

# **LINUX** 应用编程

## (GEC 内部讲义 V1.0)

<b>第1章</b>	<b>LINUX 入门.....</b>	<b>6</b>
1.1	LINUX 系统简介.....	6
1.1.1	Linux 概述.....	6
1.1.2	Linux 发展过程.....	6
1.1.3	Linux 应用方向.....	7
1.1.4	Linux 版本.....	8
1.2	LINUX 系统的安装.....	9
1.2.1	硬件需求.....	9
1.2.2	安装准备.....	9
1.2.3	安装过程.....	9
1.3	LINUX 文件及目录.....	18
1.3.1	Linux 文件.....	18
1.3.2	Linux 文件系统.....	20
1.3.3	Linux 目录.....	20
	本章小结.....	21
<b>第2章</b>	<b>LINUX 基础命令.....</b>	<b>23</b>
2.1	文件相关命令.....	23
2.1.1	文件管理.....	24
2.1.2	文件处理.....	27
2.2	系统相关命令.....	31
2.2.1	系统信息查询.....	32
2.2.2	进程管理.....	35
2.2.3	用户管理.....	37
2.3	网络相关命令.....	38
2.4	压缩打包相关命令.....	40
2.5	其他命令.....	41
	本章小结.....	44
<b>第3章</b>	<b>LINUX 下的 C 编程环境.....</b>	<b>45</b>
3.1	LINUX 下 C 语言编程环境概述.....	45
3.2	编辑器.....	47
3.2.1	关于文本编辑器.....	47
3.2.2	vi 编辑器的使用方法.....	47
3.3	GCC 编译器.....	53

3.3.1	Gcc 编译流程.....	53
3.3.2	Gcc 编译选项.....	56
3.3.3	gcc 生成静态库和动态库.....	59
3.3.4	教你解决 GCC 错误.....	63
3.4	3.4 GDB 调试器.....	64
3.4.1	Gdb 使用流程.....	65
3.4.2	Gdb 基本命令.....	71
3.4.3	用 Gdb 调试有问题的程序.....	75
3.5	MAKE 工程管理器.....	77
3.5.1	为什么要学 Makefile.....	77
3.5.2	make 工作步骤.....	78
3.5.3	简单 Make 程序创建.....	78
3.5.4	Makefile 改进.....	81
3.5.5	万能 MAKEFILE 模板.....	84
	本章小结.....	90
<b>第 4 章</b>	<b>SHELL 编程.....</b>	<b>92</b>
4.1	为什么要学 SHELL.....	92
4.2	认识 SHELL.....	93
4.2.1	什么是 shell.....	93
4.2.2	Bash Shell.....	95
4.2.3	简单 shell 应用.....	96
4.2.4	Shell 常用命令.....	97
4.2.5	重定向与管道.....	103
4.3	SHELL 编程.....	107
4.3.1	shell 变量.....	107
4.3.2	shell 控制结构.....	117
4.3.3	其它结构.....	126
4.3.4	shell 函数.....	127
4.4	综合应用.....	131
	本章小结.....	139
<b>第 5 章</b>	<b>文件 IO 编程.....</b>	<b>141</b>
5.1	LINUX 文件结构.....	141
5.1.1	文件.....	141
5.1.2	文件描述符.....	141
5.2	系统调用与库函数.....	142
5.2.1	系统调用.....	142

5.2.2	库函数.....	142
5.3	文件 IO 基本操作.....	143
5.3.1	不带缓存的 IO 操作.....	143
5.3.2	带缓存的 IO 操作.....	147
5.4	文件 IO 高级操作.....	151
5.4.1	文件锁.....	151
5.4.2	多路复用.....	158
5.5	本章小结.....	164
5.6	综合实验：文件读写及上锁.....	164
5.7	思考练习.....	174
<b>第 6 章 进程.....</b>		<b>175</b>
6.1	LINUX 进程概述.....	175
6.1.1	程序与进程.....	175
6.1.2	进程结构.....	176
6.1.3	进程属性.....	177
6.1.4	进程管理.....	179
6.1.5	进程模式.....	180
6.2	LINUX 进程控制.....	180
6.2.1	fork 函数.....	180
6.2.2	函数族.....	182
6.2.3	exit 和 _exit 函数.....	187
6.2.4	wait 和 waitpid 函数.....	189
6.3	守护进程.....	192
6.3.1	守护进程概述.....	192
6.3.2	编写守护进程.....	192
6.4	本章小结.....	199
<b>第 7 章 LINUX 线程概述.....</b>		<b>200</b>
7.1	线程.....	200
7.1.1	线程概述.....	200
7.1.2	线程发展历程.....	201
7.2	LINUX 线程控制.....	201
7.2.1	线程基本操作.....	201
7.2.2	线程属性.....	204
7.2.3	互斥锁.....	209
7.2.4	信号量.....	213

本章小结 .....	220
<b>第 8 章 进程间通信 .....</b>	<b>221</b>
8.1 管道 .....	221
8.1.1 无名管道 .....	222
8.1.2 有名管道 .....	225
8.2 共享内存 .....	229
8.2.1 共享内存概述 .....	229
8.2.2 共享内存应用 .....	229
8.3 消息队列 .....	233
8.3.1 消息队列概述 .....	233
8.3.2 消息队列应用 .....	233
8.4 信号量 .....	237
8.4.1 信号量基本操作 .....	237
8.4.2 信号量应用实例 .....	238
本章小结 .....	242
<b>第 9 章 LINUX 网络编程 .....</b>	<b>243</b>
9.1 网络编程基础概念 .....	243
9.1.1 TCP/IP 基本概念 .....	243
9.1.2 IP 地址、端口与域名 .....	244
9.1.3 套接字 <i>socket</i> .....	244
9.1.4 套接字数据结构 .....	245
9.1.5 网络相关函数 .....	246
9.2 网络基础编程 .....	250
9.2.1 面向连接的套接字通信 ( <i>TCP</i> ) .....	251
9.2.2 无连接的套接字通信 ( <i>UDP</i> ) .....	258
9.3 网络高级编程 .....	263
9.4 本章小结 .....	272
9.5 课后练习 .....	272
<b>第 10 章 QT 编程基础 .....</b>	<b>273</b>
10.1 QT 介绍 .....	273
10.1.1 GUI 的作用 .....	273
10.1.2 QT 的特点 .....	274
10.1.3 QT 的安装 .....	275
10.2 DESIGNER 快速创建工程 .....	277
10.2.1 Designer 使用 .....	277



10.2.2	Desinger 创建 Hello World.....	279
10.2.3	Hello World 编译.....	280
10.2.4	Hello World 分析.....	284
10.3	QT 对话框的完善.....	288
10.3.1	QT 对话框的布局.....	288
10.3.2	QT 对话框的属性.....	289
10.3.3	QT 的控件.....	290
10.4	QT 的信号与槽.....	292
10.4.1	QT 的事件.....	292
10.4.2	QT 的信号.....	292
10.4.3	QT 的槽.....	293
10.4.4	信号与槽的关联.....	294
10.5	本章小结.....	295
10.6	信号与槽 FAQ.....	295
10.7	实践操作.....	297
<b>第 11 章 QT 的资源与技巧.....</b>		<b>298</b>
11.1	QT 的类.....	298
11.1.1	QT 的类的层次.....	298
11.1.2	QT 类的使用技巧.....	298
11.2	如何从参考文档获得帮助.....	305
11.3	本章小结.....	308

# 第 1 章Linux 入门

## 学习目标

- 独立安装 Linux 系统
- Linux 文件
- Linux 文件系统
- Linux 目录树结构

## 1.1 Linux 系统简介

### 1.1.1 Linux 概述

Linux 是一个多用户、多任务的类 UNIX 操作系统，它可免费使用和自由传播。Linux 系统最大的特色就是源代码完全开放，任何人在遵守 GPL 协议下，都有获得、修改和发布其代码的自由。Linux 其版权所有者是芬兰 linus Torvalds 等开发人员。

Linux 系统一直以系统的稳定性和强大的网络功能而著称，很多企业将 LINUX 作为其公司的服务器操作系统。随着 Linux 的发展，Linux 桌面应用越来越丰富，易用性越来越高，又以低廉的价格而给越来越多的用户所接受。

### 1.1.2 Linux 发展过程

Linux 起源于一个名为 linus 的芬兰大学生，当时他主修一门操作系统课程，而这门课程提供了一个用于教学科研的称为 Minix 操作系统（Minix 系统是由 Andrew Tannebaum 教授所开发）。linus 发现 Minix 系统的功能很不完善，就给 Minix 增加一些功能，这样，Linux 最早版本产生了。

1991 年 10 月，linus Torvalds 在 com.os.minix 发布了 Linux 第一个版本，当时 Linux 可运行 bash(GNU 的一个 UNIXshell 程序)和 GCC(GNU 的 C 编译器)。

1994 年 3 月，Linux1.0 版本发布，随后正式采用 GPL 协议。

1995 年，Linux 能在 Intel、Digital 以及 Sun SPARC 处理器上运行，用户数量超过 50 万。

1996 年 6 月，Linux2.0 版内核正式发布，此时内核有大约 40 万行代码。

1998 年 red hat 高级研发实验室成立, 同年 RedHat 5.0 获得了 InfoWorld 的操作系统奖项。4 月 Mozilla 代码发布, 成为 Linux 图形界面上的王牌浏览器。12 月, IBM 发布了适用于 Linux 的文件系统 AFS 3.5 以及 Jikes Java 编辑器和 Secure Mailer 及 DB2 测试版, IBM 的此番行为, 可以看作是与 Linux 羞答答地第一次亲密接触。

1999 年 IBM 与 Redhat 公司建立合作关系, 从而确保 Redhat 能在 IBM 生产的机器上正确运行。

2000 年 2 月 RedHat 发布了嵌入式 Linux 的开发环境, Linux 在嵌入式行业的潜力逐渐被发掘出来。

2001 年 Oracle 宣布可有条件索取 Oracle 9i 的 Linux 版本。8 月红色代码爆发, 引得许多站点纷纷从 windows 操作系统转向 Linux 操作系统。

2002 年 Linux 内核支持 64 位的计算机。

2003 年 NEC 宣布将在其手机中使用 Linux 操作系统, 代表着 Linux 成功进军手机领域。9 月中科红旗发布 Red Flag Server4 版本。

2004 年 6 月的统计报告显示在世界 500 强超级计算机系统中, 使用 Linux 操作系统的已经占到了 280 席, 抢占了原本属于各种 Unix 的份额。

### 1.1.3 Linux 应用方向

目前, Linux 系统主要被应用到: 桌面应用、服务器、软件开发和嵌入式领域。

#### 1. 桌面应用

随着 Linux 的发展, Linux 系统的桌面应用得到了很大的改进, 其系统集成办公、多媒体、开发和网络等功能。

#### 2. 服务器

Linux 操作系统的稳定性和安全性, 是许多企业选择其作为服务器的重要原因, 又因其是支持多平台的, 软件是开源免费的, 便得到越来越多的企业青睐。同时 Linux 能和其它操作系统如 unix、windows 共存, 这也奠定其作为服务器的重要原因。Linux 常用作以下服务器: web 服务器, DNS 服务器和文件与打印服务器。

#### 3. 软件开发

在安装系统时, 可以自行定制相关的开发工具。Linux 发展到今天, 已经拥有了很完善的开发环境。例如: gcc、gdb、make、Emacs 等

#### 4. 嵌入式领域

在嵌入式领域, 嵌入式 Linux 拥有别的嵌入式系统不代替的地位。它所拥有的优势有以下几点:

- (1) 低成本。Linux 源码可自由获取和修改, 这可大大的降低开发所需要的成本。



(2) 支持多硬件平台。Linux 已经被移植到多种硬件平台，这对于经费，时间受限制的研究与开发项目是很有吸引力。Linux 采用一个统一的框架对硬件进行管理，从一个硬件平台到另一个硬件平台的改动与上层应用无关。

(3) 内核可裁剪。Linux 很大特色就是内核模块化，用户可根据实际需要，而对其进行裁剪，从而得到所需要实现基本功能的最小的内核。

(4) 性能稳定。Linux 系统占用系统资源少，运行速度快。

### 1.1.4 Linux 版本

Linux 有两种版本，一种是内核版本，另一种是发行版本。

#### 1. 内核版本

内核版本指的是 Linux 的内核，是由 linux 等人在不断地开发和推出新的内核。其版本号命令是由三部分数字组成：

`x.y.z`

x 为主版本号。

y 次版本号，偶数代表稳定版本，奇数代表是测试版本。

z 对前版本的修改次数。

例如：2.6.12 表示对核心版本的第 12 次修改。

#### 2. 发行版本

发行版本指的是将 Linux 内核与各种应用软件和文档包装起来，并提供一些安装界面和系统设置与管理工具的软件包。目前常见的 Linux 发行版本如下：

##### (1) RedHat

RedHat 是目前世界上最资深的 Linux 和开放源代码提供商，同时也是最获认可的 Linux 品牌之一。RedHat 支持多硬件平台，拥有优秀的安装界面，独特的 RPM 升级方式。Redhat 自 9.0 以后，不再发布桌面版。

##### (2) Debian

Debian 最早由 Ian Murdock 于 1993 年创建。它的开发模式与 Linux 及其它开放性源代码操作系统的精神一样，都是由超过 800 位志愿者通过互连网合作开发而成的。一直以来，Debian 被认为是最正宗的 Linux 发行版本，而且它是完全免费的、高质量的与 unix 兼容的操作系统。Debian 系统分为 3 个版本：stable\testing 和 unstable. 其中 stable 是稳定版，testing 是测试版、unstable 是不稳定版。

##### (3) Ubuntu

Ubuntu 是一个相对较新的发行版。但是，它的出现改变了许多潜在用户对 Linux 安装和使用困难的看法，ubuntu 的安装方便、简单。Ubuntu 被称为对硬件支持最好、最全面的 Linux 发行版之一，许多在其它发行版上无法使用或者默认配置时无法使用的硬件在 ubuntu 上都可以使用。

##### (4) Gentoo

Gentoo 首个稳定版本发布于 2002 年。Gentoo 的主要特别是其可以高度的自定义：它是一个基于源代码的发行版。尽管安装时可以选择预先编译好的软件包，但是大部分使用 gentoo 的用户都选择自己手动编译。

#### (5) red flag

red flag 是由中科红旗软件技术有限公司开发研制，在中国 Linux 用户中占有一定的比例。Red flag 有红旗桌面版本和红旗服务器版本。

## 1.2 Linux 系统的安装

### 1.2.1 硬件需求

Linux 对硬件的需求很低。在字符方式下运行，那么一台 386 的计算机就可以用来安装 Linux 了；在图形界面下运行，那也只需要一台 16MB 内存，600MB 硬盘的计算机即可安装 Linux。这听起来比那些至少需要 256MB 内存，2.0GHz 的操作系统要好得多。

现在计算机行业的趋势是让用户购买硬件配置更高的计算机，而 Linux 却不受这个趋势的影响。虽然随着 Linux 的发展，在其上运行的软件越来越多，它所需要的配置也越来越高，但是用户是可以有选择地安装软件，从而节省资源。既可以运行在最新的 Pentium4 处理器上，也可以运行在 400MHz 的 pentium II 上，甚至如果用户需要，也可以在更低配置上运行（以字符方式运行）。由此可见 Linux 非常适合需求各异的嵌入式硬件平台。而且 Linux 可以很好地支持标准配件。如果用户的计算机是采用了标准配件，那么运行 Linux 就没有任何问题。

### 1.2.2 安装准备

在安装之前，首先应该知道以下与机器相关的信息。

#### (1) 机器的硬件信息

硬盘接口的类型和容量大小、内存的大小、鼠标类型（串口、PS/2、USB）、显卡的制造者和型号，显存有多大。

#### (2) 连接网络所用

连接上网所需的 IP 地址，子网掩码，网关，DNS、网卡的型号。

#### (3) 安装 Linux 所需的光盘

最后，应确认硬盘是否还有空间可以安装 Linux 系统，推荐硬盘需要 5G 以上，内存容量 256M 以上。

### 1.2.3 安装过程

首先把 BIOS 设置为从光盘引导，再把 LINUX 系统安装盘放入光驱，然后重启电脑，按如下步骤安装 Linux 系统：

#### 1. 启动界面

从光盘启动安装程序后，就会出现安装提示画面，如图 1-1 所示。



图 1-1 安装提示画面

安装界面上有 3 个选项供用户选择：

- (1) 按<Enter>键，直接进入图形模式（graphical mode）安装模式。
- (2) 在界面最后显示“boot:”之后输入：“Linux text”，然后按<Enter>键，以文本模式（text mode）安装。
- (3) 用下面以红色字体列出的功能键（function keys）可以获得更多信息的方式安装。

选择按 Enter 键就进入图形安装模式。

#### 2. 检测安装盘

在选择图形模式安装后，系统先检测计算机的各种硬件，接着出现询问是否测试 CD 媒体界面，如图 1-2 所示，由于测试时间很长，一般不会测试，选择 Skip 跳过。

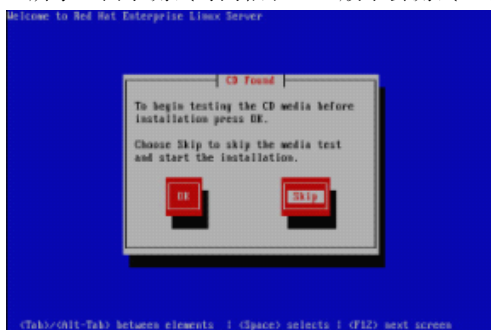


图 1-2 CD 媒体测试选择界面

### 3. 欢迎界面

弹出欢迎界面，如图所示，单击“Next”。



图 1 3 欢迎界面

### 4. 语言选择

弹出语言选择的界面，如图 1 4 所示，选择“简体中文”，单击“Next”。



图 1 4 语言选择的界面

### 5. 系统键盘的选择

弹出系统键盘选择界面，如图 1 5 所示，默认为“美国英语式”，无需修改，单击“下一步”。



图 1 5 键盘选择界面

6. 安装号码

弹出“安装号码”对话框，如图 1 6 所示，在安装号码文本框中输入相应的安装号码。如果没有，可以选择“跳过输入安装号码”，单击“确定”。



图 1 6 “安装号码”对话框

注：输入安装号码和跳过的区别在于，前者在安装过程中可能会添加其他额外的服务，而后者则只安装核心服务。

7. 磁盘分区设置

弹出“警告”对话框，如图 1 7 所示，选“是”。

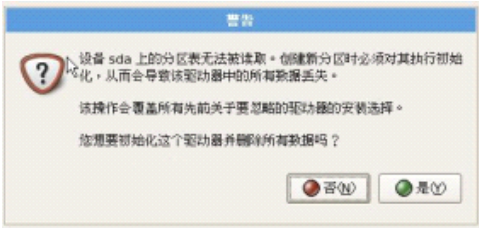


图 1 7 “警告”对话框

弹出“分区结构选择”界面，如图 1-8 所示。选择选“建立自定义分区结构”。单击“下一步”。



图 1-8 “分区结构选择”界面

## 8. 创建分区

弹出“新建分区”对话框，如图 1-9 所示。



图 1-9 “新建分区”对话框

执行“新建”，弹出“添加分区”对话框，如所示，在“挂载点”列表框选择“/boot”，在“指定空间大小”文本框中输入“100”，单击“确定”。

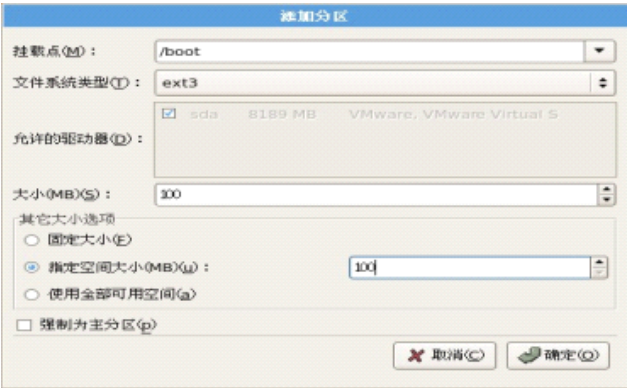


图 1 10 “添加分区”对话框

同理，再执行“新建”，在“文件系统类型”列表框中选择“Swap”，在“指定空间大小”文本框中输入“512”，单击“确定”。

执行“新建”，选“挂载点”列表框中选择“/”，选中“使用全部可用空间”单选框，注意在这块硬盘上我们只用来装 Red Hat Enterprise Linux 5，如果您的硬盘上还有其它的系统，您可以根据自己的需要进行合理的分配空间，单击“确定”。

这里创建“/boot”、“swap”和“/”三个分区，如图 1 11 所示。

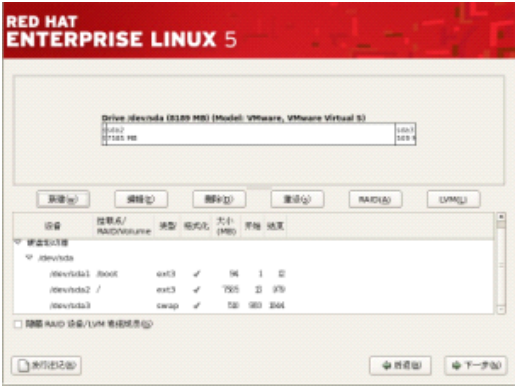


图 1 11 建立分区效果图

如果对建立的分区想进行修改，通过“编辑”、“删除”、“重设”等按钮来实现，当确定建立好分区后，单击“下一步”。

9. 引导装载程序配置

弹出“引导装载程序配置”对话框，如图 1 12 所示。按默认选项，单击“下一步”。



图 1 12 “引导装载程序配置”对话框

## 10. 网络配置

弹出“网络配置”对话框，如图 1 12 所示，一般默认设置，单击“下一步”。



图 1 13 “网络配置”对话框

## 11. 选择时区

弹出“选择时区”对话框，如图 1 14 所示，默认选择“亚洲/上海”，单击“下一步”。





图 1 14 “选择时区”对话框

## 12. root 帐号的密码

弹出“root 帐号的密码”输入对话框，如图 1 15 所示。输入相应账号和密码，单击“下一步”。



图 1 15 “root 帐号的密码”输入对话框

## 13. 选择安装组件

弹出“选择安装组件”对话框，如图 1 16 所示。用户可根据需要，选择安装软件包；如果您有足够大的空间，可以全部安装。单击“下一步”。



图 1 16 “选择安装组件”对话框

接下来，系统开始安装、拷贝软件、安装完毕后，重启动系统。重启动系统后，系统桌面如图 1 17 所示。

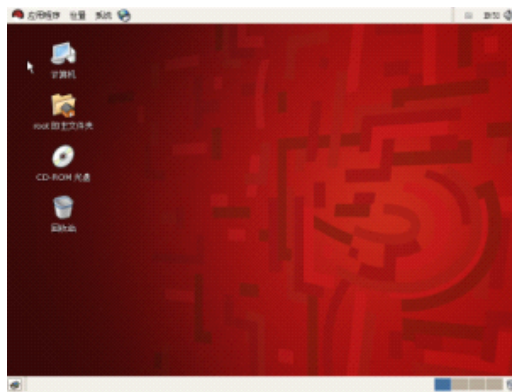


图 1 17 系统桌面

一般情况下，我们建议在虚拟机下使用 Linux。原因如下：

(1) 很多用户都习惯 windows 下的图形操作方式，使用虚拟机可以让用户快速地在两个操作系统中进行切换。

(2) 在虚拟机中，可以使用快照，快速将 Linux 系统挂起和启动。

关于虚拟机下 Linux 安装见附件一

## 1.3 Linux 文件及目录

### 1.3.1 Linux 文件

#### 1. 文件类型

Linux 下主要的文件类型可分为 4 种：普通文件、目录文件、设备文件和链接文件。

##### (1) 普通文件

普通文件是用户最常使用的文件。它包括了文本文件、数据文件、二进制可执行程序。

##### (2) 目录文件

在 Linux 中目录也是文件，其内容包含了文件名和子目录名以及指向那些文件和子目录的指针。目录文件是 Linux 中存储文件名的惟一地方，当把文件和目录相对应起来时，也就是用指针将其链接起来之后，就构成了目录文件。因此，在对目录文件进行操作时，通常不涉及对文件内容的操作，而只是对目录名和文件名的对应关系进行了操作。

在 Linux 系统中的每个文件都有一个惟一的数据，而这个数值被称为索引节点。索引节点存储在一个称作索引点表中。该表在磁盘格式化时被分配。每个实际的磁盘或分区都有自己的索引节点表。一个索引节点包含文件的所有信息，包括磁盘上数据的地址和文件类型。Linux 文件系统把索引节点号 1 赋予根目录，这也就是 Linux 的根目录文件在磁盘上的地址。根目录文件包括文件名、目录名及它们各自的索引节点号的列表，Linux 可以通过查找从根目录开始的一个目录链来到达系统中的任何一个文件。

##### (3) 链接文件

链接文件类似于 windows 系统的快捷方式，但并不完全一样。链接文件可分为软链接文件和硬链接文件，其区别如表 1-1 所示。

表 1-1 软链接文件和硬链接文件区别

软链接文件	硬链接文件
软链接文件又叫符号链接，软链接文件包含了另一个文件的路径名，可以是任意文件或目录	硬链接文件是已存在另一个文件，不允许经目录创建硬链接
可以链接不同文件系统的文件或目录	只有同一文件系统中的文件之间才能创建链接
在对符号文件进行读或写操作时，系统会自动把操作转换为对源文件的操作，但删除链接文件时，系统仅仅删除链接文件夹，而不删除源文件本身。	对硬链接文件进行读写和删除操作时，结果和软链接相同。但如果删除硬链接文件的源文件，硬链接文件仍然存在，而且保留了原有的内容。这时，系统就“忘记”了它曾经是硬链接文件，而把它当成了一个普通文件。

#### (4) 设备文件

在 Linux 中是把设备抽象成文件，然后对设备文件的操作就像对普通文件那样进行操作。需要注意的是，Linux 中设备相关的文件一般都在 /dev 目录下，它主要包括两种，一种是字符设备文件；一种是块设备文件。字符设备文件主要指的是串行端口的接口设备。块设备文件是指数据的读写是以块为单位的设备。如硬盘。

### 1. 文件属性

在 Linux 终端下输入如下命令 `ls -l` 列出当前目录下的所有文件和目录的相关信息，（命令的具体解释在第二章中介绍）

```
[rot@localhost ~]# ls -l
-rwxrwxrwx- 1 root root 39173 jan 1 2005 install.log
```

信息列表的最前字段的表达意思如图 1 18 所示。

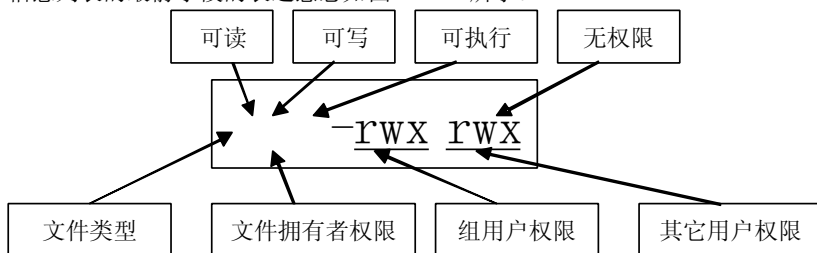


图 1 18 文件属性

第一个字符表示文件类型，第二个字符到第四个字符表示文件拥有者的权限，第五个字符到第七个字符表示所属组用户的权限，第八个字符到第十个字符表示其它组用户的权限。

第一个字符表示文件的类型：

- ① “-” 表示普通文件
- ② “d” 表示目录文件
- ③ “l” 表示链接文件
- ④ “c” 表示字符设备
- ⑤ “b” 表示块设备
- ⑥ “p” 表示命令管道
- ⑦ “f” 表示堆栈文件

第一个字符后有 3 个三位字符组：

（在 Linux 中的文件的拥有者可以把文件的访问属性设成 3 种不同的访问权限：可读（r）、可写（w）、可执行（x）。文件又有 3 个不同的用户级别：文件拥有者（u）、所属的用户组（g）、其它用户（o）。）

第一个三位字符组表示对于文件拥有者对该文件的权限；

第二个三位字符组表示文件用户组对该文件的权限；

第三个三位字符组表示系统其它用户对该文件的权限；  
若该用户组对此没有权限，一般显示“-”字符；

### 1.3.2 Linux 文件系统

文件系统是操作系统用于确定磁盘或分区上的文件的方法和数据结构；即在磁盘上组织文件的方法。也指用于存储文件的磁盘或分区，或文件系统种类。在 Linux 系统中，每个分区都是一个文件系统，都有自己的目录层次结构。Linux 系统最重要特征之一就是支持多种文件系统，这样它更加灵活。并可以和许多其他操作系统共享。

随着 Linux 系统的不断发展，Linux 系统内核可以支持几十种文件系统类型：JFS，ReiserFS，ext，ext2，ext3，XFS，Minx，MSDOS、UMSDOS、VFAT，NTFS，HPFS，NFS，SMB，SysV，PROC 等。

Linux 系统最常用的几种文件系统，如表 1 2 所示。

表 1 2 Linux 系统最常用的文件系统表

文件系统类型	描述
ext3	ext3 是现在 Linux 常见的文件系统，它是 ext2 的升级版本。Ext3 中采用了日志式的管理机制，它使用了日志式的管理机制，它使文件系统具有很强的快速恢复能力。
swap	swap 文件系统是 Linux 中作为交换分区使用的。在安装 Linux 的时候，是必须建立交换分区，其文件系统类型就是 swap 而没有其它选择，其大小一般是实际物理内存的两倍。
vfat	Linux 中也支持 Dos 中所采用的 FAT 文件系统（包括 FAT12，FAT16，FAT32），在 Linux 中 FAT 文件系统都称为 vfat 文件系统。
NFS	NFS 文件系统是指网络文件系统，这种文件系统是 LINUX 的特性之一。它可以很方便地在局域网内实现文件共享，并且使多台主机共享同一主机上的文件系统。NFS 文件系统访问速度快，稳定性高，已经得到了广泛的使用。尤其在嵌入式领域，使用 NFS 文件系统可以很方便地实现文件本地修改，而免去了一次次读写 flash，从而损坏 flash。

### 1.3.3 Linux 目录

Linux 的目录为树形结构，有一个在文件系统中唯一的“根”，既“/”。如前所述，目录也是一种文件，是具有目录属性的文件。当系统建立一个目录时，还会在这个目录下自动建立两个目录文件，一个是“.”，代表当前目录，另一个是“..”，代表当前目录的父目录。对于根目录，“.”和“..”都代表其自己。

详细的目录介绍如表 1 3 所示。

表 1 3 Linux 文件系统目录

目 录	作 用
/bin	存放系统所需要的那些命令,比如 ls,cp,mkdir 等命令;功能和/usr/bin 类似,这个目录中的文件都是可执行的、普通用户可以使用的命令。
/boot	这是 Linux 系统启动时所需要的文件目录,文件目录下存放有比如 initrd.img 等文件, grub 系统引导管理器也位于这个目录。
/dev	设备文件存储目录,比磁盘、光驱。
/etc	系统配置文件的所在,一些服务器的配置文件也在这里;比如用户帐号及密码配置文件。当系统启动时,需要读取其参数进行相应的配置
/home	普通用户目录默认存放目录
/lib	存放库文件的目录,目录中存放在着系统动态链接共享库的。
/sbin	该目录存放 root 用户的常用系统命令用户无权限执行这个目录下的命令。
/tmp	临时文件目录,有时用户运行程序的时候,会产生临时文件就放在此目录下
/usr	这个是系统存放程序的目录,比如命令、帮助文件等。当我们安装一个 Linux 发行版官方提供的软件包时,大多安装在这里。
/usr/bin	普通用户可执行文件目录
/usr/sbin	超级权限用户 root 可执行命令存放目录
/usr/src	内核源代码默认的放置目录
/var	这个目录的内容是经常变动的,/var 下有/var/log 这是用来存放系统日志的目录
/media	本目录是空的,是用于挂载的
/srv	一些服务需要访问的文件存放在这
/sys	系统的核心文件
/lost+found	系统异常信息存放目录
/misc	存放从 Dos 下进行安装的实用工具
/root	超级用户登录的主目录

## 本章小结

本章首先介绍了 Linux 发展历史、嵌入式 Linux 操作系统的优势、Linux 不同发行

版本的区别。

接着介绍了如何安装 Linux，这里最关键的一步是分区。

在安装完 Linux 后，本章讲解了 Linux 中文件和文件系统的概念。

# 第 2 章Linux 基础命令

## 学习目标

- Linux 基础命令
- Linux 文件命令
- Linux 系统命令
- Linux 网络命令
- Linux 压缩命令

Linux 是一款高可靠性、高性能的操作平台，而其所有优越性只有在用户直接使用 Linux 命令行（shell 环境）进行才做时才能够充分体现出来。

Linux 命令行的功能非常齐全且相当强大，这主要得益于 Linux 丰富的命令，一个 Red Hat Linux 的普通安装就拥有数千条命令，且支持用户自定义的命令。本章将分类对 52 条常用的 Linux 基础命令进行介绍，学会这些命令的使用，你将能够在 Linux 命令行操作中游刃有余。

## 2.1 文件相关命令

Linux 中常用的文件相关命令分为文件管理和文件处理两部分，常用命令如表 2 1 所示。

表 2 1 Linux 中常用的文件相关命令

类型	命 令	说 明	格 式
文件管理	pwd	显示当前路径	pwd
	ls	显示当前路径下的内容	ls
	mkdir	创建目录	mkdir [选项] 目录名
	rmdir	删除目录	rmdir 目录名
	cd	切换工作目录	cd [目录]
	touch	修改文件访问时间或修改时间	touch [选项] 文件名
	mv	重命名或移动文件	mv [选项] 源文件名 目标文件名
	cp	复制文件	cp [选项] 源文件 目标文件
	rm	删除文件	rm [选项] 文件名
文件	wc	显示行数、单词数和字节数	wc [选项] [文件名]



类型	命 令	说 明	格 式
处理	find	查找文件	find [文件名] [条件]
	file	显示文件类别	file 文件名
	du	显示文件占用磁盘信息	du [选项] [文件名]
	chmod	修改文件访问权限	chmod [选项] 权限字符串 文件名
	grep	抽取并列出包含文本的行	grep [选项] 文本 [文件名]

## 2.1.1 文件管理

### 1. pwd

#### 功能说明

显示当前路径。

#### 语法格式

wd

#### 使用实例

```
# pwd
/root
```

### 2. mkdir

#### 功能说明

创建目录。

#### 语法格式

mkdir [选项] 目录名

#### 使用实例

```
# mkdir mydir
```

### 3. ls

#### 功能说明

显示当前路径下的内容。

#### 语法格式

mkdir [选项] 目录名

#### 选项参数

- a 显示所有档案及目录
- l 除档案名称外，亦将档案型态、权限、拥有者、档案大小等资讯详细列出
- r 将档案以相反次序显示(原定依英文字母次序)
- t 将档案依建立时间之先后次序列出
- A 同 -a，但不列出“.”(目前目录)及“..”(父目录)
- F 在列出的档案名称后加一符号；例如可执行档则加“\*”，目录则加“/”

-R 若目录下有档案，则以下之档案亦皆依序列出

### 使用实例

```
# ls -a
mydir
```

如果按照本书之前所列举的所有 Linux 命令示例的顺序来做(本章中其后所有命令示例，如未做特殊说明，均遵循此原则，即均按照前文所列 Linux 命令示例逐一执行)，可以看到自己先前使用 mkdir 命令创建的目录 mydir，以及当前目录下的所有文件和目录(包括以 “.” 开头的隐藏文件和目录)。

## 4. cd

### 功能说明

切换工作目录。

### 语法格式

cd [目录]

### 使用实例

```
# cd mydir
```

此时再使用 pwd 命令，可发现工作目录已经变更为 /root/mydir 了。

## 5. touch

### 功能说明

修改文件访问时间或修改时间，也可以通过该命令创建一个空的文件。

### 语法格式

touch [选项] 文件名

### 使用实例

```
# touch myfile.txt
```

此时再使用 ls -l 命令显示当前目录下文件和目录的详细信息，可发现有一个名为 myfile.txt 的文件。接下来我们通过 touch 命令改变它的创建时间。

```
# touch myfile.txt -t 201201010000.30
```

此时再使用 ls -l 命令，可发现文件 myfile.txt 已经变更。

### 选项参数：

-a 修改文件 file 的存取时间.

-c 不创建文件 file

-m 修改文件 file 的修改时间

-r ref\_file 将参照文件 ref\_file 相应的时间戳记的数值作为指定文件 file 时间戳记的新值.

-t time 使用指定的时间值 time 作为指定文件 file 相应时间戳记的新值. 此处的 time 规定为如下形式的十进制数：[[CC]YY]MMDDhhmm[.SS]

## 6. cp

### 功能说明

复制文件。

### 语法格式

cp [选项] 源文件 目标文件

### 使用实例

```
# cp myfile.txt ./myfile_1.txt
```

此时再使用 ls 显示当前目录下文件和目录的详细信息，可发现有一个名为 myfile\_1.txt 的文件。

### 选项参数

- a: 相当于-pdr 的意思;
- d: 若来源文件为连结文件的属性(link file), 则复制连结文件属性而非档案本身;
- f: 为强制 (force) 的意思, 若有重复或其它疑问时, 不会询问使用者, 而强制复制;
- i: 若目的文件(destination) 已经存在时, 在覆盖时会先询问是否真的动作!
- l: 进行硬式连结(hard link) 的连结档建立, 而非复制档案本身;
- p: 连同档案的属性一起复制过去, 而非使用预设属性;
- r: 递归持续复制, 用于目录的复制行为;
- s: 复制成为符号连结文件(symbolic link), 亦即『快捷方式』档案;
- u: 若 destination 比 source 旧才更新 destination!

## 7. rm

### 功能说明

删除文件。

### 语法格式

rm [选项] 文件名

### 使用实例

```
# rm -rf myfile.txt
```

此时再使用 ls 显示当前目录下文件和目录的详细信息，可发现有文件 myfile.txt 已经不存在了。

### 选项参数

- f 忽略不存在的文件, 从不给出提示。
- r 指示 rm 将参数中列出的全部目录和子目录均递归地删除。
- i 进行交互式删除。

## 8. mv

### 功能说明

重命名或移动文件。

### 语法格式

mv [选项] 源文件名 目标文件名

### 使用实例

```
# mv myfile_1.txt myfile.txt
```

此时再使用 `ls` 显示当前目录下文件和目录的详细信息,可发现有文件 `myfile_1.txt` 已经变成了名为 `myfile.txt` 的文件。

### 选项参数

-i: 交互方式操作。如果 `mv` 操作将导致对已存在的目标文件的覆盖,此时系统询问是否重写,要求用户回答“y”或“n”,这样可以避免误覆盖文件。

-f: 禁止交互操作。`mv` 操作要覆盖某个已有的目标文件时不给任何指示,指定此参数后 `i` 参数将不再起作用。

## 9. rmdir

### 功能说明

删除目录。

### 语法格式

`rmdir` 目录名

### 使用实例

```
# rmdir mydir/
```

首先,在执行该命令前必须先返回上一级目录,即 `/root` 目录,否则提示:“没有那个文件或目录”并且删除失败。另外,即使返回了上一级目录,直接执行该命令也会提示:“目录非空”并且删除失败;此时,则需要先删除 `/root/mydir/` 目录下的所有文件,再执行此命令才可成功。

此时再使用 `ls` 显示 `/root` 目录下文件和目录,可发现有目录 `mydir/` 已经不存在了。

## 2.1.2 文件处理

### 1. wc

#### 功能说明

显示行数、单词数和字节数。

#### 语法格式

`wc` [选项] [文件名]

#### 使用实例

```
# wc /etc/bashrc
53 163 1253 /etc/bashrc
```

结果表示 `/etc/bashrc` 文件共有 53 行、163 个单词,文件大小为 1253 个字节(注意: `/etc/bashrc` 为系统配置文件,初学者不应将其内容改动)。

#### 选项参数

- c 统计字节数。
- l 统计行数。

-w 统计字数。

## 2. file

### 功能说明

显示文件类别。

### 语法格式

file 文件名

### 使用实例

```
# file /etc/bashrc
/etc/bashrc: ASCII text
```

结果表示/etc/bashrc 文件是一个使用 ASCII 字符编码的文本文件。

## 3. du

### 功能说明

显示文件占用磁盘信息。

### 语法格式

du [选项] [文件名]

### 使用实例

```
# du -sh /etc/bashrc
8.0K    /etc/bashrc
```

结果表示/etc/bashrc 文件占用了 8K 字节的磁盘空间。

### 选项参数：

- a 显示全部目录和其次目录下的每个档案所占的磁盘空间
- b 大小用 bytes 来表示（默认值为 k bytes）
- c 最后再加上总计（默认值）
- s 只显示各档案大小的总合
- x 只计算同属同一个档案系统的档案
- L 计算所有的档案大小

## 4. chmod

### 功能说明

修改文件访问权限。

### 语法格式

chmod [选项] 权限字符串 文件名

### 使用实例

```
#chmod 777 /etc/bashrc
```

此时使用 `ls -l /etc/bashrc` 命令可以发现文件/etc/bashrc 的权限已经变成“-rwxrwxrwx”。为了保持系统安全性，建议使用同样的方法（使用 `chmod 644 /etc/bashrc` 命令）将文件/etc/bashrc 改为一个比较安全的权限“-rw-r--r--”。

### 选项参数:

-v 显示权限改变的详细资料

-R 表示对当前目录下的所有文件和子目录进行相同的权限更改

权限字符串:

格式为[ugoa] [+ -=] [rwx]

u 表示文件的拥有者

g 表示与此文件拥有者属于一个组群的人

o 表示其他人

a 表示包含以上三者即文件拥有者(u)、群组(g)、其他(o)

+ 表示增加权限

= 表示唯一设置权限

- 表示取消权限, 数字代号为“0”

r 表示有读取的权限, 数字代号为“4”

w 表示有写入的权限, 数字代号为“2”

x 表示有执行的权限, 数字代号为“1”

## 5. find

### 功能说明

查找文件。

### 语法格式

find [文件名] [条件]

### 使用实例

```
# find /etc/ -name "bas*"
/etc/bashrc
/etc/selinux/targeted/src/policy/macros/base_user_macros.te
```

本例中在/etc/目录下查找所有文件名以“bas”开头的文件。

### 选项参数

条件:

-name filename : 搜索文件名为 filename 的文件

-size [+ -] SIZE : 搜索比 SIZE 还要大(+)或小(-)的文件,这个 SIZE 的规格有:c : 表示 byte,k:表示 1024bytes,所以,要找比 50KB 还要大的文件,就是“-size +50K”

-type TYPE : 搜索文件的类型为 TYPE,类型首要有:通常正轨文件(f),装备文件(b,c),目录(d),衔接文件(l),套接字(s),及 FIFO(p)等属性.

-perm mode :搜索文件属性“刚好等于”mode 的文件,这个 mode 为相似 chmod 的属性值,举例来说,-rwsr-xr-x 的属性为 4755 时,也会列出来.

-perm -mode :搜索文件属性“必须要 一切 包含 mode 的属性”的文件,举例来说,要搜索 -rwxr--r--, 即 0744 的文件,运用 -perm -0744, 当一个文件的属性为 -rwxr--r--, 即 4755 时,也会列出来,由于 -rwxr-xr-x 的属性已经包含了“-rwxr--r--

”的属性了.

-perm +mode :搜索文件属性”包含任一 mode 的属性”的文件, 举例来说, 我们 搜索-rwxr-xr-x, 即 -perm +755 时, 但一个文件属性为 -rw-----也会被列出, 由于它有-rw... 的属性存在.

-atime n : n 为数字, 意思为在 n 天之前的”一天之内”访问 (access) 过的文件.

-ctime n : n 为数字, 意思为在 n 天之前的”一天之内”修正 (change) 过的形状 的文件.

-mtime n: n 为数字, 意思为在 n 天之前的”一天之内”修正 (modification) 过的文件.

-newer file : file 为一个存在的文件, 意思是说, 只需文件比 file 还要新, 就会被列出来.

-uid n : n 为数字, 这个数字是用户的帐号 ID, 即 UID, 这个 UID 记载在/etc/passwd 里与帐号称号对应的数字.

-gid n : n 为数字, 这个数字是用户组称号的 ID, 即 GID, 这个 GID 记载在/etc/group.

-user name : name 为用户帐号称号 , 比方 dmtsai

-group name: name 为用户组称号 , 比方 users

-nouser :查找文件的拥有者不在/etc/passwd

-nogroup :查找文件的拥有者不在/etc/group 中的文件.

## 6. grep

### 功能说明

抽取并列出包含文本的行。

### 语法格式

grep [选项] 文本 [文件名]

### 使用实例

```
# grep "export PATH" /etc/*
/etc/bashrc:export PATH=/usr/local/arm/3.4.4/bin:$PATH
/etc/profile:export PATH USER LOGNAME MAIL HOSTNAME HISTSIZE
INPUTRC
/etc/zprofile:export PATH
```

本例中在/etc/目录下所有文件中抽取并列出了包含有字符串“export PATH”的行。

选项:

-n n 为数字, 同时显示匹配行上下的 n 行。

-b 打印匹配行前面打印该行所在的块号码。

-c 只打印匹配的行数, 不显示匹配的内容。

-f 从文件中提取模板。空文件中包含 0 个模板, 所以什么都不匹配。

-h 当搜索多个文件时, 不显示匹配文件名前缀。

- I 忽略大小写差别。
- q 取消显示，只返回退出状态。0 则表示找到了匹配的行。
- l 打印匹配模板的文件清单。
- L 打印不匹配模板的文件清单。
- n 在匹配的行前面打印行号。
- s 不显示关于不存在或者无法读取文件的错误信息。
- v 反检索，只显示不匹配的行。
- w 如果被<和>引用，就把表达式做为一个单词搜索。
- V 显示软件版本信息。

## 2.2 系统相关命令

Linux 系统命令分为系统信息查询、进程管理和用户管理三个部分，常见命令及名称及格式如表 2 2 所示。

表 2 2 常用 Linux 系统命令表

类型	命 令	说 明	格 式
系统信息查询	uname	显示当前操作系统名称	uname [选项]
	hostname	用以显示或设置系统的主机名称。	hostname [选项]
	date	显示和设置日期	date [选项] [日期]
	cal	显示日历	cal [选项] [年份]
	uptime	显示系统运行时长	uptime
	dmesg	显示开机信息	dmesg [选项]
进程管理	top	显示当前系统状态信息	top [选项]
	ps	显示进程状态	ps [选项] [进程号]
	kill	终止进程	kill [选项] 进程号
用户管理	who	显示登录到系统的所有用户	who
	whoami	显示当前用户	whoami
	last	显示近期登录的用户	last
	useradd	添加用户	useradd [选项] 用户名
	usermod	设置用户账号属性	usermod [选项] 属性值
	userdel	删除用户	userdel [选项] 用户名
	su	用户切换	su [选项] 用户名
	passwd	设置用户密码	passwd [用户名]
	groupadd	添加用户组	groupadd [选项] 用户组名
	groupmod	设置用户组账号属性	groupmod [选项] 属性值



	groupdel	删除用户组	groupdel [选项] 用户组名
	id	显示用户 ID、组 ID 和所属组列表	id [用户名]
	groups	显示用户所属组	groups [用户名]
环境变量	echo	将字符串标准输出	echo [选项] 字符串
	export	设置或显示环境变量	export [选项][变量名称]=[变量设置值]
	env	显示当前用户的环境变量	env [选项] [-] [变量名=值] [命令 [参数]]
	set	显示当前 shell 的环境变量	set [选项]...
	readonly	显示或定义只读环境变量	readonly [变量名]

### 2.2.1 系统信息查询

#### 1. uname

##### 功能说明

显示当前操作系统名称。

##### 语法格式

uname [选项]

##### 使用实例

```
# uname -a
Linux localhost.localdomain 2.6.9-89.ELsmp #1 SMP Mon Apr 20
10:34:33 EDT 2009 i686 i686 i386 GNU/Linux
```

本例中打印了包括操作系统名称在内的所有系统相关信息。

##### 选项参数

-a, --all #打印出所有信息，如果-p, -i 的结果为 unknown，将不显示-p, -i 的结果

-s, --kernel-name #打印出内核名称

-n, --nodename #打印出网络上主机名称

-r, --kernel-release #打印出操作系统发行号

-v, --kernel-version #打印出操作系统的版本

-m, --machine #打印出电脑硬件类型

-p, --processor #打印出处理器类型

-i, --hardware-platform #显示硬件平台类型

-o, --operating-system #打印出运行的系统

#### 2. hostname

##### 功能说明

显示或设置系统的主机名称。

### 语法格式

hostname [选项]

### 使用实例

```
# hostname  
localhost.localdomain
```

本例显示了系统的主机名称。

### 选项参数

-n: 显示主机在网络节点上的名称。

-o: 显示操作系统类型。

-r: 显示内核发行版本。

-s: 显示内核名称。

## 3. date

### 功能说明

显示和设置日期。

### 语法格式

date [选项] [日期]

### 使用实例

```
#date 062510322010.30  
五 6月 25 10:32:30 CST 2010
```

本例将系统时间设置为了 2010 年 6 月 25 日 10 点 32 分 30 秒。

### 选项参数

-d datestr, --date datestr 显示由 datestr 描述的日期

-s datestr, --set datestr 设置 datestr 描述的日期

-u, --universal 显示或设置通用时间

日期（以+号开头）：

% H 小时（00..23）

% I 小时（01..12）

% k 小时（0..23）

% l 小时（1..12）

% M 分（00..59）

% p 显示出 AM 或 PM

% r 时间（hh: mm: ss AM 或 PM），12 小时

% s 从 1970 年 1 月 1 日 00: 00: 00 到目前经历的秒数

% S 秒（00..59）

% T 时间（24 小时制）（hh:mm:ss）

% X 显示时间的格式（%H:%M:%S）

% Z 时区 日期域  
% a 星期几的简称 ( Sun..Sat )  
% A 星期几的全称 ( Sunday..Saturday )  
% b 月的简称 ( Jan..Dec )  
% B 月的全称 ( January..December )  
% c 日期和时间 ( Mon Nov 8 14: 12: 46 CST 1999 )  
% d 一个月的第几天 ( 01..31 )  
% D 日期 ( mm / dd / yy )  
% h 和 %b 选项相同  
% j 一年的第几天 ( 001..366 )  
% m 月 ( 01..12 )  
% w 一个星期的第几天 ( 0 代表星期天 )  
% W 一年的第几个星期 ( 00..53, 星期一为第一天 )  
% x 显示日期的格式 ( mm/dd/yy )  
% y 年的最后两个数字 ( 1999 则是 99 )  
% Y 年 ( 例如: 1970, 1996 等 )

#### 4. cal

##### 功能说明

显示日历。

##### 语法格式

cal [选项] [年份]

##### 使用实例

```
# cal
六月 2010
日    一    二    三    四    五    六
      1    2    3    4    5
6    7    8    9   10   11   12
13   14   15   16   17   18   19
20   21   22   23   24   25   26
27   28   29   30
```

本例显示了本月的日历。

##### 选项参数

- m : 以星期一为每周的第一天方式显示。
- j : 以凯撒历显示, 即以一月一日起的天数显示。
- y : 显示今年年历。

#### 5. uptime

##### 功能说明

显示系统运行时长。

### 语法格式

uptime

### 使用实例

```
uptime
10:34:24 up 10:50, 3 users, load average: 0.01, 0.05, 0.07
```

本例显示了当前时间 10:34:24 及开机时长 10 小时 50 分等信息。

## 6. dmesg

### 功能说明

显示开机信息。

### 语法格式

dmesg [选项]

### 使用实例

```
# dmesg
Linux version 2.6.9-89.ELsmp
(mockbuild@hs20-bc1-2.build.redhat.com)
(gcc version 3.4.6 20060404 (Red Hat 3.4.6-11))
#1 SMP Mon Apr 20 10:34:33 EDT 2009
BIOS-provided physical RAM map:
BIOS-e820: 0000000000000000 - 0000000000009f800 (usable)
BIOS-e820: 0000000000009f800 - 000000000000a0000 (reserved)
.....
Installing knfsd (copyright (C) 1996 okir@monad.swb.de).
eth0: no IPv6 routers present
VMCIUtil: Updating context id from 0xffffffff to 0x43b6d2c7 on
event 0.
```

本例显示了与开机有关的大量信息。

### 选项参数：

- c 显示信息后，清除 ring buffer 中的内容。
- s 预设置为 8196，刚好等于 ring buffer 的大小。
- n 设置记录信息的层级。

## 2.2.2 进程管理

### 1. ps

### 功能说明

显示进程状态。

**语法格式**

ps [选项] [进程号]

**使用实例**

```
# ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root          1        0  0 Jun24 ?           00:00:03 init [5]
root          2        1  0 Jun24 ?           00:00:00 [migration/0]
root          3        1  0 Jun24 ?           00:00:00 [ksoftirqd/0]
.....
root       19508   10018   0 03:30 pts/2        00:00:00 bash
root       22561   10018   0 05:20 pts/1        00:00:00 bash
root       31630   19508   89 10:53 pts/2        00:00:05 ./deadLoop
root       31512   19508   0 10:48 pts/2        00:00:00 ps -ef
```

本例以树形结构显示了所有正在运行的进程的状态。

**选项参数：**

- l 长格式输出
- u 按用户名和启动时间的顺序来显示进程
- j 用任务格式来显示进程
- f 用树形格式来显示进程
- a 显示所有用户的所有进程（包括其它用户）
- x 显示无控制终端的进程
- r 显示运行中的进程
- ww 避免详细参数被截断

**2. kill**

**功能说明**

终止进程。

**语法格式**

kill [选项] 进程号

**使用实例**

```
# kill 31630
```

根据前一个例子的显示结果，我们要结束掉“root 31630 19508 89 10:53 pts/2 00:00:05 ./deadLoop”这一进程。执行以上命令后，再使用“ps -ef”查看进程状态，可以发现该进程已经消失，即已被结束掉

**选项参数**

- s 指定需要送出的信号。既可以是信号名也可以对应数字。
- p 指定 kill 命令只是显示进程的 pid，并不真正送出结束信号。
- l 显示信号名称列表，这也可以在/usr/include/Linux/signal.h 文件找到。

## 2.2.3 用户管理

### 1. who

#### 功能说明

显示登录到系统的所有用户。

#### 语法格式

who

#### 使用实例

```
# who
root      :0          Jun 17 11:30
root      pts/1        Jun 17 17:03 (:0.0)
root      pts/2        Jun 17 15:13 (:0.0)
```

本例显示了当前登录到系统的所有用户。

### 2. whoami

#### 功能说明

显示当前用户。

#### 语法格式

whoami

#### 使用实例

```
# whoami
root
```

本例显示当前用户为 root。

### 3. last

#### 功能说明

显示近期登录的用户。

#### 语法格式

last

#### 使用实例

```
# last
root      pts/3        :0.0          Fri Jun 25 10:09 - 10:26  (00:17)
root      pts/3        :0.0          Fri Jun 25 09:41 - 10:09  (00:27)
root      pts/1        :0.0          Thu Jun 17 17:03      still logged
in
root      pts/2        :0.0          Thu Jun 17 15:13      still logged
in
wtmp begins Thu Jun 17 15:13:46 2010
```

本例显示了最近登录到系统的 4 个用户。

4. passwd

功能说明

设置用户密码。

语法格式

passwd [用户名]

使用实例

```
# passwd
Changing password for user root.
New UNIX password:
BAD PASSWORD: it is too simplistic/systematic
Retype new UNIX password:
passwd: all authentication tokens updated successfully.
```

该命令执行后会要求连续两次输入新密码，若两次输入一致则密码修改成功。值得注意的是：在你输入密码的过程中将不会显示任何字符。

2.3 网络相关命令

Linux 中常见的网络相关命令如下表所示，本书将选取其中使用较频繁的命令进行讲解。

命 令	说 明	格 式
ifconfig	显示或设置网络设备	ifconfig [网络设备][选项]...
ping	检测主机	ping [选项] 主机名或 IP

1. ifconfig

功能说明：

显示或设置网络设备。

语法格式：

ifconfig [网络设备][选项]...

使用实例：

```
# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:0C:29:0B:33:DF
          inet      addr:172.22.60.125      Bcast:172.22.255.255
Mask:255.255.0.0
          inet6 addr: fe80::20c:29ff:fe0b:33df/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
```

```
RX packets:12 errors:0 dropped:0 overruns:0 frame:0
TX packets:13 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:9409 (9.1 KiB) TX bytes:758 (758.0 b)
Interrupt:185 Base address:0x2024
```

本例显示了 eth0（类似 Windows 的本地连接 1）的 IP 等相关信息。

**选项参数：**

网络设备：

指定的网络接口名，如 eth0 和 eth1。

选项：

up：激活指定的网络接口卡。

down：关闭指定的网络接口。

broadcast address：设置接口的广播地址。

pointopoint：启用点对点方式。

## 2. ping

**功能说明：**

检测主机。

**语法格式：**

ping [选项] 主机名或 IP

**使用实例：**

```
# ping 172.22.60.1 -c 3
PING 172.22.60.1 (172.22.60.1) 56(84) bytes of data.
64 bytes from 172.22.60.1: icmp_seq=0 ttl=64 time=0.445 ms
64 bytes from 172.22.60.1: icmp_seq=1 ttl=64 time=1.09 ms
64 bytes from 172.22.60.1: icmp_seq=2 ttl=64 time=2.05 ms

--- 172.22.60.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 0.445/1.197/2.055/0.661 ms, pipe 2
```

本例中向 IP 为 172.22.60.1 的主机发起三次检测信号，并显示目标主机的响应时长。值得注意的是：本地主机应与目标主机在同一网段，从上一个例子中“ifconfig”命令所显示的 IP 地址“172.22.60.125”，可知本地主机应与目标主机 172.22.60.1 处在同一网段中。

**选项参数：**

-d 使用 Socket 的 SO\_DEBUG 功能。

-c <完成次数>设置完成要求回应的次数。

-f 极限检测。



- I <间隔秒数>指定收发信息的间隔时间。
- I <网络界面>使用指定的网络界面送出数据包。
- l <前置载入>设置在送出要求信息之前，先行发出的数据包。
- n 只输出数值。
- p <范本样式>设置填满数据包的范本样式。
- q 不显示指令执行过程，开头和结尾的相关信息除外。
- r 忽略普通的 Routing Table，直接将数据包送到远端主机上。
- R 记录路由过程。
- s <数据包大小>设置数据包的大小。
- t <存活数值>设置存活数值 TTL 的大小。
- v 详细显示指令的执行过程。

## 2.4 压缩打包相关命令

Linux 中常见的压缩打包相关命令如下表所示，本书将选取其中使用较频繁的命令进行讲解。

命 令	说 明	格 式
tar	打包备份文件	tar [选项]... [文件]...
bzip2	bz2 文件格式压缩或解压	bzip2 [选项] [文件名]
bunzip2	bz2 文件格式解压	bzip2 [选项] 文件名
bzip2recover	修复损坏的 bz2 文件格式	bzip2recover 文件名
gzip	gz 文件格式压缩	gzip [选项] [文件名]
gunzip	gz 文件格式解压	gunzip [选项] 文件名
unzip	zip 文件格式（由 winzip 压缩）解压	unzip [选项] 文件名
compress	早期的压缩解压（后缀名为.Z）	compress [选项] 文件名

### 1. tar

**功能说明：**

打包备份文件。

**语法格式：**

tar [选项]... [文件]...

**使用实例：**

```
# tar -cjf mydir.tar.bz2 mydir/
```

执行完该命令后，再使用“ls”命令进行查询，发现新增了一个名为“mydir.tar.bz2”的文件，该文件是以 bz2 的格式打包压缩而成的。

**选项参数：**

- t 列出归档文件内容目录
- x 从归档文件中解析文件
- c 创建新的归档文件]
- (上面三个参数，不能同时存在，仅能使用其中一个，即 t/x/c)
- f 指定备份文件，或设备，例如磁带机 /dev/st0
- v 显示命令的执行过程
- Z 使用 compress 命令处理备份文件
- z 使用 gzip 命令处理备份文件
- j 使用 bzip2 命令处理备份文件
- (-z 和 -j 都是在备份文件打包后才进行压缩的操作的，并且，该操作会影响其他的参数)
- C 先进入指定的目录，再释放

2.5 其他命令

Linux 中一些常见的其他命令如下表所示，本书将选取其中使用较频繁的命令进行讲解。

命 令	说 明	格 式
clear	清屏	clear
cat	显示文本文件内容	cat [选项] [文件名]
mount	挂载	mount [选项] 设备或节点 目标目录
man	显示命令手册	man [领域代号] 命令名

1. clear

功能说明

清屏。

语法格式

clear

使用实例

```
#clear
```

执行结果为屏幕刷新并清空。

2. cat

功能说明

显示文本文件内容。

## 语法格式

cat [选项] [文件名]

## 使用实例

```
# cat /etc/bashrc
# /etc/bashrc

# System wide functions and aliases
# Environment stuff goes in /etc/profile
.....
# vim:ts=4:sw=4
export PATH=/usr/local/arm/3.4.4/bin:$PATH
```

执行该命令后，文件“/etc/bashrc”的内容被读取并以文字形式打印出来。

## 选项参数

- n 由 1 开始对所有输出的行数编号
- b 和 -n 相似，只不过对于空白行不编号
- s 当遇到有连续两行以上的空白行，就代换为一行的空白行
- v

## 3. mount

### 功能说明

挂载。

### 语法格式

mount [选项] 设备或节点 目标目录

### 使用实例

```
#mount -t vfat /dev/sdb1 /root/mydir
```

本例可以挂载 U 盘，其中设备节点“/dev/sdb1”的主次设备号分别为 8 和 17。

## 选项参数

- t<文件系统类型> 指定设备的文件系统类型
- h: 显示辅助信息。
- v: 显示信息，通常和-f 用来除错。
- a: 把/etc/fstab 中定义的所有文件系统挂上。
- F: 这个命令通常和-a 一起使用，它会为每一个 mount 的动作产生一个行程负责执行。在系统需要挂上大量 NFS 文件系统时可以加快加载的速度。
- 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。
- n: 一般而言，mount 挂上后会在/etc/mtab 中写入一笔资料，在系统中没有可写入文件系统的情况下，可以用这个选项取消这个动作。
- o async 打开非同步模式，所有的档案读写动作都会用非同步模式执行。

-o sync 在同步模式下执行。

-o atime , -o noatime

当 atime 打开时，系统会在每次读取档案时更新档案的『上一次调用时间』。当我们使用 flash 档案系统时可能会选项把这个选项关闭以减少写入的次数。

-o auto , -o noauto 打开/关闭自动挂上模式。

-o defaults 使用预设的选项 rw, suid, dev, exec, auto, nouser, and async.

-o dev , -o nodev

-o exec , -o noexec 允许执行档被执行。

-o suid , -o nosuid 允许执行档在 root 权限下执行。

-o user , -o nouser 使用者可以执行 mount/umount 的动作。

-o remount 将一个已经挂下的档案系统重新用不同的方式挂上。例如原先是唯读的系统，现在用可读写的模式重新挂上。

-o ro 用唯读模式挂上。

-o rw 用可读写模式挂上。

-o loop 使用 loop 模式用来将一个档案当成硬盘分割挂上系统。

#### 4. man

##### 功能说明

显示命令手册。

##### 语法格式

man [领域代号] 命令名

##### 使用实例

```
#man man
```

该命令执行后，将显示命令手册中与 man 命令相关的部分。

##### 选项参数

领域代号：

- 1 用户命令，可由任何人启动的。
- 2 系统调用，即由内核提供的函数。
- 3 例程，即库函数。
- 4 设备，即/dev 目录下的特殊文件。
- 5 文件格式描述，例如/etc/passwd。
- 6 游戏
- 7 杂项，如宏命令包、惯例等。
- 8 系统管理员工具，只能由 root 启动。
- 9 其他（Linux 特定的），用来存放内核例行程序的文档。
- n 新文档，可能要移到更适合的领域。
- o 老文档，可能会在一段期限内保留。
- l 本地文档，与本特定系统有关的。

## 本章小结

Linux 的命令行博大精深，有数不清的内容等待读者去探索。由于 Linux 中的命令非常多，要全部介绍也是几乎不可能的。因此本章主要讲解了 Linux 操作的基本命令，这些命令是使用 Linux 的基础。

Linux 基本命令包括文件相关命令、系统相关命令、网络相关命令、压缩打包相关命令及其他命令。其中，着重介绍了每一类命令中使用频繁且具有代表性的重要命令及其用法，并给出了具体实例，对于未给予详细讲解的命令则列出了其使用方法。希望读者能够根据各命令之间用法和功能的相似性和通用性，举一反三、灵活运用。

## 第 3 章Linux 下的 C 编程环境

### 学习目标

- 熟悉 Linux 系统下的开发环境
- 掌握 Vi 的基本操作
- 理解 Gcc 编译器的基本原理
- 熟练使用 Gcc 编译器的常用选项
- 熟练使用 Gdb 调试技术
- 熟悉 Makefile 基本原理及语法规则

### 3.1 Linux 下 C 语言编程环境概述

在进行 Linux 系统下的开发时，我们必须确定我们的开发环境具备编辑器及开发工具这两个软件包。如图 3-1 和图 3-2 所示。

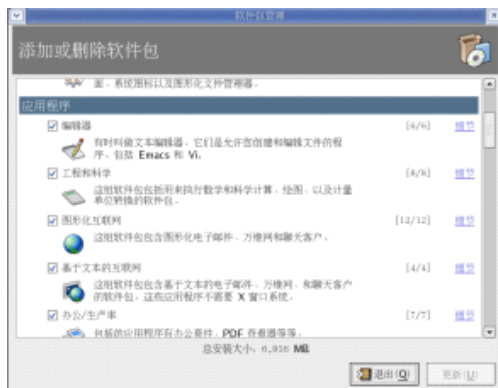


图 3-1 应用程序软件包



图 3 2 开发软件包

Linux 下的 C 语言程序设计与在其他环境中的 C 程序设计一样，主要涉及到编辑器、编译链接器、调试器及项目管理工具。

## 1. 编辑器

Linux 下的编辑器就如 Windows 下的 word、记事本等一样，完成对所录入文字的编辑功能。Linux 中最常用的编辑器有 Vi (Vim) 编辑器，功能强大，使用方便，广受编程爱好者的喜爱。

## 2. 编译链接器

编译是指源代码转化生成可执行代码的过程，它所完成工作主要如图 3 3 所示。

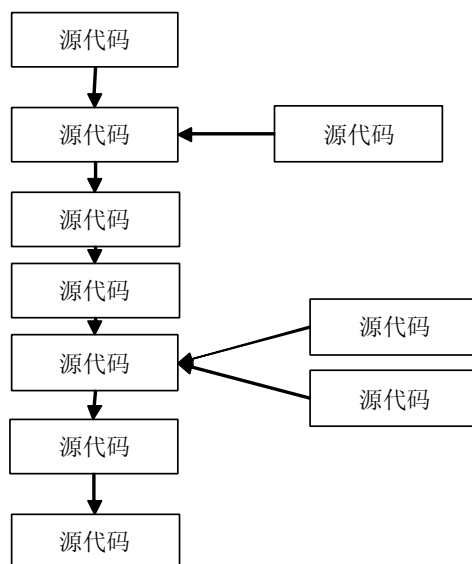


图 3 3 编译过程

可见，编译过程是非常复杂，它包括词法、语法和语义的分析、中间代码的生成和优化、符号表的管理和出错处理等。在 Linux 中，最常用的编译器是 Gcc 编译器。它是 GNU 推出的功能强大、性能优越的多平台编译器，其执行效率与一般的编译器相比平均效率要高 20%~30%，堪称为 GNU 的代表作品之一。

### 3. 调试器

调试器并不是代码执行的必备工具，而是专为程序员方便调试程序而用的。有编程经验的读者都知道，在编程的过程当中，往往调试所消耗的时间远远大于编写代码的所消耗时间。因此，有一个功能强大、使用方便的调试器是必不可少的。Gdb 是绝大多数 Linux 开发人员所使用的调试器，它可以方便地设置断点、单步跟踪等，以满足开发人员的需要。

### 4. 项目管理器

Linux 中的项目管理器“make”有些类似于 Windows 中 VisualC++ 里的“工程”，它是一种控制编译或者重复编译软件的工具，另外，它还能自动管理软件编译的内容、方式和时机，使程序员能够把精力集中在编写代码上而不是在源代码的组织上。

## 3.2 编辑器

### 3.2.1 关于文本编辑器

Linux 系统提供了一个完整的编辑器家族系列，例如 Ed、Ex、Vi 和 Emacs 等。按功能它们可以分为两大类：行编辑器（Ed、Ex）和全屏编辑器（Vi、Emacs）。行编辑器每次只能对一行进行操作，使用起来很不方便。而全屏编辑器可以对整个屏幕进行编辑，用户编辑的文件直接显示在屏幕上，从而克服了行编辑的那种不直观的操作方式，便于用户学习和使用，具有强大的功能。

Vi 是 Linux 系统的第一个全屏交互式编辑程序，它从诞生至今一直得到广大用户的青睐，历经数十年仍然是人们主要使用的文本编辑工具，由此可见其生命力之强，而强大的生命力是其强大的功能带来的。由于大多数读者在此之前都已经用惯了 Windows 的 word 等编辑器，因此，在刚刚接触时总会或多或少不适应，但只要习惯之后，就能感受到它的方便与快捷。

### 3.2.2 vi 编辑器的使用方法

#### 1. 如何调用 vi

```
[root@localhost ~]# vi filename
```



运行结果：

```
~
~
~
~
```

2. vi 的三种命令模式

- Command（命令）模式，用于输入命令。
- Insert（插入）模式，用于插入文本。
- Visual（可视）模式，用于可视化的的高亮并选定正文。

3. 文件的保存和退出

Command 模式是 vi 或 vim 的默认模式，如果我们处于其它命令模式时，要通过 ESC 键切换。当我们按 ESC 键后，接着再输入 “:” 号时，vi 会在屏幕的最下方等待我们输入命令。Vi 命令的底行模式如表 3-1 所示。

表 3-1 Vi 命令的底行模式

目 录	目 录 内 容
w	保存
w filename	filename 另存为 filename
wq!	保存退出
wq! filename	filename 注：以 filename 为文件名保存后退出
q!	不保存退出
x	应该是保存并退出，功能和:wq!相同

4. 光标移动

当我们按 ESC 进入 Command 模式后，我们可以用下面的一些键位来移动光标；命令参数如表 3-2 所示。

表 3-2 移动光标功能键

目 录	目 录 内 容
j	向下移动一行
k	向上移动一行
h	向左移动一个字符
l	向右移动一个字符

目 录	目 录 内 容
ctrl+b	向上移动一屏
ctrl+f	向下移动一屏
向上箭头	向上移动
向下箭头	向下移动
向左箭头	向左移动
向右箭头	向右移动

我们编辑一个文件时，对于 j、k、l 和 h 键，还能在这些动作命令的前面加上数字，比如 3j，表示向下移动 3 行。

## 5. 插入模式（文本的插入）

文本插入功能键如表 3-3 所示

表 3-3 文本插入功能键

目 录	目 录 内 容
i	在光标之前插入
a	在光标之后插入
I	在光标所在行的行首插入
A	在光标所在行的行末插入
o	在光标所在的行的上面插入一行
O	在光标所在的行的下面插入一行
s	删除光标后的一个字符，然后进入插入模式
S	删除光标所在的行，然后进入插入模式

## 6. 文本内容的删除操作

Vi 命令行模式功能键如表 3-4 所示

表 3-4 Vi 命令行模式功能键

目 录	目 录 内 容
x	删除一个字符
Nx	删除几个字符，N 表示数字，比如 3x
dw	删除一个单词

Ndw	删除几个单词，N用数字表示，比如 3dw 表示删除三个单词
dd	删除一行
Ndd	删除多个行，N代表数字，比如 3dd 表示删除光标行及光标的下两行
d\$	删除光标到行尾的内容
J	清除光标所处的行与上一行之间的空格，把光标行和上一行接在一起

### 7. 恢复修改及恢复删除操作

u 撤消修改或删除操作；

按 ESC 键返回 Command（命令）模式，然后按 u 键来撤消删除以前的删除或修改；如果您想撤消多个以前的修改或删除操作，请按多按几次 u。这和 Word 的撤消操作没有太大的区别；

### 8. 可视模式

在最新的 Linux 发行版本中，vi 提供了可视模式，因为这个功能是 vim 才有的。如果您用的 vi 没有这个功能，就换成 vim 就有了。打开可视模式，按 ESC 键，然后按 v 就进入可视模式；

可视模式为我们提供了极为友好的选取文本范围，以高亮显示；在屏幕的最下方显示有：

-- 可视 --

或

--VISUAL--

如图 3 4 所示。

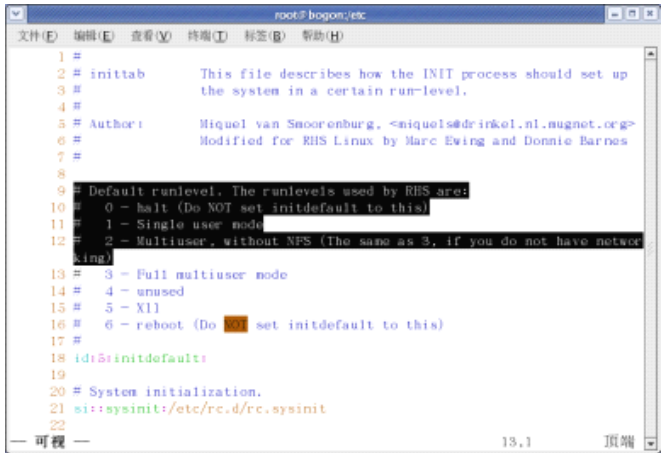


图 3 4 vi 的可视模式

进入可视模式，我们就可以用前面所说的命令行模式中的光标移动指令，可以进行

文本范围的选取。

选取文本范围有何用？

我们可以对某部份删除作业，按 `d` 键就删除了我们选中的内容。

选中内容后，我们按 `y` 就表示复制，按 `d` 表示删除。

值得一提的是的删除的同时，也表示复制。我们返回到命令模式，然后移动光标到某个位置，然后按 `shift+p` 键，就把刚才删除的内容贴上了。我们先在这里提一句，在后文，我们还得详细说说。

退出可视模式，还是用 `ESC` 键。

## 9. 复制和粘帖的操作

其实删除也带有剪切的意思，当我们删除文字时，可以把光标移动到某处，然后按 `shift+p` 键就把内容贴在原处，然后再移动光标到某处，然后再按 `p` 或 `shift+p` 又能贴上。

`p` 在光标之后粘帖。

`shift+p` 在光标之前粘帖。

来举一例：

比如我们想把一个文档的第三行复制下来，然后帖到第五行的后面，我们应该怎么做呢？

有两种方法；

第一种方法：

先把第三行删除，把光标移动到第三行处，然后用 `dd` 动作，接着再按一下 `shift+p` 键。这样就把刚才删除的第三行帖在原处了。

接着我们再用 `k` 键移动光标到第五行，然后再按一下 `p` 键，这样就把第三行的内容又帖到第五行的后面了；

第二种方法：

进入可视模式，按 `ESC` 键，然后按 `v` 键。移动鼠标指针，选中第三行的内容，然后按 `y` 键复制；再移动指针到第五行，最后按 `p` 键；

所以复制和粘帖操作，是命令模式、插入模式及可视模式的综合运用；我们要学会各种模式之间的切换，要常用 `ESC` 键；更为重要的学会在命令模式下移动光标。

## 10. 关于行号

有时我们配置一个程序运行时，会出现配置文件 `X` 行出现错误。这时我们要用到行号相关的操作；

为所有内容添加行号

按 `ESC` 键，然后输入：

```
:set number
```

光标所处位置

在屏幕的右下角，有类似如下的；

```
57,8 27%
```

在这之中，57 表示第 57 行，8 表示第 8 个字符；

## 11. 查找和替换

### (1) 查找

首先，我们要进入 ESC 键，进入命令模式；我们输入 / 或 ? 就进入查找模式了；

/SEARCH 注：正向查找，按 n 键把光标移动到下一个符合条件的地方。

?SEARCH 注：反向查找，按 shift+n 键，把光标移动到下一个符合条件的地方。

举一例：比如我想在一个文件中找到 swap 单词，我应该如下做；

首先按 ESC 键，进入命令模式，然后输入：

```
/swap
```

或

```
?swap
```

### (2) 替换

按 ESC 键进入命令模式。

```
:s /SEARCH/REPLACE/g
```

注：把当前光标所处的行中的 SEARCH 单词，替换成 REPLACE，并把所有 SEARCH 高亮显示；

```
:%s /SEARCH/REPLACE
```

注：把文档中所有 SEARCH 替换成 REPLACE；

```
:#,# s /SEARCH/REPLACE/g
```

注：# 号表示数字，表示从多少行到多少行，把 SEARCH 替换成 REPLACE；

注：在这之中，g 表示全局查找；我们注意到，就是没有替换的地方，也会把 SEARCH 高亮显示；

#### 举例说明：

比如我们有一篇文档要修改；

我们把光标所在的行，把所有单词 the，替换成 THE，应该是：

```
:s /the/THE/g
```

我们把整篇文档的所有的 the 都替换成 THE，应该是：

```
:%s /the/THE
```

我们仅仅是把第 1 行到第 10 行中的 the，替换成 THE，应该是：

```
:1,10 s /the/THE/g
```

补充说明：本文的目的是让新手在最短的时间内用 vi 或 vim 创建、编辑和修改文件，所以说本文并不是大而全的 vi 手册。如果把 vi 所有的功能都说全了，至少得写一本千页的手册；本也没有涉及更为高级的 vi 用法。如果想了解的更多，请查找 man 和 help。

## 3.3 Gcc 编译器

不经意间，GCC 已发展到了 4.5 的版本，尽管在软件开发社区之外乏人闻问，但因为它在几乎所有开源软件和自由软件中都会用到，因此它的编译性能的涨落会直接影响到 Linux、Firefox 乃至 OpenOffice.org 和 Apache 等几千个项目的开发。因此，把 GCC 摆在开源软件的核心地位是一点也不为过。另一方面，GCC4.5 的出现，正在牵引着广大程序员们的心。如果我们非要用一个词来说明 GCC 与程序员之间的关系，那无疑是“心随心动”。

Linux 系统下的 gcc (GNU C Compiler) 是 GNU 推出的功能强大、性能优越的多平台编译器，是 GNU 的代表作品之一。Gcc 是在多种硬件平台上编译出可执行程序的超级编译器，其执行效率与一般的编译器相比平均效率要高 20%~30%。

gcc 是 linux 的唯一编译器，没有 gcc 就没有 linux，gcc 的重要性就不可言喻。居然这么重要，那就很值得我们来好好研究下。好啦，开始我们的 gcc 之旅吧！

### 3.3.1 Gcc 编译流程

如本章开头提到的，Gcc 的编译流程分为了 4 个步骤，分别为：

- 预处理 (Pre-Processing);
- 编译 (Compiling);
- 汇编 (Assembling);
- 链接 (Linking)。

下面就具体来查看一下 Gcc 编译器是如何完成 4 个步骤的。

首先，有以下 hello.c 源代码：

```
#include<stdio.h>
int main()
{
printf("Hello! This is our embedded world!\n");
return 0;
}
```

#### 1. 预处理阶段

在该阶段，编译器将上述代码中的 stdio.h 编译进来，并且用户可以使用 Gcc 的选项“-E”进行查看，该选项的作用是让 Gcc 在预处理结束后停止编译过程。

注意

Gcc 指令的一般格式为：Gcc [选项] 要编译的文件 [选项] [目标文件]

其中，目标文件可缺省，Gcc 默认生成可执行的文件，命为：编译文件.out

```
[root@localhost Gcc]# Gcc -E hello.c -o hello.i
```

在此处，选项“-o”是指目标文件，“.i”文件为已经过预处理的C原始程序。以下列出了hello.i文件的部分内容：

```
typedef int (__gconv_trans_fct) (struct __gconv_step *,
struct __gconv_step_data *, void *,
__const unsigned char *,
__const unsigned char **,
__const unsigned char *, unsigned char **,
size_t *);
...

# 2 "hello.c" 2
int main()
{
printf("Hello! This is our embedded world!\n");
return 0;
}
```

由此可见，Gcc的确进行了预处理，它把“stdio.h”的内容插入到hello.i文件中。

## 2. 编译阶段

接下来进行的是编译阶段，在这个阶段中，Gcc首先要检查代码的规范性、是否有语法错误等，以确定代码的实际要做的工作，在检查无误后，Gcc把代码翻译成汇编语言。用户可以使用“-S”选项来进行查看，该选项只进行编译而不进行汇编，生成汇编代码。

```
[root@localhost Gcc]# gcc -S hello.i -o hello.s
```

以下列出了hello.s的内容，由此可见Gcc已经将其转化为汇编了，感兴趣的读者可以分析一下这几行简单的C语言小程序是怎样用汇编代码实现的。

```
.file "hello.c"
.section .rodata
.align 4
.LC0:
.string "Hello! This is our embedded world!"
.text
.globl main
.type main, @function
main:
pushl %ebp
movl %esp, %ebp
subl $8, %esp
```

```

andl $-16, %esp
movl $0, %eax
addl $15, %eax
addl $15, %eax
shrl $4, %eax
sall $4, %eax
subl %eax, %esp
subl $12, %esp
pushl $.LC0
call puts
addl $16, %esp
movl $0, %eax
leave
ret
.size main, .-main
.ident "GCC: (GNU) 4.0.0 20050519 (Red Hat 4.0.0-8)"
.section .note.GNU-stack,"",@progbits

```

### 3. 汇编阶段

汇编阶段是把编译阶段生成的“.s”文件转成“.o”的目标文件，读者在此可使用选项“-c”就可看到汇编代码已转化为“.o”的二进制目标代码了。如下所示：

```
[root@localhost Gcc]# gcc -c hello.s -o hello.o
```

### 4. 链接阶段

在成功编译之后，就进入了链接阶段。在这里涉及到一个重要的概念：函数库。

读者可以重新查看这个小程序，在这个程序中并没有定义“printf”的函数实现，且在预编译中包含进的“stdio.h”中也只有该函数的声明，而没有定义函数的实现，那么，是在哪里实现“printf”函数的呢？最后的答案是：系统把这些函数实现都被做到名为 libc.so.6 的库文件中去了，在没有特别指定的情况下，Gcc 会到系统默认的搜索路径“/usr/lib”下进行查找，也就是链接到 libc.so.6 库函数中去，这样就能实现函数“printf”了。函数库一般分为静态库和动态库两种。静态库是指编译链接时，把库文件的代码全部加入到可执行文件中，因此生成的文件比较大，但在运行时也就不再需要库文件了。其后缀名一般为“.a”。动态库与之相反，在编译链接时并没有把库文件的代码加入到可执行文件中，而是在程序执行时由运行时链接文件加载库，这样可以节省系统的开销。动态库一般后缀名为“.so”，如前面所述的 libc.so.6 就是动态库。Gcc 在编译时默认使用动态库。完成了链接之后，Gcc 就可以生成可执行文件，如下所示。

```
[root@localhost Gcc]# gcc hello.o -o hello
```

运行该可执行文件，出现的结果如下。

```
[root@localhost Gcc]# ./hello
```



```
Hello! This is our embedded world!
```

### 3.3.2 Gcc 编译选项

Gcc 有超过 100 个的可用选项，主要包括总体选项、告警和出错选项、优化选项和体系结构相关选项。以下对每一类中最常用的选项进行讲解。

#### 1. 总体选项

Gcc 的总结选项如表 3-5 所示，很多在前面的示例中已经有所涉及。

表 3-5 Gcc 总体选项列表

后缀名	所对应的语言
-c	只是编译不链接，生成目标文件“.o”
-S	只是编译不汇编，生成汇编代码
-E	只进行预编译，不做其他处理
-g	在可执行程序中包含标准调试信息
-o file	把输出文件输出到 file 里
-v	打印出编译器内部编译各过程的命令行信息和编译器的版本
-I dir	在头文件的搜索路径列表中添加 dir 目录
-L dir	在库文件的搜索路径列表中添加 dir 目录
-static	链接静态库

对于“-c”、“-E”、“-o”、“-S”选项在前一小节中已经讲解了其使用方法，在此主要讲解另外两个非常常用的库依赖选项“-I dir”和“-L dir”。

##### • “-I dir”

正如上表中所述，“-I dir”选项可以在头文件的搜索路径列表中添加 dir 目录。由于 Linux 中头文件都默认放到了“/usr/include/”目录下，因此，当用户要添加放置在其他位置的头文件时，就可以通过“-I dir”选项来指定，这样，Gcc 就会到相应的位置查找对应的目录。比如在“/root/workplace/Gcc”下有两个文件：

```
/*hello1.c*/
#include<my.h>
int main()
{
printf("Hello!!\n");
return 0;
```

```

}
/*my.h*/
#include<stdio.h>

```

这样，就可在 Gcc 命令行中加入“-I”选项：

```

[root@localhost Gcc] gcc hello1.c -I /root/workplace/Gcc/ -o
hello1

```

这样，gcc 就能够执行出正确结果。

### 小知识

在 include 语句中，“<>”表示在标准路径中搜索头文件，而““ ””表示在本目录中搜索。故在上例中，可把 hello1.c 的“#include<my.h>”改为“#include“my.h””，就不需要加上“-I”选项了。

```

· “-L dir”

```

选项“-L dir”的功能与“-I dir”类似，能够在库文件的搜索路径列表中添加 dir 目录。例如有程序 hello\_sq.c 需要用到目录“/root/workplace/Gcc/lib”下的一个动态库 libsung.so，则需要键入如下命令即可：

```

[root@localhost Gcc] gcc hello_sq.c -L
/root/workplace/Gcc/lib -lsung -o hello_sq

```

需要注意的是，“-I dir”和“-L dir”都只是指定了路径，而没有指定文件，因此不能在路径中包含文件名。

另外值得详细解释一下的是“-l”选项，它指示 Gcc 去连接库文件 libsung.so。由于在 Linux 下的库文件命名时有一个规定：必须以 l、i、b 这 3 个字母开头。因此在用 -l 选项指定链接的库文件名时可以省去 l、i、b 3 个字母。也就是说 Gcc 在对“-lsung”进行处理时，会自动去链接名为 libsung.so 的文件。

## 2. （2）告警和出错选项

Gcc 的告警和出错选项如表 3-6 所示。

表 3-6 Gcc 的告警和出错选项

选 项	含 义
-ansi	支持符合 ANSI 标准的 C 程序
-pedantic	允许发出 ANSI C 标准所列的全部警告信息
-pedantic-err or	允许发出 ANSI C 标准所列的全部错误信息
-w	关闭所有告警
-Wall	允许发出 Gcc 提供的所有有用的报警信息
-werror	把所有的告警信息转化为错误信息，并在告警发生时终止编译过程

下面结合实例对这几个告警和出错选项进行简单的讲解。

如有以下程序段：

```
#include<stdio.h>
void main()
{
    long long tmp = 1;
    printf("This is a bad code!\n");
    return 0;
}
```

这是一个很糟糕的程序，读者可以考虑一下 存在哪些问题？

- “-ansi”

该选项强制 Gcc 生成标准语法所要求的告警信息，尽管这还并不能保证所有没有警告的程序都是符合 ANSI C 标准的。运行结果如下所示：

```
[root@localhost Gcc]# gcc -ansi warning.c -o warning
warning.c: 在函数“main”中：
warning.c:7 警告：在无返回值的函数中，“return”带返回值
warning.c:4 警告：“main”的返回类型不是“int”
可以看出，该选项并没有找到“long long”这个无效数据类型的错误。
```

- “-pedantic”

允许发出 ANSI C 标准所列的全部警告信息，同样也保证所有没有警告的程序都是符合 ANSI C 标准的。其运行结果如下所示：

```
[root@localhost Gcc]# gcc -pedantic warning.c -o warning
warning.c: 在函数“main”中：
warning.c:5 警告：ISO C90 不支持“long long”
warning.c:7 警告：在无返回值的函数中，“return”带返回值
warning.c:4 警告：“main”的返回类型不是“int”
可以看出，使用该选项找出了“long long”这个无效数据类型的错误。
```

- “-Wall”

允许发出 Gcc 能够提供的所有的有用的报警信息。该选项的运行结果如下所示：

```
[root@localhost Gcc]# gcc -Wall warning.c -o warning
warning.c:4 警告：“main”的返回类型不是“int”
warning.c: 在函数“main”中：
warning.c:7 警告：在无返回值的函数中，“return”带返回值
warning.c:5 警告：未使用的变量“tmp”
```

使用“-Wall”选项找出了未使用的 tmp 变量，但它并没有找出无效数据类型的错误。另外，Gcc 还可以利用选项对单独的常见错误分别指定警告，有关具体选项的含义感兴趣的读者可以查看 Gcc 手册进行学习。

### 3. 优化选项

GCC 具有优化代码的功能，代码的优化是一项比较复杂的工作，它可归为：源代码级优化、速度与空间的权衡、执行代码的调度。

GCC 提供了下列优化选项：

-O0：默认不优化（若要生成调试信息，最好不优化）

-O1：简单优化，不进行速度与空间的权衡优化；

-O2：进一步的优化，包括了调度。（若要优化，该选项最适合，它是 GNU 发布软件的默认优化级别；

-O3：鸡肋，兴许使程序速度更慢；

-funroll-loops：展开循环，会使可执行文件增大，而速度是否增加取决于特定环境；

-Os：生成最小执行文件；

一般来说，调试时不优化，一般的优化选项用-O2（gcc 允许-g 与-O2 联用，这也是 GNU 软件包发布的默认选项），embedded 可以考虑-Os。

### 4. 体系结构相关选项

Gcc 的体系结构相关选项如表 3-7 所示。

表 3-7 Gcc 体系结构相关选项

选 项	含 义
-mcpu=type	针对不同的 CPU 使用相应的 CPU 指令。可选择的 type 有 i386、i486、pentium 及 i686 等
-mieee-fp	使用 IEEE 标准进行浮点数的比较
-mno-ieee-fp	不使用 IEEE 标准进行浮点数的比较
-msoft-float	输出包含浮点库调用的目标代码
-mshort	把 int 类型作为 16 位处理，相当于 short int
-mrtld	强行将函数参数个数固定的函数用 ret NUM 返回，节省调用函数的一条指令

这些体系结构相关选项在嵌入式的设计中会有较多的应用，读者需根据不同体系结构将对应的选项进行组合处理。在本书后面涉及到具体实例会有针对性的讲解。

### 3.3.3 gcc 生成静态库和动态库

我们通常把一些公用函数制作成函数库，供其它程序使用。函数库分为静态库和动态库两种。

静态库在程序编译时会被连接到目标代码中，程序运行时将不再需要该静态库。动

态库在程序编译时并不会被连接到目标代码中，而是在程序运行是才被载入，因此在程序运行时还需要动态库存在。

在创建函数库前，我们先来准备举例用的源程序，并将函数库的源程序编译成.o文件。

### 1. 第1步：编辑得到举例的程序--hello.h、hello.c和main.c;

hello.c(见程序2)是函数库的源程序，其中包含公用函数hello，该函数将在屏幕上输出“Hello XXX!”。hello.h(见程序1)为该函数库的头文件。main.c(见程序3)为测试库文件的主程序，在主程序中调用了公用函数hello。

程序1: hello.h

```
#ifndef HELLO_H
#define HELLO_H
void hello(const char *name);
#endif //HELLO_H
```

程序2: hello.c

```
#include <stdio.h>
void hello(const char *name)
{
    printf("Hello %s!\n", name);
}
```

程序3: main.c

```
#include "hello.h"
int main()
{
    hello("everyone");
    return 0;
}
```

### 2. 第2步：将hello.c编译成.o文件;

无论静态库，还是动态库，都是由.o文件创建的。因此，我们必须将源程序hello.c通过gcc先编译成.o文件。

在系统提示符下键入以下命令得到hello.o文件。

```
# gcc -c hello.c
```

我们运行ls命令看看是否生存了hello.o文件。

```
# ls
hello.c hello.h hello.o main.c
```

在ls命令结果中，我们看到了hello.o文件，本步操作完成。

下面我们先来看看如何创建静态库，以及使用它。

### 3. 第3步：由.o文件创建静态库;

静态库文件名的命名规范是以 lib 为前缀，紧接着跟静态库名，扩展名为 .a。例如：我们将创建的静态库名为 myhello，则静态库文件名就是 libmyhello.a。在创建和使用静态库时，需要注意这点。创建静态库用 ar 命令。

在系统提示符下键入以下命令将创建静态库文件 libmyhello.a。

```
# ar crv libmyhello.a hello.o
```

我们同样运行 ls 命令查看结果：

```
# ls
hello.c hello.h hello.o libmyhello.a main.c
```

ls 命令结果中有 libmyhello.a。

#### 4. 第4步：在程序中使用静态库；

静态库制作完了，如何使用它内部的函数呢？只需要在使用到这些公用函数的源程序中包含这些公用函数的原型声明，然后在用 gcc 命令生成目标文件时指明静态库名，gcc 将会从静态库中将公用函数连接到目标文件中。注意，gcc 会在静态库名前加上前缀 lib，然后追加扩展名.a 得到的静态库文件名来查找静态库文件。

在程序 3:main.c 中，我们包含了静态库的头文件 hello.h，然后在主程序 main 中直接调用公用函数 hello。下面先生成目标程序 hello，然后运行 hello 程序看看结果如何。

```
# gcc -o hello main.c -L. -lmyhello
# gcc main.c libmyhello.a -o main
# ./hello
Hello everyone!
```

我们删除静态库文件试试公用函数 hello 是否真的连接到目标文件 hello 中了。

```
# rm libmyhello.a
rm: remove regular file `libmyhello.a'? y
# ./hello
Hello everyone!
```

程序照常运行，静态库中的公用函数已经连接到目标文件中了。

在第 1、2 步我们生成了 .o 文件，第 3、4 步进行了静态库编译，接下来我们继续看看如何在 Linux 中创建动态库。我们还是从 .o 文件开始。

#### 5. 第5步：由.o文件创建动态库文件；

动态库文件名命名规范和静态库文件名命名规范类似，也是在动态库名增加前缀 lib，但其文件扩展名为 .so。例如：我们将创建的动态库名为 myhello，则动态库文件名就是 libmyhello.so。

用 gcc 来创建动态库。在系统提示符下键入以下命令得到动态库文件 libmyhello.so。

```
# gcc -shared -fPIC -o libmyhello.so hello.o
```

我们照样使用 ls 命令看看动态库文件是否生成。

```
# ls
hello.c hello.h hello.o libmyhello.so main.c
```

## 6. 第6步：在程序中使用动态库；

在程序中使用动态库和使用静态库完全一样，也是在使用到这些公用函数的源程序中包含这些公用函数的原型声明，然后在用 gcc 命令生成目标文件时指明动态库名进行编译。我们先运行 gcc 命令生成目标文件，再运行它看看结果。

```
# gcc -o hello main.c -L. -lmyhello
# ./hello
./hello: error while loading shared libraries: libmyhello.so:
cannot open shared object file: No such file or directory
```

出错。错误提示找不到动态库文件 libmyhello.so。程序在运行时，会在 /usr/lib 和 /lib 等目录中查找需要的动态库文件。若找到，则载入动态库，否则将提示类似上述错误而终止程序运行。我们将文件 libmyhello.so 复制到目录 /usr/lib 中，再试试。

```
# mv libmyhello.so /usr/lib
# ./hello
Hello everyone!
```

成功。这也进一步说明了动态库在程序运行时是需要的。

我们回过头看看，发现使用静态库和使用动态库编译成目标程序使用的 gcc 命令完全一样，那当静态库和动态库同名时，gcc 命令会使用哪个库文件呢？

先删除.c 和.h 外的所有文件，恢复成我们刚刚编辑完举例程序状态。

```
# rm -f hello hello.o /usr/lib/libmyhello.so
# ls
hello.c hello.h main.c
```

再来创建静态库文件 libmyhello.a 和动态库文件 libmyhello.so。

```
# gcc -c hello.c
# ar cr libmyhello.a hello.o
# gcc -shared -fPIC -o libmyhello.so hello.o
# ls
hello.c hello.h hello.o libmyhello.a libmyhello.so main.c
```

通过上述最后一条 ls 命令，可以发现静态库文件 libmyhello.a 和动态库文件 libmyhello.so 都已经生成，并都在当前目录中。然后，我们运行 gcc 命令来使用函数库 myhello 生成目标文件 hello，并运行程序 hello。

```
# gcc -o hello main.c -L. -lmyhello
# ./hello
./hello: error while loading shared libraries: libmyhello.so:
cannot open shared object file: No such file or directory
```

从程序 hello 运行的结果中很容易知道，当静态库和动态库同名时，gcc 命令将优

先使用动态库。

**Note:**

编译参数解析，最主要的是 GCC 命令行的一个选项：

-shared 该选项指定生成动态连接库（让连接器生成 T 类型的导出符号表，有时候也生成弱连接 W 类型的导出符号），不用该标志外部程序无法连接。相当于一个可执行文件

-fPIC：表示编译为位置独立的代码，不用此选项的话编译后的代码是位置相关的所以动态载入时是通过代码拷贝的方式来满足不同进程的需要，而不能达到真正代码段共享的目的。

-L：表示要连接的库在当前目录中

-ltest：编译器查找动态连接库时有隐含的命名规则，即在给出的名字前面加上 lib，后面加上 .so 来确定库的名称

LD\_LIBRARY\_PATH：这个环境变量指示动态连接器可以装载动态库的路径。当然如果有 root 权限的话，可以修改/etc/ld.so.conf 文件，然后调/sbin/ldconfig 来达到同样的目的，不过如果没有 root 权限，那么只能采用输 LD\_LIBRARY\_PATH 的方法了。

调用动态库的时候有几个问题会经常碰到，有时，明明已经将库的头文件所在目录通过“-I”include 进来了，库所在文件通过“-L”参数引导，并指定了“-l”的库名，但通过 ldd 命令察看时，就是死活找不到你指定链接的 so 文件，这时你要作的就是通过修改 LD\_LIBRARY\_PATH 或者/etc/ld.so.conf 文件来指定动态库的目录。通常这样做就可以解决库无法链接的问题了。

### 3.3.4 教你解决 GCC 错误

Gcc 编译器如果发现源程序中有错误，就无法继续进行，也无法生成最终的可执行文件。为了便于修改，gcc 给出错误资讯，我们必须对这些错误资讯逐个进行分析、处理，并修改相应的语言，才能保证源代码的正确编译连接。gcc 给出的错误资讯一般可以分为四大类，下面我们分别讨论其产生的原因和对策。

#### 1. 第一类：C 语法错误

错误资讯：文件 source.c 中第 n 行有语法错误(syntax error)。这种类型的错误，一般都是 C 语言的语法错误，应该仔细检查源代码文件中第 n 行及该行之前的程序，有时也需要对该文件所包含的头文件进行检查。有些情况下，一个很简单的语法错误，gcc 会给出一大堆错误，我们最主要的是要保持清醒的头脑，不要被其吓倒，必要的时候再参考一下 C 语言的基本教材。

#### 2. 第二类：头文件错误

错误资讯：找不到头文件 head.h(Can not find include file head.h)。这类错误是源代码文件中的包含头文件有问题，可能的原因有头文件名错误、指定的头文件所在目录名错误等，也可能是错误地使用了双引号和尖括号。



### 3. 第三类：档案库错误

错误资讯：连接程序找不到所需的函数库，例如：

```
ld: -lm: No such file or directory
```

这类错误是与目标文件相连接的函数库有错误，可能的原因是函数库名错误、指定的函数库所在目录名称错误等，检查的方法是使用 find 命令在可能的目录中寻找相应的函数库名，确定档案库及目录的名称并修改程序中及编译选项中的名称。

### 4. 第四类：未定义符号

错误资讯：有未定义的符号(Undefined symbol)。这类错误是在连接过程中出现的，可能有两种原因：一是使用者自己定义的函数或者全局变量所在源代码文件，没有被编译、连接，或者干脆还没有定义，这需要使用者根据实际情况修改源程序，给出全局变量或者函数的定义体；二是未定义的符号是一个标准的库函数，在源程序中使用了该库函数，而连接过程中还没有给定相应的函数库的名称，或者是该档案库的目录名称有问题，这时需要使用档案库维护命令 ar 检查我们需要的库函数到底位于哪一个函数库中，确定之后，修改 gcc 连接选项中的 -l 和 -L 项。

排除编译、连接过程中的错误，应该说这只是程序设计中最简单、最基本的一个步骤，可以说只是开了个头。这个过程错误，只是我们在使用 C 语言描述一个算法中所产生的错误，是比较容易排除的。我们写一个程序，到编译、连接通过为止，应该说刚刚开始，程序在运行过程中所出现的问题，是算法设计有问题，说得更玄点是对问题的认识和理解不够，还需要更加深入地测试、调试和修改。一个程序，稍为复杂的程序，往往要经过多次的编译、连接和测试、修改。下面我们学习的程序维护、调试工具和版本维护就是在程序调试、测试过程中使用的，用来解决调测阶段所出现的问题。

## 3.4 3.4 Gdb 调试器

调试是所有程序员都会面临的问题。如何提高程序员的调试效率，更好更快地定位程序中的问题从而加快程序开发的进度，是大家共同要面对问题。就如读者熟知的 Windows 下的一些调试工具，如 VC 自带的如设置断点、单步跟踪等，都受到了广大用户的赞赏。那么，在 Linux 下有什么很好的调试工具呢？

本文所介绍的 Gdb 调试器是一款 GNU 开发组织并发布的 UNIX/Linux 下的程序调试工具。虽然，它没有图形化的友好界面，但是它强大的功能也足以与微软的 VC 工具等媲美。

下面就请跟随笔者一步步学习 Gdb 调试器。

### 3.4.1 Gdb 使用流程

这里给出了一个短小的程序，由此带领读者熟悉一下 Gdb 的使用流程。建议读者能够实际动手操作。

首先，打开 Linux 下的编辑器 Vi，编辑如下代码。

```
/******
                                Copyright (C), 2001-2011, GEC Co., Ltd.
File Name      : net_parse.c
Version       : Initial Draft
Author        : liuzhigang
Created       : 2010/5/3
Description   : 网络模块解析网络数据的文件
History      :
1.Date       : 2010/5/3
   Author    : liuzhigang
Modification: Created file
******/
#include <stdio.h>

int sum(int m);

int main()
{
    int i,n=0;
    sum(50);
    for(i=1; i<=50; i++)
    {
        n += i;
    }
    printf("The sum of 1-50 is %d \n", n );
}

int sum(int m)
{
    int i,n=0;
    for(i=1; i<=m;i++)
    n += i;
    printf("The sum of 1-m is %d\n", n);
}
```

```

}

/*****
      Copyright (C), 2001-2011, GEC Co., Ltd.
File Name      : net_parse.c
Version        : Initial Draft
Author         : liuzhigang
Created        : 2010/5/3
Description    : xxxxxx
History        :
1.Date         : 2010/5/3
   Author      : xxxxx
   Modification: Create file
*****/

#include <stdio.h>

int sum(int m);

/*****
Prototype      : main
Description    : ???
Input          : None
Output         : None
Return Value   :
*****/
int main()
{
    int i, n = 0;
    int ret;

    /* xxxxxxxxxxxxxxxxxxxxxxxxx */
    ret = sum(50);
    if (0 != ret)
    {
        printf("sum funciton error!\n");
        return -1;
    }
}

```

```

    for(i = 1; i <= 50; i++)
    {
        n += i;
    }

    printf("The sum of 1-50 is %d \n", n );

    return 0;
}

/*****
Prototype      : sum
Description    : xxxxx
Input          : int m
Output         : None
Return Value   :
*****/
int sum(int m)
{
    int i, n = 0;

    /* 计算 xxxxx */
    for(i = 1; i <= m; i++)
    {
        n += i;
    }

    printf("The sum of 1-m is %d\n", n);

    return 0;
}

```

在保存退出后首先使用 Gcc 对 test.c 进行编译,注意一定要加上选项“-g”这个选项,这样编译出的可执行代码中才包含调试信息,否则之后 Gdb 无法载入该可执行文件。

```
[root@localhost Gdb]# gcc -g test.c -o test
```

虽然这段程序没有错误,但调试完全正确的程序可以更加了解 Gdb 的使用流程。接

下来就是启动 Gdb 进行调试。注意，Gdb 进行调试的是可执行文件，而不是如“.c”的源代码，因此，需要先通过 Gcc 编译生成可执行文件才能用 Gdb 进行调试。

```
[root@localhost Gdb]# gdb test
GNU Gdb Red Hat Linux (6.3.0.0-1.21rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License,
and you are
welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty"
for details.
This GDB was configured as "i386-redhat-linux-gnu"...Using host
libthread_db
library "/lib/libthread_db.so.1".
(gdb)
```

可以看出，在 Gdb 的启动画面中指出了 Gdb 的版本号、使用的库文件等信息，接下来就进入了由“(gdb)”开头的命令行界面了。

### 1. 查看文件

在 Gdb 中键入“l”（list）就可以查看所载入的文件，如下所示：

注意

在 Gdb 的命令中都可使用缩略形式的命令，如“l”代便“list”，“b”代表“breakpoint”，“p”代表“print”等，读者也可使用“help”命令来查看帮助信息。

```
(Gdb) l
1 #include <stdio.h>
2 int sum(int m);
3 int main()
4 {
5     int i,n=0;
6     sum(50);
7     for(i=1; i<=50; i++)
8 {
9     n += i;
10 }
(Gdb) l
11 printf("The sum of 1~50 is %d \n", n );
12
```

```

13 }
14 int sum(int m)
15 {
16     int i,n=0;
17     for(i=1; i<=m;i++)
18         n += i;
19     printf("The sum of 1~m is = %d\n", n);
20 }

```

可以看出，Gdb 列出的源代码中明确地给出了对应的行号，这样就可以大大地方便代码的定位。

## 2. 设置断点

设置断点是调试程序中是一个非常重要的手段，它可以使程序到一定位置暂停它的运行。因此，程序员在该位置可以方便地查看变量的值、堆栈等情况，从而找出代码的症结所在。在 Gdb 中设置断点非常简单，只需在“b”后加入对应的行号即可（这是最常用的方式，另外还有其他方式设置断点）。如下所示：

```

(Gdb) b 6
Breakpoint 1 at 0x804846d: file test.c, line 6.

```

要注意的是，在 Gdb 中利用行号设置断点是指代码运行到对应行之前将其停止，如上例中，代码运行到第 5 行之前就暂停（并没有运行第 5 行）。

## 3. 查看断点情况

在设置完断点之后，用户可以键入“info b”来查看设置断点的情况，在 Gdb 中可以设置多个断点。

```

(Gdb) info b
Num Type Disp Enb Address What
1 breakpoint keep y 0x0804846d in main at test.c:6

```

## 4. 运行代码

接下来就可运行代码，Gdb 默认从首行开始运行代码，可键入“r”（run）即可（若想从程序中指定行开始运行，可在 r 后面加上行号）。

```

(Gdb) r
Starting program: /root/workplace/Gdb/test
Reading symbols from shared object read from target memory done.
Loaded system supplied DSO at 0x5fb000
Breakpoint 1, main () at test.c:6
6 sum(50);

```

可以看到，程序运行到断点处就停止了。

## 5. 查看变量值

在程序停止运行之后，程序员所要做的工作是查看断点处的相关的变量值。在 Gdb 中只需键入 “p” + 变量值即可，如下所示：

```
(Gdb) p n
$1 = 0
(Gdb) p i
$2 = 134518440
```

在此处，为什么变量 “i” 的值为如此奇怪的一个数字呢？原因就在于程序是在断点设置的对应行之前停止的，那么在此时，并没把 “i” 的数值赋为零，而只是一个随机的数字。但变量 “n” 是在第四行赋值的，故在此时已经为零。

小技巧

Gdb 在显示变量值时都会在对应变值之前加上 “\$N” 标记，它是当前变量值的引用标记，所以以后若想再次引用此变量就可以直接写作 “\$N”，而无需写冗长的变量名。

## 6. 单步运行

单步运行可以使用命令 “n” (next) 或 “s” (step)，它们之间的区别是：若有函数调用的时候，“s” 会进入该函数而 “n” 不会进入该函数。因此，“s” 就类似于 VC 等工具中的 “step in”，“n” 类似与 VC 等工具中的 “step over”。它们的使用如下所示：

```
(Gdb) n
The sum of 1-m is 1275
7 for(i=1; i<=50; i++)
(Gdb) s
sum (m=50) at test.c:16
16 int i,n=0;
```

可见，使用 “n” 后，程序显示函数 sum 的运行结果并向下执行，而使用 “s” 后则进入到 sum 函数之中单步运行。

## 7. 恢复程序运行

在查看完所需变量及堆栈情况后，就可以使用命令 “c” (continue) 恢复程序的正常运行了。这时，它会把剩余还未执行的程序执行完，并显示剩余程序中的执行结果。以下是之前使用 “n” 命令恢复执行后的结果：

```
(Gdb) c
Continuing.
The sum of 1-50 is :1275
Program exited with code 031.
```

可以看出，程序在运行完后退出，之后程序处于 “停止状态”。

### 小知识

在 Gdb 中，程序的运行状态有 “运行”、“暂停” 和 “停止” 3 种，其中 “暂停”

状态为程序遇到断点或观察点之类的，程序暂时停止运行，而此时函数的地址、函数参数、函数内的局部变量都会被压入“栈”（Stack）中。故在这种状态下可以查看函数的变量值等各种属性。但在函数处于“停止”状态之后，“栈”就会自动撤销，也就无法查看各种信息了。

### 3.4.2 Gdb 基本命令

Gdb 的命令可以通过查看 help 进行查找，由于 Gdb 的命令很多，因此 Gdb 的 help 将其分成了很多种类（class），用户可以通过进一步查看相关 class 找到相应的命令。如下所示：

```
(gdb) help
List of classes of commands:
aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
...
Type "help" followed by a class name for a list of commands in
that class.
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
```

上述列出了 Gdb 各个分类的命令。接下来可以具体查找各分类种的命令。如下所示：

```
(gdb) help data
Examining data.
List of commands:
call -- Call a function in the program
delete display -- Cancel some expressions to be displayed when
program stops
delete mem -- Delete memory region
disable display -- Disable some expressions to be displayed when
program stops
...
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
至此，若用户想要查找 call 命令，就可键入“help call”。
(gdb) help call
```



```
Call a function in the program.
The argument is the function name and arguments, in the notation
of the
current working language. The result is printed and saved in the
value
history, if it is not void.
```

当然，若用户已知命令名，直接键入“help [command]”也是可以的。

Gdb 中的命令主要可以分为以下几类：工作环境相关命令、设置断点与恢复命令、源代码查看命令、查看运行数据相关命令及修改运行参数命令。下面就分别对这几类的命令进行讲解。

1. 工作环境相关命令

Gdb 中不仅可以调试所运行的程序，而且还可以对程序相关的工作环境进行相应的设定，甚至还可以使用 shell 中的命令进行相关的操作，其功能极其强大。Gdb 的常见工作环境相关命令如表 3 8 所示。

表 3 8 Gdb 工作相关命令

命令格式	含 义
set args 运行时的参数	指定运行时参数，如 set args 2
show args	查看设置好的运行参数
path dir	设定程序的运行路径
show paths	查看程序的运行路径
set enVironment var [=value]	设置环境变量
show enVironment [var]	查看环境变量
cd dir	进入到 dir 目录，相当于 shell 中的 cd 命令
pwd	显示当前工作目录
shell command	运行 shell 的 command 命令

2. 设置断点与恢复命令

Gdb 中设置断点与恢复的常见命令如表 3 9 所示。

表 3 9 Gdb 设置断点与恢复相关命令

命令格式	含 义
bnfo b	查看所设断点

break 行号或函数名 <条件表达式>	设置断点
tbreak 行号或函数名 <条件表达式>	设置临时断点，到达后被自动删除
delete [断点号]	删除指定断点，其断点号为“info b”中的第一栏。若缺省断点号则删除所有断点
disable [断点号]	停止指定断点，使用“info b”仍能查看此断点。同 delete 一样，省断点号则停止所有断点
enable [断点号]	激活指定断点，即激活被 disable 停止的断点
condition [断点号] <条件表达式>	修改对应断点的条件
ignore [断点号]<num>	在程序执行中，忽略对应断点 num 次
step	单步恢复程序运行，且进入函数调用
next	单步恢复程序运行，但不进入函数调用
finish	运行程序，直到当前函数完成返回
c	继续执行函数，直到函数结束或遇到新的断点

由于设置断点在 Gdb 的调试中非常重要，所以在此再着重讲解一下 Gdb 中设置断点的方法。Gdb 中设置断点有多种方式：其一是按行设置断点，设置方法在前面已经指出，在此就不重复了。另外还可以设置函数断点和条件断点，在此结合上一小节的代码，具体介绍后两种设置断点的方法。

#### ① 函数断点

Gdb 中按函数设置断点只需要把函数名列在命令“b”之后，如下所示：

```
(gdb) b sum
Breakpoint 1 at 0x80484ba: file test.c, line 16.
(gdb) info b
Num Type Disp Enb Address What
1 breakpoint keep y 0x080484ba in sum at test.c:16
```

要注意的是，此时的断点实际是在函数的定义处，也就是在 16 行处（注意第 16 行还未执行）。

#### ② 条件断点

Gdb 中设置条件断点的格式为：b 行数或函数名 if 表达式。具体实例如下面所示：

```
(gdb) b 8 if i==10
Breakpoint 1 at 0x804848c: file test.c, line 8.
(gdb) info b
Num Type Disp Enb Address What
1 breakpoint keep y 0x0804848c in main at test.c:8
stop only if i == 10
```

```
(gdb) r
Starting program: /home/yul/test
The sum of 1-m is 1275
Breakpoint 1, main () at test.c:9
9 n += i;
(gdb) p i
$1 = 10
```

可以看到，该例中在第 8 行（也就是运行完第 7 行的 for 循环）设置了一个“i==0”的条件断点，在程序运行之后可以看出，程序确实在 i 为 10 的时候暂停运行。

### 3. Gdb 中源码查看相关命令

在 Gdb 中可以查看源码以方便其他操作，它的常见相关命令如表 3 10 所示。

表 3 10 Gdb 源码查看相关命令

命令格式	含 义
list <行号> <函数名>	查看指定位置代码
file [文件名]	加载指定文件
forward-search 正则表达式	源代码前向搜索
reverse-search 正则表达式	源代码后向搜索
dir dir	停止路径名
show directories	显示定义的源文件搜索路径
info line	显示加载到 Gdb 内存中的代码

### 4. Gdb 中查看运行数据相关命令

Gdb 中查看运行数据是指当程序处于“运行”或“暂停”状态时，可以查看的变量及表达式的信息，其常见命令如表 3 11 所示：

表 3 11 Gdb 查看运行数据相关命令

命令格式	含 义
print 表达式 变量	查看程序运行时对应表达式和变量的值
x <n/f/u>	查看内存变量内容。其中 n 为整数表示显示内存的长度，f 表示显示的格式，u 表示从当前地址往后请求显示的字节数
display 表达式	设定在单步运行或其他情况中，自动显示的对应表达式的内容

### 5. Gdb 中修改运行参数相关命令

Gdb 还可以修改运行时的参数，并使该变量按照用户当前输入值继续运行。它的设置方式为：在单步执行的过程中，键入命令“set 变量=设定值”。这样，在此之后，程序就会按照该设定的值运行了。下面，笔者结合上一节的代码将 n 的初始值设为 4，其代码如下所示：

```
(Gdb) b 7
Breakpoint 5 at 0x804847a: file test.c, line 7.
(Gdb) r
Starting program: /home/yul/test
The sum of 1-m is 1275
Breakpoint 5, main () at test.c:7
7 for(i=1; i<=50; i++)
(Gdb) set n=4
(Gdb) c
Continuing.
The sum of 1-50 is 1279
Program exited with code 031.
```

可以看到，最后的运行结果的确比之前的值大了 4。

Gdb 的使用切记点：

- 在 Gcc 编译选项中一定要加入“-g”。
- 只有在代码处于“运行”或“暂停”状态时才能查看变量值。
- 设置断点后程序在指定行之前停止。

### 3.4.3 用 Gdb 调试有问题的程序

#### 1. 实验目的

通过调试一个有问题的程序，使读者进一步熟练使用 Vi 操作，而且熟练掌握 Gcc 编译命令及 Gdb 的调试命令，通过对有问题程序的跟踪调试，进一步提高发现问题和解决问题的能力。这是一个很小的程序，只有 35 行，希望读者能够认真调试。

#### 2. 实验内容

(1) 使用 Vi 编辑器，将以下代码输入到名为 greet.c 的文件中。此代码的原意为输出倒序 main 函数中定义的字符串，但其结果显示没有输出。代码如下所示：

```
#include <stdio.h>
int display1(char *string);
int display2(char *string);
int main ()
{
```

```

char string[] = "Embedded Linux";
display1 (string);
display2 (string);
}
int display1 (char *string)
{
printf ("The original string is %s \n", string);
}
int display2 (char *string1)
{
char *string2;
int size,i;
size = strlen (string1);
string2 = (char *) malloc (size + 1);
for (i = 0; i < size; i++)
string2[size - i] = string1[i];
string2[size+1] = ' ';
printf("The string afterward is %s\n",string2);
}

```

- (2) 使用 Gcc 编译这段代码，注意要加上“-g”选项方便之后的调试。
- (3) 运行生成的可执行文件，观察其运行结果。
- (4) 使用 Gdb 调试程序，通过设置断点、单步跟踪，一步步找出错误所在。
- (5) 纠正错误，更改源程序并得到正确的结果。

### 3. 实验步骤

- (1) 在工作目录上新建文件 greet.c，并用 Vi 启动：vi greet.c。
- (2) 在 Vi 中输入以上代码。
- (3) 在 Vi 中保存并退出：wq。
- (4) 用 Gcc 进行编译：gcc -g greet.c -o greet。
- (5) 运行 greet：./greet，输出为：

```

The original string is Embedded Linux
The string afterward is

```

可见，该程序没有能够倒序输出。

- (6) 启动 Gdb 调试：gdb greet。
- (7) 查看源代码，使用命令“l”。
- (8) 在 30 行（for 循环处）设置断点，使用命令“b 30”。
- (9) 在 33 行（printf 函数处）设置断点，使用命令“b 33”。
- (10) 查看断点处设置情况，使用命令“info b”。

(11) 运行代码，使用命令 “r”。

(12) 单步运行代码，使用命令 “n”。

(13) 查看暂停点变量值，使用命令 “p string2[size - i]”。

(14) 继续单步运行代码数次，并使用命令查看，发现 string2[size-1] 的值正确。

(15) 继续程序的运行，使用命令 “c”。

(16) 程序在 printf 前停止运行，此时依次查看 string2[0]、string2[1]…，发现 string[0] 并没有被正确赋值，而后面的复制都是正确的，这时，定位程序第 31 行，发现程序运行结果错误的原因在于 “size-1”。由于 i 只能增到 “size-1”，这样 string2[0] 就永远不能被赋值而保持 NULL，故输不出任何结果。

(17) 退出 Gdb，使用命令 q。

(18) 重新编辑 greet.c，把其中的 “string2[size - i] = string1[i]” 改为 “string2[size - i - 1] =string1[i];” 即可。

(19) 使用 Gcc 重新编译：gcc -g greet.c -o greet。

(20) 查看运行结果：./greet

```
The original string is Embedded Linux
The string afterward is xuniL deddedbmE
```

这时，正确输出结果。

#### 4. 实验结果

将原来有错的程序经过 Gdb 调试，找出问题所在，并修改源代码，输出正确的倒序显示字符串的结果。

## 3.5 Make 工程管理器

### 3.5.1 为什么要学 Makefile

到此为止，读者已经了解了如何在 Linux 下使用编辑器编写代码，如何使用 Gcc 把代码编译成可执行文件，还学习了如何使用 Gdb 来调试程序，那么，所有的工作看似已经完成了，为什么还需要 Make 这个工程管理器呢？

所谓工程管理器，顾名思义，是指管理较多的文件的。读者可以试想一下，有一个上百个代码文件构成的项目，如果其中只有一个或少数几个文件进行了修改的话，按照之前所学的 Gcc 编译工具，就不得不把这所有的文件重新编译一遍，因为编译器并不知道哪些文件是最近更新的，而只知道需要包含这些文件才能把源代码编译成可执行文件，于是，程序员就不能不再重新输入数目如此庞大的文件名以完成最后的编译工作。

但是，请读者仔细回想一下本书所阐述的编译过程，编译过程是分为编译、汇编、

链接不同阶段的,其中编译阶段仅检查语法错误以及函数与变量的声明是否正确声明了,在链接阶段则主要完成是函数链接和全局变量的链接。因此,那些没有改动的源代码根本不需要重新编译,而只要把它们重新链接进去就可以了。所以,人们就希望有一个工程管理器能够自动识别更新了的文件代码,同时又不需要重复输入冗长的命令行,这样,Make 工程管理器也就应运而生了。

实际上,Make 工程管理器也就是个“自动编译管理器”,这里的“自动”是指它能够根据文件的时间戳自动发现更新过的文件而减少编译的工作量,同时,它通过读入 Makefile 文件的内容来执行大量的编译工作。用户只需要编写一次简单的编译语句就可以了。它大大提高了实际项目的工作效率,而且几乎所有 Linux 下的项目编程均会涉及它,希望读者能够认真学习本节内容。

### 3.5.2 make 工作步骤

GNU 的 make 工作时的步骤为:

- 1、读入所有的 Makefile。
- 2、读入被 include 的其它 Makefile。
- 3、初始化文件中的变量。
- 4、推导隐晦规则,并分析所有规则。
- 5、为所有目标文件创建依赖关系链。
- 6、根据依赖关系,决定哪些目标要重新生成。
- 7、执行生成命令。

1-5 步为第一个阶段,6-7 为第二个阶段。第一个阶段中,如果定义的变量被使用了,那么,make 会把其展开在使用的位置。但 make 并不会完全马上展开,make 使用的是拖延战术,如果变量出现在依赖关系的规则中,那么仅当这条依赖被决定要使用时,变量才会在其内部展开。

当然,这个工作方式你不一定要清楚,但是知道这个方式你也会对 make 更为熟悉。有了这个基础,后续部分也就容易看懂了。

### 3.5.3 简单 Make 程序创建

使用 Vi 编辑器,将以下代码输入到文件中。

我们有一个程序由 5 个文件组成,源代码如下:

```
/*main.c*/  
#include "mytool1.h"  
#include "mytool2.h"
```

```

int main()
{
mytool1_print("hello mytool1!");
mytool2_print("hello mytool2!");
return 0;
}
/*mytool1.c*/
#include "mytool1.h"
#include <stdio.h>
void mytool1_print(char *print_str)
{
printf("This is mytool1 print : %s ",print_str);
}
/*mytool1.h*/
#ifndef _MYTOOL_1_H
#define _MYTOOL_1_H
void mytool1_print(char *print_str);
#endif
/*mytool2.c*/
#include "mytool2.h"
#include <stdio.h>
void mytool2_print(char *print_str)
{
printf("This is mytool2 print : %s ",print_str);
}
/*mytool2.h*/
#ifndef _MYTOOL_2_H
#define _MYTOOL_2_H
void mytool2_print(char *print_str);
#endif

```

常规法写第一个 Makefile

```

main:main.o mytool1.o mytool2.o
gcc -o main main.o mytool1.o mytool2.o
main.o:main.c mytool1.h mytool2.h
gcc -c main.c
mytool1.o:mytool1.c mytool1.h
gcc -c mytool1.c
mytool2.o:mytool2.c mytool2.h

```



```
gcc -c mytool2.c
clean:
rm -f *.o main
```

在 shell 提示符下输入 make，执行显示：

```
gcc -c main.c
gcc -c mytool1.c
gcc -c mytool2.c
gcc -o main main.o mytool1.o mytool2.o
```

执行结果如下：

```
[armlinux@lqm makefile-easy]$ ./main
This is mytool1 print : hello mytool1!
This is mytool2 print : hello mytool2!
```

这只是最为初级的 Makefile，main 是我们的最终目标，main.o、mytool1.o 和 mytool2.o 是目标所依赖的源文件，下面是一条命令“gcc -o main main.o mytool1.o mytool2.o”（以 Tab 键开头）。这个规则告诉我们两件事：

1、文件的依赖关系，main 依赖于 main.o、mytool1.o 和 mytool2.o 文件，如果 main.o、mytool1.o 和 mytool2.o 的文件日期要比 main 文件日期要新，或是 main 不存在，那么依赖关系发生。

2、如何生成（或更新）main 文件。也就是那个 gcc 命令，其说明了，如何生成 main 这个文件。

目标、依赖、命令的书写格式为：

```
targets : prerequisites
command
```

或是这样：

```
targets : prerequisites ; command
command
```

小知识

command 是命令行，如果其不与“target:prerequisites”在一行，那么，必须以 [Tab 键] 开头，如果和 prerequisites 在一行，那么可以用分号做为分隔。

clean 是一个伪目标，即然我们生成了许多编译文件，我们也应该提供一个清除它们的“目标”以备完整地重编译而用。（以“make clean”来使用该目标）

因为我们并不生成“clean”这个文件。“伪目标”并不是一个文件，只是一个标签，由于“伪目标”不是文件，所以 make 无法生成它的依赖关系和决定它是否要执行。我们只有通过显示地指明这个“目标”才能让其生效。当然，“伪目标”的取名不能和文件名重名，不然其就失去了“伪目标”的意义了。

当然，为了避免和文件重名的这种情况，我们可以使用一个特殊的标记“.PHONY”来显示地指明一个目标是“伪目标”，向 make 说明，不管是否有这个文件，这个目标就是“伪目标”。

```
.PHONY : clean
```

只要有这个声明，不管是否有“clean”文件，要运行“clean”这个目标，只有“make clean”这样。于是整个过程可以这样写：

```
.PHONY: clean
clean:
rm *.o temp
```

在这个 Makefile 中有 4 个目标体 (target)，分别为 main、main.o、mytool1 和 mytool2，其中第一个目标体的依赖文件就是后三个目标体。如果用户使用命令“make main”，则 make 管理器就是找到 main 目标体开始执行。这时，make 会自动检查相关文件的时间戳。首先，在检查“main.o”、“mytool1.o”、“mytool2.o”和“main”4 个文件的时间戳之前，它会向下查找那些把“main.o”或“mytool1.o”或“mytool2.o”作为目标文件的时间戳。比如，“mytool1.o”的依赖文件为“mytool1.c”、“mytool1.h”。如果这些文件中任何一个的时间戳比“mytool1.o”新，则命令“gcc -c mytool1.c”将会执行，从而更新文件“mytool1.o”。在更新完“mytool1.o”或“mytool2.o”或“main.o”之后，make 会检查最初的“main.o”、“mytool1.o”、“mytool2.o”和“main”3 个文件，只要文件“main.o”或“mytool1.o”或“mytool2.o”中的任比文件时间戳比“main”新，则第二行命令就会被执行。这样，make 就完成了自动检查时间戳的工作，开始执行编译工作。这也就是 Make 工作的基本流程。

### 3.5.4 Makefile 改进

现在，我们对上面的 makefile 进行逐步改进，改进过程如下：

改进一：使用变量

```
OBJ=main.o mytool1.o mytool2.o
make:${OBJ}
gcc -o main ${OBJ}
main.o:main.c mytool1.h mytool2.h
gcc -c main.c
mytool1.o:mytool1.c mytool1.h
gcc -c mytool1.c
mytool2.o:mytool2.c mytool2.h
gcc -c mytool2.c
clean:
rm -f main ${OBJ}
```

Makefile 中的变量分为用户自定义变量、预定义变量、自动变量及环境变量。如上例中的 OBJ 就是用户自定义变量，自定义变量的值由用户自行设定，而预定义变量和自动变量为通常在 Makefile 都会出现的变量，其中部分有默认值，也就是常见的设定值，

当然用户可以对其进行修改。

在 Makefile 中的定义的变量，就像是 C/C++ 语言中的宏一样，他代表了一个文本字符串，在 Makefile 中执行的时候其会自动原模原样地展开在所使用的地方。其与 C/C++ 所不同的是，你可以在 Makefile 中改变其值。在 Makefile 中，变量可以使用在“目标”、“依赖目标”、“命令”或是 Makefile 的其它部分中。

预定义变量包含了常见编译器、汇编器的名称及其编译选项。Makefile 中常见预定义变量及其部分默认值如表 3-12 所示。

表 3-12 Makefile 中常见预定义变量

命令格式	含 义
AR	库文件维护程序的名称，默认值为 ar
AS	汇编程序的名称，默认值为 as
CC	C 编译器的名称，默认值为 cc
CPP	C 预编译器的名称，默认值为 \$(CC) -E
CXX	C++编译器的名称，默认值为 g++
FC	FORTRAN 编译器的名称，默认值为 f77
RM	文件删除程序的名称，默认值为 rm -f
ARFLAGS	库文件维护程序的选项，无默认值
ASFLAGS	汇编程序的选项，无默认值
CFLAGS	C 编译器的选项，无默认值
CPPFLAGS	C 预编译的选项，无默认值
CXXFLAGS	C++编译器的选项，无默认值
FFLAGS	FORTRAN 编译器的选项，无默认值

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

在定义变量的值时，在 Makefile 中有两种方式来定义变量的值。

先看第一种方式，也就是简单的使用“=”号，在“=”左侧是变量，右侧是变量的值，右侧变量的值可以定义在文件的任何一处，也就是说，右侧中的变量不一定非要是已定义好的值，其也可以使用后面定义的值。我们可以使用 make 中的另一种用变量来定义变量的方法。这种方法使用的是“:=”操作符。

一般在书写 Makefile 时，各部分变量引用的格式如下：

1. make 变量（Makefile 中定义的或者是 make 的环境变量）的引用使用 “\$(VAR)” 格式，无论 “VAR” 是单字符变量名还是多字符变量名。
2. 出现在规则命令行中 shell 变量（一般为执行命令过程中的临时变量，它不属于 Makefile 变量，而是一个 shell 变量）引用使用 shell 的 “\$tmp” 格式。
3. 对出现在命令行中的 make 变量同样使用 “\$(CMDVAR)” 格式来引用。

#### 改进二：使用自动推导

```
CC = gcc
OBJ = main.o mytool1.o mytool2.o
make: $(OBJ)
$(CC) -o main $(OBJ)
main.o: mytool1.h mytool2.h
mytool1.o: mytool1.h
mytool2.o: mytool2.h
.PHONY: clean
clean:
rm -f main $(OBJ)
```

让 make 自动推导，只要 make 看到一个.o 文件，它就会自动的把对应的.c 文件加到依赖文件中，并且 gcc -c \*.c 也会被推导出来，所以 Makefile 就简化了。

#### 改进三：自动变量（\$^ \$< \$@）的应用

```
CC = gcc
OBJ = main.o mytool1.o mytool2.o
main: $(OBJ)
$(CC) -o $@ $^
main.o: main.c mytool1.h mytool2.h
$(CC) -c $<
mytool1.o: mytool1.c mytool1.h
$(CC) -c $<
mytool2.o: mytool2.c mytool2.h
$(CC) -c $<
.PHONY: clean
clean:
rm -f main $(OBJ)
```

命令中的 “\$<” 和 “\$@” 是自动化变量，“\$<” 表示所有的依赖目标集，“\$@” 表示目标集。

#### 改进四：使用函数

```
CC = gcc
CFLAGS = -Wall -c
LDFLAGS = -lpthread
```

```

SRCS = $(wildcard *.c)
OBJS = $(patsubst %c,%o,$(SRCS))
TARGET = main
.PHONY: all clean //“.PHONY”表示, all 和 clean 是个伪目标文件。
all: $(TARGET)
$(TARGET): $(OBJS)
$(CC) $(LDFLAGS) -o $@ $^
%.o: %.c
$(CC) $(CFLAGS) -o $@ $<
clean:
@rm -f *.o $(TARGET)

```

在这个 Makefile 中使用 wildcard 和 patsubst 函数来处理变量, 从而让我们的命令或是规则更为的灵活和具有智能。make 所支持的函数也不算很多, 不过已经足够我们的操作了。函数调用后, 函数的返回值可以当做变量来使用。wildcard 和 patsubst 函数的作用分别是扩展通配符和替换通配符。SRCS = \$(wildcard \*.c) 等于指定编译当前目录下所有 .c 文件。在 \$(patsubst %.c,%o,\$(SRCS)) 中, patsubst 把 \$(SRCS) 中的变量符合后缀是 .c 的全部替换成 .o。

通过上面的例子我们得到如下总结:

makefile 文件保存了编译器和连接器的参数选项, 还表述了所有源文件之间的关系(源代码文件需要的特定的包含文件, 可执行文件要求包含的目标文件模块及库等)。创建程序(make 程序)首先读取 makefile 文件, 然后再激活编译器、汇编器、资源编译器和连接器以便产生最后的输出, 最后输出并生成的通常是可执行文件。创建程序利用内置的推理规则来激活编译器, 以便通过对特定 cpp 文件的编译来产生特定的 obj 文件。

### 3.5.5 万能 MAKEFILE 模板

```

#####
##
#
# Generic Makefile for C/C++ Program
#
# Author: whylinux (whylinux AT hotmail DOT com)
# Date: 2006/03/04
#
# Description:
# The makefile searches in <SRCDIRS> directories for the source
files

```

```

# with extensions specified in <SOURCE_EXT>, then compiles the
sources
# and finally produces the <PROGRAM>, the executable file, by
linking
# the objectives.

# Usage:
# $ make compile and link the program.
# $ make objs      compile only (no linking. Rarely used).
# $ make clean     clean the objectives and dependencies.
# $ make cleanall  clean the objectives, dependencies and
executable.
# $ make rebuild   rebuild the program. The same as make clean
&& make all.
#=====

## Customizing Section: adjust the following if necessary.
##=====
=

# The executable file name.
# It must be specified.
# PROGRAM := a.out    # the executable name
PROGRAM :=

# The directories in which source files reside.
# At least one path should be specified.
# SRCDIRS := .        # current directory
SRCDIRS :=

# The source file types (headers excluded).
# At least one type should be specified.
# The valid suffixes are among
of .c, .C, .cc, .cpp, .CPP, .c++, .cp, or .cxx.
# SRCEXTS := .c      # C program
# SRCEXTS := .cpp    # C++ program
# SRCEXTS := .c .cpp # C/C++ program
SRCEXTS :=

```

```

# The flags used by the cpp (man cpp for more).
# CPPFLAGS := -Wall -Werror # show all warnings and take them
as errors
CPPFLAGS :=

# The compiling flags used only for C.
# If it is a C++ program, no need to set these flags.
# If it is a C and C++ merging program, set these flags for the
C parts.
CFLAGS :=
CFLAGS +=

# The compiling flags used only for C++.
# If it is a C program, no need to set these flags.
# If it is a C and C++ merging program, set these flags for the
C++ parts.
CXXFLAGS :=
CXXFLAGS +=

# The library and the link options ( C and C++ common).
LDFLAGS :=
LDFLAGS +=

## Implicit Section: change the following only when necessary.
##=====
=
# The C program compiler. Uncomment it to specify yours
explicitly.
#CC = gcc

# The C++ program compiler. Uncomment it to specify yours
explicitly.
#CXX = g++

# Uncomment the 2 lines to compile C programs as C++ ones.
#CC = $(CXX)
#CFLAGS = $(CXXFLAGS)

```

```

# The command used to delete file.
#RM          = rm -f

## Stable Section: usually no need to be changed. But you can
add more.
##=====
SHELL      = /bin/sh
SOURCES    = $(foreach d,$(SRCDIRS),$(wildcard $(addprefix
$(d)/*,$(SRCEXTS))))
OBJS       = $(foreach x,$(SRCEXTS), \
              $(patsubst %$(x),%.o,$(filter %$(x),$(SOURCES))))
DEPS       = $(patsubst %.o,%.d,$(OBJS))

.PHONY : all objs clean cleanall rebuild

all : $(PROGRAM)

# Rules for creating the dependency files (.d).
#-----
%.d : %.c
      @$ (CC) -MM -MD $(CFLAGS) $<

%.d : %.C
      @$ (CC) -MM -MD $(CXXFLAGS) $<

%.d : %.cc
      @$ (CC) -MM -MD $(CXXFLAGS) $<

%.d : %.cpp
      @$ (CC) -MM -MD $(CXXFLAGS) $<

%.d : %.CPP
      @$ (CC) -MM -MD $(CXXFLAGS) $<

%.d : %.c++
      @$ (CC) -MM -MD $(CXXFLAGS) $<

```



```

%.d : %.cp
    @$(CC) -MM -MD $(CXXFLAGS) $<

%.d : %.cxx
    @$(CC) -MM -MD $(CXXFLAGS) $<

# Rules for producing the objects.
#-----
objs : $(OBJS)

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

%.o : %.C
    $(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $<

%.o : %.cc
    $(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $<

%.o : %.cpp
    $(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $<

%.o : %.CPP
    $(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $<

%.o : %.c++
    $(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $<

%.o : %.cp
    $(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $<

%.o : %.cxx
    $(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $<

# Rules for producing the executable.
#-----
$(PROGRAM) : $(OBJS)
ifeq ($(strip $(SRCEXTS)), .c) # C file

```

```

        $(CC) -o $(PROGRAM) $(OBJS) $(LDFLAGS)
else
        # C++ file
        $(CXX) -o $(PROGRAM) $(OBJS) $(LDFLAGS)
endif

-include $(DEPS)

rebuild: clean all

clean :
        @$(RM) *.o *.d

cleanall: clean
        @$(RM) $(PROGRAM) $(PROGRAM).exe

###End of the Makefile#####

```

程序所在的路径在 SRCDIRS 中设定。如果源程序分布在不同的目录中，那么需要在 SRCDIRS 中一一指定，并且路径名之间用空格分隔。

在 SRCEXTS 中指定程序中使用的文件类型。C/C++程序的扩展名一般比较固定，有几种形式：`.c`、`.C`、`.cc`、`.cpp`、`.CPP`、`.c++`、`.cp`、或者`.cxx`（参见 `man gcc`）。扩展名决定了程序是 C 还是 C++ 程序：`.c` 是 C 程序，其它扩展名表示 C++ 程序。一般固定使用其中的一种扩展名即可。但是也有可能需要使用多种扩展名，这可以在 `SOURCE_EXT` 中一一指定，各个扩展名之间用空格分隔。虽然并不常用，但是 C 程序也可以被作为 C++ 程序编译。这可以通过在 Makefile 中设置 `CC=$(CXX)` 和 `CFLAGS=$(CXXFLAGS)` 两项即可实现。

这个 Makefile 支持 C、C++ 以及 C/C++ 混合三种编译方式：

如果只指定 `.c` 扩展名，那么这是一个 C 程序，用 `$(CC)` 表示的编译命令进行编译和连接。

如果指定的是除 `.c` 之外的其它扩展名（如 `.cc`、`.cpp`、`.cxx` 等），那么这是一个 C++ 程序，用 `$(CXX)` 进行编译和连接。

如果既指定了 `.c`，又指定了其它 C++ 扩展名，那么这是 C/C++ 混合程序，将用 `$(CC)` 编译其中的 C 程序，用 `$(CXX)` 编译其中的 C++ 程序，最后再用 `$(CXX)` 连接程序。

这些工作都是 make 根据在 Makefile 中提供的程序文件类型（扩展名）自动判断进行的，不需要用户干预。

#### • 指定编译选项

编译选项由三部分组成：预处理选项、编译选项以及连接选项，分别由 `CPPFLAGS`、`CFLAGS` 与 `CXXFLAGS`、`LDFLAGS` 指定。

`CPPFLAGS` 选项可参考 C 预处理命令 `cpp` 的说明，但是注意不能包含 `-M` 以及和 `-M` 有关的选项。如果是 C/C++ 混合编程，也可以在这里设置 C/C++ 的一些共同的编译选项。

CFLAGS 和 CXXFLAGS 两个变量通常用来指定编译选项。前者仅仅用于指定 C 程序的编译选项，后者仅仅用于指定 C++ 程序的编译选项。其实也可以在两个变量中指定一些预处理选项（即一些本来应该放在 CPPFLAGS 中的选项），和 CPPFLAGS 并没有明确的界限。

连接选项在 LDFLAGS 中指定。如果只使用 C/C++ 标准库，一般没有必要设置。如果使用了非标准库，应该在这里指定连接需要的选项，如库所在的路径、库名以及其它联接选项。

现在的库一般都提供了一个相应的 .pc 文件来记录使用库所需要的预编译选项、编译选项和连接选项等信息，通过 pkg-config 可以动态提取这些选项。与由用户显式指定各个选项相比，使用 pkg-config 来访问库提供的选项更方便、更具通用性。在后面可以看到一个 GTK+ 程序的例子，其编译和连接选项的指定就是用 pkg-config 实现的。

### • 编译和连接

上面的各项设置好之后保存 Makefile 文件。执行 make 命令，程序就开始编译了。

命令 make 会根据 Makefile 中设置好的路径和文件类型搜索源程序文件，然后根据文件的类型调用相应的编译命令、使用相应的编译选项对程序进行编译。

编译成功之后程序的连接会自动进行。如果没有错误的话最终会产生程序的可执行文件。

注意：在对程序编译之后，会产生和源程序文件一一对应的 .d 文件。这是表示依赖关系的文件，通过它们 make 决定在源程序文件变动之后要进行哪些更新。为每一个源程序文件建立相应的 .d 文件这也是 GNU Make 推荐的方式。

#### • Makefile 目标 (Targets)

下面是关于这个 Makefile 提供的目标以及它所完成的功能：

`make`

编译和连接程序。相当于 `make all`。

`make objs`

仅仅编译程序产生 .o 目标文件，不进行连接（一般很少单独使用）。

`make clean`

删除编译产生的目标文件和依赖文件。

`make cleanall`

删除目标文件、依赖文件以及可执行文件。

`make rebuild`

重新编译和连接程序。相当于 `make clean && make all`。

## 本章小结

本章是 Linux 中进行 C 语言编程的基础，首先讲解了 C 语言编程的关键点，这里关键要了解编辑器、编译链接器、调试器及项目管理工具等关系。

接下来，本章介绍了 Gcc 编译器的使用和 Gdb 调试器的使用，并结合了具体的实例进行讲解。虽然它们的选项比较多，但是常用的并不多，读者着重掌握笔者例子中使用的一些选项即可。

之后，本章又介绍了 Make 工程管理器的使用，这里包括 Makefile 的基本结构、Makefile 的变量定义及其规则和 Make 的使用。

本章的讲的内容较多，包括了 GCC、Gdb 和 Makefile 的使用，由于这些都是 Linux 中的常用软件，因此希望读者确实掌握。

# 第 4 章 SHELL 编程

## 学习目标

- 认识 shell
- 熟悉 shell 常用命令
- 掌握 shell 编程

## 4.1 为什么要学 shell

作为 Linux 操作系统的来源，UNIX 是许多伟大构想和操作系统哲学的源泉，它提供了多种不同的 shell 程序。在 UNIX 商业版本中最常见的可能是 Korn shell，但还没有许多其它的 shell。那为什么要使用 shell 进行程序设计呢？这么说吧，shell 扮演了一个双重角色。虽然它表面上和 Windows 的命令提示符相似，但是它具备完成相当复杂的程序的强大功能。你不仅可以通过它执行命令、调用 Linux 工具，还可以自己编写程序。Shell 使用解释开型语言，这使得调试工作比较容易进行，因为你可以逐行地执行指令，而且节省了重新编译的时间。然而，这也使得 shell 不适合用来完成时间紧迫和处理器忙碌的任务。

使用 shell 进行程序设计的原因之一是，你可以快速、简单地完成编程。同时，即使是在最基本的 linux 安装中也会提供一个 shell。因此，如果你有一简单的构想，则可以通过它检查自己的想法是否可行。Shell 也非常适合于编写一些执行相对简单的任务的小工具，因为它们更强调的是易于配置、维护和可移植性，而不是很看重执行的效率。你还可以使用 shell 对进程进行组织，使命令按照预定顺序在前一阶段命令成功完成的前提下顺序执行。

为什么要学习文字模式的 shell

我们常常提到的 shell 其实是比较狭隘的定义，一般来说，在 linux 里，所谓的 shell，就是指 BASH 这个文字模式的 shell。但是，广义的 shell 也可以是 KDE 之类的图形界面控制软件，因为它也可以帮我们与核心进行沟通。不过，在笔者的 linux 私房菜里，如果没有特别说明，我们的 shell 指的是比较狭义的，也就是文字模式的 shell。

另外，常常听到这样子的一个问题：“我为什么要学习 shell？不是已经有很多的工具可以设置主机了吗？我为什么要花这么多时间去学命令？不是以 X Window 按一按几个按钮就可以搞掂了吗？为什么要这么麻烦？”。的确，X Window 以及 Web 接口的设置工具，例如 webmin，它真的可能帮助我们很轻松地设置好主机，甚至是一些很高级的高级。

但是，X Window 的界面虽然友好，功能也强大，而 Web 接口的工具也可以提供友好的服务，但毕竟它是将所有的套件都整合在一起的一个套件而已，并非是一个完整的套件。所以，有时候，比如升级或者使用其他套件管理模块时，设置上就会有问题。

当远程联机时，命令行界面的传输速度一定比较愉，而且，不容易出现相断线或者是信息外流的问题，因此，shell 值得一学。而且，它可以让你更深入 linux，更解它，而不是只会按一按鼠标而已。多学一点文字模式的东西，会让你与 linux 更新近。

有些人也很可爱，常会说：“我学这么多干什么？又不常用，也用不到。”有没有听过“书到用时方恨少？”当主机一切安然无样的时候，你当然会觉得好像学这么多的东西一点帮助也没有，万一某天不幸出了问题，怎么办？是直接重新安装，还是先追踪入侵来源后进行漏洞的修补？或者是干脆就关站好了？这当然涉及很多因素的考虑。

此外，如果真有心想要将主机管理好，那么，必需要掌握 shell 程序的编写。比如机房管理来说，只有 10 台，这不算多，但如果每台主机者要花上几十分钟来查看其文件及相关信息，工作量就太大了，也太没有效率了。这个时候，如果能够通过 shell 提供的命令重导向以及管道命令，不用 10 分钟就能看完所有主机的重要信息了。

## 4.2 认识 shell

### 4.2.1 什么是 shell

这应该是一个很有意思的话题：“什么是 shell”相信只要用过计算机，对操作系统（无论是 linux、UNIX，还是 Windows）有点概念的人，大多听过这个名词，因为只要有“操作系统”，那么就离不开 shell。不过，在讨论 shell 之前，我们先来了解一下计算机的运行状态。举个例子来说：要计算机输出“音乐”的时候，你的计算机需要什么？

- (1) 需要有“声卡芯片”硬件设备，否则怎么会有声音。
- (2) 操作系统的核心可以支持这个芯片集，当然还需要提供芯片的驱动程序。
- (3) 需要用户（就是你）输入发生声音的命令。

这就是基本的输出声音的步骤。也就是说，你必须“输入”一个命令之后，“硬件”才会通过你下达的命令来工作。那么，硬件如何知道你下达的命令呢？那就是核心在起作用了。也就是说，我们必须要通过“shell”将输入的命令与核心沟通，让核心可以控制硬件来正确无误地工作。我们可以通过图 4-1 来说明。

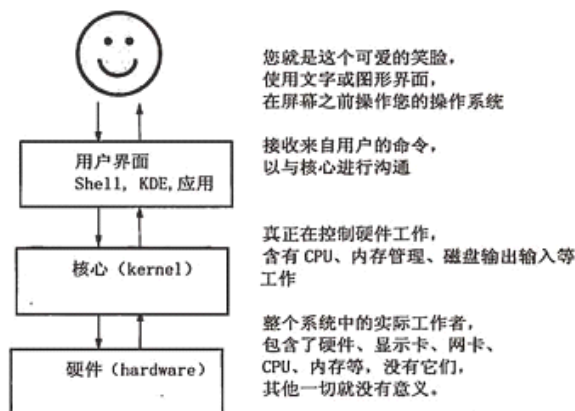


图 4 1 硬件、核心、用户之间的联系

通常，替我们工作的是“硬件”，而控制硬件的是“核心”。用户是利用“shell”控制一些内核提供的工具，来控制硬件替我们正确工作。进一步来说，由于内核听不懂人类语言，而人类也没有办法直接记住内核的语言，所以两者的沟通就要通过 shell 来支持了。

以字面的意思来说，kernel 是“核心”的意思，而 shell 是“壳”的意思。也就是说，shell 在最外头，核心是最内层的。核心是操作系统最底层的东西。这个核心里包括了各种支持硬件的工具。当然，如果硬件太新，不被内核所支持的话，那么，shell 的能力再强，也没有办法使硬件工作。使计算机主机工作的正是核心的任务，但操作核心来替用户工作的，却是 shell。因此，有时候 shell 做了半天，硬件却不能工作的时候，请注意“核心”是否正确。

在开始讨论如何使用 shell 进行程序设计之前，我们先来回顾一下 shell 的作用以及 linux 系统中提供的各种 shell。

Shell 是一个作为用户与 linux 系统间接口的程序，它允许用户向操作系统输入需要执行的命令。这点与 windows 的命令提示符类似。

Linux 系统由以下 3 部分组成：内核、shell、应用程序

内核：操作者不易和它直接沟通，因此，必须要有一个友善的界面使得操作时能更为方便，这个界面便是 shell 通俗地讲，shell 就是位于内核与操作者之间的一层使用者界面。

Shell：本意是壳的意思。在内核的外面包着一层外壳，用来负责接收使用者输入的指令，然后将指令解释成内核能够了解的方式，传给内核去执行，再将结果传回至预设的输出周边。

在 AT&T 的 Dennis Ritchie 和 Ken Thompson 设计 UNIX 的时候，他们想要为用户创建一种与他们的新系统交流的方法。那时的操作系统带有命令解释器，命令解释器接受用户的命令，然后解析它们，因而计算机可以使用这些命令。

但是 Ritchie 和 Thompson 想要的不只是这些功能,他们想提供比当时的命令解析器更优异的工具。这导致了 bourne shell (简称 sh) 的开发,由 S.R.bourne 创建。自从 Bourne shell 被创建,其他 shell 也被一一开发,如 C shell (csh) 和 Korn shell (ksh)。

## 4.2.2 Bash Shell

Bash shell 是 GNU 计划中最重要的工具软件之一,当前也是 GNU 操作系统中标准的 shell,主要兼容于 sh,并且是根据一些客户的需求而增强的 shell 版本,当前几乎所有 linux 版本都使用 bash 作为系统管理的核心 shell。相比其它 shell, bash 具有更加强大的功能。

### 1. 命令记忆功能

通过翻按键盘上的『上下键』可以查看到之前使用过的指令。在默认情况下记录的个数可达到 1000 个,可以在家目录内的 bash\_history 中修改。每次登录后执行的指令都被暂存在缓冲区中,成功退出系统后,该指令记忆便会记录到 bash\_history 当中。通过这一功能,你就可以很方便地修正错误的执行命令,但同时也为骇客入侵提供了方便,他可以通过翻阅你输入的与系统相关的指令来破解你的 linux 主机。所以记录的指令数要根据个人的情况来设置。

### 2. 命令与文件补全功能

使用此功能,可以少打很多字,并且确定输入的数据是正确的。[Tab]接在一串命令的第一个字的后面,为“命令补全”;[Tab]接在一串命令的第二个字的后面,则为“文件补全”。通过这一功能,可以快速查看或匹配当前目录下相关命令或文件。

### 3. 命令别名设置功能

Linux 系统中包含有千差万别的命令名及参数,这既不方便使用也不好管理。Bash shell 中提供了利用 alias 自定义命令别名的功能。

### 4. 作业控制、前后台控制

使用前、背景的控制可以让工作进行的更为顺利!至于工作控制(jobs)的用途则更广,可以让我们随时将工作丢到背景中执行!而不怕不小心使用了 [Ctrl] + C 来停掉该程序!

### 5. Shell 编程功能

Shell 不仅可以作为命令解释器,用来定制工作环境,还可以作为一门高级编程语言,编写执行用户指令的脚本,从而更加快速有效地处理复杂的任务。

基于 bash shell 上面强大的功能,因此本章余下章节将全面围绕 bash shell 来展开,后面如无特殊说明,shell 均代表 bash shell。



### 4.2.3 简单 shell 应用

shell 作为命令解析器，管理维护系统，互动式地解释和执行用户输入的命令是 shell 的基本功能之一，shell 作为一门编程语言，处理存放在所谓 shell 脚本文件中的命令，Shell 除了作为命令编译器用于管理命令外，还可以用来进行程序设计，它提供了定义变量和参数的手段以及丰富的过程控制结构。使用 shell 编程类似于使用 DOS 中的批处理文件，称为 shell 脚本，又叫 shell 程序或 shell 命令文件。

例如在命令提示符后面新建一个目录、显示并删除这个目录

```
[root@localhost shell]# mkdir me
[root@localhost shell]# ls
me
[root@localhost shell]# rm -rf me
[root@localhost shell]# ls
```

#### 1. 基本语法

##### (1) 开头

程序必须下面的行开始（必须放在文件的第一行）：

```
#!/bin/bash
```

符号“#!”用来告诉系统它后面的参数是用来执行该文件的程序，在这个例子中使用/bin/bash 来执行程序。当编辑好脚本时，如果要执行该脚本，还必须使其可执行。

要使脚本可执行，需赋予该文件可执行的权限，使用如下命令文件才能运行。

```
#chmod 777 [文件]
```

##### (2) 注释

在进行 shell 编程时，以“#”开头的句子表示注释，直到这一行的结束，建议在程序中使用注释。如果使用注释，那么即使相当长的时间内没有使用该脚本，也能在很短的时间内明白该脚本的作用及工作原理。

#### 2. 创建过程

##### (1) 创建文件

建立一个内容如下的文件，文件名为 date，将其放在 /root 目录。

```
#!/bin/bash
#program date
#show the date in this way.

echo "Mr.$USER,Today is:"
#echo $(date)
echo `date`          # 注意符号不是单引号
echo Wish you a lucky day !
```

## (2) 设置可执行权限

编辑完该文件之后它还不能执行，需要给它设置可执行权限，使用如下命令。

```
Chmod 777 date
```

## (3) 执行程序

写上整个文件的完整路径，执行 shell 程序，使用如下命令可看到执行结果。

```
[root@localhost ~]# ./date
Mr.root,Today is:
三 6 月 30 11:24:47 CST 2010
Wish you a lucky day !
```

## 4.2.4 Shell 常用命令

当用户登录到 linux 系统时，便开始与 bash 进行互动，一直到用户注销为止。如果是普通用户，则 Bash 的默认提示符为“\$”（代表一般身份使用者），如果是超级用户（root），提示符则变为“#”。用户与系统互动的过程便是通过在提示符后面输入操作命令来完成的。按照操作命令的来源，我们可以将其分为以下两大类：

### 1. bash 内置命令

为了方便 shell 的操作，加快用户与系统互动的效率，bash 中“内置”了很大一部分常用操作命令，如 cd、umask 等等

### 2. 外部应用命令

为了加强 shell 的处理能力，除本身内置一部分命令外，还增加了对外部应用命令支持，如 ls、ps 等等

在 shell 的命令提示符后面输入命令，如果是 bash shell 内置的命令，则由其自己负责回应。若是外部应用命令，则 shell 会找出其对应的外部应用程序，然后将控制权交给内核，由内核执行该应用程序之后再将控制权交回给 shell。

怎样知道这个命令是来自于外部命令还是内置在 bash 中的？这个可以利用 type 命令来观察。下面将会简单介绍 type 及几个常用命令：

#### (1) type

命令格式：type 参数 命令

功能：判断一个命令是内置命令还是外部命令

参数分析：

参 数	作 用
没有	显示出命令是外部命令还是 bash 内置命令
-t	File : 表示为外部命令 Alias : 表示该命令为命令别名所设置的名称。 Builtin: 表示该命令为 bash 内置的命令功能。

-p	显示完整文件名（外部命令）或显示内置命令
-a	在 PATH 变量定义的路径中，列出所有含有 name 的命令，包含 alias

实验 1：查询 ls 这个命令是否为 bash 内置命令？

```
[root@localhost ~]# type ls
ls is aliased to `ls --color=tty'
```

没有任何参数，仅列出 ls 命令的最主要使用情况

```
[root@localhost ~]# type -t ls
Alias
```

-t 参数仅列出 ls 命令的最主要使用情况

```
[root@localhost ~]# type -a ls
ls is aliased to `ls --color=tty'
ls is /bin/ls
```

-a 利用所有方法找出来的 ls 相关信息都会列举出来。

实验 2：那么，cd 呢？

```
[root@localhost ~]# type cd
cd is a shell builtin
```

实验解析：通过 type 命令的用法，可以知道，每个命令是否为 bash 的内置命令。

此外，由于使用 type 搜索后面的名称时，如果后接的名称并不能以执行文件的状态找到，那么，该名称是不会显示出来。

（2）echo

命令格式：Echo arg

功能：在屏幕上显示出由 arg 指定的字符串。

参数分析：

参数	功能

实验：

```
[root@localhost test]# vi sh_01.sh
#!/bin/bash
echo "hello world!"
[root@localhost test]# ./sh_01.sh
hello world!
```

（3）export

命令格式 1: Export variable

功能: shell 可以用 export 把它的变量向下带入子 shell, 从而让子进程继承父进程中的环境变量。但子 shell 不能用 export 把它的变量向上带入父进程。

实验: (shell 变量章节详细分析)

命令格式 2: Export

功能: 显示出当前所有环境变量及其内容。

实验:

```
[root@localhost ~]# export
declare -x COLORTERM="gnome-terminal"
declare -x DBUS_SESSION_BUS_ADDRESS="unix:abstract=
/tmp/dbus-4ELcBCtxnL"
declare -x DESKTOP_SESSION="default"
declare -x DISPLAY=":0.0"
declare -x GDMSESSION="default"
.....
declare -x USER="root"
declare -x WINDOWID="50331729"
declare -x XAUTHORITY="/root/.Xauthority"
declare -x XMODIFIERS="@im=htt"
```

(4) readonly

命令格式 1: readonly variable

功能: 将一个用户自定义的 shell 变量标识为不可变。

命令格式 2: readonly

功能: 显示出所有只读的 shell 变量。

实验:

```
[root@localhost ~]# readonly
declare -ar BASH_VERSINFO='([0]="3" [1]="00" [2]="15" [3]="1"
[4]="release" [5]="i386-redhat-linux-gnu")'
declare -ir EUID="0"
declare -ir PPID="16000"
declare -r SHELLOPTS=
"braceexpand:emacs:hashall:histexpand:history:interactive-co
mments:monitor"
declare -ir UID="0"
```

(5) read

命令格式: Read variable

功能: 从标准输入设备读入一行, 分解成若干行, 赋值给 shell 程序定义的变量。

实验:

```
[root@localhost test]# vi sh_04.sh
```

```
#!/bin/bash
echo -e "Please enter: \c"
read x
echo "you enter: $x"
[root@localhost test]# ./sh_04.sh
Please enter: hello
you enter: hello
```

实验解析:

(6) env

命令格式: Env

功能: 显示环境变量及其内容。

实验:

```
[root@localhost test]# env
SSH_AGENT_PID=5223
HOSTNAME=localhost.localdomain
SHELL=/bin/bash
TERM=xterm
HISTSIZE=1000
GTK_RC_FILES=/etc/gtk/gtkrc:/root/.gtkrc-1.2-gnome2
WINDOWID=50331729
.....
G_BROKEN_FILENAMES=1
COLORTERM=gnome-terminal
XAUTHORITY=/root/.Xauthority
_=/bin/env
OLDPWD=/root/shell
```

(7) 7、set

命令格式: Set

功能: 显示所有变量及其内容。

实验:

```
[root@localhost test]# set
BASH=/bin/bash
BASH_ARGC=()
BASH_ARGV=()
BASH_LINENO=()
BASH_SOURCE=()
.....
UID=0
```

```
USER=root
WINDOWID=50331729
XAUTHORITY=/root/.Xauthority
XMODIFIERS=@im=htt
_--color=tty
```

(8) unset

命令格式:

功能: unset 命令的作用是从环境中删除变量或函数。这个命令不能删除 shell 本身定义的只读变量 (如 IFS)。这个命令并不常用。

实验:

```
#!/bin/bash
    foo="Hello World"
    echo $foo

    unset foo
    echo $foo
```

实验解析: 第一次输出字符串 “Hello World”, 但第二次只输出一个换行符

(9) trap

命令格式: Trap command signal

功能: 在脚本程序发生相应中断时执行相应命令内容

实验:

```
#!/bin/bash
    trap 'rm -f /tmp/my_tmpfile_$$' INT
    echo creating file /tmp/my_tmp_file_$$
    date > /tmp/my_tmp_file_$$

    echo "press interrupt (CTRL-C) to interrupt....."
    while [ -f /tmp/my_tmp_file_$$ ];do
        echo File exists
        sleep 1
    done
    echo The file no longer exists

    trap INT
    echo creating file /tmp/my_tmp_file_$$
    date > /tmp/my_tmp_file_$$

    echo "press interrupt (CTRL-C) to interrupt....."
```

```

while [ -f /tmp/my_tmp_file_$$ ];do
    echo File exists
    sleep 1
done

echo we never get here
exit 0

```

#### 实验解析

在这个脚本程序中，我们先用 trap 命令安排它在出现一个 INT（中断）信号时执行 `rm -f /tmp/my_tmp_file_$$` 命令删除临时文件。脚本程序然后进入一个 while 循环，只要临时文件存在，循环就一直持续下去。当用户按下 Ctrl+C 组合键时，就会执行 `rm -f /tmp/my_tmp_file_$$` 语句，然后继续下一个循环。因为临时文件现在已经被删除了，所以第一个 while 循环将正常退出。

接下来，脚本程序再次调用 trap 命令，这次是指定 当一个 INT 信号出现时不执行任何命令。脚本程序然后重新创建临时文件并进入第二个 while 循环。这次当用户按下 Ctrl+C 组合键时，没有语句被指定 执行，所以采取默认处理方式，即立即终止脚本程序。因为脚本程序被立即终止了，所以永远也不会执行最后的 echo 和 exit 语句。

#### (10) grep

命令格式 `grep 参数 string 目标文件`

功能：在指定文件一堆文件中查找一个特定的字串，并将字串所在行输出到终端或平台。

参数分析：

参数	功能

实验：

编辑查找文件

```

[root@localhost test]# vi test_file
CD0001 ,jakey ,hello world
CD0002 ,peter ,good morning
CD0003 ,kety ,how are you
CD0004 ,tony ,see you later

```

编辑实验脚本

```

[root@localhost test]# vi sh.sh
#!/bin/bash
grep -v "^CD0002" test_file

```

运行脚本

```
[root@localhost test]# ./sh.sh
CD0001 ,jakey ,hello world
CD0003 ,kety ,how are you
CD0004 ,tony ,see you later
```

实验解析：-v 反义选项，使 grep 选择所有和模式不匹配的行。所以输出文件中除了 CD0001 所在行之外的所在信息。

^ 匹配行开始标记

(11) wc

命令格式：wc 参数 文件 1 文件 2 …….

功能：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。

参数分析：

参数	功能
-c	统计文件字节数
-l	统计文件行数
-w	统计文件字数
备注：几个参数可同时使用      输出格式：行数、字数、字节数、文件名	

实验：

```
[root@localhost test]# wc -lcw test_file
4 18 111 test_file
```

实验解析：

## 4.2.5 重定向与管道

Shell 命令在执行时，会自动打开三个标准文件，即标准输入文件（stdin），一般对应终端的键盘；标准输出文件（stdout）和标准出错输出文件（stderr），这两个文件对应终端的屏幕。但在实际应用中，这三个文件常常需要按照新的格式进行定向，从其它文件中导入内容或将内容导出到其它文件中，此过程就是重定向；使内容按一定格式输出，这就是管道。

### 1. 重定向

一个命令的执行可以用图 4-2 表示。



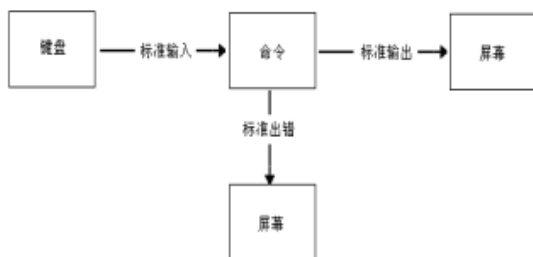
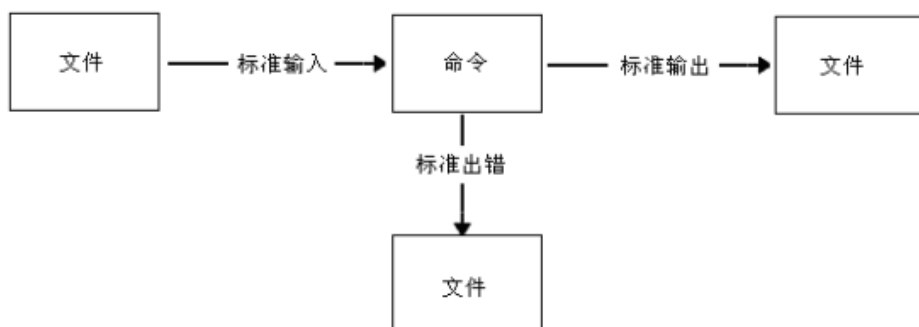


图 4 2

执行命令时，这个命令会通过键盘读入数据，经过处理后，再将数据输出到屏幕上。数据流重定向就是命令执行后，从某文件中读入数据，经过处理后，再将数据输出到另一文件中。其执行过程如下图所示：



重定向可分为输出重定向、错误重定向与输入重定向。下面将举例分析：

### (1) 输出重定向

通过重定向符“>”或“>>”将命令的标准输出重新定向到指定文件中。

一般形式：命令 > 文件名

命令 >> 文件名

“>”与“>>”都能将内容重新写入到文件中，但如果文件中有内容，执行“>”完后新的内容将会覆盖掉原来的内容，而“>>”则是将新的输出内容附加到原来内容的结尾。

实验 1：

```

[root@localhost shell]# ls
[root@localhost shell]# ps > test
[root@localhost shell]# ls
test
[root@localhost shell]# cat test
  
```

PID	TTY	TIME	CMD
6459	pts/2	00:00:00	bash
6479	pts/2	00:00:00	bash
6774	pts/2	00:00:00	ps

实验解析:

将 ps 命令的内容重定向到 test 文件中, 执行完命令后, 终端上确实没有输出内容, 而当前目录下却多了 test 文件, 再利用 cat 可以看到内容的确写到了里面去了。

实验 2:

```
[root@localhost shell]# ls -a >> test
[root@localhost shell]# cat test
PID TTY          TIME CMD
6459 pts/2        00:00:00 bash
6479 pts/2        00:00:00 bash
6774 pts/2        00:00:00 ps
.
..
test
```

实验解析:

用“>>”将当前目录下的所有文件名写入到实验 1 中创建的 test 文件中, 执行完命令后用 cat 命令查看, 新的内容的确是添加到文件原有内容的尾部。

## (2) 错误重定向

通过重定向符“2>”或“2>>”将命令的标准错误输出重新定向到指定文件中。

一般形式: 命令 2> 文件名

命令 2>> 文件名

实验 3:

```
[root@localhost shell]# cat ./jingzhao
cat: ./jingzhao: 没有那个文件或目录
[root@localhost shell]# cat ./jingzhao > test
cat: ./jingzhao: 没有那个文件或目录
[root@localhost shell]# cat ./jingzhao 2> test
[root@localhost shell]# ls
test
[root@localhost shell]# cat test
cat: ./jingzhao: 没有那个文件或目录
```

实验解析:

由于当前目录中不存在文件 jingzhao, 用 cat 命令输出其中内容时会在终端上打印出错信息。改用输出重定向符号时错误信息还是没能重定向到 test 中, 最后利用错误重

定向符号将错误提示输出到 test 文件中。

从上面也可以知道，采用 “>” 或 “>>” 是不能将错误的信息重定向的。

### (3) 输入重定向

通过重定向符 “<” 将命令的标准输入重新定位到指定文件中。

一般形式：命令 < 文件名

实验：

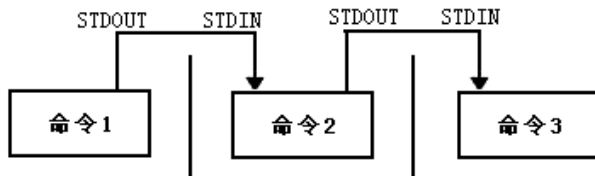
```
[root@localhost shell]# ls
sh.sh
[root@localhost shell]# cat sh.sh
echo "your working directory is `pwd` "
echo "the time is `date`"
[root@localhost shell]# bash < sh.sh
your working directory is /shell
the time is 四  月 29 15:16:23 CST 2010
```

实验解析：

Shell 命令解析程序将从脚本程序 sh.sh 中读取命令行，并加以执行。

## 2. 管道

Linux 下可以采用管道操作符 “|” 来连接多个命令或进程，如下图示：



在连接的管道线两边，每个命令执行时都是一个独立的进程。前一个命令的输出正是下一个命令的输入。这些进程可以同时运行，并且随着数据流在它们之间的传递可以自动地进行协调，从而能够完成较为复杂的任务。

一般形式：[命令 1] | [命令 2] | [命令 3]

实验：显示/etc 目录下面内容

```
[root@localhost shell]# ls /etc/
.....
esd.conf          pwdb.conf
exports           quotagrpadmins
fb.modes          quotatab
fdprm             racoon
.....
```

因为/etc下面的文件太多了，当利用ls /etc/来查看时发现整个屏幕都被塞满了文件，非常不方便，可利用more来分页显示。

```
[root@localhost bin]# ls /etc |more
a2ps.cfg
a2ps-site.cfg
acpi
adjtime
alchemist
aliases
aliases.db
alsa
.....
--More--
```

命令ls /etc显示/etc目录的内容，命令more是分页显示内容。

## 4.3 shell 编程

(这里要增加一些说明性文字)

### 4.3.1 shell 变量

与各种高级程序设计语言相似，shell环境下也可以一组文字或符号，来替换一些设置或者是一串保留的数据，这组文字或符号便是shell变量。

在linux环境中，存在大量不同的shell变量，既有环境本身自带的系统变量，也有用户自定义的普通变量。根据前面常用命令，我们可以用set命令查看linux系统当前所有变量及其内容。

```
[root@localhost test]# set
BASH=/bin/bash
BASH_ARGC=()
BASH_ARGV=()
BASH_LINENO=()
BASH_SOURCE=()
BASH_VERSION='3.00.15(1)-release'
COLORS=/etc/DIR_COLORS.xterm
COLORTERM=gnome-terminal
.....
```

```
TERM=xterm
UID=0
USER=root
WINDOWID=50342033
XAUTHORITY=/root/.Xauthority
XMODIFIERS=@im=htt
_--color=tty
```

下面作者根据使用功能不同将 shell 变量中常用的用户自定义变量、位置参数与环境变量三大块作详细分析。

## 1. 自定义变量

定义：变量名=变量值

在使用变量之前不需要事先声明，只需要通过“=”给它们赋初始值便可使用。但等号两边不能留空格，若一定要出现空格，就要用双引号括起来。

如：

```
[root@localhost shell]# myname=jingzhao
```

此时系统便定义了 myname 这个内容为 jingzhao 的变量

查看变量内容：在变量名前面加上一个\$符号，再用 echo 命令将其内容输出到终端上。如：

```
[root@localhost shell]# echo $myname
Jingzhao
```

此时可以看到前面定义好的变量 myname 的内容 jingzhao，同样方法可以查看系统已有的变量。如：

```
[root@localhost shell]# echo $HOME
/root
```

取消变量：利用“unset 变量”可以取消系统中已有变量。

如：

```
[root@localhost shell]# echo $myname
jingzhao
[root@localhost shell]# unset myname
[root@localhost shell]# echo $myname

[root@localhost shell]#
```

创建全局变量：在此之前所创建的变量均为当前 shell 下的局部变量，不能被其它 shell 利用。因此可以使用“export 变量名”将局部变量转化为全局变量，也可以直接利用“export 变量名=变量值”来创建一个全局变量。

如：

```
[root@localhost shell]# myname=jingzhao
```

```
[root@localhost shell]# echo $myname
jingzhao
[root@localhost shell]# bash
[root@localhost shell]# echo $myname

[root@localhost shell]# exit
exit
[root@localhost shell]# export myname
[root@localhost shell]# bash
[root@localhost shell]# echo $myname
jingzhao
[root@localhost shell]# ps
PID TTY          TIME CMD
12528 pts/1      00:00:00 bash
16815 pts/1      00:00:00 bash
16831 pts/1      00:00:00 ps
[root@localhost shell]#
```

## 2. 位置参数变量

在 linux/UNIX 系统中，shell 脚本执行时是可以带实参的。这些实参在脚本执行期间将会被赋予系统中自动定义好的一类变量中，这类变量就是位置参数变量。

命令行实参与脚本中位置参数变量的对应关系如下所示：

Exam	m1	m2	m3	m4							
\$0	\$1	\$2	\$3	\$4	\$5	\$6	\$7	\$8	\$9	{10}	{11}

\$ 0 : 获取（包含）脚本名称

\$1 - \$9 : 获取（包含）第 1 至第 9 个参数

{ } : 获取第 9 个以上参数

# : 表示传给脚本或者函数的位置参数的个数（不包括”\$0”）

\* : 所有位置参数的列表，形式是一个单个字符串，串中第 1 个参数由第 1 个字符串分隔。

@ : 所有位置参数被分别表示为双引号中的 N（参数个数 --- 不含\$0）个字符串。

例如：[root@localhost ~]# echo one two three

分析：有 4 个位置参数 ---→ 1 个命令名(echo) + 3 个参量(one、two 和 three )

\$0 = echo \$1 = one \$2 = two \$3 = three

从上面对位置参数的解析中，我们可以发现：

（1）如果执行 shell 脚本时没有传递任何实参，\$0（脚本名称）、#（为 0）等也存在，而\$1、\$2 等则不存在。等脚本执行完毕，位置参数又恢复回系统初始值。

实验:

```
[root@localhost test]# vi sh.sh
#!/bin/bash
echo "first_parameter= $1"
echo "second_parameter=$2"
echo "third_parameter=$3"
echo "count_num=$#"
echo "all_parameter*=$*"
echo "all_parameter@=$@"
```

运行脚本执行如下:

```
[root@localhost test]# ./sh.sh yang jing zhao
first_parameter= yang
second_parameter=jing
third_parameter=zhao
count_num=3
all_parameter*=yang jing zhao
all_parameter@=yang jing zhao
```

(2) “\$\*” 和 “\$@” 均可以表示所有位置参数，但它们之间却存在着很大的不同，这种不同允许你用两种方法来处理命令行参数。第一种 “\$\*”，因为它是一个单个字符，所以可以不需要很多 shell 代码来显示它，相比之下更加灵活。第二种 “\$@”，它允许你独立处理每个参数，因为它的值是 N 个分离参数。

实验:

```
[root@localhost test]# vi bash.sh
#!/bin/bash
function cntparm
{
    echo -e "inside cntparm: $# parms: $@\n"
}

echo -e "outside cntparm: $*\n"
echo -e "outside cntparm: $@\n"
cntparm "$*"
cntparm "$@"
```

运行脚本执行如下:

```
[root@localhost test]# ./bash.sh yang jing zhao
outside cntparm: yang jing zhao
```

```
outside cntparm: yang jing zhao

inside cntparm: 1 parms: yang jing zhao

inside cntparm: 3 parms: yang jing zhao
```

(3) 在执行 shell 程序时, 位置参数变量并不是固定不变的, 利用 set 命令可以为位置参数赋值或重新赋值。其格式为:

```
Set paramet1 paramet2 paramet3
```

实验:

```
[root@localhost test]# vi sh.sh
#!/bin/bash
echo "first_parameter= $1"
echo "second_parameter=$2"
echo "third_parameter=$3"

set one two three
echo "parameters chance:"

echo "first_parameter= $1"
echo "second_parameter=$2"
echo "third_parameter=$3"
```

运行脚本执行如下:

```
[root@localhost test]# ./sh.sh yang jing zhao
first_parameter= yang
second_parameter=jing
third_parameter=zhao
parameters chance:
first_parameter= one
second_parameter=two
third_parameter=three
```

程序分析: (结合来整理)

函数第一次使用 “\$\*”, 它把位置参数作为单个字符串, 所以 cntparm 打印只有一个参数。但是第二次调用 cntparm, 把脚本的命令行参数当作了三个字符串, 所以 cntparm 报告了三个参数。在打印的时候, 参数的外观没有任何区别, 最后两行输出清楚地表明了这一点。

Echo 命令的 -e 选项强行要求它把字符串序列 \n 当作一个新行。如果没有 -e 选项, 就不要用 \n 来产生一个新行, 因为 echo 会自动地在输出后面加上一个 \n。



### 3. 环境变量

在用户注册过程（会话建立过程）中系统需要做的一件事就是建立用户环境。所有的 linux 进程都有各自独立，并且不同于程序本身的环境。Linux 环境由许多变量及这些变量的值组成。这些变量及变量的值决定了用户环境的外观。

Shell 环境包括使用的 shell 类型、主目录所在位置及正在使用的终端类型等多方面的内容。决定这些内容的变量有许多是在注册过程中定义的，一些为只读，意味着不能改变这些变量；而一般为非只读变量，可以随意增加或修改。（要做修改哦）

#### （1）常用环境变量

当一个 shell 开始执行时，一些变量会根据环境设置中的值进行初始化。我们的 shell 脚本程序就是根据这些初始化的环境变量来解析运行。

在 bash、sh 及 ksh 中，可以利用 env 或 export 命令查看系统中环境变量如下：

```
[root@localhost test]# env
SSH_AGENT_PID=5025
HOSTNAME=localhost.localdomain
SHELL=/bin/bash
TERM=xterm
HISTSIZE=1000
GTK_RC_FILES=/etc/gtk/gtkrc:/root/.gtkrc-1.2-gnome2
WINDOWID=50342033
QTDIR=/usr/lib/qt-3.3
USER=root
.....
LESSOPEN=|/usr/bin/lesspipe.sh %s
DISPLAY=:0.0
GTK_IM_MODULE=iiim
G_BROKEN_FILENAMES=1
COLORTERM=gnome-terminal
XAUTHORITY=/root/.Xauthority
_=/bin/env
OLDPWD=/root/shell/test
```

#### （2）环境文件（鼠标右击打开终端是要重新去读取一次环境文件）

上面用 env 所查看到的均为环境变量及其内容，这些变量都是通过系统中一系列脚本来配置完成的。从系统起来到用户注册进入系统，shell 会读取一系列称为脚本的环境文件，并执行其中命令。常见的环境文件包括：

系统级：

/etc/profile：初始化一些基本的环境变量，并执行/etc/profile.d/目录下的脚本引导系统进一步启动，在系统启动会话时被调用

/etc/bashrc: 保存一些变量设置并执行/etc/profile.d/目录下脚本的命令, 每次打开终端时会被.bashrc 调用

/etc/profile.d: 包含了配置特别程序的跨系统行为的文件

/etc/inputrc: 配置命令行响铃风格的跨系统的行读取初始化文件

用户级: (每个用户均有独立的配置文件, 在此以超级用户 root 为例)

.bash\_history: 记录用户以前输入的命令

.bash\_logout: 用户退出注册时执行的命令

.bash\_profile: 功能同上面系统级一样, 只是属于具体同户

.bashrc: 功能同上面系统级一样, 只是属于具体同户

从系统启动到用户登录 bash shell, 上面的环境文件是遵照一定的先后顺序来读取的。

首先读取/etc/profile, 根据/etc/profile 中的内容去执行相关命令或读取其它附加的环境文件, 如/etc/profile.d、/etc/inputrc 等, 根据其中内容搭建起基本的系统环境。根据不同用户, 到家目录去读取.bash\_profile, 根据内容执行相关命令并读取.bashrc 以搭建起满足不同用户的环境。

下面将就主要环境变量的使用举例说明。

PATH: 搜索路径环境变量。

定义: Shell 从中查找命令的目录列表. 它是一个非常重要的 shell 变量。PATH 变量包含带冒号分界符的字符串, 这些字符串指向含有你所使用命令。(可以补充常用的命令目录)

查看 PATH 变量内容

```
[root@localhost ~]# echo $PATH
/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/usr/X11R6/bin:/root/bin
```

## 2) 修改 PATH 变量

因 PATH 是命令搜索路径的环境变量, 所以修改变量时不能替换变量, 只能采用添加的方式, 如主目录下有一个 jingzhao 目录, 存放你编写的所有可执行命令, 要把这个目录加到 PATH 变量中, 可以输入以下命令行:

```
[root@localhost ~]# echo $PATH
/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/usr/X11R6/bin:/root/bin
[root@localhost ~]# PATH=$HOME/jingzhao
[root@localhost ~]# echo $PATH
/root/directory
[root@localhost ~]# ls
bash: ls: command not found
```

修改后，除了 bash shell 内置命令外，其它外部命令再也无法使用，主要是外部命令的

搜索路径被新的变量替代。所以，应该如下修改：

```
[root@localhost ~]# echo $PATH
/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/usr/X11R6/bin:/root/bin
[root@localhost ~]# PATH=$PATH:$HOME/jingzhao
[root@localhost ~]# echo $PATH
/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/usr/X11R6/bin:/root/bin:/root/directory
[root@localhost shell]# ls
achievements_file  applicate.sh  students_file  test
```

至此，只是为当前终端设置了新的“\$PATH”变量，如果打开一个新的终端，运行“echo \$PATH”，显示的还是没修改前的“\$PATH”值。因为先前重新定义的是一个局部环境变量，只能作用在当前终端。

要将其定义为一个全局变量，使其在以后打开的终端中生效，可以采用 export 来定义并将其写入配置文件实现。/etc/profile 与 .bash\_profile 只在会话时被读取一次，因此常用来设置不常变化的环境变量，如 PATH 等。而 .bashrc 与 /etc/bashrc 则在每次打开终端时都要被读取一次，故可以用来配置一些个性设置，如终端字体大小、色调等等。

```
[root@localhost ~]# vi .bash_profile
# .bash_profile
# Get the aliases and functions
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi
# User specific environment and startup programs
PATH=$PATH:$HOME/bin
export PATH
unset USERNAME
export PATH=$PATH:$HOME/jingzhao
```

保存退出后，重新打开一个新的终端，键入“echo \$PATH”，屏幕上显示的还是修改前的“\$PATH”，这是怎么回事呢？原来/etc/profile 与 .bash\_profile 只在会话时被读取一次，所以必须重启机器让会话进行一次。

```
[root@localhost ~]# reboot
```

随后在重启后的机器中再次键入“echo \$PATH”，这次可以看到修改后的变量了。

```
[root@localhost ~]# echo $PATH
/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/usr/X11R6/bin:/root/bin:/root/jingzhao
```

如果不想每次修改完 PATH 后都要重启, 而且打开新的终端也能应用修改后的变量, 就只能在 .bashrc 与 /etc/bashrc 环境文件中修改了。

```
[root@localhost ~]# vi .bashrc
# .bashrc
# User specific aliases and functions
alias rm='rm -i'
alias cp='cp -i'
alias mv='mv -i'
# Source global definitions
if [ -f /etc/bashrc ]; then
    . /etc/bashrc
fi
export PATH=$HOME/jingzhao
```

保存退出后查看下变量

```
[root@localhost ~]# echo $PATH
/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/usr/X11R6/bin:/root/bin
```

执行 source 命令

```
[root@localhost ~]# source .bashrc
```

再次查看变量内容

```
[root@localhost ~]# echo $PATH
/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/usr/X11R6/bin:/root/bin:/root/jingzhao
```

另外打开一个新的终端也可以看到变量已修改。但这样做也会存在一个较为严重的问题, 就是每次打开一个终端, 文件被读取一次, 目录也会添加一次, 这样将导致 PATH 变量由于目录复制, 不断地增长。如下示:

```
[root@localhost ~]# echo $PATH
/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/usr/X11R6/bin:/root/bin:/root/jingzhao

[root@localhost ~]# bash
[root@localhost ~]# ps
  PID TTY          TIME CMD
  1111 TTY          0:00 ps
```

```

6459 pts/2    00:00:00 bash
6479 pts/2    00:00:00 bash
6508 pts/2    00:00:00 ps
[root@localhost ~]# echo $PATH
/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/usr/X11R6/bin:/root/bin:/root/jingzhao:/root/jingzhao

```

如果使用 bash，可以把它加到 .bash\_profile 文件中（环境文件），这样，它会在每次注册时起作用。一般情况下，PATH 变量中往往有一个目录 /usr/local/bin，这个目录中的命令不是 linux 标准的命令，而是由系统管理员添加和维护的、供所有用户使用的命令。如果 PATH 变量中不存在这个目录，你可以自己将它添加进去。

PS1：提示符：

每次当打开一个控制台时，最先看到的就是提示符，如下所示：

```
[root@localhost ~]#
```

在默认的设置下，提示符将显示用户名、主机名、当前所在目录（“~”表示用户的宿主目录），最后一个字符可以指示是普通用户（\$），还是 root 管理员（#）。

可以通过 \$PS1 变量来设置提示符。如下命令将显示当前的设定值：

```
[root@localhost ~]# echo $PS1
[\u@\h \w]\$
```

上面这些符号都是什么呢？是 PS1 变量的值。

PS1 的值由一系列静态文本或 \ 和转义字符序列组成：

比较有用的转义序列有：

\e ASCII 转义字符

\h 主机名

\H 完整的主机名

\l 终端设备名

\t 24 小时制时间

\T 12 小时制时间

\u 用户名

\w 当前工作目录（绝对路径）

\W 当前工作目录（basename）

!\# 当前命令在历史缓冲区的位置

\\$ 如果当前用户是 super user，则插入字符#；否则插入字符\$

\[ 出现在不移动光标的字符序列之前

\] 出现在非打印字符之后

通过这些转义序列来设置 PS1 的值，可以将提示符进行个性化设置：

对于初学者来说，系统自带的一些设置有些不友好，因为提示符只显示当前目录的最后一部分。如果看到如下所示的提示符：

```
[root@localhost bin]#
```

当前目录可能是/bin、/usr/bin、/usr/local/bin 或者是/me/X11R6/bin，只能用 pwd 命令来查看绝对路径。

能不能让 shell 自动告诉当前目录呢？

当然可以。这里将提到的设置，包括提示符，大都包含在文件/etc/bashrc 中。可以通过编辑各用户/home 目录下的“.bashrc\_profile”和“.bashrc”文件来改变设置。

如下面基本设置可以看到绝对路径：

```
export PS1="[w]\\"$"  
[~/vod/photo]#
```

此外，我们还可以设置提示符的颜色、当前时间、命令的历史记录等等

```
export PS1="\[033[0;31m\][w]\\"$"\[033[0m\] "  
[~/vod/photo]# (颜色改变)
```

## 2、环境变量设置

实验：如何设置一个新的 PATH 环境变量？

### 4.3.2 shell 控制结构

shell 程序设计语言的基础是对条件进行测试判断，根据不同的测试结果采取相应的程序处理。因此，下面我们将会先来分析在 shell 脚本程序中可以使用的判断条件，然后再来学习使用这些条件的控制结构。

#### 1. 4.3.2.1 判断条件

/\*\*\*\*\*/

##### 1 条件测试

条件测试有二种常用形式：

第一种是用 test 命令与系统运算符一起使用，如上所示（例子说明）。

第二种是用一对方括号与系统运算符一起使用。这两种形式是完全等价的。例如，测试位置参数\$1 是否是已存在的普通文件，可写为 test -f "\$1"。也可写成[-f "\$1"]。利用一对方括号表示条件测试时，在左方括号 “[” 之后、右方括号 “]” 之前各应有一个空格。（下面举例说明）

```
#!/bin/bash  
if test -f "$1"  
then echo "$1 is an ordinary file . "
```

```
else echo "$1 is not an ordinary file . "  
fi
```

运行脚本程序得结果与上面完全一样。

/\*\*\*\*\*\*  
/

shell 程序设计中被广泛使用的判断条件主要是布尔判断命令[与 test,在大多数系统中,两者没有太大的区别,都使得程序设计语法看起来更加简单与明了,只是在形式上为了增强可读性,[命令会使用符号]来结尾。

下面分别为两者的语法结构:

```
test 测试条件  
测试条件 ]
```

Test/[ 命令可以使用的条件类型可以归为以下三类:

字符串比较:

算术比较:

与文件有关的条件测试:

字符串比较	结果
String1 = string2	如果两个字符相同则结果为真
String1 != string2	如果两个字符不相同则结果为真
-n String	如果字符串不为空则为真
-z string	如果字符串为空（一个空串）则结果为真

算术比较	结果
Expression1 -eq expression2	如果两个表达式相等则结果为真
Expression1 -ne expression2	如果两个表达式不同则结果为真
Expression1 -g t expression2	如果 expression1 大于 expression2 则结果为真
Expression1 -ge expression2	如果 expression1 大于或等于 expression2 则结果为真
Expression1 - l t expression2	如果 expression1 小于 expression2 则结果为真
Expression1 -le expression2	如果 expression1 小于或等于 expression2 则结果为真
! Expression1	如果表达式为假则结果为真, 反之亦然

文件条件比较	结果
-d file	如果文件是一个目录则结果为真
-e file	如果文件存在则结果为真。要注意的是历史上-e选项不可移植，所以通常使用的是-f选项
-f file	如果文件是一个普通文件则结果为真
-g file	如果文件的 SGID 位被设置则结果为真
-r file	如果文件可读则结果为真
-s file	如果文件的长度不为 0 则结果为真
-u file	如果文件的 SUID 位被设置则结果为真
-w file	如果文件可写则结果为真
-x file	如果文件可执行则结果为真

例子：检查环境变量 PATH 是否为空

```
test -z $PATH
[ -z $PATH ]
```

使用[]时要特别注意，在每个组件之间均要用空格键分隔，否则将会提示出错。

在实际 shell 程序设计中，使用单一判断条件的情况很少，其往往与&&、||等组成复合判断条件来使用。

语法结构：

判断条件 1 && 判断条件 2

判断条件 3 || 判断条件 4

语法分析：

判断条件 1 为真时才会执行判断条件 2，否则忽略判断条件 2

判断条件 3 为假时才会执行判断条件 4，否则忽略判断条件 4

实验 5.3.2.1：设计出满足下面条件的 shell 程序

1、当执行一个程序的时候，这个程序会让用户选择 y 或 n

2、如果用户输入 y，就显示“OK, continue”

3、如果用户输入 n，就显示“Oh, interrupt”

4、如果是其它字符，就显示“Please enter y or n”

```
[/home]# vi sh_1.sh
#!/bin/bash
echo -e "Are you sure?(y/n):\c"
read yn
[ $yn == "y" ] && echo "OK,continue" && exit 0
[ $yn == "n" ] && echo "Oh,interrupt" && exit 0
echo -e "Please enter y or n"
```



```
[/home]# ./sh_1.sh
Are you sure?(y/n):y
OK,continue
```

## 2. 4.3.2.2 条件判断结构

1、if 语句

if 语句用于条件控制结构中

语法格式 1

```
If [ 条件判断表达式 ]; then
```

当条件表达式成立时，可以执行的命令

```
Fi
```

实验 5.3.2.2: 修改实验 5.3.2.1 以下

```
[/home]# vi sh_2.sh
#!/bin/bash
echo -e "Are you sure?(y/n):\c"
read yn
if [ $yn == "y" ] ;then
echo "OK,continue"
exit 0
fi
if [ $yn == "n" ]
echo "Oh,interrupt"
exit 0
fi
echo -e "Please enter y or n"
[/home]# ./sh_1.sh
Are you sure?(y/n):y
OK,continue
```

语法格式 2

```
If [ 条件判断表达式 ];then
```

当条件表达式成立时，可以执行的命令

```
Else
```

当条件表达式不成立时，可以执行的命令

```
Fi
```

实验：修改实验 5.3.2.1 以下

```
[/home]# vi sh_2.sh
#!/bin/bash
```

```

echo -e "Are you sure?(y/n):\c"
read yn
if [ $yn == "y" || $yn == "n" ] ;then
    if [ $yn == "y" ];then
        echo "OK,continue"
        exit 0
    else
        echo "Oh,interrupt"
        exit 0
    fi
else
    echo -e "Please enter y or n"
fi
[/home]# ./sh_2.sh
Are you sure?(y/n):y
OK,continue

```

### 语法格式 3

```
If [ 条件判断表达式 1 ];then
```

当条件表达式 1 成立时，可以执行的命令

```
Elif [ 条件判断表达式 2 ];then
```

当条件表达式 2 成立时，可以执行的命令

```
Else
```

当条件表达式 1 与 2 均不成立时，可以执行的命令

```
fi
```

实验：修改 5.3.2.1 如下

```

[/home]# vi sh_3.sh
#!/bin/bash
echo -e "Are you sure?(y/n):\c"
read yn
if [ $yn == "y" ] ;then
    echo "OK,continue"
    exit 0
elif [$yn == "n" ];then
    echo "Oh,interrupt"
    exit 0

```

```

else
echo -e "Please enter y or n"
fi
[/home]# ./sh_3.sh
Are you sure?(y/n):y
OK,continue

```

## 2、case 语句

case 语句允许进行多重条件选择

语法结构:

```

case $变量名称 in
第一个变量内容)
    程序段(满足第一个变量内容)
;;
第二个变量内容)
    程序段(满足第二个变量内容)
;;
...
*)
    程序段(均不满足前面的条件下)
...
;;
Esac

```

语法分析:

其执行过程是用“字符串”的值依次与各模式字符串进行比较,如果发现同某一个匹配,那么就执行该模式字符串之后的各个命令,直至遇到两个分号为止。如果没有任何模式字符串与该字符串的值相符合,则不执行任何命令。

在使用 case 语句时应注意:

(1) 每个模式字符串后面可有一条或多条命令,其最后一条命令必须以两个分号(即;;)结束。

(2) 模式字符串中可以使用通配符。

(3) 如果一个模式字符串中包含多个模式,那么各模式之间应以竖线(|)隔开,表示各模式是“或”的关系,即只要给定字符串与其中一个模式相配,就会执行其后的命令表。

(4) 各模式字符串应是惟一的,不应重复出现。并且要合理安排它们的出现顺序。例如,不应将“\*”作为头一个模式字符串,因为“\*”可以与任何字符串匹配,它若第一个出现,就不会再检查其它模式了。

(5) case 语句以关键字 case 开头,以关键字 esac(是 case 倒过来写!)结束。

(6) case 的退出（返回）值是整个结构中最后执行的那个命令的退出值。若没有执行任何命令，则退出值为零。

实验：设计一个简单选择菜单，用户输入不同选择时，执行不同动作

```
[root@localhost home]# vi sh.sh
#!/bin/bash
echo "    a) choice a"
echo "    b) choice b"
echo "    c) choice c"
echo -e "Please enter your choice:\c"
read menu_choice
    case "$menu_choice" in
        a) echo "you choice a" ;;
        b) echo "you choice b" ;;
        c) echo "you choice c" ;;
        *) echo "sorry,choice not exist" ;;
    esac
```

运行脚本执行如下：

```
[root@localhost home]# ./sh.sh
    a) choice a
    b) choice b
    c) choice c
Please enter your choice:c
you choice c
[root@localhost home]# ./sh.sh
    a) choice a
    b) choice b
    c) choice c
Please enter your choice:dfjks
sorry,choice not exist
```

### 3. 4.3.2.3 循环结构

循环可以不断执行某个程序段，直到用户设置的条件实现为止。下面介绍几种常用的循环结构。

1、while do done 语句

语法结构：

```
While [ 条件判断表达式 ]
do
程序段
```

done

语法分析:

当条件判断表达式成立时, 就进行循环, 直到条件判断表达式不成立才停止。

实验:

```
[root@localhost home]# ./sh.sh
#!/bin/bash
echo "    a) choice a"
echo "    b) choice b"
echo "    c) choice c"
echo -e "Please enter your choice:\c"
read menu_choice
while [ "$menu_choice" != "a" ] && [ "$menu_choice" != "b" ] &&
[ "$menu_choice" != "c" ]
do
    echo -e "Please enter your choice(a/b/c) to stop this
program:\c"
    read menu_choice
done
```

脚本运行如下:

```
[root@localhost home]# ./sh.sh
    a) choice a
    b) choice b
    c) choice c
Please enter your choice:k
Please enter your choice(a/b/c) to stop this program :e
Please enter your choice(a/b/c) to stop this program :a
```

2、until do done 语句。

语法结构:

Until [ 条件判断表达式 ]

do

程序段

done

语法分析:

与前面的 while do done 刚好相反, 它说的是“当条件判断表达式成立时, 就终止循环, 否则就持续执行循环的程序段”

实验: 用 until do done 结构修改上一实验以实现相同功能

```
[root@localhost home]# vi sh.sh
```

```
#!/bin/bash
echo "    a) choice a"
echo "    b) choice b"
echo "    c) choice c"
echo -e "Please enter your choice:\c"
read menu_choice
until [ "$menu_choice" == "a" ] || [ "$menu_choice" == "b" ] ||
[ "$menu_choice" == "c" ]
do
    echo -e "Please enter your choice(a/b/c) to stop this
program:\c"
    read menu_choice
done
```

脚本运行如下：

```
[root@localhost home]# ./sh.sh
    a) choice a
    b) choice b
    c) choice c
Please enter your choice:k
Please enter your choice(a/b/c) to stop this program :e
Please enter your choice(a/b/c) to stop this program :a
```

读者可以从以上两个实验来好好体会下两种结构间的区别。

### 3、for do done 语句

语法结构：

```
for [ 条件判断表达式 ]
```

```
do
```

程序段

```
done
```

语法分析：

for 语句是最常用的建立循环结构的语句，其条件判断表达式更是形式多样。同 while 一样，是当满足条件判断时，就进行循环，直到条件不成立才停止。

实验 1：用 for 结构实现计算 1+2+3+...+100 之和

```
[root@localhost home]# vi sh.sh
#!/bin/bash
s=0
for ((i=1;i<=100;i=i+1))
do
```

```
s=$((s+$i))
done
echo "the result of '1+2+3+...+100' is ==> $s"
```

脚本运行如下：

```
[root@localhost home]# ./sh.sh
the result of '1+2+3+...+100' is ==> 5050
```

实验 2：用一个变量实现多次赋值

```
#!/bin/bash
for day in Monday Wednesday Friday Sunday
do
    echo "$day"
done
```

运行脚本如下：

```
[root@localhost home]# ./sh.sh
Monday
Wednesday
Friday
Sunday
```

程序分析：

其执行过程是，变量 day 依次取值表中各字符串，即第一次将“Monday”赋给 day，然后进入循环体，执行其中的命令，显示出 Monday。第二次将“Wednesday”赋给 day，然后执行循环体中命令，显示出 Wednesday。依次处理，当 day 把值表中各字符串都取过一次之后，下面 day 的值就变为空串，从而结束 for 循环。因此，值表中字符串的个数就决定了 for 循环执行的次数。在格式上，值表中各字符串之间以空格隔开。

实验 3：（用 for in 结构来实现多个脚本依次执行的功能）

### 4.3.3 其它结构

break 命令和 continue 命令

break 命令可以使我们从循环体中退出来。其语法格式是：

```
break [ n ]
```

其中，n 表示要跳出几层循环。默认值是 1，表示只跳出一层循环。

continue 命令跳过循环体中在它之后的语句，回到本层循环的开头，进行下一次循环。其语法格式是：

```
continue [ n ]
```

其中，n 表示从包含 continue 语句的最内层循环体向外跳到第几层循环。默认值为 1。循环层数是由内向外编号。

### 4.3.4 shell 函数

直到目前为止，我们做实验所编写的 shell 程序都是非常短小的。可在实际应用中，有时我们为项目所编写的脚本程序是非常大型的，这时我们该如何来构造自己的代码呢。很多人可能想到说将我们大型脚本按照功能模块拆分成多个小型脚本，但这种做法存在如下几个缺点：

- 1、在一个脚本程序中运行另外一个脚本程序要比执行一个函数慢得多。
- 2、返回执行结果变得更加困难，而且可能存在非常多的小脚本。

基于上面原因及拆分思想，我们可以定义并使用 shell 函数，其语法格式为：

```
[function] 函数名 ( )  
{  
    命令表 (Statements)  
}
```

语法结构分析：

其中，关键字 function 可以缺省。

通常，函数中的最后一个命令执行之后，就退出被调函数。也可利用 return 命令立即退出函数，其语法格式是：

```
return [ n ]
```

其中，n 值是退出函数时的退出值（退出状态），即\$?的值。当 n 值缺省时，则退出值是最后一个命令执

函数应先定义，后使用。调用函数时，直接利用函数名，如 foo，不必带圆括号，就像一般命令那样使用。其最大作用就是可以简化很多代码，这在较大的 shell 脚本设计中可能会更加明显，如本章中的综合应用。

实验 1：利用 shell 函数写一个简单的显示例子

```
[root@localhost shell]# vi sh.sh  
#!/bin/bash  
first()  
{  
    echo "*****"  
}  
second()  
{  
    echo "====="}
```



```

}
third()
{
    echo "*"
}
four()
{
    echo "* hello,welcome to linux world *"
}
five()
{
    echo "* (http://www.gec-edu.org) *"
}
second
first
third
third
four
five
third
third
first
second

```

运行这个脚本程序会显示如下的输出信息:

```
[root@localhost shell]# ./sh.sh
```

```

=====
*****
*
*
* hello,welcome to linux world *
* (http://www.gec-edu.org) *
*
*
*****
=====

```

实验解析:

这个脚本程序还是从自己的顶部开始执行, 这一点与其他脚本程序没有什么分别。但当它遇见 `first()` 结构时, 它知道定义了一个名为 `first` 的函数。它会记住 `foo` 代表

一个函数并从}字符之后的位置继续执行。当执行到单独的行 foo 时, shell 就知道应该去执行刚才定义的函数了, 当这个函数执行完毕以后, 执行过程会返回到调用 foo 函数的那条语句的后面继续执行。

你必须在调用一个函数之前先对它进行定义, 这有点像 Pascal 语言里函数必须先于调用而被定义的概念, 只是在 shell 中不存在前向声明。

Shell 脚本与函数间的参数传递可利用位置参数和变量直接传递。当一个函数被调用时,

变量的值可以直接由 Shell 脚本传递给被调用的函数, 而脚本程序中所用的位置参数 \$\*, \$@、\$#, \$1、\$2 等则会被替换为函数的参数。当函数执行完毕后, 这些参数会恢复为它们之前的值。

#### 实验 2: 变量作为参数被 shell 函数调用

```
[root@localhost shell]# vi sh_01.sh
#!/bin/bash
input_data()
{
    echo -e "Enter your data:\c"
    read tmp
    data=${tmp%%, *}
}

insert_title()
{
    echo $* >> title_file
    return
}

input_data
insert_title $data
```

运行这个脚本程序会显示如下的输出信息:

```
[root@localhost shell]# ls
sh_01.sh
[root@localhost shell]# ./sh_01.sh
Enter your data:yang jing zhao
[root@localhost shell]# ls
sh_01.sh  title_file
[root@localhost shell]# cat title_file
```

yang jing zhao

### 实验 3: 脚本的位置参数作为参数被 shell 函数调用

```
[root@localhost shell]# vi sh_02.sh
#!/bin/bash
get_sure ()
{
    echo "Is your data: $"
    echo -n "Enter yes or no:"
    while true
    do
        read x
        case "$x" in
            y | yes ) return 0;;
            n | no ) return 1;;
            *)      echo "answer yes or no" ;;
        esac
    done
}

echo "shell parameters you input are: $"
if get_sure "$*"
then
    echo "the data you enter is: $"
else
    echo "you enter nothing"
fi
exit 0
```

运行这个脚本程序会显示如下的输出信息:

```
[root@localhost shell]# ./sh_02.sh yang jing zhao
shell parameters you input are:yang jing zhao
Is your data: yang jing zhao
Enter yes or no:y
the data you enter is:yang jing zhao
[root@localhost shell]# ./sh_02.sh yang jing zhao
shell parameters you input are:yang jing zhao
Is your data: yang jing zhao
Enter yes or no:n
```

```
you enter nothing
```

## 4.4 综合应用

至此，我们已经学习完作为程序设计语言的 shell 的主要功能。是时候将我们所学的知识结合起来以编写一个实际的示范程序了。

纵贯全章，我们将编写及设计一个记录学生成绩的应用程序。

需求分析

设计框图

对数据进行更新、检索和显示的这三项需求表明采用一个简单的菜单应该就足够了。由于所有需要存储的数据全部都是文本，而且假设要记录的学生数不是很多，所以我们没有必要使用复杂的数据库，使用一些简单的文本文件即可。将资料保存在文本文件中将使我们的应用程序比较简单，而且如果需求发生了变化，操纵文本文件总是要比操纵其它类型的文件更加容易。至少我们可以使用一个编辑器来手工输入和删除数据，而不必非要通过编写程序来完成。

首先，

程序说明

(1) 先定义学生个人信息文件、成绩文件和一个临时文件。同时还设置 Ctrl+C 组合键的中断处理，以确保用户中断脚本程序时删除临时文件。

```
students_file="students_file"
achievements_file="achievements_file"
temp_file="temp_file"
trap 'rm -f $temp_file' EXIT
```

(2) 由于在与脚本互动时，经常要对一些选项进行判断选择，所以下面将一些利用较高的功能代码做成两个简单的工具型函数。

```
get_return()
{
    echo -e "Please return \c"
    # read x
    return 0
}

get_confirm()
{
    echo -e "Are you sure?\n"
```

```

while true
do
    read x
    case "$x" in
        y | Y | YES | yes )
            return 0;;
        n | N | no | NO )
            #      echo
            echo "Cancelled"
            return 1;;
        * ) echo "Please enter yes or no" ;;
    esac
done
}

```

### 菜单项

下面函数为主菜单函数。菜单的选项多样，这主要是根据功能模块来设计的，而且内容是动态变化的，当用户选择了

```

set_menu_choice()
{
    echo "Options :-"
    echo
    echo "  a) Add new Student_Information"
    echo "  f) Find Student_Information"
    echo "  c) Count the Student_Informations and scores in the
catalog"
    echo "  s) Reference Information about students"
    if [ "$stu_catnum" != "" ];then
        echo "  l) List scores on $stu_name"
        echo "  r) Remove $stu_name"
        echo "  u) Update achievements for $stu_name"
    fi
    echo "  q) Quit"
    echo
    echo -e "Please enter choice then press return: \c"
    read menu_choice
    return
}

```

#### 4、新增

```
insert_student()          # what
{
    echo $* >> $students_file
    return
}
insert_scores()           # what
{
    echo $* >> $achievements_file
    return
}

add_record_achievements()
{
    echo "Enter score information for this student"
    echo "When no more scores enter q"
    score_num=1
    subject_tttitle=""
    while [ "$subject_tttitle" != "q" ]      # where from
    do
        echo -e "Enter Subject $score_num title: \c"
        read tmp
        subject_tttitle=${tmp%%,*}
        if [ "$tmp" != "$subject_tttitle" ]; then
            echo "Sorry, no commas allowed"
            continue
        fi
        if [ -n "$subject_tttitle" ] ; then
            if [ "$subject_tttitle" != "q" ] ;then
                echo -e "Enter score: \c"
                read tmp
                score=${tmp%%,*}
                insert_scores
                $stu_catnum,$score_num,$subject_tttitle,$score
            fi
        else
            score_num=$((score_num-1))
        fi
    done
}
```

```

        fi
        score_num=$((score_num+1))
    done
}

add_records()
{
    echo -e "Enter student's numble: \c"
    read tmp
    stu_catnum=${tmp%%,*}

    echo -e "Enter student's name: \c"
    read tmp
    stu_name=${tmp%%,*}

    echo -e "Enter student's major: \c"
    read tmp
    stu_major=${tmp%%,*}

    echo -e "Enter student's professor: \c"
    read tmp
    stu_professor=${tmp%%,*}

    echo "about to add new entry"
    echo "$stu_catnum $stu_name $stu_major $stu_professor"

    if get_confirm ;then
        insert_student
$stu_catnum,$stu_name,$stu_major,$stu_professor
        add_record_achievements
    else
        remove_records
    fi
    return
}

```

## 5、删除

```
remove_records()
```

```

{
    if [ -z "$stu_catnum" ];then
        echo "You must select a student first"
        find_student n
    fi
    if [ -n "$stu_catnum" ];then
        echo "You are about to delete $stu_name"
        get_confirm && {
            grep -v "^${stu_catnum}," $students_file > $temp_file
            mv $temp_file $students_file
            grep -v "^${stu_catnum}," $achievements_file >
$temp_file
            mv $temp_file $achievements_file
            stu_catnum=""
            echo "Enter removed"
        }
        get_return
    fi
    return
}

```

## 6、更新

```

update_cd()
{
    if [ -z "$stu_catnum" ];then
        echo "You must select a student first"
        find_student n
    fi

    if [ -n "$stu_catnum" ];then
        echo "Current achievement are :-"
        list_achievements
        echo
        echo "This will re-enter the achievements for $stu_name"
        get_confirm && {
            grep -v "^${stu_catnum}," $achievements_file > $temp_file
            mv $temp_file $achievements_file
            echo

```



```

        add_record_achievements
    }
fi
return
}

```

## 7、检索

```

find_student()
{
    if [ "$1" = "n" ];then
        asklist=n
    else
        asklist=y
    fi
    stu_catnum=""
    echo -e "Enter a string to search for in the student titles:
\c"

    read searchstr
    if [ "$searchstr" = "" ];then
        return 0
    fi
    grep "$searchstr" $students_file > $temp_file
    set $(wc -l $temp_file)
    linesfound=$1
    case "$linesfound" in
    0)  echo "Sorry,nothing found"
        get_return
        return 0
        ;;
    1)  ;;
    2)  echo "Sorry,not unique."
        echo "Found the following"
        cat $temp_file
        get_return
        return 0
    esac
    IFS=","
    read stu_catnum stu_name stu_major stu_professor < $temp_file

```

```

IFS=" "
if [ -z "$stu_catnum" ];then
    echo "Sorry, could not extract catalog field from $temp_file"
    get_return
    return 0
fi
echo
echo "Student' number: $stu_catnum"
echo "Student' name: $stu_name"
echo "Student's major: $stu_major"
echo "Student's professor: $stu_professor"
echo
get_return
if [ "$asklist" = "y" ];then
    echo -e "View scores for this student(y or n?): \c"
    read x
    if [ "$x" = "y" ];then
        echo
        list_achievements
        echo
    fi
fi
return 1
}

```

8、

```

count_students()
{
    set $(wc -l $students_file)
    num_student=$1
    set $(wc -l $achievements_file)
    num_achievement=$1
    echo "found $num_student students, with a total of
$num_achievement achievements"
    get_return
    return
}

```

9、

```
list_achievements()
{
    echo "student's numble=$stu_catnum"
    if [ "$stu_catnum" = "" ];then
        echo "no student select yet"
        return
    else
        grep "^${stu_catnum}," $achievements_file > $temp_file
        num_achievements=$(wc -l $temp_file)
        if [ "$num_achievements" = "0" ];then
            echo "no achievements found for $stu_name"
        else {
            echo
            echo "$stu_name :-"
            echo
            cut -f 2- -d , $temp_file
            echo
        } | ${PAGER:-more}
    fi
    get_return
    return
}
```

10、主程序

```
rm -f $temp_file
if [ ! -f students_file ];then
    touch students_file
fi
if [ ! -f achievements_file ];then
    touch achievements_file
fi

clear
echo
echo
```

```

"*****"
    echo ">>"                                Student_achievement_manager
<<"
    echo ">>"                                http://www.gec-edu.org
<<"
    echo
"*****"

    echo
    sleep 1

quit=n
while [ "$quit" != "y" ];
do
    set_menu_choice
    case "$menu_choice" in
        a)  add_records;;
        r)  remove_records;;
        f)  find_student y;;
        u)  update_cd;;
        c)  count_students;;
        l)  list_achievements;;
        s)
            echo
            more $students_file
            echo
            get_return;;
        q | Q) quit=y;;
        *)  echo "Sorry,choice not recognized";;
    esac
done

rm -f $temp_file
echo "Finish"
exit 0

```

## 本章小结

# 第 5 章文件 IO 编程

## 学习目标

- 文件
- 文件描述符
- 系统调用
- 库函数
- 文件锁
- 多路复用

## 5.1 Linux 文件结构

### 5.1.1 文件

Linux 环境中的文件具有特别重要的意义，因为它们为操作系统服务和设备提供了一个简单而统一的接口。在 Linux 中，一切（几乎一切）都是文件。

设备对操作系统而言也可以看做是文件，通常程序完全可以像使用文件那样使用磁盘文件、串口、打印机和其他设备。还有一些抽象的对象也可以看做是文件，如后面将讲到的网络连接 socket 套接字。大多数情况下，我们只需要使用 5 个基本函数：open、close、read、write 和 lseek。

目录也是一种文件，但它是一种特殊类型的文件。在 Linux 系统中，即使是超级用户可能也不再被允许直接对目录进行操作。正常情况下，所有用户都必须用上层的 opendir/readdir 接口来读取目录，而无需了解特定系统中目录实现的具体细节。

可以这么说 Linux 中的任何事物都可以用一个文件代表，或者可以通过特殊的文件进行操作。当然，它们会与我们熟悉的传统文件有一些细微的区别，但两者的基本原则是一致的

### 5.1.2 文件描述符

在 Linux 中对目录和设备的操作都等同于文件的操作，因此，大大简化了系统对不同设备的处理，提高了效率。内核如何区分和引用特定的文件呢？这里用到的就是一个

重要的概念——文件描述符。对于 Linux 而言，所有对设备和文件的操作都使用文件描述符来进行的。文件描述符是一个非负的整数，它是一个索引值，并指向内核中每个进程打开文件的记录表。当打开一个现存文件或创建一个新文件时，内核就向进程返回一个文件描述符；当需要读写文件时，也需要把文件描述符作为参数传递给相应的函数。

通常，一个进程启动时，都会打开 3 个文件：标准输入、标准输出和标准出错处理。这 3 个文件分别对应文件描述符为 0、1 和 2（也就是宏替换 `STDIN_FILENO`、`STDOUT_FILENO` 和 `STDERR_FILENO`，鼓励读者使用这些宏替换）。

## 5.2 系统调用与库函数

### 5.2.1 系统调用

我们用很少量的函数就可以对文件和设备进行访问和控制。这些函数被称为系统调用，由 Linux 系统直接提供，它们也是通向操作系统本身的接口。

Linux 系统调用部分是非常精简的系统调用（只有 250 个左右），它继承了 UNIX 系统调用中最基本和最有用的部分。这些系统调用按照功能逻辑大致可分为进程控制、进程间通信、文件系统控制、系统控制、存储管理、网络管理、socket 控制、用户管理等几类。

所谓系统调用是指操作系统提供给用户程序调用的一组“特殊”接口，用户程序可以通过这组“特殊”接口来获得操作系统内核提供的服务。为什么用户程序不能直接访问系统内核提供的服务呢？这是由于在 Linux 中，为了更好地保护内核空间，将程序的运行空间分为内核空间 and 用户空间（也就是常称的内核态和用户态），它们分别运行在不同的级别上，在逻辑上是相互隔离的。因此，用户进程在通常情况下不允许访问内核数据，也无法使用内核函数，它们只能在用户空间操作用户数据，调用用户空间的函数。

但是，在有些情况下，用户空间的进程需要获得一定的系统服务（调用内核空间程序），这时操作系统就必须利用系统提供给用户的“特殊接口”——系统调用规定用户进程进入内核空间的具体位置。进行系统调用时，程序运行空间需要从用户空间进入内核空间，处理完后再返回到用户空间。

### 5.2.2 库函数

在输入输出操作中，直接使用底层系统调用的问题是它们的效率非常的低。为什么呢？

（1）系统调用会影响系统的性能。与函数调用相比，系统调用的开销要大些，因为在执行系统调用时，Linux 必须从用户代码切换到内核代码运行，然后再返回用户代

码。减少这种开销的好方法是，在程序中尽量减少系统调用次数，并且让每次系统调用完成尽可能多的工作。

(2) 硬件会对底层系统调用一次所能读写的数据块做出一定的限制。

为了给设备和磁盘文件提供更高层的接口，与 UNIX 一样，Linux 发行版提供了一系列的标准函数库。它们是由一些函数构成的集合，我们可以把它们包含在自己的程序中去处理那些与设备和文件有关的问题。提供输出缓冲功能的标准 IO 库就是一个这样的例子。我们可以高效的写任意长度的数据块，函数则在数据满足数据块长度要求时安排执行底层系统调用。这样极大降低了系统调用的负面影响。

Linux 系统中各种文件函数与用户、设备驱动程序、内核和硬件之间的关系。如图 5-1 所示。

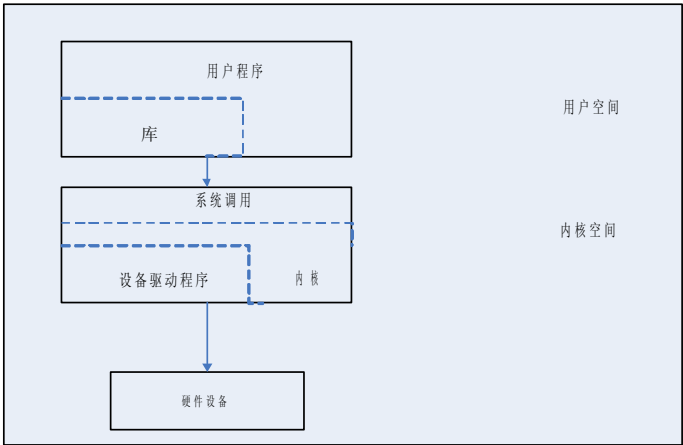


图 5-1 关系图

### 5.3 文件 IO 基本操作

Linux 系统中文件操作主要有：不带缓存的 IO 操作和带缓存的 IO 操作。

#### 5.3.1 不带缓存的 IO 操作

不带缓存的 IO 操作又称底层 IO 操作。文件底层 I/O 操作的系统调用主要用到 5 个函数：open()、close()、read()、write()、lseek()。这些函数的特点是不带缓存，直接对文件进行操作。

虽然不带缓存的文件 IO 操作程序不能移植到非 POSIX 标准的系统(如 Windows 系统)上去，但是在嵌入式程序设计、TCP/IP 的 Socket 套接字程序设计、多路 IO 操作程序设



计等方面应用广泛。因此，不带缓存的文件 IO 程序设计是 Linux 文件操作程序设计的重点。

1. 函数说明

表 5 1 带缓存的文件 IO 操作主要用到的函数

函 数	作 用
open	打开或创建文件（在打开或创建文件时可以指定文件的属性及用户的权限等各种参数）。
close	关闭文件
read	从指定的文件描述符中读出的数据放到缓存区中，并返回实际读入的字节数。
write	从指定的文件描述符中读出的数据放到缓存区中，并返回实际读入的字节数。
lseek	在指定的文件描述符中将文件指针定位到相应的位置。

2. 函数格式

表 5 2 open()函数语法

所需头文件	#include <sys/types.h>	
	#include <sys/stat.h>	
	#include <fcntl.h>	
函数原型	int open(const char * pathname, int flags);	
	int open(const char * pathname, int flags, int perms);	
函数传入值	pathname	被打开的文件名（可包括路径名）
	Flags(文件打开方式)	O_RDONLY: 只读方式打开文件
		O_WRONLY: 可写方式打开文件
		O_RDWR: 读写方式打开文件
		O_CREAT: 如果该文件不存在，就创建一个新的文件，并用第三个参数为其设置权限
		O_EXCL: 如果使用 O_CREAT 时文件存在，则可返回错误消息。这一参数可测试文件是否存在
		O_NOCTTY: 使用本参数时，如文件为终端，那么终端不可以作为调用 open()系统调用的那个进程的控制终端
		O_TRUNC: 如文件已经存在，并且以只读或只写成功打开，那么会先全部删除文件中原有数据
		O_APPEND: 以添加方式打开文件，在打开文件的同时，文件指针指向文件的末尾
	perms	被打开文件的存取权限，为 8 进制表示法
函数返回值	成功：返回文件描述符	

	出错：-1
--	-------

表 5 3 close()函数语法

所需头文件	#include <unistd.h>	
函数原型	int close(int fd);	
函数传入值	fd	文件描述符
函数返回值	0: 成功 -1: 出错	

表 5 4 read()函数语法

所需头文件	#include <unistd.h>	
函数原型	ssize_t read(int fd, void *buf, size_t count);	
函数传入值	fd	文件描述符
	buf	存储内容的内存空间
	count	读取的字节数
函数返回值	0: 成功 -1: 出错	

表 5 5 write()函数语法

所需头文件	#include <unistd.h>	
函数原型	ssize_t write(int fd, void *buf, size_t count);	
函数传入值	fd	文件描述符
	buf	存储内容的内存空间
	count	读取的字节数
函数返回值	0: 成功 -1: 出错	

表 5 6 lseek()函数语法

所需头文件	#include <unistd.h> #include <sys/types.h>	
函数原型	ssize_t lseek(int fd, off_t offset, int whence);	
函数传入值	fd	文件描述符
	offset	偏移量（可为负值）
	whence （基点）	SEEK_SET: 文件开头+offset 为新读写位置
		SEEK_CUR: 目前读写位置+offset 为新读写位置
		SEEK_END: 文件结尾+offset 为新读写位置

函数返回值	0: 成功
	-1: 出错

### 3. 使用实例

程序功能：设计一个程序，要求从一个源文件 src\_file（如不存在则创建）中读取倒数第二个 10KB 数据并复制到目标文件 dest\_file。

程序说明：

程序代码：

```
/* copy_file.c */

#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>

#define BUFFER_SIZE 1024 /* 每次读写缓存大小，影响运行效率*/
#define SRC_FILE_NAME "src_file" /* 源文件名 */
#define DEST_FILE_NAME "dest_file" /* 目标文件名文件名 */
#define OFFSET 20480 /* 拷贝的数据大小 */

int main()
{
    int src_file, dest_file;
    unsigned char buff[BUFFER_SIZE];
    int real_read_len;
    int flag;
    /* 以只读方式打开源文件 */
    src_file = open(SRC_FILE_NAME, O_RDONLY);

    /* 以只写方式打开目标文件，若此文件不存在则创建，访问权限值为 644 */
    dest_file = open(DEST_FILE_NAME,
O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);
```

```

    if (src_file < 0 || dest_file < 0)
    {
        printf("Open file error\n");
        exit(1);
    }

    /* 将源文件的读写指针移到最后 20KB 的起始位置*/
    lseek(src_file, -OFFSET, SEEK_END);

    /* 读取源文件的最后 20KB 数据并写到目标文件中，每次读写 1KB，读取 10
次 */
    while ((real_read_len = read(src_file, buff,
sizeof(buff))) > 0 || flag < 0)
    {
        flag--;
        write(dest_file, buff, real_read_len);
    }

    close(dest_file);
    close(src_file);

    return 0;
}

```

结果：

```

# ./copy_file
# ls -lh dest_file

```

### 5.3.2 带缓存的 IO 操作

前面所述的文件 IO 操作是基于文件描述符的。这些都是基本的 IO 控制，是不带缓存的。

本节说要介绍的 IO 操作是基于流缓冲的，它是符合 ANSI C 的标准 IO 处理，这里有很多读者已经非常熟悉的函数如：printf()、scanf() 等。因此本节仅介绍最主要的函数。

带缓存的文件 IO 操作是在内存中开辟一个“缓冲区”，为程序中的每一个文件使

用。当执行读文件的操作时，从磁盘文件中将数据先读入内存“缓冲区”，装满后再从内存“缓冲区”依次读入接收的数据。反之亦然。

带缓存的文件 IO 操作主要用到的函数如下表。

1. 函数说明

表 5 7 带缓存的文件 IO 操作主要用到的函数

函数	作    用
fopen	打开或创建文件
fclose	关闭文件
fread	由文件中读取一个字符
fwrite	将数据成块写入文件流
fseek	移动文件流的读写位置

2. 函数格式

表 5 8 fopen()函数语法

所需头文件	#include <stdio.h>	
函数原型	FILE * fopen(const char * path, const char * mode)	
函数传入值	path	包含欲打开的文件路径及文件名
	mode	文件打开状态
函数返回值	成功：指向 FILE 的指针	
	出错：返回 NULL	

表 5 9 mode 取值说明

函    数	作    用
r	打开只读文件，该文件必须存在
r+	打开可读写文件，该文件必须存在
w	打开只写文件，若文件存在则文件长度清为 0，即擦除文件以前内容。若文件不存在则建立该文件
w+	打开可读写文件，若文件存在则文件长度清为 0，即会擦些文件以前内容。若文件不存在则建立该文件
a	以附加的方式打开只写文件。若文件不存在，则会建立该文件；如果文件存在，写入的数据会被加到文件尾，即文件原先的内容会被保留
a+	以附加方式打开可读写的文件。若文件不存在，则会建立该文件；如果文件存在，写入的数据会被加到文件尾后，即文件原先的内容会被保留

上述形态字符串都可以再加上一个 b 字符，如 rb、w+b、ab+等组合，加入 b 字符用来告诉函数库打开的文件为二进制文件，而非纯文字文件。

表 5 10 fclose()函数语法

所需头文件	#include <stdio.h>
函数原型	int fclose(FILE * stream)
函数传入值	文件地址
函数返回值	成功：返回 0 出错：返回 EOF

表 5 11 fread()函数语法

所需头文件	#include <stdio.h>
函数原型	size_t fread(void* ptr, size_t size, size_t nmemb, FILE* stream)
函数传入值	ptr: 欲写入的数据地址
	size: 字符串长度
	nmemb: 字符串数目
	stream: 一个文件流
函数返回值	成功：返回实际读取到的 nmemb 数目 出错：返回 EOF

表 5 12 fwrite()函数语法

所需头文件	#include <stdio.h>
函数原型	size_t fwrite(const void * ptr, size_t size, size_t nmemb, FILE * stream)
函数传入值	ptr: 欲写入的数据地址
	size: 字符串长度
	nmemb: 字符串数目
	stream: 一个文件流
函数返回值	成功：返回实际写入的 nmemb 数目 出错：返回 EOF

表 5 13 fseek()函数语法

所需头文件	#include <stdio.h>	
函数原型	int fseek(FILE * stream, long offset, int whence)	
函数传入值	*stream	文件地址
	offset	偏移量（可为负值）
	whence	SEEK_SET: 文件开头+offset 为新读写位置

	(基点)	SEEK_CUR: 目前读写位置+offset 为新读写位置
		SEEK_END: 文件结尾+offset 为新读写位置
函数返回值	0: 成功 -1: 出错	

### 3. 使用实例

程序功能：设计一个程序，要求从一个源文件 src\_file（如不存在则创建）中读取倒数第二个 10KB 数据并复制到目标文件 dest\_file。

程序说明：

程序代码：

```
/* standard_io.c */

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

#define BUFFER_SIZE 1024 /* 每次读写缓存大小 */
#define SRC_FILE_NAME "src_file" /* 源文件名 */
#define DEST_FILE_NAME "dest_file" /* 目标文件名文件名 */
#define OFFSET 20480 /* 拷贝的数据大小 */

int main()
{
    FILE *src_file, *dest_file;
    unsigned char buff[BUFFER_SIZE];
    int real_read_len;
    int flag;

    /* 以只读方式打开源文件 */
    src_file = fopen(SRC_FILE_NAME, "r");

    /* 以只写方式打开目标文件，若此文件不存在则创建 */
    dest_file = fopen(DEST_FILE_NAME, "w");
```

```

    if (!src_file || !dest_file)
    {
        printf("Open file error\n");
        exit(1);
    }

    /* 将源文件的读写指针移到最后 20KB 的起始位置*/
    fseek(src_file, -OFFSET, SEEK_END);

    /* 读取源文件的最后 20KB 数据并写到目标文件中, 每次读写 1KB, 读取 10
次 */
    while ((real_read_len = fread(buff, 1, sizeof(buff),
src_file)) > 0 || flag < 0)
    {
        Flag--;
        fwrite(buff, 1, real_read_len, dest_file);
    }

    fclose(dest_file);
    fclose(src_file);
    return 0;
}

```

## 5.4 文件 IO 高级操作

上节为文件 IO 基本操作, 是 Linux 系统最基本操作。本节将在基本操作的基础上学习关于文件 IO 的高级操作, 包括: 文件锁与多路复用。以下操作以不带缓存的 IO 为例, 带缓存的 IO 高级操作留给读者自己思考。

### 5.4.1 文件锁

前面的这 5 个基本函数实现了文件的打开、读写等基本操作, 这一节将讨论的是, 在文件已经共享的情况下如何操作, 也就是当多个用户共同使用、操作一个文件的情况, 这时, Linux 通常采用的方法是给文件上锁, 来避免共享的资源产生竞争的状态。

文件锁包括建议性锁和强制性锁。建议性锁要求每个上锁文件的进程都要检查是否



有锁存在，并且尊重已有的锁。在一般情况下，内核和系统都不使用建议性锁。强制性锁是由内核执行的锁，当一个文件被上锁进行写入操作的时候，内核将阻止其他任何文件对其进行读写操作。采用强制性锁对性能的影响很大，每次读写操作都必须检查是否有锁存在。

在 Linux 中，实现文件上锁的函数有 `lockf` 和 `fcntl`，其中 `lockf` 用于对文件施加建议性锁，而 `fcntl` 不仅可以施加建议性锁，还可以施加强制锁。同时，`fcntl` 还能对文件的某一记录进行上锁，也就是记录锁。

记录锁又可分为读取锁和写入锁，其中读取锁又称为共享锁，它能够使多个进程都能在文件的同一部分建立读取锁。而写入锁又称为排斥锁，在任何时刻只能有一个进程在文件的某个部分上建立写入锁。当然，在文件的同一部分不能同时建立读取锁和写入锁。

1. 函数说明

表 5 14 文件锁

函数	作用
<code>fcntl</code>	根据文件描述词来操作文件的特性

2. 函数格式

表 5 15 `fcntl()`函数语法

所需头文件	<code>#include &lt;sys/types.h&gt;</code> <code>#include &lt;unistd.h&gt;</code> <code>#include &lt;fcntl.h&gt;</code>	
函数原型	<code>int fcntl(int fd, int cmd);</code> <code>int fcntl(int fd, int cmd, long arg);</code> <code>int fcntl(int fd, int cmd, struct flock *lock);</code> /*本节用到的原型*/	
函数传入值	<code>fd</code>	文件描述符
	<code>cmd</code>	操作命令
	<code>lock</code>	结构为 <code>flock</code> ，设置记录锁的具体状态，后面会详细说明
函数返回值	<code>0</code> : 成功 <code>-1</code> : 出错，错误原因存于 <code>errno</code>	

根据 `cmd` 参数的不同，`fcntl` 函数有 5 种功能：

- (1) 复制一个现有的描述符 (`cmd=F_DUPFD`)。
- (2) 获得/设置文件描述符标记 (`cmd=F_GETFD` 或 `F_SETFD`)。
- (3) 获得/设置文件状态标志 (`cmd=F_GETFL` 或 `F_SETFL`)。
- (4) 获得/设置异步 I/O 所有权 (`cmd=F_GETOWN` 或 `F_SETOWN`)。
- (5) 获得/设置记录锁 (`cmd=F_GETLK`、`F_SETLK` 或 `F_SETLKW`)。

cmd 参数说明如下。

表 5 16 cmd 取值说明

参数 cmd	说 明
F_DUPFD	复制文件描述符
F_GETFD	获得 fd 的 close-on-exec 标志，若标志未设置，则文件经过 exec
F_SETFD	设置 close-on-exec 标志，该标志以参数 arg 的 FD_CLOEXEC 位决定
F_GETFL	读取文件状态标志
F_SETFL	设置文件状态标志（其中 O_RDONLY, O_WRONLY, O_RDWR, O_CREAT, O_EXCL, O_NOCTTY 和 O_TRUNC 不受影响，可以更改的标志有 O_APPEND, O_ASYNC, O_DIRECT, O_NOATIME 和 O_NONBLOCK）
F_GETOWN	检索将收到 SIGIO 和 SIGURG 信号的进程号或进程组号
F_SETOWN	设置进程号或进程组号
F_GETLK	根据 lock 描述，决定是否上文件锁
F_SETLK	设置 lock 描述的文件锁
F_SETLKW	这是 F_SETLK 的阻塞版本

注意：

fcntl 是一个非常通用的函数，它可以改变已打开的文件的性质。fcntl 有以上所述的 5 种功能，在本节中主要介绍第 5 种功能：获得/设置记录锁。

一般情况下，第三个参数总是一个整数。但是在本节说明记录锁时，第三个参数则是指向一个结构的指针 flock。

这里，lock 的结构如下所示：

```
Struct flock
{
    short l_type;
    off_t l_start;
    short l_whence;
    off_t l_len;
    pid_t l_pid;    ""
}
```

lock 结构中每个变量的取值含义如表 5 17 所示。

表 5 17 lock 结构体变量取值

参 数	说 明
l_type	F_RDLCK: 读取锁（共享锁）
	F_WRLCK: 写入锁（排斥锁）

	F_UNLCK: 解锁
l_start	相对位移量 (字节)
l_whence	SEEK_SET: 文件开头+l_start 为新读写位置
	SEEK_CUR: 目前读写位置+l_start 为新读写位置
	SEEK_END: 文件结尾+l_start 为新读写位置
l_len	加锁区域的长度
l_pid	

### 3. 使用实例

程序功能：使用 fcntl（）设计一个文件上锁子程序。在分别设计两个程序 write\_lock.c 和 read\_lock.c 调用这个上锁子程序。开多个中断同时运行这两个程序，观察结果。

程序说明：

程序代码：

```
/* lock_set.c */

int lock_set(int fd, int type)
{
    struct flock old_lock, lock;
    lock.l_whence = SEEK_SET;
    lock.l_start = 0;
    lock.l_len = 0;
    lock.l_type = type;
    lock.l_pid = -1;

    /* 判断文件是否可以上锁 */
    fcntl(fd, F_GETLK, &lock);

    if (lock.l_type != F_UNLCK)
    {
        /* 判断文件不能上锁的原因 */
        if (lock.l_type == F_RDLCK) /* 该文件已有读取锁 */
```

```

    {
        printf("Read lock already set by %d\n", lock.l_pid);
    }
    else if (lock.l_type == F_WRLCK) /* 该文件已有写入锁 */
    {
        printf("Write lock already set by %d\n", lock.l_pid);
    }
}

/* l_type 可能已被 F_GETLK 修改过 */
lock.l_type = type;

/* 根据不同的 type 值进行阻塞式上锁或解锁 */
if ((fcntl(fd, F_SETLKW, &lock)) < 0)
{
    printf("Lock failed:type = %d\n", lock.l_type);
    return 1;
}

switch(lock.l_type)
{
    case F_RDLCK:
    {
        printf("Read lock set by %d\n", getpid());
    }
    break;

    case F_WRLCK:
    {
        printf("Write lock set by %d\n", getpid());
    }
    break;

    case F_UNLCK:
    {
        printf("Release lock by %d\n", getpid());
        return 1;
    }
}

```

```

        break;

        default:
            break;
    }/* end of switch */

    return 0;
}

```

下面的实例是文件写入锁的测试用例，这里首先创建一个 hello 文件，之后对其上写入锁，最后释放写入锁，代码如下所示：

```

/* write_lock.c */
#include <unistd.h>
#include <sys/file.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include "lock_set.c"

int main(void)
{
    int fd;

    /* 首先打开文件 */
    fd = open("hello", O_RDWR | O_CREAT, 0644);
    if(fd < 0)
    {
        printf("Open file error\n");
        exit(1);
    }

    /* 给文件上写入锁 */
    lock_set(fd, F_WRLCK);
    getchar();
}

```

```

    /* 给文件解锁 */
    lock_set(fd, F_UNLCK);
    getchar();

    close(fd);

    exit(0);
}

```

接下来的程序是文件读取锁的测试用例，原理和上面的程序一样。

```

/* read_lock.c */

#include <unistd.h>
#include <sys/file.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include "lock_set.c"

int main(void)
{
    int fd;

    fd = open("hello", O_RDWR | O_CREAT, 0644);
    if(fd < 0)
    {
        printf("Open file error\n");
        exit(1);
    }

    /* 给文件上读取锁 */
    lock_set(fd, F_RDLCK);
    getchar();

    /* 给文件解锁 */
    lock_set(fd, F_UNLCK);
}

```

```

    getchar();

    close(fd);
    exit(0);
}

```

## 5.4.2 多路复用

前面的 `fcntl` 函数解决了文件的共享问题，接下来该处理 I/O 复用的情况了。

总的来说，I/O 处理的模型有 5 种。

**阻塞 I/O 模型：**在这种模型下，若所调用的 I/O 函数没有完成相关的功能就会使进程挂起，直到相关数据到才会出错返回。如常见对管道设备、终端设备和网络设备进行读写时经常会出现这种情况。

**非阻塞模型：**在这种模型下，当请求的 I/O 操作不能完成时，则不让进程睡眠，而且返回一个错误。非阻塞 I/O 使用户可以调用不会永远阻塞的 I/O 操作，如 `open`、`write` 和 `read`。如果该操作不能完成，则会立即出错返回，且表示该 I/O 如果该操作继续执行就会阻塞。

**I/O 多路转接模型：**在这种模型下，如果请求的 I/O 操作阻塞，且它不是真正阻塞 I/O，而是让其中的一个函数等待，在这期间，I/O 还能进行其他操作。如本节要介绍的 `select` 函数和 `poll` 函数，就是属于这种模型。

**信号驱动 I/O 模型：**在这种模型下，通过安装一个信号处理程序，系统可以自动捕获特定信号的到来，从而启动 I/O。这是由内核通知用户何时可以启动一个 I/O 操作决定的。

**异步 I/O 模型：**在这种模型下，当一个描述符已准备好，可以启动 I/O 时，进程会通知内核。现在，并不是所有的系统都支持这种模型。

可以看到，`select` 的 I/O 多路转接模型是处理 I/O 复用的一个高效的方法。它可以具体设置每一个所关心的文件描述符的条件、希望等待的时间等，从 `select` 函数返回时，内核会通知用户已准备好的文件描述符的数量、已准备好的条件等。通过使用 `select` 返回值，就可以调用相应的 I/O 处理函数了。

### 1. 函数说明

表 6.19 文件锁

函 数	作 用
<code>select</code>	根据希望进行的文件操作对文件描述符进行了分类处理
<code>poll</code>	同上，但效率更高(本章省略具体函数用法)

2. 2、函数格式

表 6.20 select ( ) 函数语法

所需头文件	#include <sys/types.h> #include <sys/time.h> #include <unistd.h>	
函数原型	int select(int numfds , fd_set *readfds, fd_set *writefds, fd_set *exeptfds, struct timeval *timeout)	
函数传入值	numfds: 需要检查的号码最高的文件描述符加 1	
	readfds: 由 select() 监视的读文件描述符集合	
	writefds: 由 select() 监视的写文件描述符集合	
	exeptfds: 由 select() 监视的异常处理文件描述符集合	
	timeout	NULL: 永远等待, 直到捕捉到信号或文件描述符已准备好为止 具体值: struct timeval 类型的指针, 若等待为 timeout 时间还没有文件描符准备好, 就立即返回 0: 从不等待, 测试所有指定的描述符并立即返回
函数返回值	成功: 准备好的文件描述符 出错: -1	

思考：如何确定被监视的文件描述符的最大值？

可以看到，select() 函数根据希望进行的文件操作对文件描述符进行了分类处理，这里，对文件描述符的处理主要涉及 4 个宏函数，如表 所示

表 6.21 select ( ) 文件描述符处理函数

FD_ZERO(fd_set *set)	清除一个文件描述符集
FD_SET(int fd, fd_set *set)	将一个文件描述符加入文件描述符集中
FD_CLR(int fd, fd_set *set)	将一个文件描述符从文件描述符集中清除
FD_ISSET(int fd, fd_set *set)	测试该集中的一个给定位是否有变化

一般来说，在使用 select 函数之前，首先使用 FD\_ZERO 和 FD\_SET 来初始化文件描述符集，在使用了 select 函数时，可循环使用 FD\_ISSET 测试描述符集，在执行完对相关后文件描述符后，使用 FD\_CLR 来清除描述符集。

另外，select 函数中的 timeout 是一个 struct timeval 类型的指针，该结构体如下所示：

```
Struct timeval{
```



```

long tv_sec; /*second*/
long tv_unsec; /*andmicroseconds*/
}

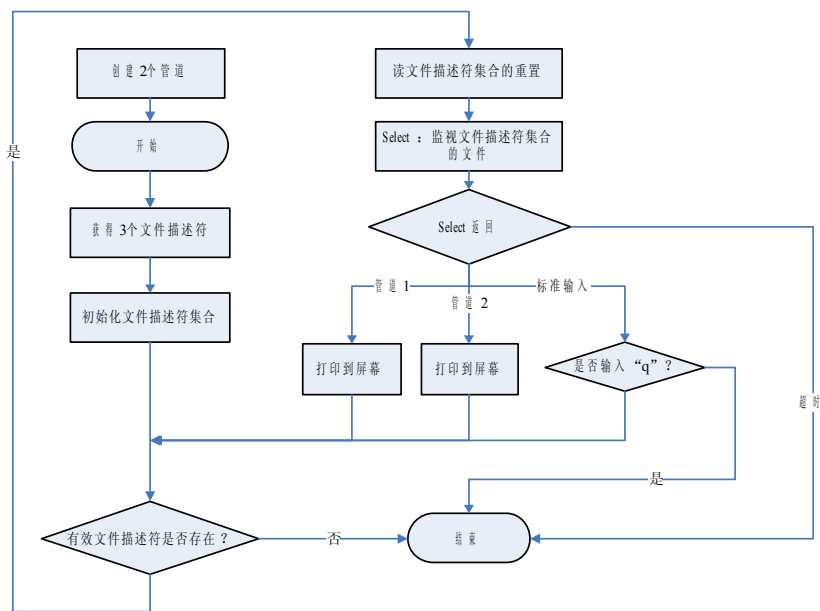
```

可以看到，这个时间结构体的精确度可以设置到微秒级，这对于大多数的应用而言已经足够了。

### 3. 使用实例

程序功能：首先用 `mknod` 命令（`mknod in1 p`）手动创建 2 个管道文件，通过调用 `select()` 函数来监听 3 个终端的输入（标准输入和 2 个管道文件）。以两个管道作为数据输入，主程序从两个管道文件读取字符串并写入到标准输出文件（屏幕）。

程序说明：



图

6. select 程序流程图

程序代码：

```

/* multiplex_select.c */

#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

```

```

#include <time.h>
#include <errno.h>

#define MAX_BUFFER_SIZE    1024          /* 缓冲区大小*/
#define IN_FILES           3             /* 多路复用输入文件数目*/
#define TIME_DELAY         60           /* 超时时间秒数 */
#define MAX(a, b)          ((a > b)?(a):(b))

int main(void)
{
    int fds[IN_FILES];
    char buf[MAX_BUFFER_SIZE];
    int i, res, real_read, maxfd;
    struct timeval tv;
    fd_set inset, tmp_inset;

    /*首先按一定的权限打开两个源文件*/
    fds[0] = 0;

    if((fds[1] = open ("in1", O_RDONLY|O_NONBLOCK)) < 0)
    {
        printf("Open in1 error\n");
        return 1;
    }

    if((fds[2] = open ("in2", O_RDONLY|O_NONBLOCK)) < 0)
    {
        printf("Open in2 error\n");
        return 1;
    }

    /*取出两个文件描述符中的较大者*/
    maxfd = MAX(MAX(fds[0], fds[1]), fds[2]);

    /*初始化读集合 inset，并在读集合中加入相应的描述集*/
    FD_ZERO(&inset);
    for (i = 0; i < IN_FILES; i++)

```

```

{
    FD_SET(fds[i], &inset);
}
FD_SET(0, &inset);

```

```

tv.tv_sec = TIME_DELAY;
tv.tv_usec = 0;

```

/\*循环测试该文件描述符是否准备就绪，并调用 select 函数对相关文件描述符做对应操作\*/

```

while(FD_ISSET(fds[0], &inset) || FD_ISSET(fds[1], &inset)
|| FD_ISSET(fds[2], &inset))
{
    tmp_inset = inset;
    res = select(maxfd + 1, &tmp_inset, NULL, NULL, &tv);

    switch(res)
    {
        case -1:
        {
            printf("Select error\n");
            return 1;
        }
        break;

        case 0: /* Timeout */
        {
            printf("Time out\n");
            return 1;
        }
        break;

        default:
        {
            for (i = 0; i < IN_FILES; i++)
            {
                if (FD_ISSET(fds[i], &tmp_inset))
                {

```

```

        memset(buf, 0, MAX_BUFFER_SIZE);
        real_read = read(fds[i], buf,
MAX_BUFFER_SIZE);

        if (real_read < 0)
        {
            if (errno != EAGAIN)
            {
                return 1;
            }
        }
        else if (!real_read)
        {
            close(fds[i]);
            FD_CLR(fds[i], &inset);
        }
        else
        {
            if (i == 0)
            {
                if ((buf[0] == 'q') || (buf[0] ==
'Q'))
                {
                    return 1;
                }
            }
            else
            {
                buf[real_read] = '\0';
                printf("%s", buf);
            }
        }
    } /* end of if */
} /* end of for */
break;

} /* end of switch */

```

```
    } /*end of while */  
  
    exit(0);  
}
```

读者可以将以上程序交叉编译，并下载到开发板上运行。以下是运行结果：

```
# mknod in1 p  
# mknod in2 p
```

在管道 1 的终端中输入：

```
# cat>in1  
12345
```

在管道 2 的终端中输入：

```
# cat>in2  
abcde
```

在标准输入终端中可以看到：

```
# ./multiplex_select  
12345  
abcde
```

## 5.5 本章小结

文件底层 I/O 操作主要介绍了 5 个系统调用函数：`open()`、`close()`、`read()`、`write()`、`lseek()`。这些函数的特点是不带缓存，直接对文件进行操作。

文件底层 I/O 操作是 Linux 应用编程的基础，后面章节的内容都需要使用这部分知识。如网络编程中的 `socket` 套接字文件，多台电脑之间通过对套接字文件的读写操作既完成了网络通信。

## 5.6 综合实验：文件读写及上锁

### 1. 实验目的

通过编写文件读写及上锁的程序，进一步熟悉 Linux 中文件 I/O 相关的应用开发，并且熟练掌握 `open()`、`read()`、`write()`、`fcntl()` 等函数的使用。

### 2. 实验内容

在 Linux 中 FIFO（先进先出）是一种进程间的管道通信机制。本实验通过使用文件

操作，仿真 FIFO 结构以及生产者-消费者运行模型。

3. 实验步骤

(1) 流程图

该实验流程图如图 6.4 所示。

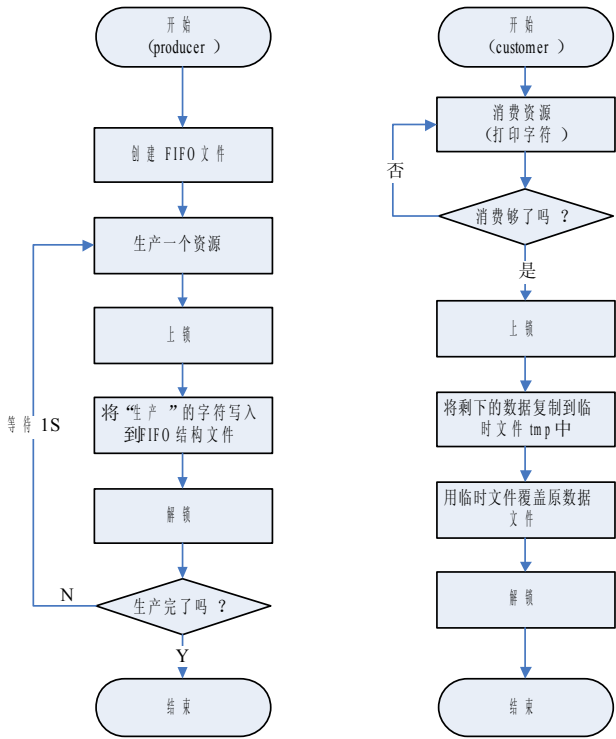


图 6.2 流程图

(2) 程序说明

本实验需要打开两个虚拟终端，分别运行生产者程序（producer）和消费者程序（customer）。此时两个进程同时对同一个文件进行读写操作。因为这个文件是临界资源，所以可以使用文件锁机制保证两个进程对文件的访问都是原子操作。

先启动生产者进程，它负责创建仿真 FIFO 结构文件（实际是一个普通文件）并投入生产，就是按照给定的时间间隔，向 FIFO 文件写入自动生成的字符（在程序中用宏定义选择使用数字还是使用英文字符），生产周期以及要生产的资源数通过参数传递给进程（默认生产周期 1S，要生产的资源数为 10 个字符）。

后启动的消费者进程按照给定的数目进行消费，首先从文件中读取相应数目的字符并在屏幕显示，然后从文件中删除刚才消费过的数据。为了仿真 FIFO 结构，此时需要使用两次复制来实现文件内容的偏移。每次消费的资源数通过参数传递给进程，默认值为

10 个字符。

(3) 代码

```
/* lock_set.c */
int lock_set(int fd, int type)
{
    struct flock old_lock, lock;
    lock.l_whence = SEEK_SET;
    lock.l_start = 0;
    lock.l_len = 0;
    lock.l_type = type;
    lock.l_pid = -1;

    /* 判断文件是否可以上锁 */
    fcntl(fd, F_GETLK, &lock);

    if (lock.l_type != F_UNLCK)
    {
        /* 判断文件不能上锁的原因 */
        if (lock.l_type == F_RDLCK) /* 该文件已有读取锁 */
        {
            printf("Read lock already set by %d\n", lock.l_pid);
        }
        else if (lock.l_type == F_WRLCK) /* 该文件已有写入锁 */
        {
            printf("Write lock already set by %d\n", lock.l_pid);
        }
    }

    /* l_type 可能已被 F_GETLK 修改过 */
    lock.l_type = type;

    /* 根据不同的 type 值进行阻塞式上锁或解锁 */
    if ((fcntl(fd, F_SETLKW, &lock)) < 0)
    {
        printf("Lock failed:type = %d\n", lock.l_type);
        return 1;
    }
}
```

```

switch(lock.l_type)
{
    case F_RDLCK:
    {
        printf("Read lock set by %d\n", getpid());
    }
    break;

    case F_WRLCK:
    {
        printf("Write lock set by %d\n", getpid());
    }
    break;

    case F_UNLCK:
    {
        printf("Release lock by %d\n", getpid());
        return 1;
    }
    break;

    default:
    break;
}/* end of switch */

return 0;
}

```

本实验中的生产者程序的源代码如下所示，其中用到的 lock\_set() 函数。

```

/* producer.c */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include "mylock.h"

```



```

#define MAXLEN          10 /* 缓冲区大小最大值*/

#define ALPHABET        1  /* 表示使用英文字符 */
#define ALPHABET_START  'a' /* 头一个字符, 可以用 'A' */
#define COUNT_OF_ALPHABET 26 /* 字母字符的个数 */

#define DIGIT           2  /* 表示使用数字字符 */
#define DIGIT_START     '0' /* 头一个字符 */
#define COUNT_OF_DIGIT  10 /* 数字字符的个数 */

#define SIGN_TYPE       ALPHABET /* 本实例选用英文字符 */

const char *fifo_file = "./myfifo"; /* 仿真 FIFO 文件名 */

char buff[MAXLEN]; /* 缓冲区 */

/* 功能: 生产一个字符并写入到仿真 FIFO 文件中 */
int product(void)
{
    int fd;
    unsigned int sign_type, sign_start, sign_count, size;
    static unsigned int counter = 0;

    /* 打开仿真 FIFO 文件 */
    if ((fd = open(fifo_file, O_CREAT|O_RDWR|O_APPEND, 0644))
< 0)
    {
        printf("Open fifo file error\n");
        exit(1);
    }

    sign_type = SIGN_TYPE;

    switch(sign_type)
    {
        case ALPHABET: /* 英文字符 */
            {

```

```

        sign_start = ALPHABET_START;
        sign_count = COUNT_OF_ALPHABET;
    }
    break;

    case DIGIT:/* 数字字符 */
    {
        sign_start = DIGIT_START;
        sign_count = COUNT_OF_DIGIT;
    }
    break;

    default:
    {
        return -1;
    }
}/*end of switch*/

sprintf(buff, "%c", (sign_start + counter));
counter = (counter + 1) % sign_count;

lock_set(fd, F_WRLCK); /* 上写锁*/
if ((size = write(fd, buff, strlen(buff))) < 0)
{
    printf("Producer: write error\n");
    return -1;
}
lock_set(fd, F_UNLCK); /* 解锁 */

close(fd);
return 0;
}

int main(int argc ,char *argv[])
{
    int time_step = 1; /* 生产周期 */
    int time_life = 10; /* 需要生产的资源数 */

```

```

    if (argc > 1)
    {
        sscanf(argv[1], "%d", &time_step);
    }

    if (argc > 2)
    {
        sscanf(argv[2], "%d", &time_life);
    }

    while (time_life--)
    {
        if (product() < 0)
        {
            break;
        }
        sleep(time_step);
    }

    exit(EXIT_SUCCESS);
}

```

本实验中的消费者程序的源代码如下所示。

```

/* customer.c */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>

#define MAX_FILE_SIZE      100 * 1024 * 1024 /* 100M*/

const char *fifo_file = "./myfifo";      /* 仿真 FIFO 文件名 */
const char *tmp_file = "./tmp";          /* 临时文件名 */

/* 资源消费函数 */
int customing(const char *myfifo, int need)
{

```

```

int fd;
char buff;
int counter = 0;

if ((fd = open(myfifo, O_RDONLY)) < 0)
{
    printf("Function customing error\n");
    return -1;
}

printf("Enjoy:");
lseek(fd, SEEK_SET, 0);

while (counter < need)
{
    while ((read(fd, &buff, 1) == 1) && (counter < need))
    {
        fputc(buff, stdout); /* 消费就是在屏幕上简单的显示 */
        counter++;
    }
}

fputs("\n", stdout);
close(fd);
return 0;
}

```

/\* 功能:从 sour\_file 文件的 offset 偏移处开始将 count 字节大小的数据拷贝到 dest\_file 文件 \*/

```

int myfilecopy(const char *sour_file, const char *dest_file, int
offset,
int count, int copy_mode)
{
    int in_file, out_file;
    int counter = 0;
    char buff_unit;

    if ((in_file = open(sour_file,O_RDONLY|O_NONBLOCK))<0)

```

```

    {
        printf("Function myfilecopy error in source file\n");
        return -1;
    }

    if((out_file=open(dest_file,
O_CREAT|O_RDWR|O_TRUNC|O_NONBLOCK, 0644)) < 0)
    {
        printf("Function myfilecopy error in
destination file:");
        return -1;
    }

    lseek(in_file, offset, SEEK_SET);
    while((read(in_file, &buff_unit, 1) == 1) && (counter < count))
    {
        write(out_file, &buff_unit, 1);
        counter++;
    }

    close(in_file);
    close(out_file);

    return 0;
}

/* 功能: 实现 FIFO 消费者 */
int custom(int need)
{
    int fd;

    /* 对资源进行消费, need 表示该消费的资源数目 */
    customing(fifo_file, need);

    if ((fd = open(fifo_file, O_RDWR)) < 0)
    {
        printf("Function myfilecopy error in source_file:");
        return -1;
    }

```

```

    }

    /* 为了模拟 FIFO 结构, 对整个文件内容进行平行移动 */
    lock_set(fd, F_WRLCK);
    myfilecopy(fifo_file, tmp_file, need, MAX_FILE_SIZE, 0);
    myfilecopy(tmp_file, fifo_file, 0, MAX_FILE_SIZE, 0);
    lock_set(fd, F_UNLCK);

    unlink(tmp_file);
    close(fd);
    return 0;
}

int main(int argc ,char *argv[])
{
    int customer_capacity = 10;

    if (argc > 1) /* 第一个参数指定需要消费的资源数目, 默认值为 10 */
    {
        sscanf(argv[1], "%d", &customer_capacity);
    }

    if (customer_capacity > 0)
    {
        custom(customer_capacity);
    }

    exit(EXIT_SUCCESS);
}

```

#### 4、实验结果

此实验的运行结果如下所示。实验结果会和这两个进程运行的具体过程相关。

终端一:

```

# ./producer 1 15 /*生产周期为 1s, 需要生产的资源数为 20 个*/
Write lock set by 11867
Release lock by 11867
.....

```

终端二:

```
# ./customer 5 /*需要消费的资源数为 5 个*/  
Enjoy: abcde  
Write lock set by 11879  
Release lock by 11879
```

## 5.7 思考练习

(1) 设计一个程序，要求打开文件“pass”，如果没有这个文件，新建此文件，权限设置为只有所有者有只读权限。

(2) 设计一个程序，要求新建一个文件“hello”，利用 write 函数将“Linux 下 C 软件设计”字符串写入该文件。

# 第 6 章进程

## 学习目标

- 理解进程概念
- 复制进程 fork
- 替换进程映像 exec
- 守护进程

## 6.1 linux 进程概述

### 6.1.1 程序与进程

程序是一个普通文件，是机器代码指令和数据的集合，这些指令和数据存储在磁盘上的一个可执行映像（Executable Image）中。

所谓可执行映像就是一个可执行文件的内容，例如，你编写了一个 C 源程序，最终这个源程序要经过编译、连接成为一个可执行文件后才能运行。

源程序中你要定义许多变量，在可执行文件中，这些变量就组成了数据段的一部分：源程序中的许多语句，例如“`i++; for(i=0; i<10; i++);`”等，在可执行文件中，它们对应着许多不同的机器代码指令，这些机器代码指令经 CPU 执行，就完成了你所期望的工作。可以这么说，程序代表你期望完成某工作的计划和步骤，它还浮在纸面上，等待具体实现。而具体的实现过程就是由进程来完成的，可以认为进程是运行中的程序，它除了包含程序中的所有内容外，还包含一些额外的数据。

我们知道，程序装入内存后才得以运行。在程序计数器的控制下，指令被不断地从内存取至 CPU 中运行。实际上，程序的执行过程可以说是一个执行环境的总和，这个执行环境包括程序中各种指令和数据外，还有一些额外数据，比如寄存器的值、用来保存临时数据（例如传递给某个函数的参数、函数的返回地址、保存的临时变量等）的堆栈、被打开的文件及输入输出设备的状态等等。上述执行环境的动态变化表征了程序的运行。为了对这个动态变化的过程进行描述，程序这个概念已经远远不够，于是就引入了“进程”概念。

进程代表程序的执行过程，它是一个动态的实体，随着程序中指令的执行而不断地变化。在某个时刻进程的内容被称为进程映像（Process Image）



Linux 是多任务操作系统，也就是说可以有多个程序同时装入内存并运行，操作系统为每个程序建立一个运行环境即创建进程。从逻辑上说，每个进程拥有它自己的虚拟 CPU。当然，实际上真正的 CPU 在各进程之间来回切换。但如果我们想研究这种系统，而去跟踪 CPU 如何在程序间来回切换将会是一件相当复杂的事情，于是换个角度，集中考虑在（伪）并行情况下运行的进程集就使问题变得简单、清晰得多。这种快速的切换称作多道程序执行。在一些 Unix 书籍中，又把“进程切换”（Process Switching）称为“环境切换”或“上下文切换”（Context Switching）。这里“进程的上下文”就是指进程的执行环境。

进程运行过程中，还需要其他的一些系统资源，例如，要用 CPU 来运行它的指令、要用系统的物理内存来容纳进程本身和它的有关数据、要在文件系统中打开和使用文件、并且可能直接或间接的使用系统的物理设备，例如打印机、扫描仪等。由于这些系统资源是由所有进程共享的，所以操作系统必须监视进程和它所拥有的系统资源，使它们可以公平地拥有系统资源以得到运行。

小知识：假设有三道程序 A、B、C 在系统中运行。程序一旦运行起来，我们就称它为进程，因此称它们为三个进程 Pa、Pb、Pc。假定进程 Pa 执行到一条输入语句，因为这时要从外设读入数据，于是进程 Pa 主动放弃 CPU。此时操作系统中的调度程序就要选择一个进程投入运行，假设选中 Pc，这就会发生进程切换，从 Pa 切换到 Pc。同理，在某个时刻可能切换到进程 Pb。从某一时间段看，三个进程在同时执行，从某一时刻看，只有一个进程在运行，我们把这几个进程的伪并行执行叫做进程的并发执行。

## 6.1.2 进程结构

在 linux 系统中，每一个进程都是拥有自己的虚拟地址空间，都运行在独立的虚拟地址空间上。这也就是说，进程间是分离的任务，拥有各自的权利和责任。在 linux 系统中运行着多个进程，其中一个进程发生异常，它不会影响到系统中的其它进程。

Linux 中的进程包括了 3 个段，分别为“数据段”、“代码段”和“堆栈段”。

数据段：存放的数据为全局变量、常数及动态数据分配的数据空间（如 malloc 函数取得的内存空间）等

代码段：存放的是程序代码数据。

堆栈段：存入的是子程序返回地址、子程序的参数以及程序的局部变量。

补充知识：

一个由 c/C++ 编译的程序占用的内存分为以下几个部分

1、栈区（stack）—— 由编译器自动分配释放，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。

2、堆区（heap）—— 一般由程序员分配释放，若程序员不释放，程序结束时可能由 OS 回收。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表。

3、全局区（静态区）（static）—— 全局变量和静态变量的存储是放在一块的，初

始化的全局变量和静态变量在一块区域，未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。（程序结束后由系统释放）

- 4、文字常量区—— 常量字符串就是放在这里的。（程序结束后由系统释放）
- 5、程序代码区—— 存放函数体的二进制代码。

见例子程序：

```
//main.cpp
int a = 0; //全局初始化区
char *p1; //全局未初始化区
main()
{
    int b; //栈
    char s[] = "abc"; //栈
    char *p2; //栈
    char *p3 = "123456"; //123456\0 在常量区，p3 在栈上。
    static int c =0; //全局（静态）初始化区
    p1 = (char *)malloc(10);
    p2 = (char *)malloc(20);
    //分配得来 10 和 20 字节的区域就在堆区。
    strcpy(p1, "123456"); /*123456\0 放在常量区，编译器可能会将它与 p3
所指向的"123456"优化成一个地方。*/
}
```

### 6.1.3 进程属性

#### 1. 进程标识

进程最主要的属性就是进程号（PID, process ID）和它的子进程号（PPID, parent process ID）。PID 和 PPID 都是非零正整数。从进程 ID 的名字就可以看出，它就是进程的身份证号码，每个人的身份证号码都不会相同，每个进程的进程 ID 也不会相同。系统调用 `getpid()` 就是获得进程标识符。

一个 PID 唯一地标识一个进程。一个进程创建一个新进程称为创建了子进程，创建子进程的进程称为父进程。

在 Linux 中获得当前进程的 PID 和 PPID 的系统调用函数为 `getpid()` 和 `getppid()`。常常在程序获得进程的 PID 和 PPID 后，可以将其写入日志文件以做备份。运用 `getpid` 和 `getppid` 获得当进程 PID 和 PPID 的例子。

```
/*pidandppid.c*/
```

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main()
{
    printf("PID = %d\n", getpid());
    printf("PPID = %d\n", getppid());
    exit(0);
}
```

通过编译后，运行程序得到以下结果（该值在不同的系统上会有不同的值）

```
[root@localhost process]# ./pidandppid
PID=18985
PPID=14481
```

进程中还有真实用户 ID (UID) 和有效的用户 ID (EUID)。进程的 UID 就是其创建者的用户标识号，或者说就是复制了父进程的 UID 值，通常，只允许创建者（也称为属主）和超级用户对进程进行操作。EUID 是“有效”的用户 ID，这是一个额外的 UID，用来确定进程在任何给定的时刻对哪些资源和文件具有访问权限。它们的函数分别是 `getuid` 和 `geteuid`。

## 2. 进程状态

为了对进程从产生到消亡的这个动态变化过程进行捕获和描述，就需要定义进程各种状态并制定相应的状态转换策略，以此来控制进程的运行。

因为不同操作系统对进程的管理方式和对进程的状态解释可以不同，所以不同操作系统中描述进程状态的数量和命名也会有所不同，但最基本的进程状态有三种：

- （1）运行态：进程占有 CPU，并在 CPU 上运行。
- （2）就绪态：进程已经具备运行条件，但由于 CPU 忙而暂时不能运行
- （3）阻塞态（或等待态）：进程因等待某种事件的发生而暂时不能运行。（即使 CPU 空闲，进程也不可运行）。

进程在生命期内处于且仅处于三种基本状态之一，如图 6-1 所示。

这三种状态之间有四种可能的转换关系：

（1）运行态→阻塞态：进程发现它不能运行下去时发生这种转换。这是因为进程发生 I/O 请求或等待某件事情。

（2）运行态→就绪态：在系统认为运行进程占用 CPU 的时间已经过长，决定让其它进程占用 CPU 时发生这种转换。这是由调度程序引起的。调度程序是操作系统的一部分，进程甚至感觉不到它的存在。

(3)就绪态→运行态: 运行进程已经用完分给它的 CPU 时间, 调度程序从处于就绪态的进程中选择一个投入运行。

(4)阻塞态→就绪态: 当一个进程等待的一个外部事件发生时(例如输入数据到达), 则发生这种转换。如果这时没有其它进程运行, 则转换③立即被触发, 该进程便开始运行。

调度程序的主要工作是决定哪个进程应当运行, 以及它应当运行多长时间。这点很重要, 我们将在后面对其进行讨论。

图 6 1

6.1.4 进程管理

1. 启动进程

进程加载有两种途径: 手工加载和调度加载

手工加载:

手工加载又分为前台加载, 和后台加载。

前台加载: 是手工加载一个进程最常用方式。一般地, 当用户输入一个命令, 如 “ls -l” 时就已经产生了一个进程, 并且是一个前台进程。

后台加载: 往往是在该进程非常耗时, 且用户也不急着需要结果的时候启动。常常用户在终端输入一个命令时同时在命令尾加上一个 “&” 符号。

调度加载:

在系统中有时要进行比较费时而且占用资源的维护工作, 并且这些工作适合在深夜而无人值守时运行, 这时用户就可以事先进行调度安排, 指定任务运行的时间或者场合, 到时系统就会自动完成一切任务。

2. 调度进程

调度进程包括对进程的中断操作, 改变优先级, 查看进程状态等, linux 中常见的调度进程的系统命令。如表 6 1 所示:

表 6 1 常见的调度进程的系统命令

选 项	参 数 含 义
ps	查看系统中的进程
top	动态显示系统中的进程
nice	按用户指定的优先级运行
renice	改变正在运行进程的优先级

选 项	参 数 含 义
kill	终止进程（包括后台进程）
crontab	用于安装、删除或者列出用于驱动 cron 后台进程的任务。
bg	将挂起的进程放到后台执行

### 6.1.5 进程模式

在 Linux 系统中，进程的执行模式划分为用户模式和内核模式。如果当前运行的是用户程序、应用程序或内核之外的系统程序，那么对应进程就在用户模式下运行：如果在用户程序执行过程中出现系统调用或发生中断事件，那么就要运行操作系统程序，进程模式就变成内核模式。

用户进程既可以在用户模式下运行，也可以在内核模式下运行。如下图所示：

## 6.2 linux 进程控制

在 Linux 系统中，常用于进程控制的函数有 `fork()` 函数、`exec()` 函数族、`exit()` 和 `wait()` 函数等。

### 6.2.1 fork 函数

`fork` 函数和一般的函数有着很大区别，`fork` 函数执行一次却返回两个值。

#### 1. fork 函数说明

在进程中使用 `fork` 函数，则会创建一个新进程，新进程则称为子进程，原进程称为父进程。由于 `fork` 函数返回两个值，则这两个进程分别带回它们各自的返回值，其中父进程的返回值是子进程的进程号，而子进程则返回 0。因此，可以通过返回值来判定该进程是父进程还是子进程。

使用 `fork` 函数得到的子进程是父进程的一个复制品，它从父进程处继承了整个进程的地址空间，包括进程上下文、进程堆栈、内存信息、打开的文件描述符、信号控制设定、进程优先级、进程组号、当前工作目录、根目录、资源限制、控制终端等，而子进程所独有的只有它的进程号、资源使用和计时器等。因此可以看出，使用 `fork` 函数的代

价是很大的，它复制了父进程中的代码段、数据段和堆栈段里的大部分内容，使得 fork 函数的执行速度并不很快。

2. fork 函数语法

Fork（）函数语法要点如表 6 2 所示

表 6 2 Fork（）函数语法要点

所需头文件	#include <sys/types.h> //提供类型 pid_t 的定义 #include <unistd.h>
函数原型	pid_t fork(void)
函数返回值	0: 子进程
	子进程 ID (大于 0 的整数): 父进程
	-1: 出错

3. fork 函数实例

```
/*fork.c*/
#include<sys/types.h>
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
int main()
{
pid_t result;
result=fork();
if(result==-1)
{
perror("fork error");
}
else if(result==0)
{
printf("current value is %d In child process,child PID = %d\n",result,getpid());
}
else
{
printf("current value is %d In father process,father
```

```
PID=%d\n",result,getpid());
}
}
```

编译:

```
[root@localhost fork]# gcc -o fork fork.c
```

运行的结果:

```
[root@localhost fork]# ./fork
current value is 0 In child process,child PID = 21273
current value is 21273 In father process,father PID=21272
```

从结果可以看出，子进程返回值等于 0, 而父进程返回子进程的进程号 (>0) .

6.2.2 函数族

1. exec 函数族说明

fork() 函数是用于创建一个子进程，该子进程几乎拷贝了父进程的全部内容。那在新进程中如何运行新的程序呢？exec 函数簇提供了一个在进程中启动另一个程序执行的方法。

exec 函数簇可以根据指定的文件名或目录找到可执行文件，并用它来取代原调用进程的数据段、代码段和堆栈段，在执行完之后，原调用进程的内容除了进程号外，其他全部被新进程替换了。另外，这里的可执行文件既可以是二进制文件，也可以是 Linux 下任何可执行的脚本文件。

2. exec 函数族语法

在 linux 中并没有 exec 函数，而是有 6 个 exec 开头的函数族，它们之音语法有细微差别，本书在下面会详细讲解。

表 6 3 exec ( ) 函数族成员函数语法

所需头文件	#include <unistd.h>
函数原型	int execl(const char *path, const char *arg, ...)
	int execv(const char *path, char *const argv[])
	int execlp(const char *path, const char *arg, ..., char *const envp[])
	int execve(const char *path, char *const argv[], char *const envp[])
	int execlp(const char *file, const char *arg, ...)
	int execvp(const char *file, char *const argv[])

函数返回值	-1: 出错
-------	--------

下表 7.4 再对这几个函数中函数名和对应语法做一总结，主要指出了函数名中每一位所表明的含义，希望读者结合此表加以记忆。

表 6 4    exec ( ) 函数名对应含义

前 4 位统一为	exec	
第 5 位	l: 参数传递为逐个列举方式	execl、execle、execlp
	v: 参数传递为构造指针数组方式	execv、execve、execvp
第 6 位	e: 可传递新进程环境变量	execle、execve
	p: 可执行文件查找方式为文件名	execlp、execvp

3.   exec 函数族实例

l   execlp

用 execlp 函数作实例，说明如何使用文件名的方式来查找可执行文件，同时使用参数列表的方式。

```
/*execlp.c*/
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
int main()
{
    pid_t result;
    result=fork();
    if(result==0)
    {
        if(execlp("ls","ls","-l",NULL)<0)
        {
            perror("execlp error");
        }
    }
}
```

在该程序中，首先使用创建一个子进程，然后在程序里使用 execlp 函数。这里 execlp 使用文件名的方式进行查找，系统会在默认的环境变量 PATH 中寻找该可执行文件，参数列表是在 shell 中使用的命令和选项。参数列表最后一项应为 NULL. 运行程序后，运行结果为列出当前目录下所有文件，不同的目录下有不同的结果。例如：



```
[root@localhost exec]# ./execlp
总计 12
-rwxr-xr-x  1  root  root 4954 07-05 09:48  execlp
-rw-r-r--  1  root  root 207  07-05 09:48  execlp.c
```

2 用 execl 函数作实例,说明如何使用完整的文件目录来查找对应的可执行文件(注意目录必须以 “/” 开头,否则将其视为文件名),同时使用参数列表的方式。

```
/*execl.c*/
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
int main()
{
    pid_t result;
    result=fork();
    if(result==0)
    {
        if(execl("/bin/ls","ls","-l",NULL)<0)
        {
            perror("execlp error");
        }
    }
}
```

编译后,运行所得结果和运行 exelp 函数的例子一样。

### 3 execv

参数传递为构造指针数组方式。

```
/*execv.c*/
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
int main()
{
    pid_t result;
    char *arg[]={"ls","-l",NULL};
    result=fork();
```

```

if(result==0)
{
if(execv("/bin/ls",arg)<0)
{
perror("execlp error");
}
}
}

```

运行结果为列出当前目录下的所有文件。

#### 4 execve

e 为设置环境变量。

```

/*execv.c*/
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
int main()
{
pid_t result;
char *arg[]={ "env",NULL};
char *envp[]={ "PATH=/tmp","USER=lzg",NULL};
result = fork();
if(result==0)
{
if(execve("/bin/env",arg,envp)<0)
{
Perror("execlp error");
}
}
}

```

编译运行得到结果为

```

[root@localhost exec]# ./execve
PATH=/tmp
USER=lzg

```

#### (4) 使用 exec 函数族应注意的地方

在使用 exec 函数族时，一定要加上错误判断语句。因为 exec 很容易执行失败，其中最常见的原因有：

找不到文件或路径，此时 `errno` 被设置为 `ENOENT`；  
数组 `argv` 和 `envp` 忘记用 `NULL` 结束，此时 `errno` 被设置为 `EFAULT`；  
没有对应可执行文件的运行权限，此时 `errno` 被设置为 `EACCES`。

小知识：事实上，这 6 个函数中真正的系统调用只有 `execve()`，其他 5 个都是库函数，它们最终都会调用 `execve()` 这个系统调用。

补充知识：system 函数

(1) system 函数说明

`system` 函数是一个与操作系统紧密相关的函数。用户可以使用它在自己的程序中调用系统提供的各种命令。因此，使用 `system` 函数比使用 `exec` 函数族更方便。

(2) system 函数说明

表 6 5

所需头文件	#include<stdlib.h>
函数原型	int system(const char *string)
函数返回值	执行成功则返回执行 shell 命令后的返回值，调用/bin/sh失败则返回 127, 其它失败原因则返回-1, 参数 string 为空(NULL), 则返回非零值。

(3) system 函数实例

```
/*system.c*/
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int result;
    result=system("ls -l");
    return 0;
}
```

运行结果是列出当前目录下的所有文件。在不同的目录下运行程序会有不同的结果。

### 6.2.3 exit和\_exit函数

#### 1. exit和\_exit函数说明

在系统中有大量的进程时，有可能会让系统资源消耗殆尽。因此，要在用完进程后终止进程。Linux 用到的函数为 exit 和 \_exit 函数。当程序执行 exit 和 \_exit 时，进程会条件地停止所有操作，终止本进程的运行。这两个数的区别如图 6 2 所示：

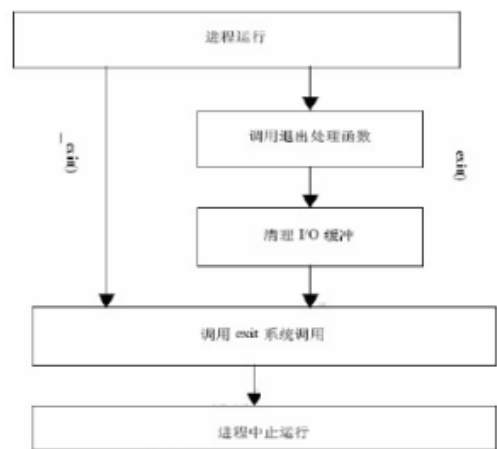


图 6 2

从图中可以看出，\_exit 函数的作用是：直接使进程停止运行，清除其使用内存空间，并清除其在内核中的各种数据结构；exit 函数则在这些基础上作了一些动作，在执行退出之前加了若干道工序。Exit 函数和 \_exit 函数最大的区别就在于 exit 函数在调用 exit 系统调用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。就是图中的“清理 I/O 缓冲”。

#### 2. exit和\_exit函数语法

表 6 6 exit 和 \_exit 函数语法

所需头文件	exit: #include <stdlib.h>
	_exit: #include <unistd.h>
函数原型	exit: void exit(int status)
	_exit: void _exit(int status)
函数传入值	status 是一个整型的参数，可以利用这个参数传递进程结束时的状态。一般来说，0 表示正常结束；其他的数值表示出现了错误，进程非正常结束。在实际编程时，可以用 wait 系统调用接收子进程的返回值，从而针对不同的情况进行不同的处理

### 3. exit 和 \_exit 函数实例

print 函数使用的是缓冲 I/O 方式，该函数在遇到“\n”换行符时自动从缓冲区中将记录读出。以下实例就是利用此性质来进行比较。

```
/*exit.c*/
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
int main()
{
    pid_t result;
    result=fork();
    if(result==1)

    {
        perror("fork fail");
        exit(0);
    }

    else if(result==0)
    {
        printf("testing _exit()\n");
        printf("this is the content in buffer");
        _exit(0);
    }

    else
    {
        printf("testing exit()\n");
        printf("this is the content in buffer");
        exit(0);
    }
}
```

运行结果为：

```
[root@localhost exit]#./exit
```

```
testing _exit()
testing exit()
this is the content in buffer
```

从实例结果来看，exit 函数前的字符串输出了两句，\_exit() 函数前的字符串输出一句。这也说明调用 exit 函数时，缓冲区的记录能正常输出；而调用 \_exit 函数时，缓冲区中的记录无法输出。

6.2.4 wait 和 waitpid 函数

1. wait 和 waitpid 函数说明

wait 函数是用于使父进程阻塞，直到一个子进程终止或者该进程接到了一个指定的信号为止。如果该父进程没有子进程或者他的子进程已经终止，则 wait 就会立即返回。

waitpid 函数的作用和 wait 一样，但它并不一定要等待第一个终止的子进程，它还有若干项，如可提供一个非阻塞版本的 wait 功能。实际上 wait 函数只是 waitpid 函数的一个特例。

2. wait 和 waitpid 函数格式说明

表 6 7 wait()函数语法

所需头文件	#include <sys/types.h> #include <sys/wait.h>
函数原型	pid_t wait(int *status)
函数传入值	这里的 status 是一个整型指针，是该子进程退出时的状态： status 若为空，则代表任意状态结束的子进程； status 若不为空，则代表指定状态结束的子进程； 另外，子进程的结束状态可由 Linux 中一些特定的宏来测定。
函数返回值	成功：已结束运行的子进程的进程号
	失败：-1

表 6 8 waitpid()函数语法

所需头文件	#include <sys/types.h> #include <sys/wait.h>
函数原型	pid_t waitpid(pid_t pid, int *status, int options)

函数传入值	pid	pid>0: 只等待进程 ID 等于 pid 的子进程, 不管已经有其他子进程运行结束退出了, 只要指定的子进程还没有结束, waitpid 就会一直等下去
		pid=1: 等待任何一个子进程退出, 此时和 wait 作用一样
		pid=0: 等待其组 ID 等于调用进程的组 ID 的任一子进程
		pid<1: 等待其组 ID 等于 pid 的绝对值的任一子进程
	status	同 wait
	options	WNOHANG: 若由 pid 指定的子进程不立即可用, 则 waitpid 不阻塞, 此时返回值为 0
		WUNTRACED: 若实现某支持作业控制, 则由 pid 指定的任一子进程状态已暂停, 且其状态自暂停以来还未报告过, 则返回其状态
		0: 同 wait, 阻塞父进程, 等待子进程退出
函数返回值	正常: 子进程的进程号 使用选项 WNOHANG 且没有子进程退出: 0 调用出错: -1	

### 3. waitpid 函数实例

wait 函数只是 waitpid 函数的一个特例, 所以这里只举使用 waitpid 函数的实例。

```
/*waitpid.c*/
#include<sys/types.h>
#include<sys/wait.h>
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
int main()
{
    pid_t result1,result2;
    result1=fork(); (创建新进程)
    if(result1<0)
        printf("fork fail\n");
    else if(result1==0)
    {
        printf("sleep 3s in child\n ");
        sleep(3); (子进程暂停 3s)
        exit(0);
    }
}
```

```

}
else
{
    /*不断地在测试子进程是否退出*/
    do
    {
        result2=waitpid(result1,NULL,WNOHANG);
        if(result2==0) (如果子进程没有退出就返回值为 0)
        {
            printf("The child process has not exited\n");
            sleep(1);
        }

        }while(result2==0);
        if(result1==result2)
        printf("The child process has exited\n");
    }

}

```

编译运行结果为

```

[root@localhost waitpid] ./waitpid
sleep 3s in child
The child process has not exited
The child process has not exited
The child process has not exited
The child process has exited

```

通过运行结果可知道父进程在没有捕获到子进程的退出信号，就会不断地循环，直到子进程退出为止，子进程不退出，waitpid 返回值为 0。



## 6.3 守护进程

### 6.3.1 守护进程概述

守护进程（Daemon）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且周期性地执行某种任务或等待处理某些发生的事件。守护进程常常在系统引导装入时启动，在系统关闭时终止。Linux 系统有很多守护进程，大多数服务都是通过守护进程实现的，同时，守护进程还能完成许多系统任务，例如，作业规划进程 `crond`、打印进程 `lpd` 等（这里的结尾字母 `d` 就是 Daemon 的意思）。

由于在 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，这个终端就称为这些进程的控制终端，当控制终端被关闭时，相应的进程都会自动关闭。但是守护进程却能够突破这种限制，它从被执行开始运转，直到整个系统关闭时才退出。如果想让某个进程不因为用户或终端或其他地变化而受到影响，那么就必须把这个进程变成一个守护进程。

### 6.3.2 编写守护进程

创建守护进程是可以按一定的流程来实现的，可分为五个步骤：

#### 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。

在 Linux 中父进程先于子进程退出会造成子进程成为孤儿进程，而每当系统发现一个孤儿进程是，就会自动由 1 号进程（`init`）收养它，这样，原先的子进程就会变成 `init` 进程的子进程。

小知识：

作为进程树如何使用的一个简单例子，让我们来看 Linux 启动时是怎样对其自己进行初始化的。Linux 在启动时就创建一个称为 `init` 的特殊进程，其进程标识符 PID 为 1，它是用户态下所有进程的祖先进程，以后诞生的所有进程都是它的子进程——或是它的儿子，或是它的孙子。1 号进程运行时查询系统当前存在的终端数，然后为每个终端创建一个新的管理进程，这些进程在终端上等待着用户的登录。当用户正确登录后，系统再为每一个用户启动一个 `shell` 进程，由 `shell` 进程等待并接受用户输入的命令信息，如图 3.1 是一颗进程树。此外，`init` 进程还负责管理系统中的“孤儿”进程。如果某个进程创建子进程之后就终止，而子进程还“活着”，则子进程成为孤儿进程。`init` 进程

负责“收养”该进程，即孤儿进程会立即成为 init 进程的子进程。这是为了保持进程树的完整性。

创建子进程, 父进程退出代码如下:

```
pid=fork();  
if(pid>0)  
exit(0);
```

## 2. 在子进程中创建新会话

这个步骤，虽然它的实现非常简单，但它的意义却非常重大，是创建守护进程中最重要的一步。在这里使用的是系统函数 `setsid`，在具体介绍 `setsid` 之前，首先要了解两个概念：进程组和会话期。

◆进程组：是一个或多个进程的集合。进程组有进程组 ID 来唯一标识。除了进程号（PID）之外，进程组 ID 也是一个进程的必备属性。每个进程组都有一个组长进程，其组长进程的进程号等于进程组 ID。且该进程组 ID 不会因组长进程的退出而受到影响。

◆会话周期：会话期是一个或多个进程组的集合。通常，一个会话开始于用户登录，终止于用户退出，在此期间该用户运行的所有进程都属于这个会话期。

接下来就可以具体介绍 `setsid` 的相关内容：

(1) `setsid` 函数作用：

`setsid` 函数用于创建一个新的会话，并担任该会话组的组长。调用 `setsid` 有下面的 3 个作用：

- ①让进程摆脱原会话的控制
- ②让进程摆脱原进程组的控制
- ③让进程摆脱原控制终端的控制

那么，在创建守护进程时为什么要调用 `setsid` 函数呢？由于创建守护进程的第一步调用了 `fork` 函数来创建子进程，再将父进程退出。由于在调用了 `fork` 函数时，子进程全盘拷贝了父进程的会话期、进程组、控制终端等，虽然父进程退出了，但会话期、进程组、控制终端等并没有改变，因此，还还不是真正意义上的独立开来，而 `setsid` 函数能够使进程完全独立出来，从而摆脱其他进程的控制。

## 3. 改变工作目录

使用 `fork` 创建的子进程继承了父进程的当前工作目录。由于在进程运行中，当前目录所在的文件系统是不能卸载的，这对以后的使用会造成诸多的麻烦（比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的问题，当然，如有特殊需要，也可以把当前工作目录换成其他的路径，如 `/tmp`。改变工作目录的常见函数是 `chdir`。

如改变当前工作目录为“/”。

```
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。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的麻烦。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)，从而不会屏蔽文件权限中的任何对应位。

#### 5. 关闭任何不需要的文件描述符

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卸下。在继承的文件描述符为 0、1、和 2 的三个文件，它们分别对应着常用的到的三个文件，分别是：输入，输出和报错。需守护进程是和终端脱离联系，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。因此至少要关闭这三个打开的文件描述符，如果还打开其它文件描述符也要关闭掉。

```
例：for(i=0;i<MAXFILE;i++)  
close(i);
```

经过上面的 5 个步骤就可以创建起了一个守护进程。如把步骤流成流程图则如下：

实例如下：守护进程每 5 秒向日志文件 dameon.log 输入日记信息。

```
#include<stdio.h>  
#include<stdlib.h>  
#include<string.h>  
#include<fcntl.h>  
#include<sys/types.h>  
#include<unistd.h>  
#include<sys/wait.h>  
  
#define MAXFILE 3  
int main()  
{  
pid_t pid;  
int fd,len,i,num;  
char *buf="the dameon is running\n";
```

```

len=strlen(buf)+1;
pid =fork();
if(pid<0)
{
printf("fork fail\n");
exit(1);
}
if(pid>0)
exit(0);
setsid();
chdir("/");
umask(0);
for(i=0;i<MAXFILE;i++)
close(i);
while(1)
{
fd=open("/var/log/dameon.log",O_CREAT|O_WRONLY|O_APPEND,0666);
write(fd,buf,len);
close(fd);
sleep(5);
}
}

```

读者在前面编写守护进程的具体调试过程中会发现，由于守护进程完全脱离了控制终端，因此，不能像其他进程的程序一样通过输出错误信息到控制终端来通知程序员即使使用 gdb 也无法正常调试。那么，守护进程的进程要如何调试呢？一种通用的办法是使用 syslog 服务，将程序中的出错信息输入到“/var/log/messages”系统日志文件中，从而可以直观地看到程序的问题所在。

注意：“/var/log/message”系统日志文件只能由拥有 root 权限的超级用户查看。

syslog 是 Linux 中的系统日志管理服务，通过守护进程 syslogd 来维护。该守护进程在启动时会读一个配置文件“/etc/syslog.conf”。该文件决定了不同种类的消息会发送向何处。例如，紧急消息可被送向系统管理员并在控制台上显示，而警告消息则可记录到一个文件中。

该机制提供了 3 个 syslog 函数，分别为 openlog、syslog 和 closelog。下面就分别介绍这 3 个函数。

#### (1) syslog 相关函数说明

通常，openlog 函数用于打开系统日志服务的一个连接；syslog 函数是用于向日志文件中写入消息，在这里可以规定消息的优先级、消息输出格式等；closelog 函数是用于关闭系统日志服务的连接。

(2) syslog 相关函数格式

表 8.1

表 6 9   openlog()函数的语法

所需头文件	#include <syslog.h>	
函数原型	void openlog (char *ident,int option ,int facility)	
函数传入值	Ident	要向每个消息加入的字符串，通常为程序的名称
	Option	LOG_CONS: 如果消息无法送到系统日志服务，则直接输出到系 统控制终端
		LOG_NDELAY: 立即打开系统日志服务的连接。在正常情况下，直到发送到第一条消息时才打开连接
		LOG_PERROR: 将消息也同时送到 stderr 上
		LOG_PID: 在每条消息中包含进程的 PID
	facility: 指定程序发送的消息类型	LOG_AUTHPRIV: 安全/授权讯息 LOG_CRON: 时间守护进程 (cron 及 at) LOG_DAEMON: 其他系统守护进程 LOG_KERN: 内核信息 LOG_LOCAL[0~7]: 保留 LOG_LPR: 行打印机子系统 LOG_MAIL: 邮件子系统 LOG_NEWS: 新闻子系统 LOG_SYSLOG: syslogd 内部所产生的信息 LOG_USER: 一般使用者等级讯息 LOG_UUCP: UUCP 子系统

表 6 10   syslog()函数语法

所需头文件	#include <syslog.h>
函数原型	void syslog( int priority, char *format, ...)

函数传入值	priority: 指定消息的重要性	LOG_EMERG: 系统无法使用 LOG_ALERT: 需要立即采取措施 LOG_CRIT: 有重要情况发生 LOG_ERR: 有错误发生 LOG_WARNING: 有警告发生 LOG_NOTICE: 正常情况, 但也是重要情况 LOG_INFO: 信息消息 LOG_DEBUG: 调试信息
	format	以字符串指针的形式表示输出的格式, 类似 printf 中的格式

表 6 11 closelog()函数语法

所需头文件	#include <syslog.h>
函数原型	void closelog( void )

(3) syslog 函数实例:

这里将上一节的例子用 syslog 服务进行重写, 其中有区别的地方用加粗的字体表示, 源代码如下所示:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<fcntl.h>
#include<sys/types.h>
#include<unistd.h>
#include<sys/wait.h>
#include<syslog.h>
#define MAXFILE 3

int main()
{
    pid_t pid,sid;
    int fd,len,i,num;
    char *buf="the dameon is running\n";
    len=strlen(buf)+1;
    pid =fork();
    if(pid<0)
    {
```

```

printf("fork fail\n");
exit(1);
}
if(pid>0)
exit(0);
openlog("dameon", LOG_PID, LOG_DAEMON);
if((sid=setsid())<0)
{
syslog(LOG_ERR, "%s\n", "stesid");
exit(1);
}
if((sid=chdir("/"))<0)
{
syslog(LOG_ERR, "%s\n", "chdir");
}
umask(0);
for(i=0; i<MAXFILE; i++)
close(i);
while(1)
{
if((fd=open("/var/log/dameon.log", O_CREAT|O_WRONLY|O_APPEND, 060
0))<0)
{
syslog(LOG_ERR, "open");
exit(1);
}
write(fd, buf, len);
close(fd);
sleep(5);
}
closelog();
exit(0);
}

```

在加入 syslog 的日志功能之后，如果以普通用户执行这程序，由于这里 open 函数必须具有 root 权限，因此，syslog 就会将错误信息定到“/var/log/messages”中，类似下面的消息。

```
Aug614:40:33localhostdameon[10026]:open
```

## 6.4 本章小结

本章主要介绍了进程的控制开发，首先给出了进程的基本概念。接下来，本章重点讲解了进程控制编程，主要讲解了 `fork()` 函数和 `exec()` 函数族。

进程是 Linux 系统的调度单位，我们看到所有的 Linux 进程，包括 `init` 在内，都使用着同样的系统调用，每个程序员都可以用它们来开发自己的程序。

最后，本章讲解了 Linux 守护进程的编写，包括守护进程的概念、编写守护进程的步骤和守护进程的出错处理。



# 第 7 章Linux 线程概述

## 学习目标

- 掌握线程概念
- 线程基本控制
- 线程同步与互斥

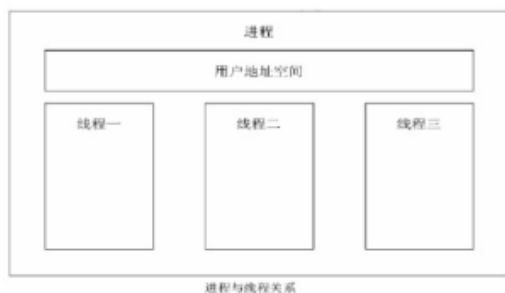
## 7.1 线程

### 7.1.1 线程概述

在 linux 系统中，当进程进行切换等操作时需要负责的上下文切换等动作，而因每一个进程都拥有自己的数据段，代码段和堆栈段，从造成进程的切换造成很大的花销。为了减少处理机的空转时间，支持多处理器和减少上下文切换开销，这样出现了一个新概念—线程。线程是一个进程内的基本调度单位，也可以称为轻量级进程，一个进程内可有多个线程。线程是在共享内存空间中并发的多道执行路径，它们共享一个进程的资源，如文件描符和信号处理。这样线程在切换时，大大减少了上下文切换的开销。

一个进程内的多线程共享一个用户地址空间。由于线程共享了进程的资源 and 地址空间，因此，任何线程对系统资源的操作都会给其他线程带来影响，这样就要实现多线程之间的同步。

进程和线程的关系如图：



---

## 7.1.2 线程发展历程

线程技术是在上世纪 60 年代被提出，而真正应用到操作系统中还是上世纪 80 年代中期。现在，多线程已经被许多操作系统所应用，其中包括 Linux 系统。

在 Linux2.2 内核中，线程也是通过 fork 创建的“轻”进程，并不存在真正意义上的线程。这也就是说，线程实际上是通过进程来模拟的，新的进程是原来进程的子进程。需要注意的是，这时线程的个数很有限，最多只能有 4096 个进程/线程同时运行。

在 Linux2.4 内核中消除了线程个数的限制，并且允许在系统运行中动态地调整进程/线程数上限，进程数的多少只受物理内存的多少影响。这时采用的是 LinuxThread 线程库，它对应的线程模型是“一对一”线程模型，也就是一个用户级线程对应一个内核线程，而线程之间的管理在内核外函数库中实现。

在 Linux2.6 内核中，为了解决以上问题，进程调度重新编写，删除了以前版本中效率不高的算法。内核线程框架也被重新编写，开始使用 NPTL (Native POSIX Thread Library) 线程库。NPTL 线程库有低启动开销、低链接开销的特性。

## 7.2 Linux 线程控制

### 7.2.1 线程基本操作

在这里我们介绍的是 Pthread 线程库，它是由 POSIX 提供的一套通用的线程库，具有很好的移植性。Pthread 线程库是一套用户级线程库，在 linux 上实现时，是使用了内核级线程来完成，目的是为了提升线程的并发性。

线程创建和退出

#### 1. 函数说明

在 linux 中，创建线程所用的函数是 pthread\_create。而创建线程实际上就是确定调用该线程函数的入口点。线程退出有两种方法：一种是在线程被创建后，就开始运行相关的线程函数，在该函数运行完之后，该线程也就退出了；另一种是使用函数 pthread\_exit 主动退出。在这里应注意到，线程退出使用函数 pthread\_exit，而进程退出是使用函数 exit，当使用函数 exit 让进程终止时，进程中所有线程都会终止。上一章说到在进程之间用函数 wait 来同步终止和释放资源，而线程之间实现这样的机制是用函数 pthread\_join。函数 pthread\_join 可用于将当前线程挂起，等待线程的结束。这个函数是一个线程阻塞函数，调用它的函数将一直到被等待的线程结束为止，当函数返回时，被等待线程的资源被回收。

#### 2. 函数格式

(1) pthread\_create 函数语法要点

pthread\_create 函数语法要点如所示

所需头文件	#include <pthread.h>
函数原型	int pthread_create ((pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg))
函数传入值	thread: 线程标识符
	attr: 线程属性设置 (具体设定在 9.2.2 会进行讲解)
	start_routine: 线程函数的起始地址
	arg: 传递给 start_routine 的参数 成
函数返回值	成功:0
	失败:-1

(2) pthread\_exit 函数的语法要点表 9.2 列出了。

所需头文件	#include <pthread.h>
函数原型	void pthread_exit(void *retval)
函数传入值	Retval: pthread_exit() 调用者线程的返回值, 可由其他函数如 pthread_join 来检索获取

(3) pthread\_join 函数的语法要点

pthread\_join 函数语法要点表 9.3

所需头文件	#include <pthread.h>
函数原型	int pthread_join ((pthread_t th, void **thread_return))
函数传入值	th: 等待线程的标识符
	thread_return: 用户定义的指针, 用来存储被等待线程的返回值 (不为 0)
函数返回值	成功: 0
	出错: -1

### 3. 函数实例

分别创建两个线程, 当线程结束时调用 pthread\_exit 函数退出, 其一个线程在运行的过程中进行 sleep。在主线程中收集这两个线程的退出信息, 并释放资源。

```
/*thread.c*/
#include<stdio.h>
#include<pthread.h>
#include<stdlib.h>
void thread1(void)
```

---

```
{
    int i=0;
    while(i<3)
    {
        printf("i= %d in pthread1\n",i);
        i++;
        sleep(5);
    }
    pthread_exit(0);
}

void thread2(void)
{
    int i=0;
    while(i<5)
    {
        printf("i= %d in pthread2\n",i);
        i++;
    }
    pthread_exit(0);
}

int main(void)
{
    pthread_t thrd1,thrd2;
    int ret;
    ret=pthread_create(&thrd1,NULL,(void *)thread1,NULL);
    if(ret=0)
    {
        printf("create thread1 fail\n");
        exit(1);
    }
    ret=pthread_create(&thrd2,NULL,(void *)thread2,NULL);
    if(ret=0)
    {
        printf("create thread2 fail\n");
        exit(1);
    }
    pthread_join(thrd1,NULL);
    pthread_join(thrd2,NULL);
}
```

```
        exit(0);  
    }
```

运行结果为:

```
[root@localhost thread]# ./thread  
i= 0 in pthread1  
i= 0 in pthread2  
i= 1 in pthread2  
i= 2 in pthread2  
i= 3 in pthread2  
i= 4 in pthread2  
i= 1 in pthread1  
i= 2 in pthread1
```

## 7.2.2 线程属性

### 1. 函数说明

什么是线程属性? 线程属性控制着一个线程在它整个生命周期里的行为。在创建线程函数 `pthread_create` 函数的第二个参数就是设置线程属性。线程属性包括绑定属性、分离属性、堆栈地址、堆栈大小、优先级。当 `pthread_create` 函数的第二个参数设置为 `NULL` 时, 就是采用默认属性, 默认属性为非绑定, 非分离、缺省 1M 的堆栈, 与父进程同样级别的优先级。

在这里重点讲解绑定属性和分离属性的基本概念

**绑定属性:** 线程可分为用户级线程和核心级线程两种, 而绑定属性正是设置用户级线程和核心级线程之间的关系。绑定属性分为两种情况: 绑定和非绑定。在绑定属性下, 一个用户线程固守分配给一个内核线程, 因为 `cpu` 时间片的调度是面向内核线程的, 因此具有绑定属性的线程可以保证在需要的时候总有一个内核线程与之对应。在非绑定属性下, 用户线程和内核线程的关系不是始终固定的, 而是由系统来根据实际情况分配。

**分离属性:** 分离属性是用来决定一个线程以什么样的方式来终止自己。分离属性分别两种情况: ①在非分离情况下, 当线程结束时, 它所占用的系统资源并没有立即释放。只有当 `pthread_join()` 函数返回时, 创建的线程才会释放自己占用的系统资源。②在分离属性情况下, 线程结束时立即释放它所占有的系统资源。在这里特别需要注意的是: 如果设置一个线程的分离属性, 而这个线程运行又非常快, 那么它很可能在 `pthread_create` 函数返回之前就终止了, 它终止以后就可能将线程号和系统资源移交给其他的线程使用, 这时调用 `pthread_create` 的线程就得到了错误的线程号。

线程属性的设置都是通过某些函数来完成的, 通常首先调用 `pthread_attr_init` 函数对线程属性进行初始化, 之后再调用相应的属性设置函数。设置绑定属性的函数为 `pthread_attr_setscope`, 设置线程分离属性的函数为 `pthread_attr_setdetachstate`,

设置线程优先级的相关函数为 pthread\_attr\_getschedparam（获取线程优先级）和 pthread\_attr\_setschedparam（设置线程优先级）。在设置完相关属性后，就可以通过函数 pthread\_create 来创建线程了。

2. 函数格式

(1) pthread\_attr\_init 函数的语法要点  
pthread\_attr\_init 函数语法要点，如所示。

所需头文件	#include <pthread.h>
函数原型	int pthread_attr_init(pthread_attr_t *attr)
函数传入值	attr: 线程属性
函数返回值	成功: 0
	出错: -1

(2) pthread\_attr\_setscope 函数的语法要点  
pthread\_attr\_setscope 函数语法要点如所示。

所需头文件	#include <pthread.h>	
函数原型	int pthread_attr_setscope(pthread_attr_t *attr, int scope)	
函数传入值	attr: 线程属性	
	scope	PTHREAD_SCOPE_SYSTEM: 绑定
		PTHREAD_SCOPE_PROCESS: 非绑定
函数返回值	成功: 0	
	出错: -1	

(3) pthread\_attr\_setdetachstate 函数的语法要点。  
pthread\_attr\_setdetachstate 函数语法要点

所需头文件	#include <pthread.h>		
函数原型	int pthread_attr_setscope(pthread_attr_t *attr, int detachstate)		
函数传入值	attr: 线程属性		
	detachstate	PTHREAD_CREATE_DETACHED: 分离	
		PTHREAD_CREATE_JOINABLE: 非分离	
函数返回值	成功: 0		
	出错: -1		

(4) pthread\_attr\_getschedparam 函数的语法要点。  
pthread\_attr\_getschedparam 函数语法要点

所需头文件	#include <pthread.h>
函数原型	int pthread_attr_getschedparam (pthread_attr_t *attr, struct sched_param *param)
函数传入值	attr: 线程属性
	param: 线程优先级
函数返回值	成功: 0
	出错: -1

(5) pthread\_attr\_setschedparam 函数的语法要点。

pthread\_attr\_setschedparam 函数语法要点

所需头文件	#include <pthread.h>
函数原型	int pthread_attr_setschedparam (pthread_attr_t *attr, struct sched_param *param)
函数传入值	attr: 线程属性
	param: 线程优先级
函数返回值	成功: 0
	出错: -1

### 3. 函数实例

(1) 实例内容

创建两线程，第一个线程时设置为绑定、分离属性，第二个线程为默认属性。

(2) 实例流程图

(3) 实例代码

```
/*threadattr.c*/
#include<stdio.h>
#include<pthread.h>
#include<stdlib.h>
void thread1(void)
{
    int i=0;
    while(i<5)
    {
        printf("i= %d in pthread1\n",i);
        i++;
    }
}
```

---

```
        if(i==2)
            break;
        sleep(1);
    }
    pthread_exit(0);
}
void thread2(void)
{
    int i=0;
    while(i<10)
    {
        printf("i= %d in pthread2\n",i);
        i++;
        sleep(1);
    }
    pthread_exit(0);
}
int main(void)
{
    pthread_t thrd1,thrd2;
    int ret;
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    pthread_attr_setscope(&attr,PTHREAD_SCOPE_SYSTEM);
pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
    ret=pthread_create(&thrd1,&attr,(void *)thread1,NULL);
    if(ret=0)
    {
        printf("create thread1 fail\n");
        exit(1);
    }
    ret=pthread_create(&thrd2,NULL,(void *)thread2,NULL);
    if(ret=0)
    {
        printf("create thread2 fail\n");
        exit(1);
    }
}
```



```

        pthread_join(thrd2, NULL);
        exit(0);
    }

```

编译生成可执行文件 threadattr。

首先使用 free 命令查看内存使用情况：

```

[root@localhost thread]# free

```

	total	used	free	shared	buffers	cached
Mem:	515492	476120	39372	0	174304	120756
-/+ buffers/cache:		181060	334432			
Swap:	1048568	100	1048468			

然后运行生成的可执行文件 threadattr：

```

[root@localhost thread]# ./threadattr
i= 0 in pthread1
i= 0 in pthread2
i= 1 in pthread1
i= 1 in pthread2
i= 2 in pthread2
i= 3 in pthread2
i= 4 in pthread2
i= 5 in pthread2
i= 6 in pthread2
i= 7 in pthread2
.....

```

在 threadattr 程序运行过程中，在别一终端输入 free 命令查看内存情况

```

[root@localhost thread2]# free

```

	total	used	free	shared	buffers	cached
Mem:	515492	476392	39100	0	174320	120756
-/+ buffers/cache:		181316	334176			
Swap:	1048568	100	1048468			

在程序运行完后，再输入 free 命令查看内存情况

```

[root@localhost thread]# free

```

	total	used	free	shared	buffers	cached
Mem:	515492	476120	39372	0	174304	

```
120756
-/+ buffers/cache:      181060      334432
Swap:      1048568      100      1048468
```

总结：比较程序运行过程中和运行后内存使用情况，当程序运行完后系统就回收了内存。

### 7.2.3 互斥锁

在同一进程中的线程是共享进程的资源 and 地址空间，因此对资源的访问每次只能有一个线程，这时就需要线程同步与互斥。线程间的同步（实指线程间的通信）：通常一个线程相对于另一个线程的运行速度是不确定的，这就是说线程是在异步环境下运行，每个线程都会以不可预知的速度向前运行。但是相互合作的线程需要在某些确定点上协调工作，当一个线程到达了这些点后，除非另一进程已经完成了某些操作，否则就不得不停下来等待别的线程来完成这些操作，这就是线程间的同步。线程间的互斥（实指对共享资源约束访问）：在多线程环境中，各线程可以共享各类资源，但有些资源一次只能允许一个线程使用，这种资源称“临界资源”，这时就需要线程间的互斥了。而在 POSIX 中，实现线程同步的方法有互斥锁（mutex）和信号量。

#### 1. mutex 互斥锁

互斥锁主要作用是用来保护由多个线程共享的数据和结构不被同时修改。互斥锁只有两种状态：上锁和解锁。在同一时刻只能有一个线程拥有某个互斥上的锁，拥有上锁的线程能够对共享资源进行操作。若其它线程希望上锁一个已经上锁了的互斥锁，则该线程就会挂起，直到上锁的线程释放掉互斥锁为止。其中，互斥锁分为快速互斥锁、递归互斥锁和检错互斥锁。快速互斥锁是指调用线程会阻塞直至拥有互斥锁为止；递归互斥锁能够成功地返回并且增加调用线程在互斥上加锁的次数，而检错互斥锁则为快速互斥锁的非阻塞版本，它会立即返回一个错误信息。

#### 2. 数说明

使用互斥锁前首先得初始化，定义所创建的互斥锁是什么类型的互斥锁，使用的函数是 `pthread_mutex_init`。对互斥锁上锁的函数是 `pthread_mutex_lock`，对互斥锁判断上锁函数是 `pthread_mutex_trylock`，对互斥锁解锁函数是 `pthread_mutex_unlock`，对互斥锁消除函数是 `pthread_mutex_destroy`。

#### 3. 函数格式

- (1) `pthread_mutex_init` 函数的语法要点

pthread\_mutex\_init 函数语法要点

所需头文件	#include <pthread.h>	
函数原型	int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr)	
函数传入值	Mutex: 互斥锁	
	Mutexattr	PTHREAD_MUTEX_INITIALIZER: 创建快速互斥锁
		PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP: 创建递归互斥锁
		PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP: 创建检错互斥锁
函数返回值	成功: 0	
	出错: -1	

(2) pthread\_mutex\_lock 等函数的语法要点

pthread\_mutex\_lock 等函数语法要点

所需头文件	#include <pthread.h>	
函数原型	int pthread_mutex_lock(pthread_mutex_t *mutex,) int pthread_mutex_trylock(pthread_mutex_t *mutex,) int pthread_mutex_unlock(pthread_mutex_t *mutex,) int pthread_mutex_destroy(pthread_mutex_t *mutex,)	
函数传入值	Mutex: 互斥锁	
函数返回值	成功: 0	
	出错: -1	

4. 函数实例

(1) 实例内容

创建两个线程，线程 1 执行计时累计，而线程 2 监视计时器的值。

(2) 实例流程图

(3) 实例代码

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
#include<errno.h>
#define INDEX 10000000
```

---

```

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
long int ticks;
time_t end_time;

void thread1(void);
void thread2(void);
int main(int argc, char *argv[])
{
    pthread_t id1, id2;
    int ret;
    end_time = time(NULL) + 10;
    pthread_mutex_init(&mutex, NULL);
    ret = pthread_create(&id1, NULL, (void *)thread1, NULL);
    if (ret != 0)
    {
        perror(" create pthread1 fail");
        exit(0);
    }
    ret = pthread_create(&id2, NULL, (void *)thread2, NULL);
    if (ret != 0)
    {
        perror(" create pthread2 fail");
        exit(0);
    }
    pthread_join(id1, NULL);
    pthread_join(id2, NULL);
    exit(0);
}

void thread1(void)
{
    long i;
    while (time(NULL) < end_time)
    {
        if (pthread_mutex_lock(&mutex) != 0)
        {
            perror("pthread_mutex_lock");
        }
    }
}

```

上锁

互斥锁

锁

```
    }
    for (i=0;i<INDEX;++i)
    {
        ++ticks;
    }
    if(pthread_mutex_unlock(&mutex) !=0)        互斥锁解
    {
        perror("pthread_mutex_unlock");
    }
    sleep(1);
}
```

}

```
void thread2(void)
```

```
{
```

```
    int nlock=0;
```

```
    int ret;
```

```
    while(time(NULL)<end_time)
```

```
    {
```

```
        sleep(3);
```

```
        ret=pthread_mutex_trylock(&mutex);        互斥锁判上
```

锁

```
        if(ret!=EBUSY)
```

```
        {
```

```
            if(ret!=0)
```

```
            {
```

```
                perror("thread_mutex_trylock");
```

```
                exit(0);
```

```
            }
```

```
            printf("thread2:got    lock    at    %ld
```

ticks\n",ticks);

```
            if(pthread_mutex_unlock(&mutex) !=0)
```

```
            {
```

```
                perror("pthread_muttex_unlock");
```

```
                exit(0);
```

```
            }
```

```
        }else{
```

```
            nlock++;
```

```
        }  
    }  
    printf("thread2 missed lock %d times\n",nolock);  
}
```

运行结果:

```
[root@localhost thread]# ./mutex  
Thread2:got lock at 30000000 ticks  
Thread2:got lock at 60000000 ticks  
Thread2:got lock at 90000000 ticks  
Thread2:got lock at 100000000 ticks  
Thread2:missed lock 0 times
```

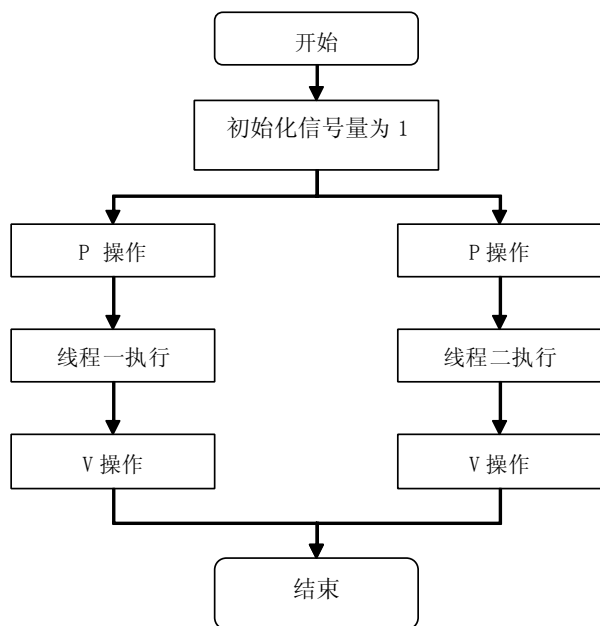
## 7.2.4 信号量

信号量是一个非负的整数计数器，是操作系统中所用到的 PV 原语，它主要应用于进程或线程间的同步与互斥。那么 PV 原语是什么呢？PV 原语的工作原理如下：

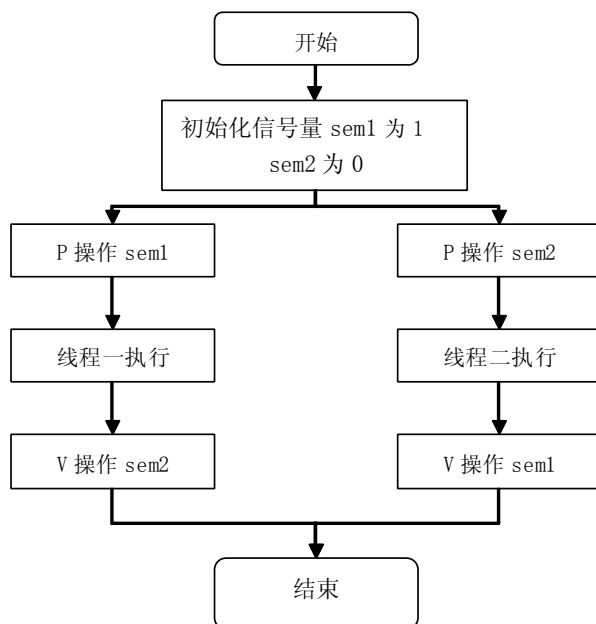
PV 原语是对整数计数器信号量 sem 的操作。一次 P 操作使 sem 减一，而一次 V 操作，使 sem 加一。线程（或进程）根据信号量的值来判断是否对公共资源具有访问权限。当信号量 sem 的值大于等于零时，该线程（或进程）具有公共资源的访问权限；相反，当信号量 sem 的值小于零时，该线程（或进程）就将阻塞直到信号量 sem 的值大于等于 0 为止。

### 1. PV 原语主要用于进程或线程间的同步和互斥这两种典型情况

（1）用于互斥，几个线程(或进程)往往只设置一个信号量 sem，它们的操作流程如所示。



(2) 当信号量用于同步操作时，往往设置多个信号量，并安排不同的初始值来实现它们之间的顺序执行，它们的操作流程如所示



## 2. 函数格式

在使用信号量时所涉及到以下函数:函数 `sem_init` 用于创建信号量,并初始化其值;函数 `sem_wait` 和函数 `sem_trywait` 相当于 P 操作,是将信号量的值减一,其区别在于若信号量小于零时,`sem_wait` 将会阻塞进程,而 `sem_trywait` 则会立即返回;函数 `sem_post` 相当于 V 操作,它将信号量的值加一同时发出信号唤醒等待的进程;函数 `sem_getvalue` 用于得到信号量的值;函数 `sem_destroy` 用于删除信号量。

(1) `sem_init` 函数的语法要点

`sem_init` 函数的语法要点

所需头文件	#include <semaphore.h>
函数原型	int sem_init(sem_t *sem, int pshared,unsigned int value)
函数传入值	sem: 信号量
	pshared: 决定信号量能否在几个进程间共享。由于目前 Linux 还没有实现进程间共享信号量,所以这个值只能取 0



	value: 信号量初始化值
函数返回值	成功: 0
	出错: -1

(2) sem\_wait 等函数的语法要点。

sem\_wait 等函数语法要点

所需头文件	#include <pthread.h>
函数原型	int sem_wait(sem_t *sem) int sem_trywait(sem_t *sem) int sem_post(sem_t *sem) int sem_getvalue(sem_t *sem) int sem_destroy(sem_t *sem)
函数传入值	sem: 信号量
函数返回值	成功: 0
	出错: -1

3. 函数实例

实例 1:

(1) 实例内容

使用信号量实现了两线程是互斥操作，也就是只使用一个信号量来实现。

(2) 实例流程图

(3) 实例代码

```
/*sem_mutex.c*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <errno.h>
#include <sys/ipc.h>
#include <semaphore.h>
int lock_var;
time_t end_time;
sem_t sem;
void pthread1(void *arg);
```

---

```
void pthread2(void *arg);
int main(int argc, char *argv[])
{
    pthread_t id1,id2;
    pthread_t mon_th_id;
    int ret;
    end_time = time(NULL)+30;
    /*初始化信号量为 1*/
    ret=sem_init(&sem,0,1);
    if(ret!=0)
    {
        perror("sem_init");
    }
    /*创建两个线程*/
    ret=pthread_create(&id1,NULL,(void *)pthread1, NULL);
    if(ret!=0)
        perror("pthread cread1");
    ret=pthread_create(&id2,NULL,(void *)pthread2, NULL);
    if(ret!=0)
        perror("pthread cread2");

    pthread_join(id1,NULL);
    pthread_join(id2,NULL);
    exit(0);
}

void pthread1(void *arg)
{
    int i;
    while(time(NULL) < end_time)
    {
        /*信号量减一，P 操作*/
        sem_wait(&sem);
        for(i=0;i<2;i++)
        {
            sleep(1);
            lock_var++;
            printf("lock_var=%d\n",lock_var);
        }
    }
}
```

```

printf("pthread1:lock_var=%d\n",lock_var);
/*信号量加一, V 操作*/
sem_post(&sem);
sleep(1);
}
}
void pthread2(void *arg)
{
int nlock=0;
int ret;
while(time(NULL) < end_time){
/*信号量减一, P 操作*/
sem_wait(&sem);
printf("pthread2:pthread1 got lock;lock_var=%d\n",lock_var);
/*信号量加一, V 操作*/
sem_post(&sem);
sleep(3);
}
}

```

程序运行结果如下所示:

```

[root@(none) tmp]# ./sem_num
lock_var=1
lock_var=2
pthread1:lock_var=2
pthread2:pthread1 got lock;lock_var=2
lock_var=3
lock_var=4
pthread1:lock_var=4
pthread2:pthread1 got lock;lock_var=4

```

## 实例 2

(1) 实例内容

通过两个信号量来实现两个线程间的同步。

(2) 实例流程图

---

### (3) 实例代码

```
/*sem_syn.c*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <errno.h>
#include <sys/ipc.h>
#include <semaphore.h>

int lock_var;
time_t end_time;
sem_t sem1, sem2;

void pthread1(void *arg);
void pthread2(void *arg);

int main(int argc, char *argv[])
{
    pthread_t id1, id2;
    pthread_t mon_th_id;
    int ret;
    end_time = time(NULL)+30;
    /*初始化两个信号量，一个信号量为 1，一个信号量为 0*/
    ret=sem_init(&sem1,0,1);
    ret=sem_init(&sem2,0,0);
    if(ret!=0)
    {
        perror("sem_init");
    }
    /*创建两个线程*/
    ret=pthread_create(&id1,NULL,(void *)pthread1, NULL);
    if(ret!=0)
        perror("pthread cread1");
    ret=pthread_create(&id2,NULL,(void *)pthread2, NULL);
    if(ret!=0)
        perror("pthread cread2");
    pthread_join(id1,NULL);
```

```

pthread_join(id2,NULL);
exit(0);
}

void pthread1(void *arg)
{
    int i;
    while(time(NULL) < end_time){
        /*P 操作信号量 2*/
        sem_wait(&sem2);
        for(i=0;i<2;i++)
        {
            sleep(1);
            lock_var++;
            printf("lock_var=%d\n",lock_var);
        }
        printf("pthread1:lock_var=%d\n",lock_var);
        /*V 操作信号量 1*/
        sem_post(&sem1);
        sleep(1);
    }
}

void pthread2(void *arg)
{
    int nlock=0;
    int ret;
    while(time(NULL) < end_time){
        /*P 操作信号量 1*/
        sem_wait(&sem1);
        printf("pthread2:pthread1 got lock;lock_var=%d\n",lock_var);
        /*V 操作信号量 2*/
        sem_post(&sem2);
        sleep(3);
    }
}

```

从以下结果中可以看出，该程序确实实现了先运行线程二，再运行线程一。

---

```
[root@(none) tmp]# ./sem_num
pthread2:pthread1 got lock;lock_var=0
lock_var=1
lock_var=2
pthread1:lock_var=2
pthread2:pthread1 got lock;lock_var=2
lock_var=3
lock_var=4
pthread1:lock_var=4
```

## 本章小结

本章首先介绍了线程和进程之间的关系，线程的发展史，接着讲解了线程的基本操作，包括线程的创建和退出；再接着说明线程属性的作用和如何使用，最后重点讲解了，线程的控制操作---互斥锁。

# 第 8 章 进程间通信

## 学习目标

- 无名管道
- 有名管道
- 共享内存
- 消息队列
- 信号量

通常情况下，程序只能访问自身的数据，和其它进程没有沟通，每个进程都是一个单独存在的个体，进程之间不需要协作就可以完成自身的任务了。但随着需要解决问题复杂性的增加，一个进程不可能完成所有的工作，必须由多个进程之间互相配合才能更快、更好、更强的解决问题。但是，处于安全性的考虑，系统会限制进程只能访问自身的数据，不能直接访问其它进程的内部数据，这时候需要有沟通的媒介；为了公平和公正，沟通的媒介的控制权不应该属于沟通的任何一方。以此推论下去，在计算机系统中，承担沟通媒介控制任务的就只有系统自身了。所以，系统要提供了沟通的媒介供进程之间“对话”使用。系统提供了多种沟通的方式，每种方式的沟通成本也不尽相同，使用成本和沟通效率也有所不同。我们经常听到的 管道、消息队列、共享内存、信号量都是系统提供的供进程之间对话的方式和媒介。

当两个进程时要进行数据共享时，这时就要进程间的通信。例如：一个进程把数据写入到通信媒介（如管道）上，别一个进程可从媒介（如管道）中拿出数据。这样，就能让两个进程进行协同工作。

## 8.1 管道

管道是 linux 系统中最古老的进程间通信手段，它是作用是把一个程序的输出直接连接到别一个程序的输入。例如在 shell 中输入命令：`ls | more` 这条命令的作用是列出当前目录下的所有文件和子目录，如果内容超过一页则自动进行分页。符号“|”就是 shell 为“ls”和“more”命令建立的一条管道，它将 ls 的输出直接送进了 more 的输入，如图 8-1 所示。

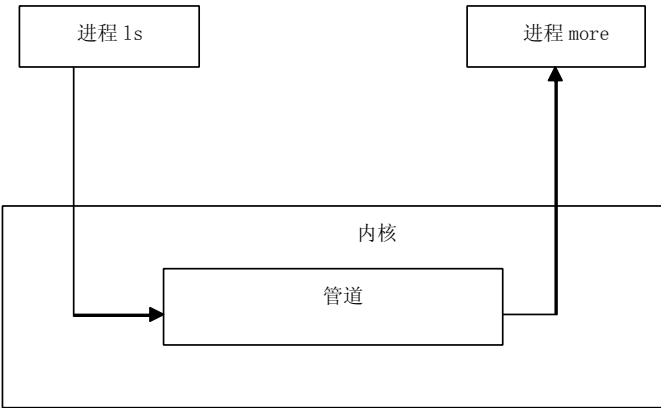


图 8 1 管道

8.1.1 无名管道

1. 无名管道的概述

平时我们所说管道，就是指无名管道，它具有以下特点：

管道是半双工的，数据只能向一个方向流动，需要双方通信时，需要建立两个管道只能用于具有亲缘关系的进程之间通信，也就是父子进程或兄弟进程之间。

对进程而言，管道就是一个文件，但它不是一个普通的文件，只存在于内存中。

一个进程向管道中写的内容被管道的另一端的进程读出。写入的内容每次都添加在管道缓冲区的末尾，并且每次都是从缓冲区的头部读出数据。

2. 无名管道的创建

无名管道的创建函数是 pipe, 它的语法要点如表 8 1 所示。

表 8 1 pipe 函数语法要点

所需头文件	#include <unistd.h>
函数原型	int pipe(int fd[2])
函数传入值	fd[2]: 管道的两个文件描述符，之后就可以直接操作这两个文件描述
函数返回值	成功: 0
	出错: 1

通过 pipe 创建管道成功则打开两个文件描述符，分别为 fd[0]和 fd[1], 其中 fd[0]用于管道读端，fd[1]用于管道写端。 管道关闭时只需将这两个文件描述符关闭即可，像关闭普通的文件描述符那样通过 close 函数分别关闭各个文件描述符。

在两个进程间使用管道进行通信时，是先在父进程使用 pipe 函数创建管道，然后再



通过 fork 函数创建子进程，这时子进程继承父进程所创建的管道。这时，父子进程管道的文件描述符对应关系如图 8-2 所示：

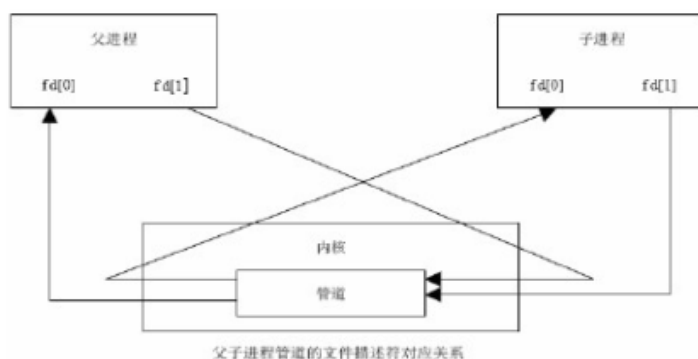


图 8-2 父子进程管道的文件描述符对应关系

这时父子进程共同使用同一管道，为了实现父子进程之间的读写，需把无关的读或写端分别进行关闭。例如实现：子进程写数据进入管道，而父进程从管道读数据。这时只需关闭父进程的 fd[1] 和子进程的读端 fd[0]，如图 8-3 所示：

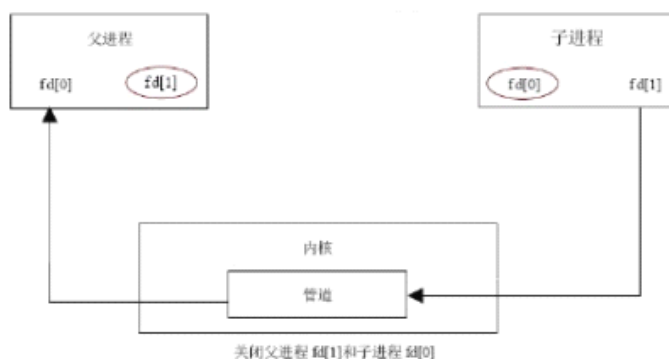


图 8-3 关闭父进程的 fd[1] 和子进程的读端 fd[0]

### 3. 无名管道创建实例

创建管道实现两进程的通信，在运行程序时，后带参数，参数内容通过一个进程写入管道，另一进程读出，打印。

```
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
```

---

```
#include <string.h>

int main(int argc, char **argv)
{
    int pipe_fd[2];
    pid_t pid;
    char buf_r[100];
    char* p_wbuf;
    int r_num, count;

    memset(buf_r, 0, sizeof(buf_r));
    if(pipe(pipe_fd) < 0)
    {
        printf("pipe create error\n");
        return -1;
    }
    if((pid=fork())==0)
    {
        close(pipe_fd[1]);
        sleep(2);
        if((r_num=read(pipe_fd[0], buf_r, 100)) > 0){
            printf("    %d numbers read from the pipe is\n", r_num);
        }
        close(pipe_fd[0]);
        exit(0);
    }
    else if(pid > 0)
    {
        close(pipe_fd[0]);

        if(write(pipe_fd[1], argv[1], strlen(argv[1])) != -1)
            printf("parent write success!\n");
        close(pipe_fd[1]);
        sleep(3);
        waitpid(pid, NULL, 0);
        exit(0);
    }
}
```

```
}
```

编译后，运行结果：

```
[root@localhost class]# ./pipe_rw hello
parent write success!
5 numbers read from the pipe is "hello"
```

### 8.1.2 有名管道

#### 1. 有名管道概述

有名管道又称 FIFO。前面所讲的无名管道，只能用于具有亲缘关系的进程间通信，在有名管道提出后，该限制得到了克服，它可以使互不相关的两个进程实现彼此的通信。FIFO 不同于无名管道，它提供了一个路径与其关联，以文件形式存在于文件系统中。在建立有名管道后，就可以把它当作普通文件来进行读写操作。值得注意的是，FIFO 严格遵循先进先出。管道不支持如 lseek() 等文件定位操作。

#### 2. 有名管道创建

创建有名管道的创建使用函数 mkfifo(), 该函数类似文件中的 open() 操作形式，可以指定管道的路径和打开模式。函数 mkfifo 语法如表 8-2 所示：

表 8-2 mkfifo 函数语法要点

所需头文件	#include <sys/types.h> #include <sys/stat.h>	
函数原型	int mkfifo(const char *filename, mode_t mode)	
函数传入值	filename: 要创建的管道	
函数传入值	mode:	O_RDONLY: 读管道
		O_WRONLY: 写管道
		O_RDWR: 读写管道
		O_NONBLOCK: 非阻塞
		O_CREAT: 如果该文件不存在，那么就创建一个新的文件，并用第三的参数为其设置权限
		O_EXCL: 如果使用 O_CREAT 时文件存在，那么可返回错误消息。 这一参数可测试文件是否存在
函数返回值	成功：0	
	出错：-1	

通过 mkfifo 创建管道成功后，就可以使用 open, read, write 这些函数。在这里要注

---

意一点，就是对普通文件进行读写时，不会出现阻塞问题，而读写管道就有阻塞的可能。这时如果需要读写非阻塞，那么在 open 函数中设定为 O\_NONBLOCK。

对于管道阻塞打开和非阻塞打开，读写进程应注意的问题

**读进程：**

- 若该管道是阻塞打开，且当前 FIFO 内没有数据，则对读进程而言将一直阻塞直到有数据写入
- 若该管道是非阻塞打开，则不论 FIFO 内是否有数据，读进程都会立即执行读操作。

**写进程：**

- 若该管道是阻塞打开，则对写进程而言将一直阻塞到有读进程读出数据。
- 若该管道是非阻塞打开，则当前 FIFO 内没有读操作，写进程都会立即执行读操作。

### 3. 有名管道使用实例

分别写两个程序文件(.C文件,都带有 main() 函数),一个实现往管道中 read 数据。一个实现写入数据。Read 文件创建管道,并不断地 printf("read %s from FIFO\n",buf\_r);(buf\_r 为收到的数据,没有数据则显示为空)。write 文件是在运行时,把所带的参数写入管道(如:./write aaaa 则把 aaaa 写入到管道中)。

管道读程序如下:

```
/*fifo_read.c*/
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define FIFO "/tmp/myfifo"

main(int argc,char** argv)
{
    char buf_r[100];
    int fd;
    int nread;

    if((mkfifo(FIFO,O_CREAT|O_EXCL)<0) && (errno!=EEXIST))
        printf("cannot create fifoserver\n");

    printf("Preparing for reading bytes...\n");
```

```

memset(buf_r,0,sizeof(buf_r));

fd=open(FIFO,O_RDONLY|O_NONBLOCK,0);
if(fd==-1)
{
    perror("open");
    exit(1);
}
while(1)
{
    memset(buf_r,0,sizeof(buf_r));
    if((nread=read(fd,buf_r,100))==-1){
        if(errno==EAGAIN)
            printf("no data yet\n");
    }
    printf("read %s from FIFO\n",buf_r);
    sleep(1);
}
pause();
unlink(FIFO);
}

```

管道写程序如下:

```

/*fifo_write.c*/
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define FIFO_SERVER "/tmp/myfifo"

main(int argc, char** argv)
{
    int fd;
    char w_buf[100];
    int nwrite;

```

---

```

        if(fd==-1)
            if(errno==ENXIO)
                printf("open      error;      no      reading
process\n");

        fd=open(FIFO_SERVER,O_WRONLY|O_NONBLOCK,0);

        if(argc==1)
            printf("Please send something\n");
        else
            {
                strcpy(w_buf,argv[1]);
                if((nwrite=write(fd,w_buf,100))==-1)
                {
                    if(errno==EAGAIN)
                        printf("The FIFO has not been
read yet.Please try later\n");
                }
                else
                    printf("write      %s      to      the
FIFO\n",w_buf);
            }
    }

```

首先运行 read 程序，再运行 write 程序。

得到结果是：

read 终端：

```

[root@localhost class]# ./fifo_read
Preparing for reading bytes....
read from FIFO
read from FIFO
read from FIFO
read aaaa from FIFO
read from FIFO
read from FIFO

```

wirte 终端：

```

[root@localhost class]# ./fifo_write aaaa
Write aaaa to the FIFO

```

## 8.2 共享内存

### 8.2.1 共享内存概述

共享内存可以说是最有用的最有用的进程间通信方式。不同进程共享内存的意思是：同一块物理内存被映射到进程各自的进程地址空间，不同进程可以及时看到某进程对共享内存的数据进行更新。采用内存共享通信的显而易见的好处是效率高，进程可直接读写内存，不需要任何数据的复制。当多个进程共享一段内存时，这时就需要某种同步机制了，如前而所说的互斥锁。

共享内存和进程间的关系图如图 8 4 所示：

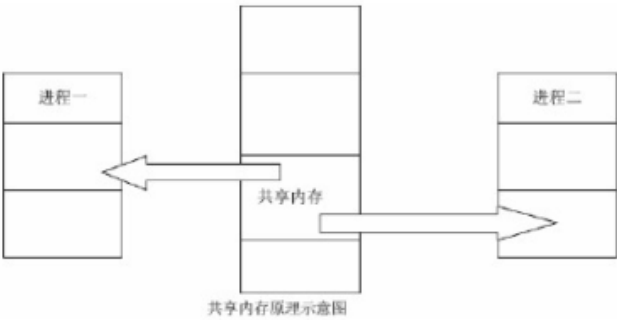


图 8 4 共享内存原理示意图

### 8.2.2 共享内存应用

#### 1. 共享内存相关函数

使用共享内存存在进程间通信，首先得创建共享内存，这里使用函数是 `shmget` (`shmget` 返回相应的标识符)，然后调用 `shmat` () 完成共享内存区域映射到进程地址空间。这时候进程就可以对共享内存进行操作了。操作完成后可以撤销映射，通过函数 `shmdt` 实现。

共享内存相关函数语法如表 8 3、表 8 4、表 8 5 所示：

表 8 3 `shmget` 函数的语法要点

所需头文件	<code>#include &lt;sys/types.h&gt;</code> <code>#include &lt;sys/ipc.h&gt;</code>
-------	--

	#include <sys/shm.h>
函数原型	int shmget(key_t key, int size, int shmflg)
函数传入值	Key: IPC_PRIVATE
	Size: 共享内存区大小
	Shmflg: 同 open 函数的权限位，也可以用八进制表示法
函数返回值	成功: 共享内存段标识符
	出错: -1

表 8 4 shmat 函数的语法要点

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>	
函数原型	char *shmat(int shmid, const void *shmaddr, int shmflg)	
函数传入值	shmid: 要映射的共享内存区标识符	
	shmaddr: 将共享内存映射到指定位置（若为 0 则表示把该段共享内存映射到调用进程的地址空间）	
	Shmflg	SHM_RDONLY: 共享内存只读
		默认 0: 共享内存可读写
函数返回值	成功: 被映射的段地址	
	出错: -1	

表 8 5 shmdt 函数语法要点

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>	
函数原型	int shmdt(const void *shmaddr)	
函数传入值	Shmaddr: 被映射的共享内存段地址	
函数返回值	成功: 0	
	出错: -1	

2. 共享内存实例

分别写两个程序，一程序把数据写入共享内存，一程序从共享内存拿数据。

程序一：

```
/*shm_write.c*?
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/types.h>
#include<unistd.h>
```



```

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

typedef struct
{
    char name[5];
    int age;
}people;

main(int argc,char **argv)
{
    int shm_id,i;
    char temp;
    people *p_map;
    char* shmname="/dev/shm/myshm";
    key_t key=ftok(shmname,0);
    shm_id=shmget(key,4096,IPC_CREAT);
    if(shm_id== -1)
        perror("error shmget");
        return;
    }
    p_map=(people*) shmat(shm_id,NULL,0);
    temp='a';
    for(i=0;i<8;i++)
    {
        temp+=1;
        memcpy((*(p_map+i)).name,&temp,1);
        (*(p_map+i)).age=18+i;
    }
    if(shmdt(p_map)==-1)
        perror("error shmdt");
}

```

程序二:

```

/*shm_read.c*/
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/types.h>
#include<unistd.h>

```

```

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

typedef struct
{
    char name[5];
    int age;
}people;

main(int argc,char **argv)
{
    int shm_id,i;
    char temp;
    people *p_map;
    char* shmname="/dev/shm/myshm";
    key_t key=ftok(shmname,0);
    shm_id=shmget(key,4096,IPC_CREAT);
    if(shm_id== -1)
        perror("error shmget");
        return;
    }
    p_map=(people *)shmat(shm_id,NULL,0);
    temp='a';
    for(i=0;i<8;i++)
    {
        printf("name:%s                                age:%d\n",(* (p_map+i)).name, (* (p_map+i)).age);
    }
    if(shmdt(p_map)==-1)
        perror("error shmdt");
}

```

编译后，首先运行程序一，再运行程序二。结果如下：

```

[root@localhost shm]# ./shm_write
[root@localhost shm]# ./shm_rread
name:b   age:18
name:c   age:19
name:d   age:20
name:e   age:21

```

```
name:f    age:22
name:g    age:23
name:h    age:24
name:i    age:25
```

#### 注意事项

共享内存相比其他几种方式有着更方便的数据控制能力，数据在读写过程中会更透明。当成功导入一块共享内存后，它只是相当于一个字符串指针来指向一块内存，在当前进程下用户可以随意的访问。缺点是，数据写入进程或数据读出进程中，需要附加的数据结构控制。在共享内存段中都是以字符串的默认结束符为一条信息的结尾。每个进程在读写时都遵守这个规则，就不会破坏数据的完整性。

## 8.3 消息队列

### 8.3.1 消息队列概述

消息队列简称队列，其标示符为队列 ID，可以通过命令 `ipcs -q` 查看当前系统的消息队列。消息队列就是一个消息的链表。用户可以从消息队列中添加消息，读取消息等。从这点上看，消息队列具有一定的 FIFO 的特性，但是它可以实现消息的随机查询，比 FIFO 具有更大的优势。可以把消息看做一个记录，具有特定的格式以及特定的优先级。对消息队列有写权限的进程可以向其中添加消息；对消息队列有读权限的进程见分晓可以从消息队列中读走消息。

### 8.3.2 消息队列应用

#### 1. 消息队列的相关函数

消息队列的实现包括创建或打开消息队列、添加消息、读取消息和控制消息队列这四种操作。其中创建或打开消息队列使用的函数是 `msgget`，这里创建的消息队列的数量会受到系统消息队列数量的限制；添加消息使用的函数是 `msgsnd` 函数，它把消息添加到已打开的消息队列末尾；读取消息使用的函数是 `msgrcv`，它把消息从消息队列中取走，与 FIFO 不同的是，这里可以指定取走某一条消息；最后控制消息队列使用的函数是 `msgctl`，它可以完成多项功能。

消息队列相关函数的语法如表 8-6、表 8-7、表 8-8 和表 8-9 所示：

表 8 6 msgget 函数语法要点

所需头文件	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/ipc.h&gt; #include &lt;sys/shm.h&gt;</pre>
函数原型	int msgget(key_t key,int flag)
函数传入值	Key: 返回新的或已有队列的队列 ID, IPC_PRIVATE
	Flag:
函数返回值	成功: 消息队列 ID
	出错: -1

表 8 7 msgsnd 函数语法要点

所需头文件	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/ipc.h&gt; #include &lt;sys/shm.h&gt;</pre>
函数原型	int msgsnd(int msqid, const void *prt, size_t size, int flag)
函数传入值	msqid: 消息队列的队列 ID
	prt: 指向消息结构的指针。该消息结构 msgbuf 为: <pre>struct msgbuf{     long mtype;//消息类型     char mtext[1];//消息正文 }</pre>
	size: 消息的字节数, 不要以 null 结尾
	flag: IPC_NOWAIT 若消息并没有立即发送而调用进程会立即返回 0: msgsnd 调用阻塞直到条件满足为止
函数返回值	成功: 0
	出错: -1

表 8 8 msgrcv 函数语法要点

所需头文件	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/ipc.h&gt; #include &lt;sys/shm.h&gt;</pre>
函数原型	<pre>iint msgrcv(int msgid, struct msgbuf *msgp, int size, long msgtype, int flag)</pre>

函数传入值	msqid: 消息队列的队列 ID	
	msgp: 消息缓冲区	
	size: 消息的字节数, 不要以 null 结尾	
	Msgtype:	0: 接收消息队列中第一个消息
		大于 0: 接收消息队列中第一个类型为 msgtyp 的消息
		小于 0: 接收消息队列中第一个类型值不小于 msgtyp 绝对值且类型 值又最小的消息
	flag:	MSG_NOERROR: 若返回的消息比 size 字节多, 则消息就会截短到 size 字节, 且不通知消息发送进程
IPC_NOWAIT 若消息并没有立即发送而调用进程会立即返回		
0: msgsnd 调用阻塞直到条件满足为止		
函数返回值	成功: 0	
	出错: -1	

表 8 9 msgctl 函数语法要点

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>	
函数原型	int msgctl ( int msgqid, int cmd, struct msqid_ds *buf )	
函数传入值	msqid: 消息队列的队列 ID	
	cmd:	IPC_STAT: 读取消息队列的数据结构 msqid_ds, 并将其存储在 buf 指定的地址中
		IPC_SET: 设置消息队列的数据结构 msqid_ds 中的 ipc_perm 元素的值。这个值取自 buf 参数
		IPC_RMID: 从系统内核中移走消息队列
	Buf: 消息队列缓冲区	
函数返回值	成功: 0	
	出错: -1	

2. 消息队列实例

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<sys/types.h>
```

---

```
#include<sys/msg.h>
#include<sys/ipc.h>
#include<unistd.h>

struct msg_data
{
    long msg_type;
    char msg_text[512];
};

int main()
{
    int msgid;
    key_t key;
    int len;
    struct msg_data msg;
    if((key=ftok(".", 'a'))== -1)
    {
        printf("error ftok\n");
        return 1;
    }
    if((msgid=msgget(key, IPC_CREAT|0666))== -1)
    {
        printf("error msgget\n");
        return 1;
    }
    puts("please enter the message to queue:");
    if((fgets(&msg->msg_text, 512, stdin))==NULL)
    {
        puts("no message");
        return 1;
    }
    msg.msg_type=getpid();
    len=strlen(msg.msg_text);
    if(msgsnd(msgid, &msg, len, 0)<0)
    {
        printf("error msgsnd\n");
        return 1;
    }
}
```

```

        if(msggrcv(msgid, &msg, 512, 0, 0) < 0)
        {
            printf("error msggrcv\n");
            return 1;
        }
        printf("rev message is :%s", (&msg)->msg_text);
        if(msgctl(msgid, IPC_RMID, NULL) < 0)
        {
            printf("error msgctl\n");
            return 1;
        }
        return 0;
    }
}

```

编译运行结果如下：

```

[root@localhost msg]# ./msgrs
please enter the message to queue:
how are you
rev message is :how are you

```

## 8.4 信号量

### 8.4.1 信号量基本操作

信号量与其他进程间通信方式不大相同，它主要提供对进程共享资源访问控制机制。相当于内存中的标志，进程可以根据它判定是否能够访问某些共享资源，同时，进程也可以修改该标志。除了用于访问控制外，还可用于进程同步。信号量本质上是一个非负的整数计数器。

信号量同步的原理实际上就是操作系统中所用到的 PV 原语。一次 P 操作使信号量 sem 减 1，而一次 V 操作使 sem 加 1。进程（线程）根据信号量的值来判断是否对公共资源具有访问权限。当 sem 的值大于等于 0 时，该进程（或线程）具有公共资源的访问权限；相反，当 sem 的值小于 0 时，该进程（线程）就将阻塞直到 sem 的值大于等于 0 为止。

信号量有两组系统调用函数，一种叫做 System V 信号量，常用于进程的同步；另一种来源于 POSIX，常用于线程同步。在这里介绍 System V 信号量。

## 8.4.2 信号量应用实例

### 1. 信号量相关函数

System V 信号量所用到的基本系统调用有三个：函数 `semget()` 是创建一个新信号量或取得一个已有信号量的键。函数 `semop` 函数用于改变信号量的值。函数 `semctl` 允许直接控制信号量信息。

信号量相关函数的语法如表 8 10、表 8 11、表 8 12 和表 8 13 所示：

表 8 10 Semget()函数语法

所需头文件	<code>#include&lt;sys/types.h&gt;</code> <code>#include&lt;sys/ipc.h&gt;</code> <code>#include&lt;sys/sem.h&gt;</code>
函数原型	<code>int semget(key_t key,int nsems,int semflg)</code>
函数传入值	key:标识一个信号量集的表示符 nsems:指定打开或者新创建的信号量集中将包含信号量的数目 semflg 一些标志位,
返回值	成功：非零值 失败：-1

表 8 11 Semopt()函数语法

所需头文件	<code>#include&lt;sys/types.h&gt;</code> <code>#include&lt;sys/ipc.h&gt;</code> <code>#include&lt;sys/sem.h&gt;</code>
函数原型	<code>int semop(int semid,struct sembuf *sops,unsigned nsops)</code>
函数传入值	semid:信号量集 ID sops:指向数组的每一个 sembuf 结构都刻画一个在特定信号量上的操作 struct sembuf{ unsigned short     sem_num; short             sem_op; short             sem_flg; } nsops 为 sops 指向数组的大小
返回值	成功：0 失败：-1

表 8 12 Semctl()函数语法

所需头文件	<code>#include&lt;sys/types.h&gt;</code> <code>#include&lt;sys/ipc.h&gt;</code>
-------	--



	#include<sys/sem.h>
函数原型	int semctl(int semid,int semnum,int cmd,union semun arg)
函数传入值	semid:指定信号量集 cmd 指定具体的操作类型 semnum:指定对哪个信号量操作，只对几个特殊的 cmd 操作有意义 arg:用于设置或返回信号量信息 union semu{ int        val; struct semid_ds  *buf; unsigned short  *array; }
返回值	成功：与 cmd 有关 失败：-1

表 8 13 参数 cmd 的取值

IPC_STAT	获取信号量信息
IPC_SET	设置信号量信息
GETALL	返回所有信号量的值
GETNCNT	返回等待 semnum 所代表信号量的值增加的进程数，相当于目前有多少进程在等待 semnum 代表的信号量所代表的共享资源。
GETPID	返回最后一个对 semnum 所代表信号量执行 semop 操作的进程 ID,成功时返回值为 sempid
GETVAL	返回 semnum 所代表信号量的值；成功返回值为 semval
GETZCNT	返回等待 semnum 所代表信号量的值变成 0 的进程数；成功时返回值为 sem
SETALL	通过 arg.array 更新所有信号量的值。
SETVAL	设置 semnum 所代表信号量的值为 arg.val

2. 信号量实例

```
#include<unistd.h>
#include<stdlib.h>
#include<stdio.h>
#include<sys/types.h>
#include<sys/sem.h>
#include<sys/ipc.h>

union semun{
    int val;
    struct semid_ds *buf;
```

---

```
        unsigned short *array;
    };
    static int set_semvalue(void);
    static void del_semvalue(void);
    static int semaphore_p(void);
    static int semaphore_v(void);
    static int sem_id;

    int main(int argc, char *argv[])
    {
        int i;
        int pause_time;
        char op_char = '0';
        srand((unsigned int) getpid());
        sem_id = semget((key_t) 1234, 1, 0666 | IPC_CREAT);
        if (argc > 1)
        {
            if (!set_semvalue())
            {
                printf("Failed to initialize\n");
                return 1;
            }
            op_char = 'X';
            sleep(2);
        }
        for (i = 0; i < 10; i++)
        {
            if (!semaphore_p())
                return 1;
            printf("%c", op_char);
            pause_time = rand() % 3;
            sleep(pause_time);
            printf("%c", op_char);
            fflush(stdout);
            if (!semaphore_v())
                return 1;
            pause_time = rand() % 2;
```

```

        sleep(pause_time);
    }
    printf("\n%d - finished\n",getpid());
    if(argc>1)
    {
        sleep(10);
        del_semvalue();
    }
    return 0;
}

static int set_semvalue(void)
{
    union semun sem_union;
    sem_union.val =1;
    if(semctl(sem_id,0,SETVAL,sem_union)==-1) return(0);
    return (1);
}

static void del_semvalue(void)
{
    union semun sem_union;
    if(semctl(sem_id,0,IPC_RMID,sem_union)==-1)
        printf("Failed to delete semaphore\n");
}

static int semaphore_p(void)
{
    struct sembuf sem_b;
    sem_b.sem_num=0;
    sem_b.sem_op=-1;
    sem_b.sem_flg=SEM_UNDO;
    if(semop(sem_id,&sem_b,1)==-1)
    {
        printf("semaphore_p failed\n");
        return (0);
    }
    return (1);
}

static int semaphore_v(void)
{

```

---

```
    struct sembuf sem_b;
    sem_b.sem_num=0;
    sem_b.sem_op=1;
    sem_b.sem_flg=SEM_UNDO;
    if(semop(sem_id, &sem_b, 1) == -1)
    {
        printf("semaphore_v failed\n");
        return 0;
    }
    return 1;
}
```

运行结果:

```
[root@local host sem]# ./sem1 8
XXXXXXXXXXXXXXXXXXXXXXX
31864 - finished
```

## 本章小结

# 第 9 章Linux 网络编程

## 学习目标

- TCP/IP
- socket 套接字
- 网络基础编程
- 网络高级编程
- 聊天室实例

## 9.1 网络编程基础概念

### 9.1.1 TCP/IP 基本概念

TCP/IP 协议（Transmission Control Protocol/ Internet Protocol）叫做传输控制/网际协议，又叫网络通信协议。

TCP/IP 虽然叫传输控制协议（TCP）和网际协议（IP），但实际上是一组协议，它包含了上百个功能的协议，如 ICMP、RIP、TELNET、FTP、SMTP、ARP、TFTP 等，这些协议一起被称为 TCP/IP 协议。TCP/IP 协议族中一些常用协议的英文名称及含义如表 9-1 所示。

表 9-1 TCP/IP 协议族中一些常用协议

常用协议的英文名称	含义
TCP	传输控制协议
IP	网际协议
UDP	用户数据报协议
ICMP	互联网控制信息协议
SMTP	简单邮件传输协议
SNMP	简单网络管理协议
FTP	文件传输协议
ARP	地址解析协议

通俗而言：TCP 负责发现传输的问题，一有问题就发出信号，要求重新传输，直到所有数据安全正确地传输到目的地。而 IP 则是给因特网的每一台电脑规定一个地址。

## 9.1.2 IP 地址、端口与域名

### 1. IP 地址

IP 地址的作用是标识计算机的网卡地址，每一台计算机都有一个 IP 地址。在程序中是通过 IP 地址来访问一台计算机的。本节将讲述 IP 地址的一些知识。IP 地址是用来标识全球计算机地址的一种符号，就比如一个手机的号码，使用这个地址可以访问一个计算机。

IP 地址具有统一的格式。IP 地址是 32 位长度的二进制数值，存储空间是 4 个字节。例如：11000000 10101000 00000001 00000110 是一台计算机的 IP 地址，但二进制的数值是不便于记忆的，可以把每个字节用一个十进制的整数来表示，既 192.168.1.1。

在同一个网络中，IP 地址是唯一的。因为需要根据 IP 地址来访问一台计算机，所以在可以访问的范围以内，每一台计算机的 IP 地址是唯一的。在终端中输入命令 `ifconfig` 可以查看本机 IP 信息。

### 2. 端口

所谓端口，是指计算机中为了标识同一计算机中不同程序访问网络而设置的编号。每一个程序在访问网络时都会分配一个标识符，程序在访问网络或接受访问时，会用这个标识符表示这一网络数据属于这个程序。这里的端口并非网卡接线的端口，而是不同程序的逻辑编号，实际并不存在。

端口号是一个 16 位的无符号整数，对应的十进制取值范围是 0~65535。不同编号范围的端口有不同的作用。低于 256 的端口是系统保留端口号，主要用于系统进程通信。如 WWW 服务使用的是 80 号端口，FTP 服务使用的是 21 号端口。不在这一范围的端口号是自由端口号，在编程时可以调用这些端口号。

### 3. 域名

域名是用来代替 IP 地址来标识计算机的一种直观名称。如百度网站的 IP 地址是 119.75.213.50，这个 IP 没有任何逻辑含义，是不便于记忆的。所以在访问计算机时，可以用这个域名来代替 IP 地址。

小知识：可以使用 `ping` 命令来查看一个域名所对应的 IP 地址。

## 9.1.3 套接字 socket

套接字（socket）的本义是插座，在网络中用来描述计算机中不同程序与其他计算机程序的通信方式。人们常说的 socket 是一种特殊的 IO 接口，它也是一种文件描述符。Socket 是一种常用的进程间通信机制，通过它不仅能实现本机上的进程间通信，而且通过网络能够在不同机器上的进程间进行通信。

### 1. socket 定义

套接字由 3 个参数构成：IP 地址、端口号、传输层协议，以区分不同应用程序进程间的网络通信与连接。

在 Linux 中的网络编程是通过 socket 接口来进行的。人们常说的 socket 接口是一种特殊的 I/O，它也是一种文件描述符。每一个 socket 都用一个半相关描述{协议，本地地址、本地端口}来表示；一个完整的套接字则用一个相关描述{协议，本地地址、本地端口、远程地址、远程端口}。socket 也有一个类似于打开文件的函数调用，该函数返回一个整型的 socket 描述符，随后的连接建立、数据传输等操作都是通过 socket 来实现的。

## 2. socket 类型

常见的 socket 有 3 种类型如下。

### (1) 流式 socket (SOCK\_STREAM)

流式套接字提供可靠的、面向连接的通信流；它使用 TCP 协议，从而保证了数据传输的正确性和顺序性。

### (2) 数据报 socket (SOCK\_DGRAM)

数据报套接字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，并且不保证是可靠、无差错的。它使用数据报协议 UDP。

### (3) 原始 socket

原始套接字允许对底层协议如 IP 或 ICMP 进行直接访问，它功能强大但使用较为不便，主要用于一些协议的开发。

## 9.1.4 套接字数据结构

C 程序进行套接字编程时，经常会使用到 sockaddr 数据类型和 sockaddr\_in 数据类型。这两种数据类型是系统中定义的结构体，用于保存套接字信息。

下面首先介绍这两个重要的数据类型：sockaddr 和 sockaddr\_in，这两个结构类型都是用来保存 socket 信息的，如下所示：

```
struct sockaddr {
    unsigned short sa_family; /*地址族*/
    char sa_data[14];
    /*14 字节的协议地址，包含该 socket 的 IP 地址和端口号。*/
};

struct sockaddr_in {
    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*/
};
```

这两个数据类型是等效的，可以相互转化，通常 `sockaddr_in` 数据类型使用更为方便。在建立 `socketadd` 或 `sockaddr_in` 后，就可以对该 `socket` 进行适当的操作了。

`sa_family` 字段可选的常见值如表 9 2 所示

表 9 2 sa\_family 字段可选的常见值

结构定义头文件	#include <netinet/in.h>
Sa_family	AF_INET: IPv4 协议
	AF_INET6: IPv6 协议
	AF_LOCAL: UNIX 域协议
	AF_LINK: 链路地址协议
	AF_KEY: 密钥套接字 (socket)

注：结构字段对了解 `sockaddr_in` 其他字段的含义非常清楚，具体的设置涉及到其他函数，在后面会有详细讲解。

9.1.5 网络相关函数

1. 域名、主机名与 IP 地址转换

(1) 函数说明

通常，人们在使用过程中都不愿意记忆冗长的 IP 地址，尤其到 IPv6 时，地址长度多达 128 位，那时就更加不可能一次次记忆那么长的 IP 地址了。因此，使用主机名将会是很好的选择。在 Linux 中，同样有一些函数可以实现主机名和地址的转化，最为常见的有 `gethostbyname`、`gethostbyaddr`、`getaddrinfo` 等，它们都可以实现 IPv4 和 IPv6 的地址和主机名之间的转化。其中 `gethostbyname` 是将主机名转化为 IP 地址，`gethostbyaddr` 则是逆操作，是将 IP 地址转化为主机名，另外 `getaddrinfo` 还能实现自动识别 IPv4 地址和 IPv6 地址。

`gethostbyname` 和 `gethostbyaddr` 都涉及到一个 `hostent` 的结构体，如下所示：

```
struct hostent{
char *h_name; /*正式主机名*/
char **h_aliases; /*主机别名*/
int h_addrtype; /*地址类型*/
int h_length; /*地址长度*/
char **h_addr_list; /*指向 IPv4 或 IPv6 的地址指针数组*/
}
```

调用该函数后就能返回 `hostent` 结构体的相关信息。



getaddrinfo 函数涉及到一个 addrinfo 的结构体，如下所示：

```
struct addrinfo{
    int ai_flags;           /*AI_PASSIVE, AI_CANONNAME;*/
    int ai_family;         /*地址族*/
    int ai_socktype;       /*socket 类型*/
    int ai_protocol;       /*协议类型*/
    size_t ai_addrlen;     /*地址长度*/
    char *ai_canoname;     /*主机名*/
    struct sockaddr *ai_addr; /*socket 结构体*/
    struct addrinfo *ai_next; /*下一个指针链表*/
}
```

相对 hostent 结构体而言，addrinfo 结构体包含更多的信息。

(2) 函数格式

gethostbyname 函数语法要点如表 9-3 所示，getaddrinfo 函数语法要点如表 9-4 所示。

表 9-3 gethostbyname 函数语法

所需头文件	#include <netdb.h>
函数原型	struct hostent *gethostbyname(const char *hostname)
函数传入值	hostname: 主机名
函数返回值	成功: hostent 类型指针 出错: -1

调用该函数时可以首先对 addrinfo 结构体中的 h\_addrtype 和 h\_length 进行设置，若为 IPv4 可设置为 AF\_INET 和 4；若为 IPv6 可设置为 AF\_INET6 和 16；若不设置则默认为 IPv4 地址类型。

表 9-4 getaddrinfo 函数语法要点

所需头文件	#include <netdb.h>
函数原型	int getaddrinfo(const char *hostname, const char *service, const struct addrinfo *hints, struct addrinfo **result)
函数传入值	hostname: 主机名
	service: 服务名或十进制的端口号字符串
	hints: 服务线索
	result: 返回结果

<b>函数返回值</b>	成功：0
	出错：-1

在调用之前，首先要对 hints 服务线索进行设置。它是一个 addrinfo 结构体，该结构体常见的选项值如表 9-5 所示。

表 9-5 addrinfo 结构体常见选项值

结构体头文件	#include <netdb.h>
ai_flags	AI_PASSIVE: 该套接口是用作被动地打开
	AI_CANONNAME: 通知 getaddrinfo 函数返回主机的名字
family	AF_INET: IPv4 协议
	AF_INET6: IPv6 协议
	AF_UNSPE: IPv4 或 IPv6 均可
ai_socktype	SOCK_STREAM: 字节流套接字 socket (TCP)
	SOCK_DGRAM: 数据报套接字 socket (UDP)
ai_protocol	IPPROTO_IP: IP 协议
	IPPROTO_IPV4: IPv4 协议
	IPPROTO_IPV6: IPv6 协议
	IPPROTO_UDP: UDP
	IPPROTO_TCP: TCP

2. 地址格式转换

(1) 函数说明

通常用户在表达地址时采用的是点分十进制表示的数值（或者是以冒号分开的十进制 IPv6 地址），而在通常使用的 socket 编程中所使用的则是二进制值，这就需要将这两个数值进行转换。这里在 IPv4 中用到的函数有 inet\_aton、inet\_addr 和 inet\_ntoa，而 IPv4 和 IPv6 兼容的函数有 inet\_pton 和 inet\_ntop。由于 IPv6 是下一代互联网的标准协议，因此，本书讲解的函数都能够同时兼容 IPv4 和 IPv6，但在具体举例时仍以 IPv4 为例。

这里 inet\_pton 函数是将点分十进制地址映射为二进制地址，而 inet\_ntop 是将二进制地址映射为点分十进制地址。

(2) 函数格式

inet\_pton 函数语法要点如表 9-6 所示，inet\_ntop 函数语法要点如表 9-7 所示。

表 9-6 inet\_pton 函数语法要点

<b>所需头文件</b>	#include <arpa/inet.h>
<b>函数原型</b>	int inet_pton(int family, const char *strptr, void *addrptr)

函数传入值	family	AF_INET (IPv4 协议)
		AF_INET6 (IPv6 协议)
	strptr:	要转化的值
	addrptr:	转化后的地址
函数返回值	成功: 0	
	出错: -1	

表 9 7 inet\_ntop 函数语法要点

所需头文件	#include <arpa/inet.h>	
函数原型	int inet_ntop(int family, void *addrptr, char *strptr, size_t len)	
函数传入值	family	AF_INET: IPv4 协议
		AF_INET6: IPv6 协议
	addrptr:	转化后的地址
	strptr:	要转化的值
	Len:	转化后值的大小
函数返回值	成功: 0	
	出错: -1	

3. 数据存储优先顺序

(1) 函数说明

计算机数据存储有两种字节优先顺序：高位字节优先和低位字节优先。Internet 上数据以高位字节优先顺序在网络上传输，因此在有些情况下，需要对这两个字节存储优先顺序进行相互转化。这里用到了四个函数：htons、ntohs、htonl、ntohl。这四个地址分别实现网络字节序和主机字节序的转化，这里的 h 代表 host，n 代表 network，s 代表 short，l 代表 long。通常 16 位的 IP 端口号用 s 代表，而 IP 地址用 l 来代表。

(2) 函数格式说明

htons 等函数语法要点如表 9 8 所示。

表 9 8 htons 等函数语法要点

所需头文件	#include <netinet/in.h>	
函数原型	uint16_t htons(uint16_t host16bit)	
	uint32_t htonl(uint32_t host32bit)	
	uint16_t ntohs(uint16_t net16bit)	
	uint32_t ntoh32(uint32_t net32bit)	
函数传入值	host16bit: 主机字节序的 16bit 数据	
	host32bit: 主机字节序的 32bit 数据	
	net16bit: 网络字节序的 16bit 数据	
	net32bit: 网络字节序的 32bit 数据	

函数返回值	成功：返回要转换的字节序
	出错：-1

调用该函数只是使其得到相应的字节序，用户不需清楚该系统的主机字节序和网络字节序是否真正相等。如果是相同不需要转换的话，该系统的这些函数会定义成空宏。

## 9.2 网络基础编程

网络基础编程主要介绍传输层中的 TCP 与 UDP 协议。TCP 和 UDP 是两种不同的网络传输方式。

### 1. TCP

#### (1) 概述

同其他任何协议栈一样，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/目的可以惟一的区分网络中两个设备的关联，通过 socket 的源/目的可以惟一的区分网络中两个应用程序的关联。

#### (2) 三次握手协议

TCP 对话通过三次握手来初始化的。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。下面描述了这三次握手的简单过程。

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。

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。

请求主机再回送一个数据段，并带有确认顺序号和确认号。

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报时，它将启动计时器。当该数据报到达目的地后，接收方的 TCP 实体向回发送一个数据报，其中包含有一个确认序号，它意思是希望收到的下一个数据报的顺序号。如果发送方的定时器在确认信息到达之前超时，那么发送方会重发该数据报。

### 2. UDP 概述

UDP 即用户数据报协议，它是一种无连接协议，因此不需要像 TCP 那样通过三次握手来建立一个连接。同时，一个 UDP 应用可同时作为应用的客户或服务器方。由于 UDP 协议并不需要建立一个明确的连接，因此建立 UDP 应用要比建立 TCP 应用简单得多。

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 的应用得到了大大的增强。它比 TCP 协议更为高效，也能更好地解决实时性的问题。如今，包括网络视频会议系统在内的众多的客户/服务器模式的网络应用都使用 UDP 协议。

9.2.1 面向连接的套接字通信（TCP）

网络上绝大多数的通信服务采用服务器机制（Client/Server），TCP 提供的是一种可靠的、面向连接的服务。下面介绍基于 TCP 协议的编程，其最主要的特点是建立完连接后才进行通信。

1. 函数说明

基于 TCP 协议编程相关函数如表 9 9 所示。

表 9 9 基于 TCP 协议编程相关函数

函 数	作 用
socket	用于建立一个 socket 连接
bind	将 socket 与本机上的一个端口绑定，随后就可以在该端口监听服务请求
connect	面向连接的客户程序使用 connect 函数来配置 socket，并与远端服务器建立一个 TCP 连接
listen	listen 函数使 socket 处于被动的监听模式，并为该 socket 建立一个输入数据队列，将达到的服务器请求保存在此队列中，直到程序处理他们
accept	accept 函数让服务器接收客户的连接请求
close	停止在该 socket 上的任何数据操作
send	数据发送函数
recv	数据接收函数

2. 函数格式

socket 函数语法要点如表 9 10 所示，bind、listen、accept、connect、send 和 recv 函数语法要点分别如表 9 11、表 9 12、表 9 13、表 9 14、表 9 15 和表 9 16 所示。

表 9 10 socket 函数语法要点

所需头文件	#include <sys/socket.h>	
函数原型	int socket(int family, int type, int protocol)	
函数传入值	family: 协议族	AF_INET: IPv4 协议
		AF_INET6: IPv6 协议
		AF_LOCAL: UNIX 域协议
		AF_ROUTE: 路由套接字（socket）

		AF_KEY: 密钥套接字 (socket)
	type: 套接字类型	SOCK_STREAM: 字节流套接字 socket
		SOCK_DGRAM: 数据报套接字 socket
		SOCK_RAW: 原始套接字 socket
	protocol: 0 (原始套接字除外)	
函数返回值	成功: 非负套接字描述符 出错: -1	

表 9 11 bind 函数语法要点

所需头文件	#include <sys/socket.h>
函数原型	int bind(int sockfd, struct sockaddr *my_addr, int addrlen)
函数传入值	sockfd: 套接字描述符
	my_addr: 本地地址
	addrlen: 地址长度
函数返回值	成功: 0 出错: -1

端口号和地址在 my\_addr 中给出了, 若不指定地址, 则内核随意分配一个临时端口给该应用程序。

表 9 12 listen 函数语法要点

所需头文件	#include <sys/socket.h>
函数原型	int listen(int sockfd, int backlog)
函数传入值	sockfd: 套接字描述符
	Backlog: 请求队列中允许的最大请求数, 大多数系统缺省值为 20
函数返回值	成功: 0 出错: -1

表 9 13 accept 函数语法要点

所需头文件	#include <sys/socket.h>
函数原型	int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)
函数传入值	sockfd: 套接字描述符
	addr: 客户端地址
	addrlen: 地址长度
函数返回值	成功: 0 出错: -1

表 9 14 connect 函数语法要点

所需头文件	#include <sys/socket.h>
-------	-------------------------

<b>函数原型</b>	int connect(int sockfd, struct sockaddr *serv_addr, int addrlen)
<b>函数传入值</b>	socktd: 套接字描述符
	serv_addr: 服务器端地址
	addrlen: 地址长度
<b>函数返回值</b>	成功: 0
	出错: -1

表 9 15 send 函数语法要点

<b>所需头文件</b>	#include <sys/socket.h>
<b>函数原型</b>	int send(int sockfd, const void *msg, int len, int flags)
<b>函数传入值</b>	socktd: 套接字描述符
	msg: 指向要发送数据的指针
	len: 数据长度
	flags: 一般为 0
<b>函数返回值</b>	成功: 0
	出错: -1

表 9 16 recv 函数语法要点

<b>所需头文件</b>	#include <sys/socket.h>
<b>函数原型</b>	int recv(int sockfd, void *buf, int len, unsigned int flags)
<b>函数传入值</b>	socktd: 套接字描述符
	buf: 存放接收数据的缓冲区
	len: 数据长度
	flags: 一般为 0
<b>函数返回值</b>	成功: 接收的字节数
	出错: -1

### 3. 3、使用实例

程序功能：该实例分为客户端和服务端，其中服务端首先建立起 socket，然后调用本地端口的绑定，接着就开始与客户端建立联系，并接收客户端发送的消息。客户端则在建立 socket 之后调用 connect 函数来建立连接。

程序说明：

程序代码：

服务器端的代码如下所示：

```
/*server.c*/
#include <sys/types.h>
#include <sys/socket.h>
```

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <netinet/in.h>

#define PORT          4321
#define BUFFER_SIZE   1024
#define MAX_QUE_CONN 5

int main()
{
    struct sockaddr_in server_sockaddr, client_sockaddr;
    int sin_size, recvbytes;
    int sockfd, client_fd;
    char buf[BUFFER_SIZE];

    /*建立 socket 连接*/
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        perror("socket");
        exit(1);
    }
    printf("Socket id = %d\n", sockfd);

    /*设置 sockaddr_in 结构体中相关参数*/
    server_sockaddr.sin_family = AF_INET;
    server_sockaddr.sin_port = htons(PORT);
    server_sockaddr.sin_addr.s_addr = INADDR_ANY;
    bzero(&(server_sockaddr.sin_zero), 8);

    int i = 1; /* 使得重复使用本地地址与套接字进行绑定 */
    setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &i, sizeof(i));

    /*绑定函数 bind*/
    if (bind(sockfd, (struct sockaddr *)&server_sockaddr,

```



```

sizeof(struct sockaddr)== -1)
{
    perror("bind");
    exit(1);
}
printf("Bind success!\n");

/*调用 listen 函数*/
if (listen(sockfd, MAX_QUE_CONN_NM) == -1)
{
    perror("listen");
    exit(1);
}
printf("Listening....\n");

/*调用 accept 函数，等待客户端的连接*/
if ((client_fd = accept(sockfd, (struct sockaddr
*)&client_sockaddr, &sin_size)) == -1)
{
    perror("accept");
    exit(1);
}

/*调用 recv 函数接收客户端的请求*/
memset(buf, 0, sizeof(buf));
if ((recvbytes = recv(client_fd, buf, BUFFER_SIZE, 0)) ==
-1)
{
    perror("recv");
    exit(1);
}
printf("Received a message: %s\n", buf);
close(sockfd);
exit(0);
}

```

客户端的代码如下所示：

```
/*client.c*/
```

```

#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <netdb.h>
#include <netinet/in.h>

#define PORT 4321
#define BUFFER_SIZE 1024

int main(int argc, char *argv[])
{
    int sockfd, sendbytes;
    char buf[BUFFER_SIZE];
    struct hostent *host;
    struct sockaddr_in serv_addr;

    if(argc < 3)
    {
        fprintf(stderr, "USAGE: ./client Hostname(or ip address)
Text\n");
        exit(1);
    }

    /*地址解析函数*/
    if ((host = gethostbyname(argv[1])) == NULL)
    {
        perror("gethostbyname");
        exit(1);
    }

    memset(buf, 0, sizeof(buf));
    sprintf(buf, "%s", argv[2]);

    /*创建 socket*/

```

```

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        perror("socket");
        exit(1);
    }

    /*设置 sockaddr_in 结构体中相关参数*/
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);
    serv_addr.sin_addr = *((struct in_addr *)host->h_addr);
    bzero(&(serv_addr.sin_zero), 8);

    /*调用 connect 函数主动发起对服务器端的连接*/
    if( connect(sockfd, (struct sockaddr *)&serv_addr,
sizeof(struct sockaddr)) == -1)
    {
        perror("connect");
        exit(1);
    }

    /*发送消息给服务器端*/
    if ((sendbytes = send(sockfd, buf, strlen(buf), 0)) == -1)
    {
        perror("send");
        exit(1);
    }
    close(sockfd);
    exit(0);
}

```

在运行时需要先启动服务器端程序，再启动客户端程序。这里可以把服务器端下载到开发板上，客户端在宿主机上运行，然后配置双方的 IP 地址，在确保双方可以通信的情况下运行程序即可。

```

# ./server
socket id = 3
Bind success!
Listening ...
Received a message: Hello!

```

```
# ./client localhost (或者 IP 地址) Hello!
```

9.2.2 无连接的套接字通信（UDP）

所谓无连接的套接字通信，指的是使用 UDP 协议进行信息传输。使用这种协议进行通信时，两个计算机之前没有建立连接的过程。需要处理的内容只是把信息发送到另一个计算机。这种通信方式比较简单。

1. 函数说明

无连接的套接字通信相关函数如表 9 17 所示

表 9 17 无连接的套接字通信相关函数

函 数	作 用
bind	将 socket 与本机上的一个端口绑定，随后就可以在该端口监听服务请求
close	停止在该 socket 上的任何数据操作
sendto	数据发送函数
recvfrom	数据接收函数

2. 函数格式

sendto 函数语法要点如表 9 18 所示，recvfrom 函数语法要点如表 9 19 所示。

表 9 18 sendto 函数语法要点

所需头文件	#include <sys/socket.h>
函数原型	int sendto(int sockfd, const void *msg,int len,unsigned int flags,const struct sockaddr *to, int tolen)
函数传入值	sockfd: 套接字描述符
	msg: 指向要发送数据的指针
	len: 数据长度
	flags: 一般为 0
	to: 目的地机的 IP 地址和端口号信息
函数返回值	tolen: 地址长度
	成功: 发送的字节数 出错: -1

表 9 19 recvfrom 函数语法要点

所需头文件	#include <sys/socket.h>
函数原型	int recvfrom(int sockfd,void *buf,int len, unsigned int flags, struct sockaddr *from, int *fromlen)

函数传入值	socktd: 套接字描述符
	buf: 存放接收数据的缓冲区
	len: 数据长度
	flags: 一般为 0
	from: 源机的 IP 地址和端口号信息
函数返回值	tolen: 地址长度
	成功: 接收的字节数 出错: -1

### 3. 使用实例

程序功能:。

程序说明:

程序代码:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<netdb.h>
#include<errno.h>
#include<sys/types.h>

int port=8888;

int main()
{
    int sockfd;
    int len;
    int z;
    char buf[256];
    struct sockaddr_in adr_inet;
    struct sockaddr_in adr_clnt;
    printf("等待客户端...\n");
```

```

/* 建立 IP 地址 */
adr_inet.sin_family=AF_INET;
adr_inet.sin_port=htons(port);
adr_inet.sin_addr.s_addr =htonl(INADDR_ANY);
bzero(&(adr_inet.sin_zero),8);
len=sizeof(adr_clnt);
/* 建立 socket */
sockfd=socket(AF_INET, SOCK_DGRAM, 0);
if(sockfd==-1)
{
    perror("socket 出错");
    exit(1);
}
/* 绑定 socket */
z=bind(sockfd, (struct      sockaddr      *)&adr_inet,
sizeof(adr_inet));
if(z==-1)
{
    perror("bind 出错");
    exit(1);
}
while(1)
{
    /* 接受传来的信息 */
    z=recvfrom(sockfd,buf,sizeof(buf),0,(struct      sockaddr
*)&adr_clnt,&len);
    if(z<0)
    {
        perror("recvfrom 出错");
        exit(1);
    }
    buf[z]=0;
    printf("接收:%s", buf);
    /* 收到 stop 字符串, 终止连接*/
    if(strncmp(buf,"stop",4)==0)
    {
        printf("结束...\n");
        break;
    }
}

```

```

    }
}
close(sockfd);
exit(0);
}

```

```

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<netdb.h>
#include<errno.h>
#include<sys/types.h>

int port=8888;

int main()
{
    int sockfd;
    int i=0;
    int z;
    char buf[80],str1[80];
    struct sockaddr_in adr_srvr;
    FILE *fp;
    printf("打开文件.....\n");
    /*以只读的方式打开 liu 文件*/
    fp=fopen("liu","r");
    if(fp==NULL)
    {
        perror("打开文件失败");
        exit(1);
    }
    printf("连接服务端...\n");
    /* 建立 IP 地址 */

```

```

    adr_srvr.sin_family=AF_INET;
    adr_srvr.sin_port=htons(port);
    adr_srvr.sin_addr.s_addr = htonl(INADDR_ANY);
    bzero(&(adr_srvr.sin_zero),8);
    sockfd=socket(AF_INET,SOCK_DGRAM,0);
    if(sockfd==-1)
    {
        perror("socket 出错");
        exit(1);
    }
    printf("发送文件 ....\n");
    /* 读取三行数据, 传给 udpserver*/
    for(i=0;i<3;i++)
    {
        fgets(str1,80,fp);
        printf("%d:%s",i,str1);
        sprintf(buf,"%d:%s",i,str1);
        z=sendto(sockfd,buf,sizeof(buf),0,(struct sockaddr
*)&adr_srvr,
        sizeof(adr_srvr));
        if(z<0)
        {
            perror("recvfrom 出错");
            exit(1);
        }
    }
    printf("发送.....\n");
    sprintf(buf,"stop\n");
    z=sendto(sockfd,buf,sizeof(buf),0,(struct sockaddr
*)&adr_srvr,
    sizeof(adr_srvr));
    if(z<0)
    {
        perror("sendto 出错");
        exit(1);
    }
    fclose(fp);
    close(sockfd);

```



```
        exit(0);
    }
```

9.3 网络高级编程

在实际情况中，人们往往遇到多个客户端连接服务器端的情况。由于之前介绍的如 connet、recv、send 都是阻塞性函数，若资源没有准备好，则调用该函数的进程将进入睡眠状态，这样就无法处理 I/O 多路复用的情况了。本节给出了两种解决 I/O 多路复用的解决方法，这两个函数都是之前学过的 fcntl 和 select（请读者先复习第 6 章中的相关内容）。可以看到，由于在 Linux 中把 socket 也作为一种特殊文件描述符，这给用户的处理带来了很大的方便。

1. 函数说明

网络高级编程相关函数如表 9 20 所示。

表 9 20 网络高级编程相关函数

函 数	作 用
fcntl	实现非阻塞 IO 或信号驱动 IO
select	实现阻塞 IO，解决 IO 多路复用

2. 函数格式

fcntl() 函数语法如表 9 21 所示。

表 9 21 fcntl()函数语法

所需头文件	#include <sys/types.h>	
	#include <unistd.h>	
	#include <fcntl.h>	
函数原型	int fcntl(int fd, int cmd, struct flock *lock)	
函数传入值	fd	文件描述符
	cmd	欲操作的类型
	lock	结构为 flock，设置记录锁的具体状态，后面会详细说明
函数返回值	0: 成功	
	-1: 出错，错误原因存于 errno	

根据 cmd 参数的不同，fcntl 函数有 5 种功能：

- (1) 复制一个现有的描述符（cmd=F\_DUPFD）。

- (2) 获得/设置文件描述符标记 (cmd=F\_GETFD 或 F\_SETFD)。
  - (3) 获得/设置文件状态标志 (cmd=F\_GETFL 或 F\_SETFL)。
  - (4) 获得/设置异步 I/O 所有权 (cmd=F\_GETOWN 或 F\_SETOWN)。
  - (5) 获得/设置记录锁 (cmd=F\_GETLK、F\_SETLK 或 F\_SETLKW)。
- cmd 参数说明如表 9 22 所示。

表 9 22 cmd 参数

参数 cmd	说 明
F_DUPFD	复制文件描述符
F_GETFD	获得 fd 的 close-on-exec 标志，若标志未设置，则文件经过 exec
F_SETFD	设置 close-on-exec 标志，该标志以参数 arg 的 FD_CLOEXEC 位决定
F_GETFL	得到 open 设置的标志
F_SETFL	改变 open 设置的标志
F_GETLK	根据 lock 描述，决定是否上文件锁
F_SETLK	设置 lock 描述的文件锁
F_SETLKW	这是 F_SETLK 的阻塞版本
F_GETOWN	检索将收到 SIGIO 和 SIGURG 信号的进程号或进程组号
F_SETOWN	设置进程号或进程组号

注意：

fcntl（）是一个非常通用的函数，它可以改变已打开的文件的性质。fcntl 有以上所述的 5 种功能，在本节中主要介绍第 3 种功能：获得/设置文件状态标志。

对于 fcntl（）函数里面的参数，在调用此函数作为不阻塞处理时，都用其固定参数 fcntl(sockfd, F\_SETFL, 0\_NONBLOCK)

select（）函数具体用法请参照第 6 章：文件 IO 操作。

下面的例子是 fcntl（）和 select（）函数在网络编程中的具体应用。

3. 使用实例

程序功能：编写一个网络聊天室。

程序说明：

程序代码：

服务器端源代码如下所示：

```
#include <stdlib.h>
#include <stdio.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h>
```

```

#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <fcntl.h>

#define MAXDATASIZE 256

#define SERVPORT 4444 /*服务器监听端口号*/
#define BACKLOG 1 /*最大同时连接请求数*/
#define STDIN 0 /*标准输入文件描述符*/

int main(void)
{
    FILE *fp; /*定义文件类型指针 fp*/
    int sockfd, client_fd; /* 监听 socket.sock_fd, 数据 传 输
socket.new_fd*/
    int sin_size;
    struct sockaddr_in my_addr, remote_addr; /*本机地址信息, 客户
地址信息*/
    char buf[256]; /*用于聊天的缓冲区*/

    char buff[256]; /*用于输入用户名的缓冲区*/
    char send_str[256]; /*最多发出的字符不能超过 256*/
    int recvbytes;
    fd_set rfd_set, wfd_set, efd_set; /*被 select() 监视的读、写、
异常处理的文件描述符集合*/
    struct timeval timeout; /*本次 select 的超时结束时间*/
    int ret; /*与 client 连接的结果*/

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) { /*
错误检测*/
        perror("socket");
        exit(1);
    }
    /* 端填充 sockaddr 结构 */
    bzero(&my_addr, sizeof(struct sockaddr_in));
    my_addr.sin_family=AF_INET; /*地址族*/
    my_addr.sin_port=htons(SERVPORT); /*端口号为 4444*/

```

```

    inet_aton("127.0.0.1", &my_addr.sin_addr);

    if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct
sockaddr)) == -1) {          /*错误检测*/
        perror("bind");
        exit(1);
    }
    if (listen(sockfd, BACKLOG) == -1) { /*错误检测*/
        perror("listen");
        exit(1);
    }

    sin_size = sizeof(struct sockaddr_in);
    if ((client_fd = accept(sockfd, (struct sockaddr
*)&remote_addr, &sin_size)) == -1) {
        /*错误检测*/
        perror("accept");
        exit(1);
    }

    fcntl(client_fd, F_SETFD, O_NONBLOCK); /* 服务器设为非阻塞*/
    recvbytes=recv(client_fd, buff, MAXDATASIZE, 0);
    /*接收从客户端传来的用户名*/
    buff[recvbytes] = '\0';
    fflush(stdout);
    /*强制立即内容*/
    if((fp=fopen("name.txt", "a+"))==NULL)
    {
        printf("can not open file,exit...\n");
        return -1;
    }
    fprintf(fp, "%s\n", buff);
    /*将用户名写入 name.txt 中*/
    while (1) {
        FD_ZERO(&rfd_set); /*将 select() 监视的读的文件描述符集合清除
*/
        FD_ZERO(&wfd_set); /*将 select() 监视的写的文件描述符集合清除
*/

```

```

        FD_ZERO(&efd_set); /*将 select() 监视的异常的文件描述符集合清除*/

        FD_SET(STDIN, &rfd_set);
        /*将标准输入文件描述符加到 select() 监视的读的文件描述符集合中*/
        FD_SET(client_fd, &rfd_set);
        /*将新建的描述符加到 select() 监视的读的文件描述符集合中*/
        FD_SET(client_fd, &wfd_set);
        /*将新建的描述符加到 select() 监视的写的文件描述符集合中*/
        FD_SET(client_fd, &efd_set);
        /*将新建的描述符加到 select() 监视的异常的文件描述符集合中*/
        timeout.tv_sec = 10; /*select 在被监视窗口等待的秒数*/
        timeout.tv_usec = 0; /*select 在被监视窗口等待的微秒数*/
        ret = select(client_fd + 1, &rfd_set, &wfd_set, &efd_set,
&timeout);
        if (ret == 0) {
            continue;
        }
        if (ret < 0) {
            perror("select error: ");
            exit(-1);
        }
        /*判断是否已将标准输入文件描述符加到 select() 监视的读的文件描述符集合中*/
        if(FD_ISSET(STDIN, &rfd_set))
        {
            fgets(send_str, 256, stdin); /*取出输入的内容*/
            send_str[strlen(send_str)-1] = '\0';
            if (strncmp("quit", send_str, 4) == 0) { /*退出程序*/
                close(client_fd);
                close(sockfd); /*关闭套接字*/
                exit(0);
            }
            send(client_fd, send_str, strlen(send_str), 0);
        }
        /*判断是否已将新建的描述符加到 select() 监视的读的文件描述符集合中*/
        if (FD_ISSET(client_fd, &rfd_set))

```

```

    {
        recvbytes=recv(client_fd, buf, MAXDATASIZE, 0);/*接收从
客户端传来的聊天内容*/
        if (recvbytes == 0) {
            close(client_fd);
            close(sockfd); /*关闭套接字*/
            exit(0);
        }
        buf[recvbytes] = '\0';
        printf("%s:%s\n",buff,buf);
        printf("Server: ");
        fflush(stdout);
    }
    /*判断是否已将新建的描述符加到 select() 监视的异常的文件描述符集合
中*/
    if (FD_ISSET(client_fd, &efd_set)) {
        close(client_fd); /*关闭套接字*/
        exit(0);
    }
}
}

```

客户端源代码如下所示:

```

#include <stdlib.h>
#include <stdio.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <fcntl.h>

#define SERVPORT 4444 /*服务器监听端口号*/
#define MAXDATASIZE 256 /*最大同时连接请求数*/
#define STDIN 0 /*标准输入文件描述符*/

int main(void)
{

```

```

int sockfd;          /*套接字描述符*/
int recvbytes;
char buf[MAXDATASIZE];    /*用于处理输入的缓冲区*/
char *str;

char name[MAXDATASIZE];    /*定义用户名*/
char send_str[MAXDATASIZE]; /* 最多发出的字符不能超过
MAXDATASIZE*/

struct sockaddr_in serv_addr; /*Internet 套接字地址结构*/
fd_set rfd_set, wfd_set, efd_set;
/*被 select() 监视的读、写、异常处理的文件描述符集合*/
struct timeval timeout; /*本次 select 的超时结束时间*/
int ret;                /*与 server 连接的结果*/
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) { /*错
误检测*/
    perror("socket");
    exit(1);
}

/* 填充 sockaddr 结构 */
bzero(&serv_addr, sizeof(struct sockaddr_in));
serv_addr.sin_family=AF_INET;
serv_addr.sin_port=htons(SERVPORT);
inet_aton("127.0.0.1", &serv_addr.sin_addr);
/*serv_addr.sin_addr.s_addr=inet_addr("192.168.0.101");*/

if (connect(sockfd, (struct sockaddr *)&serv_addr,
sizeof(struct sockaddr)) == -1)
{
    /*错误检测*/
    perror("connect");
    exit(1);
}
fcntl(sockfd, F_SETFD, O_NONBLOCK);

printf("要聊天首先输入你的名字:");

scanf("%s", name);

```

```

name[strlen(name)] = '\0';

printf("%s: ",name);

fflush(stdout);

send(sockfd, name, strlen(name), 0);
/*发送用户名到 sockfd*/

while (1)
{
    FD_ZERO(&rfd_set);/*将 select() 监视的读的文件描述符集合清除*/
    FD_ZERO(&wfd_set);/*将 select() 监视的写的文件描述符集合清除*/
    FD_ZERO(&efd_set);/*将 select() 监视的异常的文件描述符集合清除*/
    FD_SET(STDIN, &rfd_set);
    /*将标准输入文件描述符加到 selectct() 监视的读的文件描述符集合中*/
    FD_SET(sockfd, &rfd_set);
    /*将新建的描述符加到 selectct() 监视的读的文件描述符集合中*/
    FD_SET(sockfd, &efd_set);
    /*将新建的描述符加到 selectct() 监视的异常的文件描述符集合中*/
    timeout.tv_sec = 10; /*select 在被监视窗口等待的秒数*/
    timeout.tv_usec = 0; /*select 在被监视窗口等待的微秒数*/
    ret = select(sockfd + 1, &rfd_set, &wfd_set, &efd_set,
&timeout);
    if (ret == 0)
    {
        continue;
    }

    if (ret < 0)
    {
        perror("select error: ");
        exit(-1);
    }

    /*判断是否已将标准输入文件描述符加到 selectct() 监视的读的文件描述符集合
中*/
    if (FD_ISSET(STDIN, &rfd_set))

```



```

{
fgets(send_str, 256, stdin);
send_str[strlen(send_str)-1] = '\0';

if (strncmp("quit", send_str, 4) == 0)
{ /*退出程序*/
close(sockfd);
exit(0);
}
send(sockfd, send_str, strlen(send_str), 0);
}

/*判断是否已将新建的描述符加到 select() 监视的读的文件描述符集合中*/
if (FD_ISSET(sockfd, &rfd_set))
{
recvbytes=recv(sockfd, buf, MAXDATASIZE, 0);
if (recvbytes == 0)
{
close(sockfd);
exit(0);
}
buf[recvbytes] = '\0';
printf("Server: %s\n", buf);

printf("%s: ",name);
fflush(stdout);
}
/*判断是否已将新建的描述符加到 select() 监视的异常的文件描述符集合中*/
if (FD_ISSET(sockfd, &efd_set))
{
close(sockfd);
exit(0);
}
}
}

```

当同时运行客户端和服务端程序后，只要在客户端建立一个账号，双方就可以收发数据了。

## 9.4 本章小结

本章首先简单的介绍了 TCP/IP 协议，接着主要讲述了 IP 地址、域名、网络套接字、端口等概念和相关的处理。

接着本章介绍了 socket 的定义及其类型，并逐个介绍了常见的 socket 相关基本函数。

接着介绍的是面向连接的套接字通信和无连接的套接字通信。本章通过大量的实例，让读者熟练掌握基于套接字的网络编程。

## 9.5 课后练习

1、在 Linux 系统下编写一个 socket 程序，要求服务器端等待客户的连接请求。一旦有客户连接，服务器端打印出客户端的 IP 地址和端口，并向服务器端发送欢迎信息和时间。

2、编写一个基于 TCP 协议的网络通信程序，要求服务器通过 socket 连接后，并且要求输入用户，判断为 liu 时，才向客户端发送字符串“Hello, you are connected!”。在服务器端显示客户端的 IP 地址或域名。

3、编写一个以客户机/服务器模式工作的

# 第 10 章 Qt 编程基础

## 学习目标

- 掌握 QT 安装方法
- 学会 QT 简单程序的创建
- 信号
- 槽

## 10.1 QT 介绍

本节将要简要介绍 GUI 和 QT 库,以及讨论 QT 的优点——这也是我们选择 QT 而不是其他 GUI 设计库的原因,并教会你如何安装 QT。

QT 是一个适用于多平台图形界面程序开发的 C++工具包。除 C++库之外,QT 还包含了一些便捷的开发工具,让编程变得快速直接。QT 的跨平台能力和国际化支持保证了 QT 应用程序占有尽可能广阔的市场。

自 1995 年以来,基于 QT 的 C++应用程序就在商业应用中占据核心地位。QT 在各领域被广泛应用,比如消费类电子,医疗器械、工控机床,金融交通终端等,同时也受到各大小企业的青睐,如 Adobe, IBM, Motorola, NASA, Volvo 和大量的小公司及组织。

目前 QT 的版本发展已经达到 4 系列,很多新增的功能不断完善,但在嵌入式领域,由于硬件资源的有限性,QT4 对硬件资源的要求较高,并不是最适合的嵌入式 GUI 版本,相比之下,QT3.3 保持着早期版本的易用性和功能,同时也增加了一些重要功能和新类,最重要的一点是其对硬件资源的要求不高,能流畅的在目前基本标配的嵌入式设备上运行。

QT 是面向对象的。而且 QT 类的特征是减少开发者的工作量,并且提供可靠的接口来加速用户的学习。

### 10.1.1 GUI 的作用

GUI (Graphical User Interface),即图形用户界面(又作图形用户接口),是指采用图形方式显示的计算机操作用户界面。与早期计算机使用的命令行界面相比,图形界面对于用户来说在视觉上更易于接受,且更便于操作。

GUI 工具包(或 GUI 库)是构造图形用户界面(程序)所使用的一套按钮、滚动条、

菜单和其他对象的集合。Linux 下有很多可供使用的 GUI 库，开发常见的 GUI 设计库有 QT、Gtk、MiniGUI、MicroWindow 等，本书中仅对 QT 进行简要讲解和说明。

## 10.1.2 QT 的特点

QT 是一个基于 C++ 编程语言的 GUI 工具包。由于 QT 是基于 C++，速度快，易于使用，并具有很好的可移植性。所以，当需要开发 Linux/Unix 或 MS Windows 环境下的 GUI 程序时，QT 是最佳选择。

### 1. 可移植性

QT 不只是适用于 Linux/Unix，它同样适用与 MS Windows。而作为当今世界的两大主流操作系统，这两款平台都分别拥有几百万的用户，如果想要你所开发的 GUI 程序受到更多用户的青睐，当然需要选择一款既使用于 Linux/Unix 又使用于 Windows 的 GUI 工具包，而 QT 则恰好可以充当这个角色。

### 2. 易用性

上文中已经讲到，QT 是一个 C++ 工具包，它由几百个 C++ 类构成，自然在你的 QT 程序中使用这些类。因为 C++ 是面相对象的编程 (Object-Oriented Programming, OPP) 语言，而 QT 是基于 C++ 构造的，因此，QT 也具有 OPP 的所有特性和优点。

### 3. 运行速度

QT 非常容易使用，且也具有很快的速度，这两方面通常不可能同时达到。QT 的这一优点得以于 QT 开发者花费了大量的时间来优化他们的产品。

导致 QT 鼻其他许多 GUI 工具包运行速度快的另一个原因是它的实现方式。QT 是一个 GUI 仿真工具包，这意味着它不调用任何本地工具包作调用。许多 GUI 开发包使用 API 层或 API 仿真，这些方法均以不同的方式调用本地工具包，从而导致程序运行速度的下降。而 QT 直接调用操作平台上的低级绘图函数仿真操作平台的 GUI 库，自然使得程序的运行速度得到提高。

### 4. 其他特点

QT 包括一组丰富的提供图形界面功能支持的窗口部件 (Windows 术语叫控件)。QT 采用了一种全新并且可选的被称为“信号与槽”的对象间通信机制，以代替老的，不安全的回调技术。QT 也提供传统的事件模型用以处理诸如鼠标点击、击键等动作。QT 的跨平台 GUI 程序能使用现代程序所要求的各种用户界面，比如菜单，背景菜单，拖拽与放下，工具栏等。

直观的命名规则和统一的编程步骤简化了代码编写。QT 还包含了一个叫 QT 设计器的图形化设计界面。QT 设计器支持含无限制定位在内的强大的布局能力。QT 设计器可以用于纯粹的 GUI 设计，也可以利用其内建的代码编辑器来创建完整的应用程序。

QT 拥有对 2D/3D 图形的完美支持。QT 是事实上用于独立于平台的 OpenGL 编程的标准 GUI 工具包。

利用标准的数据库,QT 使创建独立于平台的数据库应用成为可能。QT 的内建驱动支持 Oracle,Microsoft SQL Server,Sybase Adaptive Server,IBM DB2,PostgreSQL,MySQL,Borland, Interbase,SQLite,以及各种 ODBC 的数据库。QT 的数据库功能完全集成到了 QT 设计器中,能提供数据库的生动的预览。QT 包括专门的数据库组件,并且任何内建的或自定义的组件都可以数据相关。

在所有支持 QT 风格和主题的平台,QT 程序的外观与本机相同。源于单一的源码树,要产生 Windows 95~XP,Mac OS X,Linux,Solaris,HP-UX 和其他带 X11 的 Unix 上的应用程序,只需重新编译一次即可。QT 应用程序也可以编译为在嵌入式环境中运行。QT 的 qmake 构件工具能产生 makefiles 或者 .dsp 文件以适应目标平台。

自从 QT 的体系结构在基本系统上体现其优势以来,许多客户在 Windows,Mac OS X 和 Unix 上用 QT 来进行单一平台的开发,因为他们喜欢 QT 的方法。QT 包括许多重要的特殊平台的特征,比如 Windows 上的 ActiveX,Unix 上的 Motif。

QT 普遍使用 Unicode 并且有良好的国际化支持。QT 包括 QT Linguist 等工具可用来协助翻译。应用程序可以很容易地使用和混合使用阿拉伯语,汉语,英语,希伯来语,日语,俄语以及其他 Unicode 所支持的语言所写成的文本。

QT 包括许多专用领域的类。比如,QT 拥有包括 SAX、DOM 语法分析器的 XML 模块。使用 QT 的 STL 兼容的集合类能够把对象储存在内存里面。QT 的 I/O 类和网络类提供使用标准协议来处理本地和远程文件的能力。

通过插件和动态连接库可以扩展 QT 程序的功能。插件可以提供附加的代码,数据库驱动,图象格式,风格和部件。插件和库可以以它们自己的版权为名义出售。

QT 是全世界广为使用的成熟的 C++ 工具包。除广泛的商业应用之外,QT 自由版是一个叫 KDE 的 Linux 桌面环境的基础。QT 的跨平台构件系统,可视化设计,优美的 API 使程序开发成为一件乐事。

### 10.1.3 QT 的安装

QT 安装过程非常简单。这一节介绍怎样在 Linux/Unix 下安装自由版的 QT。

Qt 自由版支持 Unix/X11,例如 Red Hat Linux 在安装完毕之后,Qt 就会安装在 /usr/lib/qt\* 之中(其中\*表示版本名称),在 Red Hat AS4 中的 Qt 版本是 Qt 3.3。

您也可以自行至 Trolltech 网站下载新的版本来进行编译或安装,下载网址是:

```
http://qt.nokia.com/downloads-cn
```

在 Unix/X11 的平台下安装(新的)Qt,您可能需要 root 帐号来进行移动与编译等动作,这取决于您要安装 Qt 的路径的权限。

#### 1. 解压安装包

```
cd /usr/local
gunzip qt-x11-version.tar.gz      # 对这个包进行解压速
tar xf qt-x11-version.tar         # 对这个包进行解包
```

## 2. 设置环境变量

这样就会创建一个包含主要的包中文件的/usr/local/qt-version 目录。

把 qt-version 重新命名为 qt（或者建立一个链接）：

```
mv qt-version qt
```

这里假设 Qt 要被安装到/usr/local/qt 路径下。

在你的主目录下/etc/profile 文件中设置一些环境变量。如果它们并不存在的话，就创建它们。设置环境变量的格式如下：

QTDIR=你安装 Qt 的路径

PATH=用来定位 moc 程序和其它 Qt 工具

MANPATH=访问 Qt man 格式帮助文档的路径

LD\_LIBRARY\_PATH=共享 Qt 库的路径

举例说明：

在/etc/profile 文件中，添加下面这些行：

```
QTDIR=/usr/local/qt
```

```
PATH=$QTDIR/bin:$PATH
```

```
MANPATH=$QTDIR/man:$MANPATH
```

```
LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
```

```
export QTDIR PATH MANPATH LD_LIBRARY_PATH
```

在/etc/pam.d/login 文件中，添加下面这些行：

```
setenv QTDIR /usr/local/qt
```

```
setenv PATH $QTDIR/bin:$PATH
```

```
setenv MANPATH $QTDIR/man:$MANPATH
```

```
setenv LD_LIBRARY_PATH $QTDIR/lib:$LD_LIBRARY_PATH
```

做完这些之后，你需要重新登录，或者在继续工作之前重新加载/etc/profile 文件，方法如下：

```
[root@localhost qt]# source /etc/profile
```

这样环境变量设置好了。

## 3. 配置 QT 库

在进行编译之前 Qt 库先要进行相关的配置，包括连编实例程序、教程和工具（比如 Qt Designer 设计器）。配置方法如下：

```
[root@localhost qt]# ./configure
```

这样的话就为你的机器配置 Qt 库。

注意在默认条件下 GIF 文件支持选项是关闭的。运行./configure -help 就会得到配置选项的一个列表。阅读 PLATFORMS 文件能够得到被支持的平台的列表。

## 4. 编译 QT 库

接下来进行编译，方法如下：

```
[root@localhost qt]# make
```

这样就 QT 就编译好了，生成和编译了所有的库、例程和教程。

如果你还有问题，请看 <http://www.trolltech.com/platforms/>。

## 5. 查看 QT 版本

你可以通过查看你机器上的 QT 版本来检验刚才的安装是否成功。

输入命令如下：

```
[root@localhost ~]# qmake -v
Qmake version: 1.07a (Qt 3.3.3)
Qmake is free software from Trolltech AS.
```

看到你机器上的 QT 版本，说明 Qt 已经安装完毕。

关于在 MS Windows 上安装 QT 及从 Troll Tech 公司购买 QT 专业版和企业版的软件使用许可，更多信息请访问 <http://www.troll.no> 站点

## 10.2 Designer 快速创建工程

接下来我们学习如何使用图形化设计器 Designer，快速创建 QT 工程。使用 Designer 进行 QT 编程的好处显而易见——“所见即所得”，就是说用户将看到的 GUI 会在我们的设计过程中非常直观地体现出来，并能够随心所欲的摆放和调整设计器中的控件及布局。

下面我们学习怎样创建、编译和运行一个非常简单的 QT 程序，向你初步介绍 QT 程序的开发过程。

### 10.2.1 Designer 使用

#### 1. 启动 Designer 设计器

在命令行输入如下命令：

```
# designer
```

#### 2. Designer 设计器界面

Designer 设计器界面如图 10-1 和图 10-2 所示



图 10 1 Designer 设计器启动界面

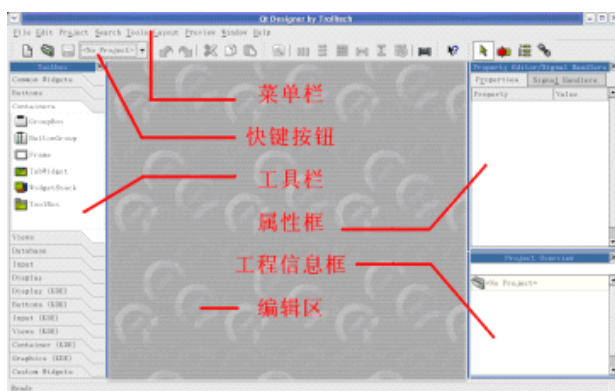


图 10 2 Designer 设计器界面

### 3. Designer 设计器简介

#### 菜单栏:

包含了 File、Edit、Project、Search、Tools、Layout、Preview、Window、Help 等菜单，其中囊括了 Designer 设计器的全部功能。

#### 快捷按钮:

包含 New、Open、Save、Undo、Redo 等 Designer 设计器常用功能的快捷按钮。

#### 工具栏 (Toolbox):

分类列举了 QT 中常用的各种控件，如 PushBotton、TextLabel、TextEdit、LineEdit 等。

#### 属性框 (Property Edit/Signal Handlers):

用于显示当前窗体或控件的相关属性。

#### 工程信息框 (Project Overview):

用于显示当前工程的相关信息。




### 编辑区:

可以在这里编辑你想要的 GUI。

## 10.2.2 Desinger 创建 Hello World

### 1. 创建工程

通过菜单栏，选择“File”→“New”→“C++ Project”，输入工程名及路径后，点击“OK”并点击快捷按钮 Save 进行保存，此时创建了一个新的工程如图 10 3 所示。

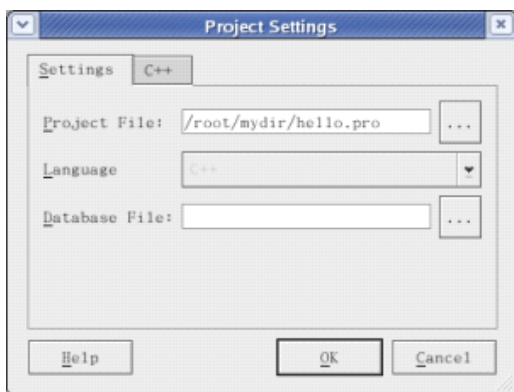



图 10 3 创建了一个新的工程

### 2. 创建窗体

同样通过菜单栏，选择 File→New→Dialog，输入工程名及路径后，点击“OK”创建一个对话框窗体，此时该窗体的内容是空的，且标题栏和控件名均默认为“Form1”。

接下来通过鼠标拖曳窗体的边缘，将窗体调整到想要的大小，再选取工具栏（Toolbox）→Common Widgets→PushButton，在窗体上画一个按钮控件并双击这个按钮将其显示文本改为“Hello World”。

点击快捷按钮 Save 进行保存，此时创建了一个新的窗体如图 10 4 所示。

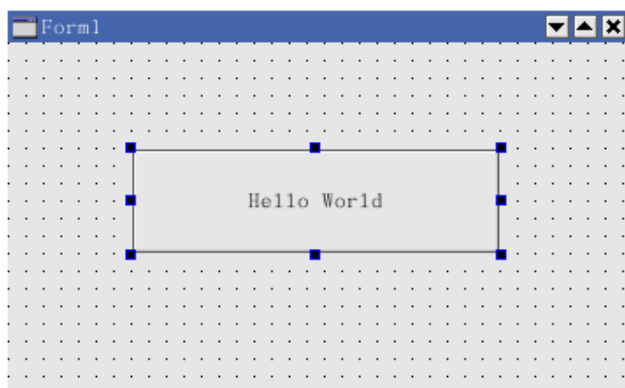



图 10 4 创建了一个新的窗体

### 3. 创建入口

同样通过菜单栏，选择 File—>New—>C++ Main-File (main.cpp)，点击“OK”弹出一个对话框。选取 Form1 作为主窗体，点击“OK”确认。接下来会自动生成一段 C++ 源代码，其中包含了该 QT 程序的入口（主函数）。

点击快捷按钮  Save 进行保存，此时创建了一个新的 main.cpp 文件如图 10 5 所示。

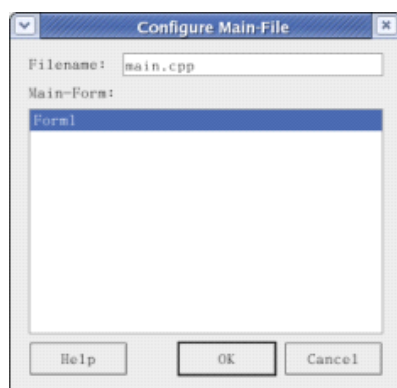


图 10 5 创建了一个新的 main.cpp 文件

## 10.2.3 Hello World 编译

创建完 Hello World，我们可以在之前指定的目录下至少找到三个文件，分别是：hello.pro、form1.ui 和 main.cpp。

```
[root@localhost mydir]# ls
form1.ui  hello.pro  main.cpp
```

## 1. 使用 Designer 生成的文件编译

我们可以直接编译 Designer 设计器生成的文件，只需做如下步骤：

(1) 生成 Makefile

```
[root@localhost mydir]# qmake
[root@localhost mydir]#
[root@localhost mydir]# ls
form1.ui  hello.pro  main.cpp  Makefile
```

(2) 编译

```
[root@localhost mydir]# make
/usr/lib/qt-3.3/bin/uic form1.ui -o .ui/form1.h
g++ -c -pipe -Wall -W -O2 -g -pipe -m32 -march=i386
-mtune=pentium4 -DQT_NO_DEBUG -DQT_SHARED -DQT_THREAD_SUPPORT
-I/usr/lib/qt-3.3/mkspecs/default -I. -I/usr/lib/qt-3.3/include
-I.ui/ -I. -I.moc/ -o .obj/main.o main.cpp
/usr/lib/qt-3.3/bin/uic form1.ui -i form1.h -o .ui/form1.cpp
g++ -c -pipe -Wall -W -O2 -g -pipe -m32 -march=i386
-mtune=pentium4 -DQT_NO_DEBUG -DQT_SHARED -DQT_THREAD_SUPPORT
-I/usr/lib/qt-3.3/mkspecs/default -I. -I/usr/lib/qt-3.3/include
-I.ui/ -I. -I.moc/ -o .obj/form1.o .ui/form1.cpp
/usr/lib/qt-3.3/bin/moc .ui/form1.h -o .moc/moc_form1.cpp
g++ -c -pipe -Wall -W -O2 -g -pipe -m32 -march=i386
-mtune=pentium4 -DQT_NO_DEBUG -DQT_SHARED -DQT_THREAD_SUPPORT
-I/usr/lib/qt-3.3/mkspecs/default -I. -I/usr/lib/qt-3.3/include
-I.ui/ -I. -I.moc/ -o .obj/moc_form1.o .moc/moc_form1.cpp
g++ -o hello .obj/main.o .obj/form1.o .obj/moc_form1.o
-L/usr/lib/qt-3.3/lib -L/usr/X11R6/lib -lqt-mt -lXext -lX11 -lm
```

```
[root@localhost mydir]# ls
form1.ui  hello  hello.pro  main.cpp  Makefile
```

编译完成，生成文件 hello 就是我们想要的 QT 程序。

(3) 执行

```
[root@localhost mydir]# ./hello
```

在终端运行 hello 文件，就可以看到我们刚才创建的窗体了如图 10-6 所示：

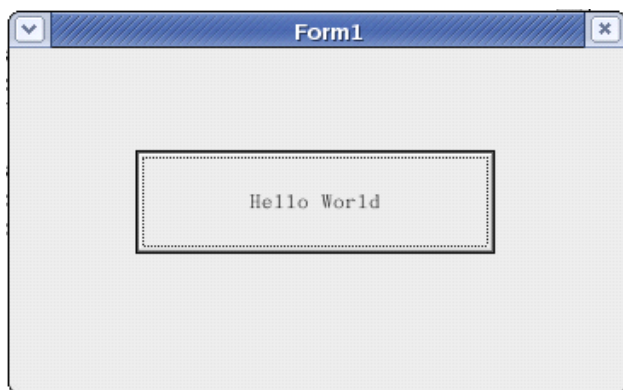


图 10 6 创建的窗体

## 2. 使用源代码编译

Designer 设计器让界面开发变得简单快捷，但使用 Designer 设计器生成的文件直接编译 QT 程序则存在程序代码修改不便导致开发困难的致命缺陷。故很多程序员较少使用 Designer 设计器生成的文件直接进行编译，而是使用 uic 转换工具将 Designer 设计器生成的\*.ui 文件转换生成后缀名为\*.h 和\*.cpp 的 C++源代码文件，再通过对这些源代码文件进行必要的二次修改后，方才进行编译。

那么究竟什么是 uic 转换工具呢？不要急，且听我慢慢道来。

在以上使用 Designer 生成的文件编译过程中，编译时我们看到了大量系统提示信息，如：

```
/usr/lib/qt-3.3/bin/uic form1.ui -o .ui/form1.h
.....
/usr/lib/qt-3.3/bin/uic form1.ui -i form1.h -o .ui/form1.cpp
.....
```

以上列举的这两句系统提示信息，即是将 Designer 设计器生成原始图形文件\*.ui 逐一转换为对应的\*.h、\*.cpp 文件的过程，这一过程的执行涉及到一个关键的 QT 工具，也就是 uic 转换工具。

下面我们仍以 Designer 设计器产生的原始 form1.ui 文件为起点，使用 uic 转换工具生成 C++源代码文件并编译这些源代码，藉此分析 uic 工具的具体作用及如何使用源代码进行编译。其步骤如下：

### （1）生成或编辑源文件

使用 Designer 设计器生成了上文中提到包括 form1.ui 在内的三个文件，首先我们将原有的工程文件 hello.pro 文件删除。

```
[root@localhost mydir]# rm -rf hello.pro
```

随后，使用 uic 转换工具，在此例中它能够将 form1.ui 转换为 form1.h 和 form1.cpp

两个 C++ 源代码文件，其中在 form1.h 中定义了对话框类 Form1，在 form1.cpp 中实现了这个类 Form1。

```
[root@localhost mydir]# uic form1.ui -o form1.h
[root@localhost mydir]# uic form1.ui -i form1.h -o form1.cpp
```

接下来，删除原有的 form1.ui 文件（或移动至其他目录以备份）。

```
[root@localhost mydir]# rm -rf form1.ui
```

最后当前目录下仅保留 form1.h、form1.cpp 和最初生成的 main.cpp 三个文件即可。

```
[root@localhost mydir]# ls
form1.cpp form1.h main.cpp
```

此时，你可以通过修改 form1.h、form1.cpp 这两个 C++ 源代码文件中的代码，对你想要编译生成的 QT 程序及窗体进行修改。若不进行任何改动，将编译得到与使用 Designer 设计器生成的文件直接编译所得程序完全相同的 QT 程序。

另外，假使你并没有使用 Designer 设计器来编写 QT 程序，甚至打算完全脱离 Designer 设计器，需要按照 QT 及 C++ 的语法，同样编辑相应 form1.cpp、form1.h 和 main.cpp 文件。

## (2) 生成工程文件

```
[root@localhost mydir]# qmake -project -o hello.pro
[root@localhost mydir]# ls
form1.cpp form1.h hello.pro main.cpp
```

此处若不带参数“-o hello.pro”则将默认生成一个与目录名相同的工程文件，在本例中即为 mydir.pro。

## (3) 生成 Makefile

```
[root@localhost mydir]# qmake
[root@localhost mydir]# ls
form1.cpp form1.h hello.pro main.cpp Makefile
```

## (4) 编译

```
[root@localhost mydir]# make
g++ -c -pipe -Wall -W -O2 -g -pipe -m32 -march=i386
-mtune=pentium4 -DQT_NO_DEBUG -DQT_SHARED -DQT_THREAD_SUPPORT
-I/usr/lib/qt-3.3/mkspecs/default -I. -I.
-I/usr/lib/qt-3.3/include -o form1.o form1.cpp
g++ -c -pipe -Wall -W -O2 -g -pipe -m32 -march=i386
-mtune=pentium4 -DQT_NO_DEBUG -DQT_SHARED -DQT_THREAD_SUPPORT
-I/usr/lib/qt-3.3/mkspecs/default -I. -I.
-I/usr/lib/qt-3.3/include -o main.o main.cpp
/usr/lib/qt-3.3/bin/moc form1.h -o moc_form1.cpp
```

```
g++ -c -pipe -Wall -W -O2 -g -pipe -m32 -march=i386
-mtune=pentium4 -DQT_NO_DEBUG -DQT_SHARED -DQT_THREAD_SUPPORT
-I/usr/lib/qt-3.3/mkspecs/default -I. -I.
-I/usr/lib/qt-3.3/include -o moc_form1.o moc_form1.cpp
g++ -o hello form1.o main.o moc_form1.o
-L/usr/lib/qt-3.3/lib -L/usr/X11R6/lib -lqt-mt -lXext -lX11 -lm
```

```
[root@localhost mydir]# ls
form1.ui hello hello.pro main.cpp Makefile
```

编译完成，同样生成了我们想要的 QT 程序 hello。

#### (4) 运行 Hello World

同样运行这个 hello 文件，由于没有对代码进行改动，仍然可以看到与我们刚才在 Designer 设计器中创建的窗体相同的 QT 程序如所示。

```
[root@localhost mydir]# ./hello
```



图 10 7 Hello World 窗体

## 10.2.4 Hello World 分析

使用 vi 打开并查看 Hello World 的 form1.cpp、form1.h（二者使用 uic 转换工具由 form1.ui 转换而来）、main.cpp 及 hello.pro 文件。

### 1. 工程文件

hello.pro 是该工程的工程文件，内容如下：

```
TEMPLATE = app                #QT 模板为 app
INCLUDEPATH += .              #工程文件路径为当前目录

# Input
```

```
HEADERS += form1.h           #工程头文件包含 form1.h
SOURCES += form1.cpp main.cpp #工程文件源码包含 form1.cpp 与
main.cpp
```

## 2. 入口文件

入口文件内容如下：

```
#include <qapplication.h>           // 包含 QT 窗口类的头文件
qapplication.h
#include "form1.h"                   // 包含窗口 1 的类原型声明头文件
form1.h

int main( int argc, char ** argv )   // 主函数入口
{
    QApplication a( argc, argv );    // 创建 QApplication 对象 a,
对窗口程序进行管理
    Form1 w;                          // 创建窗口 1 的对象 w
    w.show();                         // 调用对象 w 的父类成员方法 show() 进
行窗口显示
    // 下面使用 connect 方法将对象 a 的最后一个窗口的关闭信号与系统槽函数
quit() 进行连接
    a.connect( &a, SIGNAL( lastWindowClosed() ), &a,
SLOT( quit() ) );
    return a.exec();                // 调用对象 a 的 exec() 方法进行窗口执行与切换
}
```

## 3. 头文件

头文件内容如下：

```
#ifndef FORM1_H                     // 文件标识宏
#define FORM1_H

// 包含 QT 相关头文件
#include <qvariant.h>
#include <qdialog.h>

// 被引用的类体的前置声明
class QVBoxLayout;
class QHBoxLayout;
class QGridLayout;
class QSpacerItem;
```

```

class QPushButton;

// 类 Form1 的原型声明，从 QDialog 父类公有继承
class Form1 : public QDialog
{
    Q_OBJECT      //声明 Q_OBJECT 宏，支持信号槽机制

    //公有成员方法
public:
    //构造函数
    Form1( QWidget* parent = 0, const char* name = 0, bool modal
= FALSE, WFlags fl = 0 );
    //析构函数
    ~Form1();

    //定义一个 QPushButton 类型的按钮对象
    QPushButton* pushButton1;

protected:

protected slots:          // 受保护的槽函数
    virtual void languageChange();    //用于显示文本信息的虚函数,
实现多态继承

};

#endif // FORM1_H      //文件标识宏

```

#### 4. cpp 文件

cpp 文件内容如下:

```

#include "form1.h"          //包含类 form1 的原型声明的头文件

//包含各引用类的原型声明的头文件

#include <qvariant.h>
#include <qpushbutton.h>
#include <qlayout.h>
#include <qtooltip.h>

```



```

#include <qwhatsthis.h>
#include <qimage.h>
#include <qpixmap.h>

//Form1 的构造函数
Form1::Form1( QWidget* parent, const char* name, bool modal,
WFlags fl )
    : QDialog( parent, name, modal, fl )
{
    if ( !name )    //如若窗口未定义名称
        setName( "Form1" ); //设置默认窗体名称为“Form1”

    pushButton1 = new QPushButton( this, "pushButton1" ); //
实例化一个新的按钮
    pushButton1->setGeometry( QRect( 70, 60, 211, 61 ) ); //
设置该按钮的位置属性
    languageChange(); //显示控件上的
文本信息
    resize( QSize(354, 199).expandedTo(minimumSizeHint()) );
    //设置窗口显示属性,从 QSize(354, 199) 和 minimumSizeHint() (返
回一个 QSize 对象)中取两者之中的最大宽度和最大高度组成一个新的 QSize 对象。
    clearWState( WState_Polished );
    //常见于用 Designer 生成的代码中。由于 WState_Polished 是一个内
部标志,在用户的代码中一般不要使用它。
}

Form1::~~Form1()    //析构函数
{
    // no need to delete child widgets, Qt does it all for us
}

void Form1::languageChange()    //显示空间文本信息
{
    setCaption( tr( "Form1" ) );
    pushButton1->setText( tr( "Hello World" ) ); //在按键上显
示 Hello World
}

```

## 10.3QT 对话框的完善

本节将进行 QT 的进一步学习，我们通过对 Hello World 进行完善作为学习的主要内容。下面我们来给窗体增加更多的控件并通过对其进行初步的美化。

### 10.3.1 QT 对话框的布局

#### 1. 增加控件

我们尝试给 Hello World 的窗体增加一个标签和一个行文本编辑框，此时需要修改 form1.h 和 form1.cpp 两个文件中的源代码。

修改 form1.h，声明两个新的类，如下：

```
class QLabel;  
class QLineEdit;
```

修改 form1.h，在 Form1 类中增加两个公有（public）成员变量，如下：

```
QLabel* textLabel1;  
QLineEdit* lineEdit1;
```

修改 form1.cpp，增加两个头文件，如下：

```
#include <qlabel.h>  
#include <qlineedit.h>
```

修改 form1.cpp，在 Form1 类的构造函数中增加创建控件的代码，如下：

```
lineEdit1 = new QLineEdit( this, "lineEdit1" );  
textLabel1 = new QLabel( this, "textLabel1" );
```

修改 form1.cpp，在 Form1 类的构造函数中增加设置控件位置布局的代码，如下：

```
lineEdit1->setGeometry( QRect( 118, 140, 115, 40 ) );  
textLabel1->setGeometry( QRect( 60, 11, 231, 40 ) );
```

修改 form1.cpp，在 Form1 类的 languageChange() 函数中增加设置控件文本的代码，如下：

```
textLabel1->setText( tr( "Press the button... " ) );  
lineEdit1->setText( QString::null );
```

#### 2. 初步美化

我们通过改变各控件的颜色、修改控件字体等方法实现窗体的初步美化效果。

修改 form1.cpp，在 Form1 类的构造函数中增加设置窗体大小上下限的代码，若上限等于下限则可起到使窗体大小固定不可变的作用，如下：

```
setMinimumSize( QSize( 354, 199 ) );  
setMaximumSize( QSize( 354, 199 ) );
```

修改 form1.cpp，在 Form1 类的构造函数中增加设置窗体及控件背景颜色的代码，

如下:

```
setPaletteBackgroundColor( QColor( 255, 255, 127 ) );  
textLabel1->setPaletteBackgroundColor(   QColor( 170, 255,  
255 ) );  
pushButton1->setPaletteBackgroundColor(   QColor( 255, 170,  
255 ) );
```

修改 form1.cpp, 在 Form1 类的构造函数中增加设置文本颜色的代码, 如下:

```
textLabel1->setPaletteForegroundColor(   QColor( 170, 85,  
255 ) );
```

修改 form1.cpp, 在 Form1 类的构造函数中增加设置文本字体的代码, 如下:

```
QFont textLabel1_font( textLabel1->font() );  
textLabel1_font.setPointSize( 18 );  
textLabel1_font.setBold( TRUE );  
textLabel1->setFont( textLabel1_font );
```

修改 form1.cpp, 在 Form1 类的 languageChange() 函数中设置控件文本的代码中使用脚本语言可实现文本的居中等效果, 如下:

```
textLabel1->setText(   tr( "<p align=\"center\">Press the  
button...</ p>" ) );
```

### 3. 完善结果

编译并运行经过完善的 QT 程序, 可以看到一个全新的 Hello World 界面如图 10 8 所示。

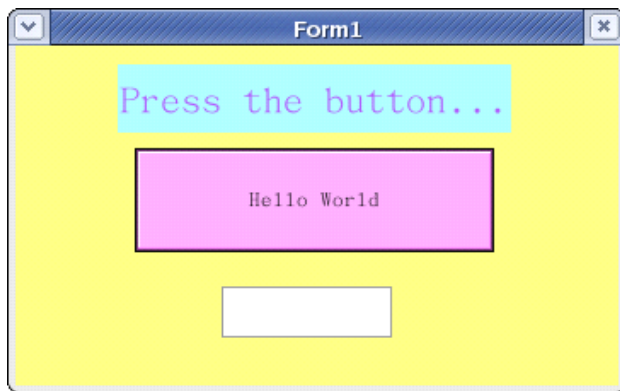


图 10 8 全新的 Hello World 界面

### 10.3.2 QT 对话框的属性

下面简单介绍一下 QT 对话框的部分属性。

#### 1. **name**

继承自 Qobject，控件（对话框）的名称和标识，该属性在构造对象时设置。

#### 2. **enabled**

继承自 QWidget，标识控件（对话框）是否有效，该属性可通过函数 `setEnabled()` 设置。

#### 3. **paletteForegroundColor**

继承自 QWidget，控件（对话框）的前景色，该属性可通过函数 `setPaletteForegroundColor()` 设置。

#### 4. **paletteBackgroundColor**

继承自 QWidget，控件（对话框）的背景色，该属性可通过函数 `setPaletteBackgroundColor()` 设置。

#### 5. **font**

继承自 QWidget，控件（对话框）的字体信息，该属性可通过函数 `setFont()` 设置。

#### 6. **caption**

继承自 QWidget，控件（对话框）的标题，该属性可通过函数 `setCaption()` 设置。

#### 7. **model (isModel)**

继承自 QWidget，标识控件（对话框）是否为模态，一个模态控件防止对其它所有顶级窗口部件得到任何输入，该属性在构造对象时设置。

#### 8. **text**

保存控件上显示的文本，该属性可通过函数 `setText()` 设置。

### 10.3.3 QT 的控件

下面列举 QT 中使用频度较高一些控件类。

#### 1. **QApplication 应用程序类**

#### 2. **QLabel 标签类**

#### 3. **QPushButton 按钮类**

#### 4. **QButtonGroup 按钮组合类**

#### 5. **QGroupBox 群组类**

#### 6. **QDateTimeEdit 日期时间编辑框类**

7. QLineEdit 行编辑框类
8. QTextEdit 文本编辑框类
9. QComboBox 组合框类
10. QProgressBar 进度条类
11. QLCDNumber 数字显示框类
12. QScrollBar 滚动条类
13. QSpinBox 微调框类
14. QSlider 滑动条类
15. QIconView 图标视图类
16. QListView 列表视图类
17. QListBox 列表框类
18. QTable 表格类
19. QImage 图像类
20. QMainWindow 主窗口类
21. QPopupMenu 弹出性菜单类
22. QMenuBar 菜单栏类
23. QToolButton 工具按钮类
24. QToolTip 提示类
25. QAction 动作类
26. QEvent 事件类
27. QHBoxLayout 水平布局类
28. QVBoxLayout 垂直布局类
29. QGridLayout 表格布局类
30. QMessageBox 消息对话框类
31. QProgressDialog 进度条对话框类
32. QWizard 向导对话框类
33. QFileDialog 文件对话框类
34. QColorDialog 颜色对话框类
35. QFontDialog 字体对话框类
36. QPrintDialog 打印对话框类

QT 中所有的控件类（含上述常用控件）及其全部属性均可在第十二章中学习的 QT 帮助文档中找到，就不在此再做累述了。

## 10.4 QT 的信号与槽

用户交互是所有 GUI 应用程序所关心的，通常的做法是通过将某种用户事件（如鼠标点击）与程序动作（例如程序退出）联系起来，使用户能够在图形界面中只使用鼠标来控制程序。

许多 GUI 工具包使用回调函数创建用户交互。但是，回调函数非常复杂，容易混淆，又难以理解。

因此，QT 开发者使用了一种不同于其他 GUI 工具包的方式——信号与槽机制，来完成 QT 程序和控件与用户的交互。这种工作方法依赖于 QT 特有的两个功能：信号和槽。而使用这种新方法只需要一行代码就能够将用户事件和程序动作连接起来。

信号与槽机制是 QT 的核心机制，也是 QT 的核心特性，这是 QT 区别于其它工具包的重要原因。信号和槽是一种高级接口，应用于对象之间的通信。信号与槽技术 QT 独有的通信机制，由 Troll Tech 公司独立开发，而不是 C/C++ 的功能，因此它立于标准的 C/C++ 语言。这种使用信号和槽将用户事件连接到程序动作的方法比回调函数更加容易使用，因此这也是你选择 QT 库而不是其他工具包的两个重要原因。

这一节将首先介绍信号和槽的基本知识和一些 QT 预定义的槽，随后将带你将在 QT 程序中自定义一些信号与槽，最后我们通过把这些信号与槽连接起来使之为我们要实现的功能服务。

### 10.4.1 QT 的事件

大多数 GUI 程序都是由事件驱动，QT 程序也不例外，它的每个动作也都是由幕后某个事件所触发，而能够响应这些事件使得 QT 程序能够与用户和操作系统进行交互。

QT 中有许多事件，常见的有鼠标事件（`QMouseEvent`）、键盘事件（`QKeyEvent`）、定时事件（`QTimerEvent`）、绘屏事件（`QPaintEvent`）、移动事件（`QMoveEvent`）、大小改变事件（`QResizeEvent`）、显示事件（`QShowEvent`）、隐藏事件（`QHideEvent`）、滚轮事件（`QWheelEvent`）、焦点事件（`QFocusEvent`）和用户自定义事件（`QCustomEvent`）等。

某些控件实现了对这些事件进行响应和处理，如 `QWidget` 的函数 `mousePressEvent()` 可以响应和处理 `QMouseEvent` 事件。

### 10.4.2 QT 的信号

#### 1. 概念简介

信号也是成员函数，但是，它并不需要用户在代码中将其实现，而是以一种“连接（后文中将会详细讲述）”的方式间接实现。因此，信号是一种特殊类型的成员函数。

当某个信号所有者（通常为某一控件的对象）内部状态发生改变时，该信号就被发

射(emit)。也只有定义过这个信号的类及其派生类的对象才能够发射这个信号，也就是说这个控件对象内必须含有此信号。

同时，QT 给部分控件预定义了一些信号，例如按钮(QButton 及其子类 QPushButton) 自带有 clicked() 信号，即当按钮被按下时这个信号将被发送。

## 2. 自定义信号

你只需要在类体的原型声明头文件中使用 signals 关键字添加对信号声明既可创建一个自定义的 QT 信号，非常方便。

### (1) 添加信号

仍然以 Hello World 为例，我们尝试给 Hello World 的窗体添加一个自定义的信号 mySignal。此过程需要在 form1.h 文件中的窗体类 Form1 的声明中进行添加，代码如下：

```
signals:
    void mySignal();
```

### (1) 发射信号

假使我们想让窗体在被鼠标点击后发射此信号，那么我们需要重写窗体类 Form1 的鼠标事件响应函数 mousePressEvent()，并在这个函数中发射自定义信 mySignal()。此过程需要在 form1.cpp 文件中添加如下代码：

```
void Form1::mousePressEvent (QMouseEvent *e)
{
    emit mySignal();
}
```

编译并运行这个添加有自定义信号的 Hello World 程序，此时若用鼠标点击窗体，我们的自定义信号 mySignal() 就会被发射。但是，目前我们似乎看不到程序任何反应，因为我们还没有让 QT 程序中的控件对象接受和响应这个信号，那么接着往下看，后面我们将实现对这个信号的响应。

## 10.4.3 QT 的槽

### 1. 概念简介

槽，实际上是标准的成员函数，因此槽有时也被称作槽函数。槽函数不仅可以像调用其他函数一样被调用，而且它们的写法同编写一个普通成员函数并无大的差别。但是槽函数比一般的成员函数增加了一些特殊的功能，这使得它们能够被信号连接。

QT 也给部分控件预定义了一些公有槽，例如对话框(QDialog 及其父类 QWidget) 自带有 close() 槽函数，该槽函数的功能是关闭该对话框。

### 2. 自定义槽

槽的声明也是在头文件类体的原型声明中，不同的是槽的声明不仅需要 slots 关键字，还需要在 slots 关键字加上存取权限（private、protected 或 public），这是因为槽函数标准成员函数的缘故。

#### （1）添加槽

接着上面的例子，我们给 Hello World 的窗体添加一个自定义的槽 mySlot。此过程需要在 form1.h 文件中的窗体类 Form1 的声明中进行添加，代码如下：

```
protected slots:
    virtual void mySlot();
```

因为槽函数标准成员函数，槽也能够声明为虚函数。此例中将其以虚函数声明并非必须，只是为其子类的多态继承提供方便。

#### （2）实现槽

同样因为槽函数标准成员函数的缘故，我们也必须在像实现普通成员函数一样实现我们自定义的槽函数。假使我们想让这个槽函数实现改变在 Hello World 窗体中 lineEdit1 控件的文本内容，此过程需要在 form1.cpp 文件中编写 mySlot() 函数，代码如下：

```
void Form1::mySlot()
{
    lineEdit1->setText( "This is mySlot!" );
}
```

此时编译并运行修改后的 Hello World 程序仍然看不到任何响应，这是因为此时我们自定义的槽函数还没有被调用，也就是说我们还没有在代码中设定如何触发这个槽函数的执行，这也是我们后面紧接着要学习去做的事情。

## 10.4.4 信号与槽的关联

### 1. 概念及语法

信号与槽的关联相关的函数为 QObject 类的静态公有成员函数 connect ()，其原型如下：

```
bool connect ( const QObject * sender, const char * signal,
const QObject * receiver, const char * member )
```

#### （1）功能说明

通过调用函数 connect () 可以将信号与槽联系起来的。这种连接一旦被建立，只要被连接的信号被发送，则被连接的槽函数将会被调用执行。

一个信号可以连接多个槽函数，当信号被发射，这些槽函数将以任意顺序被调用执行。



## (2) 参数

sender, 信号的发送对象; signal, 需要连接的信号; receiver, 信号的接受对象, 即槽函数的拥有者; member, 需要连接的槽函数。

其中 signal 和 member 为字符串, 其文本格式分别为“SIGNAL (信号名)”和“SLOT (槽函数名)”。

## (3) 返回值

如果信号和槽被成功连接, 返回真。如果它不能创建连接, 返回假, 例如, 如果 QObject 不能检验 signal 或 member 的存在, 或者如果它们的标签不协调。

## 2. 演示实例

回到经过我们完善的 Hello World 程序, 尝试利用 QT 给按钮 (QPushButton 及其子类 QPushButton) 预定义的信号 clicked () 以及对话框 (QDialog 及其父类 QWidget) 预定义的槽函数 close (), 建立一个信号与槽的连接。

此时, 需要修改 form1.cpp, 在其构造函数中添加如下代码:

```
connect (pushButton1, SIGNAL(clicked()), this, SLOT(close()));
```

(注意: 此代码应添加在信号的发送对象和接受对象被创建之后, 否则会有段错误出现。)

编译并运行经过修改的 QT 程序, 尝试点击界面上的按钮, 观看会有怎样的事情发生。

同理, 我们可以在 form1.cpp 文件构造函数中添加连接刚才我们自定义的信号和槽的代码。其代码如下:

```
connect (this, SIGNAL(mySignal()), this, SLOT(mySlot()));
```

再次编译并运行经过修改的 QT 程序, 尝试点击窗体的空白处, 如果此时你看到行文本编辑框中出现“This is mySlot!”字样, 那么说明你已经成功了!

请思考我们的自定义信号 mySignal () 与自定义槽 mySlot () 进行连接之后, 我们调用这个槽函数的过程: 鼠标单击事件→鼠标事件响应函数→发射信号→接收信号→调用槽函数→改变行文本编辑框中的内容。

## 10.5本章小结

## 10.6信号与槽 FAQ:

### 1. 元对象编译工具 moc

问题: 信号与槽机制独立于标准的 C/C++ 语言, 但编译 QT 程序实际上仍然使用的是

标准 C/C++ 的编译器，那标准 C/C++ 的编译器究竟是如何编译含有信号与槽的 QT 程序的呢？

答案：在编译含有信号与槽的 QT 程序时，借助了一个称为 moc (Meta Object Compiler) 的 QT 工具，该工具是一个 C/C++ 预处理程序，它为高层次的事件处理自动生成所需要的附加代码。有了这些附加代码标准 C/C++ 的编译器才能编译含有独立于标准的 C/C++ 语言的信号与槽的 QT 程序。

## **2. 信号与槽的适用范围**

问题：信号与槽是不是在 QT 中随处可用？

答案：并不是这样的，只有从 QObject 或其子类 (例如 QWidget) 派生的类才包含信号和槽，另一方面所有从 QObject 或其子类派生的类都能够包含信号和槽。

## **3. 信号与槽的非常规连接**

问题：信号可否连接普通的成员函数？出来槽函数信号就不能与其他函数连接了吗？

答案：信号不可以连接普通的成员函数，但信号可以与信号连接。当发送者的信号发射，与之相连的接收者的信号也会被发射，反之不然。

## **4. 信号的参数**

问题：信号既然是成员函数，那么信号可否带参数？

答案：信号是可以带参数的，当带参数的信号与带有相同类型参数的槽连接，发射这个信号，与之相连的槽函数被调用的同时参数的值也会由信号传至槽。但建议尽量少的去使用信号与槽机制传递参数，因为信号参数与槽各自所带参数的类型和数量不同等情形可能导致程序出现不可预知的错误。

问题：信号可以连接多个槽吗？

答案：信号是可以连接多个槽的。但当信号被发射时，这些被同一个信号连接的槽函数的调用顺序是随机的。

## **5. 信号的返回值**

问题：信号既然是成员函数，那么信号可否有返回值？

答案：信号也可以有返回值，但其返回值无任何意义。另外，信号会在相连的所有槽函数全部返回后返回。

## **6. 槽的权限类型**

问题：既然槽的声明中必须要有权限类型的声明，那么不同存取权限 (private、protected 或 public) 的槽使用起来有什么区别？

答案：区别自然是有的。public slots: 声明为 public 类型的槽，任何对象都可将信号与之相连接。protected slots: 声明为 protected 类型的槽，当前类及其子类的对象可以将信号与之相连接。private slots: 声明为 private 类型的槽，只有类自己的对象可以将信号与之相连接。

## 10.7 实践操作

# 第 11 章 QT 的资源与技巧

## 学习目标

- 掌握 QT 类
- QT 类的使用技巧
- 如何使用 QT 参考文档

## 11.1 QT 的类

QT 沿用了 C++ 的语法和面向对象的编程特性，同时 QT 还封装了自己的类。

### 11.1.1 QT 的类的层次

QT 中类的部分继承关系如图 11-1 所示：

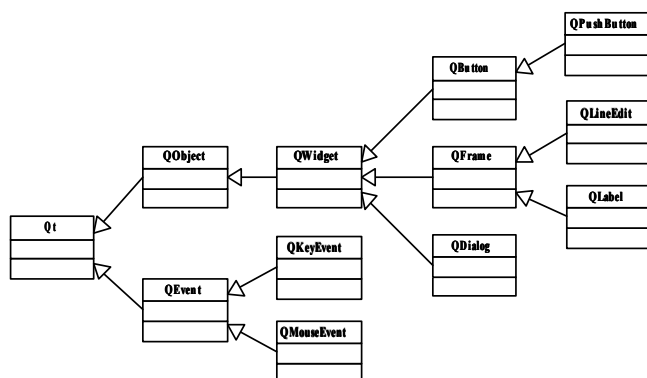


图 11-1 类的部分继承关系

### 11.1.2 QT 类的使用技巧

下面我们利用 QT 中的类和方法，通过 QT 的事件、信号与槽制作一个软键盘程序 virtualKeyBoard，实现与硬键盘输入相同的功能和效果。

## 1. 实现思路

在编辑代码之前，我们先思考一下我们如何实现这个软键盘，然后分析它的实现过程和步骤。

### (1) 画界面

我们的程序既然要软键盘，就必不可少的需要多个按钮来模仿我们的键盘。我们先来模拟最简单的数字键盘，即在窗体中摆放从 0 到 9 的 10 个数字键。接下来我们还在窗体中摆放两个行文本编辑框 QLineEdit，以检验该软键盘能否向不同的控件输入。

### (2) 按钮的信号与槽

将 10 个按钮的 clicked() 信号分别连接 10 个不同的槽函数，而每个槽函数要做的事情就是调用一个方法，以模拟硬键盘上 0 到 9 的 10 个不同按键。

### (3) 发送键盘事件

编写一个用以发送键盘按键事件的公有方法，此方法根据传入的参数，创建键盘按键事件，并将其发送给当前焦点的控件。

## 2. 代码实现

此处仅列出 form1.h 和 form1.cpp 两个较为关键的源代码文件，其余文件同 Hello World。实现代码如下：

### (1) form1.h

```
#ifndef FORM1_H
#define FORM1_H

#include <qvariant.h>
#include <qdialog.h>
#include <qevent.h>
#include <qapplication.h>

class QVBoxLayout;
class QHBoxLayout;
class QGridLayout;
class QSpacerItem;
class QLineEdit;
class QPushButton;

class Form1 : public QDialog
{
    Q_OBJECT

public:
```

```

    Form1( QWidget* parent = 0, const char* name = 0, bool modal
= FALSE, WFlags fl = 0 );
    ~Form1();

    QLineEdit* lineEdit2;
    QLineEdit* lineEdit1;
    QPushButton* pushButton5;
    QPushButton* pushButton3;
    QPushButton* pushButton2;
    QPushButton* pushButton0;
    QPushButton* pushButton6;
    QPushButton* pushButton8;
    QPushButton* pushButton4;
    QPushButton* pushButton9;
    QPushButton* pushButton7;
    QPushButton* pushButton1;

protected:
    void keyPressEvent( QKeyEvent *k );

protected slots:
    virtual void languageChange();
    virtual void on1();
    virtual void on2();
    virtual void on3();
    virtual void on4();
    virtual void on5();
    virtual void on6();
    virtual void on7();
    virtual void on8();
    virtual void on9();
    virtual void on0();
    virtual void sendKey(QEvent::Type type, int key, int ascii,
int state, const QString & text);
};

#endif // FORM1_H

```

## (2) form1.cpp

```

#include "form1.h"

#include <qvariant.h>
#include <qlineedit.h>
#include <qpushbutton.h>
#include <qlayout.h>
#include <qtooltip.h>
#include <qwhatsthis.h>
#include <qimage.h>
#include <qpixmap.h>

Form1::Form1( QWidget* parent, const char* name, bool modal,
WFlags fl )
    : QDialog( parent, name, modal, fl )
{
    if ( !name )
        setName( "Form1" );

    lineEdit1 = new QLineEdit( this, "lineEdit1" );
    lineEdit1->setGeometry( QRect( 10, 10, 340, 21 ) );

    lineEdit2 = new QLineEdit( this, "lineEdit2" );
    lineEdit2->setGeometry( QRect( 10, 40, 340, 21 ) );

    pushButton1 = new QPushButton( this, "pushButton1" );
    pushButton1->setGeometry( QRect( 30, 90, 31, 31 ) );

    pushButton2 = new QPushButton( this, "pushButton2" );
    pushButton2->setGeometry( QRect( 60, 90, 31, 31 ) );

    pushButton3 = new QPushButton( this, "pushButton3" );
    pushButton3->setGeometry( QRect( 90, 90, 31, 31 ) );

    pushButton4 = new QPushButton( this, "pushButton4" );
    pushButton4->setGeometry( QRect( 120, 90, 31, 31 ) );

    pushButton5 = new QPushButton( this, "pushButton5" );
    pushButton5->setGeometry( QRect( 150, 90, 31, 31 ) );

```

```

pushButton6 = new QPushButton( this, "pushButton6" );
pushButton6->setGeometry( QRect( 180, 90, 31, 31 ) );

pushButton7 = new QPushButton( this, "pushButton7" );
pushButton7->setGeometry( QRect( 210, 90, 31, 31 ) );

pushButton8 = new QPushButton( this, "pushButton8" );
pushButton8->setGeometry( QRect( 240, 90, 31, 31 ) );

pushButton9 = new QPushButton( this, "pushButton9" );
pushButton9->setGeometry( QRect( 270, 90, 31, 31 ) );

pushButton0 = new QPushButton( this, "pushButton0" );
pushButton0->setGeometry( QRect( 300, 90, 31, 31 ) );

languageChange();
resize( QSize(362, 175).expandedTo(minimumSizeHint()) );
clearWState( WState_Polished );

connect( pushButton1 , SIGNAL(clicked()) , this ,
SLOT(on1()) );
connect( pushButton2 , SIGNAL(clicked()) , this ,
SLOT(on2()) );
connect( pushButton3 , SIGNAL(clicked()) , this ,
SLOT(on3()) );
connect( pushButton4 , SIGNAL(clicked()) , this ,
SLOT(on4()) );
connect( pushButton5 , SIGNAL(clicked()) , this ,
SLOT(on5()) );
connect( pushButton6 , SIGNAL(clicked()) , this ,
SLOT(on6()) );
connect( pushButton7 , SIGNAL(clicked()) , this ,
SLOT(on7()) );
connect( pushButton8 , SIGNAL(clicked()) , this ,
SLOT(on8()) );
connect( pushButton9 , SIGNAL(clicked()) , this ,
SLOT(on9()) );

```



```

        connect( pushButton0 , SIGNAL(clicked()) , this ,
SLOT(on0()) );

    }

    Form1::~Form1 ()
    {
    }

    void Form1::languageChange()
    {
        setCaption( tr( "Form1" ) );
        pushButton5->setText( tr( "5" ) );
        pushButton3->setText( tr( "3" ) );
        pushButton2->setText( tr( "2" ) );
        pushButton0->setText( tr( "0" ) );
        pushButton6->setText( tr( "6" ) );
        pushButton8->setText( tr( "8" ) );
        pushButton4->setText( tr( "4" ) );
        pushButton9->setText( tr( "9" ) );
        pushButton7->setText( tr( "7" ) );
        pushButton1->setText( tr( "1" ) );
    }

    void Form1::keyPressEvent( QKeyEvent *k )
    {
        printf("key\n");
        if(k->key() == Key_1) printf("1\n");
    }

    void Form1::on1()
    {
        sendKey(QEvent::KeyPress , Qt::Key_1 , 48+1 ,
Qt::UNICODE_ACCEL, "1" );
    }

    void Form1::on2()
    {
        sendKey(QEvent::KeyPress , Qt::Key_2 , 48+2 ,

```

```

Qt::UNICODE_ACCEL, "2" );
    }
    void Form1::on3()
    {
        sendKey(QEvent::KeyPress, Qt::Key_3, 48+3,
Qt::UNICODE_ACCEL, "3" );
    }
    void Form1::on4()
    {
        sendKey(QEvent::KeyPress, Qt::Key_4, 48+4,
Qt::UNICODE_ACCEL, "4" );
    }
    void Form1::on5()
    {
        sendKey(QEvent::KeyPress, Qt::Key_5, 48+5,
Qt::UNICODE_ACCEL, "5" );
    }
    void Form1::on6()
    {
        sendKey(QEvent::KeyPress, Qt::Key_6, 48+6,
Qt::UNICODE_ACCEL, "6" );
    }
    void Form1::on7()
    {
        sendKey(QEvent::KeyPress, Qt::Key_7, 48+7,
Qt::UNICODE_ACCEL, "7" );
    }
    void Form1::on8()
    {
        sendKey(QEvent::KeyPress, Qt::Key_8, 48+8,
Qt::UNICODE_ACCEL, "8" );
    }
    void Form1::on9()
    {
        sendKey(QEvent::KeyPress, Qt::Key_9, 48+9,
Qt::UNICODE_ACCEL, "9" );
    }
    void Form1::on0()

```

```

{
    sendKey(QEvent::KeyPress      ,   Qt::Key_0      ,   48+0      ,
Qt::UNICODE_ACCEL, "0" );
}

void Form1::sendKey( QEvent::Type type, int key, int ascii, int
state, const QString & text = QString::null)
{
    QKeyEvent keyEvent( type , key , ascii , state, text);

    QObject *focusNow=NULL;
    if(lineEdit2->hasFocus()==true) focusNow=lineEdit2;
    if(lineEdit1->hasFocus()==true) focusNow=lineEdit1;

    QApplication::sendEvent(focusNow,&keyEvent);
}

```

### 3. 运行结果

编译并运行改程序，分别通过硬键盘和窗体中的软键盘向两个行文本编辑框输入数字，观察两种输入方式的效果是否相同。如图 11 2 所示

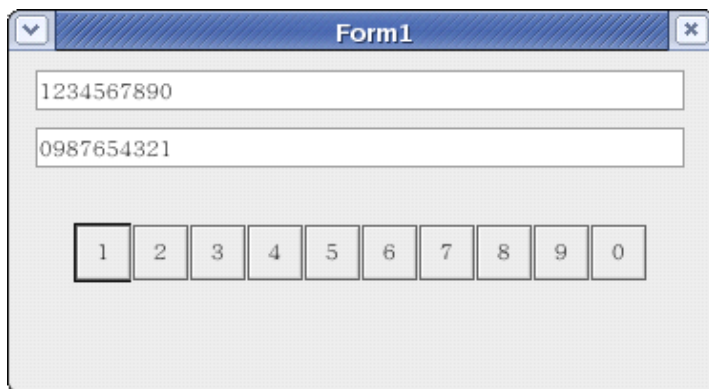


图 11 2 软键盘

通常你可以输入一个 QT 类的类名，如 “QKeyEvent”。



图 11 4 索引选项卡

双击列表中你想要查看的类，如“QKeyEvent”。此时在文档的右部将会显示 QKeyEvent 类的相关信息，其中包括类的简介、继承与被继承关系、类的成员、成员函数详解、属性详解等。

#### 4. 通过超链接查看详情

文档右部的蓝色字体（被点击过则变为紫色）为超链接，点击这些超链接我们可以查看关于这些成员方法或属性的详细说明。

如点击 QKeyEvent 的构造方法如图 11 5 所示。

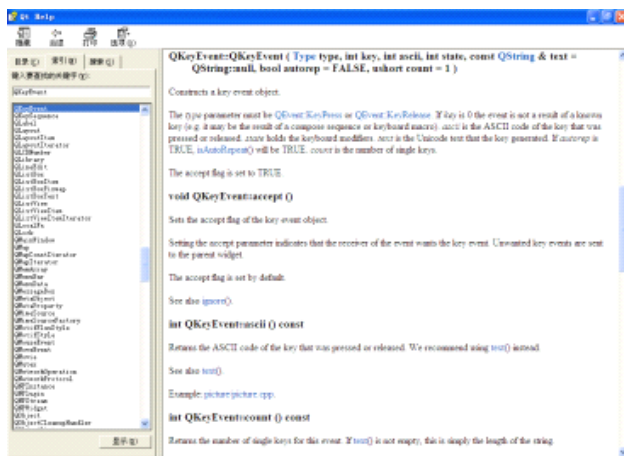


图 11 5 点击 QKeyEvent 的构造方法

此时我们可以看到 QKeyEvent 构造方法的详细说明，包括功能、参数、返回值和注意事项等信息。加入你对说明中的一些文字尚不能理解，还可以通过超链接跳到该文本

的定义或详细说明。

如点击蓝色字体 Type 查看 Type 类型的定义。

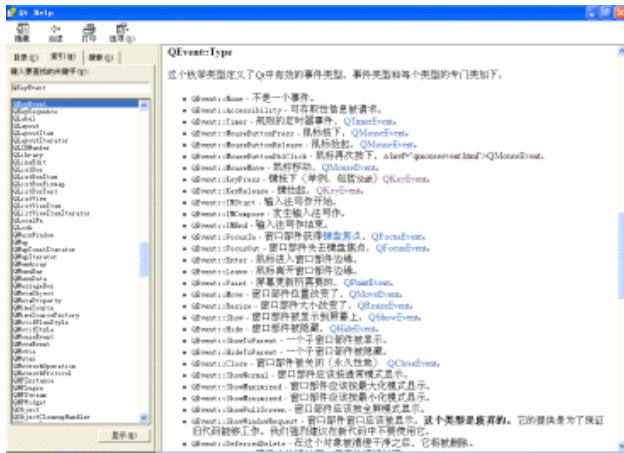


图 11-6 点击蓝色字体 Type

通过类似方法，我们几乎可以通过 QT 参考文档了解和认识 QT 中所有类，以及学会如何使用它们。

## 11.3本章小结