions.
# You may want to put all your additions into a separate file like
# ~/.bash_aliases, instead of adding them here directly.
# See /usr/share/doc/bash-doc/examples in the bash-doc package.

if [ -f ~/.bash_aliases ]; then
    . ~/.bash_aliases
fi

# enable programmable completion features (you don't need to enable
# this, if it's already enabled in /etc/bash.bashrc and /etc/profile
# sources /etc/bash.bashrc).
#if [ -f /etc/bash_completion ] && ! shopt -oq posix; then
#    . /etc/bash_completion
#fi
export PATH=/usr/local/arm/4.3.3/bin:$PATH
```

图 5.2　添加环境变量

（4）重新启动或登录 Linux 系统，在任意路径下运行 arm-linux-gcc-4.3.3-v，可以看到对应的版本信息，并且如果用 Tab 按键会自动补齐命令行，如图 5.3 所示。

```
alloy@ubuntu:/$ arm-linux-g
arm-linux-g++        arm-linux-gcc-4.3.3  arm-linux-gdb        arm-linux-gprof
arm-linux-gcc        arm-linux-gcov       arm-linux-gdbtui
alloy@ubuntu:/$ arm-linux-g
arm-linux-g++        arm-linux-gcc-4.3.3  arm-linux-gdb        arm-linux-gprof
arm-linux-gcc        arm-linux-gcov       arm-linux-gdbtui
alloy@ubuntu:/$ arm-linux-gcc-4.3.3 -v
Using built-in specs.
Target: arm-none-linux-gnueabi
Configured with: /scratch/mitchell/builds/4.3-arm-none-linux-gnueabi-respin/src/gcc-4.3/configure --build=i6
86-pc-linux-gnu --host=i686-pc-linux-gnu --target=arm-none-linux-gnueabi --enable-threads --disable-libmudfl
ap --disable-libssp --disable-libstdcxx-pch --with-gnu-as --with-gnu-ld --with-specs='%{funwind-tables|fno-u
nwind-tables|mabi=*|ffreestanding|nostdlib:;:-funwind-tables}' --enable-languages=c,c++ --enable-shared --en
able-symvers=gnu --enable-__cxa_atexit --with-pkgversion='Sourcery G++ Lite 2009q1-203' --with-bugurl=https:
//support.codesourcery.com/GNUToolchain/ --disable-nls --prefix=/opt/codesourcery --with-sysroot=/opt/codeso
urcery/arm-none-linux-gnueabi/libc --with-build-sysroot=/scratch/mitchell/builds/4.3-arm-none-linux-gnueabi-
respin/lite/install/arm-none-linux-gnueabi/libc --with-gmp=/scratch/mitchell/builds/4.3-arm-none-linux-gnuea
bi-respin/lite/obj/host-libs-2009q1-203-arm-none-linux-gnueabi-i686-pc-linux-gnu/usr --with-mpfr=/scratch/mi
tchell/builds/4.3-arm-none-linux-gnueabi-respin/lite/obj/host-libs-2009q1-203-arm-none-linux-gnueabi-i686-pc
-linux-gnu/usr --disable-libgomp --enable-poison-system-directories --with-build-time-tools=/scratch/mitchel
l/builds/4.3-arm-none-linux-gnueabi-respin/lite/install/arm-none-linux-gnueabi/bin --with-build-time-tools=/
scratch/mitchell/builds/4.3-arm-none-linux-gnueabi-respin/lite/install/arm-none-linux-gnueabi/bin
Thread model: posix
gcc version 4.3.3 (Sourcery G++ Lite 2009q1-203)
```

图 5.3　查看编译器信息

5.2 使用交叉编译环境

Linux 环境下使用交叉编译环境进行开发，需要使用一系列工具，包括编辑工具、编译工具、调试工具和维护工具，本节将介绍它们的使用方法。

5.2.1 使用编辑器 vim

在 Linux 中开发 C 语言应用代码，首先需要进行源代码的编写，此时需要一个代码编辑器，在 Linux 中最常见的代码编辑器包括 vim、emacs、gedit 等。

注意：代码编辑器的实质就是一个文本编辑器，只不过增加了一些代码编辑的辅助功能，如关键字高亮，补齐等。

vim 是“Vimsual Interface”的简称，是 vi 的功能加强升级版，其是 UNIX/Linux 下最基本的文本编辑器，工作在字符模式下，由于不需要图形界面，使它成为效率很高的文本编辑器。它在 Linux 上的地位就像 Edit 程序在 DOS 上一样。它可以执行输入、输出、删除、查找、替换、块操作等众多文本操作，而且用户可以根据自己的需要对其进行定制，这是其他编辑程序所没有的。尽管在 Linux 上也有很多图形界面的编辑器可用，但 vim 在系统和服务器管理应用中的功能是那些图形编辑器所无法比拟的。

1．vim 的启动和退出

在 Linux 终端命令提示符下输入 vim（或 vim 文件名），即可启动 vim 编辑器。如：

```
vim filename
```

或者

```
vim
```

按下“Enter”键执行该命令，系统便会自动打开文件名为“filename”的文件的 vim 编辑界面，其初始界面如图 5.4 所示，也可以通过在 X Windows 下的相应操作来打开一个图形化的操作界面。

图 5.4 vim 的操作界面

当使用“vim + 文件名”的命令时，若进行编辑的是当前工作目录下已存在的文件，启动 vim 后便可看到该文件中的内容；若是当前目录下不存在的文件，则系统首先创建该文件，再使用 vim 进行编辑。

要退出 vim，必须先按下“Esc”键回到命令行模式，然后输入“：”，此时光标会停留在最下面一行（底行模式），再输入“q”，最后按下 Enter 键即可退出。

2．vim 的工作模式

vim 拥有三种工作模式：命令行模式（command mode）、插入模式（input mode）与底行模式（last line mode）。三种模式下的功能可描述如下。

- 命令行模式：也叫“普通模式”，它是启动 vim 编辑器后的初始模式。在该模式下，主要使用隐式命令（命令不显示）来实现光标的移动、复制、粘贴、删除等操作。但是在该模式下，编辑器并不接受用户从键盘输入的任何字符来作为文档的编辑内容。
- 插入模式：在该模式下，用户输入的任何字符都被认为是编辑到某一个文件的内容，并直接显示在 vim 的文本编辑区。
- 底行模式：在该模式下，用户输入的任何字符都会在 vim 的最下面一行显示，按下 Enter 键后便会执行该命令（当然前提是这是一个正确的命令）。

使用 vim 编辑器，首先必须能够熟练掌握各种工作模式下的功能，以及各种工作模式间的切换，图 5.5 所示为 vim 三种工作模式间的切换方法。

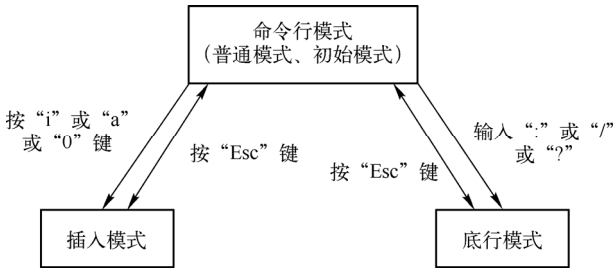


图 5.5　vim 三种工作模式间的切换

从图 5.5 中可以看到，命令行模式是 vim 编辑器的初始模式，在该模式下可以实现到任何模式的切换。而插入模式和底行模式之间不能相互切换，因为在插入模式下，任何输入的字符都被认为是编辑到某一个文件的内容，而不是命令；而在底行模式下，任何输入的字符都被看作底行命令（尽管可能是不合法的），二者都必须先通过命令行模式才能进入对方，即需要先按下 Esc 键回到初始模式。

命令行模式是进入 vim 后的初始模式，在该模式下主要使用方向键来移动光标的位置，并通过相应的命令来进行文字的编辑。在插入模式下按“Esc”键，或在底行模式下按“Esc”键，或在底行模式下执行了错误的命令，vim 都会自动回到命令行模式。本节介绍命令行模式中常用的操作命令，由于这些命令比较多，在此仅作简单介绍，用户在使用时也可以查阅帮助文档。

在命令行模式下，一般通过使用上、下、左、右 4 个方向键来移动光标的位置。但是在有些情况下，如使用 telnet 远程登录时，方向键就不能使用，必须用命令行模式下的光标

移动命令。这些命令及作用如表 5.1 所示。

表 5.1　移动光标的常用命令

命　令	操 作 说 明
h	向左移动光标
l	向右移动光标
j	向下移动光标
k	向上移动光标
^	将光标移动到该行的开头（指第一个非空字符上）
$	将光标移动到该行行尾，同键盘上的 End 键
0	将光标移动到该行行首，同键盘上的 Home 键
G	将光标移动到文档最后一行的开头（第一个非空字符）
nG	将光标移动到文档的第 n 行的开头（第一个非空字符），n 为正整数
w	光标向后移动一个字（单词）
nw	光标向后移动 n 个字（单词），n 为正整数
b	光标向前移动一个字（单词）
nb	光标向前移动 n 个字（单词），n 为正整数
e	将光标移动到本单词的最后一个字符。如果光标所在的位置为本单词的最后一个字符，则跳动到下一个单词的最后一个字符。“.”、“,”、“#”、“/”等特殊字符都会被当成一个字
{	光标移动到前面的“{”处。这在使用 vim 进行 C 语言编程时很适用
}	同“{”的使用，将光标移动到后面的“}” 处
Ctrl+b	向上翻一页，相当于 Page Up
Ctrl+f	向下翻一页，相当于 Page Down
Ctrl+u	向上移动半页
Ctrl+d	向下移动半页
Ctrl+e	向下翻一行
Ctrl+y	向上翻一行

复制、粘贴是在编辑文档时最常用的操作之一，可以大大节约用户重复输入的时间。vim 的命令行模式下常用的复制、粘贴命令如表 5.2 所示。

表 5.2　复制粘贴的常用命令

命　令	操 作 说 明
yy	复制光标所在行的整行内容
yw	复制光标所在的单词的内容
nyy	复制从光标所在行开始向下的 n 行内容，n 为正整数，表示复制的行数
nyw	复制从光标所在字开始向后的 n 个字，n 为正整数，表示复制的字数
p	粘贴，将复制的内容粘贴到光标所在的位置

vim 编辑器中的删除操作可以一次删除一个字符，也可以一次删除多个字符，或者整行字符，vim 命令行模式常用的删除命令如表 5.3 所示。

表 5.3　删除文本的常用命令

命　令	操 作 说 明
x	删除光标所在位置的字符，同键盘上的 Delete 键
X	删除光标所在位置的前一个字符
nx	删除光标所在位置及其后的 n−1 个字符，n 为正整数
nX	删除光标所在位置及其前的 n−1 个字符，n 为正整数
dw	删除光标所在位置的单词
ndw	删除光标所在位置及其后的 n−1 个单词，n 为正整数
d0	删除当前行光标所在位置前的所有字符
d$	删除当前行光标所在位置后的所有字符
dd	删除光标所在行
ndd	删除光标所在行及其向下的 n−1 行，n 为正整数
nd+上方向键	删除光标所在行及其向上的 n 行，n 为正整数
nd+下方向键	删除光标所在行及其向下的 n 行，n 为正整数

命令行模式下其他常用的命令包括字符替换、撤销操作、符号匹配等，这些在使用 vim 时也是经常遇到的命令，其操作说明如表 5.4 所示。

表 5.4　其他常用命令

命　令	操 作 说 明
r	替换光标所在位置的字符，如 rx 指将光标所在位置的字符替换为 x
R	替换光标所到之处的字符，直到按下“Esc”键为止
u	表示复原功能，即撤销上一次操作
U	取消对当前行所做的所有改变
.	重复执行上一次的命令
ZZ	保存文档后退出 vim 编辑器
%	符号匹配功能，在编辑时若输入“%(”，系统会自动匹配相应的“)”

3．vim 的插入模式

插入模式是 vim 编辑器最简单的模式了，因为在此模式下没有那些烦琐的命令，用户从键盘输入的任何有效字符都被看作写进当前正在编辑的文件中的内容，并显示在 vim 的文本编辑区。

也就是说，只有在插入模式下才可以进行文字的输入操作。表 5.5 所示为从命令行模式切换至插入模式的几个常用命令。当在插入模式下时，可以按 Esc 键回到命令行模式。

表 5.5　命令行模式切换至插入模式的命令

命　令	操 作 说 明
i	从光标所在的位置开始插入新的字符
I	从光标所在行的行首开始插入新的字符
a	从光标所在位置的下一个字符开始插入新的输入字符
A	从光标所在行的行尾开始插入新的字符
o	新增加一行，并将光标移动到下一行的开头开始插入字符
O	在当前行的上面新增加一行，并将光标移动到上一行的开头开始插入字符

4．vim 的底行模式

vim 的底行模式也叫“最后行模式”，是指可以在界面底部的一行输入控制操作命令，主要用来进行一些文字编辑的辅助功能，如字串搜寻、替代、保存文件，以及退出 vim 等。不同于命令行模式，底行模式下输入的命令都会在底部的一行中显示，按下 Enter 键 vim 便会执行底行的命令了。

在命令行模式下输入冒号“：”，或者使用“？”和“/”键，就可以进入底行模式了。比起命令行模式的诸多操作命令，最后行模式的操作命令就少多了，如表 5.6 所示。

表 5.6　底行模式下的常用命令

命　令	操 作 说 明
q	退出 vim 程序，如果文件有过修改，则必须先保存文件
q!	强制退出 vim 而不保存文件
x	（exit）保存文件并退出 vim
x!	强制保存文件并退出 vim
w	（write）保存文件，但不退出 vim
w!	对于只读文件，强制保存修改的内容，但不退出 vim
wq	保存文件并退出 vim，同 x
E	在 vim 中创建新的文件，并可为文件命名
N	在本 vim 窗口中打开新的文件
w filename	另存为 filename 文件，不退出 vim
w! filename	强制另存为 filename 文件，不退出 vim
r filename	（read）读入 filename 指定的文件内容插入到光标位置
set nu	在 vim 的每行开头处显示行号
s/pattern1/pattern2/g	将光标当前行的字符串 pattern1 替换为 pattern2
%s/pattern1/pattern2/g	将所有行的字符串 pattern1 替换为 pattern2
g/parttern1/s//parttern2	将所有行的字符串 pattern1 替换为 pattern2
num1,num2 s/pattern1/pattern2/g	将行 num1 到 num2 的字符串 partten1 替换为 partten2
/	查找匹配字符串功能。用“/ 字符串”的命令模式，系统便会自动查找，并突出显示所有找到的字符串，然后转到找到的第一个字符串。如果想继续向下查找，可以按 n 键；向前继续查找则按 N 键
?	也可以使用“? 字符串”查找特定字符串，它的使用与“/ 字符串”相似，但它是向前查找字符串

5.2.2　使用编译工具 gcc

arm-linux-gcc（以下简称 gcc，在实际的操作中将使用 arm-linux-gcc 命令替代下方实例中的 gcc 命令即可）对 C 语言的处理需要经过如下四个步骤。

- 预处理：这一步需要分析各种命令，如#define、#include、#if 等。gcc 调用 cpp 程序来进行预处理工作。
- 编译：这一阶段根据输入文件产生汇编语言，由于通常立即调用汇编程序，所以其输出一般不保存在文件中。gcc 调用 ccl 进行编译工作。
- 汇编：这一步将汇编语言用作输入，产生具有.o 扩展名的目标文件。gcc 调用 as 进

行汇编工作。

- 链接：这一阶段中，各目标文件被放在可执行文件的适当位置上，该程序引用的函数也放在可执行文件中（对使用共享库的程序稍有不同）。gcc 调用链接程序 ld 来完成最终的任务。

和大多数 Shell 命令一样，gcc 的基本使用方式是：

```
gcc [选项] 文件名
```

gcc 可以通过选项对程序的生成进行全面控制，每个选项可以有多种取值，在此只对其中常用部分进行介绍，其余的参数可以参考 gcc 手册或其他专门资料。gcc 常用选项说明如表 5.7 所示。

表 5.7　gcc 常用选项说明

选　　项	说　　明
-c	仅对源文件进行编译，不链接生成可执行文件。在对源文件进行查错或只需产生目标文件时可以使用该选项
-o filename	将经过 gcc 处理过的结果存为 filename，这个结果文件可以是预处理文件、汇编文件、目标文件或最终的可执行文件。假设被处理的源文件为 file1，如果这个选项被忽略，那么生成的可执行文件默认名称为 a.out；目标文件默认名为 file1.o；汇编文件默认名为 file1.s；生成的预处理文件则发送到标准输出设备 stdout
-g 或-gdb	在可执行文件中加入调试信息，方便进行程序的调试。如果使用-gdb 选项，表示加入 gdb 扩展的调试信息，以便使用 gdb 来进行调试
-O[0、1、2、3]	对生成的代码进行优化，括号中的部分为优化级别，默认的情况为 2 级优化，0 为不优化。优化和调试通常不兼容，同时使用-g 和-O 选项经常会使程序产生奇怪的运行结果。所以不要同时使用-g 和-O 选项
-Dmacro[=def]	将名为 macro 的宏定义为 def，如果括号中的部分默认，则宏被定义为 1
-Umacro	某些宏是被编译程序自动定义的。这些宏通常可以指定在其中进行编译的计算机系统类型的符号，用户可以在编译某程序时加上-v 选项以查看 gcc 默认定义了哪些宏。如果用户想取消其中某个宏定义，用-Umacro 选项，这相当于把#undef macro 放在要编译的源文件的开头
-Idir	将 dir 目录加到搜寻头文件的目录列表中去，并优先于在 gcc 默认的搜索目录。在有多个-I 选项的情况下，按命令行上-I 选项的前后顺序搜索。dir 可使用相对路径
-Ldir	将 dir 目录加到搜寻-L 选项指定的函数库文件的目录列表中去，并优先于 gcc 默认的搜索目录。在有多个-L 选项的情况下，按命令行上-L 选项的前后顺序搜索。dir 可使用相对路径
-lname	在链接时使用函数库 name.a，链接程序在-Ldir 选项指定的目录下和/lib，/usr/lib 目录下寻找该库文件。在没有使用-static 选项时，如果发现共享函数库 name.so，则使用 name.so 进行动态链接
-static	禁止与共享函数库链接
-shared	尽量与共享函数库链接，这是链接程序的默认选项

gcc 的命令选项可以组合使用，不过在使用时，每个命令选项都要有一个自己的连字符“-”。如果采用简写的方式，很可能使命令的含义完全不同。

在 Linux 下生成的可执行文件没有固定的扩展名。对于任何符合 Linux 要求的文件名，只要文件的访问属性中有可以执行的属性，该文件就是可以执行的。因此，在使用上面介绍的-o filename 参数时，如果是生成链接后的可执行文件，filename 变量可以取任意一个符合 Linux 要求的文件名。

gcc 命令中的第 2 部分是一个输入给 gcc 命令的文件。gcc 按照命令选项的要求对输入文件进行处理，形成结果输出文件。输入的文件不一定是 C 的源代码文件，还可能是预处理文件、目标文件等。表 5.8 是 gcc 与 C 相关的输入文件扩展名命名规范。

表 5.8　gcc 文件扩展名规范

扩　展　名	类　　型
.c	C 语言源程序，可以被 gcc 预处理、编译、汇编、链接
.C, .cc, .cp, .cpp, .c++, .cxx	C++语言源程序，可以被 gcc 预处理、编译、汇编、链接
.i	预处理后的 C 语言源程序，可以被 gcc 编译、汇编、链接
.ii	预处理后的 C++语言源程序，可以被 gcc 编译、汇编、链接
.s	预处理后的汇编程序，可以被 as 汇编、链接
.S	未预处理的汇编程序，可以被 as 预处理、汇编、链接
.h	头文件，不进行任何操作
.o	编译后的目标文件，传送给 ld
.a	目标文件库，传送给 ld

利用 vim 编写一个命名为 hello.c 的文件，其内容如下，这是一段在屏幕上输出一个字符串的代码：

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

使用 gcc 对该程序编译生成相应的可执行文件，然后试图在 Linux 下执行，会看到提示无法执行二进制文件，这是因为该二进制文件是在嵌入式 Linux 的开发环境下执行的：

```
alloy@ubuntu:～/GT2440/work$ arm-linux-gcc hello.c -o hello
alloy@ubuntu:～/GT2440/work$ ls
hello    hello.c
alloy@ubuntu:～/GT2440/work$ ./hello
-bash: ./hello: 无法执行二进制文件
```

5.2.3　使用调试工具 gdb

在实际开发过程中，除了语法之外还必须考虑设计者的逻辑意图，如果结果不正确，则可以通过相应的调试环境来跟踪调试，本小节将介绍 Linux 中最常用的 gdb 调试环境。

注意：在嵌入式系统的开发中，通常会使用 arm-linux-gdb 运行在主机平台上，成为 gdb 服务器，用以远程调试，用户可以参考相关的资料，在此不在多做赘述。

Linux 包含了一个 gdb 调试程序，gdb 是一个用来调试 C 程序的强力调试器，它使用户能在程序运行时观察程序的内部结构和内存使用情况。gdb 提供了以下一些功能：

- 监视程序中变量的值；
- 设置断点以使程序在指定的代码行上停止执行；
- 一行行地执行代码。

在命令行上输入 gdb 并按回车键就可以运行 gdb 了，如果一切正常，gdb 将被启动并且会在屏幕上看到如下类似内容：

```
alloy@ubuntu:~/GT2440/work$ gdb
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>.
(gdb)
```

1．gdb 的功能介绍

gdb 是功能强大的调试器，支持的调试命令非常丰富，可以实现不同的功能。这些命令包括从简单的文件装入到允许检查所调用的堆栈内容的复杂命令。表 5.9 列出了使用 gdb 调试时会用到的一些命令。如果想了解 gdb 的详细使用知识，可以参考 gdb 的帮助文档。

表 5.9　gdb 的基本命令

命　令	说　明
file	装入想要调试的可执行文件
kill	终止正在调试的程序
list	列出产生执行文件的源代码的一部分
next	执行一行源代码但不进入函数内部
step	执行一行源代码而且进入函数内部
run	执行当前被调试的程序
quit	退出 gdb
watch	动态监视一个变量的值
make	不退出 gdb 而重新产生可执行文件
call name(args)	调用并执行名为 name，参数为 args 的函数
return value	停止执行当前函数，并将 value 返回给调用者
break	在代码里设置断点，使程序执行到此处被挂起

2．gdb 的调用

通常来说，调用 gdb 只需要使用一个参数：

```
gdb <可执行程序名>
```

如果程序运行时产生了错误，会在当前目录下产生核心内存映像 core 文件，可以在指定执行文件的同时为可执行程序指定一个 core 文件：

```
gdb <可执行文件名> core
```

除此之外，还可以为要执行的文件指定一个进程号：

```
gdb <可执行文件名> <进程号>
```

当 gdb 运行时，把任何一个不带选项前缀的参数都作为一个可执行文件或 core 文件或要和被调试的程序相关联的进程号。不带任何选项前缀的参数和前面加了-se 或-c 选项的参数效果一样。gdb 把第一个前面没有选项说明的参数看作前面加了-se 选项，也就是需要调试的可执行文件并从此文件里读取符号表，如果有第二个前面没有选项说明的参数，将被看作是跟着-c 选项后面，也就是需要调试的 core 文件名。

如果不希望看到 gdb 开始的提示信息，可以用 gdb –silent 执行调试工作，通过更多的选项，开发者可以按自己的喜好定制 gdb 的行为。

输入 gdb –help 或-h 可以得到 gdb 启动时的所有选项提示。gdb 命令行中的所有参数都被按照排列的顺序传给 gdb，除非使用了-x 参数。

gdb 的许多选项都可以用缩写形式代表，这可以用-h 查看。在 gdb 中也可以采取任意长度的字符串代表选项，只要保证 gdb 能唯一地识别此参数就行。

表 5.10 列出了 gdb 常用的参数选项。

表 5.10　gdb 常用的参数选项

选　　项	说　　明
-s filename	从 filename 指定的文件中读取要调试的程序的符号表
-e filename	在合时的时候执行 filename 指定的文件，并通过与 core 文件作比较来检查正确的数据
-se filename	从 filename 中读取符号表并作为可执行文件进行调试
-c filename	把 filename 指定的文件作为一个 core 文件
-c num	把数字 num 作为进程号和调试的程序进行关联，与 attach 命令相似
-command filename	按照 filename 指定的文件中的命令执行 gdb 命令，在 filename 指定的文件中存放着一系列的 gdb 命令，就像一个批处理
-d path	指定源文件的路径。把 path 加入到搜索源文件的路径中
-r	从符号文件中一次读取整个符号表，而不是使用默认的方式首先调入一部分符号，当需要时再读入其他一部分。这会使 gdb 的启动较慢，但可以加快以后的调试速度

3．gdb 运行模式的选择

可以用许多模式来运行 gdb，如采用批模式或安静模式。这些模式都是在 gdb 运行时在命令行中通过选项来指定的。

表 5.11 列出了 gdb 运行模式的相关选项。

表 5.11　gdb 运行模式选项

选　　项	说　　明
-n	不执行任何初始化文件中的命令（一级初始化文件叫作.gdbinit）。一般情况下在这些文件中的命令会在所有的命令行参数都被传给 gdb 后执行
-q	设定 gdb 的运行模式为安静模式，可以不输出介绍和版权信息。这些信息在批模式中也不会显示
-batch	设定 gdb 的运行模式为批模式。gdb 在批模式下运行时，会执行命令文件中的所有命令，当所有命令都被成功地执行后 gdb 返回状态 0，如果在执行过程中出错，gdb 返回一个非零值
-cd dir	把 dir 作为 gdb 的工作目录，而非当前目录（一般 gdb 默认把当前目录作为工作目录）

5.2.4　使用管理工具 make

make 是一个工程项目管理工具，其在拥有较多文件的复杂工程项目管理中非常必要，能大大地减少开发者的工作量。

1．make 基础

写一个简单的程序，只有一到两个源文件的时候，输入：

```
gcc file1.c file2.c
```

此时只有两个文件需要编译，工作量似乎不是很大，但是如果有多个文件需要编译呢？一种方法就是使用目标文件，只在源文件有改变的情况下才重新编译源文件，因此可以采用这种方法：

```
gcc file1.o file2.o ... file40.c ...
```

上次编译后，file40.c 发生了改变，但其他文件没有。这样做可以让编译过程快很多，但是也不能解决繁杂的输入问题。当然用户可以使用一个 shell script 来解决输入问题，但是也需要重新编译所有文件。其实可以把以上两种方法结合，写一种像 shell script 一样的代码。这种文件应该包含某种技巧，可以决定什么时候该对源文件进行编译。make 就能实现这样的功能：它读入一个文件，叫 makefile，这个文件不仅决定了源文件之间的依赖关系，还决定了源文件什么时候该编译、什么时候不应该编译。

makefile 通常和相关的源文件保存在同一个目录下，可以叫作 makefile、Makefile 或 MAKEFILE。大多数程序员会使用 Makefile 这个名字，因为这样可以让这个文件被放在目录列表的顶端，可以很容易看见。

2．makefile 的使用

工具程序 make 是 GNU 提供的非常重要的软件开发工具之一，它的本质思想为：检查源代码和目标文件，以确定哪个源文件需要重新编译以创建新的目标文件。make 假设所有改动过的源文件都比已经存在的目标文件新，目标文件的生成依赖于源文件，而它们之间的依赖关系通常都写入一个脚本文件中，这个脚本文件决定着目标文件如何被产生。

下面用一个例子来简单描述如何编写脚本文件及如何使用 make 工具。

make 从 makefile（默认是当前目录下的名为“makefile”的文件）中读取项目的描述。makefile 指定了一系列目标（如可执行文件）和依赖（如对象文件和源文件）的编译规则，其格式如下：

```
target ... :prerequisites...
    command
    ...
```

对每一个目标，make 检查其对应的依赖文件修改时间来确定该目标是否需要利用对应的命令重新建立。注意到，makefile 中命令行必须以单个 TAB 字符进行缩进，不能是空格。

GNU make 包含许多默认的规则（参考隐含规则）来简化 makefile 的构建。例如，它们指定“.o”文件可以通过编译“.c”文件得到，可执行文件可以通过将“.o”链接到一起获

得。隐含规则通过被叫作 make 变量的东西指定，如 CC（C 语言编译器）和 CFLAGS（C 程序的编译选项）。对于 C++，其等价的变量是 CXX 和 CXXFLAGS，而变量 CPPFLAGS 则是编译预处理选项。

现在为 5.2.3 节的 gcc 中的项目写一个简单的 makefile 文件。

```
CC=gcc
CFLAGS=-Wall
hello: hello.o hello_fn.o
clean:
        rm -f hello hello.o hello_fn.o
```

该文件可以这样来读：使用 C 语言编译器 gcc 和编译选项“-Wall”，从对象文件“hello.o”和“hello_fn.o”生成目标可执行文件 hello（文件“hello.o”和“hello_fn.o”通过隐含规则分别由“hello.c”和“hello_fn.c”生成）。目标 clean 没有依赖文件，它只是简单地移除所有编译生成的文件。rm 命令的选项“-f”（force）抑制文件不存在时产生的错误消息。

要使用该 makefile 文件，输入 make。不加参数调用 make 时，makefile 文件中的第一个目标被建立，从而生成可执行文件“hello”，输入 make 命令：

```
$ make
```

将输出编译过程：

```
gcc -Wall -c -o hello.o hello.c
gcc -Wall -c -o hello_fn.o hello_fn.c
gcc hello.o hello_fn.o -o hello
```

最后执行编译生成的可执行文件，输入命令：

```
$ ./hello
Hello，Embedded Linux!
```

一个源文件被修改后要重新生成可执行文件，简单地再次输入 make 即可。通过检查目标文件和依赖文件的时间戳，程序 make 可识别哪些文件已经修改并依据相应的规则更新其对应的目标文件，如先修改 hello.c 文件，输入命令：

```
$ vim hello.c (打开编辑器修改一下文件，也可以直接在 gedit 中修改)
```

然后执行 make 命令：

```
$ make
```

屏幕将输出：

```
gcc -Wall -c -o hello.o hello.c
gcc hello.o hello_fn.o -o hello
```

最后再次执行：

```
$ ./hello
```

```
Hello，Embedded Linux!
```

最后，移除 make 生成的文件，输入 make clean：

```
$ make clean
rm -f hello hello.o hello_fn.o
```

makefile 文件通常可以用于实现安装（make install）和测试（make check）等额外的目标。

5.2.5　使用 autotools

在 5.2.4 小节中介绍了 make 项目管理器的强大功能，不过其编写起来相对烦琐，而使用 autotools 系列工具则可以自动生成 makefile，它只需用户输入简单的目标文件、依赖文件、文件目录等就可以轻松地生成 makefile 了，这无疑是广大用户所希望的。另外，这些工具还可以完成系统配置信息的收集，从而可以方便地处理各种移植性的问题。

autotools 是系列工具，其组成说明如下：

- aclocal；
- autoscan；
- autoconf；
- autoheader；
- automake。

使用 autotools 就是利用各个工具的脚本文件以生成最后的 makefile 的过程，其总体流程包括如下两个主要步骤：

（1）使用 aclocal 生成一个“aclocal.m4”文件，该文件主要处理本地的宏定义；

（2）改写“configure.scan”文件，并将其重命名为“configure.in”，并使用 autoconf 文件生成 configure 文件。

1．autoscan 工具

autoscan 工具会在给定目录及其子目录树中检查源文件，若没有给出目录，就在当前目录及其子目录树中进行检查。它会搜索源文件以寻找一般的移植性问题并创建一个文件“configure.scan”，该文件就是接下来 autoconf 要用到的“configure.in”原型，如下所示：

```
alloeat@ubuntu:/$ autoscan
autom4te: configure.ac: no such file or directory
autoscan: /usr/bin/autom4te failed with exit status: 1
alloeat@ubuntu:/$ ls
autoscan.log    configure.scan    hello.c
```

由上述代码可知 autoscan 首先会尝试去读入“configure.ac”（同 configure.in 的配置文件）文件，此时还没有创建该配置文件，于是它会自动生成一个“configure.in”的原型文件“configure.scan”。

2．autoconf 工具

configure.in 是 autoconf 的脚本配置文件，它的原型文件“configure.scan”，如下所示：

```
#                                                -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.
AC_PREREQ(2.59)
#The next one is modified by david
#AC_INIT(FULL-PACKAGE-NAME,VERSION,BUG-REPORT-ADDRESS)
AC_INIT(hello,1.0)
# The next one is added by david
AM_INIT_AUTOMAKE(hello,1.0)
AC_CONFIG_SRCDIR([hello.c])
AC_CONFIG_HEADER([config.h])
# Checks for programs.
AC_PROG_CC
# Checks for libraries.
# Checks for header files.
# Checks for typedefs, structures, and compiler characteristics.
# Checks for library functions.
AC_CONFIG_FILES([makefile])
AC_OUTPUT
```

在这个脚本文件中，做以下说明。

- 以“#”号开始的行是注释。
- AC_PREREQ 宏声明本文件要求的 autoconf 版本。
- AC_INIT 宏用来定义软件的名称和版本等信息。
- AM_INIT_AUTOMAKE 是 automake 所必备的宏，使 automake 自动生成 makefile.in，也同前面一样，PACKAGE 是所要产生软件套件的名称，VERSION 是版本编号。
- AC_CONFIG_SRCDIR 宏用来检查所指定的源码文件是否存在，以及确定源码目录的有效性。在此处源码文件为当前目录下的 hello.c。
- AC_CONFIG_HEADER 宏用于生成 config.h 文件，以便 autoheader 使用。
- AC_CONFIG_FILES 宏用于生成相应的 makefile 文件。
- 中间的注释之间可以分别添加用户测试程序、测试函数库、测试头文件等宏定义。

接下来首先运行 aclocal，生成一个“aclocal.m4”文件，该文件主要处理本地的宏定义。如下所示：

```
alloeat@ubuntu:/$ aclocal
```

再接着运行 autoconf，生成“configure”可执行文件，如下所示：

```
alloeat@ubuntu:/$ autoconf
alloeat@ubuntu:/$ ls
aclocal.m4  autom4te.cache  autoscan.log  configure  configure.in  hello.c
```

3．autoheader 工具

接着使用 autoheader 命令，其负责生成 config.h.in 文件。该工具通常会从“acconfig.h”文件中复制用户附加的符号定义，因为这里没有附加符号定义，所以不需要创建

“acconfig.h”文件。如下所示：

```
alloeat@ubuntu:/$ autoheader
```

4．automake 工具

这一步是创建 makefile 很重要的一步，automake 要用的脚本配置文件是 makefile.am，用户需要自己创建相应的文件。之后，automake 工具转换成 makefile.in。在该例中，这里创建的文件为 makefile.am，如下所示：

```
AUTOMAKE_OPTIONS=foreign
bin_PROGRAMS= hello
hello_SOURCES= hello.c
```

该脚本文件中的对应项说明如下：

- AUTOMAKE_OPTIONS 为设置 automake 的选项。GNU 对自己发布的软件有严格的规范，如必须附带许可证声明文件 COPYING 等，否则 automake 执行时会报错。automake 提供了 3 种软件等级：foreign、gnu 和 gnits，让用户选择采用，默认等级为 gnu。在本示例中采用 foreign 等级，它只检测必需的文件。
- bin_PROGRAMS 定义要产生的执行文件名。如果要产生多个执行文件，每个文件名之间用空格隔开。
- hello_SOURCES 定义“hello”这个执行程序所需要的原始文件。如果“hello”这个程序是由多个原始文件所产生的，则必须把它所用到的所有原始文件都列出来，并用空格隔开。例如，若目标体“hello”需要“hello.c”、“david.c”、“hello.h”3 个依赖文件，则定义 hello_SOURCES=hello.c david.c hello.h。要注意的是，如果要定义多个执行文件，则对每个执行程序都要定义相应的 file_SOURCES。

接下来可以使用 automake 命令来生成“configure.in”文件，在这里使用选项“-a”（或“—adding-missing”）可以让 automake 自动添加一些必需的脚本文件，如下所示：

```
alloeat@ubuntu:/$ automake –a（或者 automake --add-missing）
configure.in: installing './install-sh'
configure.in: installing './missing'
makefile.am: installing 'depcomp'
alloeat@ubuntu:/$ ls
aclocal.m4        autoscan.log   configure.in   hello.c       makefile.am   missing
autom4te.cache    configure      depcomp        install-sh    makefile.in   config.h.in
```

可以看到，在 automake 之后就可以生成 configure.in 文件。

5．configure 工具

通过运行自动配置设置文件 configure，把 makefile.in 变成了最终的 makefile。如下所示：

```
alloeat@ubuntu:/$ ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
```

```
checking for gcc... gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ANSI C... none needed
checking for style of include used by make... GNU
checking dependency style of gcc... gcc3
configure: creating ./config.status
config.status: creating makefile
config.status: executing depfiles commands
```

可以看到，在运行 configure 时收集了系统的信息，用户可以在 configure 命令中对其进行方便的配置。./configure 的自定义参数有两种：一种是开关式（--enable-XXX 或--disable-XXX）；另一种是开放式，即后面要填入一串字符（--with-XXX=yyyy）参数。读者可以自行尝试其使用方法，此外用户还可以查看同一目录下的“config.log”文件，以方便调试之用。

到此为止，makefile 就可以自动生成了。回忆整个步骤，用户不再需要定制不同的规则，而只需要输入简单的文件及目录名即可，这样就大大方便了用户的使用。autotools 生成 makefile 的流程如图 5.6 所示。

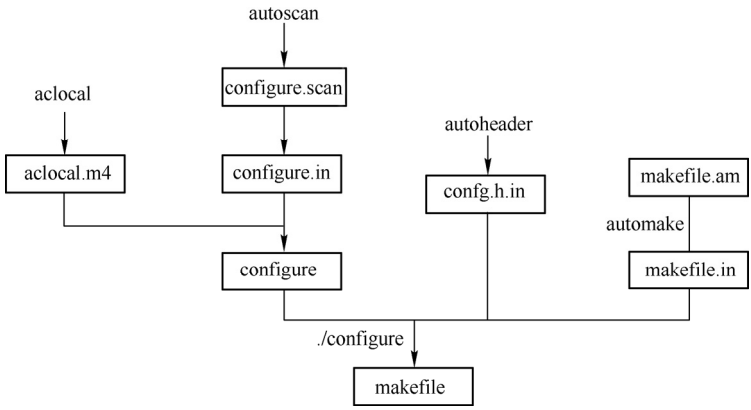


图 5.6　autotools 生成 makefile 的流程图

6．使用 autotools 所生成的 makefile

autotools 生成的 makefile 除具有普通的编译功能外，还具有以下主要功能（感兴趣的读者可以查看这个简单的 hello.c 程序的 makefile）。

- make 功能：输入 make 后默认执行“make all”命令，即目标体为 all，其执行情况如下所示：

```
alloeat@ubuntu:/$ make
if gcc -DPACKAGE_NAME=\"\" -DPACKAGE_TARNAME=\"\" -DPACKAGE_VERSION=\"\" -
DPACKAGE_STRING=\"\" -DPACKAGE_BUGREPORT=\"\" -DPACKAGE=\"hello\"-DVERSION=\"1.0\"  -I. -
```

```
I. -g -O2 -MT hello.o -MD -MP -MF ".deps/hello.Tpo" -c -o hello.o hello.c; \
        then mv -f ".deps/hello.Tpo" ".deps/hello.Po"; else rm -f ".deps/hello.Tpo"; exit 1; fi
        gcc    -g -O2      -o hello    hello.o
```

此时在本目录下就生成了可执行文件“hello”，运行“./hello”能出现正常结果，如下所示：

```
        alloeat@ubuntu:/$ ./hello
        Hello!Autoconf!
```

- make install 功能：此时，会把该程序安装到系统目录中去，如下所示：

```
        alloeat@ubuntu:/$ make install
        if gcc -DPACKAGE_NAME=\"\" -DPACKAGE_TARNAME=\"\" -DPACKAGE_VERSION=\"\" -
DPACKAGE_STRING=\"\" -DPACKAGE_BUGREPORT=\"\" -DPACKAGE=\"hello\" -DVERSION=\"1.0\"   -I.
-I. -g -O2 -MT hello.o -MD -MP -MF ".deps/hello.Tpo" -c -o hello.o hello.c; \
        then mv -f ".deps/hello.Tpo" ".deps/hello.Po"; else rm -f ".deps/hello.Tpo"; exit 1; fi
        gcc    -g -O2      -o hello    hello.o
        make[1]: Entering directory '/root/workplace/automake'
        test -z "/usr/local/bin" || mkdir -p -- "/usr/local/bin"
          /usr/bin/install -c 'hello' '/usr/local/bin/hello'
        make[1]: Nothing to be done for 'install-data-am'.
        make[1]: Leaving directory '/root/workplace/automake'
```

此时，若直接运行 hello，也能出现正确的结果，如下所示：

```
        alloeat@ubuntu:/$ hello
        Hello!Autoconf!
```

- make clean 功能：此时，make 会清除之前所编译的可执行文件及目标文件（object file, *.o），如下所示：

```
        alloeat@ubuntu:/$ make clean
        test -z "hello" || rm -f hello
        rm -f *.o
```

- make dist 功能：make 将程序和相关的文档打包为一个压缩文档以供发布，如下所示：

```
        alloeat@ubuntu:/$ make dist
        alloeat@ubuntu:/$ ls hello-1.0-tar.gz
        hello-1.0-tar.gz
```

可见该命令生成了一个 hello-1.0-tar.gz 压缩文件。

autotools 是软件维护与发布的必备工具，鉴于此，如今 GUN 的软件一般都是由 automake 来制作的。

第6章

在嵌入式系统上移植操作系统和文件系统

在嵌入式系统上移植操作系统和文件系统是在嵌入式系统上部署操作系统的最后一步，本章将详细介绍其操作过程，涉及的内容包括：

- Linux 内核的源代码组织结构；
- Linux 内核的配置方法；
- Linux 内核的移植过程；
- 文件系统的基本概念及移植过程。

6.1 Linux 内核移植基础

6.1.1 Linux 的内核组成

Linux 用来支持各种体系结构的源代码，包含大约 4500 个 C 语言程序，存放在约 270 个子目录下，总共大约包含 200 万行代码，大概占用 58MB 磁盘空间。其文件结构图如图 6.1 所示，这里使用的 Linux 系统是 2.6 的内核版本，其他版本可能会有所差异。

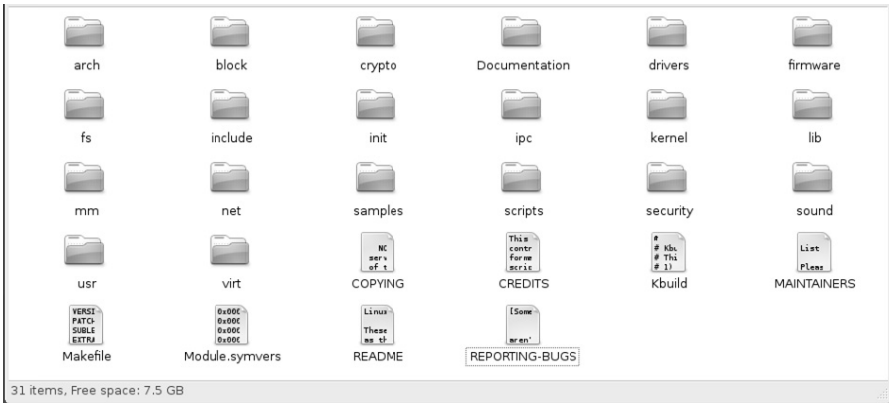


图 6.1 Linux 内核文件结构图

注意：在第 3 章中已经介绍过 Linux 的版本概念，截至 2013 年年末 Linux Kernel 内核版本已经发布到了 3.13，从稳定性和易于获取性出发，本书基于 2.6.38 版本进行介绍。

在 Linux 的内核中可以看到 Linux 的具体管理方法和结构组织，如进程管理、内存管理、文件系统等与内核源码的各个目录都是对应的，如有关驱动的内容，内核中就都组织到“drive”这个目录中去，有关网络的代码都集中组织到“net”中。当然，这里有的目录包含多个部分的内容，具体各个目录的内容组成如下。

- arch：arch 目录包括了所有和体系结构相关的核心代码。它下面的每一个子目录都代表一种 Linux 支持的体系结构，如 i386 就是 Intel CPU 及与之兼容体系结构的子目录。
- include：include 目录包括编译核心所需要的大部分头文件，如与平台无关的头文件在 include/linux 子目录下。
- init：init 目录包含核心的初始化代码（不是系统的引导代码），有 main.c 和 Version.c 两个文件。
- mm：mm 目录包含了所有内存管理代码。与具体硬件体系结构相关的内存管理代码位于 arch/*/mm 目录下。
- drivers：drivers 目录中是系统中的所有设备驱动程序。它又进一步划分成几类设备驱动，每一种有对应的子目录，如声卡的驱动对应于 drivers/sound。
- ipc：ipc 目录包含了核心进程间的通信代码。
- modules：modules 目录存放了已建好的、可动态加载的模块。
- fs：fs 目录存放 Linux 支持的文件系统代码。不同的文件系统有不同的子目录对应，如 ext3 文件系统对应的就是 ext3 子目录。
- Kernel：Kernel 内核管理的核心代码放在这里。同时与处理器结构相关的代码都放在 arch/*/kernel 目录下。
- net：net 目录里是核心的网络部分代码，其每个子目录对应网络的一个方面。
- lib：lib 目录包含了核心的库代码，不过与处理器结构相关的库代码被放在 arch/*/lib/ 目录下。
- scripts：scripts 目录包含用于配置核心的脚本文件。
- documentation：documentation 目录下是一些文档，是对每个目录作用的具体说明。一般在每个目录下都有一个 depend 文件和一个 Makefile 文件。这两个文件都是编译时使用的辅助文件。仔细阅读这两个文件，对弄清各个文件之间的联系和依托关系很有帮助。另外，有的目录下还有 Readme 文件，它是对该目录下文件的一些说明，同样有利于对内核源码的理解。

6.1.2　Linux 内核的配置工具

Linux 内核在普通 PC 操作系统下以磁盘文件的形式存在，即“映像文件”，该内核映像文件会刻录到目标板的 Flash 中。Linux 的内核映像文件有非压缩版本 Image 和压缩版本 zImage 两种形式，其中 zImage 是 Image 经过压缩形成的，所以它的大小比 Image 小。为了

能使用 zImage 这个压缩版本，必须在它的开头加上解压缩的代码，将 zImage 解压缩之后才能执行，因此它的执行速度比 Image 要慢。但考虑到嵌入式系统的存储空间容量一般都比较小，内核要常驻内存，采用 zImage 可以占用较少的存储空间，因此牺牲一点性能上的代价也是值得的，所以一般嵌入式系统均采用压缩的内核映像文件，即 zImage。

基于 ARM 的嵌入式 Linux 配置系统由三部分组成：Makefile 文件、配置文件和配置工具，其关系如图 6.2 所示。从其中可以看到扩展名为.in 的文件为提供选项的文件，通过配置工具配置之后生成配置文件，最后按照选项来调用源码，编译成待刻录到目标板的镜像文件 zImage。整个过程都是由 Makefile 文件来调用管理的。

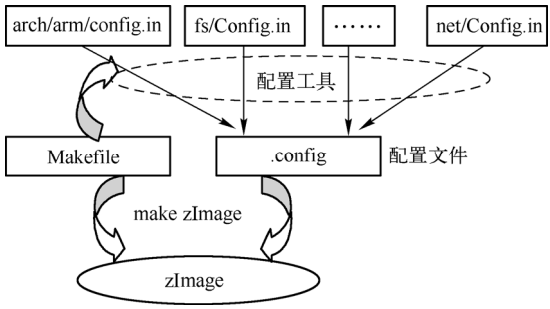


图 6.2　基于 ARM 的 Linux 内核配置组成

- Makefile 文件：其是 Linux 系统中非常重要的一个组成部分，前面也有所介绍。因为在几乎每一个子目录下都会有 Makefile 文件。其中位于根目录下的 Makefile 文件是总纲式 Makefile 文件，其他 Makefile 文件都直接或间接地被它调用。Makefile 文件定义了各个目录下文件如何被编译，并最终形成 zImage 文件。当然 zImage 文件的产生还要借助.config 文件，它会告诉 Makefile 文件哪些文件被编译进内核，哪些源文件没有被用户选中，并不需要被编译进内核文件中。
- 在基于 ARM 的嵌入式 Linux 系统中，配置文件存放在各个子目录下，它们通常被称为 config.in、Config.in、config 或 Config 文件，其中扩展名.in 表示提供选项，而扩展名.config 则表示选择了某些选项之后的配置文件。这些文件大概有几十个，其中存放在 arch/arm 目录下的 config.in 文件为总纲领式配置文件，其他 config.in 文件都直接或间接地被该文件调用。这些配置文件按照一定的格式编写，用户通过特定的工具可以读这些配置文件来进行 ARM-Linux 系统的配置，最终的配置选项结果存放在内核根目录.config 文件中。
- 配置工具一般包括配置命令解释器和配置用户界面。前者的主要作用是对配置脚本中使用的配置命令进行解释；而后者则提供基于字符界面、Ncurses 图形界面及 Xwindows 图形界面的用户配置界面，各自对应于 Make config、Make menuconfig 和 Make xconfig。这些配置工具都是使用脚本语言，如 Tcl/Tk、Perl 编写的（也包含一些用 C 语言编写的代码）。这里并不对配置系统本身进行分析，而是重点介绍如何使用配置系统。所以除非是配置系统的维护者，一般的内核开发者无须了解它们的原理，只需知道如何编写 Makefile 和配置文件就可以了。

6.2　【应用实例】——在嵌入式系统上移植 Linux 内核

嵌入式 Linux 内核的内核移植是通过 make 的不同命令来实现的，其执行配置文件即在 5.2.4 节中介绍的 makefile。Linux 内核中不同的目录结构里都有相应的 makefile，而不同的 makefile 又通过彼此之间的依赖关系构成统一的整体，共同完成建立依赖关系、建立内核等功能。

内核的编译根据不同的情况会有不同的步骤，但其中最主要的三个步骤分别为：配置内核、建立依赖关系、创建内核映像，除此之外还有一些辅助功能，如清除文件和依赖关系等。

6.2.1　配置内核

内核配置中的选项主要是用户用来为目标板选择处理器架构的选项，不同的处理器架构会有不同的处理器选项，ARM 处理器是其专用的选项，如“Multimedia capabilities port drivers”等，所以在此之前必须确保在根目录中 makefile 里“ARCH”的值已设定了目标板的类型，如：

```
ARCH            := arm
```

接下来就可以进行内核配置了，内核支持四种不同的配置方法，这几种方法只是与用户交互的界面不同，其实现的功能是一样的。每种方法都会通过读入一个默认的配置文件——根目录下“.config”隐藏文件（用户也可以手动修改该文件，但不推荐使用）。用户也可以自己加载其他配置文件，也可以将当前的配置保存为其他名字的配置文件。这四种方式说明如下。

- make config：基于文本的最为传统的配置界面，不推荐使用。
- make menuconfig：基于文本选单的配置界面，在字符终端下推荐使用。
- make xconfig：基于图形窗口模式的配置界面，Xwindow 下推荐使用。
- make oldconfig：自动读入“.config”配置文件，并且只要求用户设定前次没有设定过的选项。

在这四种方式中，make menuconfig 的使用最为广泛，下面就以 make menuconfig 为例进行讲解，其配置界面如图 6.3 所示。

从该图中可以看出，Linux 内核允许用户对其各类功能逐项配置，一共有 17 类配置选项，通常来说包括：

- CPU 选项配置；
- LCD 显示器配置；
- 触摸屏配置；
- USB 鼠标和键盘的配置；
- U 盘的配置；
- USB 摄像头的配置；
- 有线网卡驱动的配置；
- 无线网卡的配置；

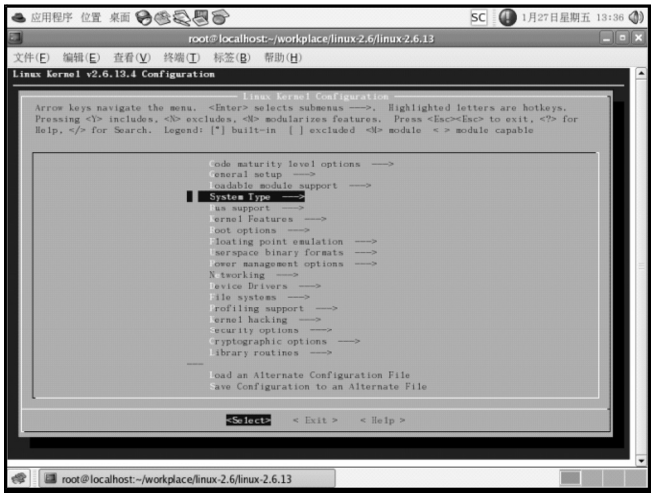


图 6.3　make menuconfig 的配置界面

- 声卡的配置；
- SD/MMC 卡的配置；
- 看门狗的配置；
- I/O 引脚的配置；
- A/D 模块的配置；
- 串口模块的配置；
- 实时时钟的配置；
- E^2PROM 的配置；
- 文件系统的配置。

在 menuconfig 的配置界面中是纯键盘的操作，用户可使用上下键和 Tab 键移动光标以进入相关子项，图 6.4 是用于对处理器进行配置的 System Type 子项界面。

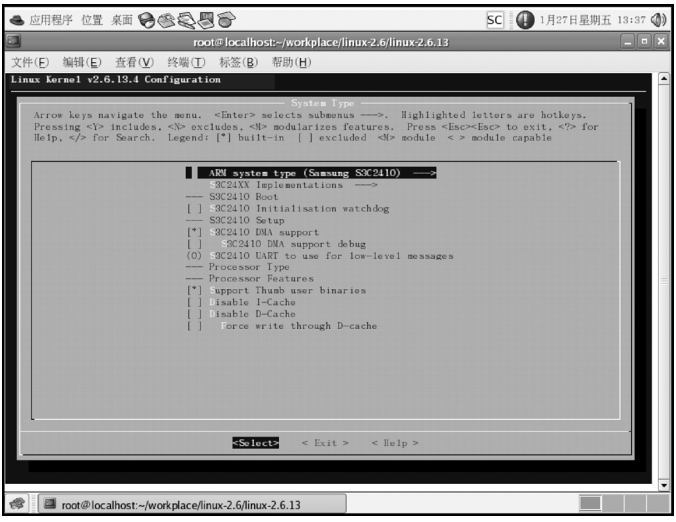


图 6.4　System Type 子项界面

从图中可以看到，每个选项前都有个括号，可以通过按空格键或 Y 键表示包含该选项，按 N 键表示不包含该选项。另外，界面中提供了 3 种括号：中括号、尖括号和圆括号。在用空格键选择相应的选项时，可以发现中括号里要么是空，要么是“*”；尖括号里可以是空，“*”和“M”分别表示包含选项、不包含选项和编译成模块；圆括号的内容是要求用户在所提供的几个选项中选择一项。

此外需要注意，不同版本的 Linux 内核在串口命名方面有一个区别：在 2.4 及之前的版本内核中“COM1”对应的是“ttyS0”，而在 2.4 版以后的内核中“COM1”对应“ttySAC0”，因此在启动参数的子项时要格外注意，如图 6.5 所示。

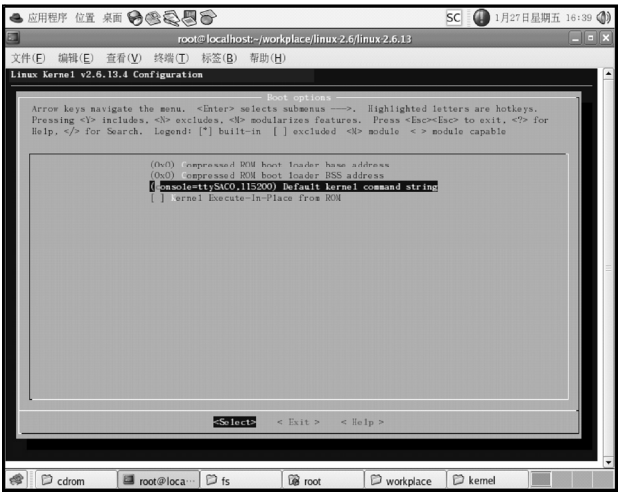


图 6.5　串口的配置

一般情况下，使用厂商提供的默认配置文件都能正常运行，所以在初次使用时可以不对其进行额外的配置，在以后需要使用其他功能时再另行添加，这样可以大大减小出错的概率，有利于错误定位。在完成配置之后，就可以保存并退出，如图 6.6 所示。

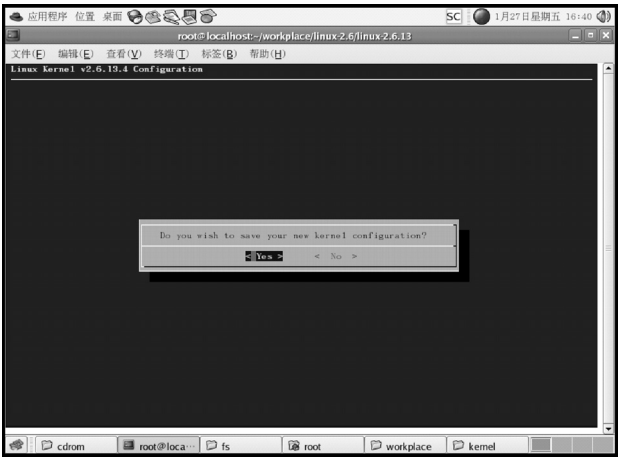


图 6.6　保存并退出

6.2.2 建立依赖关系

由于内核源码树中的大多数文件都与一些头文件有依赖关系，因此要顺利建立内核，内核源码树中的每个 Makefile 都必须知道这些依赖关系。建立依赖关系通常在第一次编译内核的时候（或源码目录树的结构发生变化的时候）进行，它会在内核源码树中每个子目录产生一个“.depend”文件。运行“make dep”即可。编译 2.6 版本的内核通常不需要这个过程，直接输入“make”即可。

6.2.3 建立内核

建立内核可以使用“make”、“make zImage”或“make bzImage”，这里建立的为压缩的内核映像。通常在 Linux 中，内核映像分为压缩的内核映像和未压缩的内核映像。其中，压缩的内核映像通常名为 zImage，位于“arch/$（ARCH）/boot”目录中。而未压缩的内核映像通常名为 vmlinux，位于源码树的根目录中。

到这一步就完成了内核源代码的编译，此时用户可以使用 4.4.2 节中所介绍的 DNW 软件将内核压缩文件下载到嵌入式系统中。

6.3 文件系统移植基础

在嵌入式操作系统中，文件系统和内核是完全独立的两个部分，加载根文件系统是嵌入式 Linux 启动中不可缺少的一部分；嵌入式系统的操作系统不能脱离文件系统独立工作，当没有在嵌入式系统上移植文件系统的时候，会发生如图 6.7 所示的“加载文件系统错误”。

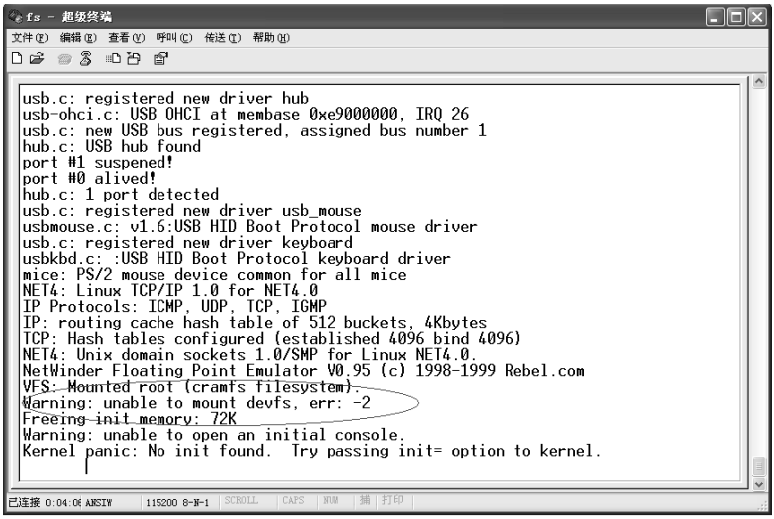


图 6.7 加载文件系统错误

本节以 Linux 为例来对文件系统的基础知识进行介绍。文件系统是指在一个物理设备上的任何文件组织和目录，在 Linux 系统中则主要用来存储操作系统运行所必需的信息，构成

了操作系统上所有数据的基础。Linux 系统中的文件不仅包括普通的文件和目录，每个和设备相关的实际实体也都被映射为一个文件，如磁盘、终端、打印机、网卡等，这些设备文件称为特殊文件，所以 Linux 下的文件又担负着操作系统服务和设备的统一接口。

6.3.1　Linux 文件系统基础

Linux 支持多种文件系统，包括 ext2、ext3、vfat、ntfs、iso9660、JFFS、YAFFS/YAFFS2、Romfs 和 NFS 等，为了对各类文件系统进行统一管理，Linux 引入了虚拟文件系统 VFS（Virtual File System），为各类文件系统提供了一个统一的操作界面和应用编程接口。Linux 的文件系统结构如图 6.8 所示，可以分为用户层、内核层和底层驱动。

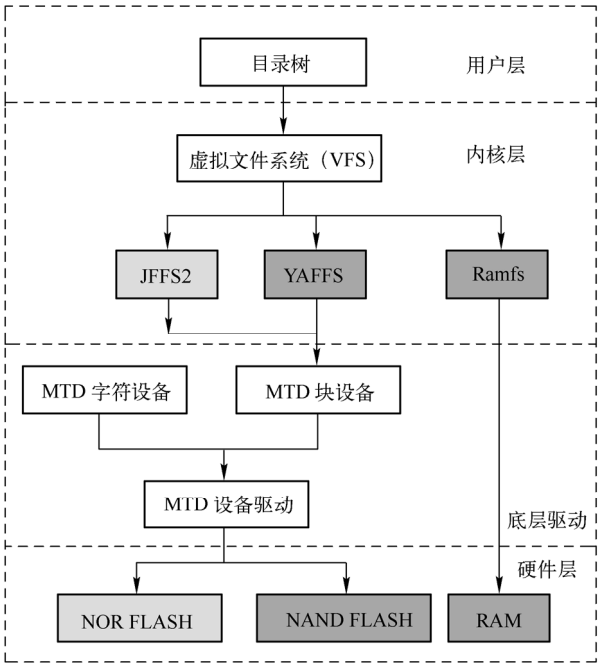


图 6.8　Linux 的文件系统结构

1．用户层

用户层包含了一些应用程序和 GNU C 库（glibc），用来为文件系统调用提供用户接口，如打开、读取、写入和关闭等。系统调用接口就像交换器，负责将系统调用从用户空间发送到内核空间中的适当端点。系统调用实际上通过调用内核虚拟文件系统提供的统一接口来完成各种设备的使用。

2．内核层

内核层的核心是虚拟文件系统，其把各种具体的文件系统的公共部分抽取出来，形成一个抽象层，是系统内核的一部分。VFS 位于用户程序和具体的文件系统之间，为用户程序提供了标准的文件系统调用接口。对于具体的文件系统，它通过一系列对不同文件系统公用的函数指针来实际调用具体的文件系统函数，完成各种不同的操作。通过这种方式，VFS 就可以对用户屏蔽底层文件系统的实现细节。

在 VFS 下则是各种实际文件系统，包括 JFFS2、YAFFS、Ramfs 等，对这些文件系统 VFS 提供了很好的通用的接口，使系统屏蔽了不同文件系统对于应用程序的差异性。各种具体的文件系统操作都可以按照自己的方式实现，如 YAFFS 文件系统和 JFFS2 系统都有各自的实现方式。目前已经稳定支持的文件系统包括 ext、ext2、ext3、vfat、iso9660、proc、NFS、JFFS、JFFS2、SMB、ReisterFS、YAFFS、Cramfs、Romfs 等。

系统启动时，第一个必须挂载的是根文件系统。若系统不能从指定设备上挂载根文件系统，则系统会因出错而退出启动。之后可以自动或手动挂载其他文件系统。因此，通过用户的手动挂载，可以在当前的系统中挂载不同的文件系统，从而实现在一个系统中同时存在不同的文件系统。

3．底层驱动

底层驱动涉及的则是一些具体的设备驱动，包括字符设备、块设备等，它们会直接和底层的存储器硬件相关。

4．嵌入式文件系统的特性

不同的文件系统类型有不同的特点，要根据存储设备的硬件特性、系统需求有选择地采用，在嵌入式 Linux 的应用中，主要的存储设备为 RAM（DRAM、SDRAM）和常用 Flash 存储器，常用的基于存储设备的文件系统类型包括 Jffs2、Yaffs、Cramfs、Romfs、Ramdisk、Ramfs/Tmpfs 等。嵌入式设备的一些特殊性使得嵌入式文件系统除了满足一般文件系统的基本要求外，还有一些自身的特性，具体如下：

- 文件系统面对的存储介质特殊；
- 文件系统有快速恢复的特殊要求；
- 物理文件系统的多样性和动态可装配性；
- 需要文件系统具有跨操作平台的安全性；
- 文件系统要能满足整个系统的实时性要求。

另外，嵌入式文件系统还具有安全性和均衡负载这样的要求，而日志型文件系统可以很好地解决安全性的问题。目前，日志型的嵌入式文件系统已成为嵌入式文件系统的主流。

6.3.2 文件系统的管理机制

构建适用于嵌入式系统的 Linux 文件系统，必然会涉及两个关键点，一是文件系统类型的选择，它关系到文件系统的读写性能、尺寸大小；另一个就是根文件系统内容的选择，它关系到根文件系统所能提供的功能及尺寸大小。

在嵌入式设备中，使用的存储器是像 Flash 闪存芯片、小型闪存卡等专为嵌入式系统设计的存储装置。Flash 闪存是目前嵌入式系统中广泛采用的主流存储器，主要特点是按整体/扇区擦除和按字节编程，具有低功耗、高密度、小体积等优点。目前，主要有 NOR 和 NAND 两种类型。

所有嵌入式系统的启动都至少需要使用某种形式的永久性存储设备，它们需要合适的驱动程序，当前在嵌入式 Linux 中有三种常用的块驱动程序可以选择，分别是以下三类。

1．Blkmem 驱动层

Blkmem 驱动是为 uClinux 专门设计的，也是最早的块驱动程序之一，现在仍然有很多

嵌入式 Linux 操作系统选用它作为块驱动程序，尤其是在 uClinux 中。它相对来说是最简单的，而且只支持建立在 NOR 型 Flash 和 RAM 中的根文件系统。使用 Blkmem 驱动，建立 Flash 分区配置比较困难，这种驱动程序为 Flash 提供了一些基本擦除/写操作。

2．RAMdisk 驱动层

RAMdisk 驱动层通常应用在标准 Linux 中，无盘工作站的启动，对 Flash 存储器并不提供任何直接支持。RAMdisk 就是在开机时把一部分的内存虚拟成块设备，并且把之前所准备好的档案系统映像解压缩到该 RAMdisk 环境中。当在 Flash 中放置一个压缩的文件系统时，可以将文件系统解压到 RAM，使用 RAMdisk 驱动层支持一个保持在 RAM 中的文件系统。

3．MTD 驱动层

为了尽可能避免针对不同的技术使用不同的工具，以及为不同的技术提供共同的能力，Linux 内核纳入了 MTD（Memory Technology Device）子系统。它提供了一致且统一的接口，让底层的 MTD 芯片驱动程序无缝地与较高层接口组合在一起。JFFS2、Cramfs、YAFFS 等文件系统都可以被安装成 MTD 块设备。MTD 驱动也可以为那些支持 CFI 接口的 NOR 型 Flash 闪存提供支持。虽然 MTD 可以建立在 RAM 上，但它是专为基于 Flash 的设备而设计的。MTD 包含特定 Flash 闪存芯片的驱动程序，开发者要选择适合自己系统的 Flash 闪存芯片驱动。Flash 闪存芯片驱动向上层提供读、写、擦除等基本的操作，MTD 对这些操作进行封装后向用户层提供 MTD char 和 MTD block 类型的设备。

MTD 驱动层也支持在一块 Flash 闪存上建立多个分区，每一个分区作为一个 MTD block 设备，可以把系统软件和数据等分配到不同的分区上，同时可以在不同的分区采用不同的文件系统格式。这一点非常重要，正是这一点为嵌入式系统多文件系统的建立提供了灵活性。

6.3.3　嵌入式系统中的常用文件系统介绍

嵌入式系统中常用的文件系统可以分为基于 Flash 的文件系统、基于 RAM 的文件系统和网络文件系统三大类。

1．基于 Flash 的文件系统

Flash 闪存是嵌入式系统的主要存储媒介，有其自身的特性。Flash 闪存的写入操作只能把对应位置的“1”修改为“0”，而不能把“0”修改为“1”（擦除就是把对应存储块的内容恢复为 1），因此向 Flash 闪存写入内容时需要先擦除对应的存储区间，这种擦除是以块（block）为单位进行的，Flash 可以分为 Nor 和 Nand 两大类，其详细说明和区别可以参考 2.4.2 节内容。

在 Linux 系统中，文件系统是针对存储器分区而言的，而非存储芯片。这是因为，Flash 可以以分区为单位拆开或合并使用。例如，一块 Flash 闪存芯片可以被划分为多个分区，各分区可以采用不同的文件系统，而两块 Flash 闪存芯片也可以合并为一个分区使用，采用同一个文件系统。

在嵌入式 Linux 中，常见的基于 Flash 的文件系统包括：JFFS2（Journalling Flash File System 2）、YAFFS/YAFFS2（Yet Another Flash File System）、ROMFS（ROM File System）

和 Cramfs（Compressed ROM File System）。

- JFFS2：日志闪存文件系统版本 2，主要用于 NOR 型闪存，基于 MTD 驱动层。JFFS2 是 RedHat 公司基于 JFFS 开发的闪存文件系统，最初是针对 RedHat 公司的嵌入式产品 eCos 开发的嵌入式文件系统，所以 JFFS2 也可以用在 Linux、uCLinux 中。而其前身 JFFS 文件系统最早是由瑞典 Axis Communications 公司基于 Linux 2.0 的内核为嵌入式系统开发的文件系统。JFFS2 是可读写的、支持数据压缩、基于哈希表的日志型文件系统，并提供了崩溃/掉电安全保护，提供“写平衡”支持，支持多种节点类型，提高了对 Flash 的利用率。目前，JFFS2 文件系统是 Flash 设备上最为流行的文件系统格式。但是，JFFS2 也存在不容忽视的缺点，就是当文件系统已满或接近满时，JFFS2 的运行会非常慢，主要是因为垃圾收集的关系而使 JFFS2 的运行速度大大放慢。目前 JFFS3 正在开发中，将这些统称为 JFFSx。关于 JFFSx 系列文件系统的使用详细文档可参考 MTD 补丁包中的 mtd-jffs-HOWTO.txt。JFFSx 不适合用于 NAND 闪存，主要是因为 NAND 闪存的容量一般较大，这将导致 JFFSx 为维护日志节点所占用的内存空间迅速增大，另外，JFFSx 文件系统在挂载时需要扫描整个 Flash 的内容，以找出所有的日志节点，建立文件结构，对于大容量的 NAND 闪存会耗费大量时间。
- Yaffs/Yaffs2：专为嵌入式系统使用 NAND 型闪存而设计的一种日志型文件系统。与 JFFS2 文件系统相比，它减少了一些功能，如不支持数据压缩等，所以速度更快，挂载时间很短，对内存的占用较小。另外，它还是跨平台的文件系统，除了 Linux 和 eCos 外，还支持 WinCE、pSOS 和 ThreadX 等操作系统。Yaffs/Yaffs2 自带 NAND 芯片的驱动，并且为嵌入式系统提供了直接访问文件系统的 API，用户可以不使用 Linux 中的 MTD 与 VFS，直接对文件系统操作。当然，Yaffs 也可与 MTD 驱动程序配合使用。Yaffs 与 Yaffs2 的主要区别在于：前者仅支持小页（512B）NAND 闪存，后者则可支持大页（2KB）NAND 闪存。同时，Yaffs2 在内存空间的占用、垃圾回收速度、读/写速度等方面均有大幅改善。由于 JFFS2 在 NAND 闪存上表现不稳定，更适合于 NOR 内存，所以相对于大容量的 NAND 闪存来说，Yaffs 是更好的选择。
- Romfs：Romfs 是一种简单的、紧凑的、只读的、传统型的文件系统。它不支持动态擦写保存，按顺序存放数据，因而只支持应用程序以 XIP（eXecute In Place，片内运行）方式运行，在系统运行时，这种运行方式节省 RAM 空间。uClinux 操作系统通常采用 Romfs 文件系统。
- Cramfs：其是 Linux 的创始人 Linus Torvalds 参与开发的一种只读的压缩文件系统。它也基于 MTD 驱动程序。在 Cramfs 文件系统中，每一页（4KB）被单独压缩，可以随机进行页访问，其压缩比高达 2∶1，为嵌入式系统节省了大量的 Flash 存储空间，使系统可通过更低容量的内存存储相同的文件，从而降低了系统成本。Cramfs 文件系统以压缩方式存储，在运行时解压缩，所以不支持应用程序以 XIP 方式运行，所有的应用程序要求被复制到 RAM 里去运行，但这并不代表比 Romfs 需求的 RAM 空间大，因为 Cramfs 是采用分页压缩的方式存放档案的，在读取档案时，不会一下

子就耗用过多的内存空间，只针对目前实际读取的部分分配内存，尚没有读取的部分不分配内存空间，当所读取的档案不在内存时，Cramfs 文件系统会自动计算压缩后的资料所存的位置，再即时解压缩到 RAM 中。另外，它速度快、效率高，其只读的特点有利于保护文件系统免受破坏，提高了系统的可靠性。由于以上特性，Cramfs 在嵌入式系统中应用广泛。但是它的只读属性同时又是它的一大缺陷，使得用户无法对其内容进行扩充。Cramfs 映像通常存放在 Flash 闪存中，但是也能存放在其他文件系统里，使用 Loopback 设备可以把它安装在其他文件系统里。

2．基于 RAM 的文件系统

RAM 驱动器实际上是把系统内存划出一部分当作存储器使用。对于操作系统来讲，内存的存取速度远远大于磁盘或 Flash 闪存。所以 RAM 驱动器肯定要比闪存快得多。用户可以把整个应用程序都安装在 RamDisk 的驱动器中，然后用内存的速度运行它，通常来说，嵌入式系统进行裸机调试的时候即采用此种方式，常见的基于 RAM 的文件系统包括 Ramdisk 和 Ramfs/Tmpfs。

- Ramdisk：其存在于 RAM 中，但功能同块设备类似。操作系统内核可以在同一时间支持多个活动的 Ramdisk。因为它们的功能犹如块设备，所以 Ramdisk 上可以使用任何磁盘系统。由于 Ramdisk 上的内容将因系统的重新开机而丢失，所以 Ramdisk 通常会从经压缩的磁盘文件系统（如 ext2）加载其内容，这就是经压缩的 Ramdisk 镜像。这类经压缩的 Ramdisk 镜像特别适合在嵌入式 Linux 系统初始化期间应用。也就是说，内核具备从存储设备中取出 Initrd 镜像作为它的根文件系统的能力。在 Linux 的启动阶段，Initrd 提供了一套机制，可以将内核映像和根文件系统一起载入内存。启动时，内核会确认引导选项是否有指示 Initrd 的存在。如果有，内核会从所选定的存储设备中取出文件系统镜像放入 Ramdisk，并且作为根文件系统。Initrd 机制是为内核提供根文件系统的最简单的方法。ext2 是 Ramdisk 中最常用的文件系统。Ramdisk 将一部分大小固定的内存当作分区来使用。它并非一个实际的文件系统，而是一种将实际的文件系统装入内存的机制，并且可以作为根文件系统。将一些经常被访问而又不会更改的文件（如只读的根文件系统）通过 Ramdisk 放在内存中，可以明显地提高系统的性能。
- Ramfs/Tmpfs：其是 Linus Torvalds 开发的一种基于内存的文件系统，工作于虚拟文件系统（VFS）层，不能格式化，可以创建多个，在创建时可以指定其最大能使用的内存大小。实际上，VFS 本质上可看成一种内存文件系统，它统一了文件在内核中的表示方式，并对磁盘文件系统进行了缓冲。Ramfs 是一个非常简单的文件系统，它输出 Linux 的磁盘缓存机制（页缓存和目录缓存），作为一个大小动态的基于内存的文件系统。通常，所有的文件由 Linux 缓存在内存中。页的数据从保持在周围以防再次需要的后备存储（一般被挂载的是块设备文件系统）中读取，并标记为可用以防虚拟内存系统（Virtual Memory System）需要这些内存作为别用。类似地，在数据写回后备存储时，数据一写回文件就立即被标记为可用，但周围的缓存被保留着直至虚拟机（VM）重新分配内存。Ramfs 并没有后备存储，文件写入 Ramfs 时像往常一样分配目录和页的缓存，但这里并没有地方可写回它们。这意味

着页的数据不再标记为可用，因此当希望回收内存时，内存不能通过 VM 释放。实现 Ramfs 所需的代码总量是极少的，因为所有的工作由现有的 Linux 缓存结构来完成。实际上，现在正挂载磁盘缓存作为一个文件系统。据此，Ramfs 并不是一个可通过菜单配置项来卸载的可选组件，它可节省的空间是微不足道的。Ramfs/Tmpfs 文件系统把所有的文件都放在 RAM 中，所以读/写操作发生在 RAM 中，可以用 Ramfs/Tmpfs 来存储一些临时性或经常要修改的数据，如/tmp 和/var 目录，这样既避免了对 Flash 存储器的读/写损耗，又提高了数据读/写速度。Ramfs/Tmpfs 相对于传统的 Ramdisk 的不同之处主要在于：不能格式化，文件系统大小可随所含文件内容的大小变化。Tmpfs 的一个缺点是当系统重新引导时会丢失所有数据。

3．网络文件系统

NFS（网络文件系统，Network File System）是 1984 年由 Sun 开发并发展起来的一项在不同机器、不同操作系统之间通过网络共享文件的技术。NFS 是一个 RPC（远程过程调用）服务，它的目的是在不同的系统间使用，所以它的通信协议与主机及作业系统无关。当用户想使用某个远程文件时，只要运行挂载命令“mount”，就可以将远程的文件系统安装在自己的文件系统下。这样一来，对远程文件的操作和对本地文件的操作将没有什么区别，极大地方便了用户对远程文件的使用。

在嵌入式 Linux 中，编译的环境和运行的环境是不一样的，所以需要使用交叉编译工具。开发的一般过程是首先在 PC 上运行交叉编译工具编译好程序，然后将镜像文件刻录到目标开发板上的 Flash 闪存中去，而对于需要频繁调试的应用程序，如果每次都需要刻录，那将是一件相当烦琐的工作。所以在嵌入式 Linux 系统的开发调试阶段，可以利用 NFS 技术在主机上建立基于 NFS 的根文件系统，挂载到嵌入式设备，这样可以很方便地修改根文件系统的内容。具体使用方法将在后面的章节中具体介绍。

注意：无论是基于 Flash 的文件系统、基于 RAM 的文件系统还是网络文件系统，都是基于存储设备的文件系统（memory-based file system），它们都可用作 Linux 的根文件系统，此外 Linux 还支持逻辑的或伪文件系统（logical or pseudo file system），如 procfs（proc 文件系统），用于获取系统信息；以及 devfs（设备文件系统）和 sysfs，用于维护设备文件。

6.4 【应用实例】——在嵌入式系统上移植文件系统

制作文件系统的方法有很多，可以从零开始手工制作，也可以在现有的基础上添加部分内容并加载到目标板上去。由于完全手工制作工作量比较大，而且也很容易出错，因此，本节将主要介绍把现有的文件系统加载到目标板上的方法，主要包括制作文件系统映像和使用 NFS 加载文件系统的方法。

6.4.1 文件系统映像的制作

在 6.3 节中已经介绍过 Linux 支持多种文件系统，同样，嵌入式 Linux 也支持多种文件系统。虽然在嵌入式系统中，由于资源受限，它的文件系统和 PC Linux 的文件系统有较大

的区别，但是，它们的总体架构是一样的，都采用目录树的结构。在嵌入式系统中常见的文件系统有 cramfs、romfs、jffs、yaffs 等，这里就以制作 cramfs 文件系统为例进行讲解。cramfs 文件系统是一种经过压缩的、极为简单的只读文件系统，因此非常适合嵌入式系统。需要注意的是，不同的文件系统都有相应的制作工具，但是其主要的原理和制作方法是类似的。

1．制作 cramfs 文件系统

在嵌入式 Linux 中，busybox 是构造文件系统最常用的软件工具包，它被非常形象地称为嵌入式 Linux 系统中的“瑞士军刀”，因为它将许多常用的 Linux 命令和工具结合到了一个单独的可执行程序（busybox）中。虽然与相应的 GNU 工具比较起来，busybox 所提供的功能和参数略少，但在比较小的系统（如启动盘）或嵌入式系统中已经足够了。busybox 在设计上就充分考虑了硬件资源受限的特殊工作环境。它采用一种很巧妙的办法减小自己的体积：所有的命令都通过“插件”的方式集中到一个可执行文件中，在实际应用过程中通过不同的符号链接来确定到底要执行哪个操作。例如，最终生成的可执行文件为 busybox，当为它建立一个符号链接 ls 时，就可以通过执行这个新命令实现列出目录的功能。采用单一执行文件的方式最大限度地共享了程序代码，甚至连文件头、内存中的程序控制块等其他系统资源都共享了，对于资源比较紧张的系统来说，真是最合适不过了。在 busybox 的编译过程中，可以非常方便地加减它的“插件”，最后的符号链接也可以由编译系统自动生成。

busybox 的运行界面如图 6.9 所示。

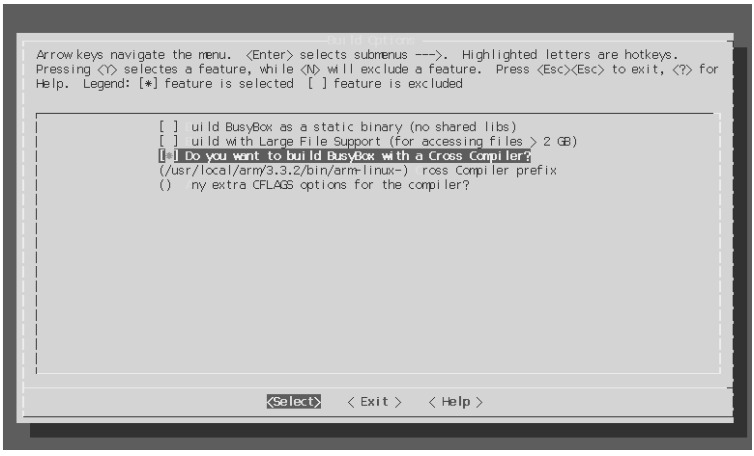


图 6.9　busybox 的运行界面

下面是 busybox 构建 GT2440 嵌入式系统的 cramfs 文件系统的详细操作步骤。

（1）从 busybox 网站下载 busybox 源码（本实例采用的 busybox-1.0.0）并解压，接下来，根据实际需要进行 busybox 的配置。

```
alloeat@ubuntu:/$ tar jxvf busybox-1.00.tar.bz2
alloeat@ubuntu:/$ cd busybox-1.00
alloeat@ubuntu:/$ make defconfig   /* 首先进行默认配置 */
alloeat@ubuntu:/$ make menuconfig
```

（2）设置平台相关的交叉编译选项，首先选中“Build Options”项的“Do you want to build Busybox with a Cross Complier？”选项，然后将“Cross Compiler prefix”设置为“/usr/local/arm/3.3.2/bin/arm-linux-”（这是在实验主机中的交叉编译器的安装路径，需要注意的是此时需要 3.3.2 版的 arm-linux-gcc 编译器）。

（3）编译并安装 busybox。

```
alloeat@ubuntu:/$ make
alloeat@ubuntu:/$ make install PREFIX=/home/david/GT2440/cramfs
```

其中，PREFIX 用于指定安装目录，如果不设置该选项，则默认在当前目录下创建_install 目录。创建的安装目录的内容如下所示：

```
[root@localhost cramfs]# ls
bin   linuxrc   sbin   usr
```

（4）此时使用 busybox 软件包所创建的文件系统还缺少很多东西，接下来可以通过创建系统所需要的目录和文件来完善文件系统的内容。

```
[root@localhost cramfs]# mkdir mnt root var tmp proc boot etc lib
[root@localhost cramfs]# mkdir /var/{lock,log,mail,run,spool}
```

（5）如果 busybox 是动态编译的（即在配置 busybox 时没选中静态编译），则把所需的交叉编译的动态链接库文件复制到 lib 目录中。

（6）接下来创建一些重要文件。首先要创建/etc/inittab 和/etc/fstab 文件。inittab 是 Linux 启动之后第一个被访问的脚本文件，而 fstab 文件定义了文件系统的各个“挂接点”，需要与实际系统相配合。接下来要创建用户和用户组文件。

此时已经用 busybox 构造了文件系统的内容，下面要创建 cramfs 文件系统映像文件。制作 cramfs 映像文件需要用到的工具是 mkcramfs。此时可以采用两种方法，一种方法是使用所构建的文件系统（在目录“/home/david/GT2440/cramfs”中），另一种方法是在已经做好的 cramfs 映像文件的基础上进行适当的改动。下面的示例使用第二种方法，因为这个方法包含了第一种方法的所有步骤（假设已经做好的映像文件名为“GT2440.cramfs”）。

（1）用 mount 命令将映像文件挂载到一个目录下，打开该目录并查看其内容。

```
alloeat@ubuntu:/$ mkdir cramfs
alloeat@ubuntu:/$ mount GT2440.cramgs cramfs –o loop
alloeat@ubuntu:/$ ls cramfs
bin   dev   etc   home   lib   linuxrc   proc   Qtopia   ramdisk   sbin   testshell   tmp   usr   var
```

因为 cramfs 文件系统是只读的，所以不能在这个挂载目录下直接进行修改，需要将文件系统中的内容复制到另一个目录中，具体操作如下所示。

```
alloeat@ubuntu:/$ mkdir backup_cramfs
alloeat@ubuntu:/$ tar xvf backup.cramfs.tar    cramfs/
alloeat@ubuntu:/$ mv backup.cramfs.tar backup_cramfs/
alloeat@ubuntu:/$ umount cramfs
```

```
alloeat@ubuntu:/$ cd backup_cramfs
[root@localhost backup_cramfs]# tar zvf backup.cramfs.tar
[root@localhost backup_cramfs]# rm backup.cramfs.tar
```

（2）此时可以像用 busybox 构建的文件系统一样，在 backup_cramfs 的 cramfs 子目录中任意进行修改。例如可以添加用户自己的程序：

```
alloeat@ubuntu:/$ cp～/hello backup_cramfs/cramfs/
```

在用户的修改工作结束之后，用下面的命令可以创建 cramfs 映像文件：

```
alloeat@ubuntu:/$    mkcramfs backup_cramfs/cramfs/ new.cramfs
```

此时通过 DNW 软件就可以将新创建的 new.cramfs 映像文件刻录到开发板的相应位置了。

2．制作 Yaffs2 文件系统

制作 Yaffs2 文件系统的镜像文件比较简单，可以通过 mkyaffsimage 软件完成，其具体操作步骤说明如下。

（1）下载并且解压 mkyaffsimage.tar.bz2 压缩包，此时可以得到 mkyaffsimage 软件，如图 6.10 所示。

图 6.10　mkyaffsimage 软件

（2）在“/opt/mkyaffsimage”目录下运行如下命令：

```
./mkyaffsimage /opt/rootfs/ /opt/myyaffs.img
```

此时即可自动生成 Yaffs2 文件系统的映像了，如图 6.11 所示。

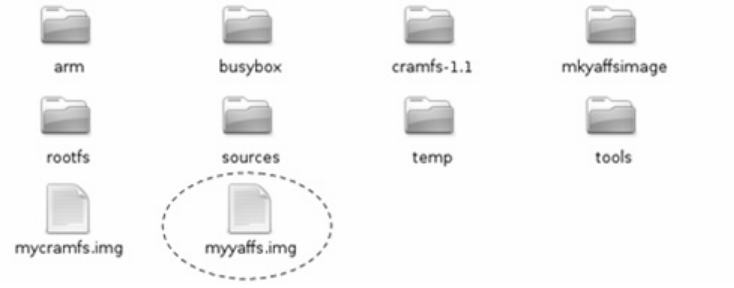


图 6.11　生成的 Yaffs2 文件系统映像

然后将该映像通过 DNW 下载到嵌入式开发板即可。

6.4.2　使用 NFS 文件系统

NFS 为 Network File System 的简称，最早是由 Sun 公司提出并发展起来的，其目的就

是让不同的机器、不同的操作系统之间通过网络可以彼此共享文件。NFS 可以让不同的主机通过网络将远端的 NFS 服务器共享出来的文件安装到自己的系统中，从客户端看来，使用 NFS 的远端文件就像使用本地文件一样。在嵌入式系统中使用 NFS 会使应用程序的开发变得十分方便，并且不用反复刻录映像文件。

NFS 的使用分为服务端和客户端，其中服务端提供要共享的文件，而客户端则通过挂载（“mount”）这一动作来实现对共享文件的访问操作。下面主要介绍 NFS 服务端的使用。在嵌入式开发中，通常 NFS 服务端在宿主机上运行，而客户端在目标板上运行。

NFS 服务端是通过读入它的配置文件“/etc/exports”来决定所共享的文件目录的。下面首先讲解这个配置文件的书写规范。

在这个配置文件中，每一行都代表一项要共享的文件目录及所指定的客户端对它的操作权限。客户端可以根据相应的权限，对该目录下的所有目录文件进行访问。配置文件中每一行的格式如下：

```
[共享的目录] [客户端主机名称或 IP] [参数 1，参数 2…]
```

在这里，主机名或 IP 是可供共享的客户端主机名或 IP，若对所有的 IP 都可以访问，则可用“*”表示。这里的参数有多种组合方式，其常见的参数如表 6.1 所示。

表 6.1　NFS 的常见参数

参　数	说　明
rw	可读/写的权限
ro	只读的权限
no_root_squash	NFS 客户端共享目录使用者的权限，即如果客户端使用的是 root 用户，那么对于这个共享的目录而言，该客户端就具有 root 的权限
sync	资料同步写入到内存与硬盘当中
async	资料会先暂存于内存当中，而非直接写入硬盘

如在本例中，配置文件“/etc/exports”的代码如下：

```
alloeat@ubuntu:/$ cat /etc/exports
/root/workplace       192.168.1.*(rw,no_root_squash)
```

在设定完配置文件之后，需要启动 nfs 服务和 portmap 服务，这里的 portmap 服务是允许 NFS 客户端查看 NFS 服务在用的端口，在它被激活之后，就会出现一个端口号为 111 的 sun RPC（远端过程调用）的服务。这是 NFS 服务中必须实现的一项，因此必须把它开启，如下所示：

```
alloeat@ubuntu:/$ service portmap start
启动 portmap:                                          [确定]
alloeat@ubuntu:/$ service nfs start
启动 NFS 服务:                                         [确定]
关掉 NFS 配额:                                         [确定]
启动 NFS 守护进程:                                     [确定]
启动 NFS mountd:                                       [确定]
```

可以看到，在启动 NFS 服务时启动了 mountd 进程。这是 NFS 挂载服务，用于处理 NFS 递交过来的客户端请求。还会激活至少两个以上的系统守护进程，然后就开始监听客户端的请求，用“cat/var/log/messages”命令可以查看操作是否成功。这样就启动了 NFS 的服务，另外还有下面两个命令，可以便于使用 NFS。

- exportfs：其可以重新扫描“/etc/exports”，使用户在修改了“/etc/exports”配置文件之后不需要每次重启 NFS 服务。其格式为：

```
exportfs [选项]
```

exportfs 常见选项说明如表 6.2 所示。

表 6.2　exportfs 常见选项说明

参　数	说　明
-a	全部挂载（或卸载）/etc/exports 中的设定文件目录
-r	重新挂载/etc/exports 中的设定文件目录
-u	卸载某一目录
-v	在 export 时，将共享的目录显示到屏幕上

- showmount：用于显示当前的挂载情况。其格式为：

```
showmount [选项] hostname
```

showmount 常见选项说明如表 6.3 所示。

表 6.3　showmount 常见选项说明

参　数	说　明
-a	在屏幕上显示目前主机与客户端所连上来的使用目录状态
-e	显示 hostname 中的/etc/exports 里设定的共享目录

第三部分

在 Linux 操作系统上进行软件开发

第7章

在嵌入式 Linux 操作系统中进行 C 语言开发

在嵌入式 Linux 操作系统中进行 C 语言开发，必须对 Linux 系统有足够的了解，包括代码的运行机制、内存的分配机制、系统调用和库函数等；此外还要求了解如何将 PC 上开发的 Linux 代码下载到嵌入式系统上，本章涉及的内容包括：

- Linux 操作系统如何执行一个程序；
- Linux 的程序存储空间和 main 函数；
- 一个 Hello World 的应用程序及其下载方法；
- 一些典型的 Linux 库函数的使用方法。

7.1 Linux 如何执行一个程序

Linux 操作系统中的程序是一个在存储空间上的一个可执行文件，内核调用一个 exec 函数将这个可执行文件调入存储器中然后执行，这个程序的执行实例被称为进程，在 Linux 中每个进程都对应一个唯一的非负数字标识符，称为进程 ID。

对于一个进程而言，有八种方式可以使其终止，这些方式说明如下：

- 从 main 函数中使用 return 语句返回；
- 调用 exit 函数终止进程；
- 调用_exit_或_Exit 函数终止进程；
- 最后一个线程从其启动例程返回；
- 最后一个线程调用了 pthread_exit 函数；
- 调用 abort 函数；
- 接到一个信号并且终止；
- 最后一个线程对取消请求作出了响应。

这些方式的前五种为正常终止一个进程，后三种是异常终止，图 7.1 是 Linux 操作系统启动和终止一个应用程序的示意。

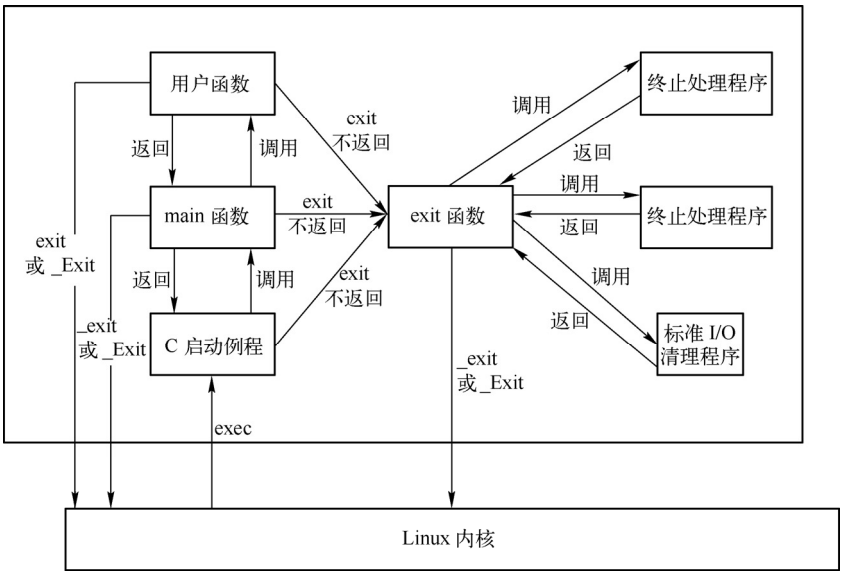


图 7.1　Linux 下程序启动和终止示意

总之，在 Linux 操作系统中，内核使程序执行的唯一方法是调用一个 exec 函数，进程自愿终止的唯一方法是显式或隐式地调用_exit 或_Exit，又或者使用一个外部信号来使得该进程终止。

通常来说，在 Linux 中运行一个用户自行设计的可执行文件的流程可以简单地表达如图 7.2 所示。

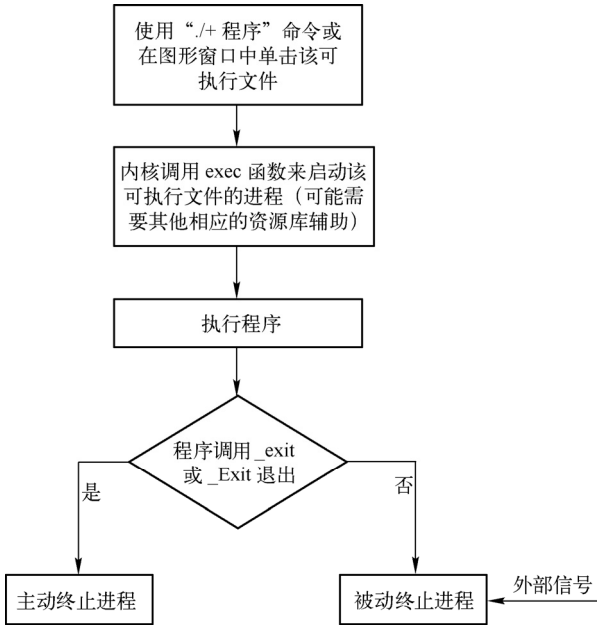


图 7.2　用户程序的运行过程

7.2　Linux 的程序存储空间

首先需要明确的是，本节所讨论的程序空间是指用户的 C 语言代码编译生成的可执行文件，而不是 C 语言源代码。

这些可执行文件的存储空间可以分为如下几部分。

- 正文段：存放了处理器执行的机器指令，通常来说正文段是可以共享的，所以包括 shell、gcc 在内的程序在存储器中只需有一个副本；正文段通常来说也是只读的，这是为了防止程序的可执行代码被意外修改。
- 初始化数据段：初始化数据段通常又被称为数据段，其包含了程序中需要进行初始化的变量值，例如如下的变量声明：

```
int counter = 0;
//counter 被初始化为 0，然后存放在初始化数据段中
//通常来说这些变量会是全局变量
//因为非全局变量会在调用的时候再分配空间并进行初始化
```

- 非初始化数据段：非初始化数据段是和初始化数据段对应的，用来存放不需要初始化（其实是被自动初始化为 0 或空指针）的变量，这个段又被称为 bss 段。
- 栈：这个段用于存放自动变量及每次函数调用时需要保存的信息。
- 堆：用于动态存储分配，这个段位于非初始化数据段和栈之间，在很多场合下这个段和栈一起被合称为堆栈段。

注意：对于一个可执行文件而言，其通常还有若干其他类型的段，如包含了符号表的段、包含了 gdb 调试信息的段和包含了动态共享库链接表的段等，但是这些段并不会在进程调用的时候被装入存储区中。

在 shell 命令行，可以使用 size 命令来查看一个可执行文件的正文段、数据段和 bss 段的长度信息，其单位是字节。

```
alloeat@ubuntu:~/chapter3Exam$ size exam18
   text    data     bss     dec     hex filename
   1582     264       8    1854     73e exam18
```

图 7.3 是 Linux 中典型的段分配方式，正文段通常从 0x0804800 地址单元开始，而栈底则位于 0xC0000000 之下，从高地址向低地址方向增长。

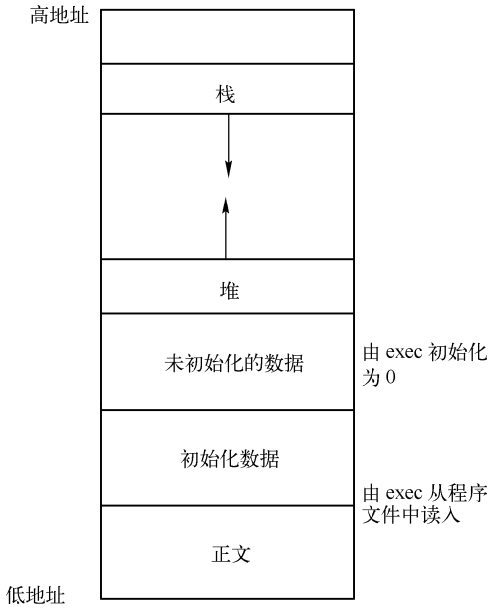


图 7.3　Linux 中典型的段分配方式

7.3　Linux C 的 main 函数

在 Linux 中，C 语言文件生成的可执行文件被 exec 调用，然后总是从 main 函数开始执行，第 1 章中所给出的 main 函数都是不带参数的，但是 Linux 下 main 函数的标准调用格式说明如下：

```
int    main(int argc, char *argv[])
```

在 main 函数的两个参数中，argc 必须是整型变量，其是命令行参数的数目；argv 必须是指向字符串的指针数组，这些指针分别指向各个命令行参数。

当 Linux 使用 exec 函数来启动一个 C 语言文件生成的可执行文件时，其在调用 main 函数之前首先调用一个特殊的启动例程并将此启动例程指定为程序的起始位置，这个启动例程将从内核取得该可执行文件的命令行参数和环境变量值，然后传递给 main 函数。

当用户要运行一个可执行文件时，在 Linux 命令行下输入文件名，再输入实际参数即可把这些实参传送到 main 函数中去。

Linux 的命令行的一般形式为：

```
可执行文件名  参数  参数……;
```

但是应该注意的是，main 的两个形参和命令行中的参数在位置上不是一一对应的。因为 main 的形参只有两个，而命令行中的参数个数原则上未加限制。argc 参数表示了命令行中参数的个数（注意：可执行文件名本身也算一个参数），argc 的值是在输入命令行时由系

统按实际参数的个数自动赋予的。例如，有命令行为：

```
gcc hello.c -o hello
```

由于文件名 gcc 本身也算一个参数，所以共有 4 个参数，因此 argc 取值为 4。argv 参数是字符串指针数组，其各元素值为命令行中各字符串（参数均按字符串处理）的首地址。指针数组的长度即为参数个数。数组元素初值由系统自动赋予。在上面的命令中，argv 数组的第 1 个元素指向的字符串为“gcc”，第 2 个元素指向的字符串为“hello.c”，第 3 个元素指向的字符串为“-o”，第 4 个元素指向的字符串为“hello”。

注意：main 函数的参数可以省略掉在应用过程中的参数输入读取步骤。例如，如果需要打开一个文件，可以直接在命令行中将文件名传递给应用代码，而不需要在应用代码中调用相应的输入代码等待用户输入。在实际使用中，如果不需要传递参数，也常常可以省略掉 main 函数的参数，直接写为 int main(void)。

此外，main 函数也带有返回值，默认的返回值类型为 int。在一般的程序中，main 函数的返回值类型 int 可以省略不写，返回值会直接传递给 Linux 内核。如果 main 函数的最后没有写 return 语句，gcc 会自动在生成的目标文件中加入：

```
return 0;
```

表示程序正常退出，main 函数的返回值可以将执行的结果反馈给内核，如使用一个多判断语句分别返回不同的 int 值。

例 7.1 是一个 main 函数的参数应用实例，分别打印传递给 main 函数的参数数目及参数内容。

【例 7.1】 main 函数的应用。

```
#include <stdio.h>
int main(int argc,char *argv[])     //第一个缓冲区存放参数的个数，第二个缓冲区存放参数
{
        unsigned int i=0;
        printf("%d\n",argc);
        for(i=0;i<argc;i++)
        {
                printf("%s\n",argv[i]);
        }
        return 0;
}
```

7.4 【应用实例】——Hello GT2440

本节介绍一个最简单的 Linux 应用实例，其用于在 GT2440 的嵌入式开发板的屏幕上显示一串字符串“Hello GT2440!”，例 7.2 是其源代码。

【例 7.2】 Hello GT2440 应用程序。

```
#include <stdio.h>
int main(int argc,char *argv[])
{
   printf("Hello GT2440!\n");
   exit(0);
}
```

使用 arm-linux-gcc 对以上代码进行编译，生成可以在 GT2440 开发板上执行的代码 Hello GT2440，然后通过 7.5 节介绍的方法即可将对应的代码下载到开发板上。

7.5 将程序下载到开发板

将编译好的代码下载到 GT2440 开发板上有如下四种方法，本节着重介绍方法一和方法三。

- 方法一：通过移动介质（如 U 盘）传递数据。
- 方法二：通过网络传递数据。
- 方法三：通过串口传递数据。
- 方法四：通过 NFS（网络文件系统）直接运行该代码。

7.5.1 【应用实例】——使用 U 盘传递数据

由于 GT2440 开发板提供了一个 USB HOST 接口（参考 2.4.3 节），所以可以将在 PC 上编译好的可执行程序复制到 U 盘上，将 U 盘连接到 GT2440 上并在 Linux 下挂载该 U 盘，然后即可将可执行程序复制到 GT2440 的嵌入式 Linux 下执行，其详细操作步骤说明如下。

（1）将 U 盘连接到 PC，执行如下命令或从图形界面中复制 Hello GT2440 可执行文件到 U 盘。

```
mount /dev/sda1/mnt          //挂接 U 盘
cp Hello GT2440 /mnt         //将 Hello GT2440 复制到 U 盘
umount /mnt                  //卸载 U 盘
```

（2）将 U 盘接到 GT2440 开发板的 USB HOST 接口，执行如下命令：

```
mount /dev/sda1/mnt          //挂接 U 盘
cp Hello GT2440 /mnt/sbin    //将 Hello GT2440 复制到 GT2440 开发板的可执行目录 sbin
umount /mnt                  //卸载 U 盘
```

7.5.2 【应用实例】——通过串口传递数据

在 4.4 节中介绍了使用串口和 GT2440 嵌入式开发板进行数据交互的方法，当通过串口终端登录 GT2440 系统之后即可使用“rz”命令通过串口从 PC 向 GT2440 开发板发送文件，其详细操作步骤说明如下。

（1）在串口终端中输入“rz”命令，开始接收从 PC 中发送的文件。

（2）在超级终端窗口中，单击鼠标右键，在弹出的菜单中选择“发送文件”命令，然后选择待发送的文件和使用的协议，即开始发送数据，如图 7.4 所示。

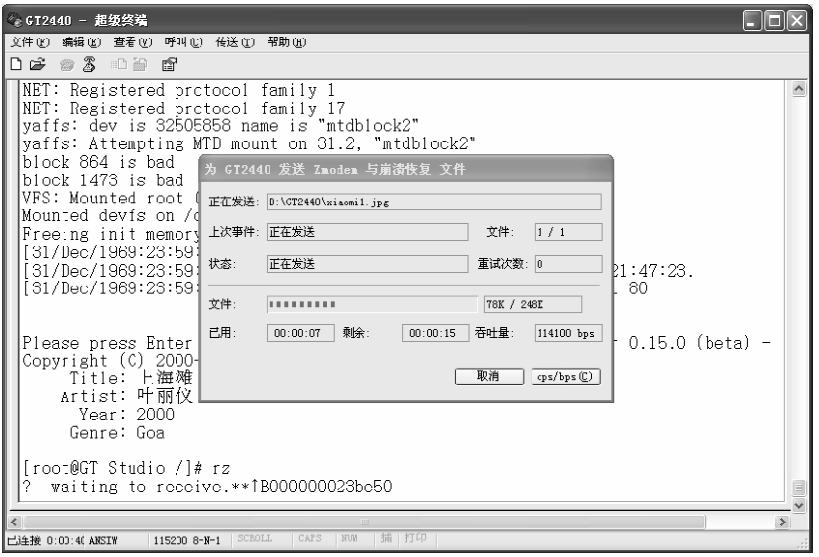


图 7.4　使用串口发送数据

（3）当 GT2440 开发板接收数据完成之后，会在当前的目录看到 Hello GT2440 文件，此时还需要使用 chmod + x 命令将文件的属性改为可执行后才能正常运行。

7.6　Linux 操作系统典型库函数介绍及其使用

Linux 操作系统提供了大量的库函数和系统调用以供用户完成相应的工作，本节将对这些系统调用和库函数进行简要的介绍。

7.6.1　Linux 的系统调用和库函数基础

Linux 内核提供了一些内建的函数，可以用来完成一些系统级别的功能，这样的函数叫作“系统调用”，英文是 systemcall，这些函数代表从用户空间到内核空间的一种转换，相关的声明可以在 syscall.h 头文件中找到。

Linux 提供的这些系统调用都对应一个具体的数字，Linux 内核通过 0x80 中断来管理这些系统调用，而这些系统调用的对应的数字和相应的参数都在被调用的时候送到对应寄存器里。

注意：系统调用的数字实际上是一个序列号，表示其在系统的一个数组 sys_call_table[] 中的位置。

在具体的使用中，Linux 为这些系统调用在标准 C 函数库中设置了一个具有相同名字的

函数，用户可以通过相应的调用方法来对这些函数进行调用，然后该函数使用系统所需要的技术调用相应的内核服务，所以从应用角度来说，可以将这些系统调用看作 C 语言函数。

另外，Linux 还提供了一些通用库函数以供用户调用，虽然这些函数可以调用一个或多个内核的系统调用，但是它们并不是内核的入口点，如 atoi 函数等。

从操作系统的角度来看，系统调用和库函数的实现方法有重大的区别，但是从用户（Linux 下 C 程序员）的角度来看它们是一样的，在很多实际应用中应用程序会调用系统调用或库函数，而库函数又会调用系统调用，如图 7.5 所示。

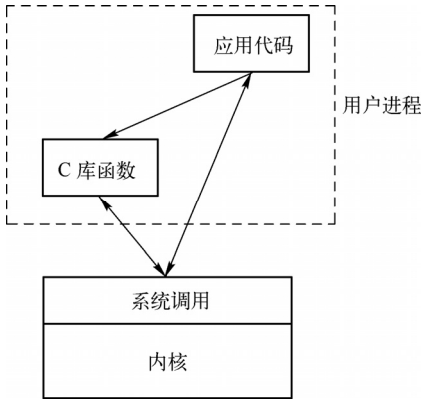


图 7.5　Linux 下库函数和系统调用的关系

注意：系统调用通常只提供一种最小的接口，而库函数通常会提供比较复杂的功能；在必要的时候用户可以自行替换或修改库函数，但是不能替换或修改系统调用。

7.6.2 【应用实例】——求平方根

平方根函数可以对一个数字求其平方根，其标准调用格式说明如下，分别针对不同类型的参数，如果操作成功则返回对应数字的平方根值；如果失败则返回对应的错误编码。

```
#include <math.h>
double sqrt(double x);
float sqrtf(float x);
long double sqrtl(long double x);
```

例 7.3 是平方根函数的应用实例，其从键盘输入 n 个实型数据，然后分别求其平方根并返回。

【例 7.3】 求平方根。

```
//键盘输入整数 n，然后输出 n 个实型数，求 n 个实型数的平方根
#include <stdio.h>
#include <math.h>
int main(void)
{
        int n,i;
```

```
        float x,y;
        scanf("%d",&n);                    //等待输入
        for(i=0;i<n;i++)                   //循环
        {
                scanf("%f",&x);            //输入 n 个数据
                y = sqrtf(x);              //求平方根
                printf("%f *****%f\n",x,y); //打印输出
        }
    return 0;
}
```

注意：使用 math.h 头文件定义相应数学库函数，在使用 gcc 编译时，需要使用-lm 关键字来定位数学库的位置。

7.6.3 【应用实例】——产生随机数

实际应用中常常需要获得一个随机数来作为输入，此时可以使用 rand 随机数函数，其标准调用格式如下：

```
#include<stdlib.h>
int     rand (void);
```

调用成功之后返回一个 0～RAND_MAX 之间的整型数据，其中 RAND_MAX 在 stdlib.h 头文件中有定义，默认是 2147483647。

需要注意的是，rand 函数的内部实现是用线性同余法做的，它并不是真的随机数，只不过是因为其周期特别长，所以在一定的范围内可看成是随机的。

在调用此函数产生随机数前，必须先调用 srand 函数来初始化随机数种子，如果未设随机数种子，rand 函数在调用时会自动设随机数种子为 1。另外需要注意的是，rand 函数产生的是假随机数字，每次执行时是相同的；若要不同，则需要调用 srand 函数不同的随机数种子来初始化该函数。

srand 函数的标准调用格式如下，其中 seed 为初始化参数，srand 函数没有返回值。

```
#include <stdlib.h>
void srand(unsigned int seed);
```

注意：在 Linux 系统中，通常使用 getpid()或 time(0)的返回值作为 srand 函数的参数。

例 7.4 是 rand 函数和 srand 函数应用实例，其调用使用 time0 作为 srand 参数对 rand 函数进行了初始化操作，然后调用 rand 函数产生了 10 个随机数并输出。

【例 7.4】 产生随机数。

```
//产生 10 个介于 1 到 10 之间的随机数
#include <stdlib.h>
#include <stdio.h>
int main(void)
```

```
{
        int i,j;
        srand((int)time(0));     //调用 srand 初始化种子
        for(i=0;i<10;i++)
        {
                j = 1+(int)(10.0 * rand()/RAND_MAX + 1.0);   //产生随机数
                printf("%d\n",j);
        }
}
```

7.6.4 【应用实例】——获得系统时间和日期

在 Linux 系统应用中，经常需要获得当前的时间信息，Linux 内核提供了一些相应函数用于操作，其标准调用格式说明如下：

```
#include <time.h>
char *asctime(const struct tm *tm);
char *asctime_r(const struct tm *tm, char *buf);
char *ctime(const time_t *timep);
char *ctime_r(const time_t *timep, char *buf);
struct tm *gmtime(const time_t *timep);
struct tm *gmtime_r(const time_t *timep, struct tm *result);
struct tm *localtime(const time_t *timep);
struct tm *localtime_r(const time_t *timep, struct tm *result);
time_t mktime(struct tm *tm);
int gettimeofday(struct timeval *tv, struct timezone *tz);
int settimeofday(const struct timeval *tv, const struct timezone *tz);
```

各个函数的说明如下。

- asctime 函数：将参数 timeptr 所指的 tm 结构中的信息转换成真实世界所使用的时间日期表示方法，然后将结果以字符串形态返回。此函数已经由时区转换成当地时间，字符串格式为:“Wed Jun 30 21:49:08 1993/n”。该函数的参数值是 tm 指针指向的存储空间，返回值是表示目前当地的时间日期的字符串。

注意：若再调用相关的时间日期函数，此字符串可能会被破坏。此函数与 ctime 的不同之处在于传入的参数是不同结构的。

asctime 函数中涉及的 tm 时间信息结构体说明如下。

```
struct tm
{
   int tm_sec;      //秒
   int tm_min;      //分钟
   int tm_hour;     //小时
   int tm_mday;     //日期
   int tm_mon;      //月份
```

```
    int tm_year;      //年份
    int tm_wday;      //星期
    int tm_yday;      //从 1 月 1 日开始到当日日期编号
    int tm_isdst;
  //夏令时标识符，实行夏令时的时候，tm_isdst 为正。不实行夏令时的时候，
  //tm_isdst 为 0；不了解情况时，tm_isdst 为负
};
```

- asctime_r 函数：是 asctime 函数的一个扩展，提供了一个缓冲器件 buf 用于存放返回值，该缓冲区的长度不能小于 26 字节。
- ctime 函数：将参数 timep 所指的 time_t 结构中的信息转换成真实世界所使用的时间日期表示方法，然后将结果以字符串形态返回。此函数已经由时区转换成当地时间，字符串格式为“Wed Jun 30 21 :49 :08 1993/n”。若再调用相关的时间日期函数，此字符串可能会被破坏。
- char *ctime_r 函数：和 ctime 函数功能相同，也提供了一个缓冲区用于存放返回值。
- gmtime 函数：将所指的 time_t 结构中的信息转换为真实世界所使用的时间日期表示方法，然后将结果返回到 tm 结构体中。
- gmtime_r 函数：和 gmtime 函数类似，同时提供了一个由 result 指针指向的内存空间用于存放返回值。
- localtime 函数：返回当地目前时间和日期，其将参数 timep 所指的结构体中的信息转换为真实世界所使用的时间日期表示方法，然后返回。
- localtime_r 函数：和 localtime 函数类似，同时提供了一个由 result 指针指向的内存空间用于存放返回值。
- mktime 函数：将参数 tm 所指向的结构体数据转换为从 1970 年 1 月 1 日 0 时 0 分 0 秒开始所经历的秒数，然后返回。
- gettimeofday：获取当前时间和时区信息，这个需要超级用户的权限，tv 参数用于指向存放返回的时间信息的缓冲区，其结构说明如下。

```
struct timeval {
    time_t      tv_sec;      // 秒
    suseconds_t tv_usec;     // 微秒
};
```

而 tz 用于存放相应的时钟信息，说明如下。

```
struct timezone {
    int tz_minuteswest;      //minutes west of Greenwich
    int tz_dsttime;          //type of DST correction
};
```

- settimeofday 设置当前时间和时区信息，其参数和使用方法可以参考 gettimeofday。

例 7.5 是系统时间函数的应用实例，其调用 gmtime 函数获得当前时间的秒数据，然后使用 asctime 函数将该秒数据转换为正常的显示格式显示。

【例 7.5】 获得系统时间和日期。

```
//打印系统的当前时钟
#include <time.h>
#include <stdio.h>
int main(void)
{
        time_t timetemp;                    //定义一个时间结构体变量
        char *wday[] = {"Sun","Mon","Tue","Wed","Thu","Fri","Sat"};
        struct tm *p;                       //结构体指针
        time(&timetemp);                    //获得时间参数
        printf("%s",asctime(gmtime(&timetemp)));
        p = localtime(&timetemp);
        printf("%d:%d:%d:\n",(1900+p->tm_year),(1+p->tm_mon),p->tm_mday);
        printf("%s    %d:%d:%d\n",wday[p->tm_wday],p->tm_hour,p->tm_min,p->tm_sec);
        return 0;
}
```

7.6.5 【应用实例】——打印单字符

只需在标准输出设备（通常是屏幕上）输出一个字符时，可以使用 putchar 函数，其标准调用格式说明如下：

```
#include <stdio.h>
int putchar(int c);
```

参数 c 是标准的待输出字符，如果输出成功则函数返回输出成功的字符，即参数 c；若输出失败则返回 EOF。

注意：putchar 函数其实质上是 putc(c,stdout)，相关的知识将在第 8 章中进行详细介绍。

例 7.6 是使用 putchar 函数输出几个字符的应用实例。

【例 7.6】 打印单字符。

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
   putchar(toupper('a'));
   putchar('\n');
   putchar(toupper('1'));
   putchar('\n');
   putchar(toupper('A'));
   putchar('\n');
   putchar(toupper(0x34));
   putchar('\n');
   putchar(toupper(0x61));
```

```
    putchar('\n');
    return 0;
}
```

7.6.6 【应用实例】——将字符串转换为数字

某些应用的输入是一些字符串，这些字符串代表了一些需要参与运算的数字，但是在 C 代码中其不能直接参与运算，必须使用相应的字符串转换函数，这些转换函数的标准调用格式说明如下：

```
#include <stdlib.h>
int atoi(const char *nptr);
long atol(const char *nptr);
long long atoll(const char *nptr);
long long atoq(const char *nptr);
```

指针 nptr 指向的字符串为待转换的字符串数字，函数的返回值为转换后得到的数据，函数会在遇到不可识别的字符时结束，并且转换时会自动忽略空白字符。

例 7.7 是字符串转换函数的应用实例。

【例 7.7】 将字符串转换为数字。

```
#include <stdio.h>
#include <stdlib.h>
#define TRUE    1
#define FALSE 0
int main(void)
{
    unsigned char temps1[]="23";
    unsigned char temps2[]="321K";
    unsigned char temps3[]="-32";
    int temp = 0;
    printf("%s\n",temps1);
    temp = atoi(&temps1[0]);                //将 temps1 字符串的值转换为整数，存放在 temp 中
    printf("%d\n",temp);                    //输出 temp 中存放的整数
    printf("%d\n",(temp*2));                //将 temp 中存放的整数*2 后输出
    printf("%s\n",temps2);
    temp = atoi(&temps2[0]);                //将 temps2 中的数据转化为整型数据
    printf("%d\n",temp);
    printf("%s\n",temps3);
    temp = atoi(&temps3[0]);
    printf("%d\n",temp);
    return 0;
}
```

7.6.7 【应用实例】——字符串复制

如果需要将一个字符串复制到一个指定区域，则可以使用 memccpy 函数，其标准调用

格式说明如下：

```
#include <string.h>
void *memccpy(void *dest, const void *src, int c, size_t n);
```

参数 dest 是指向目的缓存区的指针，src 是指向源缓冲区的指针；c 是遇到之后停止复制的字符，n 是最大的复制个数，函数将最多 n 个字符的数据从源缓冲区复制到目的缓冲区，如果遇到字符 c 则停止。

例 7.8 是字符串复制函数的应用实例。

【例 7.8】 字符串复制。

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char *temps1="hello world!";
   char temps2[20];
   char *temp;
   temp=memccpy(&temps2[0],temps1,'o',20);
//将 temps1 中的字符复制到 temps2 中，直到找到了字符 o 或到了 20 字节
   if(temp)                                          //判断是否找到了字符 o
   {
      *temp='\0';                                    //如果找到了字符 o
      printf("Char found: %s.\n",&temps2[0]);        //输出找到了字符
   }
   else
   {
      printf("Char not found.\n");                   //否则输出没有找到字符
   }
   printf("%s\n",&temps2[0]);                        //输出 temps2 的值
   temp=memccpy(&temps2[0],temps1,'\0',20);
//将 temps1 中的字符复制到 temps2 中，直到停止符或到了 20 字节
   if(temp)
   {                                                 //判断 temp 的值
      *temp='\0';
      printf("Char found: %s.\n",&temps2[0]);        //输出找到了字符
   }
   else
   {
      printf("Char not found.\n");                   //否则输出没有找到字符
   }
   printf("%s\n",&temps2[0]);                        //输出 temps2 的值
   return 0;
}
```

7.6.8 【应用实例】——添加通讯录条目

Linux 操作系统提供了三个用于存储空间动态分配的函数和一个用于释放内存空间的函数，这四个函数详细说明如下。

（1）malloc 函数：给进程分配指定字节数的存储区，此存储中的初始值不为 0。

（2）calloc 函数：为指定数量具有指定长度的对象分配存储空间，该空间中每一位都被初始化为 0。

（3）realloc 函数：更改以前分配区的长度（可以增加，也可以减少），当为增加长度时，可能需要将以前分配区间的内容迁移到另外一个足够大的区域，以便在尾部提供增加的存储区，而新增加的区间内的初始化值不确定。

（4）free 函数：用于释放其参数指针指向的存储空间，这些空间会被送入系统的可用存储区池，可以被以上三个函数再次分配。

这四个函数的标准调用格式说明如下，三个分配函数如果调用成功则返回一个指向分配区的非空指针，否则返回空指针，而 free 函数没有返回值。

```
#include   <stdlib.h>
void *malloc(size_t size);
void *calloc(size_t nobj,size_t size);
void *realloc(void *ptr,size_t newsize);
void free(void *ptr);
```

内存分配函数所返回的指针一定是适当对齐的，从而使得这些存储空间可以应用于任何数据对象，并且由于其返回值均为通用指针 void*，当用户使用它们的时候，通常是不需要进行类型转换的。

这三个内存分配函数中，realloc 函数使得用户可以增加或减少以前分配的内存空间的长度。例如，可以使其减少使用固定长度的数组从而节省了所必需的内存空间，但是需要注意的是其最后一个参数 newsize 是新分配的存储区长度而不是分配后存储区的总长度，如果 ptr 指向一个空指针，则 realloc 函数的功能和 malloc 是完全相同的。

另外，这三个函数通常都是通过调用 sbrk 系统调用来实现的，该系统调用的标准格式说明如下：

```
#include <unistd.h>
int brk(void *addr);
void *sbrk(intptr_t increment);
```

注意：在内存空间使用完成之后必须立即释放，否则可能导致内存泄漏，这是 Linux 系统开发中最常见的问题之一。如果以前分配的一片内存不再需要使用或无法访问，却并没有释放它，那么对于该进程来说，会因此导致总可用内存的减少，这时就出现了内存泄漏。

例 7.9 是内存分配函数 malloc 的应用实例，其中还调用了 getpagesize 函数来获取内存分页的大小，其标准调用格式说明如下：

```
#include <unistd.h>
```

```
int getpagesize(void);
```

getpagesize 函数没有调用参数，其返回值是内存分页的大小，需要注意的是这是 Linux 系统的分页大小，不一定和硬件分页大小相同。

应用实例模拟了一个手机的通讯录存储空间的增加情况，该通讯录的结构体定义为 struct co，其中各个分量说明如下。

- index：编号。
- name：姓名。
- MTel：手机号码。
- Tel：座机号码。

【例 7.9】 添加通讯录条目。

```
//在内存中添加一个单元
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
struct co
{
   int index;
   char name[8];
   char MTel[12];
   char Tel[12];
};
int x;

int main(void)
{
   struct co *p;
   char ch;
   printf("do you add a user? Y/N\n");
   ch =getchar();
   if(ch == 'y'||ch == 'Y')
   {
            p = (struct co *)malloc(sizeof(struct co));
            p->index = ++x;
            printf("User name:");
            scanf("%s",p->name);
            printf("Mobile:");
            scanf("%s",p->MTel);
            printf("Home Tel:");
            printf("intex:%d\n name:%s\n MoveTel:%s\n HomeTel:%s\n",p->index,p->name,p->MTel,p->Tel);
   }
   printf("page size=%d\n",getpagesize());
}
```

7.6.9 【应用实例】——内存映射

可以使用 mmap 函数，其将指定的文件映射到内存区域中，对该内存区域进行操作即可以实现对该文件的操作，mmap 函数的标准调用格式说明如下：

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags,int fd, off_t offset);
```

mmap 函数的各个参数说明如下，如果函数调用成功则返回文件映射区间的实地址，否则返回 MAP_FAILED（-1）。

- addr：指定映射存储区的起始位置，通常将其设置为 0，此时系统将选择映射区的起始位置。
- length：映射字节数。
- prot：说明映射区的保护方式，如表 7.1 所示。

表 7.1　port 映射区的保护方式

PORT_READ	映射区间可读
PORT_WRITE	映射区间可写
PORT_EXEC	映射区间可执行
PORT_NONE	映射区间不可访问

- flags：影响映射区的相应属性，其说明如表 7.2 所示。

表 7.2　映射区的属性

MAP_FIXED	addr 所指向的地址无法使用时则放弃，通常不推荐
MAP_SHARED	映射区域的写入数据复制回文件，并且允许其他映射该文件的进程共享
MAP_PRIVATE	映射区域的写入操作会产生一个映射文件
MAP_ANONYMOUS	建立匿名映射时忽略参数 fd，映射区间无法和其他进程共享
MAP_DENYWRITE	只允许对映射区域的写入操作，其他对文件的直接写入操作会被拒绝
MAP_LOCKED	映射区域被锁定

- fd：被映射的文件描述符，通常来说可以用 open（打开文件）等函数返回。
- offset：映射字节在文件中的偏移量。

例 7.10 是 mmap 函数的应用实例，其使用 mmap 来映射/etc/passwd 文件到内存区域，然后进行相应的操作。

【例 7.10】 内存映射。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
```

```
int main(void)
{
   int fd;
   void *start;
   struct stat sb;
   fd = open("/etc/passwd",O_RDONLY);
   //fd = open("/etc/passwd",O_RDONLY);
   fstat(fd,&sb);
   start = mmap(NULL,sb.st_size,PROT_READ,MAP_PRIVATE,fd,0);
   if(start==MAP_FAILED) //判断是否成功
   {
          return 0;
   }
   printf("%s",start);
   munmap(start,sb.st_size);        //解除映射
   close(fd);
}
```

7.6.10 【应用实例】——标准输入/输出

在实际应用中，Linux 的 C 语言代码需要和用户进行通信，此时可以使用 printf 函数和 scanf 函数，它们被称为标准输入输出函数。

printf 函数用于将格式化数据输出，其标准调用格式如下：

```
#include <stdio.h>
int printf(const char *format, ...);
```

其参数 format 是一个字符串，包含字符、字符序列和格式说明。其中字符部分与字符序列按顺序输出；而格式说明以%开始，格式说明使跟随的相同序号的数据按格式说明转换和输出。如果数据的数量多于格式说明，多余的数据将被忽略，如果格式说明多于数据，结果将是随机的。如果输出成功，函数的返回值为输出的字符数目，如果输出失败，则返回一个负数。

printf 函数的格式说明结构为：%_flags_width_.precision_{b|B|l|L}_type，各个部分的说明如下。

- type 用来说明参数是字符、字符串、数字还是指针字符，如表 7.3 所示。

表 7.3　printf 函数的 type 参数

type	输 出 结 果
D	有符号十进制数
U	无符号十进制数
O	无符号八进制数
x	无符号十六进制数，使用小写
X	无符号十六进制数，使用大写

续表

type	输 出 结 果
f	格式为[-]ddd.ddd 的浮点数
e	格式为[-]d.ddde+dd 的浮点数
E	格式为[-]d.dddE+dd 的浮点数
g	使用 f 或 e 中比较合适形式的浮点数
G	去 f 或 E 中比较合适形式的双精度值
c	单字符常数
s	字符串常数
p	指针，格式 t:aaaa，其中 aaaa 为十六进制的地址 t：存储类型；c：代码；i：片内 RAM；x：片外 RAM；p：片外 RAM
n	无输出，但是在下一参数所指整数中写入字符串
%	%字符

- b、B、l、L 用于 type 之前，说明整型 d、i、u、o、x、X 的 char 或 long 转换。
- flgs 是标记，其用法如表 7.4 所示。

表 7.4　printf 函数的 flgs 参数

flags	作　　用
-	左对齐
+	有符号，数值总是以正负号开始
空格	数字总是以符号或空格开始
#	变换形式：o、x、X，首字母为 O、Ox、OX G、g、e、E、f 则输出小数点
*	忽略

- width 是域宽，只能是一个非负数，用来表示输出字符的最小个数，如果打印字符较少则使用空格填充，在前面加负号则表示在域中使用左对齐，加 0 则表示用 0 填充。如果输出的字符个数大于域的宽度，仍然会输出全部字符。“*”表示后续整数参数提供域的宽度。
- precision 精度，对于不同类型意义不同，可能引起截尾或者舍入，如表 7.5 所示。

表 7.5　printf 函数的 precision 精度

数 据 类 型	说　　明
d、u、o、x、X	输出数字的最小位，如果输出数字超出也不截断尾部，如果超出在左边则填入 0
f、e、E	输出数字的小数位数，末位四舍五入
g、G	输出数字的有效位数
c、p	无影响
s	输出字符的最大字符数，超过部分将不显示

和 printf 函数相对，标准输入函数 scanf 用于用户向程序输入数据，其标准调用格式如下：

```
#include <stdio.h>
int scanf(const char *format, ...);
```

其参数结构和 printf 完全相同，如果函数调用成功则返回指定的输入项数；若输入出错或在任意变换前已至文件尾端则为 EOF。

例 7.11 是 printf 和 scanf 函数的应用实例，应用代码提示用户输入 a、b 两个整数，然后输出相乘的结果，最后要求输出一个字符后输出。

【例 7.11】 标准输入输出。

```
#include <stdio.h>
int main(void)                          //没有参数
{
   int a,b,sum;
   char str[30];                        //字符串存放
   printf("please input a,b!\n");
   scanf("%d%d",&a,&b);                 //输入两个整数
   sum = a * b;                         //计算乘积
   printf("the sum is %d\n",sum);       //输出计算结果
   printf("please input the string\n");
   scanf("%s",str);
   printf("the string is %s\n",str);    //打印刚刚输入的字符串
   return 0;
}
```

第8章

在嵌入式 Linux 中进行文件和流操作

Linux 中的文件是指以计算机的存储设备为载体的信息集合；对文件的操作是在嵌入式 Linux 中进行程序开发的最基础操作；而流操作的实质也是对文件的操作，相对于文件操作而言其不需要通过缓冲，拥有更高的效率。本章将详细介绍如何在嵌入式 Linux 中进行文件和流的操作，涉及的内容包括：

- 嵌入式 Linux 的文件系统涉及的文件类型；
- 嵌入式 Linux 的基础文件操作方法；
- 嵌入式 Linux 下的目录文件操作方法；
- 嵌入式 Linux 的流及其基础操作方法；
- 嵌入式 Linux 的流格式化输入和输出。

8.1 Linux 的文件操作基础

在第 6 章中介绍了如何在嵌入式系统中移植文件系统，从系统角度来看，无论是什么具体的文件系统，都是对文件存储器空间进行组织和分配，负责文件存储并对存入的文件进行保护和检索的系统；其负责为用户建立文件，存入、读出、修改、转储文件，控制文件的存取，当用户不再使用时撤销文件等。

8.1.1 Linux 的文件系统介绍

Linux 采用了目录树的格式来管理所有的文件和目录，其树形目录中只有唯一的一个根目录“/”（称为根，root），其他目录都是这个根目录衍生的子目录，图 8.1 是 Linux 的目录树文件结构示意。

注意：在 Linux 中，所有的内容都被看成文件，包括硬件和目录；所有的操作都可以归结为对文件的操作，Linux 可以像操作普通文件一样来对磁盘文件、串口、键盘、显示器、打印机及其他设备进行操作。

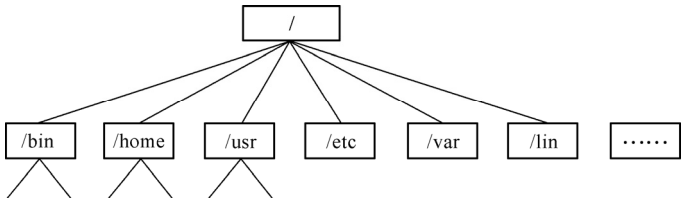


图 8.1　Linux 的目录树文件结构示意

Linux 的文件系统是目录和文件的一种层次安排，目录的起点称为根（root），其名字是一个字符“/”；目录（directory）是一个包含目录项的文件，在逻辑上，可以认为每个目录项都包含一个文件名，同时还包含说明该文件属性的信息。通过这种树形等级结构，用户可以浏览整个系统，可以进入任何一个已授权进入的目录并访问相应的文件。

注意：和 Windows 用户自己建立相应的文件夹来对所有的文件进行分门别类的管理不同，Linux 提供了相应的文件夹来主动对文件进行分类管理。

Linux 的文件是个简单的字节数据序列，所以在 Linux 下对文本文件和二进制文件的结构和访问方法是一样的。Linux 的文件是由一系列块（block）组成的，每个块可能含有 512、1024、2048 或 4096 字节，具体由系统实现决定，在同一个文件系统中块大小是相同的。当使用较大块时，由于每次磁盘操作可以传输更多的数据，操作所花的时间较少，所以可以提高磁盘和内存间数据传输速率；但是相对地，由于块太大，存储的有效容量也将会下降，也就是说会浪费一些存储空间。

Linux 的文件系统由如下四部分组成：引导块、超级块、索引节点表（inode table）和数据块，各个部分的详细说明如下。

- 引导块：用于存放文件系统的引导程序，引导程序用于系统引导或启动操作系统。如果一个文件系统不存放操作系统，其引导块将为空。
- 超级块：用来描述该文件系统管理的资源，其包含空闲索引节点表和空闲数据块表，用于具体说明文件系统的资源使用情况。
- 索引节点表：用来存储文件的控制信息，每个节点对应一个文件。
- 数据块：是磁盘上存放数据的磁盘块，包括目录文件和数据。

图 8.2 是 Linux 文件系统组织结构示意图。

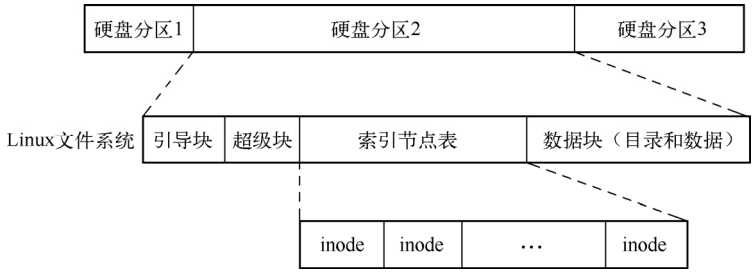


图 8.2　Linux 文件系统组织结构示意图

这四个部分中最重要的是超级块和索引节点表，它们都用于描述当前文件系统状态的组成。

超级块用于描述 Linux 文件系统的资源状态，包括文件系统的大小、空闲单元位置信息等，在文件系统对文件的管理中起着至关重要的作用，其由如下字段构成：

- 文件系统的容量性信息，如 inode 数目、数据块数目、保留块数目和块的大小等；
- 文件系统中空闲块的数目；
- 文件系统中部分可用的空闲块表；
- 空闲块表中下一个空闲块号；
- 索引节点表的大小；
- 文件系统中空闲索引节点表数目；
- 文件系统中部分空闲索引节点表；
- 空闲索引节点表中下一个空闲索引节点号；
- 超级块的锁字段，用于保证对存储单元的互斥操作；
- 空闲块表的锁字段和空闲索引节点的锁字段；
- 超级块是否被修改的标志；
- 其他字段，存放了文件系统是否完整的标志。

注意：Linux 在关机时要求先将缓冲区数据写回文件系统，并且卸载（umount）该文件系统，如果没有卸载文件系统就关机，则很可能导致数据丢失；而在 Linux 启动时在挂载（mount）一个文件系统之前首先会去检查其超级块中的相应字段，如果上次没有进行卸载操作，则需要对该文件系统完整性作检查（fsck）。

超级块给出的是文件系统的相关信息，而一个文件信息则是由索引节点表（inode）给出的，每个文件都有自己的索引节点表，在其中包含了该文件数据在磁盘上存储的位置信息、操作权限、文件所有者、操作时间等信息。

索引节点表平时存储在磁盘上，在需要进行操作的时候读入内存，通常来说把存储在磁盘上的索引节点表称作磁盘索引节点，而把其在内存中的映像称作内存索引节点表。

索引节点表由如下字段构成。

- 文件类型：Linux 的文件可以分为普通文件、目录文件、链接文件、设备文件、管道文件等，将在 8.1.2 节进行详细介绍。
- 文件链接数：记录了引用该文件的目录表项数，即记录了有多少个文件名指向该文件。
- 文件属主标识：指出该文件的所有者 id。
- 文件属主的组标识：指出该文件所有者属组的 id。
- 文件的访问权限：系统将用户分为文件属主、同组用户和其他用户三类，每类用户可能获得对文件的一种或几种访问权限。要特别指出的是，目录文件的执行权限是指修改目录的权力。
- 文件的存取时间：包括文件最后一次被修改的时间、最后一次被访问的时间和最后一次修改索引节点的时间。

- 文件的长度：以字节表示的文件长度。
- 文件的数据块指针：文件操作的当前位置指针。

在索引节点表中并不包含文件的名称，文件名的信息存放在目录文件中，其具体存放方式将在 8.1.2 节介绍。

在 Linux 中的 stat.h 头文件中使用了一个结构体来定义索引节点表的相应字段，其说明如下：

```
#ifndef _ALPHA_STAT_H
#define _ALPHA_STAT_H
//32 位的索引节点表的字段结构体定义
struct stat {
  unsigned int      st_dev;       //文件所在位置的设备号
  unsigned int      st_ino;       //文件的索引节点号
  unsigned int      st_mode;      //文件的类型
  unsigned int      st_nlink;     //连接到该文件的其他文件数量
  unsigned int      st_uid;       //文件所属用户
  unsigned int      st_gid;       //文件所属用户所在组
  unsigned int      st_rdev;      //如果是设备文件，则保存设备号，否则无效
  long              st_size;      //文件长度，如果是设备文件则为 0
  unsigned long     st_atime;     //最近一次访问文件时间
  unsigned long     st_mtime;     //最近的修改文件时间
  unsigned long     st_ctime;     //最近一次对文件状态进行修改的时间
  unsigned int      st_blksize;   //文件系统的块大小
  unsigned int      st_blocks;    //文件所分配的块数
  unsigned int      st_flags;     //文件的用户定义标志
  unsigned int      st_gen;       //文件产生编号
};
//以下是 64 位系统的一些关于索引节点表的定义，增加了一些项
//修改了一些项，可以参考上一个结构体
struct stat64 {
  unsigned long     st_dev;
  unsigned long     st_ino;
  unsigned long     st_rdev;
  long              st_size;
  unsigned long     st_blocks;
  unsigned int      st_mode;
  unsigned int      st_uid;
  unsigned int      st_gid;
  unsigned int      st_blksize;
  unsigned int      st_nlink;
  unsigned int      __pad0;
  unsigned long     st_atime;
  unsigned long     st_atime_nsec;
  unsigned long     st_mtime;
  unsigned long     st_mtime_nsec;
```

```
    unsigned long      st_ctime;
    unsigned long     st_ctime_nsec;
          long         __unused[3];
};
#endif
```

文件描述符（file descriptor）是 Linux 用于标识一个特定进程正在访问的文件用的，当打开一个文件或创建一个文件的，系统将返回一个文件描述符供其他操作引用，其可以用来标识其对应的特定文件，通常来说其是一个小的非负整数。

在 Linux 中，每个进程都可以拥有最多 1024 个文件描述符，并且有自己的文件描述符表，其中前三项对于一般的进程是固定的且是由系统自动打开的，说明如下。

- 文件描述符 0：标准输入文件，对于一般进程来说是键盘。
- 文件描述符 1：标准输出文件，一般输出到显示器。
- 文件描述符 2：标准错误输出文件，一般输出到屏幕。

这三个描述符用户程序都是不用执行文件打开操作就可直接使用的，其在头文件中的定义部分如下：

```
#define    STDIN_FILENO      0      //标准输入
#define    STDOUT_FILENO     1      //标准输出
#define    STDERR_FILENO     2      //标准错误输出
```

8.1.2　Linux 的文件类型

文件系统对文件的管理不仅是结构上的，还对文件属性进行说明和管理，文件的属性包括：文件类型、文件长度、文件所有者、文件的许可权、文件最后的修改时间等。用户可以设置目录和文件的权限，以便允许或拒绝其他人对其进行访问。使用 ls –l 命令可以看到当前路径下的文件属性说明：

```
alloy@alloy-VirtualBox:/$ ls -l
总用量 92
drwxr-xr-x   2 root root   4096 2012-06-13 11:59 bin
drwxr-xr-x   3 root root   4096 2012-06-13 12:02 boot
drwxr-xr-x   2 root root   4096 2012-06-12 22:29 cdrom
drwxr-xr-x  15 root root   4060 2012-06-13 16:21 dev
drwxr-xr-x 139 root root 12288 2012-06-13 16:21 etc
……（以下省略）
```

从中可以看到文件类型、文件属性、用户名、用户所在组、文件大小、修改时间、文件名等信息，其中类似“drwxr-xr-x”的项说明了文件的类型和属性，其包含了 10 位字符，可以分为 4 组，如图 8.3 所示，详细说明如下。

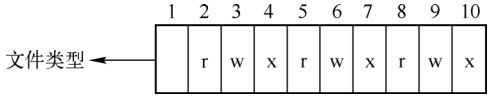


图 8.3　文件类型和属性

- 第 1 组：第 1 位，表示文件的类型，包括了普通文件、目录文件、管道文件等。
- 第 2 组：2～4 位，表示文件所有者（User）的权限，分别为读、写、执行。
- 第 3 组：5～7 位，表示文件所有者的同组用户（Group）的权限，分别为读、写、执行。
- 第 4 组：8～10 位，表示其他组用户（Other）的权限，同样分别为读、写、执行。

第一位是文件的类型说明，其由 stat 结构体中 st_mode 来决定，标志符和对应的文件及在 stat 中定义的关键字如表 8.1 所示。

表 8.1　文件类型说明

-	普通文件，对应 S_ISEEG()
l	链接文件，对应 S_ISLNK()
c	字符设备文件，对应 S_ISCHR()
s	套接字文件，对应 S_ISSOCK()
d	目录文件，对应 S_SDIR()
b	块设备文件，对应 S_ISBLK()
p	管道文件，对应 S_ISFIFO()

1．普通文件

普通文件也称正规文件，是 Linux 系统中最常见的一类文件，其特点是不包含有文件系统的结构信息，包括图形文件、数据文件、文档文件、声音文件等；普通文件按其内部结构又可为文本文件和二进制文件两种。

- 文本文件：是字符（ASCII 码）组成的文件，以行为基本结构的信息存储文件，其是 Linux 系统中最多的一种文件类型，它的内容是用户可以直接读到的数据，如数字、字母等。通常来说，Linux 的系统配置文件基本都属于这种文件类型，可以使用 cat 命令直接查看。
- 二进制文件：按信息在内存中的格式表示的文件，通常不能直接查看，而必须使用相应的软件来查看。通常来说，Linux 中的可执行文件（脚本、文本方式的批处理文件除外）基本都属于这种文件类型，可以运行。

2．目录文件

在前面介绍过，Linux 中的目录也是以文件形式存在的，称为目录文件，其是文件系统中一个目录所包含的目录项组成的文件；用户可以读取但是不能修改该目录文件的内容，其只允许系统进行修改。

目录文件在文件名与索引节点之间的转换起到桥梁作用，是文件系统树形文件结构的关键，其由文件名和索引节点号构成。

Linux 的文件系统对文件的管理是通过索引节点来进行的，目录文件只不过提供了文件名和索引节点之间的转换手段。为了保证文件系统层次的完整性，目录文件是由系统来管理的，用户只能读目录文件，而不允许直接写目录文件。每个目录文件的前两项是两个特设的文件“.”和“..”。其中“.”对应于该目录文件本身的索引节点，而“..”则对应于其父目录的索引节点。如果一个目录中只包含“.”和“..”文件，则该目录为空目录。当用户访问某

个文件时，系统需要找到它所对应的索引节点，而目录文件建立了文件名和索引节点号之间的路线。

3．链接文件

链接文件又称符号链接文件，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。链接文件提供了共享文件的一种方法。在链接文件中，不是通过文件名实现文件共享的，而是通过链接文件所包含的指向文件的指针来实现对文件的访问。普通用户可以建立链接文件，并通过其指针访问它所指向的那个文件。使用链接文件可以访问普通文件，还可以访问目录文件和不具有普通文件实态的其他文件。也就是说，链接文件可以在不同的文件系统之间建立一种链接关系。根据链接对象的不同，链接文件又可以分为硬链接文件和符号链接文件。

4．管道文件

管道文件主要用于在进程间传递数据，其是 Linux 进程间的一种通信机制。管道是进程间传递数据的“媒介”，一个进程将数据写入管道的一端，另一个进程从管道的另一端读取数据。通常管道是建立在高速缓存中的。采用先进先出的规定处理其中的数据，管道文件又可以分为有名管道和无名管道两种。

5．设备文件

在 Linux 中，其将设备也看作文件，和文件具有相同的操作方法，这种文件被称为设备文件，其是为操作系统与 I/O 设备提供连接的一种文件，分为字符设备文件和块设备文件，分别对应于字符设备和块设备，这些文件通常存放在 dev 目录中。

注意：Linux 中存在一个目录 /dev/null。所有放入这一设备的数据都将不存在，可以把这个放入操作看成删除。

- 字符设备（character device）：顺序的数据流设备，对这种设备的读/写是按字符进行的，而且这些字符连续地形成一个数据流。字符设备不具备缓冲区，所以对这种设备的读写是实时的，如串口终端、磁带机等。
- 块设备（block device）：具有一定结构的随机存取设备，对这种设备的读写是按块进行的，它使用缓冲区来存放暂时的数据，待条件成熟后，从缓存一次性写入设备或从设备中一次性读出放入到缓冲区，如磁盘和文件系统等。

6．套接字文件

套接口（Socket）文件主要用于在不同计算机的进程间的通信，也称为套接字。

套接口是操作系统内核中的一个数据结构，它是网络中的节点进行通信的门户。套接口有三种类型：流式套接口、数据报套接口和原始套接口。流式套接口也就是 TCP 套接口（或称面向连接的套接口）；数据报套接口也就是 UDP 套接口（或称无连接的套接口），原始套接口用“SOCK_RAW”表示。

流式套接口定义了一种可靠的面向连接的服务，实现了无差错、无重复的顺序数据传输。数据报套接口定义了一种无连接的服务，数据通过相互独立的报文进行传输，是无序的，并且不保证可靠、无差错。原始套接口允许对低层协议（如 IP 或 ICMP）直接访问，主要用于新的网络协议实现的测试等。

8.2 Linux 的基础文件操作

Linux 通过相应的文件 I/O 函数来完成对文件的操作，这些函数通常被称为“不带缓冲的 I/O”，这是因为它们对文件的读/写都是调用 Linux 内核的系统调用来实现的。Linux 的基础文件操作函数包括：open（打开文件）、read（读文件）、write（写文件）、iseek（设置文件指针）和 close（关闭文件）。

8.2.1 使用 open 函数打开文件

open 函数用于在 Linux 中打开一个文件，如果该文件不存在，则先创建该文件，然后打开，如果操作成功则返回文件对应的文件描述符，如果操作失败则返回“−1”。

open 函数的标准调用格式说明如下：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open (const char *pathname, int flags ) ;                    //打开一个现有的文件
int open (const char *pathname, int flags, mode_t mode ) ;
//如果打开的文件不存在，则先创建它
```

open 函数的各个参数和应用实例说明如下。

1．open 函数的 pathname 参数

pathname 是一个指针变量，用于传递包含了路径的完整文件名称，其典型的应用实例是“/dev/log”。

2．open 函数的 flags 参数

flags 是一个 int 类型的变量，用于指定文件的打开方式，常用的有如下三种标志可选。

- 只读：关键字 O_RDONLY，通常定义为 0。
- 只写：关键字 O_WRONLY，通常定义为 1。
- 读写：关键字 O_RDWR，通常定义为 2。

在对一个文件进行相应的操作时，还必须注意文件本身的权限，对一个文件进行超权限的操作将会返回一个错误，如对只读文件进行写操作。

需要注意的是 flags 参数中以上三个参数是必需且唯一的，也就是说这些关键字之间不能用“OR”来连接，只能选择其中一个，此外 flags 还可以使用可选的参数，如表 8.2 所示。

表 8.2 flags 的其他选项

选　项	说　明
O_CREAT	当文件不存在时，将建立该文件，此时会用到 open 的第三个参数
O_EXCL	如果同时指定了 O_CREAT，而文件已经存在，则会出错。用此方法可以测试一个文件是否存在，如果不存在，则创建此文件
O_NOCTTY	当文件名（可以包含路径，即第一个参数 pathname）指向一个终端设备时，它将不再是进程控制的终端，即使该进程没有一个终端设备

续表

选　项	说　明
O_TRUNC	如果文件存在，则该文件将被截断，即长度截断为 0。注意，文件没有以写方式打开也可以截断；截断后文件的属主和属性不变
O_APPEND	文件以追加方式打开 i，每次进行写操作时，文件指针都会被放置到文件末尾
O_NONBLOCK/O_NDELAY	当文件以非阻塞方式打开后，对于 open 及随后的对该文件的操作，都会及时返回，而无须进程等待。这对于普通文件和目录文件没有作用，但对于管道等进程间通信的操作很有用
O_SYNC	文件以同步 I/O 方式打开。任何写操作都会使得进程被阻塞，直到物理写动作完成为止

表 8.2 给出的标志都可以混合使用，各标志之间用“|”符号连接。其实第二个参数为 int 型参数，该数的每一位都对应一个操作，符号“|”将它们按位或，即加起来，使得需要操作的位被置“1”。

注意：上面介绍的标志中有一些可以在文件打开后用 fcntl 函数进行修改，参考后续章节。

3．open 函数的 mode 参数

如果仅需要打开一个文件，可以不使用 open 函数的第三个参数，但充分考虑到文件可能不存在，在打开之前就需要创建，此时则需要使用 mode 参数。

mode 参数说明如表 8.3 所示。

表 8.3　mode 参数说明

符　号	值	含　义
S_IRWXU	00700	文件属主有读、写、执行权限
S_IRUSR(S_IREAD)	00400	文件属主有读权限
S_IWUSR(S_IWRITE)	00200	文件属主有写权限
S_XWUSR(S_IEXEC)	00100	文件属主有执行权限
S_IRWXG	00070	文件组成员有读、写、执行权限
S_IRGRP	00040	文件组成员有读权限
S_IXGRP	00010	文件组成员有执行权限
S_IRWXO	00007	其他用户有读、写、执行权限
S_IROTH	00004	其他用户有读权限
S_IWOTH	00002	其他用户有写权限
S_IXOTH	00001	其他用户有执行权限

mode 参数支持“或”运算，也就是说，可以同时使用如表 8.3 中的一个或几个参数，其间可以使用“|”关键字来直接连接或对其对应的值进行计算之后获得最后的数值并直接调用。

4．【应用实例】——使用 open 函数打开文件

例 8.1 调用了 open 函数在当前的工作目录下以读写打开方式打开一个名为“examtest”的文件，如果该文件不存在，则创建该文件，创建该文件时使用 S_IRWXU 关键字来给予该

文件的读写操作权限。open 函数将 examtest 的文件描述符返回给一个 int 类型的变量 temp，然后使用 printf 函数将该描述符输出，并且使用 close 函数来关闭文件（close 文件的说明将在 8.2.2 节给出）。

【例 8.1】 使用 open 函数打开文件。

```
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
int main(void)
{
   int temp;
   temp = open("./examtest",O_RDWR|O_CREAT,S_IRWXU);   //创建文件
   printf("%d\n",temp);      //输出文件描述符
   temp = close(temp);       //关闭文件
   printf("%d\n",temp);      //输出关闭文件的返回值
   exit(0);
}
```

8.2.2 使用 close 函数关闭文件

close 函数用于关闭一个已经打开的文件，如果关闭成功，则返回 0；否则返回-1。

close 函数的标准调用格式说明如下：

```
#include <unistd.h>
int close ( int fd );
```

需要注意的是，当对文件进行打开和关闭操作时，还会对其相关信息产生相应的影响。

- 当打开一个文件时，该文件描述中的引用计数器值加 1。而关闭一个文件时，该文件描述中的引用计数器值减 1。当引用计数器的值减为 0 时，系统调用 close 不仅将释放该文件的描述符，也将释放该文件所占的描述表项。
- 关闭一个文件时也释放该进程加在该文件上的所有记录锁。当一个进程终止时，它所有的打开文件都由内核自动关闭。很多程序都使用这一功能而不显式地用 close 关闭所打开的文件。
- 当关闭的不是一个普通文件时，可能会产生一些其他影响。例如，关闭管道文件的一端时，将影响到管道的另一端。

close 函数的参数为文件描述符，通常来说这个符号为其他函数的返回值，如 open 函数等。

8.2.3 使用 create 函数创建文件

create 函数用于在 Linux 中创建一个文件，如果创建成功则返回该文件对应的文件描述

符，如果出错则返回-1。

create 函数的标准调用格式说明如下：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int creat(const char *pathname, mode_t mode);
```

create 函数的各个参数和应用实例说明如下。

1．create 函数的 pathname 参数

create 函数的 pathname 参数使用方法和 open 函数的 pathname 参数应用方法完全相同，可以参考第 8.2.1 节。

2．create 函数的 mode 参数

create 函数的 mode 参数使用方法和 open 函数中 mode 参数也完全相同，可以参考第 8.2.1 节。

注意：create 函数其实等同于 int open(const char *pathname, O_WRONLY|O_CREAT | O_TRUNC, mode_t mode)。

3.【应用实例】——使用 create 函数创建文件

例 8.2 使用 main 函数的参数集合的第二个参数来作为即将创建的文件的 pathname 参数，然后在当前目录下调用 create 函数来建立一个属性为 S_IRWXU 的文件。

【例 8.2】 使用 create 函数创建文件。

```
//建立一个文件，文件名由 argv 的第二个参数给出
#include <fcntl.h>
 #include <stdio.h>
 int main(int argc,char *argv[])
 {
    int temp;
    if (argc!=2) //如果参数不是两个，可执行文件+待创建的文件
    {
       printf("run error\n");                    //执行错误
       return 1;                                 //退出
    }
    temp = creat(*(argv+1),S_IRWXU);             //参数字符串的第二个数据作为文件名
    printf("%d\n",temp);
    return 0;
 }
```

8.2.4　使用 write 函数写文件

write 函数用于向一个已经打开的文件写入数据，如果操作成功则返回已经写入的数据字节数，如果操作失败则返回“-1”。

write 函数的标准调用格式说明如下：

```
#include <unistd.h>
ssize_t write (int fd, void *buf, size_t count );
```

write 函数的返回值通常与参数 count 的值相同，否则表示出错。write 出错的最常见原因是磁盘已满，或者超过了文件长度限制。

注意：对于普通文件而言，写操作从文件的当前位移量处开始。如果在打开该文件时指定了 O_APPEND 选择项，则在每次写操作之前，将文件位移量设置在文件的当前结尾处。在一次成功写之后，该文件位移量增加实际写的字节数。

write 函数的各个参数和应用实例说明如下。

1．write 函数的 fd 参数

fd 参数是待写入文件的文件描述符，其通常通过 open、create 等函数获得。

2．write 函数的 buf 参数

buf 是一个指向写入缓冲区的指针，待写入数据必须存放在该缓冲区内。

3．write 函数的 count 参数

count 表示本次操作将要写入文件的数据的字节数。

4．【应用实例】——使用 write 函数对文件进行写操作

例 8.3 是 write 函数的应用实例，应用代码首先打开参数字符串指定的文件，如果没有则创建这个文件，然后对该文件写入一个字符串“this is a test!”，该字符串存放在缓冲区 wbuf 中。

【例 8.3】 使用 write 函数对文件进行写操作。

```
#include <fcntl.h>
 #include <stdio.h>
 int main(int argc,char *argv[])
 {
    int fileID,temp;
    char wbuf[15] = "this is a test!";
    fileID = open(*(argv+1),O_RDWR|O_CREAT,S_IRWXU);
 //打开文件，如果没有则创建
    printf("%d\n",fileID);                    //打印文件描述符
    temp = write(fileID,wbuf,15);             //使用文件描述符调用文件
    printf("%d\n",temp);
    close(fileID);
    return 0;
 }
```

8.2.5 使用 lseek 函数对文件进行内部定位

在第 8.2.4 节中提到了文件偏移量的概念，在 Linux 中，每个打开的文件都有一个与其相关联的当前文件偏移量（也叫文件指针），它通常是一个非负整数，用以度量从文件开始处计算的字节数。通常，读、写操作都从当前文件偏移量处开始，并使偏移量增加所读写的

字节数。

lseek 函数来用指定文件偏移量的位置，从而实现文件的随机存取，如果操作成功则返回新的文件偏移量，如果出错则返回-1。

lseek 函数的标准调用格式说明如下：

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fds, off_t offset, int whence);
```

lseek 函数的各个参数和应用实例说明如下。

1．lseek 函数的 fds 参数

fds 参数是待写入文件的文件描述符，其通常通过 open、create 等函数获得。

2．lseek 函数的 offset 参数

offset 是文件偏移量，指的是每一次对文件的读写操作所需移动的距离，单位为字节；offset 的取值可正可负，正值指的是向前移，负值指的是向后移。

注意：对于普通文件而言，offset 通常都是正值，所以在使用时最好能先测试其值以确保取值。

3．lseek 函数的 whence 参数

whence 有如下三种不同的取值。

- SEEK_SEK：设置偏移量为文件开始位置之后的 offset 个字节。
- SEEK_CUR：设置偏移量为当前偏移量之后的 offset 个字节。
- SEEK_END：设置偏移量为当前文件长度加上 offset 个字节。

lseek 函数允许文件的偏移量被设置到超过文件结束符（EOF）处；然后在下一次调用 write 时，可以将文件的长度延伸到所需的长度，并用无意义的字符填充这个空隙。如果随后的 read 读取这个空隙间的数据，将得到无意义的值，直到这个文件数据块被真正写回到磁盘上，再读取这个空隙间的数据将得到 0。这是因为，当在文件尾之后执行 write 函数时，Linux 系统并不存储无用的数据块。在 read 函数读到该数据块时，系统为 read 函数产生一个全为 0 的数据块，返回给 read 函数。如果一个 read 函数置于文件尾或文件尾之后的文件偏移量，则产生 0 作为 read 的返回值。

另外，由于 lseek 成功执行时返回新的文件位移量，为此可以用下列方式确定一个打开文件的当前位移量：

```
off_t currpos;                          /*定义变量 currpos 的数据类型为 off_t */
currpos = lseek (fd, 0, SEEK_CUR);      /*offset 值为 0*/
```

注意：可用来确定所涉及的文件是否可以设置位移量。如果文件描述符引用的是一个管道或 FIFO，则 lseek 返回-1，并将 errno 设置为 EPIPE。

4.【应用实例】——使用 lseek 函数对文件进行连续写入操作

例 8.4 是 lseek 函数的应用实例，应用代码首先打开参数字符串指定的文件，如果没有

则打开这个文件，然后对该文件连续写入字符串“this is a test!”并且按回车键换行，该字符串存放在缓冲区 wbuf 中，在每次写入之前都需要将文件偏移量移动到下一次待写入的位置。

【例 8.4】 使用 Lseek 函数对文件进行连续写入操作。

```
#include <fcntl.h>
 #include <stdio.h>
 int main(int argc,char *argv[])
 {
    int temp,seektemp,i,j;
    int fileID;
    char wbuf[17] = "this is a test!\r\n";
    if(argc!= 2)
    {
        printf("run error!\n");
        return 1;                                   //如果参数不正确则退出
    }
    fileID = open(*(argv+1),O_RDWR|O_CREAT,S_IRWXU);
    temp = write(fileID,wbuf,sizeof(wbuf));         //写入数据
    seektemp = lseek(fileID,0,SEEK_CUR);            //获得当前的偏移量
    for(i=0;i<10;i++)
    {
      j = sizeof(wbuf) * (i+1);                     //计算下一次的偏移量
      seektemp = lseek(fileID,j,SEEK_SET);
      temp = write(fileID,wbuf,sizeof(wbuf));       //写入数据
    }
    close(fileID);
    return 0;
 }
```

8.2.6 使用 read 函数读文件

read 函数可以从一个已打开的 Linux 文件中读取指定长度的数据，如果操作成功，则返回读到的字节数；如果已经到达了文件的末端则返回 0；如果出错则返回-1。

read 函数的标准调用格式说明如下：

```
#include <unistd.h>
ssize_t read (int fd, void *buf, size_t count) ;
```

通常来说，read 函数的读操作是从文件的当前位移量处开始的，在成功返回之前，该位移量增加实际读得的字节数，但是有如下的几种情况可使实际读到的字节数少于要求读的字节数。

- 当读普通文件时，在读到要求字节数之前已到达文件尾端。例如，若在到达文件尾端之前还有 30 字节，而要求读 100 字节，则 read 返回 30，下一次再调用 read 时，它将返回 0（文件尾端）。

- 当从终端设备读时，通常一次最多读一行。
- 当从网络中读取时，网络中的缓冲机构可能造成返回值小于所要求读的字节数。
- 某些面向记录的设备，如磁带，一次最多返回一个记录。

read 函数的各个参数和应用实例说明如下。

1．read 函数的 fd 参数

fd 参数是待读出文件的文件描述符，其通常通过 open、create 等函数获得。

2．read 函数的 buf 参数

存放读出数据的缓冲区的指针。

3．read 函数的 count 参数

count 是待读取的数据长度，如果 count 为 0，则 read 函数返回 0 并且没有其他结果。如果 count 大于 32767，则结果不能确定。

注意：在 32 位系统中，count 是一个 32 位的变量，而在 64 位系统中这是一个 64 位的变量。

4.【应用实例】——使用 read 函数实现文件复制

例 8.5 是 read 函数的应用实例，应用代码首先打开参数字符串 1 指定的文件作为源文件，然后打开参数字符串 2 指定的文件作为目的文件，然后调用 read 函数从源文件中把数据读出，写入到目的文件中。

【例 8.5】 使用 read 函数实现文件复制。

```
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#define PERMS 0666
#define      DUMMY 0
#define MAXSIZE 1024
int main(int argc, char *argv[])
{
   int sourcefileID, targetfileID;     //目标文件和源文件的描述符
   int readNO = 0;                     //读出的字符数
   char WRBuf[MAXSIZE];                //定义缓冲区
   if(argc!=3)                         //如果命令行参数不争取正确
   {
      printf("run error\n");
      return 1;
   }
   if((sourcefileID=open(*(argv+1),O_RDONLY,DUMMY))==-1)     //如果源文件打开失败
   {
      printf("Source file open error!\n");
      return 2;
   }
   if((targetfileID=open(*(argv+2), O_WRONLY|O_CREAT, PERMS))==-1)   //如果目标文件打开失败
```

```
    {
        printf("Target file open error!\n");
        return 3;
    }
    while((readNO=read(sourcefileID, WRBuf, MAXSIZE))>0)   //如果读出来的数据大于 0
    if(write(targetfileID, WRBuf,readNO)!=readNO)     //如果写入的数据和读出的数据不同
    {
        printf("Target file write    error!\n");   //写数据错误
        return 4;
    }
    close(sourcefileID);
    close(targetfileID);   //关闭文件
    return 0;
}
```

8.3 文件的高级操作

Linux 的基础文件操作函数是围绕文件的普通 I/O 操作来进行的，包括打开、关闭、创建和读写文件，本节将讨论 Linux 文件系统的其他特征和文件的性质，以及这些属性的相关操作函数。

8.3.1 使用 stat 函数操作文件状态

Linux 使用了一个结构体 stat 来存放文件的相应属性，而 stat、fstat 和 lstat 函数用于获得文件的这个属性结构体，如果成功返回 0，否则返回−1。

stat、fstat 和 lstat 函数的标准调用格式说明如下：

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *pathname, struct stat *sbuf);
int fstat(int fd, struct stat *sbuf);
int lstat(const char *pathname,, struct stat *sbuf);
```

stat 函数和 lstat 函数使用文件的路径作为参数来标识需要获取属性的文件，而 fstat 函数使用文件对应的描述符来标识需要获取属性的文件。lstat 函数和 stat 函数的区别在于：如果目标文件是一个符号链接，lstat 返回的是该符号链接的有关信息，而 stat 返回的是符号链接所引用的文件信息。

注意：可以简单地把符号链接和符号链接对应文件关系理解为 Windows 中的快捷方式和快捷方式对应的文件。

stat 系列函数的参数说明如下。

- pathname：目标文件的路径，可以是绝对路径或相对路径，在 stat 和 lstat 函数中使用。

- fd：目标文件的文件描述符，通常由其他函数返回，在 fstat 函数中使用。
- sbuf：指向存放目标文件状态结构体的目标指针。

8.3.2　使用 utime 函数操作文件时间

stat 系列函数返回的是文件的属性状态，如果要对文件相应的时间信息进行操作，可以使用 utime 函数，u 调用成功将返回 0 并自动更新文件的特性修改时间 st_ctime；否则返回−1。

在 Linux 系统中，每个文件都有三个对应的时间信息，如表 8.4 所示，其分别对应 stat 结构体中的如下三个字段。

```
unsigned long    st_atime;   //最近一次访问文件时间
unsigned long    st_mtime;   //最近的修改文件时间
unsigned long    st_ctime;   //最近一次对文件状态进行修改的时间
```

表 8.4　文件的时间信息

文 件 字 段	说　　明	操 作 函 数
st_atime	文件的最后访问时间	read
st_mtime	文件的最后修改时间	write
st_ctime	文件索引节点（innode）的最后修改时间	chmod

st_atime 和 st_ctime 这两个时间的主要区别：前者是最后一次对文件本身进行修改操作的时间，而后者是对文件的索引节点 innode 进行操作的时间；前者受到相应的函数如 write 的影响，后者的改变则不一定涉及对文件内容的操作，只要修改了文件的状态如文件的访问权限等就会产生。

注意：可以利用 utime 函数来改变一个文件的访问时间和修改时间，但是没有函数可以改变文件的特性修改时间，因为 inode 是由系统来维护的。

utime 函数的标准调用格式说明如下：

```
#include <sys/types.h>
#include <utime.h>
int utime(const char *pathname,const struct utimebuf *times );
```

utime 函数的各个参数和应用实例说明如下。

- pathname：目标文件的路径参数。
- times：times 用于存放 utime 返回的时间信息，其是 utime 函数使用的一个数据结构，其说明如下。

```
struct utimbuf
{
    time_t actime;
    time_t modtime;
}
```

- actime：文件的访问时间。
- modtime：文件的修改时间。

需要注意的是，这两个时间值都是日历时间，也就是自标准时间（1970 年 1 月 1 日 00:00:00 起到当前所经过的秒数）。

如果 times 是空指针，文件的访问时间和修改时间均设置为当前时间。此时，要么进程的有效用户 ID 必须等于文件的用户 ID，要么进程必须有该文件的写权限。

如果 times 不是空指针，它解释为指向 utimebuf 结构的指针并用 times 值更新文件的访问时间和修改时间。在这种情形下，要么进程的有效用户 ID 必须等于文件的用户 ID，要么必须是超级进程。仅具有文件的写权限是不够的。

8.3.3 使用 dup 和 dup2 函数操作文件的描述符

每一个打开的文件所对应的文件描述符不是唯一的，同一个文件可能对应着多个文件描述符，而 dup 函数和 dup2 函数都可以用来复制文件描述符。

dup 和 dup2 函数的标准调用格式说明如下，如果操作成功则返回新的文件描述符，否则返回−1。

```
#include <unistd.h>
int dup (int fd);
int dup2 (int fd, int fd2);
```

由 dup 返回的新文件描述符一定是当前可用文件描述符中的最小数值。用 dup2 时则可以用 fd2 参数指定新描述符的数值。如果 fd2 已经打开，则先将其关闭。如果 fd 等于 fd2，则 dup2 返回 fd2，而不关闭它。通常使用这两个系统调用来重定向一个已打开的文件描述符。

fcntl 函数提供了进一步管理低级文件描述符的各种手段，用它可以对已打开的文件描述符执行各种控制操作，包括修改打开文件的性质、复制文件描述符、操作文件锁等。fcntl 函数的标准调用格式说明如下，如果调用成功其返回值由 cmd 参数来决定，而如果调用失败则返回−1。

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd, int arg)
```

fcntl 函数的 fd 参数是打开的需要进行操作文件的文件描述符，而 cmd 参数决定了 fcntl 的功能和返回值，其说明如表 8.5 所示。

表 8.5 fcntl 函数中 cmd 的参数说明

选　项	说　明
F_DUPD	返回大于或等于 arg 的最低序号的文件描述符。该功能可以由 dup 函数实现。新的文件描述符与旧的可以互换使用。调用成功，返回值为新的文件描述符
F_GETFD	获得 close-on-exec 标志。如果最后一位是 0，则该标志没有设置。返回值为 0 或 1

续表

选　　项	说　　明
F_SETFD	设置 close-on-exec 标志为指定的值 arg（只有最后一位有效，为 0 或 1）
F_GETFL	获得文件的打开方式。返回所有的标志位，标志位的含义与 open 一节相同
F_SETFL	设置文件打开方式的标志。设置文件打开方式为参数 arg 指定的方式。仅能设置 O_APPEND 和 O_NONBLOCK（或 O_NDELAY），有的系统还可以设置 O_SYNC。该标志被文件描述符所有的复本（dup 函数或 fcntl 的 F_DUPFD 产生）所共享
F_GETLK	获得本进程得到锁的第一个锁的 flock 结构
F_SETLK	获得离散的文件锁，不等待
F_SETLKW	获得离散的文件锁，必要时等待
F_GETOWN	返回当前接收 SIGIO 或 SIGURG 信号（signal）的进程 ID 或进程组。进程 ID 以负值返回
F_SETOWN	设置进程或进程组接收 SIGIO 和 SIGURG 信号，进程组 ID 以负值返回。进程 ID 用正值指定

fcntl 函数的第三个参数 arg 可能是一个整数，也可能是一个如下的结构体，其和 cmd 参数相关。

```
struct flock{
        long   l_start;              /* 块开始处的偏移量 starting offset */
        long        l_len;           /* 块长 */
        long        l_pid;           /* 所的属主（进程）*/
        long        l_type;          /* 锁的类型：读/写等*/
        long        l_whence;        /* 块开始处的类型 */
};
```

注意：在 cmd 取值为 F_GETFL 关键字时，fcntl 函数将返回文件状态标志，如表 8.6 所示。

表 8.6　ftcnl 函数返回的文件状态标志

文件状态标志	说　　明
O_RDONLY	只读
O_WRONLY	只写
O_RDWR	读写
O_APPEND	每次写时追加
O_NONBLOCK	非阻塞模式
O_SYNC	等待数据和属性写完成
O_DSYNC	等待数据写完成
O_RSYNC	同步读写
O_FSYNC	等待写完成
O_ASYNC	异步 I/O 操作

8.3.4　使用 rename 函数修改文件的名称

有时需要对文件的名称进行修改，此时可以使用 rename 函数，其标准调用格式说明如下，如果操作成功返回 0，否则返回-1。

```
#include <stdio.h>
int rename(const char *oldname, const char *newname);
```

rename 函数的各个参数说明如下。

- oldname：文件的旧文件名称，带路径。
- newname：文件的新文件名称，同样带路径。

需要注意的是，rename 函数既可以对文件进行操作，也可以对目录进行操作（前面说过，Linux 中的目录其实质上也是一个文件），rename 函数的参数说明可以总结如表 8.7 所示。

表 8.7 rename 函数的参数说明

	newname 所示 文件不存在	newname 指向 普通文件	newname 指向 目录文件
oldname 指向 普通文件	文件被重命名	newname 被删除，原来名为 oldname 的文件被重命名为 newname	错误
oldname 指向 目录文件	文件被重命名	错误	newname 所指向的目录文件为空目录，则该目录文件被删除，oldname 被重命名，否则出错

8.4 Linux 的目录文件操作

在 Linux 中，目录也是文件的一种，本节将介绍目录文件的基础操作方法。

8.4.1 创建和删除目录

1．使用 mkdir 函数创建目录

mkdir 函数用于在文件系统中建立一个目录，其会自动在目录中创建“.”和“..”目录项，其标准调用格式说明如下：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int mkdir(const char *pathname, mode_t mode);
```

其中 pathname 为目录的带路径名称，mode 为目录的权限，需要注意的是，对于目录来说最少要设置一个执行权限位以允许用户访问该目录中的文件。

这个新创建目录的用户 ID 被设置为调用进程的有效用户 ID，其组 ID 则为父目录的组 ID 或进程的有效组 ID。在新建一个目录之后，mkdir 将更新该目录的 st_atime、st_ctime 和 st_mtime，同时更新其父目录的 st_ctime 和 st_mtime。

注意：由 pathname 指定的新目录的父目录必须存在，并且调用进程必须具有该父目录的写权限及 pathname 涉及的各个分路径目录的搜寻权限。

2．使用 rmdir 函数删除目录

rmdir 函数用于在文件系统中删除一个目录，但是这个目录必须是空目录（只包括“.”和“..”文件项），其标准调用格式说明如下，如果调用成功返回 0，否则返回−1。

```
#include <unistd.h>
int rmdir(const char *pathname);
```

其中 pathname 为目录的带路径名称。

需要注意的是，如果此调用使目录的连接计数为 0，并且没有其他进程打开此目录，则释放由目录占用的空间。如果在连接计数达到 0 时有一个或几个进程打开了此目录，则在此函数返回前删除最后一个连接。另外，在此目录中不能再创建新文件。但是在最后一个进程关闭它之前并不释放此目录（即使某些进程打开该目录，它们在此目录下也不能执行其他操作，因为为使 rmdir 函数成功执行，该目录必须是空的）。

3．【应用实例】——使用 mkdir 函数和 rmdir 建立和删除目录

例 8.6 是使用 mkdir 函数建立一个文件夹然后调用 rmdir 将其删除的实例。

【例 8.6】 使用 mkdir 函数和 rmdir 函数建立和删除目录。

```
#include <fcntl.h>
 #include <stdio.h>
 int main(int argc,char *argv[])
 {
     int temp;
 if(argc!=2)
 {
     perror("argc is wrong!\n");
     return 2;                          //参数错误，退出
 }
 temp = mkdir(*(argv+1),S_IRWXU|S_IRGRP|S_IXOTH); //必须最少指定一个执行权限位
 if(temp == -1)
 {
      printf("new dir error\n");
      return 3;                         //退出
 }
     temp = rmdir(*(argv+1));       //删除刚刚建立的文件夹
     if(temp == 0)
     {
        printf("del done\n");          //删除完成
     }
     return 0;
 }
```

8.4.2　打开、关闭目录及对目录的读操作

在 Linux 系统中，对目录有访问权限的用户都可以对目录进行读操作，但是只有操作系统内核才有权限对目录进行写操作，本小节将介绍三个对目录进行操作的函数。

1．使用 opendir 打开目录

opendir 函数用于打开一个目录，其标准调用格式说明如下：

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir (const char *pathname);
```

函数的参数是目录的完整路径名，如果操作成功，函数返回一个 DIR 类型的指针，如果操作失败则返回 NULL。DIR 指针是一个内部结构，本小节所介绍的三个函数用来保存正被读的目录的有关信息，DIR 指针的具体结构将在后续章节中进行介绍。

2．使用 closedir 关闭目录

和文件操作相同，打开的目录在操作完成之后也必须进行关闭操作，此时可以调用 closedir 函数，其标准调用格式说明如下：

```
#include <sys/types.h>
#include <dirent.h>
int closedir (DIR *dp);
```

其中参数 dp 是一个指向待关闭目录的 DIR 类型指针，如果操作成功，返回 0；否则返回“−1”。

3．使用 readdir 读目录

对目录的读操作可以通过调用 readdir 函数来完成，其标准调用格式说明如下：

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir (DIR *dp);
```

函数同样使用 DIR 类型的指针来指向等待读操作的目录，若操作成功则返回一个 dirent 类型的指针，若在目录尾或出错则为 NULL。

参数 dp 指向要读取的目录，函数返回值为指向 dirent 结构体的指针。dirent 定义在头文件<dirent.h>中：

```
struct dirent
{
    ino_t d_ino;                    /*i-node number*/
    char d_name[NAME_MAX + 1];      /*null-terminated filename*/
}
```

其中 d_ino 表示该目录的节点号，d_name 用于存放此目录链接的文件名。当目录中没有更多链接时，其值为 0。

4．【应用实例】——获得指定目录的文件信息

例 8.7 是调用以上三个函数对指定目录进行遍历操作以得到目录中各种类型的文件数目的应用实例。

【例 8.7】 获得指定目录的文件信息。

```
#include <stdio.h>
#include <fcntl.h>
#include <dirent.h>
#include <limits.h>
#include <sys/stat.h>
//调用文件名
typedef     int     Myfunc(const char *, const struct stat *, int);
static Myfunc       myfunc;
static int          myftw(char *, Myfunc *);
static int          dopath(Myfunc *);

static long  nreg, ndir, nblk, nchr, nfifo, nslink, nsock, ntot;
char*path_alloc(int* size);
int main(int argc, char *argv[])
{
 int          ret;
 if (argc != 2)
 {
       printf("arg error!\n");   //参数错误
      }
 ret = myftw(argv[1], myfunc);                /* does it all */
 ntot = nreg + ndir + nblk + nchr + nfifo + nslink + nsock;
 if (ntot == 0)
        ntot = 1;               /* avoid divide by 0; print 0 for all counts */
 printf("regular files    = %7ld, %5.2f %%\n", nreg,
    nreg*100.0/ntot);
 printf("directories      = %7ld, %5.2f %%\n", ndir,
    ndir*100.0/ntot);
 printf("block special    = %7ld, %5.2f %%\n", nblk,
    nblk*100.0/ntot);
 printf("char special     = %7ld, %5.2f %%\n", nchr,
    nchr*100.0/ntot);
 printf("FIFOs            = %7ld, %5.2f %%\n", nfifo,
    nfifo*100.0/ntot);
 printf("symbolic links = %7ld, %5.2f %%\n", nslink,
    nslink*100.0/ntot);
 printf("sockets          = %7ld, %5.2f %%\n", nsock,
    nsock*100.0/ntot);

 return ret;
}
char*path_alloc(int* size)
{
char *p = NULL;
```

```
if(!size) return NULL;
p = malloc(256);
if(p)
*size = 256;
else
*size = 0;
return p;
}
#define     FTW_F       1               /* file other than directory */
#define     FTW_D       2               /* directory */
#define     FTW_DNR     3               /* directory that can't be read */
#define     FTW_NS      4               /* file that we can't stat */
static char  *fullpath;                 /* contains full pathname for every file */
static int myftw(char *pathname, Myfunc *func)
{
 int len;
 fullpath = path_alloc(&len);       /* malloc's for PATH_MAX+1 bytes */
                                    /* ({Prog pathalloc}) */
 strncpy(fullpath, pathname, len);  /* protect against */
 fullpath[len-1] = 0;               /* buffer overrun */

 return(dopath(func));
}
static int dopath(Myfunc* func)
{
 struct stat        statbuf;
 struct dirent      *dirp;
 DIR                *dp;
 int                ret;
 char               *ptr;

 if (lstat(fullpath, &statbuf) < 0)     /* stat error */
       return(func(fullpath, &statbuf, FTW_NS));
 if (S_ISDIR(statbuf.st_mode) == 0)     /* not a directory */
       return(func(fullpath, &statbuf, FTW_F));
 if ((ret = func(fullpath, &statbuf, FTW_D)) != 0)
       return(ret);
 ptr = fullpath + strlen(fullpath);    /* point to end of fullpath */
 *ptr++ = '/';
 *ptr = 0;

 if ((dp = opendir(fullpath)) == NULL)     /* can't read directory */
       return(func(fullpath, &statbuf, FTW_DNR));

 while ((dirp = readdir(dp)) != NULL) {
       if (strcmp(dirp->d_name, ".") == 0    ||
```

```
            strcmp(dirp->d_name, "..") == 0)
                continue;                   /* ignore dot and dot-dot */
        strcpy(ptr, dirp->d_name);          /* append name after slash */
        if ((ret = dopath(func)) != 0)      /* recursive */
            break;   /* time to leave */
    }
    ptr[-1] = 0;          /* erase everything from slash onwards */

    if (closedir(dp) < 0)
    {
        printf("can't close directory %s\n", fullpath);
       }
    return(ret);
}
static int myfunc(const char *pathname, const struct stat *statptr, int type)
{
    switch (type) {
    case FTW_F:
        switch (statptr->st_mode & S_IFMT) {
        case S_IFREG:    nreg++;        break;
        case S_IFBLK:    nblk++;        break;
        case S_IFCHR:    nchr++;        break;
        case S_IFIFO:    nfifo++;       break;
        case S_IFLNK:    nslink++;      break;
        case S_IFSOCK:   nsock++;       break;
        case S_IFDIR:
            printf("for S_IFDIR for %s\n", pathname);
        }
        break;

    case FTW_D:
        ndir++;
        break;
    case FTW_DNR:
        printf("can't read directory %s\n", pathname);
        break;
    case FTW_NS:
        printf("stat error for %s\n", pathname);
        break;
    default:
        printf("unknown type %d for pathname %s\n", type, pathname);
    }
    return(0);
}
```

8.5 Linux 的流操作基础

文件操作方式通常被称为不带缓冲的 I/O，这是因为每次调用相应的函数（read 或 write 等）对文件进行操作时都会调用内核的系统调用，这样的操作由于每次都要通过内核对文件直接进行，所以操作效率较低；本节将介绍另外一种文件操作方式：流编程；其首先对文件所映射的流进行操作，然后分阶段将相应的数据写入文件，能极大地提高相应的操作效率。

8.5.1 流和文件的关系

文件的 I/O 函数都针对文件描述符进行操作，当调用 open 或其他函数打开一个文件时，即返回一个文件描述符 fd，然后针对该文件描述符来进行后续的 I/O 操作，但是由于其需要多次反复调用对应的系统调用，效率很低，表 8.8 是在调用 read 函数时使用不同缓冲区长度（size_t nbytes）来读 103 316 352 字节文件所需要花费的时间。

表 8.8　不同缓冲区长度下的读文件效率

缓冲区长度（字节）	用户 CPU 时间（秒）	系统 CPU 时间（秒）	时钟时间（秒）	循 环 次 数
1	123.90	161.65	288.64	103 316 352
2	63.10	80.96	145.81	51 658 176
16	7.86	10.27	18.76	6 457 272
64	2.11	2.48	6.76	1 614 318
512	0.27	0.41	7.03	201 789
1024	0.17	0.23	7.84	100 894
4096	0.03	0.16	6.86	25 223
8192	0.01	0.18	6.67	12 611
65535	0.02	0.19	6.92	1576

从表 8.8 中可以看出选择一个合适缓冲区的重要性，而流 I/O 函数的操作则是围绕流（stream）进行的，当使用流 I/O 库打开或创建一个文件时，可以使得一个流与一个文件相结合，接下来的操作过程就和基于文件描述符的 I/O 操作过程类似：对流进行读写、定位操作等，最后关闭流。

图 8.4 是流、文件、基于流的 I/O 操作和基于文件的 I/O 操作的关系。

从图 8.4 中可以看到，所谓的带缓冲和不带缓冲是相对而言的。

不带缓冲的文件 I/O 操作也不是直接对文件进行的，只不过在用户层没有缓存区，所以叫作不带缓冲的 I/O，但对于 Linux 内核来说，还是进行了缓冲。当用户调用不带缓冲的 I/O 函数写入数据到文件时（即对磁盘存储区进行读/写），Linux 内核会先将数据写入到内核中所设的缓冲储存器，假如该缓冲储存器的长度是 50 字节，调用 write 函数进行写操作时，如果每次写入长度为 10 字节，则需要调用 5 次 write 函数，而在这个过程中数据还在内核的缓冲区中，并没有写入到磁盘，当 50 字节已经写满时才进行实际上的 I/O 操作，即把数

据写入到磁盘上。

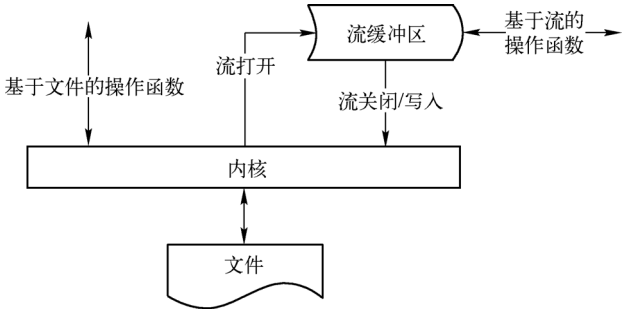


图 8.4　文件和流的关系

带缓冲的 I/O 在用户层建立了另一个缓存区（即流缓冲区），假设流缓存的长度同样也是 50 字节，调用对应的写入库函数时会将数据写入到这个流缓存区里，然后一次性进入内核缓存区，此时再使用系统调用将数据写入到文件（实质是磁盘空间）上，从而减少了系统调用。

总之，带缓冲和不带缓冲的 I/O 函数的单向数据（只有写入没有读出）流向可以总结如下。

- 不带缓冲：数据→内核缓存区→磁盘。
- 带缓冲：数据→流缓存区→内核缓存区→磁盘。

8.5.2　流的结构和操作流程

从 8.5.1 节可以知道，流操作函数的对象不是文件描述符，而是一个流缓冲区；当打开一个流时，返回一个指向 FILE 对象的指针，该对象通常是一个结构体，它包含了为管理该流所需要的所有信息，包括用于实际 I/O 的文件描述符、指向流缓存的指针、缓存的长度、当前在缓存中的字符数、出错标志等，该结构体说明如下。

```
struct file {
        struct list_head        f_list;
        struct dentry           *f_dentry;
        struct vfsmount         *f_vfsmnt;
        struct file_operations  *f_op;
        atomic_t                f_count;
        unsigned int            f_flags;
        mode_t                  f_mode;
        loff_t                  f_pos;
        unsigned long           f_reada, f_ramax, f_raend, f_ralen, f_rawin;
        struct fown_struct      f_owner;
        unsigned int            f_uid, f_gid;
        int                     f_error;
        unsigned long           f_version;
        void                    *private_data;
```

```
    struct kiobuf               *f_iobuf;
    long                        f_iobuf_lock;
};
```

用户的应用代码没有必要对 FILE 对象进行检验，在实际应用中也不需要了解 FILE 的结构，用户只需要知道为了引用一个流，需将 FILE 指针作为参数传递给对应的函数。

流操作流程如图 8.5 所示，需要注意的是和基于文件的操作类似，应在操作之后关闭流，否则容易导致数据丢失。

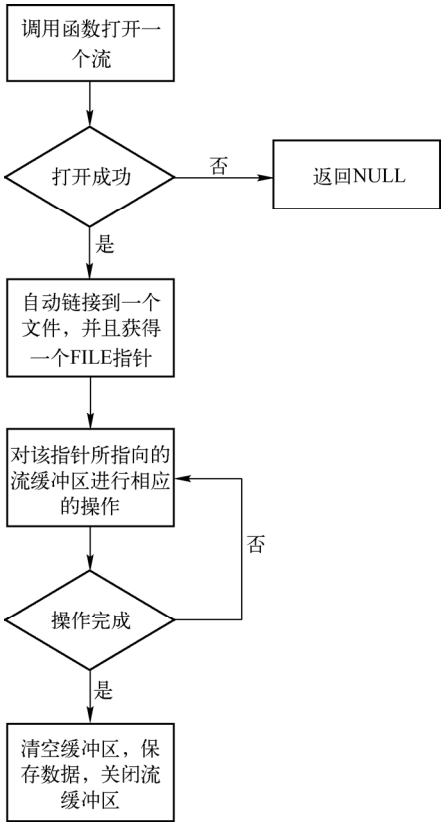


图 8.5　流操作流程

注意：通常来说，用户可以简单地把流看作一块由系统分配的内存缓冲区，在该缓冲区中存放了文件对应的数据。

8.5.3　Linux 的标准流

Linux 有三个标准文件，分别为标准输入、标准输出和标准错误输出。Linux 操作系统对这三个标准文件预定义了三个标准流，可以通过相应的指针调用，其说明如下：

```
#define    STDIN_FILENO     0     //标准输入，对应标准流指针 stdin
#define    STDOUT_FILENO    1     //标准输出，对应标准流指针 stdout
#define    STDERR_FILENO    2     //标准错误输出，对应标准流指针 stderr
```

需要注意的是，这三个标准流都是自动打开和自动关闭的。

8.6　Linux 的流操作

Linux 同样提供了大量的流操作库函数以供用户使用，这些库函数又被称为标准 I/O 库，这是因为这些库函数是跨操作系统平台的，并且是属于 ISO C 的。

8.6.1　打开和关闭流

在对流进行操作之前，必须先打开流；在操作完成之后，必须关闭流。

1．使用 fopen 系列函数打开流

打开流的过程实际上就是建立一个缓冲区，并将这个缓冲区和对应的文件相关联的过程。Linux 提供了 fopen 系列函数来完成相应的工作，其标准调用格式说明如下：

```
#include <stdio.h>
FILE *fopen(const char *path, const char *mode);
FILE *fdopen(int fd, const char *mode);
FILE *freopen(const char *path, const char *mode, FILE *stream);
```

这三个函数的区别说明如下，当调用成功之后返回一个 FILE 类型的文件指针，否则返回一个 NULL 指针。

- fopen 函数：打开一个指定的文件。
- freopen 函数：在一个指定的流上打开一个指定的文件，若该流已经打开，则先关闭该流，若该流已经定向，则立刻进行重定向操作；此函数一般用于将一个指定的文件打开为一个预定义的流：标准输入、标准输出或标准出错。
- fdopen 函数：打开一个由文件描述符所指定的流；此函数常用于由创建管道和网络通信通道函数获得的描述符，因为这些特殊类型的文件不能使用标准 I/O 的 fopen 函数打开，必须先调用设备专用函数以获得一个文件描述符，然后用 fdopen 使一个标准 I/O 流与该描述符相结合。

流打开函数的参数说明如下。

- path 参数：文件的路径。
- fd 参数：文件的文件描述符。
- stream 参数：指定的流。
- mode 参数，流的打开方式，其类似 open 函数中的 mode 参数，用于说明流的打开模式和权限，详细说明如表 8.9 所示。

表 8.9　mode 参数说明

选　项	说　明
r 或 rb	只读打开，文件必须存在
w 或 wb	只写打开，如果该文件存在，则将它的长度截为 0，即该文件将被重新写过；如果该文件不存在，则创建一个新文件

续表

选　　项	说　　明
a 或 ab	添加打开，若文件已经存在，其原来的内容不变且到该流的输出将添加在文件的末尾；否则创建一个新的空文件
r+或 rb+或 r+b	读写打开，文件必须存在，该文件的原内容不变且初始文件位置位于文件开始处
w+或 wb+或 w+b	更新打开，若文件已存在，其长度被截至 0；否则，创建一个新文件
a+或 ab+或 a+b	更新打开，若文件已存在，其原内容不变；否则，创建一个新文件。用于读的初始文件位置位于文件开始处，但输出总是添加到文件的末尾

在表 8.9 中，使用关键标识符“b”来作为 mode 类型的参数，用于区别二进制文件和文本文件，但是由于 Linux 内核并不对这两种类型的文件进行区分，所以其并没有实际意义。另外对于 fdopen 函数来说，由于在获得描述符时该描述符已经被打开，所以在使用“w”或“wb”参数时并不截短该文件，另外使用“a”或“ab”参数也不能用于创建一个文件，因为如果使用一个描述符来引用一个文件，则文件必须已经存在。

针对表 8.9，表 8.10 给出了打开一个流的 6 种不同方式。

表 8.10　打开一个流的 6 种不同方式

限 制 条 件	r	w	a	r+	w+	a+
文件必须已经存在	●	—	—	●	—	—
删除文件以前的内容	—	●	—	—	●	—
流可以读	●	—	—	●	●	●
流可以写	—	●	●	●	●	●
流只能在尾部写	—	—	●	—	—	●

注意：在指定使用“w”或“a”创建一个新文件时，并不能指定该文件的相应权限，如果需要对该文件进行相应的权限设置，必须调用 open 或 create 函数。

2．使用 fclose 函数关闭流

完成对一个流的操作之后，需要调用相应的函数将其关闭，Linux 提供了 fclose 函数实现该操作，其标准调用格式说明如下：

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

函数的参数是一个指向流的指针，调用成功之后返回“0”；否则返回“EOF”，其是一个定义为“-1”的宏。

说明：EOF 也是 THE END OF THE FILE 的缩写，通常用来表示已经到达文件的结尾，将在后续章节中进一步介绍。

当调用 fclose 函数时，将会把流中的数据写入到对应的文件中，并且清除整个缓冲区；如果应用代码不调用该函数，应用代码调用 exit 函数返回时，系统也会自动调用 fclose 函数完成相应的操作。

3.【应用实例】——使用流操作来创建一个文件

例 8.8 是 fopen 和 fclose 函数的应用实例，应用代码先调用 fopen 打开一个字符串参数 1 所指向的文件，如果没有该文件则先创建，然后关闭这个流。

【例 8.8】 使用流操作来创建一个文件。

```
#include <stdio.h>
int main(int argc, char *argv[])
{
        FILE    *fp;
        int     iflag;
        if(argc<=1)     //如果参数不正确
        {
                printf("usage: %s filename\n",argv[0]);
                return 1;
        }
        fp=fopen(argv[1],"a+b");                        //如果没有文件，则建立文件
        if(fp==NULL)
        {
                printf("Open    file %s failed!", argv[1]);
                return 2;
        }
        printf("Open file %s succeed!\n",argv[1]);
        iflag=fclose(fp);                               //关闭文件
        if(iflag==0)
        {
                printf("Close file %s succeed!\n",argv[1]);
                return 0;
    }
    else
    {
        printf("Close file %s failed! ", argv[1]);
        return 3;
    }
}
```

8.6.2　设置流的缓冲区

和基于文件的 I/O 方式比起来，基于流的 I/O 方式的最大特点就是其先对缓冲区进行操作，从而可以大大地提高效率。但是当使用 fopen 系列函数打开一个流时并没有指定这个流对应的缓冲方式和缓冲区大小，因为这是由 Linux 内核来分配的。

1．流的缓冲方式和特征

缓冲方式是指流在什么时候使用内核的系统写入/读出调用来对文件进行操作，其有三种类型的缓冲方式。

- 全缓冲：在这种缓冲方式下，直到缓冲区被填满才使用系统调用进行操作。对于读

操作来说，直到读入的内容的字节数等于缓冲区大小或文件已经到达结尾，才进行实际的 I/O 操作，将外存文件内容读入缓冲区；对于写操作来说，直到缓冲区被填满，才进行实际的 I/O 操作，缓冲区内容写到外存文件中。磁盘文件通常是全缓冲的。在 Linux 内核中使用宏定义_IO_FULL_BUF 表示全缓冲，通常来说，在一个流上进行第一次读写操作时，会调用 malloc 内存分配函数来为流分配一块内存作为缓冲区域。

- 行缓冲：在这种缓冲方式下，如果遇到换行符，使用系统调用进行操作。对于读操作来说，遇到换行符‘\n’才进行 I/O 操作，将所读内容读入缓冲区；对于写操作来说，遇到换行符‘\n’才进行 I/O 操作，将缓冲区内容写到外存中。由于缓冲区的大小是有限的，所以当缓冲区被填满时，即使没有遇到换行符‘\n’，也同样会进行实际的 I/O 操作；标准输入 stdin 和标准输出 stdout 默认都是行缓冲的。在使用行缓冲时有两个限制：第一，每一行对应的缓冲区的长度是固定的（MAXLINE），如果这个缓冲区已经被写满，即使还没有遇到换行符，也会调用系统调用进行工作；第二，在 Linux 内核要求获得数据时，将立即完成一次数据的写入或读出。
- 无缓冲：在这种缓冲方式下没有缓冲区，不进行缓冲，数据会立即读入或输出到外存文件和设备上。标准出错 stderr 是无缓冲的，这样保证了错误提示和输出能及时反馈给用户，供用户排除错误。

Linux 下的流缓冲具有以下两个特征。

- 当且仅当输入和输出不涉及交互式设备时，其才是全缓冲的；如果涉及了终端设备，大部分将是行缓冲。
- 标准出错绝对不是全缓冲的。

前面介绍过，Linux 中的流最终都需要对应到具体的文件（需要注意标准输出设备、输入设备这些在 Linux 中也是以文件形式存在的），所以每一个流都有其对应的文件描述符，可以利用流调用 fileno 函数来获得流对应的文件描述符，其标准调用格式说明如下：

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

2．使用 setbuf 系列函数操作流缓冲区

如果需要对 Linux 内核提供的流缓冲状态进行修改，可以使用 setbuf 系列函数，其标准调用格式说明如下：

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
void setbuffer(FILE *stream, char *buf, size_t size);
void setlinebuf(FILE *stream);
```

这四个函数都必须在流成功打开之后再调用，setbuf 函数没有返回值，setvbuf 函数如果调用成功返回“0”，否则返回一个非“0”，其参数说明如下。

- stream 参数：流指针，指向一个打开的流。

- buf 参数：在 setbuf 函数中指向一个长度为 BUFSIZ 的缓冲区，BUFSIZ 是在 stdio.h 中定义的一个宏，其长度为 8192 字节；在 setvbuf 函数中，缓冲区的大小由 size 确定。
- mode 参数：缓冲类型，包括_IOFBF（全缓冲）、_IOLBF（行缓冲）和_IONBF（不带缓冲）。
- size 参数：缓冲区的大小。

对于 setbuf 函数来说，其不存在 mode 和 size 参数，所以其设定的流通常是全缓冲的，除非这个流和终端设备相关，此时有可能设置为行缓冲，同时可以通过将 buf 设置为 NULL 来关闭缓冲。

对于 setvbuf 函数来说，通过设置 mode 和 size 则可以选择相应的缓冲方式和缓冲区大小，如果设置一个流带缓冲但是 buf 被设置为 NULL，则 Linux 内核会自动将其缓冲区大小设置为 BUFSIZ。表 8.11 是以上两个函数的总结。

表 8.11　setbuf 和 setvbuf 总结

函　数	mode	Buf	缓冲区及其长度	缓 冲 类 型
setbuf		非空	用户缓冲区，长度为 BUFSIZ	全缓冲或行缓冲
		NULL	无缓冲区	不带缓冲
setvbuf	_IOFBF	非空	用户缓冲区，长度为 size	全缓冲
		NULL	系统缓冲区，长度通常为 BUFSIZ	
	_IOLBF	非空	用户缓冲区，长度为 size	行缓冲
	_IONBF	NULL	系统缓冲区，长度通常为 BUFSIZ	
		—	无缓冲区	不带缓冲

- setbuffer 函数将流设置为全缓冲方式，但是其可以指定缓冲区的大小。
- setlinebuf 函数则将流设置为行缓冲方式。

注意：最好在将流打开但还未对流执行其他操作时设定流的属性。因为对流的各种操作都是和缓冲区的属性紧密相关的，改变缓冲区的属性会对所执行的操作产生意想不到的影响。

3.【应用实例】——设置标准输入设备的缓冲方式

例 8.9 是使用 setbuf 函数对标准输入设备进行缓冲设置的实例。

【例 8.9】 设置标准输入设备的缓冲方式。

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 512                                    //定义缓冲区大小
int main(void)
{
    char buf[SIZE];                                 //缓冲区
    if(setvbuf(stdin, buf, _IONBF, SIZE)!=0)        //将标准输入的缓冲类型设为无缓冲
    {
```

```
        perror("set stdin error!\n");
        exit(1);
    }
    printf("Set stdin successful!\n");
    printf("stdin is ");                                    //打印缓冲区信息
    if(stdin->_flags & _IO_UNBUFFERED)          //判断标准输入流对象的缓冲区类型
    {
        printf("unbuffered\n");
    }
    else if(stdin->_flags & _IO_LINE_BUF)
    {
        printf("line-buffered\n");
    }
    else
    {
        printf("fully-buffered\n");
    }
    printf("buffer size is %d\n", stdin->_IO_buf_end - stdin->_IO_buf_base);
    //打印缓冲区的大小
    printf("file discriptor is %d\n", fileno(stdin));   //输出文件描述符
      {
        //将标准输入的缓冲类型设为全缓冲，缓存大小为 512
        perror("error!\n");
        exit(1);        //出错退出
      }
    printf("OK, change successful!\n");
    printf("stdin is ");                                    //打印缓冲区信息
    if(stdin->_flags & _IO_UNBUFFERED)          //判断标准输入流对象的缓冲区类型
    {
        printf("unbuffered\n");
    }
    else if(stdin->_flags & _IO_LINE_BUF)
    {
        printf("line-buffered\n");
    }
    else
    {
        printf("fully-buffered\n");
    }
    printf("buffer size is %d\n", stdin->_IO_buf_end - stdin->_IO_buf_base);
    //打印缓冲区的大小
    printf("file discriptor is %d\n", fileno(stdin));   //输出文件描述符
}
```

8.6.3 使用字符方式对流进行读写

对流的操作其主要目的就是对流所指定的文件进行操作，所以流的读写是流最重要也

是最常见的操作，对流的读写操作可以按照操作的缓冲区大小分为三种。

- 字符读写：每次读写一个字符数据，如果流是带缓存的，则由流 I/O 函数处理所有缓存。
- 行读写：当遇到换行符时，将流中换行符之前的内容送到缓冲区中，即每次读写一行。
- 块（结构）读写：以块（结构）为单位进行读写。

1．使用 getc 和 putc 系列函数读写流

字符读写方式每次从流中读出或写入一个字符的数据，字符读可以调用 getc 系列函数来进行读操作，其标准调用格式说明如下：

```
#include <stdio.h>
int fgetc(FILE *stream);
int getc(FILE *stream);
int getchar(void);
```

函数如果调用成功则返回即将读取的下一个字符，如果已经到达文件结束或出错则返回 EOF，其参数是一个指向流的指针 stream。

前两个函数中，参数 fp 表示所要读入字符的文件，它们的区别是 getc 可被实现为宏，而 fgetc 则不能实现为宏。这意味着：

- getc 的参数不应当是具有副作用的表达式；
- 因为 fgetc 一定是函数，所以可以得到其地址，这就允许将 fgetc 的地址作为一个参数传送给另一个函数；
- 调用 fgetc 所需的时间很可能长于调用 getc，因为调用函数通常所需的时间长于调用宏，事实上在<stdio.h>头文件中，getc 便是以宏定义的形式实现的，其编码具有较高的工作效率。

第三个函数 getchar 只能用来从标准输入流中输入数据，其作用相当于调用以 stdin 为参数的 getc 函数，即 getc(stdin)。

另外，这三个函数以 unsigned char 类型转换为 int 的方式返回下一个字符。说明为不带符号的理由是，即使最高位为 1 也不会使返回值为负。要求整型返回值的理由是，这样就可以返回所有可能的字符值再加上一个已发生错误或已到达文件尾端的指示值。在<stdio.h>中，常数 EOF 被要求是一个负值，其值经常是−1。这就意味着不能将这三个函数的返回值存放在一个字符变量中，以后还要将这些函数的返回值与常数 EOF 相比较。

对应按字符读函数，Linux 内核同样提供了按字符写函数，其标准调用格式说明如下：

```
#include <stdio.h>
int fputc(int c, FILE *stream);
int putc(int c, FILE *stream);
int putchar(int c);
```

函数调用成功则返回输出字符 c，若出错则为 EOF，其参数说明如下。

- c 参数：需要输出的字符。

● stream 参数：接收输出的流指针。

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

2.【应用实例】——字符读写标准输入输出

例 8.10 是对流进行字符读写的应用实例，应用代码按照字符从标准输入读入字符，然后将其按照字符输出到标准输出。

【例 8.10】 字符读写标准输入输出。

```
#include <stdio.h>
#include <errno.h>
int   main(void)
{
        int     c;
        printf("pls enter some str,CTRL+D for stop\n");          //输出提示符
        while ((c = getc(stdin)) != EOF)                         //如果没有接收到 EOF
        {
                if (putc(c, stdout) == EOF)                      //如果输出到 EOF
                {
                        perror("output error");
                }
        }
        if (ferror(stdin))
                perror("input error");
        return 0;
}
```

注意：在 Linux 操作系统下，可以使用 CTRL+D 组合键来输入 EOF 符号。

8.6.4 使用行方式对流进行读写

行读写方式每次从流中读出或写入一行数据。

1. 使用 gets 和 puts 系列函数读写流

行读可以使用 gets 系列函数，其标准调用格式说明如下：

```
#include <stdio.h>
char *fgets(char *s, int size, FILE *stream);
char *gets(char *s);
```

fgets 函数用于从流 stream 中读出一行数据，并且送到由 s 指定的缓冲区中，缓冲区大小由 size 参数说明，函数一直读到遇到下一个换行符或读完了 $n-1$ 个字符；如果需要读入行超过了 $n-1$ 个字符，则只返回一个不完整的行，但是这个缓冲区总是以 NULL 结尾，下一次读取会继续执行；如果操作成功则返回缓冲区，如果已经到达文件结尾或者出错则返回 NULL。

fget 函数和 fgets 函数功能类似，不过其是从标准输入流读取数据。

注意：在实际使用中，并不推荐使用 fgets 函数，这是因为该函数不能指定缓冲区的大小，在实际使用中容易造成缓冲区溢出。

和行读入相对，Linux 内核也提供了相应的行写入 puts 系列函数，其标准调用格式说明如下：

```
#include <stdio.h>
int fputs(const char *s, FILE *stream);
int puts(const char *s);
```

函数 fputs 用于将一个以 NULL 符为终止的字符串去掉 NULL 后写到指定的流，需要注意的是该函数并不要求每次输出一行，因为其并不要求在 NULL 符之前必须是换行符，如果成功则返回一个非负值，如果出错则返回 EOF。

puts 函数先将一个以 NULL 符终止的字符串去掉 NULL，然后写入到标准输出，然后再写一个换行符。

注意：虽然 puts 并不像 gets 那样容易导致错误，但是还是应该尽量避免使用这个函数，因为其涉及第二次写入一个换行符的问题。

2.【应用实例】——流读写标准输入输出

例 8.11 是用 fputs 函数重写例 8.10 应用实例的实例。

【例 8.11】 字符读写应用实例。

```
#include "stdio.h"
#include "errno.h"
#include "stdlib.h"
#define MAXLINE 4096
int main(void)
{
        char buf[MAXLINE];
        printf("pls enter some str,CTRL+D for stop\n");   //输出提示符
        while (fgets(buf, MAXLINE, stdin) != NULL)
        {
                if (fputs(buf, stdout) == EOF)
                {
                        perror("output error");
                }
        }
        if (ferror(stdin))
        {
            perror("input error");
        }
        exit(0);
}
```

使用流缓冲读写对效率的提升是巨大的，表 8.12 是一个使用相应的流缓冲读写操作的动作结果，可以和表 8.8 进行对比。

表 8.12　使用流缓冲读写的效率

函　　数	用户 CPU	系统 CPU	时钟时间	程序正文字节数
表 8.8 中的最佳时间（缓冲区为 8192）	0.01	0.18	6.67	—
fgets 和 fputs 函数	2.59	0.19	7.15	139
getc 和 putc 函数	10.84	0.27	12.07	120
fgetc 和 fputc 函数	10.44	0.27	11.42	120
表 8.8 中单字节时间（缓冲区为 1 字节）	124.89	161.65	288.64	—

可以看到使用流缓冲函数的读写效率远远高于使用文件 I/O 函数，但是需要说明的是系统 CPU 时间差别并不大，这是因为最终都需要调用系统内核调用。

注意：表 8.12 中的最后一列是测试用代码的文本空间字节数，即为 gcc 编译产生的机器指令数目，从中可以看到，使用 getc 和 putc 的版本和使用 fgetc、fputc 的版本在文本空间长度方面大体相同，虽然 getc 和 putc 是通过宏定义来实现的，但是在 gcc 的库调用中是把宏扩展为函数调用的。

8.6.5　使用二进制方式对流进行读写

在读写操作中，如果需要操作的区域多于一个字符乃至多于一行，使用字符读写和行读写同样比较麻烦，并且如果在一行数据中包括了 NULL 字符也会导致行操作的中止，此时可以使用二进制（按块/结构）读写函数。

1．使用 fread 和 fwrite 函数读写流

Linux 提供的二进制读写函数 fread 和 fwrite 的标准调用格式说明如下：

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb,FILE *stream);
```

fread 函数用于执行直接输出操作，其参数 ptr 是指向读取数据的缓冲区的指针；size 是读取对象的大小；nmemb 表示欲读取的对象的个数；fp 是指向要读取的流的 FILE 结构指针，其返回值为读的对象数，如果出错或到达文件尾端，则此数字可以小于 nmemb。在这种情况下，应调用 ferror 或 feof 以判断究竟是哪一种情况。

fwrite 函数用于执行直接输入操作，参数 ptr 是指向存放将要输入数据的缓冲区的指针；size 是写入对象的大小；nmemb 表示欲写入的对象的个数；fp 是指向要写入的流的 FILE 结构指针，其返回值为写的对象数，如果返回值小于所要求的 nmemb，则出错。

fread 和 fwrite 函数有如下两种常见的用法。

- 读或写一个二进制数组，如将一个浮点型数组的第 2～5 个元素写至一个文件上，其代码结构说明如下。

```
float data[10];
if (fwrite(&data[2], sizeof(float), 4, fp) != 4)
printf("fwrite error!\n");
```

其中，指定 size 为每个数组元素的长度，nmemb 为欲写的元素数。

- 读或写一个结构，其代码结构说明如下。

```
struct
{
short count;
long total;
char name[NAME_SIZE];
} item;
if (fwrite(&item, sizeof(item), 1, fp) !=1)
printf ("fwrite error!\n");
```

其中，指定 size 为结构的长度，nmemb 为 1（要写的对象数）。

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

注意：fread 函数和 fwrite 函数的最大问题是，其只能用于读在同一系统上已经写入的数据，这是因为在一个系统上写入的数据可能需要在另外一个系统上运行，从而因为结构体偏移量和存储方式等原因导致出现错误。

2.【应用实例】——使用二进制读写方式实现文件复制

例 8.12 是使用 fread 和 fwirte 函数把例 8.4 所创建的 lseektest.txt 文件中的数据复制到一个新的 txt 文件的实例。

【例 8.12】 使用二进制读写方式实现文件复制。

```
#include   <stdio.h>
#include   <stdlib.h>
#include   <errno.h>
int main(int argc,char *argv[])
{
    FILE *fp1, *fp2;                    //流指针
    char buf[1024];                     //缓冲区
    int n;
    if(argc <=2)                        //如果参数错误
    {
        perror("the arg error\n!");     //参数错误
    }
    if ((fp1 = fopen(*(argv+1), "rb")) == NULL)
    //以只读方式打开源文件，读开始位置为文件开头
    {
        perror("fail to open source file\n");
```

```
            exit(1);            //出错退出
        }
    if ((fp2 = fopen(*(argv+2), "wb")) == NULL)
    //以只写方式打开目标文件，写开始位置为文件结尾
    {
            perror("fail to open des file\n");
            exit(2);            //出错退出
        }
   //开始复制文件，文件可能很大，缓冲一次装不下，所以使用一个循环进行读写*/
    while ((n = fread(buf, sizeof(char), 1024, fp1)) > 0)
    {
        //读源文件，直到将文件内容全部读完*/
        if (fwrite(buf, sizeof(char), n, fp2) == -1)
            {
            //将读出的内容全部写到目标文件中去
            perror("fail to write\n");
            exit(3);        /*出错退出*/
            }
        }
    if(n == -1)
    {
            //如果因为读入字节小于 0 而跳出循环，则说明出错了*/
            perror("fail to read\n");
            exit(4);            /*出错退出*/
    }
    fclose(fp1);                /*操作完毕，关闭源文件和目标文件*/
    fclose(fp2);
    return 0;
}
```

8.6.6 流的出错处理

Fgets、gets、putc、fread 等函数如果调用失败会返回 EOF，但是由于 EOF 既用于报告文件结束也用于报告随机出现的错误，因此，为了区分究竟是错误返回还是文件结束返回，有时还需要调用 ferror 函数来确定是否存在错误，调用 feof 函数检查是否遇到文件结束。

在大多数应用中，Liunx 内核都为流（FILE）对象提供了两个标志符。

- 出错标志：当读写文件出错时该指示器被设置为真（非 0），否则为假（0）。
- 文件结束标志：当已经到达到文件尾时该指示器被设置为真。

Linux 内核同样提供了 ferror 和 feof 两个函数用于检查这两个标志位，其标准调用格式说明如下：

```
#include <stdio.h>
int feof(FILE *stream);
int ferror(FILE *stream);
```

foef 函数和 ferror 函数的参数都是一个指定的流指针，如果其测试标志位为真（非 0）则返回非 0 值，否则返回 0。

在确定了错误之后，可以调用 clearerr 函数来清除错误，其标准调用格式说明如下，其参数是需要清除错误的流对应的指针，没有返回值：

```
#include <stdio.h>
void clearerr(FILE *stream);
```

8.6.7　流的冲洗

在使用流时，内核在系统内核中开辟了一块缓冲区用于相应的操作，在关闭流或操作完成之后，应该将缓冲区的数据清空，这种清空可以是将流的内容完全丢掉，也可以是将其保存到流对应的文件中，这个过程叫作流的冲洗。Linux 内核同样提供了相应的函数 fflush 和_fpurge 来完成流的冲洗，其标准调用格式如下：

```
#include <stdio.h>
int fflush(FILE *stream);
#include <stdio.h>
#include <stdio_ext.h>
void   __fpurge(FILE *stream);
```

fflush 将参数 stream 指定流的缓冲区中尚未写入文件的数据强制性地保存到文件中，如果调用成功，返回值为 0；若调用失败则返回 EOF。

_fpurge 函数则用于将缓冲区中的数据完全清除，由于使用较少，这个函数的定义在<stdio_ext.h>中。

注意：在调用 fclose 函数来关闭一个流，或者一个进程使用 exit、return 函数来正常终止时，流的冲洗是会自动进行的，并不需要用户特意进行操作，但是如果有其他特定的需求，也可以由用户调用以上函数手动完成。

8.6.8　在流中进行内部定位

和在 8.2.5 小节中介绍的文件偏移量类似，流在操作中也存在偏移量的概念，Linux 内核提供了如下三种方式来对流进行定位操作。

- 使用 ftell 和 fseek 函数：其缺点是必须假设偏移量可以放到一个长整型中。
- 使用 ftello 和 fseeko 函数：其使用 off_t 数据类型替代了长整型。
- 使用 fgetpos 和 setpos 函数：使用一个抽象数据类型 fpos_t 来记录文件的位置，该数据类型可以定义为一个文件位置所需要的长度。

1．使用 ftell 和 fseek 函数进行内部定位操作

函数 ftell 和 fseek 的标准调用格式说明如下：

```
#include <stdio.h>
int fseek(FILE *stream, long offset, int whence);
long ftell(FILE *stream);
```

fseek 函数用于修改 fp 所指的文件偏移量，其中参数 fp 是流结构指针；参数 whence 指明参数 offset 的偏移起点，其参考值和 lseek 函数相同，如表 8.13 所示；参数 offset 是流的偏移值，其可以是一个正值也可以是一个负值；如果函数调用成功则返回“0”，否则返回一个非“0”值。

表 8.13　whence 参数取值选项

选　　项	说　　明
SEEK_SET	文件位置定位于文件开始+offset 处
SEEK_CUR	文件位置定位于文件当前位置+offset 处
SEEK_END	文件位置定位于文件尾+offset 处

如果 fseek 函数调用成功，其将清除流的文件结束标志位，如果该流是输出流并且缓冲的数据还未写至相连的文件，fseek 将导致未写出的数据被写至文件；因此，对于以更新方式（“+”）打开的文件而言，调用 fseek 之后，在此文件上的下一个操作既可以是输入也可以是输出。

fseek 函数允许设置文件位置超过文件的当前文件尾，如果之后在此新文件位置写入了数据，则后续从原文件末尾与新写入的数据之间的空隙中读出的字节将用 0 填充，直至此空隙写入实际的数据为止。

ftell 函数的参数是需要操作的流指针，如果调用成功则返回 fp 所指定流的当前文件位置，它是从文件开始的字节数；否则返回“−1”。

针对 ftell 函数和 fseek 函数，Linux 还提供了 rewind 函数，用于将偏移量设定到流的起始部分，其标准调用格式说明如下：

```
#include <stdio.h>
void rewind(FILE *stream);
```

rewind 函数的参数是需要操作的流对应的指针，没有返回值，其等价于 fseek(fp,0L,SEEK_SET)。

2.【应用实例】——向文件写入数据

例 8.13 是 fseek 函数的应用实例，其参考例 8.4 的功能，通过流操作向一个文件中写入了部分数据。

【例 8.13】 向文件写入数据。

```
#include <stdio.h>
 int main(int argc,char *argv[])
 {
    int temp,seektemp,i,j;
    FILE *fp;                    //文件指针
    char wbuf[17] = "this is a test!\r\n";
    if(argc!= 2)
    {
            printf("run error!\n");
```

```
            return 1;                          //如果参数不正确则退出
        }
        fp = fopen(*(argv+1),"a+b");           //打开文件
        for(i=0;i<10;i++)
        {
          j = sizeof(wbuf) * (i+1);            //计算下一次的偏移量
          fseek(fp,j,SEEK_SET);
          temp = fputs(wbuf,fp);               //写入数据
        }
        fclose(fp);
        return 0;
    }
```

3．ftello 和 fseeko 函数

对于二进制文件而言，其文件偏移量可以简单地用一个长整型数据来确定，但是在文本文件中其当前位置不一定能以简单的字节偏移量来度量，这是因为虽然 Linux 并不区分二进制文件和文本文件，但是在其他的操作系统中这两个文件的存放格式可能是不同的，此时可以使用一个 off_t 类型的数据类型，并且使用 ftello 和 fseeko 函数，这两个函数除了位移量的数据类型不同之外，其他方面和 fseek、ftell 函数完全相同，其标准调用格式说明如下：

```
#include <stdio.h>
int fseeko(FILE *stream, off_t offset, int whence);
off_t ftello(FILE *stream);
```

4．fgetpos 和 fsetpos 函数

fgetpos 和 fsetpos 两个函数同样可以用于定位流的操作，fgetpos 可以得到读写指针的位置，而 fsetpos 可以定位读写指针的位置，其标准调用格式如下：

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos(FILE *stream, fpos_t *pos);
```

在这两个函数中，参数 fp 是流指针，pos 为指向 fpos_t 的指针，pos_t 是一个存放指针位置的记录类型，如果操作成功则返回“0”，如果出错则返回非“0”值。

这两个函数和 ftell、fseek 函数的主要区别在于其使用了 fpos_t 结构来存放偏移值，这是一个抽象的结构体，可以在多种不同的操作系统中具体定义。

5．【应用实例】——输出当前文件偏移量

例 8.14 是 fgetpos 函数的应用代码，其是在例 8.4 的基础上在每次写入的时候调用 fgetpos 函数来获取当前的偏移量，并且将其打印输出的实例。

【例 8.14】 输出当前文件偏移量。

```
#include <stdio.h>
 int main(int argc,char *argv[])
 {
   int temp,seektemp,i,j;
```

```
    FILE *fp;                                          //文件指针
    fpos_t ps;                                         //偏移量指针
    char wbuf[17] = "this is a test!\r\n";
    if(argc!= 2)
    {
            printf("run error!\n");
            return 1;                                  //如果参数不正确则退出
    }
    fp = fopen(*(argv+1),"a+b");                       //打开文件
    for(i=0;i<10;i++)
    {
      j = sizeof(wbuf) * (i+1);                        //计算下一次的偏移量
      fseek(fp,j,SEEK_SET);
      temp = fputs(wbuf,fp);                           //写入数据
      fgetpos(fp,&ps);                                 //获得当前的偏移量
      printf("current file end position is %ld \n",ps);//打印偏移量输出
    }
    fclose(fp);
    return 0;
}
```

第9章

在嵌入式 Linux 中进行进程和线程操作

进程是操作系统结构的基础，是一个正在执行的程序实例，是 Linux 操作系统的操作中最重要和最基础的组成部分；线程有时被称为轻量级进程（Light Weight Process，LWP），是程序执行流的最小单元。进程和线程的编程涉及 Linux 的各个方面，本章涉及的内容有：

- Linux 进程的基础知识；
- Linux 进程的基础操作方法；
- Linux 的线程基础知识；
- Linux 线程的基础操作方法。

9.1 Linux 的进程基础

进程的概念起源于 20 世纪 60 年代，目前已成为各种操作系统和并发程序设计中非常重要的概念。可以说，用户在操作系统中所做的每一件事都是通过进程实现的。例如，在第 5 章中调用 fopen 和 fclose 函数来打开和关闭一个流这个实例的执行过程就是一个进程。

进程通常被定义为程序执行时的一个实例。例如，如果 Linux 上有多个用户同时运行 vim，此时就存在多个独立的进程，虽然它们都对应一个可执行代码。

9.1.1 Linux 的进程及其执行过程

Linux 是一个多用户多任务的操作系统，多用户是指多个用户可以在同一时间使用同一台计算机系统；多任务是指 Linux 可以同时执行几个任务，它可以在还未执行完一个任务时又执行另一项任务，Linux 内核管理着多个用户的请求和多个任务。

大多数系统都只有一个处理器实体（在这里把多核处理器看作一个处理器）和一个内存段实体，但一个系统可能有多个二级存储磁盘和多个输入/输出设备。操作系统管理这些资源并在多个用户间共享资源，当用户提出一个请求时，给用户造成一种假象，

好像系统只被用户独自占用。而实际上操作系统监控着一个等待执行的任务队列，这些任务包括用户作业、操作系统任务、邮件和打印作业等。操作系统根据每个任务的优先级为每个任务分配合适的时间片，每个时间片大约都有零点几秒，虽然看起来很短，但实际上已经足够计算机完成成千上万的指令集。每个任务都会被系统运行一段时间，然后挂起，系统转而处理其他任务；过一段时间以后再回来处理这个任务，直到某个任务完成，从任务队列中去除。

Linux 系统上所有运行的任务都可以是一个进程。每个用户任务、系统管理都可以称为进程，Linux 用分时管理方法使所有的任务共同分享系统资源。在讨论进程的时候，不会去关心这些进程究竟是如何分配的，或者内核是如何管理和分配时间片的，人们所关心的是如何去控制这些进程，让它们能够很好地为用户服务。

进程的一个比较正式的定义是：在自身的虚拟地址空间运行的一个单独的程序。进程与程序是有区别的，进程是动态的，程序是静态的；进程不是程序，虽然它由程序产生。程序只是一个静态的命令集合，不占用系统的运行资源；而进程是一个随时都可能发生变化的、动态的、使用系统运行资源的程序，而且一个程序可以启动多个进程。

1．进程的四个要素

在 Linux 中，一个进程必须具有以下四个要素：

- 要有一段程序代码供该进程运行；
- 拥有专用的系统堆栈空间；
- 拥有一个由 task_struck 结构来实现进程控制块；
- 拥有独立的存储空间。

2．进程的关系和分类

Linux 系统中的所有进程都是相互联系的，程序创建的进程之间具有父/子关系，而自进程之间具有兄弟关系。

Linux 内核创建了进程标号为 0 及进程标号为 1（关于进程标号将在 9.1.2 节进行介绍）的进程，其中进程标号为 1 的进程是一个初始化进程（init），Linux 中的所有进程都是由其衍生而来的，在 shell 下执行程序启动的进程则是 shell 进程的子进程，当然所启动的进程可以再启动自己的子进程。这样形成了一棵进程树，每个进程都是树中的一个节点，其中树的根是 init。

进程的关系描述通常需要包括以下几部分。

- p_opptr（祖先，original partent）：其指向创建进程 P 的进程描述符，如果父进程不存在则指向进程 init 的描述符，所以当一个 shell 用户启动一个后台进程并从 shell 退出时，后台进程将变成 init 的子进程。
- p_pptr（父进程，parent）：其指向进程的父进程，其值通常来说和 p_opptr 一致，但是也可能不同。
- p_cptr（子进程，child）：指向进程年龄最小的子进程的描述符，即进程上一次创建的进程描述符。
- p_ysptr（弟进程，younger sibling）：指向在本进程创建之后由父进程创建的进程。
- p_osptr（兄进程，older sibling）：指向在本进程创建之前由父进程创建的进程。

图 9.1 是进程的亲属关系示意。

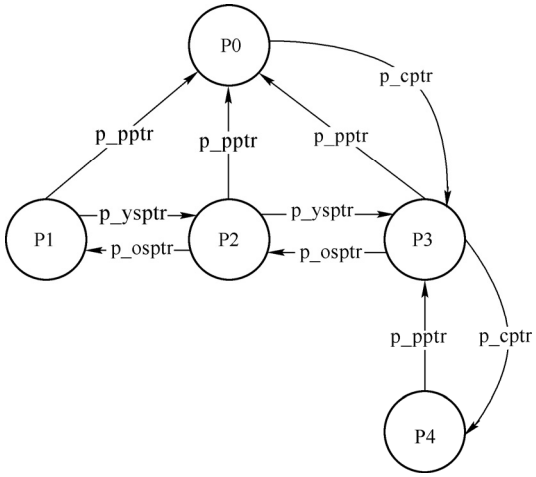


图 9.1　五个进程之间的关系

3．进程的类型

Linux 操作系通常统包括三种不同类型的进程，每种进程都有自己的特点和属性。

- 交互进程：由一个 shell 启动的进程，其既可以在前台运行，也可以在后台运行。
- 批处理进程：这种进程和终端没有联系，是一个进程序列。
- 守护进程：Linux 系统启动时启动的进程，并在后台运行。

4．进程的状态

进程在其生存周期内可能处于以下状态中，需要注意的是这些状态是互斥的，也就是说，在同一时刻进程只能位于其中的一个状态，在 task_struct 结构的状态域中使用不同关键字来定义这些状态。

- 可运行状态（TASK_RUNNING）：占用处理器执行或准备执行。
- 可中断的等待状态（TASK_INTERRUPTIBLE）：进程被挂起或睡眠，当某些条件变成真时才退出这种等待状态，这些条件包括：硬件中断、进程正在等待的系统资源被释放、传递一个信号等，退出等待状态之后的进程会回到 TASK_RUNNING 状态。
- 不可中断的等待状态（TASK_UNINTERRUPTIBLE）：和可中断的等待状态类似，差别是当接收到信号时并不能退出这个等待状态。
- 暂停状态（TASK_STOPPING）：进程的执行被暂停，通常来说当进程接收到 SIGSTOP、SIGTTIN 或 SIGTTOU 信号后，进入暂停状态；需要注意的是如果一个进程被另外一个进程监控，任何信号都可以把这个进程置于 TASK_STOPPEN 状态。
- 僵尸状态（TASK_ZOMBIE）：进程的执行已经被终止，但是父进程还没有使用 wait 系列系统调用已返回相应的信息，此时内核不能丢弃包含在该进程中的相应数据，因为父进程还可能需要这些数据。

进程在这几种状态之间相互转化，但对于用户是透明的，这个切换的过程也常常被称为进程的调度。

进程是一个随执行过程不断变化的实体。和程序要包含指令和数据一样，进程也包含程序计数器和所有处理器寄存器的值，同时它的堆栈中存储着如子程序参数、返回地址及变量之类的临时数据。当前的执行程序，或者说进程，包含着当前处理器中的活动状态。在多处理操作系统中，进程具有独立的权限与职责。如果系统中某个进程崩溃，不会影响到其余的进程。每个进程运行在各自的虚拟地址空间中，它们之间通过一定的通信机制发生联系。

9.1.2 Linux 的进程描述符和标识符

在 Linux 中有很多进程在同时运行，要区别这些进程可以使用两种方式：进程描述符的地址或进程标识符，对于一个系统中的每个独立的进程来说，其对应的进程描述符的地址及进程标识符都是唯一的。

1．进程描述符

为了对进程进行管理，Linux 内核必须了解每个进程当前的执行状态，这些状态包括进程的优先级、进程的运行状态、进程分配的地址空间等；为了达到这个目的，Linux 内核提供了一个 task_struct 类型的结构体进程描述符（process descriptor）来存放相关信息。

Linux 内核提供了一个数组 task，用于存放进程描述符，其包含指向系统中所有 task_struct 结构的指针。这意味着系统中的最大进程数目受 task 数组大小的限制，默认值一般为 512。创建新进程时，Linux 将从系统内存中分配一个 task_struct 结构并将其加入 task 数组。当前运行进程的结构用 current 指针来指示。

以下是一个进程描述符的主要结构及其说明：

```
struct task_struct {
volatile long state;
//进程的运行时状态，–1 代表不可运行，0 代表可运行，>0 代表已停止。
unsigned int flags;
//flags 是进程当前的状态标识，具体说明如下：
//0x00000002 表示进程正在被创建；
//0x00000004 表示进程正准备退出；
//0x00000040 表示此进程被 fork 出，但是并没有执行 exec；
//0x00000400 表示此进程由于其他进程发送相关信号而被杀死。
 unsigned int rt_priority;
//进程的运行优先级
 truct list_head tasks;
 //list_head 结构体
 struct mm_struct *mm;
 //程内存使用的相关情况
  int exit_state;
  int exit_code, exit_signal;
```

```
    pid_t pid;
  //进程标识号
    pid_t tgid;
  //进程组号
  struct task_struct *real_parent;
  //real_parent 是该进程的“亲生父亲”，不管其是否被“寄养”
    struct task_struct *parent;
  //parent 是该进程现在的父进程，有可能是“继父”
    struct list_head children;
  //children 指的是该进程孩子的链表，可以得到所有子进程的进程描述符
  struct list_head sibling;
  //sibling 该进程兄弟的链表，也就是其父亲的所有孩子的链表。用法与 children 相似
 struct task_struct *group_leader;
  //主线程的进程描述符
  struct list_head thread_group;
  //进程所有线程的链表
  处理器 time_t utime, stime;
//进程相关时间
  struct timespec start_time;
  struct timespec real_start_time;
  //进程启动时间
 char comm[TASK_COMM_LEN];
  //这个是该进程所有线程的链表
    int link_count, total_link_count;
  //文件系统信息计数
    struct thread_struct thread;
  //特定处理器下的状态
    struct fs_struct *fs;
  //文件系统相关信息结构体
    struct files_struct *files;
  //打开的文件相关信息结构体
    struct signal_struct *signal;
    struct sighand_struct *sighand;
  //信号相关信息的句柄
    unsigned long timer_slack_ns;
    unsigned long default_timer_slack_ns;
  //松弛时间值，用来规定 select()和 poll()的超时时间，单位是纳秒
 };
```

2．进程标识符

进程标识符（process ID）是进程描述符中最重要的组成部分，其是一个在当前 Linux 系统中唯一的非负整数，用于标识和对应唯一的进程。

Linux 内核使用了一个数据类型 pid_t 来存放进程的进程标识符，这个数据类型的实质是一个 32 位的无符号整型数据。进程标识符被顺序编号，通常来说是前一个进程的进程标

识符的值加 1；进程标识符是可以重复使用的，当一个进程被回收之后，过一段时间，其标识符则可以被再次使用。为了和 16 位处理器架构的应用系统兼容，Linux 内核上通常允许使用的进程标识符是 0～32767。

在 Linux 中，有如下几个特殊的进程标识符。

- 进程标识符 0：对应的是交换进程（swapper），其用于执行多进程的调用。
- 进程标识符 1：对应的是初始化进程（init），在自举过程结束时由内核调用，其对应的文件是/sbin/init，负责 Linux 的启动工作，这个进程在系统运行过程中是不会终止的，可以说当前操作系统中的所有进程都是这个进程衍生而来的。
- 进程标识符 2：可能对应页守护进程（pagedaemon），用于虚拟存储系统的分页操作。

使用命令 ps-aux 可以查看系统中当前正在运行的进程的标识符及其他一些信息，以下列出了开始的几个进程：

```
USER   PID  %处理器 %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root     1   0.0  0.0   3632  1972 ?        Ss   Jul02   0:00 /sbin/init
root     2   0.0  0.0      0     0 ?        S    Jul02   0:00 [kthreadd]
root     3   0.0  0.0      0     0 ?        S    Jul02   0:05 [ksoftirqd/0]
root     6   0.0  0.0      0     0 ?        S    Jul02   0:00 [migration/0]
root     7   0.0  0.0      0     0 ?        S    Jul02   0:02 [watchdog/0]
root     8   0.0  0.0      0     0 ?        S    Jul02   0:00 [migration/1]
root    10   0.0  0.0      0     0 ?        S    Jul02   0:05 [ksoftirqd/1]
```

可以使用 getpid 系列函数来获得当前进程的进程标识符，其标准调用格式说明如下：

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
```

getpid 函数用于获得当前调用进程的进程标识符，getppid 用于获得当前调用进程的父进程的进程标识符。

9.1.3 【应用实例】——获取进程的用户标识符

和 Linux 的文件访问权限类似，进程也有对应的实际用户 ID、实际组 ID、有效用户 ID、有效组 ID，对于这些用户，每个进程同样存在一个相应的标识符，Linux 提供了相应的函数用于获取这些标识符，其标准调用格式说明如下：

```
#include <unistd.h>
#include <sys/types.h>
uid_t getuid(void);
uid_t geteuid(void);
gid_t getgid(void);
gid_t getegid(void);
```

以上 4 个函数的说明如下。

- getuid：返回进程的实际用户标志符。
- geteuid：返回调用进程的有效用户标识符。
- getgid：返回调用进程的实际组标识符。
- getegid：返回调用进程的有效组标识符。

注意：uid_t 和 gid_t 的数据类型和 pid_t 类似。

例 9.1 是一个使用相应函数获得对应的标识符的应用实例。

【例 9.1】 获取进程的用户标识符。

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
main()
{
   printf("The current process ID is %d\n",getpid());          //进程标识符
   printf("The father process ID is %d\n",getppid());          //父进程标识符
   printf("The user true ID is %d\n",getuid());                //实际用户标识符
   printf("The valid user ID is%d\n",geteuid());               //有效用户标识符
   printf("The group ID is %d\n",getgid());                    //实际组标识符
   printf("The valid group ID is %d\n",getegid());             //有效组标识符
   return 0;
}
```

9.1.4　Linux 的进程调度

在 Linux 系统中，进程有两种运行模式：用户模式和系统模式。用户模式的权限比系统模式下的小很多，对于一般的进程，部分时间运行于用户模式、部分时间运行于系统模式。进程通过系统调用在这两种模式之间切换；当系统调用发生时，进程将由用户模式切换到系统模式继续执行；当系统调用返回时，进程将由系统模式切换回用户模式。

在 Linux 系统中，进程不能被抢占。只要能够运行它们就不会被停止。当进程必须等待某个系统事件时，它才决定释放处理器。进程常因为执行系统调用需要等待。由于处于等待状态的进程还可能占用处理器时间，所以 Linux 采用了预加载调度策略。在此策略中，每个进程只允许运行很短的时间（200ms），当这个时间用完之后，系统将选择另一个进程来运行，原来的进程必须等待一段时间以继续运行，这段时间称为时间片。

可运行进程是一个只等待处理器资源的进程。Linux 使用基于优先级的简单调度算法来选择下一个运行进程。当选定新进程后，系统必须将当前进程的状态、处理器中的寄存器及上下文状态保存到 task_struct 结构中。同时它将重新设置新进程的状态并将系统控制权交给此进程。为了将处理器时间合理地分配给系统中每个可执行进程，调度管理器必须将这些时间信息也保存在 task_struct 中。

在 task_struct 结构中保存的调度信息如表 9.1 所示。

表 9.1　进程调度信息

字　段　名	含　　义
policy	该字段表示了进程的调度策略。系统中有两类进程：普通进程与实时进程。实时进程的优先级要高于普通进程。实时进程也有两种策略：时间片轮转和先进先出
priority	该字段表示了实时进程的相对优先级
rt_priority	该字段表示了实时进程的相对优先级
counter	该字段表示了进程允许运行的时间。进程首次运行时为进程优先级的数值，它随时间变化递减

9.1.5　Linux 下的进程执行流程

图 9.2 是 Linux 下，一个标准进程从开始建立到取消的详细过程，9.2 节将按照这个步骤详细地介绍 Linux 下的进程操作相关知识。

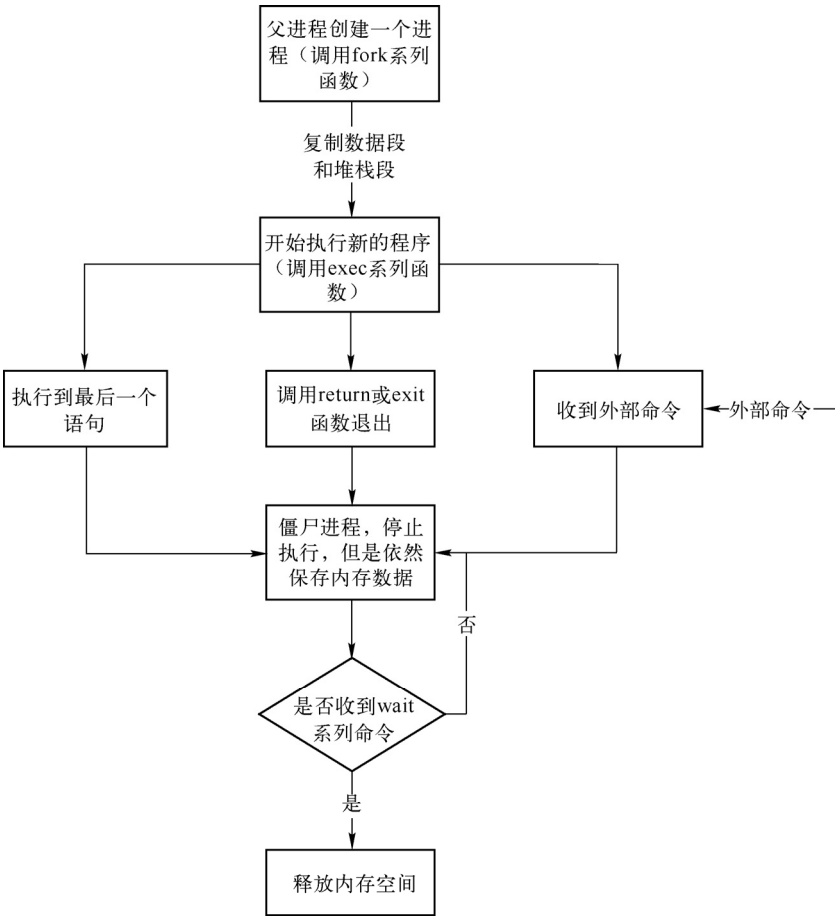


图 9.2　Linux 下的进程执行过程

9.2　在嵌入式 Linux 中进行进程操作

Linux 的进程控制包括进程的创建、执行新的应用、内存的退出和销毁等操作，这些控制通常来说都是通过相应的函数调用来实现的。

9.2.1　使用 fork 和 vfork 函数创建进程

在 Linux 内核中，创建一个进程是对进程进行操作的基础，创建一个新进程的唯一方法是由某个已存在的进程调用 fork 或 vfork 函数，被创建的新进程称为子进程（child process），已存在的进程称为父进程（father process）。

1．fork 函数基础

fork 函数的实质是一个系统调用（和 write 函数类似），其作用是创建一个新的进程，当一个进程调用它，完成后就出现两个几乎一模一样的进程，其中由 fork 创建的新进程称为子进程，而将原来的进程称为父进程。子进程是父进程的一个副本，即子进程从父进程得到了数据段和堆栈段的副本，这些需要分配新的内存；而对于只读的代码段，通常使用共享内存的方式访问。

用户通常在有如下需求时使用 fork 函数。

- 一个进程希望复制自身，从而使得父子进程能同时执行不同段的代码，通常来说这种应用会涉及网络服务：父进程等待远端的一个请求或应答，当收到这个请求或应答时调用 fork 创建一个子进程来完成处理而自己继续等待远端的请求或应答。
- 进程想执行另外一个程序，如在 shell 中调用用户所生成的应用程序。

fork 函数的标准调用格式说明如下：

```
#include <unistd.h>
pid_t fork(void);
```

fork 函数被调用一次，但返回两次。

- 对于父进程而言：函数的返回值是子进程的进程标识符，因为一个进程的子进程可以多于一个，所以没有一个函数使一个进程可以获得其所有子进程的进程标识符，必须通过这种方式来收集。
- 对于子进程而言：函数的返回值是 0，一个进程只有一个父进程，所以子进程总是可以调用 getppid 来获得其父进程的进程标识符，所以不需要在这里返回父进程的进程标识符。
- 如果出错：返回值为“–1”。

当 fork 函数返回后，子进程和父进程都从调用 fork 函数的下一条语句开始执行，例 9.2 是一个调用 fork 函数创建进程的应用实例。

注意：当 fork 函数返回之后，父进程还是子进程先执行是随机的，这个取决于具体的调度算法，如果需要确定让其中一个先运行，可以使用 sleep 等函数让其中一个“休眠”一段时间，但是这个时间是不确定的。

2.【应用实例】——创建子进程

例 9.2 是使用 fork 函数来创建一个子进程，然后在子进程和父进程中分别输出对应的字符串的实例。

【例 9.2】 创建子进程。

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main (void)
{
    int count = 0;
    pid_t pid;              //此时仅有一个进程
    pid = fork();           //此时已经有两个进程在同时运行
    if(pid < 0)
      {
        printf("error in fork!");
        exit(1);       //fork 出错，退出
      }
    else if(pid==0)
      printf("I am the child process, the count is %d, my process ID is %d\n",count,getpid());
    else
      printf("I am the parent process, the count is %d, my process ID is %d\n",++count,getpid());
    return 0;
}
```

3. 父进程和子进程的共享资源

通常来说，fork 所创建的子进程将会从父进程中复制父进程的数据空间、堆和堆栈，并且和父进程一起共享正文段，需要注意的是，子进程所复制的仅是一个副本，和父进程相应部分是完全独立的。除此之外，父进程的所有打开的文件描述符也会被复制到子进程中，此时父进程和子进程的每个打开的文件描述符会共享同一个文件表项。

图 9.3 是父进程和子进程共享文件的示意。

如图 9.3 所示，fork 所创建的子进程和父进程一起共享同一个文件的偏移量，此时如果父进程和子进程同时对同一个文件进行写操作且没有任何形式的同步操作，则会出现写文件的混乱。

在 fork 函数之后处理文件描述符通常来说可以采取如下两种方法。

- 父进程等待子进程执行完成：在这种情况下，父进程无须对文件描述符进行任何操作，而当子进程操作完成之后，文件的偏移量已经进行了相应的更新。
- 父进程和子进程各自执行其对应的程序段，在这种情况下父进程和子进程关闭掉其不需要的文件描述符以防止干扰到另外一方。

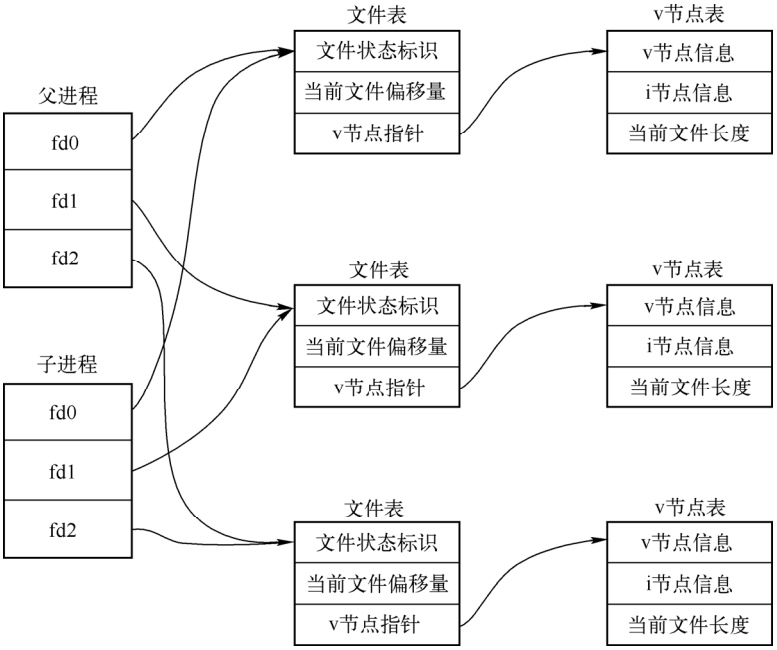


图 9.3　父进程和子进程对文件的共享

表 9.2 是父进程和子进程比较。

表 9.2　父进程和子进程比较

相　　同	不　　同
实际用户 ID、实际组 ID、有效用户 ID、有效组 ID	fork 函数的返回值
附加组 ID	进程 ID 不同
进程组 ID	父进程 ID
会话 ID	子进程的 tms_utime、tms_atime、tms_cutime 及 tms_ustime 均被设置为 0
控制终端	子进程不会继承父进程的文件锁
设置用户 ID 标志和设置组 ID 标志	子进程的未处理闹钟会被清除
当前工作目录	子进程的未处理信号集为空
根目录	
文件模式创建屏蔽字	
信号屏蔽和安排	
针对任一打开文件描述符的在执行时关闭标志	
环境	
连接的共享存储段	
存储映射	
资源限制	

注意：表 9.2 中给出了部分尚未介绍或本书不涉及的参数，读者可以自行查阅相关资料。

通常来说，fork 函数在调用过程中不会产生错误，如果出现错误，将一定是如下两种情况中的一种，关于 errno 值的概念可以参考第 3.4 节。

- 当前的进程数已经达到了系统规定的上限，这时 errno 的值被设置为 EAGAIN。
- 系统内存不足，这时 errno 的值被设置为 ENOMEM，如果出现这种错误，则说明系统已经没有可以分配的内存，这种情况在 Linux 操作系统中通常不会出现。

注意：在大多数情况下 fork 函数会和 exec 系列函数搭配使用，在创建一个进程之后用于执行另外一段代码，exec 系列函数将在 9.2.2 小节中进行介绍。

4．vfork 函数基础

在使用 fork 函数创建一个新进程之后，可以不使用 exec 系列函数来执行新的程序，如果要执行新的程序，必须手动调用 exec 系列函数，在这种情况下可以使用 vfork 函数，vfork 函数在创建完一个新的进程之后自动实现 exec 系列函数的功能，其标准调用格式说明如下：

```
#include <sys/types.h>
#include <unistd.h>
pid_t vfork(void);
```

函数的返回和 fork 函数类似，父进程中返回子进程的进程号，在子进程中返回 0，若出错则返回−1。

fork 与 vfork 之间的区别如下。

- fork 要复制父进程的数据段；而 vfork 则不需要完全复制父进程的数据段，在子进程没有调用 exec 系列函数或 exit 函数之前，子进程与父进程共享数据段。
- vfork 函数会自动调用 exec 系列函数去执行另外一个程序。
- fork 不对父子进程的执行次序进行任何限制；而在 vfork 调用中，子进程先运行，父进程挂起，直到子进程调用了 exec 系列函数或 exit 之后，父子进程的执行次序才不再有限制。

5.【应用实例】——公共计数器计数

例 9.3 是一个 vfork 函数的应用实例，其分别利用子进程和父进程对一个公共 count 计数器进行计数并输出，用于展示父进程和子进程是共享一个数据段的。

【例 9.3】 公共计数器计数。

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main (void)
{
    int count = 1;
    int child;
```

```
        printf("Before create son, the father's count is:%d\n", count);     //创建子进程之前
        if(!(child = vfork()))     //创建子进程
          {
            //由于子进程会首先执行，以下为子进程执行过程
            int i;
            for(i = 0; i < 100; i++)
              {
                printf("This is son, The i is: %d\n", i);     //反复输出打印结果
                if(i == 70)
                exit(1);
              }
             printf("This is son, his pid is: %d and the count is: %d\n", getpid(), ++count);
             exit(1);   //退出子进程
          }
         else
          {   //父进程执行区
             printf("After son, This is father, his pid is: %d and the count is: %d, and the child is: %d\n",
getpid(), coun, child);
          }
    return 0;
    }
```

9.2.2　使用 exec 系列函数执行进程

在使用 fork 函数创建一个进程之后，往往需要调用 exec 系列函数来执行另外一个程序，当进程调用 exec 系列函数时，该进程执行的程序被立即替换为新的程序，而新程序则从 main 函数开始执行，并且立刻替换掉当前进程的正文段、数据段、堆和堆栈，需要注意的是其进行标识符和进程描述符是不会改变的。

exec 系列函数的标准调用格式说明如下：

```
#include <unistd.h>
int execl(const char *path, const char *arg, ...);
int execv(const char *path, char *const argv[]);
int execle(const char *path, const char *arg, ..., char * const envp[]);
int execlp(const char *file, const char *arg, ...);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],char *const envp[]);
```

这六个函数如果调用成功，则都没有返回值，如果出错则返回“–1”，其参数说明如下。

- 参数 pathname：指出一个可执行目标文件的路径名。
- 参数 filename：指出可执行目标文件的文件名。
- 参数 arg0：作为约定，同 pathname 一样指出目标文件的路径名。
- 参数 argv：是一个字符指针数组，由它指出该目标程序使用的命令行参数表，按约

定第一个字符指针指向与 pathname 或 filename 相同的字符串，最后一个指针指向一个空字符串，其余的指针指向该程序执行时所带的命令行参数。

- 参数 envp：与 argv 一样也是一个字符指针数组，由它指出该目标程序执行时的进程环境，它也以一个空指针结束。

事实上，这六个函数中只有 execve 函数才是真正意义上的系统调用，其他函数都是在此基础上经过包装的库函数。exec 函数族的作用是根据指定的文件名找到可执行文件，并用它来取代调用进程的内容。换句话说，就是在调用进程内部执行一个可执行文件。这里的可执行文件既可以是二进制文件，也可以是任何 Linux 下可执行的脚本文件。图 9.4 是这六个 exec 系列函数之间的关系。

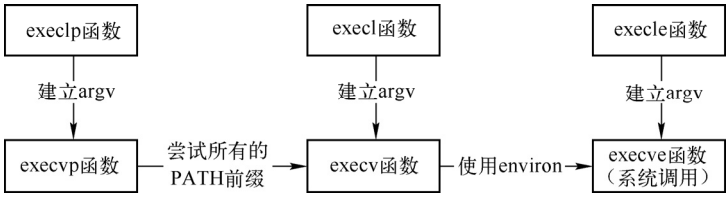


图 9.4　六个 exec 系列函数之间的关系

1．exec 系列函数基础

exec 系列函数的主要区别说明如下。

- execl 函数、execv 函数、execle 函数、execve 函数使用 pathname 参数，取路径名作为参数；而 execlp 函数和 execvp 函数取文件名作为参数。
- execl 函数、execle 函数、execlp 函数中的“l”字符表示“list”，其要求将新程序的每个命令行参数都说明为一个单独的、以空指针为结尾的参数表；execv 函数、execve 函数和 execvp 函数中的“v”字符表示“vector”，其要求先构造一个指向各个参数的指针数组，然后将该数组地址作为其参数。

execl、execel 和 execlp 这三个函数用于表示命令行参数的一般方式是：

```
char *arg0, char *arg1,....char *argn,(char *)0
```

需要注意的是，在命令行参数中使用了一个将常数 0 强制转换为空指针的字符指针来作为结尾，因为如果进行强制转换，则其会被解释为整型参数从而出错。

execle 和 execve 函数中的最后一个字符“e”表示可以向新的进程传递一个环境变量 envp，其命令参数可以说明如下：

```
char *arg0, char *arg1,....char *argn,(char *)0,char *envp[]
```

环境变量指的是一组值，从 Linux 用户登录后就一直存在，很多应用程序需要依靠它来确定系统的一些细节，最常见的环境变量是路径（PATH），其指明了应到哪里去搜索相应的应用程序，如/bin；另外，HOME 也是比较常见的环境变量，其指明了用户在系统中的个人目录；环境变量一般以字符串“XXX=xxx”的形式存在，XXX 表示变量名，xxx 表示变量的值。

2.【应用实例】——传递环境变量

例 9.4 是使用 exec 系列函数的 envp 参数来传递环境变量的应用实例。

【例 9.4】 传递环境变量。

```
#include <stdio.h>
int main(int argc, char *argv[ ], char *envp[ ])
{
    printf("This is argc\n%d\n", argc);   //首先打印参数的数目
    printf("This is argv\n");   //以下打印参数列表
    while(*argv)   //如果不为空，则输出这些字符串
    {
        printf("%s\n", *(argv++));
    }
    printf("This is envp\n");   //以下是 envp 字符串参数
    while(*envp)     //输出 envp 参数
    {
        printf("%s\n", *(envp++));
    }
     return 0;
}
```

3．exec 系列函数总结

exec 系列函数中的字母“p”表示该函数使用 filename 作为参数，并且使用 PATH 环境变量来寻找可执行文件；字母“l”表示该函数使用参数表作为参数；字母“v”表示该函数使用一个 argv 变量作为参数；字母“e”表示使用 envp 数组作为环境变量而不是使用当前的环境变量，表 9.3 是 exec 函数之间的比较。

表 9.3　exec 函数之间的比较

函　数　名	pathname 参数	filename 参数	参　数　表	argv[]	environ 参数	envp[]
execl 函数	●		●		●	
execlp 函数		●	●		●	
execle 函数	●		●			●
execv 函数	●			●	●	
execvp 函数		●		●	●	
execve 函数	●			●		●
函数名中的字母		p	l	v		e

在 exec 系列函数执行之后，不仅进程的描述符、标识符没有发生改变，该进程的如下特征也将保留：

- 进程标识符和父进程标识符；
- 实际用户 ID、实际组 ID；
- 附加组 ID；

- 进程组 ID；
- 会话 ID；
- 闹钟剩余时间；
- 控制终端；
- 当前工作目录；
- 根目录；
- 文件模式创建屏蔽字；
- 文件锁；
- 进程信号屏蔽；
- 未处理信号；
- 资源限制；
- tms_utime、tms_stime、tms_cutime 及 tms_cstime。

注意：在执行 exec 系列函数前后，实际用户 ID 和实际组 ID 是保持不变的，而有效 ID 是否改变则取决于所执行程序文件的设置用户 ID 位和设置组 ID 位是否设置，如果新程序的设置用户 ID 位已经被设置，则有效用户 ID 会变成程序文件所有者的 ID，否则有效用户 ID 不变，组 ID 的处理方式类似。

4.【应用实例】——打印当前系统时间和日期

例 9.5 是一个使用 fork 函数建立一个子进程，然后在子进程中使用 execl 函数调用一个命令的实例，应用代码打印输出当前的系统时间和日期信息。

【例 9.5】 打印当前系统时间和日期。

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(void)
{
   pid_t pid;     //进程标识符
   printf("**This is a test for exec series fun**\n"); //打印提示符
   if(fork()==0)   //创建一个子进程
   {
      execl("/bin/date","/bin/date",(char*)0);   //使用 execl 函数调用 date 命令
      exit(0);   //退出
   }
   else
  {
     sleep(2);   //主进程休眠
  }
   exit(0);
}
```

9.2.3　使用 exit 系列函数退出进程

1．exit 系列函数基础

一个进程执行完成之后必须要退出，退出时内核会进行一系列的操作，包括冲洗缓冲区等，在 Linux 中一共有八种进程退出方法，其中包括五种正常退出和三种异常退出。通常来说 Linux 的应用代码会调用 exit 系列函数来退出一个进程，其标准调用格式说明如下：

```
#include <stdlib.h>
#include <unistd.h>
void exit(int status);
void _exit(int status);
void _Exit(int status);
```

exit 系列函数没有返回值，其使用一个称为终止状态（exit status）的整型变量作为参数，Linux 内核会对这个终止状态进行检查。当异常终止时，Linux 内核会直接产生一个终止状态字，描述异常终止的原因，可以通过 wait 或 waitpid 函数（将在 9.2.4 节进行介绍）来获得终止状态字；父进程也可以通过检查终止状态来获得子进程的状态。如果是以下三种状态：

- 在调用 exit 系列函数时不带终止状态；
- main 函数执行了一个无返回值的 return；
- main 函数的返回值不是一个整型。

则 Linux 会认为该进程的终止状态是未定义，如果 main 函数的返回值定义为整型并且 main 函数执行到最后一条语句返回，则该进程的终止状态是 0。

注意：*在 main 函数中调用 return 语句返回在绝大多数情况下是等效于调用 exit 系列函数的。*

exit 函数与_exit 函数最大的区别在于：前者在调用之前要检查文件的打开情况，把文件缓冲区中的内容写回文件；而后者直接使进程停止运行，清除其使用的内存空间，并销毁其在内核中的各种数据结构。

2.【应用实例】——exit 和_exit 函数的区别

例 9.6 是一个利用 printf 函数要读到换行符才会从缓冲区读取数据的特性来对 exit 函数和_exit 函数进行比较的应用实例。

【例 9.6】 exit 和_exit 函数的区别。

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
int main(void)
{
    pid_t pid;
```

```
    if ( (pid = fork() ) ==–1 )    //如果创建子进程失败
     {
        perror ("failed to create a new process\n");   //创建子进程出错信息
        exit(0);
     }
    else if(pid==0)   //子进程
     {
        printf("This is child process, output begin\n");
        printf("This child process, content in buffer");
        //这个地方没有换行符，所以不写出数据
        exit(0);    //退出，强制清空，会输出上面未完成数据
     }
    else   //父进程
     {
        printf("\nparent process, output begin\n");
        printf("parent process, content in buffer");   //同样没有换行符
        _exit(0);   //_exit 函数会直接丢弃相应的数据
     }
     return 0;
}
```

在一个进程退出时，可能存在如下两种状态。

- 其父进程恰好忙于处理其他事务，不能接收子进程的终止状态，如果此时子进程完全消失了，那么当父进程处理完其他事务后想要检查子进程情况时，就没有可用的信息了，所以 Linux 内核为每个已结束的进程保留一定的信息，一般至少包含进程标识符、终止状态字、进程处理器时间等信息。任何时候父进程可以通过调用 waitt 或 waitpid 函数都能得到相应的数据，在此之后，Linux 内核再将保存这些信息的数据结构释放。通常把这种已经结束但其父近程尚未检查其终止状态的进程称为僵尸进程。
- 父进程可能先于子进程结束，此时 init 进程就会自动成为该子进程的父进程。通常的实现机制是，当一个进程结束时，系统逐一检查所有的活动进程，如果某进程的父进程是这个被结束的进程，系统就将这个活动进程的父进程标识符置为 1，即 init 的进程标识符，这样就保证了每个进程都有它的父进程。

注意：由以上可以知道当调用 exit 系列函数或 return 函数返回时，其实进程并没有真正地完全消失，其还在继续占用部分资源；如果这种僵尸进程过多，则会大大影响系统性能，在 9.2.4 节将介绍如何处理僵尸进程。

9.2.4 调用 wait 系列函数销毁进程

在 9.2.3 节中介绍过当一个进程使用 exit 系列函数退出时，其会在内存中保留部分数据以供父进程查询；同时其也会产生一个终止状态字，然后 Linux 内核会发出一个 SIGCHLD 信号以通知父进程，因为子进程的结束对于父进程是异步的，因而这个 SIGCHLD 信号对于

父进程也是异步的，父进程可以不响应。也可以调用 wait 函数或 waitpid 函数进行处理。

父进程对于退出之后的子进程的默认状态是不处理的。事实上，在以前给出的实例中也都没有处理，但是这样会导致系统中的僵尸进程过多而浪费系统资源。

wait 和 waitpid 函数的标准调用格式说明如下：

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

在调用 wait 或 waitpid 函数之后可能存在如下三种情况。

- 如果该父进程的所有子进程都还在运行，阻塞父进程自身以等待子进程的运行结束。
- 如果有一个子进程已经结束，父进程取得该子进程的终止状态，并且立即返回。
- 如果该父进程没有任何子进程，则立即出错返回。

1．wait 函数基础

wait 函数如果调用成功则返回子进程的标识符，如果失败则返回–1；其中参数 status 是一个整型指针，可用于存放子进程的终止状态，也可以定义为一个空指针。

wait 函数和 waitpid 函数不同，在有一个子进程终止之前，wait 函数让父进程阻塞以等待子进程退出而 waitpid 有一个参数可以让父进程不阻塞（将在 9.2.5 小节中介绍）；并且在一个父进程有多个子进程的情况下，如果其中有一个子进程退出，则会返回该子进程的进程标识符，表 9.4 是 wait 函数返回的终止状态的宏。

表 9.4　wait 函数返回的宏

宏	说　明
WIFEXITED(status)	当子进程正常结束时返回为真
WIFSIGNALED(status)	当子进程异常结束时返回为真
WEXITSTATUS(status)	当 WIFEXITED(status)为真时调用，返回状态字的低 8 位
WTERMSIG(status)	当 WIFSIGNALED(status)为真时调用，返回引起终止的信号代号

2．waitpid 函数基础

在使用 wait 函数时，如果父进程的任何一个子进程返回，wait 函数则会返回，而 waitpid 函数则可以通过参数来指定需要等待的子进程。

waitpid 函数的参数 pid 用于对子进程进行相应的筛选，其详细说明如下。

- pid > 0：只等待进程 ID 为 pid 的子进程，不管其他已经有多少子进程运行结束退出了，只要指定的子进程还没有结束，waitpid 就一直等待下去。
- pid = –1：等待任何一个子进程退出，没有任何限制，此时 waitpid 等价于 wait。
- pid = 0：等待同一个进程组中的任何子进程，如果某一子进程已经加入了别的进程组，waitpid 则不会对它做任何处理。
- pid<–1：等待一个指定进程组中的任何子进程，这个进程组的 ID 等于 pid 的绝对值。

waitpid 函数的参数 options 用于进一步的控制 waitpid 函数的操作，其可以是 0，也可以是 WNOHANG 和 WUNTRACED 两个选项之一，或者是使用“|”符号连接的或操作，这两个关键字定义如下。

- WNOHANG：如果由 pid 指定的子进程并不是立即可用的，则 waitpid 函数不阻塞，此时返回“0”。
- WUNTRACED：如果某实现支持作业控制，而由 pid 指定的任意子进程已经处于暂停状态，并且未报告过，返回其状态。

对于 waitpid 函数而言，如果指定的进程或进程组不存在，或者参数 pid 指定的进程不是父进程所调用的子进程，都将出错。

总体而言，waitpid 函数提供了 wait 函数所没有的如下三个功能：

- 能够等待指定的一个进程结束；
- 能够不阻塞父进程获得子进程状态；
- 支持作业控制（读者可以自行查阅相关的资料）。

注意：综上所述，wait 系列函数的作用主要是完全销毁进程以释放内存及获得进程的退出状态。除了 wait 函数和 waitpid 函数之外，linux 内核还提供了 wait3、wait4 和 waitid 等函数，读者可以自行参考相应的手册。

3.【应用实例】——打印进程退出状态

例 9.7 是使用 wait 函数来打印 exit 函数返回状态（进程退出状态）的实例，其分别使用 abort 函数和除数为 0 制造了两个异常退出的子进程以打印其状态。

【例 9.7】 进程销毁的实例。

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
//一个输出 exit 状态的函数，参数是 status
void pr_exit(int status)
{
        if (WIFEXITED(status))
                printf("normal termination, exit status = %d\n",
                WEXITSTATUS(status));
        else if (WIFSIGNALED(status))
                printf("abnormal termination, signal number = %d%s\n",
                WTERMSIG(status),
#ifdef  WCOREDUMP
                WCOREDUMP(status) ? " (core file generated)" : "");
#else
                "");
```

```
#endif
        else if (WIFSTOPPED(status))
                printf("child stopped, signal number = %d\n",
                WSTOPSIG(status));
}
//主函数
int main(void)
{
        pid_t    pid;    //进程标识符
        int      status;

        if ((pid = fork()) < 0)        //创建子进程失败
                perror("fork error");
        }
        else if ( pid ==0 )   //进入子进程
        {
                exit(7);        //退出
        }
        if (wait(&status) != pid)   //等待创建的这个子进程结束，通过判断 pid
        {
                perror("wait error");
        }
        pr_exit(status);   //打印状态

        if ((pid = fork()) < 0)       //再次创建新进程
        {
                perror("fork error");
        }
        else if (pid == 0)              //另外一个子进程
        {
                abort();                //调用 abort 函???产生一个信号
        }
        if (wait(&status) != pid)
        {
                perror("wait error");
        }
        pr_exit(status);              //打印状态
        if ((pid = fork()) < 0)       //继续创建一个子进程
        {
                perror("fork error");
        }
        else if (pid == 0)
        {
           status /= 0;                 //通过除数为 0 来产生一个错误事件
```

```
        }
        if (wait(&status) != pid)
        {
            perror("wait error");
        }
        pr_exit(status);
        exit(0);
}
```

9.3 Linux 的线程基础

9.1 节中介绍的典型的 Linux 进程可以看作其只有一个控制线程，所以这个进程在同一时刻只能做一件事情。如果使用线程则可以使得这个进程在“同一时刻”能够同时完成多个任务，这种方式有如下优点。

- 提高应用程序响应速度。这对图形界面的程序尤其有意义，当一个操作耗时很长时，整个系统都会等待这个操作，此时程序不会响应键盘、鼠标、菜单的操作，而使用多线程技术，将耗时长的操作（time consuming）置于一个新的线程，可以避免这种尴尬的情况。
- 使多处理器系统更加有效。操作系统会保证当线程数不大于处理器数目时，不同的线程运行于不同的处理器上。
- 改善程序结构。一个既长又复杂的进程可以分为多个线程，成为几个独立或半独立的运行部分，这样的程序便于理解和修改。

9.3.1 线程的运行方式

线程包含了表示进程内执行环境所必需的信息，这些信息包括线程标识符（线程 ID）、一组寄存器值、栈、调度优先级和策略、信号屏蔽字、errno 变量及线程私有数据。进程的所有信息对该进程的所有线程都是共享的，包括可执行的程序文本、程序的全局内存和堆内存、栈和文件描述符。

在第 8 章介绍过，在传统的 Linux 或 UNIX 系统中，很多程序需要使用多个进程来完成工作，如许多关键的服务器应用程序有一个监听进程在不停地运行，等待客户请求到来。当一个请求到达时，这个监听进程创建（fork）一个新的进程为这个请求服务，因为对请求进行服务经常包括一些 I/O 操作，其可能阻塞进程。

在一个应用程序中使用多个进程有一些明显的缺点：

- 由于 fork 是一个开销很大的系统调用，所以创建这些进程增加了一些基本开销；
- 由于每个进程都有它自己的地址空间，它必须使用进程间通信的手段（如消息传递或共享内存）。
- 把这些进程分配到不同的机器或处理器上去运行，以及在进程之间传递信息、等待进程的完成、收集结果等都需要额外的开销。

在 Linux 创建新线程时，其会有一个控制线程用于控制新线程的相应工作，也称为主线程，如图 9.5 所示。

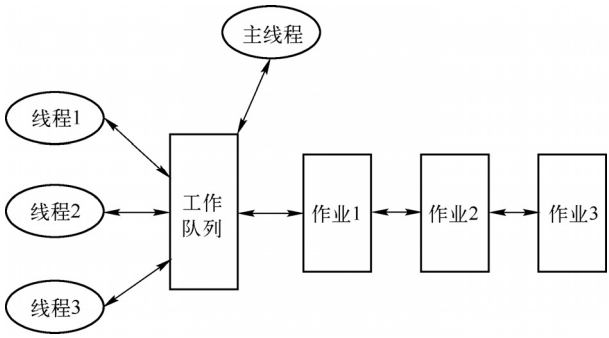


图 9.5　线程的控制

9.3.2　线程的标识符

和进程标识符类似，每一个线程都有一个在进程中唯一的线程标识符（线程 ID），其用一个数据类型 pthread_t 来表示，该数据类型在 Linux 中其实就是一个无符号长整型数据。

Linux 提供了两个函数用于对线程标识符进行操作，其标准调用格式说明如下：

```
#include <pthread.h>
pthread_t    pthread_self(void);
```

pthread_self 函数用于获得线程自身的线程标识符，其返回值是线程自身的线程标识符。

pthread_equal 函数用于比较两个线程标识符，其标准调用格式说明如下；

```
#include <pthread.h>
int pthread_equal(pthread_t tid1, pthread_t tid2);
```

函数的两个参数分别是需要比较的两个线程的标识符，如果相等则返回一个非“0”值，否则返回“0”。

注意：关于线程标识符的应用实例将在 9.4 节介绍。

9.3.3　用户态线程和核心态线程

用户态线程在管理上不需要内核的参与，所以通常又叫作“协作式多任务”，在进程内的这些线程统一由用户程序来切换，所以每一个线程在执行完任务后，调用任务切换功能，并向其发送信号，任务切换完成。线程对处理器资源的占用也切换到其他线程。通常，用户态线程在线程切换时要比内核态线程的速度快，不过在几个比较成功的内核态线程库中，线程切换的速度也相当快。虽然用户态线程有灵活和快速的特性，但是也存在一个严重的问题，即进程中的一个线程可能独占整个时间片，导致其他线程得不到处理器时间而无法运

行。例如，当一个线程由于磁盘 I/O 而阻塞时，其他线程同样不能运行。另外，用户态线程不能发挥多处理器机器（SMP）的性能。

内核态线程是由内核来管理的，在每一个时间片内，内核负责调度进程内的线程。由于内核参与了用户态进程的调度，所以涉及内核态与用户态上下文的切换。通常所说的内核态线程切换速度慢就是这个原因导致的。但是使用内核态线程的一个明显的好处是进程内的一个线程不会独占整个进程的处理器时间，这样，如果一个线程由于磁盘 I/O 而阻塞，其他线程仍可以利用处理器时间运行；使用核心态线程的另外一个好处是可以充分发挥 SMP 系统的性能，而且随着系统处理器数量的增多，应用程序运行的速度明显加快。

9.3.4 编译带线程的代码

用 gcc 编译多线程程序时，必须与 pthread 函数库连接。在终端下编译使用下列命令：

```
gcc –lpthread
```

上述编译命令把程序与 pthread 函数库相连。

9.4 在嵌入式 Linux 中进行线程操作

线程的操作包括线程的创建、退出和终止、阻碍和分离、取消和清理等，本节将详细介绍这些的操作。

9.4.1 调用 pthread_create 函数创建线程

1．pthread_create 函数基础

在 Linux 中，可以调用 pthread_create 函数创建一个新的线程，其标准调用格式说明如下；

```
#include <pthread.h>
int pthread_create (pthread_t *thread, pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);
```

函数的各个参数说明如下，如果创建成功则返回“0”，否则返回错误编号。

- 参数 thread：线程的标识符，需要说明的是这个参数并不是由用户确定的，用户只需要声明一个 pthread_t 类型的数据变量，并且将其传递给 pthread_create 函数，函数在创建新的线程的同时会将新线程的标识符放到这个变量中。
- 参数 attr：指定线程的属性，也可以将其设置为 NULL。
- 参数 start_routine：用于指定开始运行的函数，新创建的线程是从这个函数开始运行的，用户需要指定这个函数。
- 参数 arg：这是函数 start_routine 所需要的参数，是一个无类型指针，如果需要传递的参数不止一个，则需要将这些参数都放到一个结构中，然后将这个结构的地址传给 arg。

注意：pthread_create 函数在调用失败之后会返回对应的错误编码，每个线程都会提供 errno 的副本。

2.【应用实例】——创建线程并打印线程标识符

例 9.8 是一个调用 pthread_create 来创建线程并打印其线程标识符的应用实例。

【例 9.8】 创建线程并打印线程标识符。

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
pthread_t ntid;   //线程号
//打印标识符的函数
void    printids(const char *s)
{
            pid_t                pid;     //进程标识符
pthread_t    tid;     //线程标识符

pid = getpid();
tid = pthread_self();   //分别获得进程和线程编号
printf("%s pid %u tid %u (0x%x)\n", s, (unsigned int)pid,
   (unsigned int)tid, (unsigned int)tid);
          //打印线程和进程编号
}
//线程中开始运行的函数
void *thr_fn(void *arg)
{
printids("new thread: ");
return((void *)0);
}
//主函数
int    main(void)
{
int            err;
err = pthread_create(&ntid, NULL, thr_fn, NULL); //创建一个线程
if (err != 0)   //如果出错则打印错误标号
            {
              printf("can't create thread: %s\n", strerror(err));
}
      printids("main thread:");   //打印主线程号
sleep(1);
exit(0);
}
```

例 9.10 有如下两个特点。

- 需要处理主进程和新建的子进程之间的竞争。主线程需要休眠，如果主线程不休眠

其就可能退出，这样新线程还没运行整个进程就可能已经终止了，这种行为特征依赖于 Linux 的线程实现和调度算法。

- 新线程并不是通过 thread 参数来获得相应的进程标识符的，而是通过 pthread_self 函数获得的，这是因为虽然新的线程会把线程标识符存放在 thread 参数中，但是由于新线程的运行时间并不确定，所以可能出现该变量还没有初始化就已经被调用的情况，从而导致错误。

9.4.2 调用 pthread_exit 函数退出线程

进程可以调用 exit 系列函数退出当前进程，线程也可以通过如下三种方式退出，在不终止整个进程的情况下停止线程的控制流。

- 线程只是从启动例程中返回，返回值是线程的退出码。
- 线程可以被同一个进程中的其他线程终止。
- 线程调用 pthread_exit 函数退出。

1．pthread_exit 函数基础

Linux 内核提供的 pthread_exit 函数用于主动退出线程，其标准调用格式说明如下：

```
#include <pthread.h>
void pthread_exit(void *retval);
```

pthread_exit 函数没有返回值，其参数 retval 是线程的终止状态，其与 pthread_create 函数的 start_routine 参数类似，都是由用户先指定并传递给函数的一个参数，在 pthread_exit 函数完成之后可以调用这个参数来获得进程的退出状态。

2.【应用实例】——退出线程

例 9.9 是 pthread_exit 函数的应用实例。

【例 9.9】 线程退出实例。

```
#include <stddef.h>
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
void print_message(char*ptr);   //打印字符串
    //主函数
int main()
{
   pthread_t thread1, thread2;
   char *msg1="This is the frist thread!\n";
   char *msg2="This is the second thread!\n";   //两个字符串
   pthread_create(&thread1,NULL, (void *)(&print_message), (void *)msg1);
   pthread_create(&thread2,NULL, (void *)(&print_message), (void *)msg2);
   //创建两个线程并且休眠
   sleep(1);
   return 0;
}
```

```
void print_message(char *ptr)
{
   int retval;
   printf("Thread ID: %lx\n", pthread_self());   //打印进程标号
   printf("%s",ptr);
   pthread_exit(&retval);
}
```

9.4.3　调用 pthread_join 函数阻塞线程

1．pthread_join 函数基础

如果当一个线程已经执行完成，可以被其他的线程来阻塞挂起，然后等待指定的线程调用 pthread_exit，以从启动例程中返回或被取消，Linux 内核可以调用 pthread_join 函数来完成对线程的阻塞，其标准调用格式说明如下：

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **retval);
```

参数 thread 是一个线程标识符，用于指定要等待其终止的线程；参数 retval 用于存放其他线程的返回值，对于每一个可连接的线程，都必须调用该函数一次。任何线程都不能对相同的线程调用此函数，如果调用成功，函数返回 0；否则返回一个非零值。

2．【应用实例】——等待线程结束

例 9.10 是一个使用 phtread_join 函数来实现线程阻塞的实例，其使用 phread_join 函数来等待两个线程结束以保证完成工作之后主进程才会退出。

【例 9.10】 等待线程结束。

```
#include <stddef.h>
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
void print_msg(char *ptr);
//主函数
int main()
{
   pthread_t thread1, thread2;
   int i,j;
   void *retval;
   char *msg1="This is the frist thread\n";
   char *msg2="This is the second thread\n";   //存放两个字符串
   pthread_create(&thread1,NULL, (void *)(&print_msg), (void *)msg1);
   pthread_create(&thread2,NULL, (void *)(&print_msg), (void *)msg2); //创建两个线程
   pthread_join(thread1,&retval);
   pthread_join(thread2,&retval);
   return 0;
}
```

```
//打印信息函数，线程从这个函数开始运行
void   print_msg(char *ptr)
{
   int i;
   for(i=0;i<10;i++)
   printf("%s\n",ptr);   //连续输出 10 个字符串
}
```

9.4.4 调用 pthread_cancel 函数取消线程

1．pthread_cancel 函数基础

在 Linux 操作系统中，线程可以通过调用 pthread_cancel 函数来请求取消同一进程中的其他线程，该函数的标准调用格式说明如下：

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
```

函数的参数是需要取消线程的线程标识符，当操作成功时返回 0，否则会返回对应的错误编号。

2.【应用实例】——取消线程

例 9.11 是一个使用 pthread_cancel 函数取消线程的应用实例，程序在函数 thread_func()内，对参数判断成功后加入了一个死循环，会不断打印出参数的值，然后进行取消线程的操作；程序创建线程后，会不断打印线程收到的参数。主线程在等待 1s 后，调用 pthread_cancel()函数取消线程，之后主线程也运行结束，程序退出。

【例 9.11】 线程的取消实例。

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

void* thread_func(void *arg)                  // 线程函数
{
   int *val = arg;
   printf("Hi, I'm a thread!\n");
   if (NULL!=arg)
   {                                  // 如果参数不为空，打印参数内容
      while(1)
         printf("argument set: %d\n", *val);
   }
}

int main(void)
{
      pthread_t tid;                                 // 线程 ID
      int t_arg = 10;                               // 给线程传入的参数值
```

```
    if (pthread_create(&tid, NULL, thread_func, &t_arg))        // 创建线程
    perror("Fail to create thread");

    sleep(1);                                                   // 睡眠 1 秒，等待线程执行
    printf("Main thread!\n");
    pthread_cancel(tid);                                        // 取消线程

    return 0;
}
```

9.4.5　调用 pthread_cleanup 系列函数清理线程环境

1．pthread_cleanup 系列函数基础

在调用 pthread_cancel 取消了一个线程之后，需要调用相应的函数对进程退出之后的环境进行清理，这些函数称为线程清理处理程序（thread cleanup handler）。线程可以建立多个清理处理程序，这些函数的标准调用格式说明如下：

```
#include <pthread.h>
void pthread_cleanup_push(void (*routine)(void *),void *arg);
void pthread_cleanup_pop(int execute);
```

pthread_cleanup_push 函数将子程序 routine 连同它的参数 arg 一起压入当前线程的 cleanup 处理程序的堆栈；当当前线程调用 pthread_exit 是通过 pthread_cancel 终止执行时，堆栈中的处理程序将按照压栈时的相反的顺序依次调用。

而函数 pthread_cleanup_pop 从线程的 cleanup 处理程序堆栈中弹出最上面的一个处理程序并执行。

这两个函数都没有返回值。

需要注意的是，其实真正对线程执行清理工作的是在 pthread_cleanup_push 中作为参数传递的 routine 函数，其参数通过 arg 传递进去，其在线程执行如下动作时被调用：

- 调用 pthread_exit 函数时；
- 响应取消请求时；
- 用非 execute 参数调用 pthread_cleanup_pop 时。

如果 execute 参数被置为 0，清理函数将不会被调用，无论在哪种情况下，pthread_cleanup_pop 都将删除 pthread_cleanup_push 调用建立的清理处理程序。

2．【应用实例】——清理线程

例 9.12 是线程清理处理函数的应用实例。

【例 9.12】 线程的清理实例。

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <errno.h>
```

```
//清理函数，用于输出相应的参数
void *clean(void *arg)
{
    printf("cleanup :%s    \n",(char *)arg);
    return (void *)0;
}
//线程 1 的启动函数
void *thr_fn1(void *arg)
{
    printf("thread 1 start    \n");
    //格式化清理参数
    pthread_cleanup_push( (void*)clean,"thread 1 first handler");
    pthread_cleanup_push( (void*)clean,"thread 1 second hadler");
    printf("thread 1 push complete    \n");
    if(arg) //如果 arg 不为 0，返回
    {
        return((void *)1);
    }
    pthread_cleanup_pop(0);
    pthread_cleanup_pop(0);//调用清理函数
    return (void *)1;
}
//线程 2 的启动函数，参考 fn1
void *thr_fn2(void *arg)
{
    printf("thread 2 start    \n");
    pthread_cleanup_push( (void*)clean,"thread 2 first handler");
    pthread_cleanup_push( (void*)clean,"thread 2 second handler");
    printf("thread 2 push complete    \n");
    if(arg)
    {
        pthread_exit((void *)2);
    }
    pthread_cleanup_pop(0);
    pthread_cleanup_pop(0);
    pthread_exit((void *)2);
}
//主函数
int main(void)
{
    int err;
    pthread_t tid1,tid2;    //线程标识符
    void *tret;
    //创建线程 1
    err=pthread_create(&tid1,NULL,thr_fn1,(void *)1);
    if(err!=0)    //如果创建出错
```

```
    {
        perror("create pthread 1 error\n");
        return –1;
    }
    err=pthread_create(&tid2,NULL,thr_fn2,(void *)1);

    if(err!=0)
    {
        perror("create pthread 2 error \n");
        return –1;
    }
    err=pthread_join(tid1,&tret);    //阻塞线程 1 以等待结束
    if(err!=0)
    {
        perror("join thread1 error \n");
        return –1;
    }
    printf("thread 1 exit code %d    \n",(int)tret);

    err=pthread_join(tid2,&tret);    //阻塞线程 2
    if(err!=0)
    {
        perror("join thread2 error ");
        return –1;
    }

    printf("thread 2 exit code %d    \n",(int)tret);

    return 1;
}
```

9.4.6　调用 pthread_deatch 函数分离线程

在 Linux 中，线程一般有分离和非分离两种状态。在默认的情形下线程是非分离状态的，父线程维护子线程的某些信息并等待子线程结束。在没有显示调用 join 的情形下，子线程结束时，父线程维护的信息可能没有得到及时释放，如果父线程中大量创建非分离状态的子线程（在 Linux 系统中使用 pthread_create 函数），可能会出现堆栈空间不足的错误，其出错的返回值是 12。而对于分离线程来说，不会有其他线程等待它结束，它运行结束后，线程终止，资源及时释放。

1．pthread_deatch 函数基础

在 Linux 内核中，可以调用 pthread_deatch 函数来进行线程的分离，其标准调用格式说明如下：

```
#include <pthread.h>
int pthread_detach(pthread_t thread);
```

其参数是需要分离的线程标识符，如果函数调用成功则返回 0，如果调用失败则返回错误编号。

2.【应用实例】——清理线程

例 9.13 是使用 pthread_detach 函数清理线程的应用实例。

【例 9.13】 清理线程。

```
#include <errno.h>
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

//线程入口函数
void *func(void *arg)
{
   int i = *(int *)(arg);
   printf("I'am worker:%d\n",i);
}
int main(void)
{
   //线程 id
   pthread_t tid;
   int j;
   //创建大量线程
   int count = 10000;//多次循环
   for(j=0 ; j < count ; j++)
   {
      //线程参数
      int * p = &(j);
      //创建线程
      int ret= pthread_create(&tid, NULL, func, (void*)p);
      if(ret)//创建失败
      {
         printf("create thread error:%d\n",ret);
      }
      else//创建成功
      {
         //分离线程回收线程的 stack 占用的内存
         pthread_detach(tid);
      }
   }
   return 0;
}
```

9.4.7　线程和进程操作的总结和比较

线程的操作和进程的操作有很多相似之处，其比较如表 9.5 所示。

表 9.5　线程和进程操作函数比较

进 程 函 数	线 程 函 数	说　　明
fork	pthread_create	创建一个线程或进程
exit	pthread_exit	退出线程或进程
waitpid	pthread_join	处理进程或线程退出之后的状态
atexit	pthread_cleanup_push	退出控制流所调用的函数
getpid	pthread_self	获得标识符
abort	pthread_cancel	控制线程或进程退出

第10章

在嵌入式 Linux 中进行进程间和线程间通信

在 Linux 系统中，如果同时存在多个进程，则需要考虑这些异步进程之间的通信事件，包括数据和协同工作等；Linux 的信号和管道机制提供了一种处理异步事件的方法；同样，Linux 还需要考虑线程的同步，此时可以使用互斥锁和条件变量。本章涉及的知识说明如下：

- Linux 的信号机制及使用其完成进程间同步的方法；
- Linux 的管道机制及使用其完成进程间同步的方法；
- Linux 的命名管道机制及其使用方法；
- Linux 的线程的同步方法。

10.1 Linux 的进程通信和信号基础

10.1.1 Linux 的进程通信

在第 9 章中介绍的进程操作仅能通过 fork 等函数来传送一个已经打开的文件，或者通过对文件系统中文件的操作来实现多个进程中的数据交互，但是在比较复杂应用中用户通常需要使用多个相关的进程来执行有关操作；此时进程之间必须进行通信来共享资源和信息。Linux 内核提供了多种必要的机制来实现这种通信，这些机制通常叫作进程间通信或 IPC（Inter Process Communication）。

进程间通信通常需要实现如下目的。

- 数据传输：进程可能要发送数据到另一个进程。发送的数据量可以在 1 字节到几兆字节之间。
- 共享数据：多个进程想要操作共享的数据。一个进程修改了数据，其他共享该数据的进程应该立即看见这个变化。
- 通知事件：当一些事件发生时，进程也许会向另一个进程或一组进程发消息通知事件的发生。例如，进程终止时，它要通知它的父进程。接收者可能是被异步通知

的，这时候它的正常处理被中断。由此，接收者可以选择等待通知。

- 资源共享：一些要求相互操作的进程需要自行定义一些协议，这些协议针对它们要访问的特定的资源。这些协议是通过使用锁和同步机制来实现的，而锁和同步机制是建立在内核提供的基本功能之上的。
- 进程控制：有些进程，如 debugger 希望完全控制另一个进程（目标进程）的执行。控制进程希望能够拦截为目标进程设计的所有陷入和异常，并且能够及时知道目标进程状态的改变。

进程间的通信机制（IPC）其实就是多进程间相互通信、交换信息的方法。Linux 支持多种 IPC 机制，主要包括信号、管道和传统 UNIX 操作系统的 IPC 机制，本章将介绍信号。

10.1.2　Linux 中的信号机制和信号

信号机制是使用信号在进程之间相互传递消息的一种方法，其中信号全称为软中断信号，简称软中断。

软中断信号（signal，简称为信号）用来通知进程发生了异步事件，进程之间可以互相通过系统调用 kill 函数来发送软中断信号，而 Linux 内核也可以因为内部事件而给进程发送信号，通知进程发生了某个事件。

注意：信号只是用来通知某进程发生了什么事件，实质上并不给该进程传递任何数据。

每个信号都有一个名字，这些名字都以三个字符 SIG 开头，在头文件<signal.h>中，这些信号都被定义为正整数，称为信号编号。

注意：没有编号为 0 的信号，kill 函数对编号 0 有特殊的应用，在 POSIX.1 规范中，将此种信号编号值称为空信号。

Linux 内核支持 64 种不同的信号（具体内容将在 10.1.4 小节中进行详细介绍），这些信号中的大部分都有了预先定义好的意义，但是都支持自定义动作，并且还提供了类似 SIGUSR1 这样由应用程序来定义的信号。

1．信号源

在 Linux 中，一个信号的源存在如下几种可能。

（1）当用户按下某些终端按键之后引发终端产生的信号，如在程序运行中按下“CTRL+\”组合键将终止程序的运行。

（2）硬件产生的一个异常信号，如除数为 0、无效的内存引用等，这种异常信号通常会由硬件检测到并将其通知 Linux 内核，然后内核为该条件发生时正在运行的进程产生适当的信号。

（3）进程调用系统调用 kill 函数（kill(2)）可以给一个进程或进程组发送一个信号，需要注意的是此时发送和接收信号的进程/进程组的所有者必须相同。

（4）用户也可以调用 kill 命令（kill(1)）将信号发送给其他进程。

（5）当检测到某种软件条件已经发生，并应将其通知有关进程时也会产生一个信号，

如 SIGURG 信号就是在接收到一个通过网络传送的外部数据时产生的。

注意：kill(1)、kill(2)说的是在 Linux 系统帮助手册中的位置，其中“1”、“2”说明是在 man 命令后跟随的页数，具体到 kill 中来，kill(1)表示这是一个命令，而 kill(2)表示这是系统调用，其命令格式示例如下：

```
alloeat@ubuntu:~/chapter8Exam$ man 1 kill
//这是 kill(1)以及其输出
NAME
        kill - send a signal to a process

SYNOPSIS
        kill [ -signal | -s signal ] pid ...
        kill [ -L | -V, --version ]
        kill -l    [ signal ]
alloeat@ubuntu:~/chapter8Exam$ man 2 kill
//这是 kill(2)及其输出
NAME
        kill - send signal to a process
SYNOPSIS
        #include <sys/types.h>
        #include <signal.h>

        int kill(pid_t pid, int sig);
```

2．信号的处理方式

Linux 的每一个信号都有一个默认的动作，典型的默认动作是终止进程，当一个信号到来时收到这个信号的进程会根据信号的具体情况提供以下三种不同的处理方式。

- 类似中断的处理程序，对于需要处理的信号，进程可以指定处理函数，由该函数来处理。
- 忽略某个信号，对该信号不做任何处理，就像从未发生过一样。
- 对该信号的处理保留系统的默认值，这种默认操作大多数使得进程终止。进程通过系统调用 signal 函数来指定进程对某个信号的处理行为。

3．信号的缺陷

作为一种进程交互机制，信号有如下一些局限性。

- 信号的系统开销太大。
- 发送信号的进程要进行系统调用。
- 内核要中断接收信号的进程，而且要管理它的堆栈，同时还要调用处理程序，之后还要恢复执行被中断的进程。
- 信号的数量非常有限，因为只存在有限的不同的信号。
- 信号能传送的信息量十分有限，用户产生的信号不可能发送附加信息及各种参数。

所以，在实际使用中，信号机制常常用于进程之间的事件的通知，而不应用于复杂的

交互操作。

10.1.3　信号的工作方式

一个常见的信号应用实例是使用“CTRL+C”组合键来中断一个进程的运行，其操作部分说明如下，需要注意的是只有在前台运行的进程才能接收到“CTRL+C”组合键的输入。

- 用户输入命令，在 Shell 下启动一个前台进程。
- 用户按下 Ctrl+C 组合键，这个键盘输入产生一个硬件中断。
- 如果 CPU 当前正在执行这个进程的代码，则该进程的用户空间代码暂停执行，CPU 从用户态切换到内核态处理硬件中断。
- 终端驱动程序将 Ctrl+C 解释成一个 SIGINT 信号，记在该进程的 PCB 中（也可以说发送了一个 SIGINT 信号给该进程）。
- 当某个时刻要从内核返回到该进程的用户空间代码继续执行之前，首先处理 PCB 中记录的信号，发现有一个 SIGINT 信号待处理，而这个信号的默认处理动作是终止进程，所以直接终止进程而不再返回它的用户空间代码执行。

Linux 内核给一个进程发送软中断信号的方法，是在进程所在的进程表项的信号域设置对应于该信号的位（内核通过在进程的 struct task_struct 结构中的信号域中设置相应的位其来实现向一个进程发送信号）。

如果信号发送给一个正在睡眠的进程，那么要看该进程进入睡眠的优先级，如果进程睡眠在可被中断的优先级上，则唤醒进程；否则仅设置进程表中信号域相应的位，而不唤醒进程。这一点比较重要，因为进程检查是否收到信号的时机是：一个进程在即将从内核态返回到用户态时；或者，在一个进程要进入或离开一个适当的低调度优先级睡眠状态时。

内核处理一个进程收到的信号的时机是在一个进程从内核态返回用户态时。所以，当一个进程在内核态下运行时，软中断信号并不立即起作用，要等到将返回用户态时才处理。进程只有处理完信号才会返回用户态，进程在用户态下不会有未处理完的信号。

内核处理一个进程收到的软中断信号在该进程的上下文中，因此进程必须处于运行状态。前面介绍概念时讲过，处理信号有三种类型：进程接收到信号后退出；进程忽略该信号；进程收到信号后执行用户自定义的使用系统调用 signal()注册的函数。当进程接收到一个它忽略的信号时，进程丢弃该信号，就像从来没有收到该信号似的，而继续运行。如果进程收到一个要捕捉的信号，那么进程从内核态返回用户态时执行用户定义的函数。而且执行用户定义的函数的方法很巧妙，内核在用户栈上创建一个新的层，该层中将返回地址的值设置成用户定义的处理函数的地址，这样进程从内核返回，弹出栈顶时就返回到用户定义的函数处，从函数返回，再弹出栈顶时才返回原先进入内核的地方。这样做的原因是用户定义的处理函数不能且不允许在内核态下执行（如果用户定义的函数在内核态下运行，用户就可以获得任何权限）。

在信号的处理方法中有几点特别要引起注意。

- 在一些系统中，当一个进程处理完中断信号返回用户态之前，内核清除用户区中设定的对该信号的处理例程的地址，即下一次进程对该信号的处理方法又改为默认值，除非在下一次信号到来之前再次使用 signal 系统调用。

- 如果要捕捉的信号发生于进程正在一个系统调用中时，并且该进程睡眠在可中断的优先级上，这时该信号引起进程做一次 longjmp 函数调用，跳出睡眠状态，返回用户态并执行信号处理例程；当从信号处理例程返回时，进程就像从系统调用返回一样，但返回了一个错误代码，指出该次系统调用曾经被中断。
- 若进程睡眠在可中断的优先级上，则当它收到一个要忽略的信号时，该进程被唤醒，但不做 longjmp 函数调用，一般继续睡眠。但用户感觉不到进程曾经被唤醒，而是像没有产生过信号一样。
- 第四个要注意的地方，内核对子进程终止（SIGCLD）信号的处理方法与其他信号有所区别。
- 如果一个进程调用 signal()系统调用，并设置了 SIGCLD 的处理方法，并且该进程有子进程处于僵死状态，则内核将向该进程发送一个 SIGCLD 信号。

10.1.4 Linux 下的信号说明

1．Linux 的信号分类

参考 10.1.2 节介绍的 Linux 下的信号源，可以把 Linux 下的信号分为如下几大类。

- 与进程终止相关的信号。当进程退出或子进程终止时发出这类信号。
- 与进程例外事件相关的信号。如进程越界，或企图写一个只读的内存区域（如程序正文区），或执行一个特权指令及其他各种硬件错误。
- 与在系统调用期间遇到不可恢复条件相关的信号。如执行系统调用 exec 时，原有资源已经释放，而目前系统资源又已经耗尽。
- 与执行系统调用时遇到非预测错误条件相关的信号。如执行一个并不存在的系统调用。
- 在用户态下的进程发出的信号。如进程调用系统调用 kill 向其他进程发送信号。
- 与终端交互相关的信号。如用户关闭一个终端，或按下“break”键等情况。
- 跟踪进程执行的信号。

2．Linux 的预定义信号

在 Linux 中使用 kill –l 命令来查看系统支持的信号列表，或者输入“man 7 signal”查看更详细的说明，前者的输出列表说明如下：

```
alloeat@ubuntu:~/chapter8Exam$ kill -l
 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
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX–14 51) SIGRTMAX–13 52) SIGRTMAX–12
```

53) SIGRTMAX–11 54) SIGRTMAX–10 55) SIGRTMAX–9 56) SIGRTMAX–8　57) SIGRTMAX–7
58) SIGRTMAX–6　59) SIGRTMAX-5　60) SIGRTMAX–4 61) SIGRTMAX–3　62) SIGRTMAX–2
63) SIGRTMAX–1　64) SIGRTMAX

信号列表中编号 1～31 的信号为传统 Linux 内核所支持的信号，是不可靠信号（非实时的），编号 34～63 的信号是后来扩充的，称作可靠信号（实时信号）。不可靠信号和可靠信号的区别在于前者不支持排队，可能会造成信号的丢失，而后者不会。

此外，在不可靠信号中有四个比较特殊的信号，说明如下，其中前两个信号是不能被忽略的，而后两个信号是用户自定义的。

SIGSTOP（19）：这个信号将中断进程的执行。

SIGKILL（9）：这个信号将强制进程退出。

SIGUSR1（12）和 SIGUSR2（12）：用户自定义信号。

3．Linux 的预定义信号说明

编号 1～31 的预定义信号说明如下。

- SIGHUP：本信号在用户终端连接（正常或非正常）结束时发出，通常在终端的控制进程结束时，通知同一会话期（Session）内的各个作业，这时它们与控制终端不再关联。在登录 Linux 系统时，系统会自动分配给登录用户一个控制终端。在这个终端运行的所有程序，包括前台进程组和后台进程组，一般都属于同一个会话。当用户退出 Linux 登录时，前台进程组和后台有对终端输出的进程将会收到 SIGHUP 信号。这个信号的默认操作为终止进程，因此前台进程组和后台有终端输出的进程就会中止。此外，对于与终端脱离关系的守护进程来说，这个信号用于通知它重新读取配置文件。
- SIGINT：程序终止（或中断，interrupt）信号，在用户输入 INTR 字符（通常是按 Ctrl+C 组合键或 Delete 键）时发出，用于通知前台进程组终止进程。
- SIGQUIT：和 SIGINT 类似，但由 QUIT 字符（通常是 Ctrl+\组合键）来控制。进程在因收到 SIGQUIT 而退出时会产生 core 文件，在这个意义上类似于一个程序错误信号。
- SIGILL：执行了非法指令。通常是因为可执行文件本身出现错误，或者试图执行数据段，堆栈溢出时也有可能产生这个信号。
- SIGTRAP：由断点指令或其他陷阱（trap）指令产生，由调试器（debugger）使用，如跟踪陷阱信号。
- SIGABRT：调用 abort 函数时产生的信号，将使进程非正常结束。
- SIGBUS：非法地址，包括内存地址对齐（alignment）出错。例如，访问一个四个字长的整数，但其地址不是 4 的倍数。它与 SIGSEGV 的区别在于后者是由对合法存储地址的非法访问触发的（如访问不属于自己存储空间或只读存储空间）。
- SIGFPE：在发生致命的算术运算错误时发出。不仅包括浮点运算错误，还包括溢出及除数为 0 等其他所有的算术错误。
- SIGKILL：用来立即结束程序的运行。本信号不能被阻塞、处理和忽略。如果管理员发现某个进程终止不了，可尝试发送这个信号。

- SIGUSR1：留给用户使用，可由用户在应用程序中自行定义。
- SIGSEGV：试图访问未分配给登录用户的内存区，或试图向没有写权限的内存地址写数据。
- SIGUSR2：留给用户使用，可由用户在应用程序中自行定义。
- SIGPIPE：管道破裂信号，当对一个读进程已经运行结束的管道执行写操作时产生。这种情况通常发生在进程间通信时，如采用管道（FIFO）通信的两个进程，读管道还没有打开或意外终止就向管道写时，写进程会收到 SIGPIPE 信号。此外，如使用套接字（Socket）通信的两个进程，写进程在写 Socket 时，读进程已经终止。
- SIGALRM：时钟定时信号，计算的是实际的时间或时钟时间。由 alarm 函数设定的时间段终止时会产生该信号。
- SIGTERM：程序结束（terminate）信号，与 SIGKILL 不同的是，该信号可以被阻塞和处理。通常用来要求程序自己正常退出，Shell 命令“kill”默认产生这个信号。如果进程终止不了，才会尝试 SIGKILL。
- SIGSTKFLT：堆栈错误。
- SIGCHLD：子进程结束时，父进程会收到这个信号。如果父进程没有处理这个信号，也没有等待子进程，子进程虽然终止，但是还会在内核进程表中占有表项，这时的子进程称为僵尸进程，应该尽量避免这种情况。也是就说，父进程或者忽略 SIGCHILD 信号，或者捕捉它，或者等待它派生的子进程，或者父进程先终止，这时子进程的终止自动由 init 进程来接管。
- SIGCONT：让一个停止（stopped）的进程继续执行。此信号不能被阻塞，可以用一个信号处理程序来让程序在由停止状态变为继续执行时完成特定的工作。例如，重新显示提示符。
- SIGSTOP：停止（stopped）进程的执行。注意它和 terminate 及 interrupt 的区别：该进程还未结束, 只是暂停执行。此信号不能被阻塞、处理或忽略。

注意：SIGKILL 和 SIGSTOP 是两个不能被应用程序捕捉和忽略的信号，这是为了使系统管理员能在任何时候结束或停止某一特定进程的执行。

- SIGTSTP：停止进程的运行, 但该信号可以被处理和忽略。用户输入 SUSP 字符时（通常是 Ctrl+Z 组合键）发出这个信号。
- SIGTTIN：当后台作业要从用户终端读数据时，该作业中的所有进程会收到 SIGTTIN 信号，默认时这些进程会停止执行。
- SIGTTOU：类似于 SIGTTIN，但在写终端（或修改终端模式）时收到。
- SIGURG：套接字上出现紧急情况时产生此信号，如紧急数据。
- SIGXCPU：超过处理器源限制时产生的信号。这个限制可以由 getrlimit/setrlimit 来读取/改变。
- SIGXFSZ：当进程企图扩大文件以至于超过文件大小资源限制时产生此信号。
- SIGVTALRM：虚拟时钟信号，类似于 SIGALRM，但计算的是该进程占用的 CPU 时间。

- SIGPROF：类似于 SIGALRM/SIGVTALRM, 但包括该进程使用的 CPU 时间及系统调用时间。
- SIGWINCH：窗口大小改变时发出的信号。
- SIGIO：文件描述符准备就绪, 表示可以开始进行输入/输出操作。
- SIGPWR：电源失效信号（Power failure）。
- SIGSYS：非法的系统调用。

各个信号对进程的影响总结如表 10.1 所示。

表 10.1　信号对进程的影响总结

对进程的影响	信 号 列 表
不能恢复至默认动作	SIGILL、SIGTRAP
默认会导致进程流产	SIGABRT 、 SIGBUS 、 SIGFPE 、 SIGILL 、 SIGIOT 、 SIGQUIT 、 SIGSEGV 、 SIGTRAP、SIGXCPU、SIGXFSZ
默认会导致进程退出的信号	SIGALRM 、 SIGHUP 、 SIGINT 、 SIGKILL 、 SIGPIPE 、 SIGPOLL 、 SIGPROF 、 SIGSYS、SIGTERM、SIGUSR1、SIGUSR2、SIGVTALRM
默认会导致进程停止	SIGSTOP、SIGTSTP、SIGTTIN、SIGTTOU
默认进程忽略的信号	SIGCHLD、SIGPWR、SIGURG、SIGWINCH

10.1.5　调用 signal 系列函数来注册信号

要对一个信号进行处理，就需要给出此信号发生时系统所调用的处理函数，为一个特定的信号（除去无法捕捉的信号 SIGKIL 和 SIGSTOP）相应的处理函数。如果正在运行的程序的原代码里注册了针对某一特定信号的处理程序，无论当时程序执行到何处，一旦进程接收到该信号，相应的调用就会发生。

对于已经有自己的功能动作的信号而言，其注册就是用一个用户自己定义的功能动作去替换 Linux 内核预定义的功能动作的操作，如功能键“Ctrl+C”会中止当前进程的运行，当其被按下时，当前进程会接收到一个 SIGINT 信号，对应该信号的操作是终止当前进程；用户可以使用信号注册将 SIGINT 信号对应的操作定义为用户期望的操作，如在标准输出上输出一串字符串，在注册完成之后如果进程再次检测到 SIGINT 信号，则会进行输出字符串操作而不是终止退出。

在嵌入式 Linux 中提供了 singal 和 sigcation 函数，用于信号的注册。

1．singal 函数基础

要对一个信号进行处理，就需要给出此信号发生时系统所调用的处理函数，singal 函数可以为一个特定的信号（除去无法捕捉的 SIGKILL 和 SIGSTOP 信号）注册相应的处理函数。如果正在运行的程序源代码里注册了针对某一特定信号的处理程序，无论当时程序执行到何处，一旦进程接收到该信号，相应的调用就会发生，其标准调用格式说明如下；

```
#include <signal.h>
void (*signal (int signum, void (*handler) (int) ) ) (int);
```

signal 函数更为简洁的调用格式说明如下：

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

参数 signum 表示所注册函数针对的信号，其可以使用的值为 10.1.4 小节中介绍的信号的关键字或对应的编码（不推荐使用编码）；参数 handler 通常是指向调用函数的函数指针，这个函数是进程接收到信号之后的动作，这便是所谓的信号处理函数。

信号处理函数 handler 可能是用户自定义的一个函数，或是两个在 singal.h 头文件中进行了定义的值。

- SIG_IGN：忽略 signum 所指出信号。
- SIG_DFL：调用系统定义的默认信号处理。

注意：信号处理函数的参数是要处理的信号的信号值，并且不能为 SIGKILL 和 SIGSTOP 设置信号处理函数。

当 signal 函数调用成功之后，其返回信号以前的处理配置，如调用失败则返回 SIG_ERR（–1）。

当程序执行 signal 后，表示从这个时候开始由 signum 指定的信号对应的操作收到的将是 handler 所传递的函数。需要注意的是，并非程序执行到 signal 这一行就立即会对该信号做某种操作，因为信号的产生是无法预期的，程序设计人员根本无法预知该在哪一行捕捉突如其来的信号。用 signal 设置信号处理函数只是告诉系统用什么程序来处理这个信号。

2．sigaction 函数基础

如果觉得 signal 功能不够强大，可以使用功能更加强大的 sigaction 函数来完成相应的注册工作，其标准调用格式说明如下：

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act,struct sigaction *oldact);
```

其中，参数 signum 指定要处理的信号（除 SIGKILL 和 SIGSTOP 之外），act 和 oldact 都是指向信号动作结构的指针。结构的定义如下：

```
struct sigaction
{
 void (*sa_handler)(int);
 void(*sa_sigaction)(int,siginfo_t *,void *);
 sigset_t sa_mask;
 int sa_flags;
}
```

其中 sa_handler 用于指向信号处理函数的地址。参数 sa_sigaction 是指向函数的指针。它指向的函数有三个参数，其中第二个为 siginfo_t 结构体，定义如下：

```
struct siginfo_t
{
```

```
    int si_signo;           /*Signal number */
    int si_errno;           /*An errno value */
    int si_code;            /*Signal code */
    pid_t si_pid;           /*Sending process ID */
    uid_t si_uid;           /*Real user ID of sending process */
    int si_status;          /*Exit value or signal */
    clock_t si_utime;       /*User Time consumed */
    clock_t si_stime;       /*System time consumed */
    signal_t si_value;      /*Signal value */
    int si_int;             /* POSIX.1b signal */
    void *si_ptr;           /*POSIX.1b signal */
    void *si_addr;          /*Memory location that caused fault */
    int si_band;            /*Band event */
    int si_fd;              /*File descriptor */
}
```

sa_flags 指示信号处理函数的不同选项。具体可选参数见表 10.2。可以通过位运算的或运算（OR）串接不同的参数而实现所需的选项设置，将其赋值为 0 则选用所有的默认选项。

表 10.2　sa_flags 可选标志及对应设置

sa_flags	对 应 设 置
SA_NOCLDSTOP	用于指定信号 SIGCHLD，当子进程被中断时，不产生此信号，仅当子进程结束时产生该信号
SA_NOCLDWAIT	当信号为 SIGCHLD 时，此选项可以避免子进程的僵死
SA_NODEFER	当信号处理程序正在运行时，不阻塞对于信号处理函数自身的信号功能
SA_NOMASK	同 SA_NODEFER
SA_ONESHOT	当用户注册的信号处理函数被调用过一次之后，该信号的处理程序恢复为默认的处理函数
SA_RESETHAND	同 SA_ONESHOT
SA_RESTART	使本来不能进行自动重新运行的系统调用自动重新启动
SA_SIGINFO	表明信号处理函数是由 sa_sigaction 指定的，而不是由 sa_handler 指定的。它将显示更多处理函数的信息

3.【应用实例】——修改 SIGINT 信号功能

例 10.1 是一个使用自定义的字符串输出函数来取代 SIGINT 信号的中止当前进程运行响应操作的应用实例。程序使用 signalDeal 函数替代组合键“CTRL+C”对应的终止当前进程的操作，其动作是使用 printf 函数在终端上输出当前进程接收到的信号编号。

【例 10.1】 修改 SIGINT 信号功能。

```
#include <signal.h>
#include <stdio.h>
//这是信号处理函数
void signalDeal(int iSignNum) //参数是信号量的编号
{
    printf("The signal NO. is:%d\n",iSignNum);   //输出信号量编号
```

```
    return;
}
//主函数
int main(void)
{
    signal(SIGINT,signalDeal); //申明变量，使用 signalDeal 函数替代退出
    while(1)    //死循环
    {
        sleep(1);
    }
    return 0;
}
```

10.2 Linux 中信号的基础操作

信号的基础操作包括向进程发送信号，使用一个定时信号及调用相应的信号使进程退出，Linux 系统提供了相应的函数来实现对应的操作。

10.2.1 使用 kill 函数和 raise 函数发送信号

在例 10.2 中，为了测试 SIGUSR1 和 SIGUSR2，使用了 kill 函数从外部给进程发送相应的信号，在实际应用中，Linux 中的用户进程可以调用 kill 函数和 raise 函数来完成相应的信号发送操作，前者用于给其他进程发送信号而后者用于给进程自身发送信号。

1．kill 函数基础

kill 函数（这个地方使用的是 kill(2)，其对应的帮助手册是 man 2 kill）将信号发送给进程或进程组，其标准调用格式说明如下：

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

参数 pid 表示 kill 函数发送信号对象的进程或进程组号，其取值说明如表 10.3 所示。参数 sig 为需要发送的信号编码，当该函数调用成功后其返回值为 0，如果调用失败其返回值为–1。

表 10.3　kill 函数的 pid 参数

Pid	含　义
pid>0	将信号发送给进程号为 pid 的进程
pid=0	将信号发送给和目前进程相同进程组的所有进程
pid<0&& pid!=–1	向进程组 ID 为 pid 绝对值的进程组中的所有进程发送信号
pid=–1	除发送进程自身外，向所有进程 ID 大于 1 的进程发送信号

需要注意的是，进程使用 kill 函数向另外一个进程发送信号需要相应的权限，超级用户则可以将信号发送给任意进程，非超级用户则需要发送者和接受者的实际或有效用户

ID 相同。

注意：除 SIGCONT 信号外，任何进程都可以将该信号发送给属于同一会话的任何其他进程。

另外一个需要注意的方面是，sig 参数对应的是信号编码值，当其为 0 时（即空信号），实际不发送任何信号，但照常进行错误检查，因此，可检查目标进程是否存在，以及当前进程是否具有向目标发送信号的权限（root 权限的进程可以向任何进程发送信号，非 root 权限的进程只能向属于同一个 session（会话）或同一个用户的进程发送信号）。

2.【应用实例】——传递 SIGABRT 信号

例 10.2 是一个父进程利用 kill 函数向其子进程传送一个 SIGABRT 信号，使子进程非正常结束的实例。

【例 10.2】 传递 SIGABRT 信号。

```
#include<unistd.h>
#include<signal.h>
#include<sys/types.h>
#include<sys/wait.h>
#include<stdio.h>
#include<errno.h>
int main(void)
{
   pid_t pid;
   int status;
   if(!(pid= fork()))     //创建一个子进程
   {
      printf("Hi I am child process!\n");
      sleep(10);              //让子进程睡眠，看父进程的行为
      printf("Hi I am child process, again!\n");
    return;
   }
   else
   {
      printf("send signal to child process (%d) \n",pid);
      sleep(1);
    if(kill(pid ,SIGABRT) == –1)     //发送 SIGABRT 信号
    {
        perror("kill failed!\n");
    }
    wait(&status);
   if(WIFSIGNALED(status))
   {
       printf("child process receive signal %d\n",WTERMSIG(status));
   }
   }
```

```
    return 0;
}
```

3．raise 函数基础

如果当前进程需要对自身发送一个信号，可以使用 raise 函数，其标准调用格式说明如下：

```
#include <signal.h>
int raise(int sig);
```

sig 参数是需要向自身发送的信号的信号编码，如果函数调用成功则返回 0，如果出错则返回–1。

注意：raise 函数其实等同于调用 kill(getpid(),signo)。

4.【应用实例】——使用信号来处理输入字符串

例 10.3 是一个 raise 函数应用实例，通过终端输入某几个特定的字符串 int、stop、continue 和 quit 时，程序不再将字符串处理并回显，而是向其自身发送信号，再调用相应的信号处理函数。

【例 10.3】 使用信号来处理输入字符串。

```
#include <stdio.h>
#include <signal.h>
#include <errno.h>
#include <stdlib.h>
void inthandler(int signum);
void continuehandler(int signum);
void terminatehandler(int signum);
//主函数
int main(void)
{
    char buffer[100];
    if((int)(signal(SIGINT,&inthandler)) == -1)   //注册 SIGINT 信号
    {
        perror("Couldn't register signal hanlder for SIGINT!\n");
        exit(1);
    }
    if((int)(signal(SIGTSTP, &inthandler)) == -1)   //注册 SIGTSTP 信号
    {
        perror("Couldn't register signal hanlder for SIGTSTP!\n");
        exit(2);
    }
    if((int)(signal(SIGCONT, &continuehandler)) == -1)   //注册 SIGCONT 信号
    {
        perror("Couldn't register signal hanlder for SIGCONT!\n");
        exit(3);
```

```
    }
    if((int)(signal(SIGTERM, &terminatehandler)) ==-1)   //注册 SIGTERM 信号
    {
        perror("Couldn't register signal hanlder for SIGINT!\n");
        exit(4);
    }
    printf("Pid of This Process : %d \n",getpid());
    while(1)
    {
     printf("Please input:\n");
     fgets(buffer, sizeof(buffer),stdin);
     if(strcmp(buffer,"int\n")==0)       //使用 strcmp 函数获取输入，发送信号
     {
        raise(SIGINT);
     }
     else if(strcmp(buffer,"stop\n")==0)
     {
        raise(SIGTSTP);
     }
     else if(strcmp(buffer,"continue\n")==0)
     {
         raise(SIGCONT);
     }
     else if(strcmp(buffer,"quit\n")==0)
     {
         raise(SIGTERM);
     }
     else
     {
         printf("Your input is: %s \n",buffer);
     }
    }
    exit(0);
}
//终止信号处理函数
void inthandler(int signum)
{
     printf("catch signal %d \n",signum);
}
//继续信号处理函数
void continuehandler(int signum)
{
     printf("Continue code.\n");
}
//退出信号处理函数
void terminatehandler(int signum)
```

```
{
    printf("signal SIGTERM \n");
    exit(0);
}
```

10.2.2 使用 alarm 进行信号的定时操作

在 Linux 的应用程序中，常常需要定时一段时间之后使线程去执行一个动作，此时可以使用 SIGALRM 信号量，Linux 内核同样提供了相应的操作函数 alarm。

1．alarm 函数基础

alarm 函数标准调用格式说明如下：

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

参数 seconds 指定了下一次发送信号的时间，即在当期时间的 seconds 秒后向进程本身发送 SIGALRM 信号，又称为闹钟时间。进程调用 alarm 后，任何以前的 alarm 调用都将无效。如果参数 seconds 为 0，那么进程内将不再包含任何闹钟时间。

如果调用 alarm 之前进程中已经设置了闹钟时间，则返回上一个闹钟时间的剩余时间，否则返回 0。

2.【应用实例】——定时读写标准输入输出

alarm 函数可用于对可能阻塞的操作设置时间的上限值。例如，应用中有一个读低速设备的可能阻塞的操作在超过一定时间后就停止执行这个读操作。如例子 10.4 所示，其从标准输入上读一行，然后将其写到标准输出上，如果超过 1 秒则停止。

【例 10.4】 定时读写标准输入输出。

```
#include <signal.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#define MAXLINE 4096     //行最大容量
static void sigalrm(int);
int main(void)
{
        int                 n;
        char        line[MAXLINE];

        if (signal(SIGALRM, sigalrm) == SIG_ERR) //使用 sigalrm 替代 SIGALRM 信号
        {
            perror("signal(SIGALRM) error");   //出错处理
        }
        alarm(10);   //定时 10 个 tip
        if ((n = read(STDIN_FILENO, line, MAXLINE)) < 0) //从输入读
```

```
        {
            perror("read error");
        }
        alarm(0);   //取消定时

        write(STDOUT_FILENO, line, n);
        exit(0);
}
static void   sigalrm(int signo)
{
  //什么都不做，仅仅打断读信号
}
```

10.2.3　使用 setitimer 函数进行精确定时

如果希望使用更加精确的定时操作，可以使用 setitimer 函数。

1．setitimer 函数基础

setitimer 函数的标准调用格式说明如下：

```
#include <sys/time.h>
int setitimer(int which, const struct itimerval *new_value,struct itimerval *old_value);
```

参数 which 用于指定定时器类型，其支持 3 种类型的定时器，如表 10.4 所示。

表 10.4　setitimer 的 witch 参数说明

which 取值	定时器类型	发 生 信 号
ITIMER_REAL	设定绝对时间，即根据系统的时间	SIGALRM
ITIMER_VIRTUAL	设定程序执行时间，只有在用户模式下才可跟踪时间	SIGVTALRM
ITIMER_PROF	从用户进程开始后开始计时	SIGPROF

参数 value 和 old_value 为指向时间参数的结构体指针，itimerval 结构原型如下：

```
struct itimerval
{
    struct timeval it_interval;   /*计时器重启动的间歇值*/
    struct timeval it_value;      /*计时器安装后首先启动的初始值*/
};
```

成员 it_interval 和 it_value 又是 timeval 类型的结构体：

```
struct timeval
{
    long tv_sec;                  /*时间的秒数部分*/
    long tv_usec;                 /*时间的微妙(1/1000000)部分*/
};
```

setitimer 函数将 value 指向的结构体设为计时器的当前值，如果 old_value 不是 NULL，将返回计时器原有值，其若调用成功则返回 0，若出错则返回–1。

2.【应用实例】——连续打印系统时间和日期

例 10.5 是一个 setitimer 函数的应用实例，该实例每隔 1 秒便会调用信号处理函数 ElsfTimer，打印出当前系统的时间和日期，在 ElsfTimer 函数中，使用了另外两个系统调用 gettimeofday 函数和 localtime 函数。

【例 10.5】 连续打印系统时间和日期。

```
#include <signal.h>
#include <time.h>
#include <sys/time.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

static void DealTime(int signo)       //信号处理函数
{
    struct timeval tp;
    struct tm *tm;
    gettimeofday(&tp,NULL);     //获得系统当前时间（秒和微秒）
    tm=localtime(&tp.tv_sec);     //获得当地目前时间和日期
    printf(" sec = %ld \t",tp.tv_sec);       //打印从 UNIX 纪元开始到现在的秒数
    printf(" usec = %ld \n",tp.tv_usec);    //打印微秒
    printf("%d-%d-%d%d:%d:%d\n",tm->tm_year+  1900,tm->tm_mon+1,tm->tm_mday,tm->tm_hour,
tm->tm_min,tm->tm_sec);   /*打印当地目前时间和日期*/
}

static void InitTime(int tv_sec,int tv_usec)
{
    struct itimerval value;              //定义时间参数结构体 value
    signal(SIGALRM, DealTime);   //注册信号 SIGALRM  和信号处理函数
    value.it_value.tv_sec = tv_sec;     //秒
    value.it_value.tv_usec = tv_usec; //微秒
    value.it_interval.tv_sec = tv_sec;
    value.it_interval.tv_usec = tv_usec;
    setitimer(ITIMER_REAL, &value, NULL);
    //setitimer  发送信号，定时类型为 ITIMER_REAL
}

int main(void)
{
    InitTime(1,0);        //每隔 1 秒打印一次
    while(1)
```

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

10.2.4　使用 abort 发送进程退出信号

如果进程在执行过程中出现了异常，可以调用 abort 函数向进程发送 SIGABRT 信号使其退出，abort 函数的标准调用格式说明如下：

```
#include <stdlib.h>
void abort(void);
```

abort 函数用于将 SIGABRT（退出）信号发送给调用的进程，其没有返回值，例 10.6 是一个 abort 函数的应用实例。

【例 10.6】 abort 函数应用实例。

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
int main(void)
{
    abort();       //退出
    exit(EXIT_SUCCESS);
}
```

10.3　Linux 的管道和进程通信

信号机制能够交互的信息量很小，通常来说只是起到“通知”事件发生的作用，如果进程要向另外一个进程发送数据信息，此时可以使用管道。

管道（Pipe）也称为匿名管道，是 Linux 下最常见的进程间通信方式之一，它是在两个进程之间实现一个数据流通的通道。管道是一种很经典的进程之间的通信方式，其具有两个缺点。

- 部分系统下的管道是半双工的，数据只能向一个方向流动（这一项特征应该根据相应的 Linux 内核来确认）。
- 管道通常来说只能在有相同祖先的进程间使用，如父子进程、兄弟进程等。

10.3.1　管道基础

管道是 Linux/UNIX 系统中比较原始的进程间通信形式，它实现数据以一种数据流的方式在进程间流动。其在系统中相当于文件系统上的一个文件，来缓存所要传输的数据。在某些特性上又不同于文件。例如，当数据读出后，管道中就没有数据了，但文件没有这个特性。

管道是 Linux 中最古老的进程通信机制，其应用非常广泛，和信号类似，其也提供了相应的操作符“|”以供用户在 Shell 中使用。

操作符“|”将其前后两个命令连接到一起，前一个命令的输出成为后一个命令的输入，支持使用多个“|”连接多个命令，其标准调用格式说明如下，命令 A 输出即为命令 B 的输入，假如命令 A 为“ls”命令，则这个输出即为当前目录下的文件列表。

```
命令 A|命令 B|命令 C……|命令 N
```

在第 10.1.4 小节中介绍了使用“kill–l”命令来查看当前系统中所支持的信号类型列表，如果想在该信号列表中直接查找含有字串“SIGRTMAX”的信号，可以使用管道操作符“|”来连接“kill–l”和“grep”命令，此时 Shell 创建了 kill–l 和 grep 两个进程和这两个进程间的管道，将“kill–l”命令的输出作为“grep”命令的输入，也就是说，在信号列表中查找包括“SIGRTMAX”的信号，其输出如下：

```
alloeat@ubuntu:~/chapter9Exam$ kill –l
 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
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX–14 51) SIGRTMAX–13 52) SIGRTMAX–12
53) SIGRTMAX–11 54) SIGRTMAX–10 55) SIGRTMAX–9  56) SIGRTMAX–8  57) SIGRTMAX–7
58) SIGRTMAX–6  59) SIGRTMAX–5  60) SIGRTMAX–4  61) SIGRTMAX–3  62) SIGRTMAX–2
63) SIGRTMAX–1  64) SIGRTMAX
alloeat@ubuntu:~/chapter9Exam$ kill–l | grep SIGRTMAX
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX–14 51) SIGRTMAX–13 52) SIGRTMAX–12
53) SIGRTMAX–11 54) SIGRTMAX–10 55) SIGRTMAX–9  56) SIGRTMAX–8  57) SIGRTMAX–7
58) SIGRTMAX–6  59) SIGRTMAX–5  60) SIGRTMAX–4  61) SIGRTMAX–3  62) SIGRTMAX–2
63) SIGRTMAX–1  64) SIGRTMAX
```

10.3.2 管道的实现方法

当一个进程创建一个管道时，Linux 系统内核为使用管道准备了两个文件描述符，一个用于管道的输入，也就是在管道中写入数据；另一个用于管道的输出，也就是从管道中读出数据。然后进程对这两个文件描述符调用正常的系统调用，内核利用这种抽象机制实现了管道这一特殊操作，如图 10.1 所示。

如果一个管道只与一个进程相联系，只实现进程自身内部的通信，这个管道是毫无意义的；通常情况下，一个创建管道的进程接着就会创建其子进程，由于父子进程可以共享打开文件，子进程会从父进程那里继承到读写管道的文件描述符，这样，父子进程间的通信管

道就建立起来了，如图 10.2 所示。

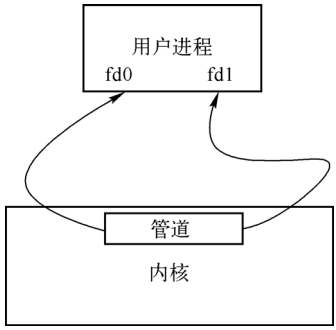


图 10.1　管道的结构

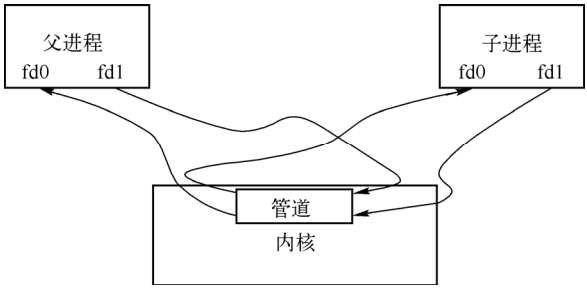


图 10.2　父进程和子进程之间的管道

最后需要要确定数据的传输方向，是从子进程传送到父进程，还是从父进程送到子进程。这一点确定之后，父子进程分别关闭与之无关的那个描述符。例如，数据从子进程传送到父进程，则子进程关闭读管道的描述符，父进程关闭写管道的描述符。这样就建立了从子进程到父进程的通信管道，如图 10.3 所示。

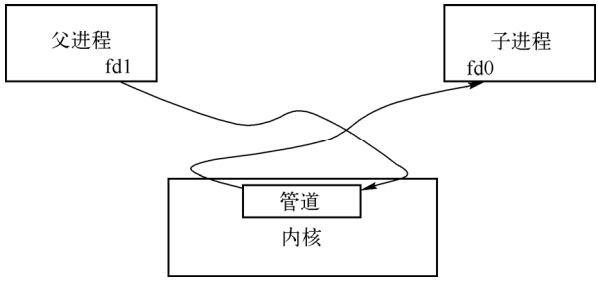


图 10.3　从父进程到子进程的管道

10.3.3　管道读写操作规则

在建立了一个管道之后即通过调用相应的文件操作函数（如 read、write 等）来读写管道以完成信息的传递。

需要注意的是，由于管道的一端已经关闭，在进行相应的操作时需要注意以下三个要点。

- 如果从一个写描述符关闭的管道中读数据，当读完所有的数据后，read 函数返回 0，表明已到达文件末尾。严格来说，只有当没有数据继续写入后，才可以说到达了文件末尾。所以应该分清到底是暂时没有数据输入，还是已经到达文件末尾，如果是前者，读进程应该等待。多进程写、单进程读的情况则更加复杂。
- 如果向一个读描述符关闭的管道中写数据，就会产生 SIGPIPE 信号。不管是忽略这个信号，还是处理它，write 函数都会返回–1。
- 常数 PIPE_BUF 规定了内核中管道缓冲的大小，所以在写管道时要注意这一点。一次向管道中写入 PIPE_BUF 或更少的字符，不会和其他进程写入的内容交错；反之，当存在多个写管道的进程时，向其中写入超过 PIPE_BUF 个字符时，就会产生交错现象。

注意：在 Linux 系统中，可以使用 pathconf 或 fpathconf 函数来确定 PIPE_BUF 的大小，在 Ubuntu 中这个值是 4096。

10.3.4 管道的特点

Linux 的管道具有以下特点。

- 管道没有名字，所以也称为匿名管道。
- 管道是半双工的，数据只能向一个方向流动；双方向通信时，需要建立起两个管道。
- 只用于父子进程或兄弟进程之间（具有亲缘关系的进程）。
- 单独构成一种独立的文件系统。管道对于管道两端的进程而言，就是一个文件。但它不是普通的文件，它不属于某种文件系统，而是自立门户，单独构成一种文件系统，并且只存在于内存中。
- 数据的读出和写入：一个进程向管道中写的内容被管道另一端的进程读出。写入的内容每次都添加在管道缓冲区的末尾，并且每次都从缓冲区的头部读出数据。
- 管道的缓冲区是有限的（管道只存在于内存中，在管道创建时，为缓冲区分配一个页面大小）。
- 管道所传送的是无格式字节流，这就要求管道的读出方和写入方必须事先约定好数据的格式，如多少字节算作一个消息（或命令、或记录）等。

注意：在实际应用中，由于管道中的数据是无格式的，所以必须采用一个事先设计好的数据格式。

10.4 在 Linux 中进行管道操作

Linux 中的管道操作包括管道的创建和管道的读写。

10.4.1　使用 pipe 函数来创建管道

Linux 内核提供的函数 pipe 用于创建一个管道。

1．pipe 函数基础

pipe 函数标准调用格式说明如下：

```
#include <unistd.h>
int pipe(int pipefd[2]);
```

函数的参数 pipefd[2]是一个长度为 2 的文件描述符数组，其中 pipefd[0]是读出端的文件描述符，fd[1]是写入端的文件描述符。也就是说，pipefd[0]只能为读打开，而 pipefd[1]是为写操作打开的。当函数调用成功之后，则自动维护了一个从 fd[1]到 fd[0]的数据通道。

函数如果调用成功，则返回 0；如果调用失败，则返回–1。

2.【应用实例】——创建管道

例 10.7 是一个调用 pipe 函数创建管道的应用实例，这是进程调用 pipe 函数创建一个管道并打印管道文件描述符的一个应用实例。

【例 **10.7**】 创建管道的实例。

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
int main(void)
{
    int fd[2];                          // 文件描述符
    char str[256];
    if((pipe(fd)) < 0)                  //创建管道
    {
        perror("create the pipe failed!\n");
        exit(0);
    }
    write(fd[1], "create the pipe successfully!\n", 31 );    //向管道写入端写入数据
    read(fd[0], str, sizeof(str) );     //从管道读出端读出数据
    printf ("%s", str ); //输出字符串
    printf ("pipe file ID   are%d,%d \n", fd[0], fd[1]) ;   //打印管道描述符
    close(fd[0]);                       // 关闭管道的读出端文件描述符
    close(fd[1]);                       // 关闭管道的写入端文件描述符
    return 0;
}
```

10.4.2 【应用实例】——父子进程使用管道通信

在实际使用中，进程自身创建一个管道是没有意义的。通常来说，进程先会创建一个子进程，然后创建一个管道，将需要传送的数据通过管道传送给子进程。例 10.2 给出了一个在父子进程间传送字符串数据的实例。

【例 10.8】 父子进程使用管道通信。

```
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <errno.h>
int main(void)
{
   int n, fd[2];
   pid_t pid;
   char buffer[BUFSIZ+1];   //8192
   if(pipe(fd)<0)        //创建一个管道，两个文件描述符在 fd 数组中
   {
      perror("pipe failed!\n ");
      exit(0);
   }
   if((pid=fork())<0)   //创建一个子进程
   {
      printf("fork failed!\n ");
      exit(0);
   }
   else if (pid>0)     //父进程
   {
      close(fd[0]);
      write(fd[1],"This is a pipe test!\n",22);   //向管道写入数据，注意回车换行符
   }
   else   //子进程
   {
      close(fd[1]); //关闭
      n = read(fd[0],buffer,BUFSIZ);   //从通道中读出数据
      write(STDOUT_FILENO,buffer,n);   //将数据写到标准输出设备
   }
   exit(0);
}
```

10.4.3 【应用实例】——兄弟进程使用管道通信

管道通信只能在有亲缘关系的进程之间进行，这些具有亲缘关系的进程除了父子进程之外，还包括兄弟进程。例 10.9 给出了一个在兄弟进程之间使用管道进行数据通信的实例。

【例 10.9】 兄弟进程使用管道通信。

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

```
#include <sys/types.h>
#include <limits.h>
#include <string.h>
#include <errno.h>
#define BUFSIZE 4096   //定义一个最大的读写空间
int main(void)
{
    int fd[2];
    char buf[BUFSIZE] = "hello!I am your    brother!\n";          // 缓冲区
    pid_t pid;
    int len;
    if ( (pipe(fd)) < 0 )                  //创建管道
     {
        perror("pipe failed\n");
     }
    if ( (pid = fork()) < 0 )              //创建第一个子进程
     {
        perror("fork failed\n");
     }
    else if ( pid == 0 )                    //子进程
    {
        close ( fd[0] );                        //关闭不使用的文件描述符
        write(fd[1], buf, strlen(buf));     //发送字符串
        exit(0);
     }
    if ( (pid = fork()) < 0 )                   //创建第二个子进程
     {
        perror("fork failed\n");
     }
    else if ( pid > 0 )                     //父进程
    {
        close ( fd[0] );
        close ( fd[1] );
        exit ( 0 );
     }
    else                                    //第二个子进程中
    {
        close ( fd[1] );                            //关闭管道文件描述符
        len = read (fd[0], buf, BUFSIZE);   //读取消息
        write(STDOUT_FILENO, buf, len); //将消息输出到标准输出
        exit(0);
     }
    return 0;
}
```

10.4.4 管道的高级操作

pipe 函数和 fork 函数通常是配合起来使用的，Linux 内核同样提供了“合二为一”的函数 popen 用于对应的操作，其标准调用格式说明如下：

```
#include <stdio.h>
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

popen 函数和 pclose 函数必须配合使用，类似 fopen 和 fclose 函数的组合，其详细说明如下。

函数 popen 用于创建管道。它内部调用 fork 和 exec 函数执行命令行 command，返回一个 FILE 结构的指针，即用于访问管道的指针。

popen 中的参数 const char *command 就是一个命令行。所有的 Shell 命令行参数和选项都可以使用，如其可以使用如下的命令行调用：

```
popen("ls *.*","r");
popen("sort > /tmp/foo","w");
popen("sotr | uniq | more", "w");
```

popen 中的参数 const char *type 指出管道的类型。如果管道是以类型“r”打开的，那么这个管道的输入端连接到了命令行 command 的标准输出端。此时，命令行的输出可以从管道中读入。反之，如果管道是以类型“w”打开的，那么这个管道的输出端连接到了命令行的标准输入端。此时，向管道中写入的数据就成为命令行的输入数据。可以看到，type 的作用与 fopen 和 fclose 中的相同，可以取“r”或“w”，表示管道可读或可写，但不可以既可读又可写。在 Linux 系统下，规定管道的打开方式取决于 type 的第一个字符，如 type 为“rw”，那么管道就是以“r”方式，即可读方式打开的。

函数 pclose 是用来关闭管道的。它关闭标准输入输出流，等待命令行执行完毕，然后返回结束时的状态。如果 Shell 不能执行这个命令行，结束时的状态就如同在 Shell 中执行了 exit 函数，例 10.10 是 popen 函数的应用实例。

【例 10.10】 管道的高级操作函数实例。

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <errno.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
   char buffer[BUFSIZ+1];
   FILE *fpin, *fpout;     //流变量
   if(argc<=1)
```

```
    {
        printf("usage: %s <pathname>\n",argv[0]);   //如果参数出错
        exit(1);
    }
    if((fpin=fopen(argv[1],"r")) == NULL) //以读方式打开指定文件
    {
        printf("Can't open %s \n", argv[1]);
        exit(1);
    }
    if((fpout=popen("./upcase","w"))==NULL) //以写方式打开，注意命令行方式
    {
        perror("popen error \n");
        exit(1);
    }
    while(fgets(buffer,BUFSIZ,fpin)!=NULL)   //写入数据
    {
        if(fputs(buffer,fpout) ==EOF)     //如果到达文件结尾
        {
            perror("fputs error to pipe. \n"); //指明错误
            exit(1);
        }
    }
    if(ferror(fpin))   //判断流是否错误
    {
        perror("fgets error. \n");   //错误
        exit(1);
    }
    if(pclose(fpout) ==-1)   //关闭
    {
        printf("pclose error.\n");
        exit(1);
    }
    exit(0);
}
```

10.5　Linux 的命名管道基础

从前面的章节可以知道，Linux 的管道只能在有亲缘关系的进制之间实现通信，所以如果两个“毫无关系”的进程需要进行数据交换，就不能使用管道。但是 Linux 内核提供了另一种“管道”可以实现这种功能，其被称为命名管道（named pipe）或先进先出队列（FIFO）。

命名管道不同于管道之处在于它提供一个路径名与之关联，以命名管道的文件形式存在于文件系统中。这样，即使是与命名管道的创建进程不存在亲缘关系的进程，只要可以访

问该路径，就能够彼此通过命名管道相互通信（能够访问该路径的进程及命名管道的创建进程之间），因此不相关的进程也能通过命名管道交换数据。

注意：管道和命名管道都是实实在在的文件，但是前者没有公开的文件名，用户在文件系统中不能直接观察并访问到它；命名管道则是以普通文件形式存在的，任何进程都可以将其当成一个普通文件进行处理。

总之，命名管道与管道的区别主要体现在以下两点。

- 命名管道可用于任何两个进程间的通信，而并不限制这两个进程同源，因此命名管道的使用此管道的使用要灵活方便得多。
- 命名管道作为一种特殊的文件存放于文件系统中，而不是像管道一样存放于内存（使用完毕后消失）中。当进程对命名管道的使用结束后，命名管道依然存在于文件系统中，除非对其进行删除操作，否则该命名管道不会消失。

命名管道的出现极好地解决了系统在应用过程中产生的大量中间临时文件的问题。命名管道可以被 Shell 调用，使数据从一个进程到另一个进程，系统不必为该中间通道而烦恼于清理不必要的垃圾，或者去释放该通道的资源，它可以留给后来的进程使用。

另外，需要注意的是，命名管道严格遵循先进先出（first in first out）的规则，对管道及命名管道的读总是从开始处返回数据，对它们的写则把数据添加到末尾，所以它们不支持诸如 lseek 函数等文件定位操作。

10.5.1 在 Linux 中使用命名管道

在 Shell 环境下，可以很简单地识别出命名管道文件。文件名后面紧跟着一个竖线，就是命名管道文件的标志。而在程序中，由于命名管道文件是一种特殊类型的文件，可以通过 S_ISFIFO 宏来检测。

在 Shell 中可以使用“mkfifo”命令建立一个命名管道，mkfifo 命令的格式如下所示：

```
mkfifo [option] name...
```

其中，option 选项中可以选择要创建的命名管道模式，使用形式为-m mode，这里 mode 指出将要创建的命名管道的八进制模式。注意，这里新创建的命名管道会像普通文件一样受到创建进程的 umask 修正。name 表示所要创建的命名管道的名称。

关于更详尽的信息，用户可随时使用“man mkfifo”命令查看帮助信息。

一旦建立了一个命名管道，就可以像普通文件那样对其使用 open、close、read、write、unlink 等文件操作函数了。但是由于命名管道是个特殊的文件，不像普通管道那样存在于内核中，仅创建并不能立即使用，必须打开才能进行读写操作。读写操作要特别注意以下几点。

像普通管道那样，如果没有其他写进程打开一个命名管道，就对其进行读操作时，会产生 SIGPIPE 信号；如果所有的写进程都关闭命命名管道，对其的读操作就会认为到达文件末尾。

在多个写进程的情况下，写交错现象就有可能发生。与普通管道相同，只要一次写入

的字符数不超过 PIPE_BUF，就不会产生写交错现象。

命名管道常常产生阻塞状态。也就是说，如果一个读进程打开命名管道，那么这个进程就要进入阻塞状态，直到其他写进程打开这个管道为止。同样，如果一个写进程打开命名管道，这个进程也会出现阻塞状态，直到其他读进程打开这个管道为止。

如果用户不希望出现这种阻塞状态，可以通过设置 O_NONBLOCK 标志来实现。这样，不管有没有写进程，读打开操作都会立即返回。但是，如果没有读进程，写打开操作就会产生错误。

10.5.2　命名管道的常用工作方式

命名管道通常有如下两种工作方式。

- 命名管道由 Shell 命令使用，以便将数据从一条管道传送到另外一条，此时无须创建一个中间临时文件。
- 命名管道用于客户进程和服务器进程的应用程序中，以在客户进程和服务器进程之间传递数据。

1．命名管道的数据流复制传送

命名管道可用于复制串行管道之间的数据流，此时不需要将数据写入到中间磁盘文件，因为命名管道具有名字，其可用于非线性连接。

注意：管道没有名字，所以只能用于进程之间的线性连接。

图 10.4 是需要对一个输入流进行两次处理的操作。

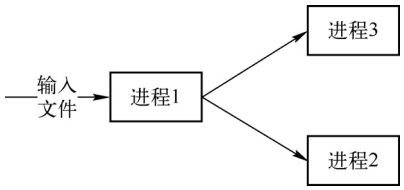


图 10.4　对一个输入流进行两次处理的操作

使用命名管道及 tee 命令则可以实现以上功能，tee 命令从标准输入设备读取数据，将其内容输出到标准输出设备，同时保存成文件，用户可利用 tee 把管道导入的数据存成文件，甚至一次保存数份文件，tee 命令的标准调用格式说明如下。

```
tee [OPTION]... [FILE]...
```

用户可以使用如下的命令序列来实现相应的操作：

```
mkfifo fifo        //创建 fifo 命名管道
prog3 < fifo&    //后台启动进程 3
prog1<infile|tee fifo1|prog3
/*从 fifo 读取数据然后启动进程 1 并且使用 tee 命令将进程 1 的输出复制到标准输出
和文件 infile*/
```

使用命名管道将一个流发送到两个进程如图 10.5 所示。

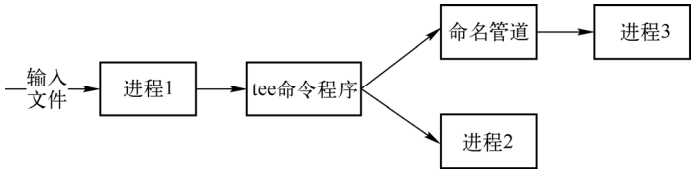


图 10.5　使用命名管道将一个流发送到两个进程

2．命名管道的终端通信

命名管道还经常用于在客户进程和服务器进程之间传送数据，如果有一个服务器进程需要和多个客户进程相关，则每个客户进程都可以将这个请求写到一个该服务器进程所创建的公共的命名管道中，如图 10.6 所示。

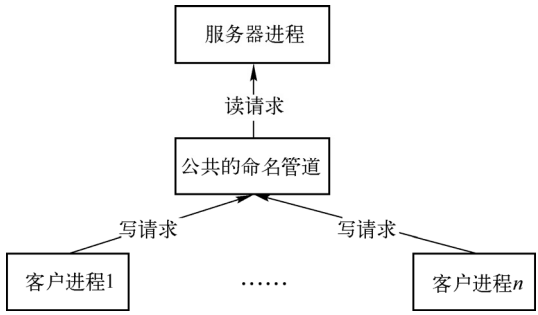


图 10.6　服务器进程和客户进程使用命名管道通信

在这个通信模型中，最重要的一个问题是服务器进程如何将应答回馈给各个客户进程，最常见的解决方案是每个客户进程都在其发送的数据包中包含进其进程 ID，然后服务器进程根据这些进程 ID 来为每个客户进程创建一个命名管道，如图 10.7 所示。

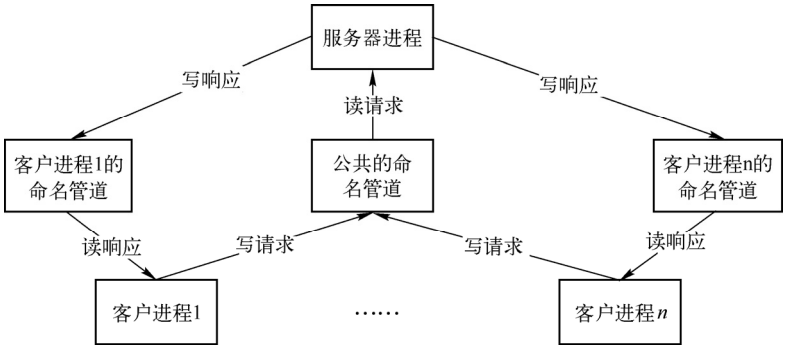


图 10.7　服务器进程和客户进程使用命名管道通信的完整模型

如图 10.7 所示的模型中，服务器进程可以只读方式打开公共的命名管道，当客户进程都关闭之后，服务器进程会在这个公共的命名管道中读到一个 EOF（文件结束标志）。这种工作模型有如下两个缺点。

- 服务器进程不能判断一个客户进程是否会崩溃终止，此时为响应客户进程所建立的客户专属命名管道可能会丧失操作者，从而成为系统垃圾。
- 由于客户进程和服务器进程采用“发送–响应”的工作模式，所以服务器进程必须捕捉 SIGPIPE 信号，否则会出现响应不及时而导致客户进程挂起的情况。

10.5.3　命名管道的打开和读写

对命名管道的操作和对普通文件的操作十分相似，可以使用系统调用 open 打开一个命名管道，使用 read 和 write 函数对命名管道进行读写，使用 close 关闭一个命名管道，若要删除一个命名管道，则使用系统调用 unlink。这些函数均在第 4 章中进行了详细的介绍。

1．打开

管道没有公开的名字，所以不能进行打开操作，当然其也不需要进行打开操作，但是命名管道是以一个普通文件存在的，用户可以对其进行打开操作（如调用 open 函数等）。但是命名管道的打开与其他文件的打开是有区别的，其打开规则说明如下。

- 在当前打开操作是为读而打开命名管道时，若已经有相应进程为写而打开该命名管道，则当前打开操作将成功返回；否则可能阻塞，直到有相应进程为写而打开该命名管道（当前打开操作设置了阻塞标志）；或者，成功返回（当前打开操作没有设置阻塞标志）。
- 在当前打开操作是为写而打开命名管道时，若已经有相应进程为读而打开该命名管道，则当前打开操作将成功返回；否则可能阻塞，直到有相应进程为读而打开该命名管道（当前打开操作设置了阻塞标志）；或者，返回 ENXIO 错误（当前打开操作没有设置阻塞标志）。

2．读

从命名管道中读取数据必须遵循以下规则。

- 如果一个进程为了从命名管道中读取数据而阻塞打开命名管道，那么称该进程内的读操作为设置了阻塞标志的读操作。
- 如果有进程写打开命名管道，且当前命名管道内没有数据，则对于设置了阻塞标志的读操作来说，将一直阻塞。对于没有设置阻塞标志读操作来说，则返回–1，当前 errno 值为 EAGAIN，提醒以后再试。
- 对于设置了阻塞标志的读操作说，造成阻塞的原因有两种：当前命名管道内有数据，但有其他进程在读这些数据；另外，命名管道内没有数据。解阻塞的原因则是命名管道中有新的数据写入，无论新写入数据量的大小，也不考虑读操作请求多少数据量。
- 读打开的阻塞标志只对本进程第一个读操作施加作用。如果本进程内有多个读操作序列，则在第一个读操作被唤醒并完成读操作后，其他将要执行的读操作将不再阻塞，即使在执行读操作，命名管道中没有数据也一样（此时，读操作返回 0）。
- 如果没有进程写打开命名管道，则设置了阻塞标志的读操作会阻塞。

注意：如果命名管道中有数据，则设置了阻塞标志的读操作不会因为命名管道中的字

节数小于请求读的字节数而阻塞，此时，读操作会返回命名管道中现有的数据量。

3．写

向命名管道中写入数据必须符合以下规则。

- 如果一个进程为了向命名管道中写入数据而阻塞打开命名管道，那么称该进程内的写操作为设置了阻塞标志的写操作。
- 对于设置了阻塞标志的写操作，当要写入的数据量不大于 PIPE_BUF 时，Linux 将保证写入的原子性。如果此时管道空闲缓冲区不足以容纳要写入的字节数，则进入睡眠状态，直到缓冲区中能够容纳要写入的字节数时才开始进行一次性写操作。
- 当要写入的数据量大于 PIPE_BUF 时，Linux 将不再保证写入的原子性。命名管道缓冲区一有空闲区域，写进程就试图向管道写入数据，写操作在写完所有请求写的数据后返回。
- 对于没有设置阻塞标志的写操作，当要写入的数据量大于 PIPE_BUF 时，Linux 将不再保证写入的原子性。在写满所有命名管道空闲缓冲区后，写操作返回。
- 当要写入的数据量不大于 PIPE_BUF 时，Linux 将保证写入的原子性。如果当前命名管道空闲缓冲区能够容纳请求写入的字节数，写完后成功返回；如果当前命名管道空闲缓冲区不能够容纳请求写入的字节数，则返回 EAGAIN 错误，提醒以后再写。

注意：原子操作（atomic operation）指的是有多个步骤组成的操作。简单来讲，操作的原子性是指某一事务中的所有操作要么全部执行、要么全部不执行，不可能只执行所有步骤的一个子集。

10.6 Linux 的命名管道操作

10.6.1 使用 mkfifo 函数来创建命名管道

Linux 内核提供了相应的函数来创建命名管道。

1．mkfifo 函数基础

mkfifo 函数的标准调用格式说明如下：

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

mkfifo 函数的 pathname 参数是一个普通的路径名，也就是创建后命名管道文件的名字；mode 参数是文件的操作权限；如果函数调用成功返回 0，否则返回–1。

如果 mkfifo 函数的 pathname 参数所指示的文件已经存在，则返回 EEXIST 错误，所以一般典型的调用代码首先会检查是否返回该错误，如果确实返回该错误，那么只要调用打开命名管道的函数就可以了。通常来说，文件的 I/O 操作函数都可以用于 FIFO，如 close、read、write 等。

注意：在使用“man”命令查看 mkfifo 函数的相关说明时，必须使用“man 3 mkfifo”。

2.【应用实例】——创建指定名称命名管道

例 10.11 是一个使用 mkfifo 来创建一个指定名称的命名管道的应用实例，命名管道的名称从参数 avgv 传入。

【例 10.11】 创建指定名称命名管道。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[ ])
{
    mode_t mode = 0666;      /*新创建的 FIFO 模式*/
    if(argc != 2)
            {
        printf("USEMSG: create_FIFO { FIFO name}\n"); /*向用户提示程序使用帮助*/
        exit(1);
     }
    if((mkfifo(argv[1], mode)) < 0)   /*  使用 mkfifo 函数创建一个 FIFO 管道*/
            {
        perror("failed to mkfifo!\n");
        exit(1);
     }
    else
        {   /*输出 FIFO 文件的名称*/
        printf("you successfully create a FIFO name is : %s\n", argv[1]);
        }
  return 0;
}
```

注意：使用“ls”命令可以在对应的文件夹中看到对应的命名管道文件。

10.6.2 【应用实例】——命名管道的读写

命名管道的读写操作和普通文件类似，可以使用第 8 章中介绍的相应文件操作函数（如 open 等），例 10.12 是一个建立命名管道并对其进行读写的应用实例。

【例 10.12】 命名管道的读写。

```
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/stat.h>
```

```
#include <fcntl.h>
#define FIFO   "FIFO2"   //定义 FIFO 名称
int main(void)
{
    pid_t pid;
    int fd_r,fd_w;
    int num_r,num_w;
    int len = 40;
    char buf_r[len];      //读写缓冲区
    char buf_w[len];
    //建立缓冲区，注意权限的设置
    if((mkfifo(FIFO,O_CREAT|O_EXCL)<0)&&(errno!=EEXIST))
    {
        perror("create fifo fail!\n ");
        exit(1);
    }
    if((pid=fork())==0)   //创建子函数
    {
        sleep(1);
        printf("进程(%d):准备打开 fifo ",getpid());
        fd_r = open(FIFO,O_RDONLY,0);
        if(fd_r==-1)//以阻塞方式读打开，没有其他进程写则打开操作阻塞
        {
            printf("进程(%d):打开 fifo 失败... ",getpid());
            return;
        };
        printf("进程(%d):打开 fifo 成功. ",getpid());
        while(1)
        {
            num_r = read(fd_r,buf_r,sizeof(buf_r));
            printf(      "读字节数：%d ",num_r);
            sleep(1);
            //这里将验证写阻塞,写进程不休眠，很快就把管道写满，再写的时候就阻塞了
        }
    }
    else if(pid>0)
    {

        printf("进程(%d):准备打开 fifo ",getpid());
        //if(open(FIFO,O_WRONLY|O_NONBLOCK,0)==-1)
        fd_w = open(FIFO,O_WRONLY,0);
        if(fd_w==-1)//以阻塞方式写打开，没有其他进程读则打开操作阻塞
        {
            printf("进程(%d):打开 fifo 失败... ",getpid());
            return;
        }
```

```
        printf("进程(%d):打开 fifo 成功. ",getpid());
        while(1)
        {
            num_w = write(fd_w,buf_w,sizeof(buf_w));
            printf(      "写字节数：%d ",num_w);

        }
    }
}
```

10.7　Linux 中的线程同步操作

当多个控制线程共享相同的内存时，需要确保每个线程看到一致的数据视图，如果每个线程使用的变量都是其他线程不会读取或修改的，就不存在一致性问题，否则就需要注意同步问题。通常来说，用户可以使用互斥量或同步变量方式来解决线程同步问题。

10.7.1　使用互斥锁实现线程同步

互斥锁是一个简单的锁定命令，它可以用来锁定对共享资源的访问。对于线程来说，整个地址空间都是共享的资源，所以线程的任何资源都是共享的资源。互斥锁具有以下三个主要特点。

- 原子性：把一个互斥锁定为一个原子操作，这意味着操作系统（或 pthread 函数库）保证了如果一个线程锁定了一个互斥锁，没有其他线程在同一时间可以成功锁定这个互斥锁。
- 唯一性：如果一个线程锁定了一个互斥锁，在它解除锁定之前没有其他线程可以锁定这个互斥量。
- 非繁忙等待：如果一个线程已经锁定了一个互斥锁，第二个线程又试图去锁定这个互斥锁，则第二个线程将被挂起（不占用任何 CPU 资源），直到第一个线程解除对这个互斥锁的锁定为止，第二个线程则被唤醒并继续执行，同时锁定这个互斥锁。

Linux 内核提供了相应的函数来完成对应的操作。

1．pthread_mutex_init 函数

pthread_mutex_init 用来初始化一个由参数 mutex 指向的互斥锁，这个互斥锁的属性由参数 attr 指定，或者通过指定 attr 为 NULL 而使用默认的属性，其标准调用格式说明如下：

```
#include <pthread.h>
pthread_mutex_t fastmutex=PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t recmutex=PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;
pthread_mutex_t errchkmutex=PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutex_attr *attr);
```

上面三个常量是常用的处理互斥锁的常量。

不会出现有多个线程同时初始化同一个互斥锁的情形，一个互斥锁在使用期间一定不会被重新初始化。

如果 pthread_mutex_init 执行成功，则返回 0，并将新创建的互斥锁的 ID 值放到参数 mutex 中。如果执行失败，那么将返回一个错误编号。

2．thread_mutex_destroy 函数

pthread_mutex_destroy 函数用于解除由参数 mutex 指向的互斥锁的任何状态，其标准调用格式说明如下：

```
#include <pthread.h>
int pthread_mutex_destroy(pthread_mutex_t *mutex)
```

需要注意的是，储存互斥锁的内存并不被释放，如果 pthread_mutex_destroy 执行成功则返回 0；如果执行失败，那么将返回一个错误编号。

3．pthread_mutex_lock 函数

pthread_mutex_lock 函数可以用于锁定由参数 mutex 指向的互斥锁，其标准调用格式说明如下：

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

如果 mutex 已经被锁定，那么当前调用的线程将阻塞，直到互斥锁被其他线程释放（阻塞线程按照线程优先级等待）。当 pthread_mutex_lock 返回时，说明互斥锁已经被当前线程成功加锁。

如果 pthread_mutex_lock 执行成功则返回 0，其他值则说明发生了错误。

4．pthread_mutex_trylock 函数

pthread_mutex_trylock 函数用于尝试给由参数 mutex 指定的互斥锁加锁，其标准调用格式说明如下：

```
#include <pthread.h>
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

该函数是 pthread_mutex_lock 的非阻塞版本。pthread_mutex_lock 在给一个互斥锁加锁时，如果互斥锁已经被锁定，那么 pthread_mutex_lock 将一直阻塞，不会立即返回。而使用 pthread_mutex_trylock 给一个互斥锁加锁时，如果互斥锁已经被锁定，那么 pthread_mutex_trylock 调用将返回错误；否则，互斥锁将被调用者加锁。

如果 pthread_mutex_trylock 执行成功则返回 0，其他值意味着错误。

5．pthread_mutex_unlock 函数

可以使用 pthread_mutex_unlock 函数给由参数 mutex 指定的互斥锁解锁，其标准调用格式说明如下：

```
#include <pthread.h>
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

互斥锁必须处于加锁状态，而且调用本函数的线程必须是给互斥锁加锁的线程，这样才能给互斥锁解锁。如果有其他线程在等待互斥锁，那么由核心调度程序决定哪个线程将获得互斥锁并脱离阻塞状态。

如果 pthread_mutex_unlock 执行成功则返回 0，其他值意味着错误。

6.【应用实例】——使用互斥锁对缓冲区进行访问控制

例 10.13 是线程同步的应用实例，一个线程从共享的缓冲区中读数据，另一个线程向共享的缓冲区中写数据，使用一个互斥锁来对共享的缓冲区进行访问控制。

【例 10.13】 使用互斥锁对缓冲区进行访问控制。

```
#include <stddef.h>
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <stdlib.h>
#define FALSE 0
#define TRUE 1
void readfun();
void writefun();
char buffer[256];
int buffer_has_item=0;
int retflag=FALSE;
pthread_mutex_t mutex;
int main(void)
{
   pthread_t reader;
   pthread_mutex_init(&mutex,NULL);
   pthread_create(&reader,NULL,(void *)&readfun,NULL);
   writefun();
   exit(0);
}
void readfun(void)
{
   while(1)
   {
      if(retflag)
      {
         return;
      }
      pthread_mutex_lock(&mutex);
      if(buffer_has_item==1)
      {
          printf("%s",buffer);
          buffer_has_item=0;
      }
```

```
            pthread_mutex_unlock(&mutex);
        }
        return;
    }

    void writefun(void)
    {
        int i=0;
        while(1)
        {
            if(i==10)
            {
                retflag=TRUE;
                return;
            }
            pthread_mutex_lock(&mutex);
            if(buffer_has_item==0)
            {
                sprintf(buffer,"This is %d\n",i++);
                buffer_has_item=1;
            }
            pthread_mutex_unlock(&mutex);
        }
        return;
    }
```

10.7.2 使用条件变量实现线程同步

在程序中使用互斥锁虽然可以解决一些资源竞争的问题，但是互斥锁只有两种状态，这使得它的用途非常有限。

Linux 还提供了另外一种同步机制，即条件变量。条件变量是对互斥锁的补充，它允许线程阻塞并等待另一个线程发送的信号。当收到信号时，阻塞的线程就被唤醒并试图锁定与之相关的互斥锁。

下面是 Linux 的线程库中处理条件变量的一些函数。

1．pthread_cond_init 函数

pthread_cond_init 函数用于初始化由参数 cond 指定的条件变量，其标准调用格式说明如下：

```
#include <pthread.h>
int pthread_cond_init(pthread_cond_t *cond, const pthread_cond_attr *attr);
```

这个条件变量的属性由参数 attr 指定。如果参数 attr 为 NULL，那么就使用默认的属性设置。

多线程不能同时初始化同一个条件变量。如果一个条件变量正在使用，它不能被重新

初始化。

如果 pthread_cond_init 执行成功，则返回 0，并将新创建的条件变量的 ID 放在参数 cond 中，如果返回其他值则意味着有错误。

2．pthread_cond_destroy 函数

pthread_cond_destroy 函数用于清除由参数 cond 指向的条件变量的任何状态，其标准调用格式说明如下：

```
#include <pthread.h>
int pthread_cond_destroy(pthread_cond_t *cond);
```

需要注意的是，储存条件变量的内存空间不被释放，如果函数 pthread_cond_destroy 执行成功则返回 0，其他值意味着错误。

3．pthread_cond_wait 函数

pthread_cond_wait 函数原型：

```
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

使用 pthread_cond_wait 释放由参数 mutex 指向的互斥锁，并且使调用线程关于参数 cond 指向的条件变量阻塞。被阻塞的线程可以被 pthread_cond_signal、pthread_cond_broadcast 或由 fork 和传递信号引起的中断唤醒。

即使返回错误信息，pthread_cond_wait 通常在互斥锁被调用线程加锁后才返回。

函数将阻塞直到条件变量被信号唤醒。它在阻塞前自动释放互斥锁，在返回前再自动获得它。

如果有多个线程关于条件变量阻塞，其退出阻塞状态的顺序将不确定。

如果 ptread_cond_wait 执行成功则返回 0，其他值意味着错误。

4．pthread_cond_timewait 函数

pthread_cond_timewait 函数原型：

```
#include <pthread.h>
int pthread_cond_timewait(pthread_cond_t *cond, pthread_mutex_t *mutex,
        const struct timespec *abstime);
```

pthread_cond_timedwait 和 pthread_cond_wait 的用法相似，区别在于 pthread_cond_timedwait 在经过由参数 abstime 指定的时间时不阻塞。

即使返回错误，pthread_cond_timedwait 也只在给互斥锁加锁后返回。

pthread_cond_timedwait 函数将阻塞，直到条件变量获得信号或经过由 abstime 指定的时间。

如果 pthread_cond_timedwait 执行成功则返回零；如果阻塞条件变量的时间超过了由参数 abstime 所指定的时间，那么返回 ETIMEOUT；其他值意味着错误。

5．pthread_cond_signal 函数

pthread_cond_signal 函数原型：

```
#include <pthread.h>
int pthread_cond_signal(pthread_cond_t *cond);
```

使用 pthread_cond_signal 使得关于由参数 cond 指向的条件变量阻塞的线程退出阻塞状态。在同一个互斥锁的保护下使用 pthread_cond_signal，否则条件变量可以在对关联条件变量的测试和 pthread_cond_wait 带来的阻塞之间获得信号，这将导致无限期的等待。

如果没有一个线程关于条件变量阻塞，那么 pthread_cond_signal 无效。

如果 pthread_cond_signal 执行成功则返回 0，其他值意味着错误。

6．pthread_cond_broadcast 函数

pthread_cond_broadcast 函数原型：

```
#include <pthread.h>
int pthread_cond_broadcast(pthread_cond_t *cond);
```

使用 pthread_cond_broadcast 使得所有关于由参数 cond 指向的条件变量阻塞的线程退出阻塞状态。如果没有阻塞的线程，cond_broadcast 无效。

这个函数将唤醒所有由 pthread_cond_wait 阻塞的线程。因为所有关于条件变量阻塞的线程都同时参与竞争，所以使用这个函数时需要小心。

如果 pthread_cond_broadcast 执行成功则返回 0，其他值意味着错误。

第11章

在嵌入式 Linux 中进行网络编程

网络是 Linux 系统和外部进行数据交互的重要通道，本章将介绍在 Linux 下使用 C 语言进行相关网络编程的基础方法，涉及以下内容：

- Linux 的网络通信模型；
- 套接字基础和使用方法；
- 在 Linux 下进行 TCP 编程的方法；
- 在 Linux 下进行 UDP 编程的方法。

11.1 Linux 的网络通信模型

Linux 系统和网络是息息相关的，其内核稳定支持包括 TCP/IP（IPv6）、IPX、DDP 等在内的多种网络协议，同时 Shell 也提供了多个功能强大的连网命令，如 FTP、Telnet 等。

11.1.1 OSI 网络模型

计算机网络模型是为了简化网络的研究、设计与实现而抽象出来的一种结构模型，通常采用层次模型；在每个层次模型中，往往将系统所要实现的复杂功能分化为若干个相对简单的细小功能，每一项分功能以相对独立的方式去实现。

开放系统互联参考模型 OSI（Open System Interconnection Reference Mode）是国际标准化组织（ISO）提出的一个设计和描述网络通信的基本框架，其结构如图 11.1 所示，包括了物理层、数据链路层、网络层、传输层、会话层、表示层、应用层共 7 层，各层的详细说明如下。

- 物理层（Physical Layer）：这是计算机网络的底层，也是基础层，是有关物理设备通过物理媒体进行互连的描述和规定。物理层协议定义了接口的机械特性、电气特性、功能特性、规程特性；其以比特流的方式传送来自数据链路层的数据，而不去理会数据的含义或格式。

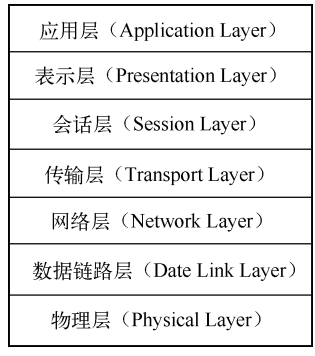


图 11.1 OSI 的网络结构模型

- 数据链路层（Data Link Layer）：该层承担了两个数据设备（计算机等）通过物理层进行无差错传输数据帧的工作，通常来说这些数据帧的传输都需要等待接收方的确认，有错误或丢失的数据帧必须重新传送。
- 网络层（Network Layer）：该层负责信息寻址和将逻辑地址与名字转换为物理地址。在网络层中传输的是数据包，其需要选择合适的路径并转发数据包，使发送方的数据包能够正确无误地按地址寻找到接收方的路径，并将数据包交给接收方。网络层通常还需要对数据包进行重组以满足数据链路层对数据帧大小的要求，并且需要考虑不同协议之间的互连问题。
- 传输层（Transport Layer）：该层负责在不同子网中的两个数据设备之间数据包可以可靠、顺序、无错地传输，在该层中传输的是数据段，其向高层用户提供端到端的可靠的透明传输服务，为不同进程间的数据交换提供可靠的传送手段。
- 会话层（Session Layer）：其利用传输层提供的端到端服务，向表示层或会话用户提供会话服务。会话层的主要功能是在两个节点间建立、维护和释放面向用户的连接，并对会话进行管理和控制，保证会话数据可靠传送。
- 表示层（Presentation Layer）：其负责在不同的数据格式之间进行转换操作，以实现不同计算机系统间的信息交换；还负责编码、加密、压缩等操作。
- 应用层（Application Layer）：其直接和用户及应用程序进行数据交互，包括大量应用协议，如 Telnet、SSH、DNS、HTTP 等。

通常来说，可以把 OSI 网络模型的低四层（物理层、数据链路层、网络层、传输层）称为数据流层，而把高三层（会话层、表示层、应用层）称为应用层。

在 OSI 网络模型的基础上发展出了许多种类的实际应用模型，11.1.2 节中将要介绍的 TCP/IP 模型即为其中一种。

11.1.2 TCP/IP 协议和其网络模型

TCP/IP（Transmission Control Protocol/Internet Protocol）是由美国国防部创建的模型，是发展至今最成功的通信协议，被应用于架构互联网（Internet），Linux 系统的网络功能也是基于该协议实现的。

1．TCP/IP 协议的分层

TCP/IP 协议是一组在网络中提供可靠数据传输和无连接数据服务的协议。其中提供可靠数据传输的协议称为传输控制协议（TCP），而提供无连接数据包服务的协议叫作网际协议（IP）。

注意：TCP/IP 协议并不只有 TCP 和 IP 两个协议，而是包含很多其他协议的一个网络协议的集合。

TCP/IP 协议出现得比 OSI 更早，其也有自己的网络模型，但是其并不存在和 OSI 的 7 层严格的一一对应关系，主要是并不存在物理层和数据链路层，可以分为网络接口层、网络层、传输层和应用层四部分，图 11.2 给出了其和 OSI 模型的比较。

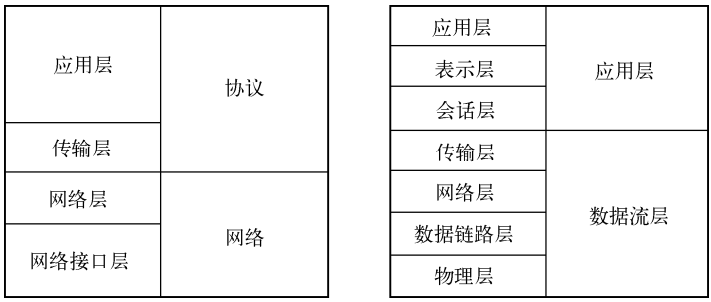


图 11.2　TCP/IP 协议模型和 OSI 模型的比较

TCP/IP 协议是一系列（到目前为止有 100 多个）协议的集合，这些协议分别对应 TCP/IP 协议的每个层次，如图 11.3 所示，这些层次和协议的说明如下。

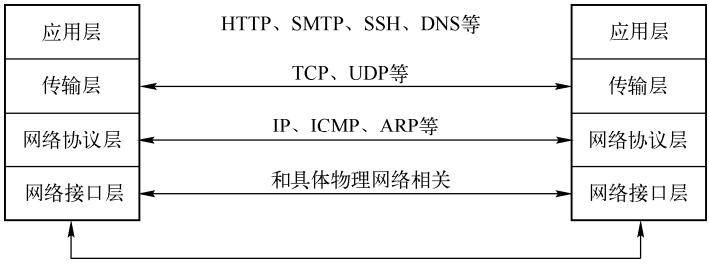


图 11.3　TCP/IP 的模型层次和对应的协议

- 应用层：提供各种常用的高层协议，包括 FTP（文件传输）、Telnet（远程登录）、SMTP（简单邮件传送）、HTTP（超文本传输）、DNS（域名服务）等。
- 传输层：提供了从发送方端口到接收方端口的数据传输协议，最常用的协议是 TCP 和 UDP，前者是一个面向连接的协议，提供无差错的字节流的可靠传输；而后者是一个不面对连接的协议，11.3 节和 11.4 节将分别对使用这两个协议进行数据传输的方法进行详细介绍。
- 网络协议层：在功能上类似于 OSI 模型中的网络层，负责检查网络拓扑结构，以决定传输数据的最佳路由，其最重要的功能是实现 IP 地址和主机的对应，最常用的协议包括 IP（网际协议）、ICMP（因特网控制消息协议）和 ARP（地址解释协议）等。
- 网络接口层：类似于 OSI 模型中的物理层和数据链路层的集合，主要用于实现数据在物理上的传输，即正确地发送和接收 IP 的分组，其涉及的协议和具体的网络相关，如令牌网、分组交换网的相关协议等。

2．IP 协议规定的 IP 地址

网络（该网络是指宏观的因特网，该地址指独立 IP）中任何一台数据设备都必须有一个独一无二的 IP 地址，在 IP 协议中规定了一个 IP 地址由 4 字节组成，如 159.77.16.17，其对应的二进制数字为：10011111. 01001101.00010000.00010111。

在 IP 协议中定义了 A、B、C、D 四种主要的地址类。

- A 类地址：第一位固定为 0，第 1 字节（前 8 位）为网络标识符，用来标识网络，其余 3 字节用来标识网络中的主机。因此最多有 127 个 A 类网络，每个 A 类网络可以容纳 1700 万台主机。
- B 类地址：前两位固定为 10，第 1 字节和第 2 字节（前 16 位）为网络标识符，用来标识网络，其余 2 字节用来标识网络中的主机。因此最多有 16 000 个 B 类网络，每个 B 类网络可以容纳 65 000 台主机。
- C 类地址：前 3 位固定为 110，前 3 字节（前 24 位）为网络标识符，用来标识网络，最后 1 字节用来标识网络中的主机。因此最多有 200 万个 C 类网络，每个 C 类网络可以容纳 254 台主机。
- D 类地址：前四位固定为 1110，D 类地址是多目地址，标识在网络上运行分布式应用的一群主机。因此，D 类主机并不标识一个在线的主机。

3．Linux 中的 TCP/IP 模型结构

Linux 中的 TCP/IP 模型结构如图 11.4 所示，其是分层模型的良好体现，各层详细说明如下。

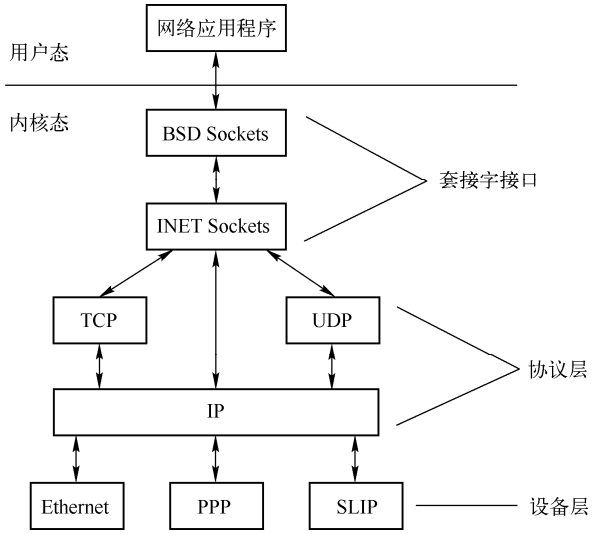


图 11.4　Linux 中的 TCP/IP 模型结构

- 在用户态的最上层是各种网络应用程序，包括浏览器、邮件服务器等，使用 FTP、SSH 等网络协议。
- 网络应用程序通过 BSD Sockets（BSD 套接字）和 INET Sockets（INET 套接字）这两个套接字接口和 TCP 协议或 UDP 协议进行数据交互，其中 INET 套接字协议还可以直接和 IP 协议进行数据交互。
- TCP 协议和 UDP 协议分别是可靠的有连接的通信协议和不可靠的无连接的通信协议，它们都会和 IP 协议进行数据交互，而它们和 IP 协议一起被统称为协议层。
- 在 IP 层（或者说协议层）下方即为网络接口层，如 PPP、以太网（Ethernet）等，需

要注意的是，网络设备并不完全等同于物理设备，因为一些网络设备（如同馈设备）是完全由软件实现的。和其他那些使用 mknod 命令创建的 Linux 系统的标准设备不同，网络设备只有在软件检测到和初始化这些设备时才在系统中出现。当构建系统内核时，即使系统中有相应的以太网设备驱动程序，也只能看到/dev/eth0（网卡）。

11.1.3　客户端/服务器结构

TCP/IP 协议允许两个数据设备建立通信并传输数据，但是其并没有规定这两个数据设备之间的数据传输方法，所以用户需要自行规定一种方法以达到数据有组织地传输，通常来说会使用客户端/服务器（Client/Server）结构模式来实现，在这种模式下要求这两个数据设备上运行的应用程序一个作为服务器端存在而另外一个作为客户端存在，它们的结构如图 11.5 所示，功能说明如下。

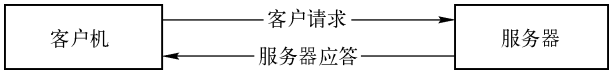


图 11.5　C/S 结构

- 客户端：这是为了得到某种服务所需要运行的应用程序，即申请服务的程序。
- 服务器端：在网络上可提供服务的程序，其接收网络上客户端的请求，完成服务后将结果返回给客户端。

C/S 结构的服务器端通常也可以分为一个主程序和几个从程序，前者负责接收来自客户端的数据，而从程序则负责处理各个客户端的请求，然后交给主程序进行处理。所以服务器端通常可以同时接受一个或多个客户的请求，当客户发送某个服务请求时，服务器将其在提供该服务的端口排队，然后从队列中提取请求，为每个请求创建一个子进程，由子进程来处理具体的服务细节，其过程如图 11.6 所示。

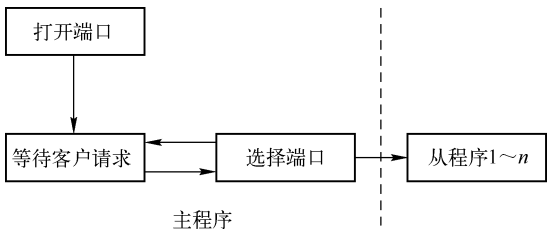


图 11.6　服务器端的主程序和从程序

11.1.4　Linux 的端口和套接字

Linux 的端口是一个逻辑概念，其由 TCP/IP 协议定义，是一个 0～65 535 之间的数字，可以分为常用的“固定”端口和通用端口两部分。所谓的“固定”端口是指一些常用的软件或 TCP/IP 协议中确定和公布的，通常来说不会被其他程序使用，Linux 中的常见端口和对应的协议如表 11.1 所示。

表 11.1　常用端口和协议

协　议	端　口　号	协　议	端　口　号
FTP	21	SSH	22
Telnet	23	HTTP	80
TFTP	69	SMTP	25
SNMP	161	DNS	53

所谓套接字（Sockets），即网络进程（服务器端程序或客户端程序）的进程 ID，和普通的进程 ID 不同，网络进程的 ID 是由运行这个进程的计算机的 IP 地址及这个进程使用的端口（port）所组成的。在同一台计算机上一个端口只能分配给一个进程，这样就可以确定网络中计算机上的一个进程。套接字的组成如图 11.7 所示。

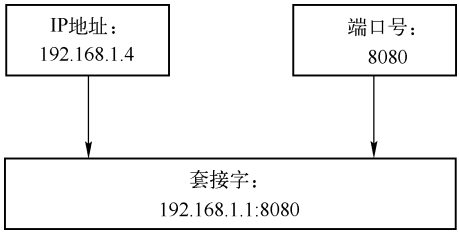


图 11.7　Linux 的套接字组成

如图 11.7 所示，Linux 的套接字包括 BSD 套接字和 INET 套接字两部分，其中 BSD 套接字接口是 Linux 套接字的基础，从某种意义上来说套接字可以看作一种特殊的管道，BSD 套接字通常包括如下几种类型。

- Stream（数据流）：该套接字提供了两个方向的序列数据流，这些数据流保证在传输过程中数据不丢失、破坏或重复，数据流套接字由 Internet（INET）地址族的 TCP 协议所支持。
- Datagram（数据报）：该套接字也提供两个方向上的数据传送，但不像数据流套接字那样，它们不提供消息到达的保证。即使到达也不保证这些数据报按照一定的顺序到达或丢失、重复。这种类型的套接字由 Internet 地址族的 UDP 协议所支持。
- Raw（原始套接字）：该套接字允许进程直接访问底层协议。例如，可以为以太网设备打开一个 Raw Socket，以使用原始 IP 数据进行传输。
- Reliable Delivered Message（可靠传递消息）：该套接字非常像数据报套接字，但是保证数据的可靠传输。
- Sequenced Packets（顺序数据报）：这个套接字像数据流套接字但数据包的大小固定。
- Packet（包）：这不是标准的 BSD 套接字类型，它是一个 Linux 特定的扩展，允许进程在设备层直接访问 Packet。

Linux 网络编程中最常使用的是支持 TCP 协议的数据流套接字、支持 UDP 协议的数据报套接字和可以直接对底层协议 IP 进行访问的原始格式套接字。

11.1.5　Linux 套接字的结构定义

Linux 在头文件 sys/socket.h 中定义了一种通用的套接字结构类型以供不同的协议进行调用，其说明如下：

```
struct sockaddr
  {
    unsigned short int sa_family;          //套接字协议地址类型
    unsigned char sa_data[14];             //14 字节的协议地址，包括 IP 地址和端口
  };
```

该结构中各分量说明如下。

（1）sa_family：套接字的协议族地址类型，表 11.2 是常见的协议所对应的 sa_family 值。

（2）sa_data：具体的协议地址，不同的协议族对应不同的地址结构。

表 11.2　常见协议对应的 sa_family 值

可　选　值	说　　明
AF_INET	IPv4 协议
AF_INET6	IPv6 协议
AF_LOCAL	UNIX 协议
AF_LINK	链路地址协议
AF_KEY	密钥套接字

除了 socketaddr 之外，Linux 还在 netinet/in.h 中定义了另外一种结构类型 sockaddr_in，这种类型的说明如下，其和 sockaddr 等效且可以互相转换，通常来说会在涉及 TCP/IP 的协议编程中使用。

```
struct sockaddr_in
{
        int sa_len:                        //长度单位
        short int sa_family;               //地址族
        unsigned short int sin_port;       //端口号
        struct in_addr sin_addr;           //IP 地址
        unsigned char sin_zero[8];         //填充 0  以保持与 struct sockaddr 同样大小
};
```

该结构中各个分量说明如下。

（1）sa_len：长度单位，无须设置，通常固定长度为 16 字节。

（2）sa_family：协议族，参考表 11.2。

（3）sin_port：端口号。

（4）sin_addr：IP 地址，其本身也也是一个结构体，该结构体的描述说明如下。

```
struct in_addr
```

```
{
    in_addr_t    s_addr;      /*32 位 IPv4 地址，网络字节顺序*/
};
```

（5）sin_zero：填充 0，目的是保持该结构和 sockaddr 结构大小相同以方便转换。

在使用结构 sockaddr_in 时需要注意以下几点。

- 结构 sockaddr_in 中的 TCP 或 UDP 端口号 sin_port 和 IP 地址 sin_addr 都是以网络字节顺序存储的，在 11.2 节中将介绍将主机字节顺序的数据转换成网络字节顺序的方法。
- 32 位的 IP 地址可以用两种不向的方法引用。例如，假设定义变量 servaddr 为 Internet 套接字地址结构，那么可以用 servaddr.sin_addr 或 servaddr.sin_addr.s_addr 来引用这个 IP 地址。需要注意的是，前一种引用是结构类型（struct in_addr）的数据，而后一种引用是整数类型的数据；当将 IP 地址作为函数参数使用时，需要明确使用哪种类型的数据，因为编译器对结构类型参数和整数类型参数的处理方式不一样。
- sin_zero 成员未被使用，它是为了和通用套接字地址（struct sockaddr）保持一致而引入的，通常会被填充为 0。
- 套接字地址结构仅供本机 TCP 协议记录套接字信息而用，这个结构变量本身是不在网络上传输的，但是其某些内容，如 IP 地址和端口号是在网络上传输的，这也是这两部分数据需要转换成网络字节顺序的原因。

11.2 在嵌入式 Linux 中进行网络基础操作

本节将介绍几个和 Linux 网络编程相关的基础操作函数，包括字节顺序转换函数族、字节操作函数族、IP 地址转换函数族和域名转换函数族。

11.2.1 使用字节顺序转换函数族来转换地址模式

嵌入式系统（计算机系统）内部的数据存储通常有大端模式和小端模式两种，其说明如下。

- 大端模式：高位字节优先。
- 小端模式：低位字节优先。

通常来说 PC 中的数据存储采用的是小端模式，某些大型机上的数据存储则采用的是大端模式，而网络中的数据传输则采用大端模式，所以需要对存储的数据进行大小端转换，图 11.8 给出了大端模式和小端模式的区别。

以 32 位宽度的数据 0x12345678 为例来展示在大端模式和小端模式下的存放方法（假设从内存地址 0x8000 开始存放），如表 11.3 所示。

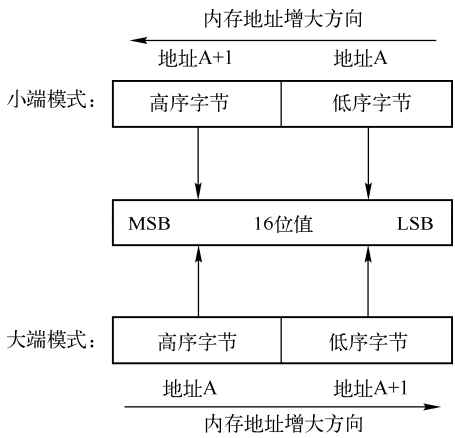


图 11.8 大端模式和小端模式的区别

表 11.3　大端模式和小端模式的数据存放

内 存 地 址	大 端 模 式	小 端 模 式
0x8000	0x12	0x78
0x8001	0x34	0x56
0x8002	0x56	0x34
0x8003	0x78	0x12

Linux 提供了 htonl、htons、ntohl 和 ntohs 四个函数，用于处理大端模式和小端模式的数据调换，其标准调用格式说明如下：

```
#include <arpa/inet.h>
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

函数调用成功则返回处理之后得到的值，如果调用失败则返回-1，函数的功能和参数说明如下。

- htonl 函数：将 32 位的 PC 数据（小端模式存放）转换为 32 位的网络传输数据（大端模式存放）。
- htons 函数：将 16 位的 PC 数据（小端模式存放）转换为 16 位的网络传输数据（大端模式存放）。
- ntohl 函数：将 32 位的网络传输数据转换为 32 位的 PC 数据。
- ntohs 函数：将 16 位的网络传输数据转换为 16 位的 PC 数据。

以上函数的参数均为对应的需要转换值，其中 h 代表 host，n 代表 network，s 代表 short，l 代表 long；32 位的 long 数据通常用于存放 IP 地址，而 16 位的 short 数据则通常用于存放端口号。

11.2.2　使用字节操作函数族操作多字节数据

由于套接字地址和 C 语言中的字符串不同，它是多字节数据，而不是以空字符结尾的，所以 Linux 提供了两组函数来处理这个多字节数据。

1．bzero、bcopy 和 bcmp 函数

第一组函数是和 BSD 系统兼容的函数，包括 bzero、bcopy 和 bcmp，其标准调用格式说明如下：

```
#include <strings.h>
void bzero(void *s, size_t n);
```

函数 bzero 将参数 s 指定的内存的前 n 个字节设置为 0。通常用它来将套接字地址清零。

```
#include <strings.h>
```

```
void bcopy(const void *src, void *dest, size_t n);
```

函数 bcopy 从参数 src 指定的内存区域复制指定数目的字节内容到参数 dest 指定的内存区域。

```
#include <strings.h>
int bcmp(const void *s1, const void *s2, size_t n);
```

函数 bcmp 比较参数 s1 指定的内存区域和参数 s2 指定的内存区域的前 n 字节的内容，如果相同则返回 0，否则返回非 0。

2．memset、memcpy 和 memcmp 函数

第二组则是标准 C（ANSI C）提供的函数，包括 memset、memcpy 和 memcmp，其标准调用格式说明如下：

```
#include <string.h>
void *memset(void *s, int c, size_t n);
```

函数 memset 将参数 s 指定的内存区域的前 n 字节设置为参数 c 的内容。

```
#include <string.h>
void *memcpy(void *dest, const void *src, size_t n);
```

函数 memcpy 和函数 bcopy 功能相似，两个函数的差别是：函数 bcopy 能处理参数 src 和参数 dest 所指定的区域有重叠的情况，而函数 memcpy 对这种情况没有定义，这时应该使用函数 bcopy。

```
#include <string.h>
int memcmp(const void *s1, const void *s2, size_t n);
```

函数 memcmp 比较参数 s1 和参数 s2 指定区域的前 n 字节的内容，如果相同则返回 0，否则返回非 0。

11.2.3 使用 IP 地址转换函数族转换 IP 地址

通常来说，IP 地址会被表示为“192.168.1.1”这样的“点分十进制”方式，而在 Linux 的网络编程中会使用 32 位二进制值，所以 Linux 提供了函数族用于对这两个数值进行转换，这些函数包括 inet_aton、inet_addr 和 inet_ntoa 等。

1．inet_aton、inet_addr 和 inet_ntoa 函数

inet_aton 函数用于将点分十进制数的 IP 地址转换成为网络字节序的 32 位二进制数值。输入的点分十进制数 IP 存放在参数 straddr 中，作为返回结果的二进制数值存放在 addrptr 中。

```
#include <arpa/inet.h>
int inet_aton (const char *straddr, struct in_addr *addrptr);
```

与 inet_aton 函数相反，inet_ntoa 函数调用的结果作为函数的返回值返回给调用它的函数。

```
#include <arpa/inet.h>
char *inet_ntoa (struct in_addr inaddr);
```

int_addr 函数的功能和 inet_aton 函数相同，但是结果传递的方式不同。输入的点分十进制数 IP 仍然存放在参数 straddr 中，但是结果以返回值的形式返回，函数类型为 in_addr_t，不同于 inet_aton 的整型。

```
#include <arpa/inet.h>
in_addr_t inet_addr (const char *straddr);
```

2. 【应用实例】——转换 IP 地址为二进制数值

例 11.1 是一个使用 inet_addr 函数将点分十进制数的 IP 地址转换为网络字节序的 32 位置二进制数值的实例。应用代码将从 argv[1]参数送入的点分十进制字符串调用 inet_addr 函数获得网络字节序的二进制数值，然后将这个数值利用 printf 函数在屏幕上输出。

【例 11.1】 转换 IP 地址为二进制数值。

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
int main(int argc,char *argv[])
{
   unsigned long iptemp;
   if(argc != 2)       //如果参数不正确
   {
     printf("请输入正确的 ip 地址值.\n");
     return 1;
  }
  iptemp = inet_addr(argv[1]);   //调用 inet_addr 函数获得网络地址
  printf("返回的 ip 数值是%lu.\n",iptemp);
  return 0;
}
```

3. 【应用实例】——转换二进制数值为 IP 地址

和例 11.1 相反，例 11.2 是一个使用 inet_ntoa 函数的实例，其展示了如何将网络地址转换为对应的点十分十进制 IP 地址。应用代码首先使用 inet_addr 函数将通过 argv[1]和 argv[2]参数传递的两个点分十进制 IP 地址转换为对应的网络地址，由于 inet_ntoa 函数的参数必须是 in_addr 类型的结构体变量，所以使用 memcpy 函数将网络地址直接复制到两个 in_addr 类型的结构体变量 netaddr1 和 netaddr2 之后再传递给 inet_ntoa 函数进行处理，然后在屏幕上输出。

【例 11.2】 转换二进制数值为 IP 地址。

```
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```
#include <arpa/inet.h>
#include <string.h>
int main(int argc, char *argv[])
{
   struct in_addr addr1,addr2;
   unsigned long netaddr1,netaddr2;
 if(argc != 3)     //如果参数不正确
 {
    printf("请输入正确的参数.\n");
    return 1;   //退出
 }
 netaddr1 = inet_addr(argv[1]);
 netaddr2 = inet_addr(argv[2]);
 memcpy(&addr1, &netaddr1, 4);
 memcpy(&addr2, &netaddr2, 4);          //复制地址
 printf("addr1 = %s : addr2 = %s\n", inet_ntoa(addr1), inet_ntoa(addr2));
 //再次输出两个 ip 地址
 printf("%s\n", inet_ntoa(addr1));
 printf("%s\n", inet_ntoa(addr2));
 return 0;
}
```

11.2.4 使用域名转换函数族转换域名

在实际的网络应用中，常常会使用类似“www.sina.com.cn”这样的域名替代 IP 地址来标识一个服务器，所以需要一个函数将这个域名转换为实际的 IP 地址，还需要一个函数将实际的 IP 地址转换为域名。

1．域名结构体

Linux 在 netdb.h 头文件中定义了一个结构体，用于描述一个主机的相关参数，其形式如下：

```
struct hostent
{
   char *h_name;        //主机的正式名称
   char *h_aliases;     //主机的别名
   int h_addrtype;      //主机的地址类型，IPv4 为 AF_INET
   int h_length;        //主机的地址长度，对于 IPv4 是 4 字节，即 32 位
   char **h_addr_list;  //主机的 IP 地址列表
};
#define h_addr h_addr_list[0]   //主机的第一个 IP 地址
```

2．gethostbyname 和 gethostbyaddr 函数

Linux 提供了 gethostbyname 和 gethostbyaddr 函数用于处理域名和地址的转换，其标准调用格式说明如下：

```
#include <netdb.h>
extern int h_errno;
struct hostent *gethostbyname(const char *name);
#include <sys/socket.h>
struct hostent *gethostbyaddr(const void *addr,socklen_t len, int type);
```

gethostbyname 函数实现域名或主机名到 IP 地址的转换，参数 hostname 指向存放域名或主机名的字符串。

gethostbyaddr 函数实现 IP 地址到域名或主机名的转换，参数 addr 是一个指向含有地址结构（in_addr 或 in_addr）的指针；参数 len 是此结构的大小，对于 IPv4 其值为 4，对于 IPv6 其值为 16，参数 family 为协议族。

调用这两个函数成功时返回一个指向 hostent 结构的指针，若调用失败则返回空指针 NULL，同时设置全局变量 h_errno 为相应的值，h_errno 的可能取值如表 11.4 所示。

表 11.4　h_errno 的可能取值

h_errno	说　明
HOST_NOT_FOUND	找不到对应的主机
TRY_AGAIN	出错重试
NO_RECOVERY	出现了不可修复的错误
NO_DATA	该名字有效，但是没有找到该记录

3. 【应用实例】——获取指定域名对应的 IP 地址

例 11.3 是一个使用 gethostbyname 函数来获取指定域名对应的 IP 地址的实例。应用代码首先定义了一个地址结构体 hpaddr 和一个 hostent 类型的指针 hptr 使用 gehostbyname 函数获取 arvg[1]指定参数的 IP 地址，然后使用 memcpy 将其复制到结构体 hpaddr 中，这是因为 inet_ntoa 需要一个地址类型的结构体作为参数，最后使用 printf 函数输出 inet_ntoa 函数的转换值。

【例 11.3】 获取指定域名对应的 IP 地址。

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <netdb.h>
#include <sys/socket.h>
#include <string.h>
#include <arpa/inet.h>
int main(int argc,char *argv[])
{
  struct hostent *hptr;
  struct in_addr hpaddr;     //定义一个地址结构体
  if((hptr = gethostbyname(argv[1])) == NULL)
  {
     printf("请输入域名.\n");
```

```
        return 1;
    }
    else
    {
        memcpy(&hpaddr,&hptr->h_addr,4);   //复制 ip 地址
        printf("IP 地址为%s.\n",inet_ntoa(hpaddr));
    }
    return 0;
}
```

11.3 在嵌入式 Linux 中操作网络套接字

套接字编程是 Linux 的网络编程基础，本节将介绍一些其相关的函数及其应用方法。

11.3.1 使用 socket 函数创建套接字

Linux 使用 socket 函数来创建一个套接字描述符。

1．socket 函数

socket 函数标准调用格式说明如下：

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

如果函数调用成功则返回套接字的描述符，这是一个正整数；如果函数调用失败则返回−1，函数的各个参数描述如下。

- domain：套接字的协议族，其支持的类型说明如表 11.5 所示，socket 函数可以支持多种网络协议，在使用时必须指定当前使用的协议。

表 11.5 socket 函数支持的协议族

协议族名称	描　　述
AF_UNIX，AF_LOCAL	本地交互协议
AF_INET	IPv4 协议
AF_INET6	IPv6 协议
AF_IPX	IPX-Novell 协议
AF_NETLINK	内核接口设备协议
AF_X25	ITU-T X.25/ISO-8208 协议
AF_AX25	业余无线电 AX.25 协议
AF_ATMPVC	原始 ATM 接入协议
AF_APPLETALK	苹果的 Appletalk 协议
AF_PACKET	底层数据包接口

- type：用于指定当前的套接字类型，socket 函数支持的套接字类型在 11.1.4 节介绍，包括 SOCK_STREAM（数据流）、SOCK_DGRAM（数据报）、SOCK_SEQPACKET（顺序数据报）、SOCK_RAW（原始套接字）、SOCK_RDM（可靠传递消息）、SOCK_PACKET（数据包）。
- protocol：在使用原始套接字之外通常设置为 0，以表示使用默认的协议。

在 Linux 系统中创建一个套接字时会在内核中创建一个套接字数据结构，然后返回 1 个套接字描述符标识这个套接字数据结构。这个套接字数据结构包含连接的各种信息，如对方地址、TCP 状态等。TCP 协议根据这个套接字数据结构的内容来控制这条连接。

2.【应用实例】——创建一个 TCP 套接字

例 11.4 是使用 socket 函数来创建一个 TCP 套接字的实例。应用代码调用 socket 函数建立了一个套接字，设定套接字使用的协议是 AF_INET（即 IPv4），套接字类型为 STREAM，如果创建成功则返回对应的套接字描述符。

【例 11.4】 使用 socket 函数创建套接字。

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
int main(int argc,char *argv[])
{
    int sockfd;       //定义套接口描述符
    if((sockfd = socket(AF_INET,SOCK_STREAM,0))<0)   //建立一个 socket
    {
        printf ("创建套接字失败.\n");
      return 1;
   }
   else //socket 创建成功
   {
       printf("套接字的 ID 是:%d\n",sockfd);
   }
   return 0;
}
```

11.3.2　使用 bind 函数绑定套接字

在创立了套接字之后，需要将本地地址和套接字绑定在一起，此时可以调用 bind 函数。

1. bind 函数

bind 函数的标准调用格式说明如下：

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *addr,socklen_t addrlen);
```

其中参数 sockfd 是使用 socket 函数创建的套接字对应的套接字描述符，addr 是本地地址，addrlen 是套接字对应的地址结构长度；如果 bind 函数执行成功则返回 0，否则返回−1。

在 11.1.3 节中介绍了 Linux 的客户端和服务器模式，在网络通信中服务器和客户端都可以使用 bind 函数来套接字地址，通常来说有以下 5 种模式。

- 服务器指定套接字地址的公认端口号，不指定 IP 地址，服务器调用函数 bind 时，如果设置套接字的 IP 地址为特殊的 INADDR_ANY，表示它愿意接收来自任何网络设备接口的客户端连接。这是服务器最经常使用的绑定方式。
- 服务器指定套接字地址的公认端口号和 IP 地址，服务器调用函数 bind 时，如果设置套接字的 IP 地址为某个本地 IP 地址，这表示服务器只接收来自对应于这个 IP 地址的特定网络设备接口的客户端连接。如果这台机器只有一个网络设备接口，这和第一种情况是没有区别的，但当这台机器有多个网络设备接口时，可以用这种方式来限制服务器的接收范围。
- 客户端指定套接字地址的连接端口号，在一般情况下，客户端不用指定自己的套接字地址的端口号，当客户端调用函数 connect 进行 TCP 连接时，系统会自动为它选择一个未用的端口号，并且用本地的 IP 地址来填充套接字地址中的相应项。但在有的情况下，客户端需要使用特定端口号。
- 指定客户端的 IP 地址和连接端口号，表示客户端使用指定的网络设备接口和端口号进行通信。
- 指定客户端的 IP 地址，表示客户端使用指定的网络设备接口进行通信，系统自动为客户端选择一个未用的端口号。一般只在主机有多个网络设备接口时使用。

以上组合的总结如表 11.6 所示。

表 11.6　使用 bind 函数对应的组合方式

C/S	IP	port	说　明
服务器	INADDR_ANY	非 0 值	指定服务器的公认端口号
服务器	本地 IP 地址	非 0 值	指定服务器的 IP 地址和公认端口号
客户端	INADDR_ANY	非 0 值	指定客户端的连接端口号
客户端	本地 IP 地址	非 0 值	指定客户端的 IP 地址和连接端口号
客户端	本地 IP 地址	0	指定客户端的 IP 地址

在编写客户端程序时，通常不使用固定的客户端端口号，除非在必须使用特定端口的情况下，因为固定客户机端口号会带来一些不便，如下面两种情况。

- 服务器执行主动关闭操作：服务器最后进入 TIME_WAIT 状态。当客户机再次与这个服务器进行连接时，仍使用相同的客户机端口号，于是这个连接与前次连接的套接字对完全一样，但是因为前次连接处于 TIME_WAIT 状态，并未消失，所以这次连接请求被拒绝，函数 connect 以错误返回。
- 客户端执行手动关闭操作：客户端最后进入 TIME_WAIT 状态。当马上再次执行这个客户机程序时，客户机将继续与这个固定客户机端口号绑定，但因为前次连接处

于 TIME_WAIT 状态，并未消失，系统会发现这个端口号仍被占用，所以这次绑定操作失败，函数 bind 以错误返回。

2. 【应用实例】——绑定一个刚刚创建的套接字

例 11.5 是一个使用 bind 函数绑定套接字的实例。应用代码定义了一个 IPv4 的套接字地址数据结构变量 addr，首先使用 socket 函数创建了一个套接字，然后使用 bzero 函数将结构变量 addr 的值清空，之后分别设置结构体的各个分量，最后调用 bind 函数将这个变量绑定到刚刚创建的套接字上。

【例 11.5】 绑定一个刚刚创建的套接字。

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#define   PORT 5555                    //定义端口号
int main(int argc,char *argv[])
{
    int sockfd;                        //定义套接口描述符
    struct sockaddr_in addr;           //定义 IPv4 套接口地址数据结构 addr
    int addr_len = sizeof(struct sockaddr_in);
    if((sockfd = socket(AF_INET,SOCK_STREAM,0))<0)          //建立一个 socket
    {
        printf("创建套接字失败!\n");
        return 1;
    }
    bzero(&addr,sizeof(struct sockaddr_in));                //清空表示地址的结构体变量
    addr.sin_family = AF_INET;                              //设置 addr 的成员信息
    addr.sin_port = htons(PORT);
    addr.sin_addr.s_addr = htonl(INADDR_ANY);               //IP 地址设为本机 IP
    if(bind(sockfd, (struct sockaddr *)(&addr), sizeof(struct sockaddr))<0)
    {
        printf("绑定端口失败!");
        return 1;
    }
    return 0;
}
```

11.3.3　使用 connect 函数建立连接

使用 socket 函数建立了一个套接字并绑定了地址之后即可使用 connect 函数和服务器建立一个连接。

1．connect 函数

connect 函数的标准调用格式说明如下：

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *addr,socklen_t addrlen);
```

其中参数 sockfd 是套接字创建函数 socket 返回的套接口描述符，参数 addr 指定远程服务器的套接口地址，包括服务器的 IP 地址和端口号；参数 addrlen 指定这个套接字地址的长度；调用成功后函数返回 0，否则返回−1。

在调用 connect 函数建立连接之前，客户端应用代码需要指定服务器端进程的套接字地址，而客户端通常不会指定自己的套接字地址，Linux 会自动地从 1024～5000 的端口范围中为客户端分配一个未被使用的端口号，然后将该端口号和本机的 IP 地址结合在一起放入套接字地址中。

当客户端调用函数 connect 来主动建立连接时，这个函数将启动 TCP 协议的 3 次握手过程（参考 11.4 节），在连接建立之后或发生错误时，函数返回。连接过程中可能有如下几种错误情况。

- 如果客户机 TCP 协议没有接收到对它的 SYN 数据段的确认，函数以错误返回，错误类型为 ETIMEOUT。通常 TCP 协议在发送 SYN 数据段失败之后，会多次发送 SYN 数据段，在所有的发送都失败之后，函数以错误返回。
- 如果远程 TCP 协议返回一个 RST 数据段，函数立即以错误返回，错误类型为 ECONNREFUSED。当远程机器在 SYN 数据段指定的目的端口号处没有服务器进程在等待连接时，远程机器的 TCP 协议将发送一个 RST 数据段，向客户机报告这个错误。客户机的 TCP 协议在接收到 RST 数据段之后，不再继续发送 SYN 数据段，函数立即以错误返回。
- 如果客户机的 SYN 数据段导致某个路由器产生“目的地不可到达”类型的 ICMP 消息，函数以错误返回，错误类型为 EHOSTUNREACH 或 ENETUNREACH。通常 TCP 协议在接收到这个 ICMP 消息之后，记录这个消息，然后继续几次发送 SYN 数据段，在所有的发送都失败之后，TCP 协议检查这个 ICMP 消息，函数以错误返回。

如果调用函数 connect 失败，应该用函数 close 关闭这个套接字描述符，不能再次用这个套接字描述符来调用函数 connect。

2．【应用实例】——连接一个指定的 IP 地址

例 11.6 是一个使用 connect 函数来建立连接的实例。应用代码使用 PORT 和 REMOTE_IP 来分别定义一个端口号和一个 IP 地址，然后分别调用 socket 和 bind 函数创建套接字和绑定套接字，最后使用 connect 函数来连接 argv[1]参数中所指定的 IP 地址。

【例 11.6】 连接一个指定的 IP 地址。

```
#include <stdio.h>
#include <netinet/in.h>
```

```
#include <arpa/inet.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h>
#define PORT 80                                    //定义一个端口号
//#define REMOTE_IP "59.175.132.70"                //定义一个 IP 地址
int main(int argc,char *argv[])
{
    int sockfd;
    struct sockaddr_in addr;                       //定义 IPv4 套接口地址数据结构 addr
    if(argc != 2)
    {
       printf("请输入正确的 IP 地址字符串.\n");
       return 2;
    }
    if( (sockfd = socket(AF_INET,SOCK_STREAM,0))<0 )   //建立一个 socket
    {
        printf("创建套接字失败!\n");
        return 1;
    }
    bzero(&addr,sizeof(struct sockaddr_in));       //清空表示地址的结构体变量
    addr.sin_family = AF_INET;                     //设置 addr 的成员信息
    addr.sin_port = htons(PORT);
    addr.sin_addr.s_addr = inet_addr(argv[1]);     //从 argv[1]中获得目标的 IP 地址
    if(connect(sockfd, (struct sockaddr *)(&addr), sizeof(struct sockaddr))<0)
    {
        printf("连接失败!\n");
        return;
    }
    else
    {
        printf("连接成功!\n");
    }
return 0;
}
```

11.3.4　使用 listen 切换套接字为倾听模式

对于服务器端的应用程序而言，在创立了套接字之后通常需要等待客户端的连接，此时可以使用 listen 函数将该套接字转换为倾听套接字。

listen 函数的标准调用格式说明如下：

```
#include <sys/types.h>
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

其参数 sockfd 为待转换的套接字的描述符，参数 backlog 为设置请求队列的最大长度，调用该函数成功则返回 0，调用失败则返回-1。

服务器需要调用函数 listen 将套接字转换成倾听套接字，以便接收客户机请求。函数 listen 的功能有如下两个。

- 函数 socket 创建的套接字是主动套接字，可以用它来进行主动连接（调用函数 connect），但是不能接收连接请求，而服务器的套接字必须能够接收客户机的请求。函数 listen 将一个尚未连接的主动套接字转换成一个被动套接字；告诉 TCP 协议，这个套接字可以接收连接请求。
- TCP 协议将到达的连接请求排队，函数 listen 的第二个参数指定这个队列的最大长度。

要创建一个倾听套接字，必须首先调用函数 socket 创建一个主动套接字，然后调用函数 bind 将它与服务器套接字地址绑定在一起，最后调用函数 listen 进行转换。这 3 步操作是所有 TCP 服务器所必需的操作。

下面讨论参数 backlog 的作用，这对于理解套接字建立连接的过程非常重要，TCP 协议为每个倾听套接字维护两个队列。

- 未完成连接队列：每个尚未完成 3 次握手操作的 TCP 连接在这个队列中占有一项。TCP 协议在接收到一个客户机 SYN 数据段之后，在这个队列中创建一个新条目，然后发送对客户机 SYN 数据段的确认和自己的 SYN 数据段（ACK+SYN 数据段），等待客户机对自己的 SYN 数据段的确队。此时，套接字处于 SYN_RCVD 状态。这个条目将保存在这个队列中，直到客户机返回对 SYN 数据段的确认，或者连接超时。
- 完成连接队列：每个已经完成 3 次握手操作但尚未被应用程序接收（调用函数 accept）的 TCP 连接在这个队列中占有一项。当一个在未完成连接队列中的连接接收到对 SYN 数据段的确认之后，完成 3 次握手操作，TCP 协议将它从未完成连接队列移到完成连接队列中。这个条目将保存在这个队列中，直到应用程序调用函数 accept 来接收它。

参数 backlog 指定倾听套接字的完成连接队列的最大长度，表示这个套接字能够接收的最大数目的未接收（unaccepted）连接。当一个客户机的 SYN 数据段到达时，倾听套接字的完成连接队列已经满了，TCP 协议将忽略这个 SYN 数据段。对于不能接收的 SYN 数据段，TCP 协议不发送 RST 数据段，原因有两个。

- 假设 TCP 协议在未完成队列满时返回 RST 数据段，那么客户机的函数 connect 将马上以错误返回，不再继续发送连接请求。根据这个 RST 数据段，客户机无法知道究竟是这个端口上没有服务器进程在等待连接，还是在这个端口上等待的服务器的未完成连接队列暂时没有空间。
- 完成队列满的情况是暂时的：经过一段时间之后，应用程序可能调用函数 accept 从

这个完成队列中接收已经建立的连接，于是完成队列中出现新的空间。客户机 TCP 协议在超时之后继续几次发送 SYN 数据段。如果在这几次发送过程中完成连接队列中出现新的空间，那么 TCP 协议将接收这个连接请求，继续正常的 3 次握手操作。如果在这几次发送过程中，完成连接队列中都没有空间，客户机将放弃发送。

注意：TCP 协议下的数据握手发送方式将在 11.4 节详细介绍。

11.3.5　使用 accept 函数接收连接

当服务器端倾听到一个连接之后，可以使用函数 accept 从倾听套接字的完成连接队列中接收一个连接，如果这个完成连接队列为空，则会使得这个进程进入睡眠状态。accept 函数的标准调用格式说明如下：

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

参数 sockfd 指定套接字描述符；参数 addr 为指向一个 Internet 套接字地址结构的指针；参数 addrlen 为指向一个整型变量的指针。当函数 accept 成功执行时，返回 3 个结果：

- 函数返回值为一个新的套接字描述符，标识这个接收的连接；
- 参数 addr 指向的结构变量中存储客户机地址；
- 参数 addrlen 指向的整型变量中存储客户机地址的长度。

如果对客户机的地址和长度都不感兴趣，可以将参数 addr 和 addrlen 设置为 NULL。函数 accept 执行失败时返回−1。

函数 accept 从倾听套接字的完成连接队列中接收一个已经建立起来的 TCP 连接，因为倾听套接字是专为接收客户机连接请求完成 3 次握手操作而用的，所以 TCP 协议不能使用倾听套接字描述符来标识这个连接，于是 TCP 协议创建一个新的套接字来标识这个要接收的连接，并将它的描述符返回给应用程序。现在有两个套接字：一个是调用函数 accept 时使用的倾听套接字；另一个是函数 accept 返回的连接套接字（connected socket）。这两个套接字的作用是完全不同的：一个服务器进程通常只需创建一个倾听套接字，在服务器进程的整个活动期间，用它来接收所有客户机的连接请求，在服务器进程终止前关闭这个倾听套接字；而对于每个接收的（accepted）连接，TCP 协议都创建一个新的连接套接字来标识这个连接，服务器使用这个连接套接字与客户机进行通信操作，当服务器处理完这个客户机请求时，关闭这个连接套接字。

当函数 accept 阻塞等待已经建立的连接时，如果进程捕获到信号，那么函数将以错误返回，错误类型为 EINTR。对于这种错误，一般重新调用函数 accept 来接收连接。

11.3.6　使用 close 函数关闭连接

当操作完成之后，可以使用 close 函数来关闭当前建立的连接，其标准调用格式说明如下，需要注意的是其和第 3 章中介绍的文件操作函数 close 的名称是一样的，但是包含在 unistd.h 头文件中，其参数也不是文件描述符 fd 而是套接字描述符 sockfd，如果函数调用成

功则返回 0，否则返回-1。

```
#include <unistd.h>
int close(int fd);
```

套接字描述符的 close 操作和文件描述符的 close 操作一样：函数 close 将套接字描述符的引用计数减 1，如果描述符的引用计数大于 0，则表示还有进程引用这个描述符，函数 close 正常返回；如果描述符的引用计数变为 0，则表示再没有进程引用这个描述符，于是启动清除套接字描述符的操作，函数 close 立即正常返回。清除套接字描述符的操作是：将这个套接字描述符标记为关闭状态，然后立即返回进程。调用了函数 close 之后，进程将不再能够访问这个套接字，但是这不表示 TCP 协议删除了这个套接字。TCP 协议将继续使用这个套接字，将尚未发送的数据传递给对方，然后发送 FIN 数据段，执行关闭操作，一直等到这个 TCP 连接完全关闭之后，TCP 协议才删除这个套接字。

11.3.7 使用 read 和 write 函数读写套接字

函数 read 和 write 从套接字中读和写数据。其定义如下：

```
int read(int fd, char *buf, int len);
int write(int fd, char *buf ,int len);
```

参数 fd 指定读写操作的套接字描述符；函数 read 的参数 buf 指定接收数据缓冲区；函数 write 的参数 buf 指定发送数据缓冲区；参数 1en 指定接收或发送的数据量大小。函数 read 成功执行时，返回读到的数据量大小；否则返回-1。函数 write 成功执行时，返回写入的数据量大小；否则返回-1。

11.3.8 使用 getsockname 和 getpeername 函数获取套接字地址

当需要获取套接字的地址时，可以使用 getsockname 函数和 getpeername 函数，前者用于返回本地的套接字地址，后者用于返回和本机套接字建立了连接的对等套接字地址，其标准调用格式说明如下：

```
#include <sys/socket.h>
int getsockname(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
int getpeername(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

getsockname 函数用于获取由描述符 socket 给出的套接字的本地捆绑名，其存储 socket 的地址在 add 参数所指的 sockaddr 结构对象中，存储其地址长度在 addrlen 所指对象中。如果地址的实际长度大于 addr 所指对象的长度，存储的地址将被截断，如果 socket 还没有捆绑地址，存储在 addr 所指对象中的值将是未定义的。该函数调用成功则返回 0，调用失败则返回 1。

存储在 addr 参数所指对象中的地址的格式依赖于该套接字的通信域。对于给定的通信域，套接字地址的长度通常是固定的，如果用户需要确切地知道空间大小，并提供实际需要的存储空间，通常的做法是用与套接字通信域相匹配的数据类型为 addr 所指对象分配空

间，然后强制其地址转为“struct socket *”并传送给 getsockname。

getsockname 函数通常会应用于以下情况。

- 对于没有使用 bind 捆绑地址至套接字的客户进程，在它成功调用 connect 之后，getsockname 可以返回内核指定给该套接字的本地地址（如 IP 地址和端口号等）。
- 用 0 端口号（告诉内核选择本地端口号）调用 bind 之后，getsockname 可以返回内核指定给该套接字的本地端口号。
- getsockname 可以获得一个套接字的地址族。
- 服务进程在接收了客户的连接之后（成功调用 accept 之后)，以 accept 返回的描述符调用 getsockname 可以获得指定给该连接的套接字地址，这个套接字是实际连接的套接字，而不是侦听套接字。

函数 getpeername 用于获取一个套接字的远程对等套接字的地址，其返回与 socket 连接的套接字的地址，并且存储这个地址在 addr 参数所指对象中，存储该地址的长度在 addrlen 所指对象中，如果地址的实际长度大于 addr 提供的长度，存储的地址将被截断，调用该函数成功时会返回值 0，如果调用该函数失败则返回−1。

注意：虽然从 accept 的返回参数中也能得到对等套接字的地址，但是当服务程序是由调用 accept 的进程通过 fork 和 exec 执行时，由于 accept 返回参数的存储空间位于父进程中，因此经 exec 后它将不复存在。在这种情况下，getpeername 是唯一能获得对等套接字地址的方法。

11.3.9　使用 send 和 recv 函数发送和接收数据

除了之前介绍的读写函数 read 和 write 之外，用户还可以使用 recv 和 send 函数来在套接字中实现数据发送和接收，recv 和 send 函数类似于标准的 read 和 write 函数，但它们只能用于套接字，并且还需要一个其他参数，此参数指明控制套接字特殊传输方式的各种标志，其标准调用格式如下：

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

send 函数用于启动从 socket 指定的套接字传送一条消息到对等的套接字，如果调用成功则返回实际发送的字节数，如果失败则返回−1。sockfd 参数为套接字描述符；buf 为待发送数据的缓冲区 len 的数据长度；但是函数发送的实际长度可能小于其指定的长度；flags 用于指定消息的传送类型，当该值为 0 时 send 函数和 write 函数完全相同，或者使用如下两种取值。

- MSG_OOB：send 函数发送的数据称为带外数据，带外数据是流套接字特有的。在流套接字上传送数据时，数据按它们写出的顺序传送。因为接收进程必须依次读套接字上的当前数据，因此，当出现一个紧急情况时，没有办法立即通知接收进程。带外数据正是用于解决这一问题的。带外数据在正常的数据流之外发送，其效果相

当于越过套接字上所有等待读的数据。当它到达接收进程时，接收进程会收到一个信号，从而进程可以立即处理这个数据。

- MSG_DONTROUTE：不在消息中包含路由信息，通常来说普通应用不会关心相应的信息。

注意：send 函数的成功返回值仅表明其把 buf 的数据发送出去了，并不代表消息已经被正确接收。

rev 函数用于从已经连接的套接字接收消息，调用成功则返回读到 buffer 所指缓冲区中的数据的字节长度；如果没有消息可接收并对等套接字已执行了 shutdown，将返回 0；否则返回-1。其 socket、buf 和 len 参数和 send 函数完全相同，flags 参数则用于指明接收到的消息的类型，该参数为 0 时 rev 函数和 read 函数完全相同，或者使用如下三种取值。

- MSG_OOB：读带外数据。
- MSG_PEEK：窥视套接字上的数据而不实际读出它们，即尽管 buffer 所指对象中填入了所请求的数据，随后的 read 或 recv 将读到相同的数据。
- MSG_WAITALL：请求函数阻塞直至所请求的全部数据都已接收到。

注意：read 和 write 函数通常用来读写套接字上的普通数据，当需要发送或接收特殊数据（如带外数据）时，就必须使用 send 和 recv 函数才能做到。

write 函数、send 函数、read 函数和 recv 函数都用于 TCP 协议下的面向连接的套接字数据发送和接收，而在 UDP 协议下面向无连接的套接字数据发送和接收则需要使用 sendto 和 recvfrom 函数，其标准调用格式说明如下：

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,const struct sockaddr *dest_addr,
      socklen_t addrlen);
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,struct sockaddr *src_addr, socklen_t *addrlen);
```

sendto 函数用于在 UDP 协议下发送数据，参数 sockfd 为套接口的描述符，buf 为指向数据发送缓冲区的指针，len 表示将要发送的字节数，flags 一般设置为 0，dest_addr 为指向数据发送的套接口地址数据结构的指针，addrlen 为指向套接口数据结构的长度，当调用该函数成功时返回实际发送的字节数，调用该函数失败时则返回-1。

recvfrom 函数用于在 UDP 协议下接收数据，参数 buf 为指向数据接收缓冲区的指针，srcaddr 为指向数据接收的套接口地址结构的指针，其他参数的含义与 sendto 函数相同，当调用该函数成功时返回实际接收的字节数，调用该函数失败则返回-1。

11.4 在嵌入式 Linux 中进行 TCP 编程

TCP 是 TCP/IP 协议族中面向连接的可靠协议，本节将介绍其工作流程及在 Linux 中对

其进行编程的方法。

11.4.1　TCP 基础

同其他协议栈一样，TCP 向相邻的高层提供服务。因为 TCP 的上一层就是应用层，因此，TCP 数据传输实现了从一个应用程序到另一个应用程序的数据传递。应用程序通过编程调用 TCP 并使用 TCP 服务，提供需要准备发送的数据，用来区分接收数据应用的目的地址和端口号。

通常应用程序通过打开一个 socket 来使用 TCP 服务，TCP 管理到其他 socket 的数据传递。可以说，通过 IP 的源/目的可以唯一地区分网络中两个设备的连接，通过 socket 的源/目的可以唯一地区分网络中两个应用程序的连接。

TCP 对话通过三次握手来进行初始化。三次握手的目的是使数据段的发送和接收同步，告诉其他主机其一次可接收的数据量，并建立虚连接。

图 11.9 描述了这三次握手的简单过程。

（1）初始化主机通过一个同步标志置位的数据段发出会话请求。

（2）接收主机通过发回具有以下项目的数据段表示回复：同步标志置位、即将发送的数据段的起始字节的顺序号、应答并带有将收到的下一个数据段的字节顺序号。

（3）请求主机再回送一个数据段，并带有确认顺序号和确认号。

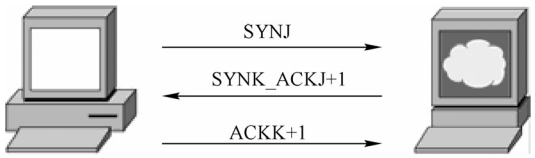


图 11.9　TCP 的三次握手过程示意

TCP 实体所采用的基本协议是滑动窗口协议。当发送方传送一个数据包时，它将启动计时器。当该数据包到达目的地后，接收方的 TCP 实体往回发送一个数据包，其中包含一个确认序号，它表示希望收到的下一个数据包的顺序号。如果发送方的定时器在确认信息到达之前超时，那么发送方会重新发送该数据包。

图 11.10 是 TCP 的数据包头格式，其各个部分说明如下。

- 源端口、目的端口：16 位长。标识出远端和本地的端口号。
- 序号：32 位长。标识发送的数据包的顺序。
- 确认号：32 位长。希望收到的下一个数据包的序列号。
- TCP 头长：4 位长。表明 TCP 头中包含多少个 32 位字。
- 6 位未用。
- ACK：ACK 位置 1 表明确认号是合法的。如果 ACK 为 0，那么数据包不包含确认信息，确认字段被省略。
- PSH：表示是带有 PUSH 标志的数据。接收方因此请求数据包一到便将其送往应用程序而不必等到缓冲区装满才传送。
- RST：用于复位由于主机崩溃或其他原因而出现的错误连接。还可用于拒绝非法的

数据包或拒绝连接请求。

- SYN：用于建立连接。
- FIN：用于释放连接。
- 窗口大小：16 位长。窗口大小字段表示在确认了字节之后还可以发送多少字节。
- 校验和：16 位长。是为了确保高可靠性而设置的。它校验头部、数据和伪 TCP 头部之和。
- 可选项：0 个或多个 32 位字。包括最大 TCP 载荷、滑动窗口比例及选择重发数据包等。

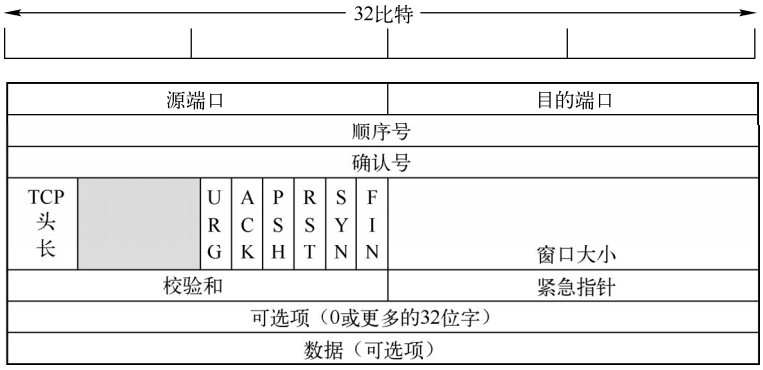


图 11.10　TCP 的数据包格式

11.4.2　TCP 的工作流程

基于 TCP 传输协议的服务器与客户端间的通信工作流程可以用如图 11.11 所示的过程来描述。

（1）服务器先用 socke 函数来建立一个套接口，用这个套接口完成通信的监听及数据的收发。

（2）服务器用 bind 函数来绑定一个端口号和 IP 地址，使套接口与指定的端口号和 IP 地址相关联。

（3）服务器调用 listen 函数，使服务器的这个端口和 IP 处于监听状态，等待网络中某一客户机的连接请求。

（4）客户机用 socket 函数建立一个套接口，设定远程 IP 和端口。

（5）客户机调用 connect 函数连接远程计算机指定的端口。

（6）服务器调用 accept 函数来接收远程计算机的连接请求，建立起与客户机之间的通信连接。

（7）建立连接以后，客户机用 write 函数或 send 函数向 socket 中写入数据。也可以用 read 函数或 recv 函数读取服务器发送来的数据。

（8）服务器用 read 函数或 recv 函数读取客户机发送来的数据，也可以用 write 函数或 send 函数来发送数据。

（9）完成通信以后，使用 close 函数关闭 socket 连接。

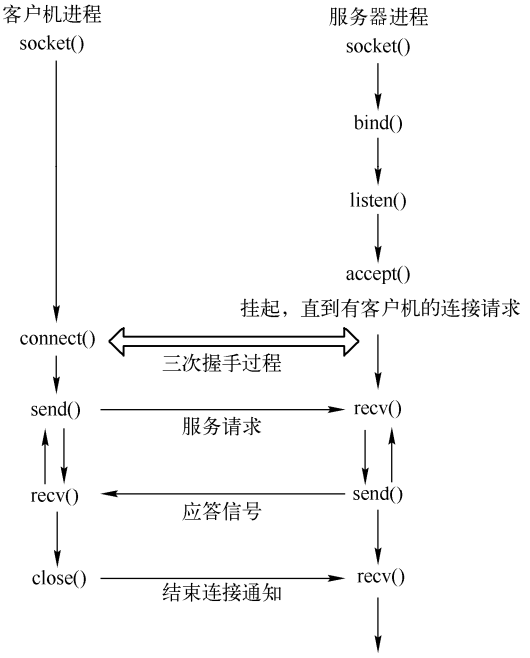


图 11.11　TCP 套接口通信工作流程

11.4.3　【应用实例】——使用 TCP 协议发送当前系统时间

本节介绍一个使用 TCP 进行通信的服务器端和客户端应用实例。服务器端接收客户端请求，创建一个子进程来给客户端发送当前系统时间；客户端则读取服务器端发送的信息。

例 11.7 提供的服务器端应用代码使用端口 25555 作为通信端口，首先调用 socket 函数和 bind 函数建立套接字并绑定端口，然后调用 listen 函数等待客户端连接，如果有客户端的连接信息则使用 accept 函数接收该连接并创建一个子进程，在子进程中发送当前的时间信息，最后在子进程退出时关闭该套接字接口。

【例 11.7】 服务器端。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <time.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define SERV_PORT 25555          //服务器接听端口号
#define BACKLOG     20           //请求队列中允许的请求数
#define BUF_SIZE    256          //缓冲区大小
```

```
int main(int argc,char *argv[])
{
   int ret;
   time_t tt;
   struct tm *ttm;
   char buf[BUF_SIZE];
   pid_t pid;                                  //定义管道描述符
   int sockfd;                                 //定义 sock 描述符
   int clientfd;                               //定义数据传输 sock 描述符
   struct sockaddr_in host_addr;               //本机 IP 地址和端口信息
   struct sockaddr_in client_addr;             //客户端 IP 地址和端口信息
   int length = sizeof client_addr;
   //创建套接字
   sockfd = socket(AF_INET, SOCK_STREAM, 0); //TCP/IP 协议，数据流套接字
   if(sockfd == -1)                            //判断 socket 函数的返回值
   {
      printf("创建 socket 失败.\n");
      return 0;
   }
   //绑定套接字
   bzero(&host_addr, sizeof host_addr);
   host_addr.sin_family = AF_INET;         //TCP/IP 协议
   host_addr.sin_port = htons(SERV_PORT);                  //设定端口号
   host_addr.sin_addr.s_addr = INADDR_ANY;                 //本地 IP 地址
   ret = bind(sockfd, (struct sockaddr *)&host_addr, sizeof host_addr); //绑定套接字
   if(ret == -1)                               //判断 bind 函数的返回值
   {
      printf("调用 bind 失败.\n");
      return 1;
   }
   //监听网络端口
   ret = listen(sockfd, BACKLOG);
   if(ret == -1)                               //判断 listen 函数的返回值
   {
      printf("调用 listen 函数失败.\n");
      return 1;
   }
   while(1)
   {
      clientfd = accept(sockfd, (struct sockaddr *)&client_addr, &length); //接收连接请求
      if(clientfd == -1)
      {
         printf("调用 accept 接收连接失败.\n");
         return 1;
      }
      pid = fork();                            //创建子进程
```

```
        if(pid == 0)                          //在子进程中处理
        {
          while(1)
          {
            bzero(buf, sizeof buf);           //首先清空缓冲区
            tt = time(NULL);
            ttm = localtime(&tt);             //获取当前时间参数
            strcpy(buf, asctime(ttm));        //将时间信息复制到缓冲区
            send(clientfd, buf, strlen(buf), 0);          //发送数据
            sleep(2);
          }
          close(clientfd);                    //调用 close 函数关闭连接
        }
        else if(pid > 0)
        {
          close(clientfd);                    //父进程关闭套接字，准备下一个客户端连接
        }
      }
      return 0;
    }
```

例 11.8 提供的客户端应用代码使用 argv[1]作为连接的 IP 地址，将这个 IP 地址放入 serv_addr 所指定的地址结构体中，在创建套接字之后使用 connect 函数和服务器建立连接，使用 recv 接收服务器发送过来时间信息并打印输出。

【例 11.8】 客户端。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define SERV_PORT 25555       //服务器接听端口号
#define BACKLOG   20          //请求队列中允许的请求数
#define BUF_SIZE   256        //缓冲区大小

int main(int argc, char *argv[])
{
  int ret;
  char buf[BUF_SIZE];
  int sockfd;                 //定义 sock 描述符
  struct sockaddr_in serv_addr;   //服务器 IP 地址和端口信息
  if(argc != 2)
```

```
    {
        printf("命令行输入有误.\n");                    //命令行带 IP
        return 1;
    }
    //创建套接字
    sockfd = socket(AF_INET, SOCK_STREAM, 0);          //TCP/IP 协议，数据流套接字
    if(sockfd == -1)
    {
        printf("调用 socket 函数失败.\n");
        return 2;
    }
    //建立连接
    bzero(&serv_addr, sizeof serv_addr);
    serv_addr.sin_family = AF_INET;                    //TCP/IP 协议
    serv_addr.sin_port = htons(SERV_PORT);             //设定端口号
    //serv_addr.sin_addr.s_addr = INADDR_ANY;          //使用回环地址 127.0.0.1
    inet_aton(argv[1], (struct sockaddr *)&serv_addr.sin_addr.s_addr);      //设定 IP 地址
    ret = connect(sockfd, (struct sockaddr *)&serv_addr, sizeof serv_addr); //绑定套接字
    if(ret == -1)
    {
        printf("调用 connect 函数失败.\n");
        return 3;
    }
    while(1)
    {
        bzero(buf, sizeof buf);
        recv(sockfd, buf, sizeof(buf), 0);             //接收数据
        printf("接收到: %s", buf);
        sleep(1);
    }
    close(sockfd);                                     //关闭连接
    return 0;
}
```

11.5 在嵌入式 Linux 中进行 UDP 编程

UDP 和 TCP 不同，它是一个无连接的协议，所以其建立起来相对简单，不需要进行“三次握手”等操作，但是也具有相对来说不够安全的缺点。

11.5.1 UDP 基础

UDP 协议从问世至今已经被使用了很多年，虽然其最初的光彩已经被一些类似协议所掩盖，但是在网络质量越来越高的今天，UDP 的应用得到了大大增强。它比 TCP 协议更为

高效，也能更好地解决实时性问题。如今，包括网络视频会议系统在内的众多客户/服务器模式的网络应用都使用 UDP 协议。

UDP 的数据报头如图 11.12 所示，其各个部分说明如下。

图 11.12　UDP 的数据报头

- 源地址、目的地址：16 位长。标识出远端和本地的端口号。
- 数据报的长度是指包括报头和数据部分在内的总的字节数。因为报头的长度是固定的，所以该域主要用来计算可变长度的数据部分（又称为数据负载）。

协议的选择应该考虑以下 3 个方面。

（1）数据可靠性：对数据要求高可靠性的应用需选择 TCP 协议，如验证、密码字段的传送都是不允许出错的，而对数据的可靠性要求不那么高的应用可选择 UDP 传送。

（2）应用实时性：TCP 协议在传送过程中要使用三次握手、重传确认等手段来保证数据传输的可靠性。使用 TCP 协议会有较大的时延，因此不适合对实时性要求较高的应用，如 VoIP、视频监控等。相反，UDP 协议则能在这些应用中发挥很好的作用。

（3）网络可靠性：由于 TCP 协议的提出主要是为了解决网络的可靠性问题，它通过各种机制来减少错误发生概率。因此，在网络状况不是很好的情况下需选用 TCP 协议（如在广域网等情况），在网络状况很好的情况下（如局域网等）不需要再采用 TCP 协议了，而建议选择 UDP 协议来减轻网络负荷。

11.5.2　UDP 的工作流程

基于 UDP 传输协议的服务器与客户机间的通信工作流程可以用如图 11.13 所示的过程来描述。

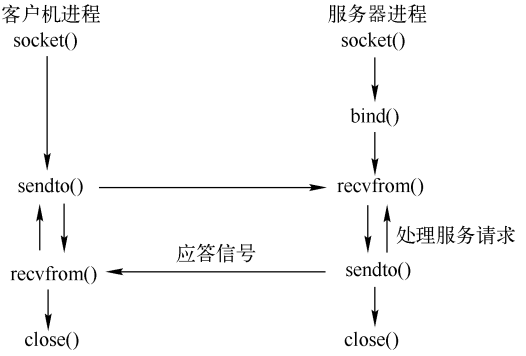


图 11.13　UDP 套接字的工作流程

将图 11.13 与图 11.11 相比较，它们的主要区别在于：使用 TCP 套接口必须先建立连接（如客户进程使用的 connect 函数、服务器进程使用的 listen 函数和 accept 函数），而 UDP 套接口不需要预先建立连接，它在调用 socket 函数生成一个套接口后，在服务器端调用 bind 函数绑定一个端口，然后服务器进程挂起于 recvfrom 函数调用，等待并接收网络中某一客户机的数据请求，而客户端调用 sendto 函数发送数据请求，同样也挂起于 recvfrom 函数调用，等待并接收服务器的应答信号。当数据传送完毕后，UDP 套接口中的客户端调用 close 函数释放通信链路，但不再发送“断开连接通知”信息来通知服务器端释放通信链路。

11.5.3 【应用实例】——使用 UDP 协议发送当前系统时间

本节是一个使用 UDP 协议进行通信的服务器端和客户端应用实例，和 11.4.2 节中介绍的 TCP 协议实例类似，只是发送时间信息的一方改成了客户端，服务器端接收客户端发送的时间信息。

例 11.9 中提供的服务器端应用代码使用端口 25555 作为通信端口，首先调用 socket 函数和 bind 函数建立套接字并绑定端口，然后即可调用 recvfrom 函数来接收数据并将其存放到 buf 缓冲区并判断其返回值，当 recvfrom 函数的返回值不为-1 时表明接收到客户端的数据，此时调用 printf 函数打印输出 buf 缓冲区的值。

【例 11.9】 服务器端。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define SERV_PORT 25555                 //服务器接听端口号
#define BACKLOG      20                 //请求队列中允许的请求数
#define BUF_SIZE   256                  //缓冲区大小

int main(int argc,char *argv[])
{
    int ret;
    char buf[BUF_SIZE];
    pid_t pid;                          //定义管道描述符
    int sockfd;                         //定义 sock 描述符
    int clientfd;                       //定义数据传输 sock 描述符
    struct sockaddr_in host_addr;       //本机 IP 地址和端口信息
    struct sockaddr_in client_addr;     //客户端 IP 地址和端口信息
```

```
    int length = size of client_addr;
    //创建套接字
    sockfd = socket(AF_INET, SOCK_DGRAM, 0);        //TCP/IP 协议，数据流套接字
    if(sockfd == -1)                                //判断 socket 函数返回值
    {
        printf("创建 socket 失败.\n");
        return 1;
    }
    //绑定套接字
    bzero(&host_addr, sizeof host_addr);
    host_addr.sin_family = AF_INET;                 //TCP/IP 协议
    host_addr.sin_port = htons(SERV_PORT);          //设定端口号
    host_addr.sin_addr.s_addr = INADDR_ANY;         //本地 IP 地址
    ret = bind(sockfd, (struct sockaddr *)&host_addr, sizeof host_addr); //绑定套接字
    if(ret == -1)                                   //判断 bind 函数返回值
    {
        printf("调用 bind 函数失败.\n");
        return 1;
    }
    while(1)
    {
        bzero(buf, sizeof buf);
        ret = recvfrom(sockfd, buf, sizeof(buf), 0 ,(struct sockaddr *)&client_addr, &length);
        //接收接连请求
        if(ret ==-1)                                //判断 recvfrom 函数的返回值
       {
            printf("接收连接失败");
            return 1;
        }
       //输出客户端 IP
        printf("接收到: %s\n", buf);
        sleep(2);
    }
    close(clientfd);                                //关闭连接
    return 0;
}
```

例 11.10 提供的客户端应用代码使用时间相关函数获得了时间信息并将其使用 strcpy 函数存放到 buf 缓冲区中，并在建立了套接字之后即调用 sendto 函数来发送 buf 缓冲区的内容。

【例 11.10】 客户端。

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

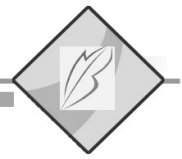
```
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <time.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define SERV_PORT 25555                                    //服务器接听端口号
#define BACKLOG    20                                      //请求队列中允许的请求数
#define BUF_SIZE    256                                    //缓冲区大小

int main(int argc, char *argv[])
{
   int ret;
   time_t tt;
   struct tm *ttm;
   char buf[BUF_SIZE];
   int sockfd;                                             //定义 sock 描述符
   struct sockaddr_in serv_addr;                           //服务器 IP 地址和端口信息
   if(argc != 2)
   {
      printf("命令行输入有误\n");                          //命令行带 IP
      return 1;
   }
   /**创建套接字**/
   sockfd = socket(AF_INET, SOCK_DGRAM, 0);                //TCP/IP 协议，数据流套接字
   if(sockfd == -1)                                        //判断 socket 函数的返回值
   {
      printf("调用 socket 函数创建连接失败.\n");
      return 0;
   }
   /**建立连接**/
   bzero(&serv_addr, sizeof serv_addr);
   serv_addr.sin_family = AF_INET;                         //TCP/IP 协议
   serv_addr.sin_port = htons(SERV_PORT);                  //设定端口号
   inet_aton(argv[1], (struct sockaddr *)&serv_addr.sin_addr.s_addr); //设定 IP 地址
   while(1)
   {
      bzero(buf, sizeof buf);                              //首先清除缓冲区
      tt = time(NULL);
      ttm = localtime(&tt);
```

```
        strcpy(buf, asctime(ttm));                                    //复制缓冲区数据
        sendto(sockfd, buf, strlen(buf), 0, (struct sockaddr *)&serv_addr, sizeof serv_addr);
        //接收数据，然后放入缓冲区
        sleep(2);
    }
    close(sockfd);
    return 0;
}
```

第四部分



综 合 应 用

第 12 章　嵌入式 Linux 综合应用实例

第12章

嵌入式 Linux 综合应用实例

本章提供了 5 个嵌入式 Linux 下的综合应用实例，分别涉及文件操作、硬件操作、进程操作、线程操作和网络操作。

12.1 【应用实例】——定时创建文件写入数据

12.1.1 实例的需求说明和分析

某个应用需要每隔一分钟在当前目录下建立一个以包括当前时间的字符串为文件名的文件，并且每隔一秒钟将一个当前的时间信息的字符串写入文件，其流程如图 12.1 所示。

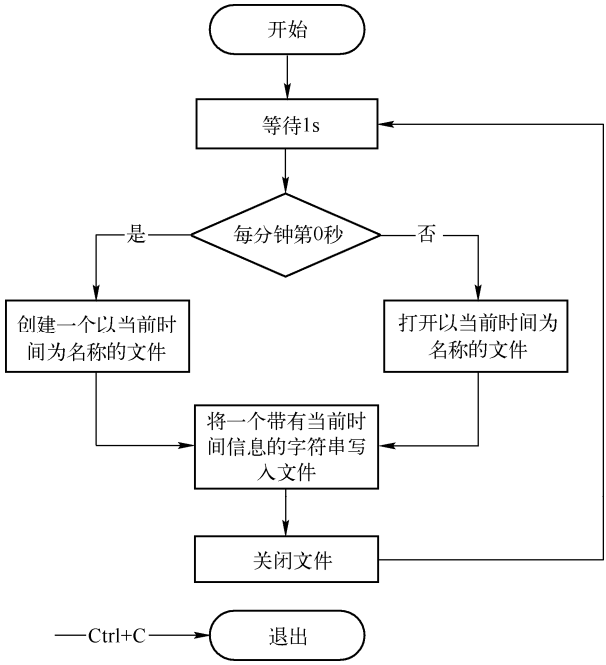


图 12.1　定时创建文件写入数据实例的需求分析

12.1.2 实例的基础设计

对实例的需求进行分析可知，需要考虑如下方面的实现：

- 1 秒钟的定时；
- 将当前时间信息存放进字符串，然后写入文件；
- 将当前时间信息存放进字符串，并且使用该字符串来创建一个文件；
- 将以上的各个模块综合起来。

1．实现秒定时

秒定时是为了实现每隔一秒进行一定的操作，通常来说秒定时有两种实现方法：调用 sleep 函数或通过对当前时间的反复查询然后计算时间的差值以确定秒信息的改变。

通过对当前系统时间信息的获取和判断来实现秒定时的原理是：利用 gettimeofday 函数获取当前的时间，然后和上一次获取的时间进行比较，如果相差超过 1s，则表明定时到达；由于应用一直在调用 gettimeofday 函数，所以其误差基本上只是 gettimeofday 函数及计算时间差的执行时间，比较准确。

例 12.1 是使用 gettimeofday 函数来实现间隔 1s 输出当前时间的实例，其流程如图 12.2 所示。这是一个低效率的使用 gettimeofday 来获得秒定时的应用，使用 gettimeofday 在 while 循环中连续获得当前的 timez 信息和之前的时间信息进行比较，如果还没到 1s，则等待；否则使用 break 跳出 while 循环并打印当前时间，实现每秒打印一次。

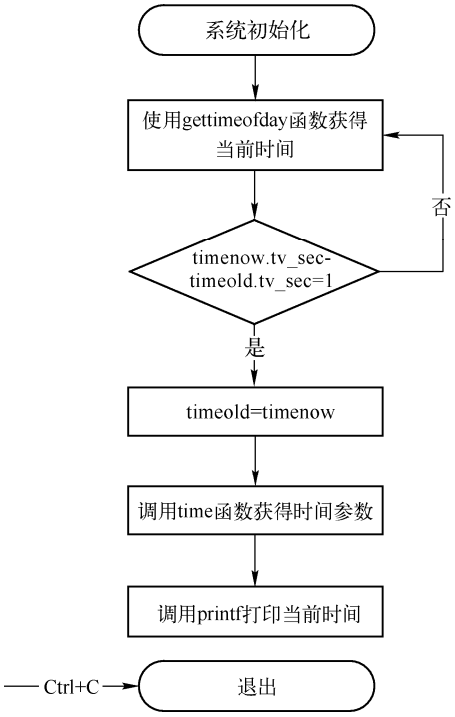


图 12.2　查询时间实现秒定时

注意：关于 gettimeofday 函数的详细说明可以参考 7.6.4 节。

【例 12.1】 查询时间实现秒定时。

```
#include<sys/time.h>
#include<stdio.h>
int main(void)
{
   struct timeval timenow,timeold;
   struct timezone timez;
   time_t timetemp;                          //时间结构体变量
   gettimeofday(&timeold,&timez);            //取得一个时间信息作为以前的数据
   while(1)
   {
      while(1)
      {
      gettimeofday(&timenow,&timez);         //获得当前时间数据
           if((timenow.tv_sec - timeold.tv_sec) == 1)   //如果时间过了 1 秒
           {
              timeold = timenow;             //更新以前的时间参考数据
              break;                         //退出当前循环
           }
       }
       //如果还没到 1 秒，则一直等待；
       time(&timetemp);                       //获得时间参数
       printf("%s",ctime(&timetemp));         //打印当前时间
   }
   return 0;
}
```

2．将当前时间信息写入文件

当前的时间信息可以通过 ctime 函数获得，但是 ctime 函数的返回值是一个字符串指针，需要将其规格化之后放入写缓冲区中，此时可以调用 sprintf 函数来完成。sprintf 是一个将输入参数规格化之后存放到数组缓冲区的函数，其详细使用方法可以参考 man 手册。

例 12.2 是一个每隔 1s 获得当前时间信息然后写入指定文件的实例，其流程如图 12.3 所示，其在参数指定文件中连续写入当前时间的应用文件以 1s 为时间间隔，将当前的时间写入文件，然后按回车键换行。

【例 12.2】 每隔 1s 将时间信息写入文件。

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <sys/time.h>
int main(int argc,char *argv[])
{
   int temp,seektemp;                        //偏移量计算中间量
   int fd;                                   //文件描述符
```

```
    char writebuf[50];                                          //写字符串缓冲区
    struct timeval timenow,timeold;                             //时间变量
    struct timezone timez;
    time_t timetemp;                                            //时间结构体变量
    int j = 0;
    int writeCounter = 0;                                       //写入计数器
    gettimeofday(&timeold,&timez);                              //取得一个时间信息作为参考时间信息
    if(argc!= 2)                                                //如果参数错误
    {
        printf("Plz input the corrcet file name as './exam39lseekFun filename string'!\n");
        return 1;                                               //如果参数不正确则退出
    }
    fd = open(*(argv+1),O_RDWR|O_CREAT,S_IRWXU);
                    //打开文件，如果没有则创建
    while(1)        //进入主循环
    {
      while(1)      //1 毫秒延时判断
      {
        gettimeofday(&timenow,&timez);                          //获取当前时间参数
        if((timenow.tv_sec - timeold.tv_sec) == 1)              //如果到达 1 秒
        {
            timeold = timenow;                                  //更新保存的时间信息
            break;                                              //1 秒时间到，退出
        }
      }
      time(&timetemp);                                          //获得当前时间参数
      sprintf(writebuf,"%s",ctime(&timetemp));                  //将当前时间参数放入写缓冲区
      printf("%s",&writebuf);                                   //在屏幕上打印 writebuf 的内容
      if(writeCounter == 0)                                     //第一次写入
      {
        temp = write(fd,writebuf,strlen(writebuf));             //写入数据
        seektemp = lseek(fd,0,SEEK_CUR);                        //获得当前的偏移量
        writeCounter++;                                         //写入计数器++
      }
      else
      {
        j = strlen(writebuf) * writeCounter;                    //获得偏移量
        seektemp = lseek(fd,j,SEEK_SET);
        temp = write(fd,writebuf,strlen(writebuf));
        writeCounter++;
     }
    }
   close(fd);
   return 0;
}
```

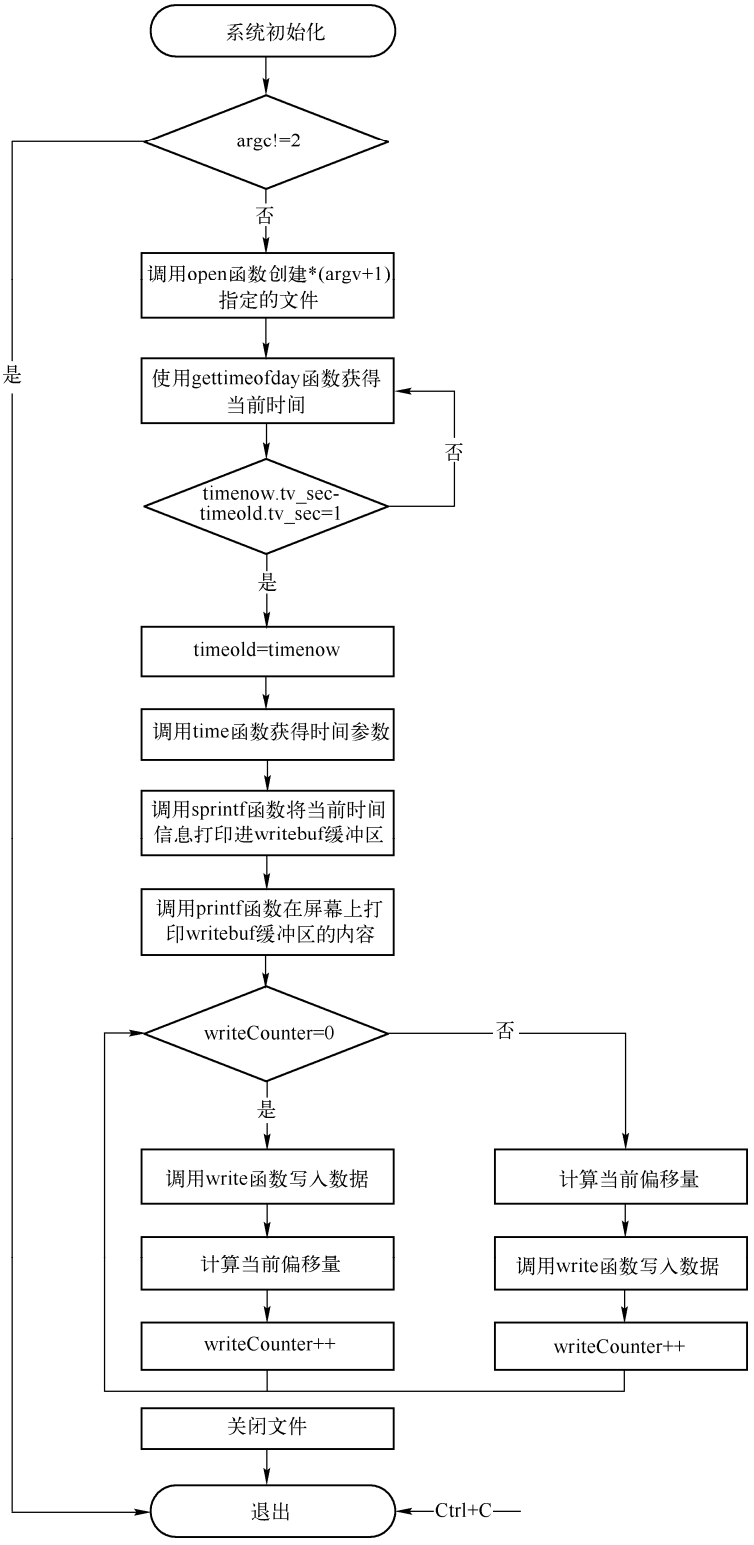


图 12.3　每隔 1s 将当前时间写入文件

3．使用时间信息作为文件名

使用时间信息作为文件名时需要首先将时间信息分离出来，组成一个“时+分+秒”的字符串，然后传递给对应的函数以创建文件，此时可以调用 time 函数来获得当前时间，其返回的时、分、秒信息会分别被存储到结构体的 hour、min 和 sec 分量中。

例 12.3 是一个使用当前时间作为文件名创建一个文件并将创建时的时间信息写入文件的实例，其文件名的结构是“File+时+分+秒”，其流程如图 12.4 所示。其利用当前时间作为参数来创建新文件，应用代码首先使用 time 系列函数获得当前的时、分、秒信息，然后通过组合获得对应的字符串传递给 Open 函数创建文件，最后在文件中写入一个含有时间参数的字符串。

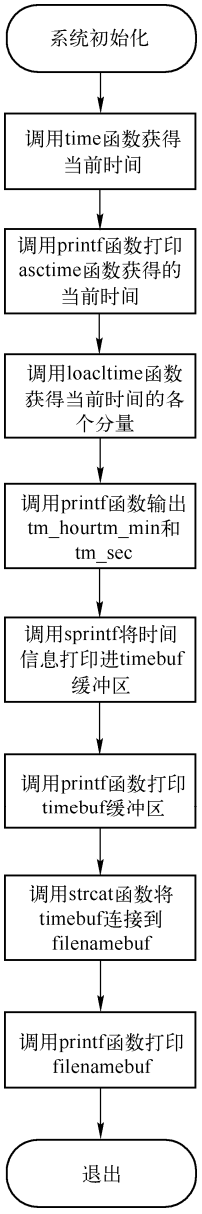


图 12.4　使用时间信息作为文件名创建文件

【例 12.3】 使用当前时间信息作为名称创建文件。

```
#include <time.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
int main(void)
{
        time_t timetemp;                    //定义一个时间结构体变量
        struct tm *p;                       //结构体指针
        int i;
        char timebuf[7];                    //时间信息，注意加上\0
        char writetimebuf[7];               //写文件时间缓冲区
        char filenamebuf[10] = "File";      //文件头
        char writebuf[30]="this is a test! the time is ";
        char enterbuf[3]="\r\n";            //回车换行 buf
        int fd;
        int temp;
        time(&timetemp);                    //获得时间参数
        printf("当前时间为%s",asctime(gmtime(&timetemp)));
                    //不需要添加回车换行符
        p = localtime(&timetemp);
        printf("%d:%d:%d\n",p->tm_hour,p->tm_min,p->tm_sec);
        sprintf(timebuf,"%02d%02d%02d",p->tm_hour,p->tm_min,p->tm_sec);
        //将时、分秒信息按照 2 位前端补 0 的方式格式化送入时间 buf
        printf("step1 timebuf is %s\n",timebuf);
        strcpy(writetimebuf,timebuf);   //复制字符串
        printf("writetimebuf is %s\n",writetimebuf);
        strcat(filenamebuf,timebuf);
        printf("step2 timebuf is %s\n",timebuf);
        printf("filenamebuf is %s\n",filenamebuf);
        fd = open(filenamebuf,O_RDWR|O_CREAT,S_IRWXU);     //创建文件
        strcat(writebuf,writetimebuf);   //连接两个字符串
        strcat(writebuf,enterbuf);          //回车换行
        temp = write(fd,writebuf,strlen(writebuf));   //写入一个字符串以表示正确
        temp = close(fd);
```

```
        return 0;
}
```

12.1.3　实例的综合

将例 12.1～例 12.3 的内容进行综合，即可以得到符合需求的应用，如例 12.4 所示，其流程如图 12.5 所示，应用代码每隔 1min 在当前目录下建立一个新文件，通过查验 tm_sec 是否为 0 来判断，文件以 1s 为时间间隔，将当前的时间写入文件，然后按回车键换行。

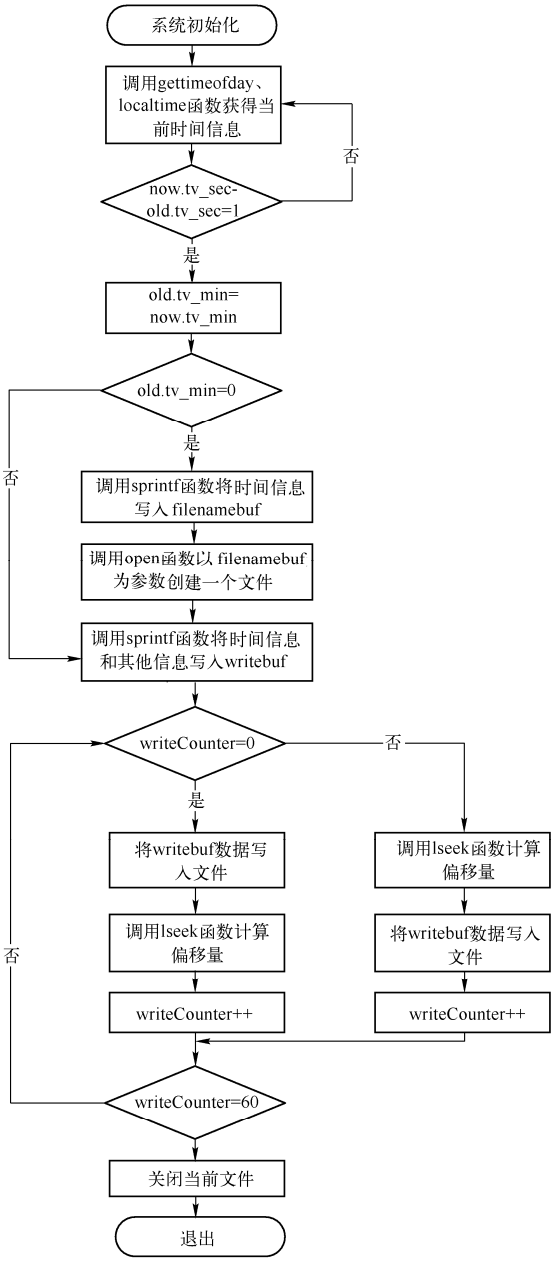


图 12.5　文件基础操作综合应用

【例 12.4】 定时创建文件写入数据。

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <sys/time.h>
int main(int argc,char *argv[])
{
   time_t filetime;
   struct tm *p;
   int temp,seektemp;                                    //偏移量计算中间量
   int fd;                                               //文件描述符
   char writebuf[50];                                    //写字符串缓冲区
   char filenamebuf[10] = "File";                        //文件头
   char timebuf[7];                                      //时、分、秒信息缓冲区
   struct timeval timenow,timeold;                       //时间变量
   struct timezone timez;
   //struct tm *p;                                       //时间结构体指针
   int j = 0;
   int writeCounter = 0;                                 //写入计数器
   gettimeofday(&timeold,&timez);                        //取得一个时间信息作为参考时间信息
   if(argc!= 2)                                          //如果参数错误
   {
     printf("Plz input the corrcet file name as './exam39lseekFun filename string'!\n");
     return 1;                                           //如果参数不正确则退出
   }
   fd = open(*(argv+1),O_RDWR|O_CREAT,S_IRWXU); //打开文件，如果没有则创建
   while(1)     //进入主循环
   {
      while(1)     //1 毫秒延时判断
      {
         gettimeofday(&timenow,&timez);                  //获取当前时间参数
         time(&filetime);
         p = localtime(&filetime);                       //获得时、分、秒参数以供创建新文件
         sprintf(timebuf,"%02d%02d%02d",p->tm_hour,p->tm_min,p->tm_sec);
         printf("%d:%d:%d\n",p->tm_hour,p->tm_min,p->tm_sec);
         //时分秒信息放入 timebuf 缓冲区备用
         gettimeofday(&timenow,&timez);                  //获取当前时间参数
         if((timenow.tv_sec - timeold.tv_sec) == 1)      //如果到达 1 秒
         {
             timeold = timenow;                          //更新保存的时间信息
             break;                                      //1 秒时间到，退出
         }
      }
      if(timeold.tv_sec == 0)                            //如果是 0 秒
      {
```

```
            strcat(filenamebuf,timebuf);                    //创建文件名
            fd = open(filenamebuf,O_RDWR|O_CREAT,S_IRWXU);  //创建文件
        }
        time(&timetemp);                                    //获得当前时间参数
        sprintf(writebuf,"%s",ctime(&timetemp));            //将当前时间参数放入写缓冲区
        printf("%s",&writebuf);                             //在屏幕上打印 writebuf 的内容
        if(writeCounter == 0)                               //第一次写入
        {
            temp = write(fd,writebuf,strlen(writebuf));     //写入数据
            seektemp = lseek(fd,0,SEEK_CUR);                //获得当前的偏移量
            writeCounter++;     //写入计数器++
        }
        else
        {
            j = strlen(writebuf) * writeCounter;            //获得偏移量
            seektemp = lseek(fd,j,SEEK_SET);
            temp = write(fd,writebuf,strlen(writebuf));
            writeCounter++;
        }
      }
    close(fd);
    return 0;
}
```

12.2　【应用实例】——串口双机通信

12.2.1　实例的需求说明和分析

在第 3 章中介绍了在 Linux 下对文件进行操作的方法，在 Linux 中其硬件设备也是以文件形式存在的，计算机或嵌入式系统可以使用串口实现数据通信。本实例即为一个 PC 和嵌入式系统设备使用串口进行双机通信的应用，PC 可以向嵌入式系统发送一组字符串，嵌入式系统接收到该字符串并将其在显示模块上打印输出。

12.2.2　实例的基础设计

串口是 Linux 系统中最常用的数据输入/输出通道之一，尤其是在和嵌入式系统进行数据交互时，其利用一条传输线将数据以比特位为单位顺序传送。特点是通信线路简单，利用简单的线缆就可实现通信，降低成本，适用于传输距离长且传输速度较慢的通信。

1．Linux 下的串口设备和文件

PC 和嵌入式设备的串口通常使用 MAX232 芯片作为接口器件，其内含两套电源变换电路，其中一个升压泵将 5V 电源提升到 10V，而另外一个反相器则提供−10V 的相关信号；该芯片是符合 RS-232-C 标准的通信芯片，一个标准的 RS-232-C 接口包括一个 25 针的 D 型

插座（有公型和母型两种），包括主信道和辅助信道两个通信信道且主信道的通信速率高于辅助信道。在实际使用中，常常只使用一个主信道，此时 RS-232-C 接口只需要 9 根连接线，使用一个简化为 9 针的 D 型插座，同样也分为公型和母型，表 12.1 是 RS-232-C 接口的引脚定义。

表 12.1　串口的引脚定义

25 针接口	9 针接口	名　　称	方　　向	功 能 说 明
2	3	TXD	输出	数据发送引脚
3	2	RXD	输入	数据接收引脚
4	7	RTS	输出	请求数据传送引脚
5	8	CTS	输入	清除数据传送引脚
6	6	DSR	输出	数据通信装置 DCE 准备就绪引脚
7	5	GND	—	信号地
8	1	DCD	输入	数据载波检测引脚
20	4	DTR	输出	数据终端设备 DTE 准备就绪引脚
22	9	RI	输入	振铃信号引脚

RS-232-C 标准推荐的最大物理传输距离为 15m，其逻辑电平“0”为+3V～+25V，而逻辑电平“1”为−3V～−25V，较高的电平保证了信号传输不会因为衰减导致信号的丢失。

在 Linux 系统中，所有的设备文件一般都位于“/dev”下，其中串口 1 和串口 2 对应的设备名依次为“/dev/ttyS0”和“/dev/ttyS1”，而且 USB 转串口的设备名通常为“/dev/ttyUSB0”和“/dev/ttyUSB1”（因版本不同该设备名会有所不同），可以查看在“/dev”下的文件来确认。在 Linux 下对设备文件的操作方法与对普通文件的操作方法是一样的，对串口的读写可以使用 read、write 等函数等进行。需要注意的是，需要对串口的其他参数另做配置。

串口对应的设备文件虽然本质也是文件，但是在具体的使用方法上还是有一些区别的，其操作主要可分为初始化串口、发送数据、接收数据、处理中断和设置波特率等几部分。

2．串口的结构体

在 termios.h 文件中定义了结构体 termios，用于对串口进行初始化和控制，其是在 POSIX 规范中定义的标准接口，表示终端设备（包括虚拟终端、串口等）。

```
#include<termios.h>
struct termios
{
    unsigned short    c_iflag;        /* 输入模式标志 */
    unsigned short    c_oflag;        /* 输出模式标志 */
    unsigned short    c_cflag;        /* 控制模式标志*/
    unsigned short    c_lflag;        /* 本地模式标志 */
    unsigned char     c_line;         /* 线路规程 */
```

```
    unsigned char   c_cc[NCC];          /* 控制特性 */
    speed_t     c_ispeed;               /* 输入速度 */
    speed_t     c_ospeed;               /* 输出速度 */
};
```

3．终端设备和其工作模式

串口是一种终端设备，一般通过终端编程接口对其进行配置和控制。终端有 3 种工作模式，分别为规范模式（canonical mode）、非规范模式（non-canonical mode）和原始模式（raw mode）。

通过在 termios 结构的 c_lflag 中设置 ICANNON 标志来定义终端是以规范模式工作（设置 ICANNON 标志）还是以非规范模式（清除 ICANNON 标志）工作，默认情况为规范模式。

在规范模式下，所有的输入都基于行进行处理。在用户输入一个行结束符（回车符、EOF 等）之前，系统调用 read 函数读不到用户输入的任何字符。EOF 之外的行结束符（回车符等）与普通字符一样会被 read()函数读取到缓冲区中。在规范模式中，行编辑是可行的，而且调用一次 read()函数最多只能读取一行数据。如果在 read 函数中被请求读取的数据字节数小于当前行可读取的字节数，则 read 函数只会读取被请求的字节数，剩下的字节下次再被读取。

在非规范模式下，所有的输入是即时有效的，不需要用户另外输入行结束符，而且不可进行行编辑。在非规范模式下，对参数 MIN（c_cc[VMIN]）和 TIME（c_cc[VTIME]）的设置决定 rea 函数的调用方式。设置通常有如下 4 种不同的情况。

- MIN = 0 和 TIME = 0：read 函数立即返回。若有可读数据，则读取数据并返回被读取的字节数，否则读取失败并返回 0。
- MIN > 0 和 TIME = 0：read 函数会被阻塞直到 MIN 字节数据可被读取。
- MIN = 0 和 TIME > 0：只要有数据可读或经过 TIME 个十分之一秒的时间，read 函数则立即返回，返回值为被读取的字节数；如果超时并且未读到数据，则 read 函数返回 0。
- MIN > 0 和 TIME > 0：当有 MIN 字节可读或两个输入字符之间的时间间隔超过 TIME 个十分之一秒时，read 函数才返回。因为在输入第一个字符之后系统才会启动定时器，所以在这种情况下，read 函数至少读取一个字节之后才返回。

严格来说，原始模式是一种特殊的非规范模式，在该模式下所有的输入数据以字节为单位被处理。在这个模式下，终端是不可回显的，而且所有特定的终端输入/输出控制处理不可用。通过调用 cfmakeraw 函数可以将终端设置为原始模式，而且该函数对应的操作代码如下：

```
termios_p->c_iflag &=~(IGNBRK | BRKINT | PARMRK | ISTRIP
                       | INLCR | IGNCR | ICRNL | IXON);
termios_p->c_oflag &=~OPOST;
termios_p->c_lflag &=~(ECHO | ECHONL | ICANON | ISIG | IEXTEN);
termios_p->c_cflag &=~(CSIZE | PARENB);
termios_p->c_cflag |= CS8;
```

4．串口的设置

在对串口进行读写操作之前应该先对其进行设置，包括波特率、校验位和停止位设置等，在 termios 结构体中最重要的成员是 c_cflag，通过对它赋值，用户可以设置波特率、字符大小、数据位、停止位、奇偶校验位和硬软流控等，Linux 系统提供了一系列的常量来对 c_cflag 进行操作，其中常用的常量的说明如表 12.2 所示。

表 12.2　c_cflag 对应的操作常量

CBAUD	波特率的位掩码
B0	0 波特率（放弃 DTR）
B1800	1800 波特率
B2400	2400 波特率
B4800	4800 波特率
B9600	9600 波特率
B19200	19 200 波特率
B38400	38 400 波特率
B57600	57 600 波特率
B115200	115 200 波特率
EXTA	外部时钟率
EXTB	外部时钟率
CSIZE	数据位的位掩码
CS5	5 个数据位
CS6	6 个数据位
CS7	7 个数据位
CS8	8 个数据位
CSTOPB	2 个停止位（不设则是 1 个停止位）
CREAD	接收使能
PARENB PARODD	校验位使能 使用奇校验而不使用偶校验
HUPCL	最后关闭时挂线（放弃 DTR）
CLOCAL	本地连接（不改变端口所有者）
CRTSCTS	硬件流控

注意：不能直接对 c_cflag 成员初始化，而要通过“与”、“或”操作使用其中的某些选项。

除了 c_cflag 之外，用户还可以通过对输入模式控制成员 c_iflag 的操作来控制对接收到的字符的处理，Linux 同样提供了如表 12.3 所示的操作常量。

表 12.3　c_iflag 对应的操作常量

INPCK	奇偶校验使能
IGNPAR	忽略奇偶校验错误
PARMRK	奇偶校验错误掩码
ISTRIP	裁减掉第 8 位
IXON	启动输出软件流控
IXOFF	启动输入软件流控
IXANY	输入任意字符可以重新启动输出（默认为输入起始字符才重启输出）
IGNBRK	忽略输入终止条件
BRKINT	当检测到输入终止条件时发送 SIGINT 信号
INLCR	将接收到的 NL（换行符）转换为 CR（回车符）
IGNCR	忽略接收到的 CR（回车符）
ICRNL	将接收到的 CR（回车符）转换为 NL（换行符）
IUCLC	将接收到的大写字符映射为小写字符
IMAXBEL	当输入队列满时响铃

c_oflag 用于对串口发送的字符进行控制，其对应的操作常量如表 12.4 所示。

表 12.4　c_oflag 对应的操作常量

OPOST	启用输出处理功能，如果不设置该标志，则其他标志都被忽略
OLCUC	将输出中的大写字符转换成小写字符
ONLCR	将输出中的换行符（‘\n’）转换成回车符（‘\r’）
ONOCR	如果当前列号为 0，则不输出回车符
OCRNL	将输出中的回车符（‘\r’）转换成换行符（‘\n’）
ONLRET	不输出回车符
OFILL	发送填充字符以提供延时
OFDEL	如果设置该标志，则表示填充字符为 DEL 字符，否则为 NUL 字符
NLDLY	换行延时掩码
CRDLY	回车延时掩码
TABDLY	制表符延时掩码
BSDLY	水平退格符延时掩码
VTDLY	垂直退格符延时掩码
FFLDY	换页符延时掩码

c_lfag 分量用于控制串口的本地数据处理和工作模式，其对应的操作常量如表 12.5 所示。

表 12.5　c_lfag 对应的操作常量

ISIG	若收到信号字符（INTR、QUIT 等），则会产生相应的信号
ICANON	启用规范模式
ECHO	启用本地回显功能
ECHOE	若设置 ICANON，则允许退格操作
ECHOK	若设置 ICANON，则 KILL 字符会删除当前行
ECHONL	若设置 ICANON，则允许回显换行符
ECHOCTL	若设置 ECHO，则控制字符（制表符、换行符等）会显示成“^X”，其中 X 的 ASCII 码等于给相应控制字符的 ASCII 码加上 0x40。例如，退格字符（0x08）会显示为“^H”（‘H’的 ASCII 码为 0x48）
ECHOPRT	若设置 ICANON 和 IECHO，则删除字符（退格符等）和被删除的字符都会被显示
ECHOKE	若设置 ICANON，则允许回显在 ECHOE 和 ECHOPRT 中设定的 KILL 字符
NOFLSH	在通常情况下，当接收到 INTR、QUIT 和 SUSP 控制字符时，会清空输入和输出队列。如果设置该标志，则所有的队列不会被清空
TOSTOP	若一个后台进程试图向它的控制终端进行写操作，则系统向该后台进程的进程组发送 SIGTTOU 信号。该信号通常终止进程的执行
IEXTEN	启用输入处理功能

cc 分量用于对串口的特殊操作进行控制，其对应的操作常量如表 12.6 所示。

表 12.6　cc 对应的操作常量

VINTR	中断控制字符，对应键为 Ctrl+C 组合键
VQUIT	退出操作符，对应键为 Ctrl+Z 组合键
VERASE	删除操作符，对应键为 Backspace（BS）
VKILL	删除行符，对应键为 Ctrl+U 组合键
VEOF	文件结尾符，对应键为 Ctrl+D 组合键
VEOL	附加行结尾符，对应键为 Carriage return（CR）
VEOL2	第二行结尾符，对应键为 Line feed（LF）
VMIN	指定最少读取的字符数
VTIME	指定读取的每个字符之间的超时时间

5．串口的初始化流程和配置函数 set_com_config 实例

在 Linux 下对串口的操作流程说明如下。

（1）调用函数 tcgetattr 来测试串口是否可用并保存串口的原始设置，该函数的标准调用格式说明如下。如果调用成功函数返回值为 0，同时得到 fd 指向的终端的配置参数，并将它们保存于 termios 结构变量 old_cfg 中；如果调用失败则函数返回值为-1。

```
#include<termios.h>
int tcgetattr(int fd, struct termios *termios_p);
```

（2）通过对 c_cflag 分量的设置（常量为 CLOCAL|CREAD）实现本地连接和接收使能，然后调用 cfmakeraw 函数把串口设置为原始模式。

（3）调用 cfsetispeed 和 cfsetospeed 来对串口的输入波特率和输出波特率进行设置，这

两个函数的标准调用格式说明如下，通常来说输入波特率和输出波特率需要设置为相同的。

```
#include<termios.h>
int cfsetospeed(struct termios *termptr, speed_t speed);
int cfsetispeed(struct termios *termptr, speed_t speed);
```

其中 struct termios *termptr 是指向 termios 结构的指针；而 speed_t speed 是需要设置的波特率，如果调用函数成功则返回 0，调用该函数失败则返回−1。

（4）通过对 c_cflag 的位掩码设置来设置字符的大小，即每个字节中存在几位数据。

（5）通过对 c_cflag 和 c_iflag 位的操作来设置奇偶校验位，其对应的操作代码如下：

```
.c_cflag |= (PARODD | PARENB);
.c_iflag |= INPCK;
//以上是奇校验的操作代码
.c_cflag |= PARENB;
.c_cflag &= ～PARODD;    //清除偶校验标志，则配置为奇校验
.c_iflag |= INPCK;
//以上是偶校验的操作代码
```

（6）通过对 c_cflag 分量的操作来设置停止位，当停止位为 1 位时清除 c_cflag 中的 CSTOPB，当停止位为 2 位时激活 CSTOPB，其对应的操作代码如下：

```
.c_cflag &=～CSTOPB;    /* 将停止位设置为一比特 */
.c_cflag |=CSTOPB;      /* 将停止位设置为两比特 */
```

（7）通过 c_cc 分量的设置来修改字符缓冲区和等待时间，如果缓冲区大小设置为 0，则在串口在读到 1 字节之后立即返回，其对应的操作代码如下：

```
.c_cc[VTIME] = 0;
.c_cc[VMIN] = 0;
```

（8）使用 tcdrain 系列函数清理当前串口的缓冲区，tcdrain 系列函数的标准调用格式说明如下：

```
#include<termios.h>
int tcdrain(int fd);
int tcflow(int fd, int action) ;
int tcflush(int fd, int queue_selector);
```

tcdrain 函数用于使程序阻塞，直到输出缓冲区的数据全部发送完毕，其中 fd 参数为串口的文件描述符。

tcflow 函数用于暂停或重新开始输出，其中 fd 参数为串口的文件描述符，action 参数有如下四种可能。

- TCOOF：输出被挂起。
- TCCON：重新启动以前被挂起的操作。
- TCIOFF：系统发送一个 STOP 字符以使得终端设备暂停发送数据。

- TCION：系统发送一个 START 字符以使得终端设备恢复发送数据。

tcflush 函数用于清空输入/输出缓冲区，其参数 queue_selector 用于选择对缓冲区的处理方式，有如下三种取值可能。

- TCIFLUSH：对接收到而未被读取的数据进行清空处理。
- TCOFLUSH：对尚未传送成功的输出数据进行清空处理。
- TCIOFLUSH：包括前两种功能，即对尚未处理的输入输出数据进行清空处理。

这三个函数如果调用成功则返回 0，调用失败则返回−1。

（9）调用 tcsetattr 函数激活当前的串口配置，该函数的标准调用格式说明如下：

```
#include<termios.h>
tcsetattr(int fd, int optional_actions, const struct termios *termios_p);
```

其中参数 termios_p 是 termios 类型的新配置变量，参数 optional_actions 可能的取值有以下 3 种。

- TCSANOW：配置的修改立即生效。
- TCSADRAIN：配置的修改在所有写入 fd 的输出都传输完毕之后生效。
- TCSAFLUSH：所有已接受但未读入的输入都将在修改生效之前被丢弃。

当配置完成之后即可按照文件的操作方式使用 open 函数、write 函数和 read 函数对串口进行操作，这些函数和普通的读写操作略有差别。例 12.5 给出了一个完整的对串口进行初始化操作的函数 set_com_config 的实例代码。

【例 12.5】 串口初始化配置函数 set_com_config。

```
int set_com_config(int fd,int baud_rate,
                                int data_bits, char parity, int stop_bits)
{
        struct termios new_cfg,old_cfg;
        int speed;

        /*保存并测试现有串口参数设置，在这里如果串口号等出错，会有相关的出错信息*/
        if   (tcgetattr(fd, &old_cfg)   !=   0)
        {
                perror("tcgetattr");
                return −1;
        }
        /*  设置字符大小*/
        new_cfg = old_cfg;
        cfmakeraw(&new_cfg); /*  配置为原始模式  */
        new_cfg.c_cflag &=  ～CSIZE;
        /*设置波特率*/
        switch (baud_rate)
        {
                case 2400:
                {
```

```
                speed = B2400;
        }
        break;
        case 4800:
        {
                speed = B4800;
        }
        break;
        case 9600:
       {
                speed = B9600;
        }
        break;
        case 19200:
        {
            speed = B19200;
        }
        break;
        case 38400:
        {
              speed = B38400;
        }
        break;

        default:
        case 115200:
        {
              speed = B115200;
        }
        break;
}
cfsetispeed(&new_cfg, speed);
cfsetospeed(&new_cfg, speed);

/*设置停止位*/
switch (data_bits)
{
        case 7:
        {
                new_cfg.c_cflag |= CS7;
        }
        break;
        default:
        case 8:
        {
                new_cfg.c_cflag |= CS8;
```

```
        }
        break;
    }
    /*设置奇偶校验位*/
    switch (parity)
    {
        default:
        case 'n':
        case 'N':
        {
            new_cfg.c_cflag &= ～PARENB;
            new_cfg.c_iflag &= ～INPCK;
        }
        break;
        case 'o':
        case 'O':
        {
            new_cfg.c_cflag |= (PARODD | PARENB);
            new_cfg.c_iflag |= INPCK;
        }
        break;
        case 'e':
        case 'E':
        {
            new_cfg.c_cflag |= PARENB;
            new_cfg.c_cflag &= ～PARODD;
            new_cfg.c_iflag |= INPCK;
        }
        break;

        case 's':   /*as no parity*/
        case 'S':
        {
        new_cfg.c_cflag &= ～PARENB;
        new_cfg.c_cflag &= ～CSTOPB;
        }
        break;
    }
    /*设置停止位*/
    switch (stop_bits)
    {
        default:
        case 1:
        {
            new_cfg.c_cflag &=～CSTOPB;
        }
```

```
                break;

                case 2:
                {
                    new_cfg.c_cflag |= CSTOPB;
                }
        }
        /*设置等待时间和最小接收字符*/
        new_cfg.c_cc[VTIME]= 0;
        new_cfg.c_cc[VMIN] = 1;

        /*处理未接收字符*/
        tcflush(fd, TCIFLUSH);
        /*激活新配置*/
        if ((tcsetattr(fd, TCSANOW, &new_cfg)) != 0)
        {
            perror("tcsetattr");
            return -1;
        }
        return 0;
    }
```

6．串口的打开操作和 open_port 函数实例

在完成串口的配置之后即可以对串口进行打开操作，其详细操作流程说明如下。

（1）打开串口，和文件一样，在使用串口之前必须使用 open 函数来打开串口，其标准调用格式如下，其中使用了 O_NOCTTY 和 O_NDELAY 参数。

```
fd = open( "/dev/ttyS0", O_RDWR|O_NOCTTY|O_NDELAY);
```

- O_NOCTTY 标志用于通知 Linux 系统，该参数不会使打开的文件成为这个进程的控制终端。如果没有指定这个标志，那么任何一个输入（如键盘中止信号等）都将影响用户的进程。
- O_NDELAY 标志通知 Linux 系统，这个程序不关心 DCD 信号线所处的状态（端口的另一端是否激活或停止）。如果用户指定了这个标志，则进程将会一直处于睡眠状态，直到 DCD 信号线被激活。

（2）打开串口之后应该调用 fcntl 函数将串口的状态设置为阻塞以等待数据输入，其标准调用格式如下：

```
fcntl(fd, F_SETFL, 0);
```

（3）如果对已经打开的串口状态不放心，可以使用 isatty 函数来测试串口是否正确打开，其标准调用格式说明如下，参数 STDIN_FINENO 为串口对应的文件描述符，如果串口打开成功会返回 0，否则返回−1。

```
#include<termios.h>
```

```
isatty(STDIN_FILENO);
```

此时一个串口就已经成功打开了，接下来就可以对这个串口进行读和写操作了。例 12.6 给出了一个完整的串口打开操作函数 open_port 的实例代码，其充分考虑了打开串口过程中的各种状态。

【例 12.6】 串口打开操作函数 open_ port。

```
int open_port(int com_port)
{
    int fd;
#if (COM_TYPE == GNR_COM)   /* 使用普通串口 */
    char *dev[] = {"/dev/ttyS0", "/dev/ttyS1", "/dev/ttyS2"};
#else /* 使用 USB 转串口 */
    char *dev[] = {"/dev/ttyUSB0", "/dev/ttyUSB1", "/dev/ttyUSB2"};
#endif
    if ((com_port < 0) || (com_port > MAX_COM_NUM))
    {
        return -1;
    }
    /* 打开串口 */
    fd = open(dev[com_port -1], O_RDWR|O_NOCTTY|O_NDELAY);
    if (fd < 0)
    {
            perror("open serial port");
            return(-1);
    }

    /*恢复串口为阻塞状态*/
    if (fcntl(fd, F_SETFL, 0) < 0)
    {
        perror("fcntl F_SETFL\n");
    }

    /*测试是否为终端设备*/
    if (isatty(STDIN_FILENO) == 0)
    {
        perror("standard input is not a terminal device");
    }
    return fd;
}
```

（4）此时可以和对普通文件操作一样使用 read 函数和 write 函数对串口进行读写操作。

12.2.3 实例的综合

例 12.7 和例 12.8 分别是 PC 端串口数据发送和嵌入式系统串口数据接收端的两个应用

程序代码，其中调用了例 12.5 和例 12.6 中提供的串口设置打开函数。

【例 12.7】 PC 端数据发送。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
int main(int argc,char *argv[])
{
    int fd;
    char buff[BUFFER_SIZE];
    if((fd = open_port(HOST_COM_PORT)) < 0) /*  打开串口  */
    {
        perror("open_port");
        return 1;
    }

    if(set_com_config(fd, 115200, 8, 'N', 1) < 0) /*  配置串口  */
    {
        perror("set_com_config");
        return 1;
    }
    do
    {
        printf("Input some words(enter 'quit' to exit):");
        memset(buff, 0, BUFFER_SIZE);
        if (fgets(buff, BUFFER_SIZE, stdin) == NULL)
        {
            perror("fgets");
            break;
        }
        write(fd, buff, strlen(buff));
    } while(strncmp(buff, "quit", 4));
    close(fd);
    return 0;
}
```

【例 12.8】 嵌入式系统数据接收。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
```

```
#include <errno.h>
int main(int argc,char argv[])
{
        int fd;
        char buff[BUFFER_SIZE];

        if((fd = open_port(TARGET_COM_PORT)) < 0) /*  打开串口  */
        {
                perror("open_port");
                return 1;
        }

        if(set_com_config(fd, 115200, 8, 'N', 1) < 0) /*  配置串口  */
        {
                perror("set_com_config");
                return 1;
        }

        do
        {
                memset(buff, 0, BUFFER_SIZE);
                if (read(fd, buff, BUFFER_SIZE) > 0)
                {
                        printf("The received words are : %s", buff);
                }
        } while(strncmp(buff, "quit", 4));
        close(fd);
        return 0;
}
```

12.3 【应用实例】——设计守护进程

12.3.1 实例的需求说明和分析

守护进程，也就是通常所说的 Daemon 进程，是 Linux 中的后台服务进程。它是一个生存期较长的进程，通常独立于控制终端并周期性地执行某种任务或等待处理某些事件。守护进程常常在系统引导载入时启动，在系统关闭时终止。Linux 有很多系统服务，大多数服务都是通过守护进程实现的；守护进程还能完成许多系统任务，如作业规划进程 crond、打印进程 lqd 等（这里的结尾字母 d 就是 Daemon 的意思）。

在 Linux 中，每一个系统与用户进行交流的界面称为终端，每一个从此终端开始运行的进程都会依附于这个终端，这个终端称为这些进程的控制终端，当控制终端被关闭时，相应的进程就会自动关闭。但是守护进程能够突破这种限制，它从被执行开始运转，直到整个系

统关闭时才会退出。如果想让某个进程不因为用户、终端或其他变化而受到影响，就必须把这个进程变成一个守护进程。可见，守护进程是非常重要的。

本应用设计了一个守护进程，该守护进程每隔 10s 向日志文件/tmp/daemon.log 写入一个字符串。

12.3.2　实例的基础设计

编写守护进程看似复杂，但实际上也是遵循特定的流程，该流程如图 12.6 所示。

1．创建子进程，父进程退出

这是编写守护进程的第一步。由于守护进程是脱离控制终端的，因此，完成第一步后就会在 shell 终端里造成一种程序已经运行完毕的假象。之后的所有工作都在子进程中完成，而用户在 shell 终端里则可以执行其他命令，从而在形式上做到了与控制终端的脱离。由于父进程在创建了子进程之后退出，会造成子进程没有父进程，从而变成一个孤儿进程。在 Linux 中，每当系统发现一个孤儿进程，就会自动由 1 号进程（也就是 init 进程）收养它，这样，原先的子进程就会变成 init 进程的子进程了，其代码如下：

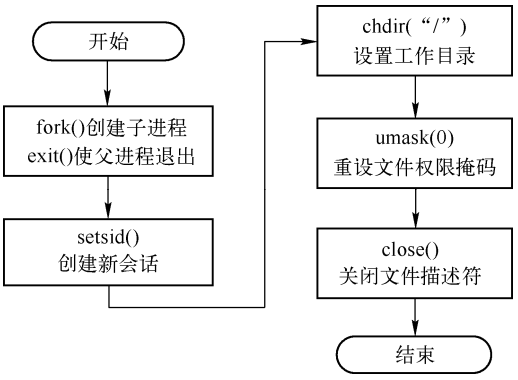


图 12.6　创建守护进程流程图

```
pid = fork();
if (pid > 0)
{
    exit(0); /*父进程退出*/
}
```

2．在子进程中创建新会话

在子进程中可以使用进程系统函数 setsid 来创建一个新的会话，并担任该会话组的组长，其有三个作用：

- 让进程摆脱原会话的控制；
- 让进程摆脱原进程组的控制；
- 让进程摆脱原控制终端的控制。

调用了 fork 函数创建子进程后再令父进程退出。由于在调用 fork 函数时，子进程全盘复制了父进程的会话期、进程组和控制终端等，虽然父进程退出了，但原先的会话期、进程组和控制终端等并没有改变，因此，还不是真正意义上的独立，而 setsid 函数能够使进程完全独立出来，从而脱离所有其他进程的控制。

setsid 函数的标准调用格式说明如下，如果成功则返回该进程组 ID，如果出错则返回 1。

```
#include <sys/types.h>
#include <unistd.h>
pid_t setsid(void)
```

3．改变当前目录为根目录

使用 fork 创建的子进程继承父进程的当前工作目录。由于在进程运行过程中，当前目录所在的文件系统（如“/mnt/usb”等）是不能卸载的，这对以后的使用会造成诸多麻烦（如系统由于某种原因要进入单用户模式）。因此，通常的做法是让“/”作为守护进程的当前工作目录，这样就可以避免上述问题。当然，如果有特殊需要，也可以把当前工作目录换成其他路径，如/tmp，改变工作目录的常见函数是 chdir。

4．重设文件权限掩码

文件权限掩码是指屏蔽掉文件权限中的对应位。例如，有一个文件权限掩码是 050，它就屏蔽了文件组拥有者的可读与可执行权限。由于使用 fork 函数新建的子进程继承了父进程的文件权限掩码，这就给该子进程使用文件带来了诸多麻烦。因此，把文件权限掩码设置为 0，可以大大增强该守护进程的灵活性。设置文件权限掩码的函数是 umask。在这里，通常的使用方法为 umask(0)。

5．关闭文件描述符

同文件权限掩码一样，用 fork 函数新建的子进程会从父进程那里继承一些已经打开了的文件。这些被打开的文件可能永远不会被守护进程读或写，但它们一样消耗系统资源，而且可能导致所在的文件系统无法被卸载。

在上面的第二步之后，守护进程已经与所属的控制终端失去了联系。因此从终端输入的字符不可能达到守护进程，守护进程中用常规方法（如 printf()）输出的字符也不可能在终端上显示出来。所以，文件描述符为 0、1 和 2 的 3 个文件（常说的输入、输出和报错这 3 个文件）已经失去了存在的价值，也应被关闭。通常按如下方式关闭文件描述符：

```
for(i = 0; i < MAXFILE; i++)
{
        close(i);
}
```

12.3.3 实例的综合

例 12.9 为守护进程的完整实例，该实例首先按照 12.3.2 节介绍的创建流程建立了一个守护进程，然后让该守护进程每隔 10s 向日志文件/tmp/daemon.log 写入一个字符串。

【例 12.9】 设计守护进程。

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<fcntl.h>
#include<sys/types.h>
#include<unistd.h>
#include<sys/wait.h>
int main()
{
    pid_t pid;
```

```
    int     i, fd;
    char    *buf = "This is a Daemon\n";

    pid = fork(); /*  第一步  */
    if (pid < 0)
    {
        printf("Error fork\n");
        exit(1);
    }
    else if (pid > 0)
    {
        exit(0); /*  父进程推出  */
    }

    setsid(); /*第二步*/
    chdir("/"); /*第三步*/
    umask(0); /*第四步*/
    for(i = 0; i < getdtablesize(); i++) /*第五步*/
    {
        close(i);
    }
    /*这时创建完守护进程，以下开始正式进入守护进程工作*/
    while(1)
    {
        if ((fd = open("/tmp/daemon.log",
                        O_CREAT|O_WRONLY|O_APPEND, 0600)) < 0)
        {
            printf("Open file error\n");
            exit(1);
        }
        write(fd, buf, strlen(buf) + 1);
        close(fd);
        sleep(10);
    }
    exit(0);
}
```

12.4　【应用实例】——设计生产者-消费者模型

“生产者-消费者”问题是一个著名的同时性编程问题的集合，解决该问题需要熟练掌握用信号量处理线程间的同步和互斥问题。

12.4.1　实例的需求说明和分析

“生产者-消费者”问题描述如下：有一个有限缓冲区和两个线程——生产者和消费者。它们分别不停地把产品放入缓冲区和从缓冲区中拿走产品。一个生产者在缓冲区满的时候必须等待，一个消费者在缓冲区空的时候也必须等待。另外，因为缓冲区是临界资源，所

以生产者和消费者之间必须互斥执行。它们之间的关系如图 12.7 所示。

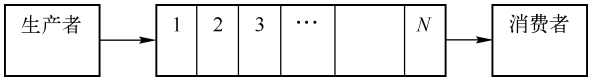


图 12.7　生产者-消费者问题描述

这里要求使用命名管道来模拟有限缓冲区，并且使用信号量来解决“生产者-消费者”问题中的同步和互斥问题。

12.4.2　实例的基础设计

在本实例中使用了 3 个信号量，其中信号量 avail 和 full 分别用于解决生产者和消费者线程之间的同步问题，mutex 用于这两个线程之间的互斥问题。其中 avail 表示有界缓冲区中的空单元数，初始值为 N；full 表示有界缓冲区中非空单元数，初始值为 0；mutex 是互斥信号量，初始值为 1，其流程如图 12.8 所示。

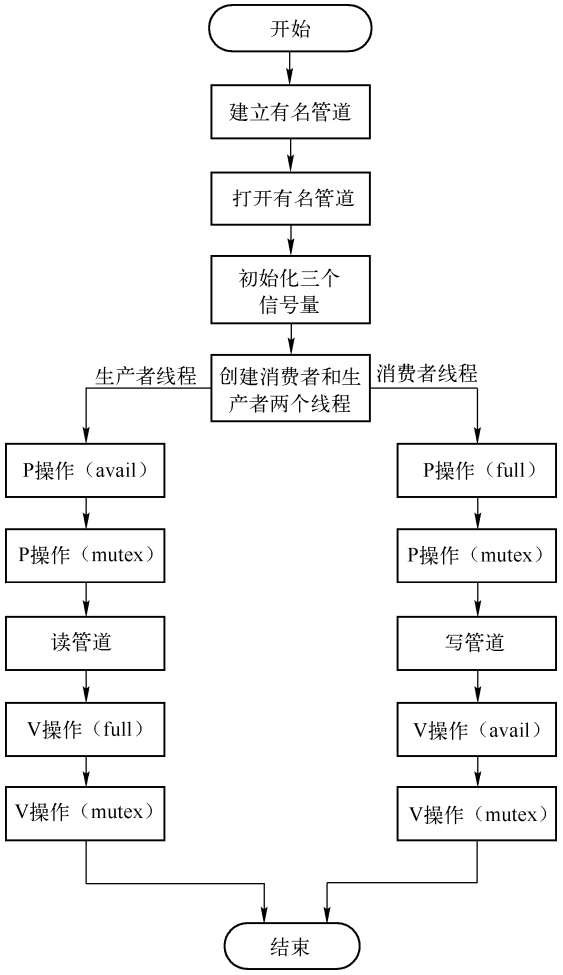


图 12.8 “生产者-消费者”实验流程图

注意：关于管道和信号量的相关知识介绍可以参考第 10 章。

生产者和消费者模型的应用代码如例 12.10 所示，其采用的有界缓冲区拥有 3 个单元，每个单元为 5 字节。为了尽量体现每个信号量的意义，在程序中生产过程和消费过程是随机（采取 0～5s 的随机时间间隔）进行的，而且生产者的速度比消费者的速度平均快两倍左右（这种关系可以相反）。生产者一次生产一个单元的产品（放入“hello”字符串），消费者一次消费一个单元的产品。

【例 12.10】 生产者和消费者模型的实现。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <pthread.h>
#include <errno.h>
#include <semaphore.h>
#include <sys/ipc.h>
#define MYFIFO                  "myfifo"          /* 缓冲区有名管道的名字 */
#define BUFFER_SIZE             3                 /* 缓冲区的单元数 */
#define UNIT_SIZE               5                 /* 每个单元的大小 */
#define RUN_TIME                30                /* 运行时间 */
#define DELAY_TIME_LEVELS       5.0               /* 周期的最大值 */
int fd;
time_t end_time;
sem_t mutex, full, avail;                         /* 3 个信号量 */
/*生产者线程*/
void *producer(void *arg)
{
    int real_write;
    int delay_time = 0;

    while(time(NULL) < end_time)
    {
        delay_time = (int)(rand() * DELAY_TIME_LEVELS/(RAND_MAX) / 2.0) + 1;
        sleep(delay_time);
        /*P 操作信号量 avail 和 mutex*/
        sem_wait(&avail);
        sem_wait(&mutex);
        printf("\nProducer: delay = %d\n", delay_time);
        /*生产者写入数据*/
        if ((real_write = write(fd, "hello", UNIT_SIZE)) == -1)
        {
            if(errno == EAGAIN)
            {
                printf("The FIFO has not been read yet.Please try later\n");
```

```
                }
            }
            else
            {
                printf("Write %d to the FIFO\n", real_write);
            }

            /*V 操作信号量 full 和 mutex*/
            sem_post(&full);
            sem_post(&mutex);
        }
        pthread_exit(NULL);
}
/*  消费者线程*/
void *customer(void *arg)
{
        unsigned char read_buffer[UNIT_SIZE];
        int real_read;
        int delay_time;

        while(time(NULL) < end_time)
        {
            delay_time = (int)(rand() * DELAY_TIME_LEVELS/(RAND_MAX)) + 1;
            sleep(delay_time);

            /*P 操作信号量 full 和 mutex*/
            sem_wait(&full);
            sem_wait(&mutex);
            memset(read_buffer, 0, UNIT_SIZE);
            printf("\nCustomer: delay = %d\n", delay_time);

            if ((real_read = read(fd, read_buffer, UNIT_SIZE)) == -1)
            {
                if (errno == EAGAIN)
                {
                    printf("No data yet\n");
                }
            }
            printf("Read %s from FIFO\n", read_buffer);
            /*V 操作信号量 avail 和 mutex*/
            sem_post(&avail);
            sem_post(&mutex);
        }
        pthread_exit(NULL);
}

int main()
{
        pthread_t thrd_prd_id,thrd_cst_id;
```

```
    pthread_t mon_th_id;
    int ret;

    srand(time(NULL));
    end_time = time(NULL) + RUN_TIME;
    /*创建有名管道*/
    if((mkfifo(MYFIFO, O_CREAT|O_EXCL) < 0) && (errno != EEXIST))
    {
        printf("Cannot create fifo\n");
        return errno;
    }
    /*打开管道*/
    fd = open(MYFIFO, O_RDWR);
    if (fd == -1)
    {
        printf("Open fifo error\n");
        return fd;
    }
    /*初始化互斥信号量为 1*/
    ret = sem_init(&mutex, 0, 1);
    /*初始化 avail 信号量为 N*/
    ret += sem_init(&avail, 0, BUFFER_SIZE);
    /*初始化 full 信号量为 0*/
    ret += sem_init(&full, 0, 0);
    if (ret != 0)
    {
        printf("Any semaphore initialization failed\n");
        return ret;
    }
    /*创建两个线程*/
    ret = pthread_create(&thrd_prd_id, NULL, producer, NULL);
    if (ret != 0)
    {
        printf("Create producer thread error\n");
        return ret;
    }
    ret = pthread_create(&thrd_cst_id, NULL, customer, NULL);
    if(ret != 0)
    {
        printf("Create customer thread error\n");
        return ret;
    }
    pthread_join(thrd_prd_id, NULL);
    pthread_join(thrd_cst_id, NULL);
    close(fd);
    unlink(MYFIFO);
    return 0;
}
```

12.5 【应用实例】——从网络服务器获取当前时间信息

12.5.1 实例的需求说明和分析

Network Time Protocol（NTP）是用来使计算机时间同步化的一种协议，它可以使计算机对其服务器或时钟源（如石英钟、GPS 等）做同步化，它可以提供高精确度的时间校正（LAN 上与标准时间差小于 1ms，WAN 上为几十毫秒），且可用加密确认的方式来防止恶意的协议攻击。

本实例是一个嵌入式系统从 NTP 服务器获取当前准确时间信息的应用，由于嵌入式系统往往没有实时时钟，不能保存时间信息，所以每次启动时从网络获取当前时间非常重要。

12.5.2 实例的基础设计

NTP 提供准确时间，首先要有准确的时间来源，这一时间应该是国际标准时间 UTC。NTP 获得 UTC 的时间来源可以是原子钟、天文台、卫星，也可以从 Internet 上获取。这样就有了准确而可靠的时间源。时间按 NTP 服务器的等级传播。按照距离外部 UTC 源的远近将所有服务器归入不同的 Stratum（层）中。Stratum-1 在顶层，有外部 UTC 接入，而 Stratum-2 则从 Stratum-1 获取时间，Stratum-3 从 Stratum-2 获取时间，以此类推，但 Stratum 层的总数限制在 15 以内。所有这些服务器在逻辑上形成阶梯式的架构并相互连接，而 Stratum-1 的时间服务器是整个系统的基础。

进行网络协议实现时最重要的是了解协议数据格式。NTP 数据包有 48 字节，其中 NTP 包头为 16 字节，时间戳为 32 字节。其协议格式如图 12.9 所示，各个部分说明如下。

2	5	8	16	24	32bit
LI	VN	Mode	Stratum	Poll	Precision
Root Delay					
Root Dispersion					
Reference Identifier					
Reference timestamp(64)					
Originate Timestamp(64)					
Receive Timestamp(64)					
Transmit Timestamp(64)					
Key Identifier(Optional)(32)					
Message digest(Optional)(128)					

图 12.9 NTP 协议的组成

LI：跳跃指示器，警告在当月最后一天的最终时刻插入的迫近闰秒（闰秒）。

VN：版本号。

Mode：工作模式。该字段包括以下值：0——预留；1——对称行为；3——客户机；4——服务器；5——广播；6——NTP 控制信息。NTP 协议具有 3 种工作模式，分别为主/

被动对称模式、客户/服务器模式、广播模式。在主/被动对称模式中，有一对一的连接，双方均可同步对方或被对方同步，先发出申请建立连接的一方工作在主动模式下，另一方工作在被动模式下；客户/服务器模式与主/被动模式基本相同，唯一区别在于客户方可被服务器同步，但服务器不能被客户同步；在广播模式中，有一对多的连接，服务器不论客户工作在何种模式下都会主动发出时间信息，客户根据此信息调整自己的时间。

- Stratum：对本地时钟级别的整体识别。
- Poll：有符号整数表示连续信息间的最大间隔。
- Precision：有符号整数表示本地时钟精确度。
- Root Delay：表示到达主参考源的一次往复的总延迟，它是有 15～16 位小数部分的符号定点小数。
- Root Dispersion：表示一次到达主参考源的标准误差，它是有 15～16 位小数部分的无符号定点小数。
- Reference Identifier：识别特殊参考源。
- Originate Timestamp：这是向服务器请求分离客户机的时间，采用 64 位时标格式。
- Receive Timestamp：这是向服务器请求到达客户机的时间，采用 64 位时标格式。
- Transmit Timestamp：这是向客户机答复分离服务器的时间，采用 64 位时标格式。
- Authenticator（Optional）：当实现了 NTP 认证模式时，主要标识符和信息数字域就包括已定义的信息认证代码（MAC）信息。

12.5.3　实例的综合

例 12.11 是使用 NTP 协议来获取当前网络时间的实例。由于 NTP 协议中涉及比较多的时间相关的操作，在本应用中仅要求实现 NTP 协议客户端部分的网络通信模块，也就是构造 NTP 协议字段进行发送和接收，最后与时间相关的操作不需要进行处理。NTP 协议作为 OSI 参考模型的高层协议比较适合采用 UDP 传输协议进行数据传输，专用端口号为 123。在实验中，以国家授时中心服务器（IP 地址为 202.72.145.44）为 NTP（网络时间）服务器，其流程如图 12.10 所示。

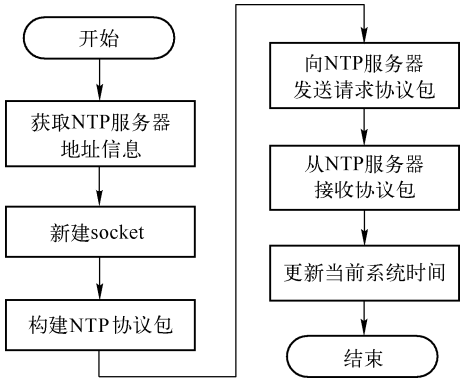


图 12.10　从网络服务器获取当前时间

应用代码构建了如下函数用于对网络时间进行操作。

- construct_packet：构建 NTP 协议包。
- get_ntp_time：获取 NTP 时间。
- set_local_time：修改本地时间。

应用代码首先建立和目标主机的连接，然后调用 get_ntp_time 获取 NTP 的时间，然后调用 set_local_time 来修改本地时间。

【例 12.11】 从网络服务器获取当前时间信息。

```
#include <sys/socket.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/un.h>
#include <sys/time.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <netinet/in.h>
#include <string.h>
#include <netdb.h>

#define NTP_PORT        123                 /*NTP 专用端口号字符串*/
#define TIME_PORT          37                  /* TIME/UDP 端口号  */
#define NTP_SERVER_IP      "61.135.250.78"   /*国家授时中心 IP*/
#define NTP_PORT_STR       "123"             /*NTP 专用端口号字符串*/
#define NTPV1              "NTP/V1"          /*协议及其版本号*/
#define NTPV2              "NTP/V2"
#define NTPV3              "NTP/V3"
#define NTPV4              "NTP/V4"
#define TIME               "TIME/UDP"

#define NTP_PCK_LEN 48
#define LI 0
#define VN 3
#define MODE 3
#define STRATUM 0
#define POLL 4
#define PREC -6

#define JAN_1970    0x83aa7e80    /* 1900 年～1970 年之间的时间秒数  */
#define NTPFRAC(x)     (4294 * (x) + ((1981 * (x)) >> 11))
#define USEC(x)       (((x) >> 12) -759 * ((((x) >> 10) + 32768) >> 16))

typedef struct _ntp_time
```

```
{
    unsigned int coarse;
    unsigned int fine;
} ntp_time;

struct ntp_packet
{
    unsigned char leap_ver_mode;
    unsigned char startum;
    char poll;
    char precision;
    int   root_delay;
    int   root_dispersion;
    int reference_identifier;
    ntp_time reference_timestamp;
    ntp_time originage_timestamp;
    ntp_time receive_timestamp;
    ntp_time transmit_timestamp;
};

char protocol[32];
/*构建 NTP 协议包*/
int construct_packet(char *packet)
{
    char version = 1;
    long tmp_wrd;
    int port;
    time_t timer;
    strcpy(protocol, NTPV3);
    /*判断协议版本*/
    if(!strcmp(protocol, NTPV1)||!strcmp(protocol, NTPV2)
        ||!strcmp(protocol, NTPV3)||!strcmp(protocol, NTPV4))
    {
        memset(packet, 0, NTP_PCK_LEN);
        port = NTP_PORT;
        /*设置 16 字节的包头*/
        version = protocol[6] - 0x30;
        tmp_wrd = htonl((LI << 30)|(version << 27)
            |(MODE << 24)|(STRATUM << 16)|(POLL << 8)|(PREC & 0xff));
        memcpy(packet, &tmp_wrd, sizeof(tmp_wrd));

        /*设置 Root Delay、Root Dispersion 和 Reference Indentifier */
        tmp_wrd = htonl(1<<16);
        memcpy(&packet[4], &tmp_wrd, sizeof(tmp_wrd));
        memcpy(&packet[8], &tmp_wrd, sizeof(tmp_wrd));
        /*设置 Timestamp 部分*/
```

```
                time(&timer);
                /*设置 Transmit Timestamp coarse*/
                tmp_wrd = htonl(JAN_1970 + (long)timer);
                memcpy(&packet[40], &tmp_wrd, sizeof(tmp_wrd));
                /*设置 Transmit Timestamp fine*/
                tmp_wrd = htonl((long)NTPFRAC(timer));
                memcpy(&packet[44], &tmp_wrd, sizeof(tmp_wrd));
                return NTP_PCK_LEN;
        }
        else if (!strcmp(protocol, TIME))/* "TIME/UDP" */
        {
                port = TIME_PORT;
                memset(packet, 0, 4);
                return 4;
        }
        return 0;
}

/*获取 NTP 时间*/
int get_ntp_time(int sk, struct addrinfo *addr, struct ntp_packet *ret_time)
{
        fd_set pending_data;
        struct timeval block_time;
        char data[NTP_PCK_LEN * 8];
        int    packet_len, data_len = addr->ai_addrlen, count = 0, result, i, re;

       if (!(packet_len = construct_packet(data)))
        {
                return 0;
        }
        /*客户端给服务器端发送 NTP 协议数据包*/
        if ((result = sendto(sk, data,
                                packet_len, 0, addr->ai_addr, data_len)) < 0)
        {
                perror("sendto");
                return 0;
        }

        /*调用 select()函数，并设定超时时间为 1s*/
        FD_ZERO(&pending_data);
        FD_SET(sk, &pending_data);
        block_time.tv_sec=10;
        block_time.tv_usec=0;
        if (select(sk + 1, &pending_data, NULL, NULL, &block_time) > 0)
        {
                /*接收服务器端的信息*/
```

```
            if ((count = recvfrom(sk, data,
                                NTP_PCK_LEN * 8, 0, addr->ai_addr, &data_len)) < 0)
            {
                perror("recvfrom");
                return 0;
            }

           if (protocol == TIME)
            {
                memcpy(&ret_time->transmit_timestamp, data, 4);
                return 1;
            }
            else if (count < NTP_PCK_LEN)
            {
                return 0;
            }
            /* 设置接收 NTP 包的数据结构 */
            ret_time->leap_ver_mode = ntohl(data[0]);
            ret_time->startum = ntohl(data[1]);
            ret_time->poll = ntohl(data[2]);
            ret_time->precision = ntohl(data[3]);
            ret_time->root_delay = ntohl(*(int*)&(data[4]));
            ret_time->root_dispersion = ntohl(*(int*)&(data[8]));
            ret_time->reference_identifier = ntohl(*(int*)&(data[12]));
            ret_time->reference_timestamp.coarse = ntohl (*(int*)&(data[16]));
            ret_time->reference_timestamp.fine = ntohl(*(int*)&(data[20]));
            ret_time->originage_timestamp.coarse = ntohl(*(int*)&(data[24]));
            ret_time->originage_timestamp.fine = ntohl(*(int*)&(data[28]));
            ret_time->receive_timestamp.coarse = ntohl(*(int*)&(data[32]));
            ret_time->receive_timestamp.fine = ntohl(*(int*)&(data[36]));
            ret_time->transmit_timestamp.coarse = ntohl(*(int*)&(data[40]));
            ret_time->transmit_timestamp.fine = ntohl(*(int*)&(data[44]));
            return 1;
        } /* end of if select */
      return 0;
}

/* 修改本地时间 */
int set_local_time(struct ntp_packet * pnew_time_packet)
{
        struct timeval tv;
        tv.tv_sec = pnew_time_packet->transmit_timestamp.coarse - JAN_1970;
        tv.tv_usec = USEC(pnew_time_packet->transmit_timestamp.fine);
        return settimeofday(&tv, NULL);
}
```

```
int main()
{
    int sockfd, rc;
    struct addrinfo hints, *res = NULL;
    struct ntp_packet new_time_packet;

    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_DGRAM;
    hints.ai_protocol = IPPROTO_UDP;
    /*调用 getaddrinfo()函数，获取地址信息*/
    rc = getaddrinfo(NTP_SERVER_IP, NTP_PORT_STR, &hints, &res);
    if (rc != 0)
    {
        perror("getaddrinfo");
        return 1;
    }
    /* 创建套接字 */
    sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
    if (sockfd <0 )
    {
        perror("socket");
        return 1;
    }
    /*调用取得 NTP 时间的函数*/
    if (get_ntp_time(sockfd, res, &new_time_packet))
    {
        /*调整本地时间*/
        if (!set_local_time(&new_time_packet))
        {
            printf("NTP client success!\n");
        }
    }
    close(sockfd);
    return 0;
}
```

反侵权盗版声明