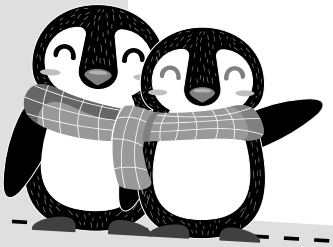

Shell 从入门到精通

（第2版）

张春晓◎编著

清華大學出版社
北京

内 容 简 介

本书是获得大量读者好评的“Linux 典藏大系”中的经典畅销书《Shell 从入门到精通》的第 2 版。本书结合大量实例，详细介绍系统管理员和 Linux 程序员解决实际问题的得力工具——Bash Shell 的用法，并对一些易混淆的内容进行重点提示和讲解。**本书提供 442 分钟教学视频、程序源代码、高清思维导图、教学 PPT 和习题参考答案等超值配套资源，帮助读者高效、直观地学习。**

本书共 15 章，分为 3 篇。第 1 篇认识 Shell 编程，主要介绍 Shell 编程的入门知识，以及 Shell 编程环境的搭建；第 2 篇 Shell 编程核心技术，主要介绍 Shell 变量和引用、条件测试和判断语句、循环结构、函数、数组、正则表达式、文本处理、流编辑器、文本处理利器 awk 命令、文件操作、子 Shell 与进程处理等；第 3 篇 Shell 编程实战，主要介绍 Shell 脚本调试技术和 2 个综合案例的实现。

本书内容丰富，实例典型，易学易用，可操作性强，非常适合 Bash Shell 入门与进阶人员阅读，也适合从事 Linux 系统管理与开发的相关人员阅读，还可作为高等院校相关专业的教材及社会培训机构的培训教材。

图书在版编目（CIP）数据

Shell 从入门到精通 / 张春晓编著. -- 2 版.
北京 : 清华大学出版社, 2024. 10. -- (Linux 典藏大系).
ISBN 978-7-302-67516-7

Ⅰ. TP316.81

中国国家版本馆 CIP 数据核字第 2024Y83C18 号

责任编辑：王中英
封面设计：欧振旭
责任校对：胡伟民
责任印制：沈　露

出版发行：清华大学出版社
网　　址：https://www.tup.com.cn，https://www.wqxuetang.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-83470000　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：大厂回族自治县彩虹印刷有限公司
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：24　　**字　　数**：605 千字
版　　次：2014 年 2 月第 1 版　　2024 年 11 月第 2 版　　**印　　次**：2024 年 11 月第 1 次印刷
定　　价：99.80 元

产品编号：101194-01

前言

随着互联网技术的发展，Linux 已经成为主流的服务器操作系统。在 Linux 系统中，Shell 是用户与系统内核之间进行交互的接口，是整个 Linux 系统非常重要的组成部分。Shell 脚本程序具有简洁、高效的特点，受到了广大系统管理员和开发者的推崇。就连微软公司都为 Windows 系统开发了类似的 Shell 产品——PowerShell，而且让其兼容 Linux 系统。

在 Linux 领域，不断有新的 Shell 产品出现，如 Fish Shell、Nushell、Dune 和 Xonsh 等，但 Bash Shell 依然是绝大多数 Linux 系统默认的 Shell 程序，因此它成为系统管理员和 Linux 系统开发人员解决实际问题的得力工具，而 Shell 脚本编程也成为优秀的系统管理员和 Linux 系统开发者必须掌握的技能之一。

本书是获得大量读者好评的“Linux 典藏大系”中的《Shell 从入门到精通》的第 2 版。截至第 2 版完稿，本书第 1 版累计 14 次印刷，印数 2 万余册。本书在第 1 版的基础上进行了全新改版，不但更新了 Linux 系统的版本，而且更新了 Bash 的版本，还对第 1 版中的一些疏漏进行了修订，并对书中的一些实例和代码重新修订，使其更加易读。

本书基于 Bash Shell 详细介绍 Shell 编程方方面面的知识和技巧。本书以实用为主旨，从 Shell 入门知识和编程环境的搭建讲起，逐步深入 Shell 编程的核心技术，并通过两个综合案例向读者展示如何使用 Shell 脚本解决实际问题。相信在本书的引领下，读者可以在较短的时间内掌握 Shell 脚本编程的相关知识。

关于“Linux 典藏大系”

“Linux 典藏大系”是专门为 Linux 技术爱好者推出的系列图书，涵盖 Linux 技术的方方面面，可以满足不同层次和各个领域的读者学习 Linux 的需求。该系列图书自 2010 年 1 月开始陆续出版，上市后深受广大读者的好评。2014 年 1 月，作者对该系列图书进行了全面改版并增加了新品种。新版图书一上市就大受欢迎，各分册长期位居 Linux 图书销售排行榜前列。截至 2023 年 10 月底，该系列图书累计印数超过 30 万册。可以说，“Linux 典藏大系”是图书市场上的明星品牌，该系列中的一些图书多次被评为清华大学出版社“年度畅销书”，还曾获得“51CTO 读书频道”颁发的“最受读者喜爱的原创 IT 技术图书奖”，另有部分图书的中文繁体字版在中国台湾出版发行。该系列图书的出版得到了国内 Linux 知名技术社区 ChinaUnix（简称 CU）的大力支持和帮助，读者与 CU 社区中的 Linux 技术爱好者进行了广泛的交流，取得了良好的学习效果。另外，该系列图书还被国内上百所高校和培训机构选为教材，得到了广大师生的一致好评。

关于第 2 版

随着技术的发展，本书第 1 版与当前流行的 Linux 系统环境和 Shell 版本有所脱节，这给读者的学习带来了不便。应广大读者的要求，笔者结合当前的主流 Linux 系统和 Bash Shell 版本对第 1 版图书进行了全面的升级改版，推出第 2 版。相比第 1 版图书，第 2 版在内容上的变化主要体现在以下几个方面：

- 将 Linux 系统升级为 Ubuntu 22.04 和 RHEL 9.1 版；
- 将 Bash Shell 升级为 5.2.0 版；
- 增加对 Z Shell 相关内容的介绍；
- 更新 Shell 脚本的 Shebang 行，以兼容新版 Ubuntu 系统；
- 修订第 1 版中的一些疏漏，并对一些不够准确的内容重新表述；
- 新增大量的助记提示，帮助读者快速记忆相关命令和选项；
- 新增思维导图和课后习题，以方便读者梳理和巩固所学知识。

本书特色

1．视频教学，高效、直观

本书特意提供 442 分钟多媒体教学视频讲解重要的知识点，帮助读者高效、直观地学习，从而取得更好的学习效果。

2．内容全面，系统性强

本书全面介绍 Shell 编程方方面面的知识，包括 Shell 编程入门基础、Shell 编程核心技术与 Shell 编程实战，基本涵盖 Shell 编程的所有重要知识点。

3．由浅入深，循序渐进

对于大多数初学者而言，掌握 Shell 编程技术并不容易。为了帮助读者顺利学习，本书从 Shell 编程的基础知识讲起，然后循序渐进地介绍 Shell 编程的核心技术，最后进行编程实战，提高读者的实际开发水平。

4．注重实践，实用性强

本书以当前流行的 Bash Shell 为基础，结合 90 多个实例详解 Shell 编程的核心技术，并对 Shell 编程的常见问题展开论述，无论初学者，还是有一定基础的 Linux 开发和运维人员，都可以学到有用的知识。

5．提供大量的助记提示

在学习 Shell 编程的过程中会碰到大量的命令和选项，这些内容非常繁杂，难于记忆。本书专门提供大量的助记提示来解决这个问题。例如，在讲解 diff 命令的“-c 选项”时，选项说明为“输出包含上下文环境（context）的格式”，其中的 context 是“上下文环境”

的英文，“-c 选项”中的字母 c 来自该单词，这样就可以做到不用死记硬背即可掌握相关命令和选项，从而提高学习效率。

6．案例典型，实战性强，有较高的应用价值

本书最后一篇介绍两个综合案例，这两个案例来源于作者开发的实际项目，有较高的应用价值和参考性。这两个案例分别使用不同的框架组合实现，便于读者融会贯通地理解相关技术，读者对这两个案例稍加修改，便可将其用于自己的项目开发中。

7．提供习题、程序源代码、思维导图和教学 PPT

本书特意在每章后提供多道习题，以帮助读者巩固和自测该章的重要知识点，还赠送教学视频、程序源代码、高清思维导图和教学 PPT 等超值配套资源，以方便读者学习和教师教学。

本书内容

第 1 篇　认识 Shell 编程

本篇涵盖第 1、2 章，主要介绍 Shell 的入门知识和编程环境的搭建，包括学习 Shell 编程的必要性以及 Shell 的起源、功能和分类，并包括 Shell 的特性、向 Shell 脚本传递参数、第一个 Shell 脚本程序以及如何在 Windows、Linux 和 FreeBSD 上搭建 Shell 编程环境，还包括编辑器的选择和系统环境的搭建等。

第 2 篇　Shell 编程核心技术

本篇涵盖第 3～13 章，主要介绍 Shell 编程涉及的所有重要知识点，包括变量和引用、条件测试和判断语句、循环结构、函数、数组、正则表达式、文本处理、流编辑器、文本处理器 awk 命令、文件操作，以及子 Shell 与进程处理等。

第 3 篇　Shell 编程实战

本篇涵盖第 14、15 章，主要介绍 Shell 脚本调试技术，以及如何利用 Shell 脚本解决实际问题，包括 Shell 编程中的常见问题、常用的 4 种 Shell 脚本调试技术，以及两个综合实例——编写系统服务脚本和通过脚本管理 Apache 服务器日志。

读者对象

- Shell 编程入门与进阶人员；
- 基于 Linux 系统的开发人员；
- Linux 系统管理与运维人员；
- 想提高 Linux 系统管理和开发水平的人员；
- 高等院校相关专业的学生；
- 专业培训机构的学员。

阅读建议

- 没有 Linux 编程基础的读者，建议从第 1 章顺次阅读并演练每一个实例；
- 有一定 Linux 编程基础的读者，可以根据实际情况有重点地选择阅读相关章节；
- 对于书中的每个实例，先思考一下实现思路再阅读，学习效果会更好；
- Shell 编程需要进行大量的操作，其相关功能有多种实现方法，读者在阅读本书的基础上可以对书中的实例进行改编，用其他方式实现实例的功能，这样对相关知识点的理解会更加深刻。

配套资源获取方式

本书涉及的配套资源如下：

- 高清教学视频；
- 程序源代码；
- 高清思维导图；
- 教学 PPT；
- 习题参考答案。

上述配套资源有 3 种获取方式：关注微信公众号“方大卓越”，然后回复数字“34”自动获取下载链接；在清华大学出版社网站（www.tup.com.cn）上搜索到本书，然后在本书页面上找到“资源下载”栏目，单击“网络资源”按钮进行下载；在本书技术论坛（www.wanjuanchina.net）上的 Linux 模块进行下载。

技术支持

虽然笔者对书中所述内容都尽量予以核实，并多次进行文字校对，但因时间所限，可能还存在疏漏和不足之处，恳请读者批评与指正。

读者在阅读本书时若有疑问，可以通过以下方式获得帮助：

- 加入本书 QQ 交流群（群号为 302742131）进行提问；
- 在本书技术论坛（见上文）上留言，会有专人负责答疑；
- 发送电子邮件到 book@ wanjuanchina.net 或 bookservice2008@163.com 获得帮助。

张春晓

2024 年 8 月

目录

第 1 篇　认识 Shell 编程

第 1 章　Shell 入门基础 …… 2

1.1　为什么学习和使用 Shell 编程 …… 2

1.2　Shell 简介 …… 2

1.2.1　Shell 的起源 …… 3

1.2.2　Shell 的功能 …… 4

1.2.3　Shell 的分类 …… 4

1.3　Shell 的特性 …… 5

1.3.1　交互式程序 …… 5

1.3.2　创建脚本 …… 6

1.3.3　设置可执行脚本 …… 6

1.4　向脚本传递参数 …… 7

1.4.1　Shell 脚本的参数 …… 8

1.4.2　参数的扩展 …… 9

1.5　第一个 Shell 程序 …… 10

1.5.1　Shell 脚本的基本元素 …… 10

1.5.2　指定命令解读器 …… 11

1.5.3　Shell 脚本的注释和风格 …… 12

1.5.4　如何执行 Shell 程序 …… 13

1.5.5　Shell 程序的退出状态 …… 13

1.6　小结 …… 15

1.7　习题 …… 15

第 2 章　Shell 编程环境的搭建 …… 17

2.1　在不同的操作系统上搭建 Shell 编程环境 …… 17

2.1.1　在 Windows 上搭建 Shell 编程环境 …… 17

2.1.2　在 Linux 上搭建 Shell 编程环境 …… 20

2.1.3　在 FreeBSD 上搭建 Shell 编程环境 …… 22

2.2　编辑器的选择 …… 23

2.2.1　图形化编辑器 …… 24

2.2.2　vi（vim）编辑器 …… 24

2.3　系统环境的搭建 …… 30

2.3.1 Shell 配置文件 …… 30
2.3.2 命令别名 …… 33
2.4 小结 …… 34
2.5 习题 …… 34

第 2 篇 Shell 编程核心技术

第 3 章 变量和引用 …… 36
3.1 深入理解变量 …… 36
3.1.1 什么是变量 …… 36
3.1.2 变量的命名 …… 36
3.1.3 变量的类型 …… 37
3.1.4 变量的定义 …… 39
3.1.5 变量和引号 …… 42
3.1.6 变量的作用域 …… 42
3.1.7 系统变量 …… 45
3.1.8 环境变量 …… 47
3.2 变量的赋值和清空 …… 48
3.2.1 变量的赋值 …… 49
3.2.2 引用变量的值 …… 49
3.2.3 清除变量 …… 50
3.3 引用和替换 …… 51
3.3.1 引用 …… 51
3.3.2 全引用 …… 52
3.3.3 部分引用 …… 52
3.3.4 命令替换 …… 53
3.3.5 转义 …… 54
3.4 小结 …… 54
3.5 习题 …… 54
第 4 章 条件测试和判断语句 …… 56
4.1 条件测试 …… 56
4.1.1 条件测试的基本语法 …… 56
4.1.2 字符串测试 …… 57
4.1.3 整数测试 …… 60
4.1.4 文件测试 …… 62
4.1.5 逻辑操作符 …… 65
4.2 条件判断语句 …… 66
4.2.1 使用简单的 if 语句进行条件判断 …… 66
4.2.2 使用 if…else 语句进行流程控制 …… 69

4.2.3 使用 if…elif 语句进行多条件判断 71
4.2.4 使用 exit 语句退出程序 72
4.3 多条件判断语句 case 74
4.3.1 case 的基本语法 74
4.3.2 利用 case 语句处理选项参数 75
4.3.3 利用 case 语句处理用户的输入 77
4.4 运算符 78
4.4.1 算术运算符 78
4.4.2 位运算符 82
4.4.3 自增或自减运算符 84
4.4.4 数字常量的进制 85
4.5 小结 86
4.6 习题 86

第 5 章 循环结构 88
5.1 步进循环语句 for 88
5.1.1 带列表的 for 循环语句 88
5.1.2 不带列表的 for 循环语句 93
5.1.3 类 C 风格的 for 循环语句 93
5.1.4 使用 for 循环语句处理数组 95
5.2 until 循环语句 96
5.2.1 until 语句的基本语法 96
5.2.2 利用 until 语句批量增加用户 97
5.3 while 循环语句 99
5.3.1 while 语句的基本语法 99
5.3.2 通过计数器控制 while 循环结构 99
5.3.3 通过结束标记控制 while 循环结构 100
5.3.4 理解 while 语句与 until 语句的区别 101
5.4 嵌套循环 102
5.5 利用 break 和 continue 语句控制循环 103
5.5.1 利用 break 语句控制循环 103
5.5.2 利用 continue 语句控制循环 105
5.5.3 分析 break 语句和 continue 语句的区别 106
5.6 小结 109
5.7 习题 109

第 6 章 函数 111
6.1 函数的基础知识 111
6.1.1 什么是函数 111
6.1.2 函数的定义 112

6.1.3 函数的调用 113
6.1.4 函数链接 114
6.1.5 函数的返回值 115
6.1.6 函数和别名 117
6.1.7 全局变量和局部变量 118
6.2 函数的参数 120
6.2.1 包含参数的函数的调用方法 120
6.2.2 获取函数参数的个数 121
6.2.3 通过位置变量接收参数值 122
6.2.4 移动位置参数 122
6.2.5 通过 getopts 接收函数的参数 123
6.2.6 传递间接参数 124
6.2.7 通过全局变量传递数据 126
6.2.8 传递数组参数 126
6.3 函数库文件 128
6.3.1 函数库文件的定义 128
6.3.2 函数库文件的调用 129
6.4 递归函数 130
6.5 小结 132
6.6 习题 133

第 7 章 数组 134

7.1 定义数组 134
7.1.1 通过指定元素值定义数组 134
7.1.2 通过 declare 语句定义数组 135
7.1.3 通过元素值集合定义数组 136
7.1.4 通过键值对定义数组 137
7.1.5 数组和普通变量 138
7.2 数组的赋值 139
7.2.1 按索引为元素赋值 139
7.2.2 通过集合为数组赋值 140
7.2.3 在数组末尾追加新元素 141
7.2.4 通过循环为数组元素赋值 142
7.3 访问数组 142
7.3.1 访问第 1 个数组元素 142
7.3.2 通过下标访问数组元素 143
7.3.3 计算数组的长度 143
7.3.4 通过循环遍历数组元素 145
7.3.5 引用所有的数组元素 145
7.3.6 以切片方式获取部分数组元素 146

7.3.7 数组元素的替换 148
7.4 删除数组 149
7.4.1 删除指定的数组元素 149
7.4.2 删除整个数组 150
7.5 数组的其他操作 151
7.5.1 复制数组 151
7.5.2 连接数组 151
7.5.3 将文件内容加载到数组中 152
7.6 小结 153
7.7 习题 153

第 8 章 正则表达式 154
8.1 正则表达式简介 154
8.1.1 为什么使用正则表达式 154
8.1.2 如何学习正则表达式 155
8.1.3 如何实践正则表达式 156
8.2 正则表达式基础 156
8.2.1 正则表达式的原理 156
8.2.2 标准正则表达式 157
8.2.3 扩展正则表达式 161
8.2.4 Perl 正则表达式 163
8.2.5 正则表达式的字符集 164
8.3 正则表达式的应用 165
8.3.1 匹配单个字符 166
8.3.2 匹配多个字符 168
8.3.3 匹配字符串的开头或者结尾 170
8.3.4 运算符的优先级 171
8.3.5 子表达式 172
8.3.6 通配符 174
8.4 grep 命令 175
8.4.1 grep 命令的基本语法 175
8.4.2 grep 命令族简介 176
8.5 小结 177
8.6 习题 177

第 9 章 文本处理 178
9.1 使用 echo 命令输出文本 178
9.1.1 显示普通字符串 178
9.1.2 显示转义字符 179
9.1.3 显示变量 181

9.1.4 换行和不换行 182
9.1.5 显示命令的执行结果 183
9.1.6 echo 命令的执行结果重定向 183
9.2 文本格式化的输出 184
9.2.1 使用 UNIX 制表符 184
9.2.2 使用 fold 命令格式化行 185
9.2.3 使用 fmt 命令格式化段落 187
9.2.4 使用 rev 命令反转字符顺序 189
9.2.5 使用 pr 命令格式化文本页 190
9.3 使用 sort 命令对文本进行排序 193
9.3.1 sort 命令的基本用法 193
9.3.2 使用单个关键字进行排序 194
9.3.3 根据指定的列进行排序 198
9.3.4 根据关键字进行降序排序 198
9.3.5 数值列的排序 200
9.3.6 自定义列分隔符 201
9.3.7 删除重复的行 202
9.3.8 根据多个关键字进行排序 202
9.3.9 使用 sort 命令合并文件 204
9.4 文本的统计 205
9.4.1 输出包含行号的文本行 205
9.4.2 统计行数 207
9.4.3 统计单词数和字符数 209
9.5 使用 cut 命令选取文本列 209
9.5.1 cut 命令及其语法 210
9.5.2 选择指定的文本列 211
9.5.3 选择指定数量的字符 212
9.5.4 排除不包含列分隔符的行 213
9.6 使用 paste 命令拼接文本列 214
9.6.1 paste 命令及其语法 214
9.6.2 自定义列分隔符 216
9.6.3 拼接指定的文本列 216
9.7 使用 join 命令连接文本列 217
9.7.1 join 命令及其语法 217
9.7.2 指定连接关键字列 219
9.7.3 内连接文本文件 220
9.7.4 左连接文本文件 220
9.7.5 右连接文本文件 221
9.7.6 全连接文本文件 222
9.7.7 自定义输出列 222

9.8 使用 tr 命令替换文件内容 223
9.8.1 tr 命令及其语法 223
9.8.2 去除重复出现的字符 224
9.8.3 删除空行 225
9.8.4 大小写转换 225
9.8.5 删除指定的字符 226
9.9 小结 227
9.10 习题 227

第 10 章 流编辑器 229
10.1 sed 命令简介 229
10.1.1 sed 命令的基本语法 229
10.1.2 sed 命令的工作方式 231
10.1.3 使用行号定位文本行 231
10.1.4 使用正则表达式定位文本行 232
10.2 sed 命令的常用操作 233
10.2.1 sed 命令的基本语法 233
10.2.2 选择文本 234
10.2.3 替换文本 236
10.2.4 删除文本 239
10.2.5 追加文本 242
10.2.6 插入文本 243
10.3 组合命令 243
10.3.1 使用-e 选项执行多个子命令 244
10.3.2 使用分号执行多个子命令 244
10.3.3 对一个地址使用多个子命令 245
10.3.4 sed 脚本文件 246
10.4 小结 248
10.5 习题 249

第 11 章 文本处理利器 awk 命令 250
11.1 awk 命令简介 250
11.1.1 awk 命令的功能 250
11.1.2 awk 命令的基本语法 251
11.1.3 awk 命令的工作流程 252
11.1.4 执行 awk 命令的几种方式 252
11.2 awk 命令的模式匹配 254
11.2.1 关系表达式 254
11.2.2 正则表达式 255
11.2.3 混合模式 256

11.2.4 区间模式 256
11.2.5 BEGIN 模式 257
11.2.6 END 模式 258
11.3 变量 259
11.3.1 变量的定义和引用 259
11.3.2 系统内置变量 260
11.3.3 记录分隔符和字段分隔符 260
11.3.4 记录和字段的引用 263
11.4 运算符和表达式 264
11.4.1 算术运算符 264
11.4.2 赋值运算符 265
11.4.3 条件运算符 266
11.4.4 逻辑运算符 266
11.4.5 关系运算符 267
11.4.6 其他运算符 268
11.5 函数 268
11.5.1 字符串函数 268
11.5.2 算术函数 272
11.6 数组 273
11.6.1 数组的定义和赋值 273
11.6.2 遍历数组 274
11.7 流程控制 276
11.7.1 if 语句 276
11.7.2 while 语句 277
11.7.3 do…while 语句 278
11.7.4 for 语句 279
11.7.5 break 语句 280
11.7.6 continue 语句 281
11.7.7 next 语句 282
11.7.8 exit 语句 283
11.8 awk 命令格式化的输出 283
11.8.1 基本的 print 语句 283
11.8.2 格式化输出 printf()函数 283
11.8.3 使用 sprintf()函数生成格式化字符串 284
11.9 awk 命令与 Shell 的交互 285
11.9.1 通过管道实现与 Shell 的交换 285
11.9.2 通过 system()函数实现与 Shell 的交互 286
11.10 小结 287
11.11 习题 287

第 12 章　文件操作 …… 288
12.1　文件的基础知识 …… 288
12.1.1　列出文件 …… 288
12.1.2　文件的类型 …… 289
12.1.3　文件的权限 …… 292
12.2　查找文件 …… 293
12.2.1　find 命令及其语法 …… 293
12.2.2　find 命令——路径 …… 294
12.2.3　find 命令——测试 …… 295
12.2.4　find 命令——使用!运算符对测试求反 …… 298
12.2.5　find 命令——处理文件权限错误信息 …… 298
12.2.6　find 命令——动作 …… 299
12.3　比较文件 …… 300
12.3.1　使用 comm 比较文件 …… 301
12.3.2　使用 diff 比较文件 …… 304
12.4　文件描述符 …… 307
12.4.1　什么是文件描述符 …… 307
12.4.2　标准输入、标准输出和标准错误 …… 308
12.5　重定向 …… 309
12.5.1　输出重定向（覆盖） …… 309
12.5.2　输出重定向（追加） …… 311
12.5.3　输入重定向 …… 311
12.5.4　当前文档 …… 312
12.5.5　重定向两个文件描述符 …… 313
12.5.6　使用 exec 命令分配文件描述符 …… 313
12.6　小结 …… 315
12.7　习题 …… 315

第 13 章　子 Shell 与进程处理 …… 317
13.1　子 Shell …… 317
13.1.1　什么是子 Shell …… 317
13.1.2　内部命令、保留字和外部命令 …… 318
13.1.3　在子 Shell 中执行命令 …… 321
13.1.4　把子 Shell 中的变量值传回父 Shell …… 325
13.2　进程处理 …… 327
13.2.1　什么是进程 …… 327
13.2.2　通过脚本监控进程 …… 328
13.2.3　作业控制 …… 329
13.2.4　信号与 trap 命令 …… 332
13.3　小结 …… 334

13.4 习题 334

第 3 篇 Shell 编程实战

第 14 章 Shell 脚本调试技术 336
14.1 Shell 脚本中的常见错误 336
14.1.1 常见的语法错误 336
14.1.2 常见的逻辑错误 339
14.2 Shell 脚本调试技术 340
14.2.1 使用 echo 命令调试脚本 340
14.2.2 使用 trap 命令调试 Shell 脚本 341
14.2.3 使用 tee 命令调试 Shell 脚本 343
14.2.4 使用调试钩子调试 Shell 脚本 344
14.3 小结 346
14.4 习题 346

第 15 章 利用 Shell 脚本解决实际问题 347
15.1 编写系统服务脚本 347
15.1.1 系统的启动过程 347
15.1.2 运行级别 348
15.1.3 服务脚本的基本语法 349
15.1.4 编写 MySQL 服务脚本 352
15.2 通过脚本管理 Apache 服务器日志 358
15.2.1 Apache 日志简介 359
15.2.2 归档文件名生成函数 360
15.2.3 过期日志归档函数 361
15.2.4 过期日志删除函数 362
15.2.5 日志归档主程序 362
15.2.6 定时运行日志归档脚本 363
15.3 小结 367
15.4 习题 367

第 1 篇
认识 Shell 编程

⏭ 第 1 章　Shell 入门基础

⏭ 第 2 章　Shell 编程环境的搭建

第 1 章　Shell 入门基础

随着 Linux 和 UNIX 的广泛应用，Shell 日益成为系统管理员一个非常重要的工具。作为一名优秀的系统管理员或者 Linux/UNIX 开发者，熟练掌握 Shell 程序设计可以使工作达到事半功倍的效果。

本章从最基本的 Shell 概念入手，依次介绍 Shell 的特性、如何向 Shell 脚本传递参数，然后通过一个最简单的例子来说明如何进行 Shell 程序设计。

本章涉及的主要知识点如下：

- 为什么学习和使用 Shell 编程：主要介绍 Shell 在日常管理工作中的重要作用。
- 什么是 Shell：主要介绍 Shell 的基本概念、起源、功能和分类等。
- 作为程序设计语言的 Shell：主要介绍什么是交互式程序，如何创建脚本，以及如何设置可执行的脚本。
- 向脚本传递参数：主要介绍什么是脚本参数及脚本参数的用途等。
- 第一个 Shell 程序：通过一个简单的例子向读者介绍 Shell 脚本的基本元素、注释和风格，如何执行 Shell 程序，以及 Shell 程序的退出状态。

1.1　为什么学习和使用 Shell 编程

对于一个合格的系统管理员来说，学习和掌握 Shell 编程是非常重要的。通过编程，可以在很大程度上简化日常的维护工作，使管理员从简单的重复劳动中解脱出来。本节将介绍学习和使用 Shell 编程的重要性。

作为程序设计语言来说，Shell 是一种脚本语言。脚本语言是相对于编译型语言而言的，前者无须进行编译，而是由解释器读取程序代码并且执行其中的语句；后者则是预先编译成可执行代码，在使用的时候可以直接执行。

脚本语言的优点在于简单、易学，因此任何人在了解了基本的知识之后都可以毫不费力地编写出一个简单的脚本。关于这一点，最后会通过一个简单的例子来说明。虽然 Shell 非常容易上手，但是想要真正精通 Shell 编程却不是一件容易的事情。这是因为 Shell 的语法非常灵活，又涉及 Shell 的许多命令。想要真正透彻地了解 Shell 程序设计，必须下一番功夫才可以。

1.2　Shell 简介

在学习 Shell 编程之前，需要清楚什么是 Shell。为了使读者在学习具体的 Shell 编程之

前对 Shell 有一个基本的了解，本节将对 Shell 进行概括性的介绍，包括 Shell 的起源、功能和分类。

1.2.1　Shell 的起源

Shell 的起源与计算机世界里面最古老的操作系统 UNIX 有着密不可分的关系。1964 年，美国 AT&T 公司的贝尔实验室、麻省理工学院及美国通用电气公司共同参与研发了一套可以安装在大型主机上的多用户、多任务的操作系统，该操作系统的名称为 Multics（MULTiplexed Information and Computing System），运行在美国通用电气公司的大型机 GE-645 上面。由于整个目标过于庞大，糅合了太多的特性，Multics 虽然发布了一些产品，但是性能都很低，最终以失败告终。1969 年，AT&T 公司最终退出了 Multics 的开发。但是该公司其中的一位开发者肯·汤普逊（Kenneth Lane Thompson）继续为 GE-645 开发软件。

大约在 1970 年，另外一位开发者丹尼斯·里奇（Dennis MacAlistair Ritchie）也加入了汤普逊的开发队伍，如图 1-1 所示。在汤普逊和里奇的组织和领导下，他们启动了另外一个新的多用户、多任务的操作系统的项目，他们把这个项目称为 UNICS（Uniplexed Information and Computing System）。后来，人们取这个单词的谐音，把这个项目称为 UNIX。

最初的 UNIX 完全采用汇编语言编写，因此可移植性非常差。为了提高系统的可移植性和开发效率，汤普逊和里奇于 1973 年使用 C 语言重新编写了 UNIX。通过这次编写，使得 UNIX 得以移植到其他小型机上面。

与此同时，第一个重要的标准 UNIX Shell 于 1979 年末在 UNIX 的第 7 版中推出，并以作者史蒂夫•伯恩（Stephen Bourne）的名字命名，叫作 Bourne Shell，简称为 sh。Bourne Shell 当时主要用于系统管理任务的自动化。此后，Bourne Shell 凭借其简单和高效的特点广受欢迎，很快就成为流行的 Shell。虽然 Bourne Shell 广受欢迎，但是其缺少一些交互功能，如命令作业控制、历史和别名等。

而在这段时期，UNIX 的另外一个著名分支 BSD UNIX 也悄然兴起。随着 BSD 的风头正劲，另一个老牌 Shell 也登场了，它就是比尔•乔伊（Bill Joy）在加州大学伯克利分校读书期间开发的 C Shell。C Shell 开发于 20 世纪 70 年代末，作为 BSD UNIX 系统的一部分发布，简称 csh。乔伊是美国 SUN 公司的创始人之一，他在伯克利分校时主持开发了最早版本的 BSD，如图 1-2 所示。

图 1-1　汤普逊和里奇

图 1-2　比尔·乔伊

C Shell 基于 C 语言开发，作为编程语言使用时，其语法类似 C 语言，所以程序员可能会很喜欢它。此外，C Shell 还提供了增强交互使用的功能，如作业控制、命令行历史和别名等。当然，C Shell 的缺点和其优点一样明显，由于它是为大型机设计的并增加了很多新功能，所以 C Shell 在小型机上的运行比较慢。更为麻烦的是，即使在大型机上，C Shell 的速度也不如 Bourne Shell，而这一点在当时的硬件条件下可以说是致命的弱点。

C Shell 之后又出现了许多其他 Shell 程序，主要包括 Tenex C Shell（tcsh）、Korn Shell（ksh）及 GNU Bourne-Again shell（bash），这些 Shell 的特点不再详细介绍。

说明：目前，无论在 UNIX 系统还是在 Linux 系统中，比较流行的 Shell 程序都是 bash。

1.2.2　Shell 的功能

Shell 这个单词的意思是“外壳”，它形象地表达出了 Shell 的作用。在 UNIX 及 Linux 中，Shell 就是套在内核外面的一层外壳，如图 1-3 所示。正因为有 Shell 的存在，才向普通的用户隐藏了许多关于系统内核的细节。

Shell 又称命令解释器，它能识别用户输入的各种命令，并传递给操作系统。它的作用与 Windows 操作系统中的命令行类似，但是，Shell 的功能远比命令行强大得多。在 UNIX 或者 Linux 中，Shell 既是用户交互的界面，也是控制系统的脚本语言。

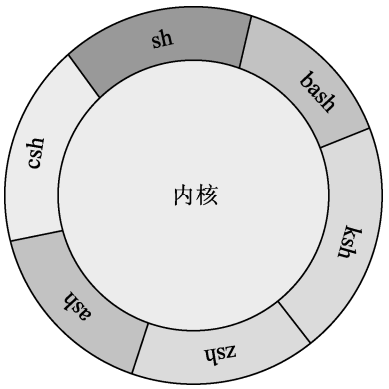

图 1-3　UNIX/Linux Shell 示意

1.2.3　Shell 的分类

关于 Shell 的分类，在介绍 Shell 的起源时已经简单介绍过一些了，下面对各种 Shell 程序做一个简单概括。常见的几种 Shell 程序如下：

- Bourne Shell：标识为 sh，该 Shell 由 Stephen Bourne 在贝尔实验室时编写。在许多 UNIX 系统中，该 Shell 是 root 用户默认的 Shell。
- Bourne-Again Shell：标识为 bash，该 Shell 是 Brian Fox 在 1987 年编写的，它是绝大多数 Linux 发行版默认的 Shell。
- Korn Shell：标识为 ksh，该 Shell 由贝尔实验室的 David Korn 在 20 世纪 80 年代早期编写。它完全向上兼容 Bourne Shell 并包含 C Shell 的很多特性。
- C Shell：标识为 csh，该 Shell 由 Bill Joy 在 BSD 系统上开发。由于其语法与 C 语言类似，因此称为 C Shell。
- Z Shell：标识为 zsh，该 Shell 是一款交互式使用的 Shell，由 Paul Falstad 在 1990 年开发。Z Shell 包含 bash、ksh、tcsh 等其他 Shell 中许多优秀的功能，也拥有诸多自身的特色。

以上 Shell 程序，其语法或多或少都有所区别。目前仍然建议使用标准的 Bourne-Again Shell。

1.3　Shell 的特性

Shell 不仅充当用户与 UNIX 或者 Linux 交互界面的角色，而且可以作为一种程序设计语言来使用。通过 Shell 编程，可以实现许多非常实用的功能，提高系统管理的自动化水平。本节介绍作为程序设计语言的 Shell 的一些特性。

1.3.1　交互式程序

现在读者已经对 Shell 有了初步的了解，接下来将会逐步接触到真正的 Shell 脚本程序。通常情况下，Shell 脚本程序有以下两种执行方式：

- 用户可以依次输入一系列命令，交互式地执行；
- 用户也可以把所有命令按照顺序保存在一个文件中，然后将该文件作为一个程序来执行。

下面首先介绍如何交互式地执行 Shell 程序。

在命令行上直接输入命令来交互式地执行 Shell 脚本是一种非常简单的方式。尤其是在测试 Shell 程序的时候，通过这种交互式方式，可以非常方便地得到程序执行的结果。

【例 1-1】　在当前目录下查找文件名中包含 xml 这 3 个字符的文件。如果找到，则在当前屏幕上打印出来。可以在 Shell 提示符后面依次输入下面的代码：

```
01  #----------------------------/chapter1/ex1-1.sh------------------
02  [root@linux ~]# for filename in `ls .`
03  > do
04  > if echo "$filename" | grep "xml"
05  > then
06  > echo "$filename"
07  > fi
08  > done
09  package.xml
10  package.xml
11  wbxml-1.0.3
12  wbxml-1.0.3
13  wbxml-1.0.3.tgz
14  wbxml-1.0.3.tgz
```

每当输入完一行代码时，都需要按 Enter 键换行。当输入完第 8 行代码之时，Shell 开始执行输入的代码，第 9～14 行是 Shell 程序的输出结果。从结果中可以得知，当前目录中有 3 个文件的文件名包含 xml 这 3 个字符。这里的输出结果显示有 6 个文件，是因为在代码中文件名输出了两次。其中：第一次是第 4 行代码，输出匹配的文件名；第二次是第 6 行代码，又输出了一次文件名。

注意：当 Shell 期待用户下一步的输入时，正常的 Shell 提示符“#”将会变为“>”。用户可以一直输入下去，由 Shell 来判断何时输入完毕并立即执行程序。

关于上面的程序所涉及的语法，将在后面的内容中介绍。在此读者只要掌握交互式执行程序的方法即可。

虽然上面的执行方法非常方便快捷，但是每次在执行同一个程序的时候都要重新输入一次非常麻烦。此外，如果对程序不是很清楚的情况下则容易发生输入错误，导致程序不能执行。因此，在实际开发中，这种交互式执行程序的方式并不常见，一般是将这些语句写入一个脚本文件中作为一个程序来执行。

1.3.2　创建脚本

对于一组需要经常重复执行的 Shell 语句，将它们保存在一个文件中来执行是一种非常明智的做法。通常称这种包含多个 Shell 语句的文件为 Shell 脚本，或者 Shell 脚本文件。脚本文件都是普通的文本文件，可以使用任何文本编辑器查看或者修改其中的内容。

【例 1-2】　使用 vi 命令创建 Shell 脚本文件。在 Shell 命令行中输入 vi 命令，然后输入以下代码：

```
#----------------------------/chapter1/ex1-2.sh------------------
#! /bin/bash

#for 循环开始
for filename in `ls .`
do
   #如果文件名包含 xml
   if echo "$filename" | grep "xml"
   then
      #输出文件名
      echo "$filename"
   fi
done
```

从上面的代码中可以得知，Shell 程序中的注释以“#”符号开始，一直持续到该行结束。请注意第一行#!/bin/bash，它是一种特殊形式的注释。其中，“#!”字符告诉系统同一行中紧跟在它后面的那个参数是用来执行本文件的程序。在这个例子中，/bin/bash 是默认的 Shell 程序。

1.3.3　设置可执行脚本

当将脚本编辑完成之后，这个脚本还不能马上执行。在 Linux 中，当用户执行某个程序时，必须拥有该文件的执行权限。用户可以通过 ls -l 或者 ll 命令（该命令实际上是 ls -l 命令的别名）来查看文件的访问权限，其中，ll 命令只可以在 Linux 中使用。下面是 ll 命令的执行结果：

```
[root@linux chapter1]# ll
total 4
-rw-r--r--  1   root    root    116     Jun 24 23:11    ex1-2.sh
```

在上面的输出结果中，每一行都代表一个文件描述信息。一共包括 6 列，其中，第一列就是文件的访问权限。通常情况下，每个文件的访问权限都由 9 位组成，其中，前 3 位表示文件的所有者对该文件的访问权限，中间 3 位表示与所有者同组的其他用户对该文件的访问权限，最后 3 位表示其他组的用户对该文件的访问权限。

在每组权限中，用 3 个字母来表示 3 种不同的权限，其中，r 表示读取权限，w 表示

写入权限，x 表示执行权限。

可以发现，在上面的 ex1-2.sh 文件的权限描述中，任何用户都没有该文件的执行权限，因此该文件无法直接执行。

为了使用户拥有某个文件的执行权限，可以使用 chmod 命令（该命令是 change file mode bits 的简写，表示改变文件的模式比特位）。该命令的基本语法如下：

```
chmod [options] filename
```

其中，options 表示各种权限选项。用户可以使用 r、w 及 x 这 3 个字母分别表示读取（read）、写入（write）和执行（excute）的权限，也可以使用数字来表示权限。在数字模式下，4 表示读取权限，2 表示写入权限，1 表示执行权限。另外，用户可以指定执行权限授予的对象，其中，u 表示文件的所有者（user），g 表示所有者所属的组（group），o 表示其他（other）组的用户。在授予权限时，操作符加号“+”表示授予权限，减号“-”表示收回权限。

例如，下面的操作授予文件 ex1-2.sh 的所有者执行权限：

```
[root@linux chapter1]# chmod u+x ex1-2.sh
[root@linux chapter1]# ll
total 4
-rwxr--r--      1   root    root    116     Jun 24 23:11    ex1-2.sh
```

上面的权限也可以使用数字来表示，例如：

```
[root@linux chapter1]# chmod 744 ex1-2.sh
[root@linux chapter1]# ll
total 4
-rwxr--r--      1   root    root    116     Jun 24 23:11    ex1-2.sh
```

在上面的命令中，作为选项的 3 个数字“744”分别表示文件所有者、所有者所属的用户组及其他组的权限。其中，7 是由 4、2 和 1 这 3 个数字相加而得，4 表示读取权限。

在授予用户执行权限之后，就可以执行该脚本了：

```
[root@linux ~]# chapter1/ex1-1.sh
package.xml
package.xml
wbxml-1.0.3
wbxml-1.0.3
wbxml-1.0.3.tgz
wbxml-1.0.3.tgz
```

注意：777 是一个特殊的权限，表示所有的用户都可以读、写和执行该文件。许多用户为了操作方便，会直接将该权限授予某些文件。通常情况下这样的操作会带来安全隐患，因此在将该权限授予用户时一定要谨慎。

1.4　向脚本传递参数

许多情况下，Shell 脚本都需要接收用户的输入，根据用户输入的参数来执行不同的操作。本节介绍 Shell 脚本的参数，以及如何在脚本中接收参数。

1.4.1　Shell 脚本的参数

从命令行传递给 Shell 脚本的参数称为位置参数，这主要是因为 Shell 脚本会根据参数的位置来接收它们的值。在 Shell 脚本内部，用户可以通过一系列系统变量来获取参数。这些变量的名称都是固定的，并且非常简单，只用一个字符来表示。例如，$0 表示当前执行的脚本名称，$1 表示传递给脚本的第 1 个参数等。表 1-1 列出了常用的与参数传递有关的系统变量。

表 1-1　常用的与参数传递有关的系统变量

变　量　名	说　　明
$n	表示传递给脚本的第*n*个参数。例如，$1表示第1个参数，$2表示第2个参数……
$#	命令行参数的个数
$0	当前脚本的名称
$*	以“参数1 参数2 参数3……”的形式返回所有参数的值
$@	以“参数1”“参数2”“参数3”……的形式返回所有参数的值
$_	保存之前执行命令的最后一个参数

通过表 1-1 可知，Shell 的位置参数按照 0，1，2……的顺序从 0 开始编号。其中，0 表示当前执行的脚本名称，而 1 表示第 1 个参数。由单引号或者双引号引起来的字符串作为一个参数进行传递，传递时会去掉引号。

注意：对于包含空白字符或者其他特殊字符的参数，需要使用单引号或者双引号进行传递。

变量$@可以以“参数 1”“参数 2”“参数 3”……的形式返回所有参数的值，因此，$@与“$1”“$2”“$3”…是等价的。如果传递的参数中包含空格或者其他特殊字符，需要使用$@来获取所有参数的值，不能使用$*。

变量$*以“参数 1 参数 2 参数 3……”的形式将所有参数作为一个字符串返回。通常情况下，参数值之间以空格、制表符或者换行符来隔开，默认情况下使用空格。

变量$#返回传递给脚本的参数的数量，不包括$0，即排除脚本的名称。

另外，如果传递的参数多于 9 个，则不能使用$10 来引用第 10 个参数。为了能够获取第 10 个参数的值，必须处理或保存第 1 个参数，即$1，然后使用 shift 命令删除参数 1 并将所有剩余的参数下移 1 位，此时$10 就变成了$9，以此类推。$#的值将被更新以反映参数的剩余数量。

【例 1-3】　传递脚本参数，代码如下：

```
#-----------------------------/chapter1/ex1-3.sh------------------
#! /bin/bash

echo "$# parameters"
echo "$@"
```

然后通过以下方式来执行：

```
[root@linux chapter1]# ./ex1-3.sh a "b c"
```

```
2 parameters
a b c
```

在上面的代码中，向 ex1-3.sh 脚本传递了两个参数，其中，第 2 个参数含有空格，因此需要使用双引号引起来。

1.4.2　参数的扩展

前面介绍了如何通过系统变量来获取脚本参数的值。对于简单的脚本，使用这个方法即可。因为可以通过变量$1、$2……依次获得全部参数，还可以通过$#获得参数的个数。但是在实践中遇到的并不总是这种简单的情况。例如，需要编写一个脚本程序，并且这个脚本程序需要一个拥有许多值的参数，在程序中，用户希望根据这个参数的值来执行不同的操作。在这种情况下，单纯地依靠$1 及$2 等变量已经不能满足需求了。此时可以考虑使用参数扩展。

如果接触过 UNIX 或者 Linux，那么对 ls 命令不会陌生。ls 命令可能是 UNIX 或者 Linux 系统中选项最多的命令了。例如，可以使用-l 选项以长格式的方式显示当前目录的内容。实际上，这个-l 也是 ls 命令的一个参数。这个参数与前面介绍的参数的不同之处在于它拥有一个前导的连字符“-”。

可以在 Shell 脚本中使用同样的技术，这称为参数扩展。为了获取（get）到这些参数（options）的值，需要在 Shell 程序中使用 getopts 命令。

【例 1-4】　参数扩展，代码如下：

```
#----------------------------/chapter1/ex1-4.sh------------------
#!/bin/bash

#输出参数索引
echo "OPTIND starts at $OPTIND"
#接收参数
while getopts ":pq:" optname
   do
   case "$optname" in
      "p")
         echo "Option $optname is specified"
         ;;
      "q")
         echo "Option $optname has value $OPTARG"
         ;;
      "?")
         echo "Unknown option $OPTARG"
         ;;
      ":")
         echo "No argument value for option $OPTARG"
         ;;
      *)
         # Should not occur
         echo "Unknown error while processing options"
         ;;
   esac
   echo "OPTIND is now $OPTIND"
done
```

对上面的代码这里不进行过多的介绍，此处只是为了了解如何使用参数扩展。在代码

的第 7 行中，getopts 命令后面的双引号中的第一个冒号用于告诉 getopts 命令忽略一般的错误消息，因为此脚本将提供它自己的错误处理。p 和 q 则是两个选项名称。选项后面的冒号表示该选项需要一个值。例如，在绝大部分命令中，-f 选项可能需要一个文件（file）名。

当找到某个选项时，getopts 命令返回 true。第二个参数是变量名 optname，该变量用于接收找到的选项的名称。以上程序的执行结果如下：

```
[root@linux chapter1]# ./ex1-4.sh -p
OPTIND starts at 1
Option p is specified
OPTIND is now 2
[root@linux chapter1]# ./ex1-4.sh -q
OPTIND starts at 1
No argument value for option q
OPTIND is now 2
[root@linux chapter1]# ./ex1-4.sh -f
OPTIND starts at 1
Unknown option f
OPTIND is now 2
```

1.5　第一个 Shell 程序

通过前面几节的学习，读者已经接触到一些 Shell 程序了。本节介绍一个完整的 Shell 程序，使读者能够掌握 Shell 程序的组成元素并写出简单的程序。

1.5.1　Shell 脚本的基本元素

在学习任何程序设计语言的时候，“Hello world!”是一个必不可少的例子。接下来介绍如何在 Shell 语言中输出 Hello, Bash Shell!。

【例 1-5】　输出 Hello, Bash Shell!，代码如下：

```
01  #-----------------------------/chapter1/ex1-5.sh------------------
02  #! /bin/bash
03
04  #输出字符串
05  echo "Hello, Bash Shell!"
```

上面是一个完整的 Shell 程序，对于拥有执行权限的用户来说，这也是一个可执行的 Shell 程序。上面的代码非常简单，实际上最主要的代码只有一行，即第 5 行，这一行的作用是在控制台上面输出一行消息。

接下来执行这个程序，看看到底会出现什么结果：

```
[root@linux chapter1]# chmod +x ex1-5.sh
[root@linux chapter1]# ./ex1-5.sh
Hello, Bash Shell!
```

从上面的执行结果中可以得知，这个程序已经得到了预期的结果。但是，读者可能会有疑问，作为一个 Shell 程序，它应该具备哪些元素呢？分析上面的例子可以得知，一个最基本的 Shell 程序应该拥有第 2 行的代码：

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

关于上面这一行代码的作用，在后面的内容中会详细介绍。另外，第 4 行是注释，用来说明下面的代码的功能。第 5 行是 echo 语句，其是实现整个程序功能的主要代码。

因此，一个基本的 Shell 程序应该包括以下基本元素：

- 第 2 行的“#! /bin/bash”。
- 注释：说明某些代码的功能。
- 可执行语句：实现程序的功能。

在接下来的内容中将依次介绍这些基本元素。

1.5.2　指定命令解读器

当用户在命令行中执行一个脚本程序的时候，Shell 首先会判断用户是否拥有该程序的执行权限。如果没有执行权限，Shell 则会给出 Permission denied 的提示；否则，Shell 会创建一个新的进程，解释并执行 Shell 程序中的语句。

但是，无论在 UNIX 还是在 Linux 中，通常会同时安装多个 Shell 程序，如 sh、bash 或者 csh 等。而这些不同的 Shell 程序的语法有一些区别，那么到底使用哪个 Shell 来执行代码呢？

实际上，在例 1-5 中，第 2 行代码的作用是告诉当前的 Shell，应该调用哪个 Shell 来执行当前的程序。

当用户在命令行中执行该程序时，当前的 Shell 会载入该程序的代码，并且读取其中的第 2 行。如果发现有“#!”标识，则表示当前的程序指定了解释并执行它的 Shell。然后会尝试读取“#!”标识后面的内容，搜寻解释器的绝对路径。如果发现指定的解释器，则会创建一个关于该解释器的进程，解释并执行当前脚本的语句。在例 1-5 中，当前的 Shell 会创建/bin/bash 的进程来执行 ex1-5.sh 脚本文件中的语句。

注意：*用户应该在“#!”标识后面指定解释器的绝对路径。*

Shell 脚本的这个规定使得用户可以非常灵活地调用任何解释器，而不仅限于 Shell 程序。下面介绍如何在脚本文件中指定其他解释器程序。

【例 1-6】 在 PHP 脚本文件中指定 PHP 语言的解释器，然后执行文件中的 PHP 代码，代码如下：

```
01  #-----------------------------/chapter1/ex1-6.php------------------
02  #指定解释器
03  #! /usr/local/bin/php
04
05  <?php
06     //输出 Hello world!字符串
07     print "Hello world!";
08  ?>
```

在上面的代码中，第 3 行指定解释当前文件的解释器的绝对路径，第 5 行是 PHP 代码的开始标识符，第 7 行使用 print 语句输出字符串“Hello world!”，第 8 行是 PHP 代码的结束标识符。

程序的执行结果如下：

```
[root@linux chapter1]# ./ex1-6.php
Hello world!
```

用户还可以指定其他命令，如 more 或者 cat 来显示当前程序的代码，请参见下面的例子。

【例 1-7】　指定 more 命令作为脚本文件的解释器，代码如下：

```
#-----------------------------/chapter1/ex1-7.sh------------------
#指定解释器
#! /bin/more

#输出语句
echo "Hello world!"
```

读者应该想象得到该程序的执行结果。没错，该程序的执行结果是调用 more 命令来显示当前程序的代码而不是执行程序中的代码本身。例如：

```
[root@linux chapter1]# ./ex1-7.sh
#! /bin/more

echo "Hello world!"
```

到此为止，读者对“#!”标识的作用已有了比较深入的理解。从本质上讲，该标识的作用就是指定解释当前脚本文件的程序，至于最后的结果是什么样，还要看指定的程序。如果指定的是 Shell 或者某些程序语言的解释器，如/usr/local/bin/php，则会执行其中的代码；如果是其他一些程序，如/bin/more，则会显示当前脚本文件的内容。

1.5.3　Shell 脚本的注释和风格

通过在代码中增加注释可以提高程序的可读性。传统的 Shell 只支持单行注释，其表示方法是一个井号“#”，从该符号开始一直到行尾都属于注释的内容。例如例 1-5 中的第 4 行：

```
#输出字符串
```

如果需要注释多行内容，则在每行注释的开头都要加上“#”，例如：

```
#注释 1
#注释 2
#注释 3
…
```

但是这并不意味着只能使用单行注释。实际上，还可以通过其他变通的方法来实现多行注释，其中，最简单的方法就是使用冒号“:”配合 here document，其语法如下：

```
:<<BLOCK

....注释内容

BLOCK
```

【例 1-8】　通过 here document 实现多行注释，代码如下：

```
#-----------------------------/chapter1/ex1-8.sh------------------
#! /bin/bash

:<<BLOCK
```

```
05   本脚本的作用是输出一行字符串
06   作者：chunxiao
07   BLOCK
08   echo "Hello world!"
```

注意：一个 here document 就是一段带有特殊目的的代码段，它使用 I/O 重定向的形式将一个命令序列传递到一个交互程序或者命令中，如 ftp、cat 或者 ex 文本编辑器。在例 1-8 中，我们是将 BLOCK 之间的代码重定向到一个不存在的命令中，从而间接地实现了多行注释。

1.5.4　如何执行 Shell 程序

在 1.3.3 节中我们介绍了如何使程序变得可执行，那就是修改脚本文件的访问权限。实际上，在 Linux 中，如果要执行某个 Shell 程序，可以通过 3 种方式来实现。这 3 种方式分别为：

- 授予用户执行 Shell 脚本文件的权限，使得该程序能够直接执行。
- 通过调用 Shell 脚本解释器来执行 Shell 程序。
- 通过 source 命令来执行 Shell 程序。

第一种方式前面已经详细介绍过了，不再重复说明。第二种方式就是将脚本文件作为参数传递给解释器。通过这种方式执行脚本时，不需要用户拥有执行该脚本文件的权限，只要拥有读取该脚本文件的权限即可。

对于例 1-5，可以使用以下方式来执行：

```
[root@linux chapter1]# /bin/bash ex1-5.sh
Hello, Bash Shell!
```

在上面的命令中，/bin/bash 是 bash Shell 的绝对路径。用户首先调用 bash，然后 bash 会载入 ex1-5.sh 并且解释其中的语句，最后给出程序的执行结果。

因此，对于第二种方式，首先调用的是解释器，然后由解释器解释脚本文件。而第一种方式是直接在脚本文件中指定解释器，当前的 Shell 会自动调用指定的解释器，然后创建进程再执行脚本文件。所以，第一种方式和第二种方式在本质上是一样的。

source 命令是一个 Shell 内部命令，其功能是读取指定的 Shell 程序文件，并且依次执行其中所有的语句。该命令与前面两种方式的区别在于只是简单地读取脚本里的语句，并且依次在当前的 Shell 里执行，并没有创建新的子 Shell 进程。在脚本中创建的变量都会保存到当前的 Shell 里。

注意：由于 source 命令是在当前的 Shell 中执行脚本文件，因此其执行结果可能会与前面两种方式不同。

例如，例 1-5 也可以使用以下方式执行：

```
[root@linux chapter1]# source ex1-5.sh
Hello, Bash Shell!
```

1.5.5　Shell 程序的退出状态

在 UNIX 或者 Linux 中，每个命令都会返回一个退出状态码。退出状态码是一个整数，

其有效范围为 0～255。通常情况下，成功的命令返回 0，而不成功的命令返回非 0 值。非 0 值通常都被解释成一个错误码。运行良好的 UNIX 命令、程序和工具都会返回 0 表示成功，偶尔也会有例外。

同样，Shell 脚本中的函数和脚本本身也会返回退出状态码。在脚本或者是脚本函数中最后执行的命令会决定退出状态码。另外，用户也可以在脚本中使用 exit 语句将指定的退出状态码传递给 Shell。

在前面的所有例子中，我们都没有通过 exit 语句退出程序。在这种情况下，整个程序的退出状态码由最后执行的那一条语句来决定。例如，在下面的脚本中，整个脚本的退出状态将由 statement_last 这条语句的退出状态来决定。

```
01  #!/bin/bash
02
03  statement1
04
05  …
06
07  #将由最后的命令来决定退出状态码
08  statement_last
```

另外，在 Shell 中，系统变量$?保存了最后一条命令的退出状态。因此，上面的程序与下面的程序的效果是完全相同的：

```
01  #!/bin/bash
02
03  statement1
04
05  …
06
07  #将由最后的命令来决定退出状态码
08  statement_last
09  exit $?
```

其中，第 9 行的$?保存了最后（last）一条语句 statement_last 的退出状态（statement）。当然，exit 语句也可以不带任何参数。此时，脚本的退出状态也由最后一条语句的退出状态决定。所以，上面的程序与下面的程序的效果也是完全相同的：

```
01  #!/bin/bash
02
03  statement1
04
05  …
06
07  #由最后的命令决定退出状态码
08  statement_last
09  exit
```

程序的退出状态非常重要，它反映了脚本执行是否成功。用户可以根据脚本的执行状态来决定下一步的操作。

【例 1-9】　演示在不同的情况下程序的退出状态，代码如下：

```
01  #----------------------------/chapter1/ex1-9.sh-----------------
02  #!/bin/bash
03
04  echo "hello world"
05  #退出状态为 0，因为命令执行成功
06  echo $?
```

```
07  #无效命令
08  abc
09  #非 0 的退出状态，因为命令执行失败
10  echo $?
11  echo
12  #返回 120 退出状态给 shell
13  exit 120
```

在上面的代码中，第 4 行是一个正常的 echo 语句，因此第 6 行的输出结果应该是 0。第 8 行是一个无效的命令，因此第 10 行会输出一个非 0 值，具体是什么值要看当前 Shell 的设置。第 11 行是一个正常的 echo 语句，同样该语句的退出状态也是 0。第 13 行通过 exit 语句将退出状态码 120 返回给当前的 Shell。

例 1-9 的执行结果如下：

```
01  [root@linux chapter1]# ./ex1-9.sh
02  hello world
03  0
04  ./ex1-9.sh: line 8: abc: command not found
05  127
06
07  [root@linux chapter1]# echo $?
08  120
```

在上面的执行结果中，第 2 行是第 4 行 echo 语句的执行结果。第 3 行的 0 是例 1-9 中第 6 行的 echo 语句的退出状态码。第 4 行的错误信息是例 1-9 中第 8 行的无效命令给出的。第 5 行的 127 是上面的无效命令的退出状态码。由于程序已经退出，所以需要用户手动输入执行结果中的第 7 行命令，以获取整个脚本的退出状态码。从执行结果可以得知，该脚本的退出状态码为 120，这正是例 1-9 中第 13 行的 exit 语句返回的数值。

1.6　小　　结

本章主要介绍了与 Shell 程序设计有关的基础知识，包括为什么要学习和使用 Shell 程序设计、什么是 Shell、作为程序设计语言的 Shell 有哪些特点，以及 Shell 脚本的参数传递问题，最后介绍了一个非常简单的例子，用来说明 Shell 程序的基本组成元素和退出状态。本章的重点在于掌握好 Shell 程序的基本组成部分，以及如何执行 Shell 程序。在第 2 章将介绍 Shell 编程环境的搭建。

1.7　习　　题

一、填空题

1．Shell 这个单词的意思是_____，Shell 又称为_____，它能识别用户输入的_____，并传递给操作系统。

2．常见的 Shell 有_____、_____、_____、_____和_____。

3．Shell 脚本的第一行必须是_____。

二、选择题

1．在 Shell 脚本中，使用的注释符是（　　）。

A．//　　B．#　　C．$　　D．<-->

2．在 Shell 脚本参数中，（　　）参数可以获取脚本的名称。

A．$n　　B．$#　　C．$0　　D．$*

三、判断题

1．编写一个 Shell 脚本后，必须为该脚本添加可执行权限才可以执行。　　（　　）

2．Shell 脚本只支持单行注释。　　（　　）

四、操作题

1．创建一个简单的 Shell 脚本 test.sh，输出“Hello World!”。

2．为 Shell 脚本 test.sh 添加可执行权限并执行该脚本。

第 2 章　Shell 编程环境的搭建

与其他程序设计语言相比，Shell 的编程环境极其简单。通常情况下，只需要一个文本编辑器就可以开始 Shell 程序设计了。当然，如果有其他辅助性的工具，则会使得 Shell 编程更加简单。本章将介绍在不同的操作系统中如何搭建 Shell 编程环境，以及 Linux 中的文本编辑器的选择。

本章涉及的主要知识点如下：

- 在不同的操作系统上搭建 Shell 编程环境：主要介绍在 Windows、Linux 及 BSD 等常见的操作系统中如何搭建 Shell 编程环境。
- 编辑器的选择：主要介绍 Linux 中的图形化的文本编辑器、终端模拟器及非图形化的文本编辑器的使用方法。
- 系统环境搭建：主要介绍 Shell 配置文件和命令别名的使用方法。

2.1　在不同的操作系统上搭建 Shell 编程环境

虽然 Shell 程序一般都是在 UNIX 或者 Linux 等操作系统上运行的，但是开发者所使用的操作系统不一定是 UNIX 或者 Linux，完全有可能是 Windows 等其他操作系统。本节介绍在不同的操作系统中如何搭建 Shell 编程环境。

2.1.1　在 Windows 上搭建 Shell 编程环境

对于开发者来说，Windows 可能是最常用的操作系统了，因为 Windows 有着非常人性化的图形界面，可以大大提高开发者的开发效率。

如果想要在 Windows 上进行 Shell 编程，则需要安装一个 UNIX 模拟器。通过 UNIX 模拟器，在 Windows 上模拟出一个类似 UNIX 或者 Linux 的 Shell 环境。通过上面的介绍可以发现，模拟器与虚拟机非常相似，但是二者有着本质的区别。这是因为大部分模拟器仅在 Win32 系统中实现了 POSIX 系统调用的 API，而不是一个完整的操作系统；虚拟机则是虚拟出一台完整的机器，包括硬件，在虚拟机里安装的是一个完整的操作系统。

虽然与真正的 UNIX 或者 Linux 相比，模拟器实现的功能极其有限，但是对于简单的 Shell 开发来说，使用模拟器可以完成大部分的功能。在众多的模拟器中，最常用的是 Cygwin。

Cygwin 是一个非常优秀的 UNIX 模拟器，最初由 Cygnus Solutions 公司开发，目前由 Red Hat 公司维护。Cygwin 是许多自由软件的集合，用于在各种版本的 Microsoft Windows 上，创建出一个 UNIX 或者 Linux 的运行环境。Cygwin 的主要目的是通过重新编译，将

POSIX 系统（如 Linux、BSD 及 UNIX 系统）中的软件移植到 Windows 平台上。对于学习 Shell 程序设计的人来说，Cygwin 无疑是一个非常强大的工具。

可以从以下网站下载 Cygwin，编写本书时的最新版本是 3.4.7：

```
http://www.cygwin.com/
```

下载完成之后，可以按照以下步骤进行安装。

（1）双击安装程序 setup-x86_64.exe，弹出安装向导，如图 2-1 所示。

（2）单击“下一步”按钮，在弹出的对话框中选择安装类型，如图 2-2 所示。如果是第一次安装，应该选择第 1 项“从互联网安装”。选择好之后，此时会从网络上自动下载 Cygwin 的程序并且执行安装操作。

图 2-1　选择安装程序

图 2-2　选择安装类型

（3）单击“下一步”按钮，在弹出的对话框中选择安装目录，如图 2-3 所示。如果使用默认的安装目录，则可以直接单击“下一步”按钮；否则，单击“浏览”按钮，在弹出的对话框中选择想要安装的目标位置。

（4）单击“下一步”按钮，在弹出的对话框中选择安装包的存储目录，如图 2-4 所示。该目录用来保存安装程序从网络上下载的安装包文件。如果使用默认目录，则可以直接单击“下一步”按钮；否则，单击“浏览”按钮，在弹出的对话框中选择其他的位置。

图 2-3　选择安装位置

图 2-4　选择安装包的存储目录

（5）在弹出的对话框中选择网络连接的类型。由于安装程序需要从网络上下载安装包

文件，所以需要指定网络连接的类型，如图 2-5 所示。

☎**提示：**通常情况下可以选择第二项，即“直接连接”。如果需要使用代理服务器，则可以选择第一项或者第三项。

（6）单击“下一步”按钮，在弹出的对话框中选择下载软件包的网站，如图 2-6 所示。可以根据自己的实际情况选择从哪个网站下载安装包。通常情况下，国内网站的下载速度相对较快，因此可以选择第一项，即网易的镜像站点。

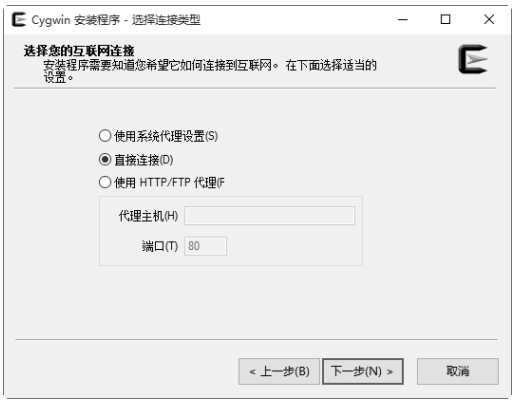

图 2-5　选择网络连接类型

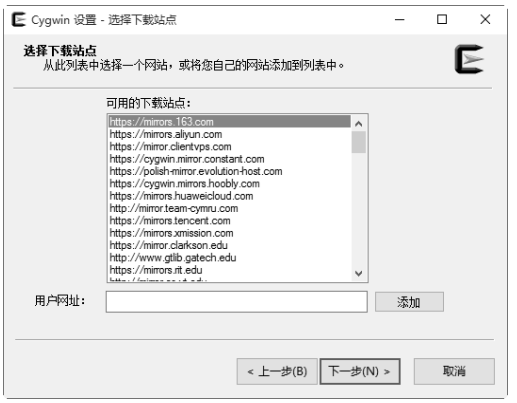

图 2-6　选择镜像站点

（7）单击“下一步”按钮，在弹出的对话框中选择软件包。Cygwin 本身是一些自由软件的集合，可以根据需要选择安装哪些软件包，如图 2-7 所示。

（8）选择好软件包之后，单击“下一步”按钮，从网站上下载软件并开始安装，如图 2-8 所示。

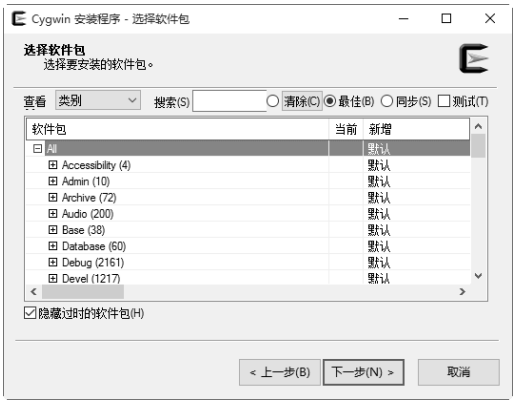

图 2-7　选择软件包

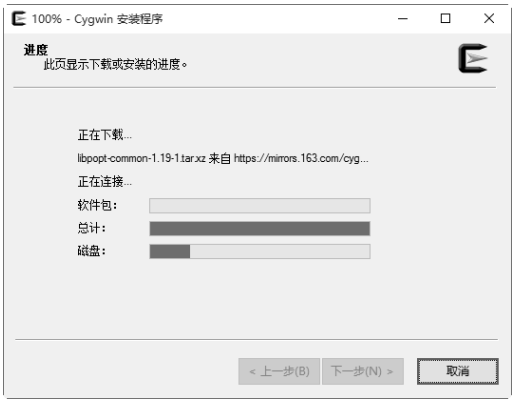

图 2-8　安装过程

（9）在所有的软件包都安装完成之后，会弹出如图 2-9 所示的对话框，单击“完成”按钮，退出安装向导。

在所有的安装操作都完成之后，单击桌面上的 Cygwin Terminal 图标，启动 Cygwin 模拟终端窗口，如图 2-10 所示。可以看出，Cygwin 的模拟终端窗口与真正的 UNIX 的终端窗口非常相似。此时可以在提示符后输入一些 Shell 命令。

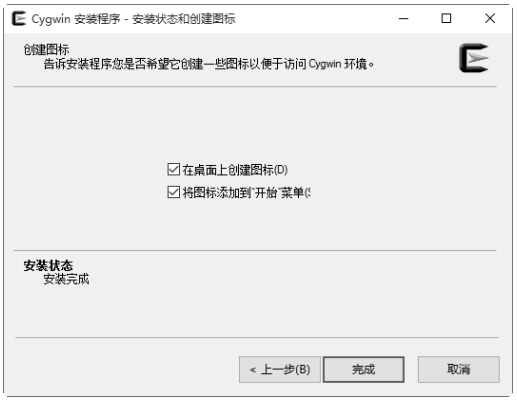

图 2-9　安装完成

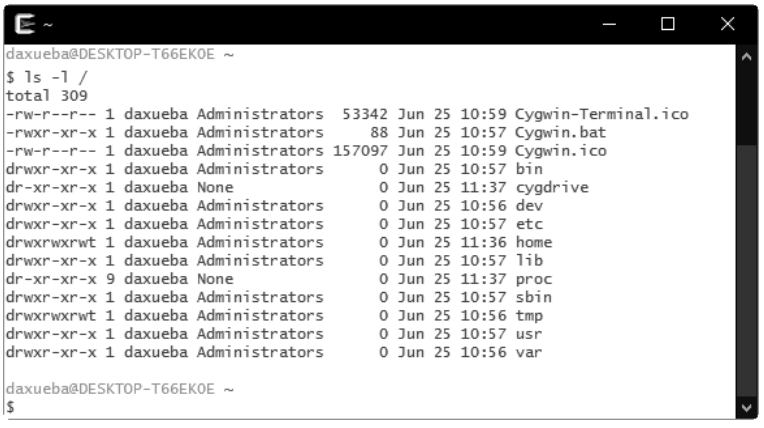

图 2-10　Cygwin 模拟终端窗口

为了验证能否在 Cygwin 的模拟环境中执行 Shell 程序，接下来尝试将第 1 章中的“Hello, Bash Shell!”程序在刚刚安装完成的环境中执行。执行结果如下：

```
$ ./ex1-5.sh
Hello, Bash Shell!
```

从上面的执行结果中可以看出，在 Cygwin 的模拟环境中，例 1-5 的执行结果与在 Linux 中的执行结果基本相同。

注意：在上面的第（7）步中选择的软件包会影响用户在 Cygwin 模拟环境中可以使用的命令。因此，在选择软件包时，要根据自己的实际需要来选取，如 vi 编辑器。如果在安装完成之后需要添加或者删除软件包，则可以重新运行安装程序。

2.1.2　在 Linux 上搭建 Shell 编程环境

由于 Linux 本身都会默认安装 Shell 脚本的运行环境，所以通常情况下并不需要额外安装什么软件。但是，前面已经介绍过，在同一台 Linux 系统上会同时安装多个 Shell，并且这些 Shell 的语法有所不同。所以，在编写和执行 Shell 脚本的时候一定要弄清楚当前使用

的是哪种 Shell。可以使用系统变量$SHELL 来获取当前系统默认的 Shell：

```
[root@linux ~]# echo $SHELL
/bin/bash
```

从上面的输出结果中可以看出，当前系统默认的 Shell 为 bash。

在很多脚本中，指定使用的 Shell 为/bin/sh，实际上它是一个符号链接。在 RHEL 中，这是一个指向/bin/bash 的符号链接：

```
[root@linux ~]# ll /bin/sh
lrwxrwxrwx.    1       root        root        4 8 8  2022     /bin/sh ->
bash
```

这意味着尽管我们在程序中指定的解释器为/bin/sh，但实际上解释 Shell 脚本的是/bin/bash。在 Ubuntu 中，这是一个指向 dash 的符号链接：

```
test@test-virtual-machine:~$ ll /bin/sh
lrwxrwxrwx 1 root root 4  9月 28  2023 /bin/sh -> dash*
```

这表示解释 Shell 脚本的是 dash。dash 是一个不同于 bash 的 Shell，它是为了执行脚本而出现的。它不支持交互，速度更快，但功能相比 bash 要少很多。

Shell 作为一个软件包，当然也有版本，可以使用如下命令来查看 bash 的版本（version）：

```
[root@linux ~]# echo $BASH_VERSION
5.1.8(1)-release
```

从上面的执行结果中可以看出，当前 bash 的版本为 5.1.8。这个版本并不是 bash 的最新版本，编写本书时的 bash 的最新版本为 5.2.0。为了能够使用最新版本的 bash，可以自己编译 bash，步骤如下：

（1）下载 bash 源代码，命令如下：

```
[root@linux ~]# wget http://ftp.gnu.org/gnu/bash/bash-5.2.tar.gz
```

在上面的命令中，wget 命令从 Web 服务器上下载（get）文件，其参数是 bash 最新版本的网址。

（2）解压源代码，命令如下：

```
[root@linux ~]# tar zxvf bash-5.2.tar.gz
```

（3）配置编译环境，命令如下：

```
[root@linux ~]# cd bash-5.2
[root@linux bash-5.2]# ./configure
```

（4）测试编译。为了判断源代码是否能够编译成功，可以使用以下命令进行测试（test）：

```
[root@linux bash-5.2]# make test
```

如果以上命令没有任何错误消息，则可以进行源代码编译操作。

（5）编译 bash，命令如下：

```
[root@linux bash-5.2]# make install
```

默认情况下，bash 将被安装到/usr/local/bin 下。

（6）查看是否安装成功。首先切换到新版本的 bash 的安装目录下：

```
[root@linux bash-5.2]# cd /usr/local/bin/
[root@linux bin]# ll
total 2904
-rwxr-xr-x  1   root    root    2964076     Jun 25 23:19    bash
-r-xr-xr-x  1   root    root    6828        Jun 25 23:19    bashbug
```

接下来切换到新版本的 bash，然后查看当前 bash 的版本，命令如下：

```
[root@linux bin]# ./bash
[root@linux bin]# echo $BASH_VERSION
5.2.0(1)-release
```

从上面的命令中可以看出，当前的 bash 版本已经是 5.2.0 了。这表示新版本的 bash 已经编译成功。但是目前还不能使用这个新的 Shell。出于安全考虑，只能使用/etc/shells 文件中列出的 Shell。下面的命令列出了该文件的内容：

```
[root@linux etc]# more shells
/bin/sh
/bin/bash
/sbin/nologin
/bin/tcsh
/bin/csh
```

从上面的输出结果中可以得知，可以使用的 Shell 有 5 个，而前两个实际上都是 bash。

为了能够使用这个新的 Shell，我们需要将其添加到配置文件中。具体添加方法有很多种，可以直接使用 vi 编辑器修改/etc/shells 文件，追加一行关于新的 Shell 的路径信息即可：

```
/bin/sh
/bin/bash
/sbin/nologin
/bin/tcsh
/bin/csh
/usr/local/bin/bash
```

2.1.3　在 FreeBSD 上搭建 Shell 编程环境

FreeBSD 是 UNIX 两大流派中 BSD 流派的比较典型的一个代表，也是目前应用比较广泛的一个 UNIX 系统。默认情况下，FreeBSD 使用的 Shell 为 csh，这一点可以通过系统变量$SHELL 来获得：

```
freebsd# echo $SHELL
/bin/csh
```

因此，如果想在其他的 Shell 环境中进行程序设计，那么必须自己安装所需要的 Shell。下面以 bash 为例来说明如何在 FreeBSD 上安装其他 Shell。

可以通过两种方式来安装 bash，一种是通过软件包进行安装，这种方式是安装已经编译好的二进制文件，因此安装起来相对较快；另一种是通过 Ports 进行安装，这种方式是从远程服务器上下载软件包的源代码，然后在本地进行编译和安装，因此需要花费额外的编译时间。下面分别介绍这两种安装方式。

1．使用软件包的方式安装bash

如果想要直接安装二进制软件包，则需要使用 pkg 命令，如下：

```
freebsd# pkg install bash
```

在上面的命令中，选项 install 表示从远程服务器上下载并安装软件包，后面的 bash 是软件包的名称。输入以上命令并且按 Enter 键之后，pkg 命令便开始搜索远程的服务器，找到 bash 软件包及其依赖的其他软件包并下载到本地。命令执行完之后，bash 就可以使用了。

2．使用Ports方式安装bash

通过 Ports 方式安装 bash 同样非常简单。在使用 Ports 之前，必须确保当前的 FreeBSD 系统已经安装了 Ports 树，然后执行以下命令安装 bash：

```
freebsd# cd /usr/ports/shell/bash
freebsd# make && make install
```

此时，FreeBSD 的 Ports 管理工具会自动从远程服务器下载 bash 的源代码，然后编译并且安装。

前面讲过，在 Linux 中安装新的 Shell 时需要编辑/etc/shells 文件，追加新的 Shell 路径。在 FreeBSD 中，当用户使用 pkg 命令安装新的 Shell 时，无须再编辑该文件，因为 pkg 命令会自动将新的 Shell 添加到该文件中。

用户唯一要做的事情就是更改登录后的默认的 Shell。这个操作可以使用 chsh（change shell 的简写）命令来完成。chsh 命令可以用来修改一些用户的信息，包括默认的 Shell。在命令行中执行 chsh 会自动调用 vi 或者 vim 打开一个编辑窗口，如图 2-11 所示。

```
#Changing user information for root.
Login: root
Password: $1$WglHdTot$T8eL0jZ6Bml1L4pID9OCw.
Uid [#]: 0
Gid [# or name]: 0
Change [month day year]:
Expire [month day year]:
Class:
Home directory: /root
Shell: /bin/csh
Full Name: Charlie &
Office Location:
Office Phone:
Home Phone:
Other information:
~
~
~
~
~
~
~
~
~
/etc/pw.FFDfvb: unmodified: line 1
```

图 2-11　chsh 命令编辑窗口

然后可以使用 vi 或者 vim 命令来修改相关的选项。如果想要修改默认的 Shell，则可以直接编辑其中的 Shell 项，将其中的/bin/csh 修改为需要的 Shell，如/usr/local/bin/bash。这样，当用户下次登录时就可以使用新的 Shell 了。

2.2　编辑器的选择

对于学习 Shell 编程的用户来说，选择一款好的编辑器是非常重要的。好的编辑器具有许多辅助功能，如语法提示和自动完成等，从而可以提高开发效率。虽然与其他程序设计语言相比 Shell 并没有专门的编辑器，但是其仍然有一些相对较好的辅助工具。本节介绍 Shell 编程中经常使用的一些编辑器。

2.2.1　图形化编辑器

通常情况下，用户都是使用文本编辑器来编写 Shell 脚本。最常用的图形化的文本编辑器有 UltraEdit。这些编辑器都在不同的程度上支持 Shell 的语法提示。如图 2-12 所示为 UltraEdit 的编辑界面，可以看出，UltraEdit 会自动识别 Shell 语句的关键字，并且以不同的颜色来显示。

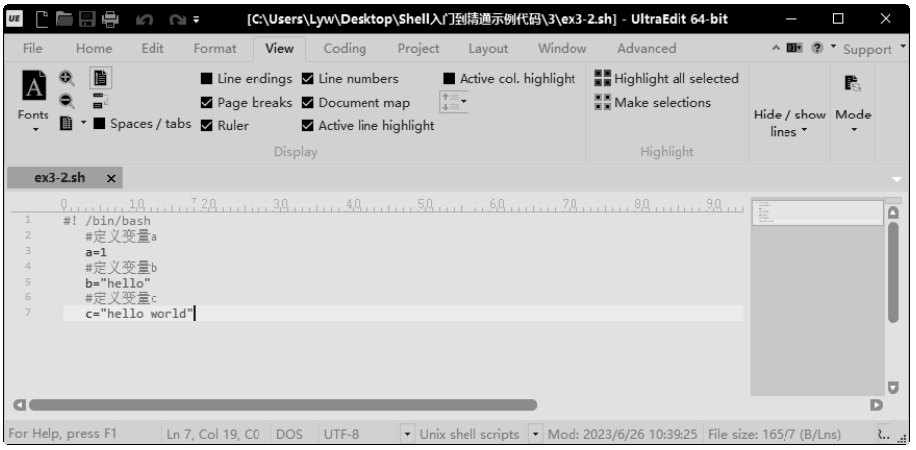

图 2-12　使用 UltraEdit 编辑 Shell 脚本

使用 Notepad++可以达到类似的效果，如图 2-13 所示。

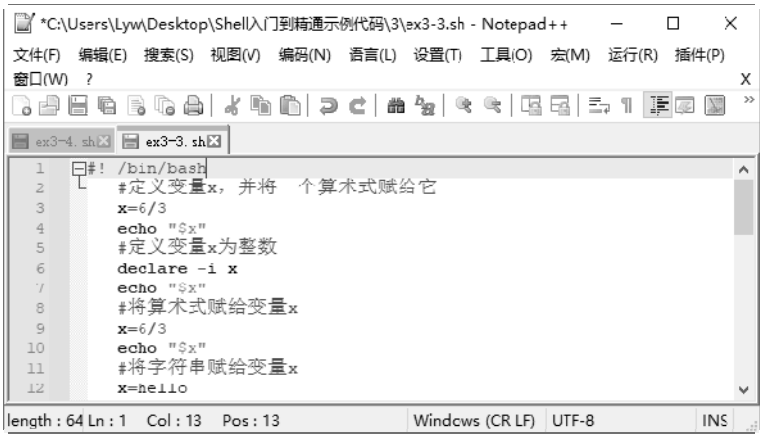

图 2-13　使用 Notepad++编辑 Shell 脚本

注意：普通的文本编辑器如 UltraEdit 并没有调试功能。

2.2.2　vi（vim）编辑器

虽然图形化的编辑器可以提高 Shell 编程的效率，但是只能在图形界面环境中使用。

如果在没有图形界面的场合编写 Shell 脚本，则这些编辑器便无用武之地。实际上，熟悉 Linux 或者 UNIX 的用户很少使用图形化的编辑器。在绝大多数情况下，他们往往更喜欢选择非图形化的编辑器，最常用的是 vi 或者 vim。vim 是 vi 的增强版。在许多情况下，vi 已经足够用了。下面详细介绍 vi 编辑器的使用方法。

vi 编辑器是 Linux 中最常用的编辑器，很多 Linux 发行版都默认安装了 vi。vi 是 visual interface 的缩写。vi 拥有非常多的命令，正因为 vi 有非常多的命令，才使得其功能非常灵活和强大。在一般的 Shell 编程和系统维护中，vi 已经完全够用了。下面详细介绍 vi 编辑器的使用方法，主要包括 vi 的使用模式，文件的打开、关闭和保存方式，插入文本或者新建行，移动光标，删除、恢复字符或者行及搜索字符等。

vi 通常有 3 种使用模式，分别为一般模式、编辑模式和命令模式。在每种模式下，用户可以执行不同的操作。例如：在一般模式下，用户可以进行光标位置的移动、删除及复制字符等；在编辑模式下，用户可以插入字符或者删除字符等；在命令模式下，用户可以保存文件或者退出编辑器等。下面分别介绍这 3 种模式的使用方法。

1．一般模式

当用户刚刚进入 vi 编辑器的时候，当前的模式就是一般模式。一般模式是 3 个模式中功能最复杂的模式，一般的操作都在该模式下完成。由于 vi 并没有提供图形界面，所以所有的操作都是通过键盘来完成的。由于在字符界面下没有鼠标辅助，光标位置的移动是一个非常麻烦的问题，所以 vi 提供了许多移动光标的快捷键，如表 2-1 所示。

表 2-1　移动光标的快捷键

操　作	快 捷 键	说　明
向下移动光标	向下方向键、j键或者空格键	每按1次键，光标向正下方移动1行
向上移动光标	向上方向键、k键或者Backspace键	每按1次键，光标向正上方移动1行
向左移动光标	向左方向键或者h键	每按1次键，光标向左移动1个字符
向右移动光标	向右方向键或者l键	每按1次键，光标向右移动1个字符
移至下一行行首	Enter键	每按1次键，光标会移动到下一行的行首
移至上一行行首	-键	每按1次键，光标会移动到上一行的行首
移至文件最后1行	G键	将光标移动到文件最后一行的行首

注意：除了表 2-1 中列出的移动光标的快捷键之外，还有部分快捷键是在命令模式下使用的。例如，行定位或者移动指定行数等，这些操作将在命令模式中进行介绍。

由于光标的移动相对比较简单，所以此处不再举例说明。读者可以使用 vi 打开一个文件，使用表 2-1 列出的快捷键来尝试移动光标。

插入文本也是编辑器的一项基本功能。为了能够快速地在指定位置插入文本，vi 编辑器提供了许多快捷键，如表 2-2 所示。

表 2-2　文本操作快捷键

操　作	快 捷 键	说　明
右插入	a	在当前光标所处位置的右边插入文本
左插入	i	在当前光标所处位置的左边插入文本

续表

操　　作	快　捷　键	说　　明
行尾追加	A	在当前行的末尾追加文本
行首插入	I	在当前行的开始处插入文本
插入行	O或者o	O键在当前行的上面插入一个新行，o键将在当前行的下面插入一个新行
覆盖文本	R	覆盖当前光标所在的位置及后面的若干文本
合并行	J	将当前光标所在行与下面的一行合并为一行

在表 2-2 中列出的快捷键中，除了 J 键之外，其他快捷键都会导致 vi 编辑器从一般模式切换到编辑模式。I 键的功能是在当前行的开始位置插入文本，不包括空白字符。

注意：使用文本操作快捷键进入编辑模式后，可以通过 Esc 键返回到一般模式。

文本的复制和粘贴等功能也是在一般模式下完成的。如表 2-3 列出了常用的与文本复制和粘贴有关的快捷键。

表 2-3　文本复制和粘贴快捷键

操　　作	快捷键	说　　明
复制行	yy	将当前行复制到缓冲区。如果想要定义多个缓冲区，可以使用ayy、byy以及cyy语法。其中，yy前面的字符表示缓冲区的名称，可以是任意单个字母。这样可以将多个单独的行复制到多个缓冲区中，各个缓冲区相互之间不受影响
复制多行	nyy	将当前行以及下面的*n*行复制到缓冲区。其中，*n*表示一个整数。与yy命令相似，用户也可以使用anyy或者bnyy等语法来命名缓冲区
复制单词	yw	复制从光标当前位置到当前单词（word）词尾的字符
复制多个单词	*n*yw	其中，*n*是一个整数，表示从光标当前位置开始，复制后面的*n*个单词
复制光标到行首	y^	从当前光标所处的位置开始，复制到当前行的行首
复制光标到行尾	y$	从当前光标所处的位置开始，复制到当前行的行尾
粘贴到光标后面的位置	p	将缓冲区中的字符串插入当前光标所处位置的后面。如果定义了多个缓冲区，则使用ap方式来粘贴，其中，字母a表示缓冲区的名称
粘贴到光标前面的位置	P	将缓冲区中的字符串插入当前光标所处位置的前面。如果定义了多个缓冲区，则使用aP的方式来粘贴，其中，字母a表示缓冲区的名称

在表 2-3 列出的快捷键中，yw 快捷键复制的不一定是整个单词，它是从光标当前的位置开始复制。如果当前光标处于某个单词的中间，则 yw 命令将会从当前光标位置开始复制该单词后面的所有字符。例如，在图 2-14 中，光标所处的位置在单词 filename 的字母“i”上面，如果使用 yw 快捷键，则复制到缓冲区的字符为 ilename。

图 2-14　复制单词

同样，如果使用 nyw 快捷键，则会从当前单词的当前字符开始，一直复制到指定单词数的结尾。例如，在图 2-14 中，如果使用 2yw 快捷键，则会复制字符串 ilename in 到缓冲区中。文本的粘贴比较简单，此处不再举例说明。

文本的删除也是在一般模式下完成的。如表 2-4 列出了常用的文本删除快捷键。

表 2-4　文本删除快捷键

操　　作	快　捷　键	说　　明
删除当前字符	x	删除光标所在的位置的字符
删除多个字符	nx	删除从光标所在位置开始的*n*个字符
删除当前行	dd	删除（delete）光标所处的整个行
删除多个行	*n*dd	删除包括当前行在内的*n*行
撤销上一步操作	u	撤销（undo）刚刚执行的操作
撤销多个操作	U	撤销针对当前行的所有操作

在表 2-4 中，常用的操作就是删除当前的字符或当前行，这两个操作分别由快捷键 x 和 dd 完成。

注意：vi 的许多快捷键都是由多个字符组成的，在使用的时候要依次快速地按指定的键。

2．编辑模式

前面已经介绍过，执行表 2-2 中除 J 键之外的任何快捷键，都会使得 vi 从一般模式切换到编辑模式。vi 的编辑模式与其他编辑器的编辑模式没有太大的区别。在编辑模式下，可以使用上、下、左和右 4 个方向键移动光标，使用 Backspace 键和 Delete 键删除光标前面的字符，还可以在光标所在的位置插入字符。

注意：在编辑模式下，不能使用 h、j、k 和 l 这 4 个键移动光标，也不能使用 x 键删除字符，因为在编辑模式下这些字母都被视为普通的字母。

3．命令模式

命令模式也是使用比较多的一种模式，在命令模式下，可以完成文件的打开、保存，将光标跳转到某行及显示行号等操作。下面分别详细介绍这些功能。vi 的命令模式需要从一般模式进入，在一般模式下，按冒号键之后，会在 vi 编辑界面的底部出现命令提示符，如图 2-15 所示。

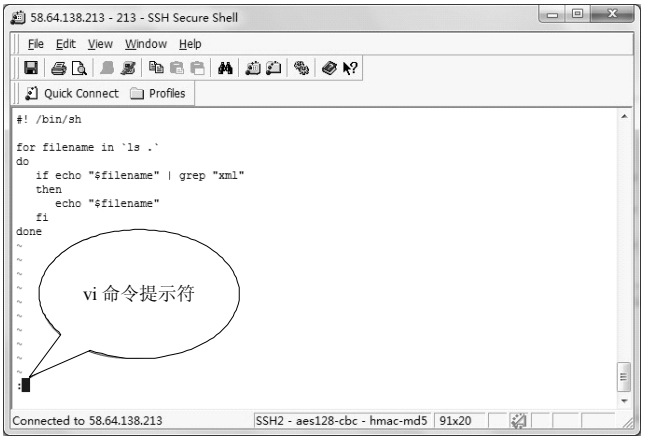

图 2-15　命令模式提示符

出现图 2-15 所示的命令提示符之后，可以输入 vi 命令，如保存文件或者退出 vi 编辑器。表 2-5 列出了常用的 vi 命令。

表 2-5　常用的vi命令

操　作	命　令	说　明
打开文件	:e	打开另外一个文件，将文件名作为参数
保存文件	:w	保存文件，即将文件的改动写入（write）磁盘。如果将文件另存为其他文件名，则可以将新的文件名作为参数
退出编辑器	:q	退出（quit）vi编辑器
直接退出编辑器	:q!	不保存修改，直接退出vi编辑器
退出并保存文件	:wq	将文件保存后退出vi编辑器

在使用 vi 编辑文件的时候，如果想要直接打开某个文件，则可以将文件名作为参数传递给 vi 命令。例如，以下命令将会调用 vi 编辑器并且打开 demo.sh 文件：

```
[root@linux chapter2]# vi demo.sh
```

如果已经启动了 vi 编辑器还想编辑另外一个文件，则可以按冒号键进入命令模式，使用:e 命令打开另外一个文件。由于 vi 只能同时编辑一个文件，所以当打开另外一个文件时，当前打开的文件将被关闭。

在编辑文件的过程中，如果想要将当前的改动写入磁盘，则可以进入命令模式，使用:w 命令可以将文件内容重新写入磁盘。

如果想要退出 vi 编辑器，则可以使用:q 命令。其中，字母 q 表示退出（quit）。在当前文件已经改动的情况下，如果用户使用:q 命令退出 vi 编辑器，则 vi 编辑器会给出保存文件的提示，如图 2-16 所示。该提示告诉用户，文件内容修改后并没有将改动写入磁盘。如果用户已经确定丢弃当前所做的修改，则可以使用:q!命令，该命令将直接退出 vi 编辑器，不给出任何提示。

图 2-16　保存文件提示信息

另外，:w 和:q 这两个命令可以组合使用，变成一个命令，即:wq。当这两个命令组合起来时，表示将文件内容写入磁盘后退出 vi 编辑器。当组合使用时，这两个命令的顺序不能颠倒，一定是 w 在前，q 在后。

注意：感叹号“!”在 vi 编辑器中表示跳过某些检查，强制执行某些操作。例如，丢弃当前的修改，直接退出 vi 编辑器，则可以使用:q!命令。如果丢弃当前的修改，直接打开另外一个文件，则可以使用:e!命令；如果系统管理员修改了某些只读文件，则可以使用:w!命令强制将改动写入磁盘，如图 2-17 所示。

图 2-17　修改只读文件

在命令模式下，除了在表 2-5 中列出的文件操作的命令之外，还有一些常用的命令，如表 2-6 所示。

表 2-6　其他常用的命令

操　作	命　令	说　明
跳至指定行	:n、:n+或者:n-	:n表示跳到行号为n的行，:n+表示向下跳n行，:n-表示向上跳*n*行
显示或者隐藏行号	:set nu或者:set nonu	:set nu（number的简写）表示在每行的前面显示行号；:set nonu（no number的简写）表示隐藏行号
替换字符串	:s/old/new、:s/old/new/g、:n,m s/old/new/g或者:%s/old/new/g	:s/old/new表示用字符串new替换当前行中首次出现的字符串old；:s/old/new/g表示用字符串new替换当前行中所有（global）的字符串old；:n,m s/old/new/g表示用字符串new替换从*n*行到*m*行所有的字符串old；:%s/old/new/g表示用字符串new替换当前文件中所有的字符串old
设置文件格式	:set fileformat=unix	将文件修改为UNIX格式，如Windows中的文本文件在Linux下会出现^M。其中，fileformat可以取unix或者dos等值

对于编写和调试程序的用户来说，在 vi 编辑器中显示行号是一项非常有用的辅助功能，它可以快速地定位出现错误的行。用户可以在命令模式下使用:set nu 命令显示行号，如图 2-18 所示。

最后介绍一下文本搜索。当文件内容比较长时，使用文本搜索功能可以快速查找某些字符串。虽然 vi 的文本搜索功能是在一般模式下进行的，但是它的使用方法与一般模式下有所区别，甚至可以将该项功能单独称为搜索模式。

用户可以在一般模式下通过反斜线“/”快捷键进入文本搜索模式，如图 2-19 所示。

当进入搜索模式时，用户可以在提示符后面输入要搜索的字符串，然后按 Enter 键。此时，光标会停留在当前文件中指定的字符串第一次出现的位置，如图 2-20 所示。

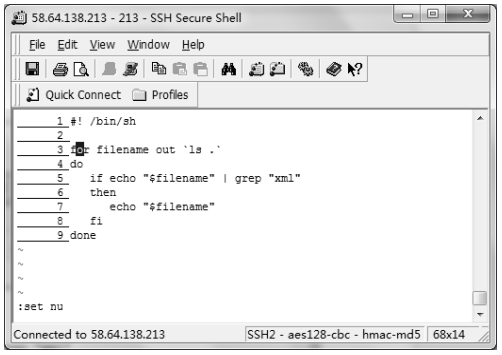

图 2-18　显示行号

图 2-19　文本搜索模式

如果要搜索的文本出现了多次，可以使用 n 键继续向下搜索下一次（next）出现的位置；使用 N 键向上搜索上一次出现的位置。如果要搜索的文本在当前文件中没有出现，则会给出以下提示：

```
E486: Pattern not found: print
```

注意：vi 虽然有比较多的命令，但是只要勤加练习，就会很快掌握，同时也会给工作或学习带来更高的效率。如果不知道自己处在什么模式下，可以按两次 Esc 键回到命令模式。最后提醒一点，注意英文字母的大小写。

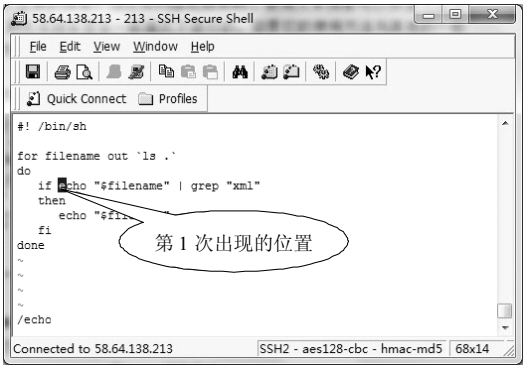

图 2-20　搜索字符串

2.3　系统环境的搭建

在运行 Shell 程序的时候，除了脚本本身之外，还有许多因素会影响 Shell 的执行结果，主要有 Shell 本身的环境及命令的别名等。本节将介绍这两方面的相关知识。

2.3.1　Shell 配置文件

前面已经介绍过，到目前为止已经出现了许多种类型的 Shell，其中常用的有 sh 和 bash 等。这两种 Shell 都有各自的系统环境变量的设置方法，分别保存在不同的配置文件中。

下面介绍这两种 Shell 的配置文件的使用方法。

注意：在 Linux 中，sh 被设计成 bash 的符号链接。

1．sh

Bourne Shell（sh）的配置文件主要有两个，分别为每个用户主目录下的.profile 文件以及/etc/profile 文件。其中，后者是所有用户共同使用的文件。每个用户在登录 Shell 之后，首先会读取和执行/etc/profile 文件中的脚本，然后读取和执行各自主目录下的.profile 文件。因此，可以将所有用户都需要执行的脚本放在/etc/profile 文件中。下面是某个/etc/profile 文件的部分内容：

```
root@solaris # more /etc/profile
# /etc/profile

# System wide environment and startup programs, for login setup
# Functions and aliases go in /etc/bashrc

# It's NOT a good idea to change this file unless you know what you
# are doing. It's much better to create a custom.sh shell script in
# /etc/profile.d/ to make custom changes to your environment, as this
# will prevent the need for merging in future updates.

pathmunge () {
    case ":${PATH}:" in
        *:"$1":*)
            ;;
        *)
            if [ "$2" = "after" ] ; then
                PATH=$PATH:$1
            else
                PATH=$1:$PATH
            fi
    esac
}
…
--More--(32%)
```

从上面的代码中可以看出，/etc/profile 文件的内容与普通的 Shell 脚本并没有太大的区别。

用户主目录下的.profile 文件是一个隐藏文件，该文件的内容与/etc/profile 文件几乎是一样的。该文件是每个用户的私有文件，每个用户在登录 Shell 的时候会自动执行各自的.profile 文件，用户之间不会相互影响。.profile 文件会在/etc/profile 文件之后读取和执行。因此，如果这两个文件中有相同的环境变量，则.profile 文件中的变量值会覆盖/etc/profile 文件中相同的变量值。

注意：在 UNIX 或者 Linux 中，以圆点开头的文件为隐藏文件。用户可以使用 ls 命令的 -a 选项来显示隐藏文件。

2．bash

Bourne-Again Shell（bash）的配置文件主要有 5 个，其中有 4 个文件位于用户主目录下，分别为.bash_profile、.bashrc、.bash_logout 和.bash_history，还有一个文件位于/etc/目

录下，名称为 bashrc。

.bash_profile 文件位于每个用户的主目录下，该文件用来保存用户自己使用的 Shell 信息。当用户登录时，该文件将被读取并执行，并且该文件仅被执行一次。默认情况下，.bash_profile 文件常常用来设置环境变量，执行用户的.bashrc 文件。下面是某个系统中 root 用户的.bash_profile 文件的内容：

```
01  # .bash_profile
02
03  # Get the aliases and functions
04  if [ -f ~/.bashrc ]; then
05          . ~/.bashrc
06  fi
07
08  # User specific environment and startup programs
09
10  PATH=$PATH:$HOME/bin:/usr/pgsql-9.2/bin
11
12  export PATH
```

从上面的代码中可以看出，.bash_profile 文件在第 5 行调用了用户主目录下的.bashrc 文件。第 10 行设置了 PATH 系统变量，第 12 行将系统变量导出。

.bashrc（rc 是 run command 的简写，表示运行命令）文件包含专属于某个用户的 bash 的相关信息，当用户登录以及每次打开新的 bash 时，该文件将被读取并执行。下面是某个系统中 root 用户的.bashrc 文件的内容：

```
01  # .bashrc
02
03  # User specific aliases and functions
04
05  alias rm='rm -i'
06  alias cp='cp -i'
07  alias mv='mv -i'
08
09  # Source global definitions
10  if [ -f /etc/bashrc ]; then
11          . /etc/bashrc
12  fi
```

从上面的代码中可以看出，该文件主要用来定义别名和函数。例如，第 5 行定义了命令 rm -i 的别名为 rm，第 6 行定义了命令 cp -i 的别名为 cp。另外，该文件会调用/etc/bashrc 文件，例如上面代码中的第 11 行。

.bash_logout 文件在当前用户每次退出（log out）Shell 时执行。如果没有特别的要求，该文件的内容通常为空。

/etc/bashrc 文件与 sh 中的/etc/profile 文件非常相似，它是所有使用 bash 的用户共同使用的文件。当任何用户登录 bash 时，都会执行该文件中的代码。下面是某个 Linux 系统中/etc/bashrc 文件的部分内容：

```
01  # /etc/bashrc
02
03  # System wide functions and aliases
04  # Environment stuff goes in /etc/profile
05
06  # It's NOT a good idea to change this file unless you know what you
07  # are doing. It's much better to create a custom.sh shell script in
08  # /etc/profile.d/ to make custom changes to your environment, as this
```

```
# will prevent the need for merging in future updates.

# are we an interactive shell?
if [ "$PS1" ]; then
  if [ -z "$PROMPT_COMMAND" ]; then
    case $TERM in
    xterm*)
        if [ -e /etc/sysconfig/bash-prompt-xterm ]; then
            PROMPT_COMMAND=/etc/sysconfig/bash-prompt-xterm
        else
            PROMPT_COMMAND='printf "\033]0;%s@%s:%s\007" "${USER}"
            "${HOSTNAME%%.*}" "${PWD/#$HOME/~}"'
        fi
        ;;
    screen)
        if [ -e /etc/sysconfig/bash-prompt-screen ]; then
            PROMPT_COMMAND=/etc/sysconfig/bash-prompt-screen
        else
            PROMPT_COMMAND='printf "\033]0;%s@%s:%s\033\\" "${USER}"
            "${HOSTNAME%%.*}" "${PWD/#$HOME/~}"'
        fi
        ;;
    *)
        [ -e /etc/sysconfig/bash-prompt-default ] && PROMPT_COMMAND=
         /etc/sysconfig/bash-prompt-default
        ;;
      esac
  fi
  # Turn on checkwinsize
  shopt -s checkwinsize
…
```

注意：Linux 不建议用户直接修改/etc/profile 或者/etc/bashrc 文件，应该尽量将用户的配置信息放在用户主目录下的对应文件中。

2.3.2　命令别名

顾名思义，命令别名是命令的另外一个名称。在 Linux 中，设置命令别名的作用主要是为了简化命令的输入。对于一个包含许多选项和参数的命令，用户可以为其设置一个别名，这样在调用该命令的时候只要使用别名就可以了。

例如，在 Linux 中为了提高安全性，通常为以下两个命令设置别名：

```
rm -i
```

和

```
cp -i
```

在这两个命令中，-i 选项的作用是相同的，都是使得前面的命令进入交互式模式。如果不使用交互模式，rm 命令可能会直接删除文件而不给用户任何提示；cp 命令也会直接覆盖已经存在的文件而不给用户任何提示。这样的操作无疑会给用户带来非常大的风险。因此，在绝大多数的 Linux 系统中，这两个命令的别名设置方法为：

```
alias rm='rm -i'
alias cp='cp -i'
```

在上面的代码中，alias 命令用来设置命令别名（alias），其基本语法如下：

```
alias command_alias=command
```

其中，参数 command_alias 表示命令的别名，command 表示某个 Shell 命令。设置命令别名之后，用户就可以像使用普通的命令一样使用别名了。例如，在上面的代码中，通过 alias 命令为 rm -i 命令设置了别名为 rm。这样用户在使用 rm 命令时，实际上使用的是 rm -i 命令。

注意：在 Linux Shell 中，别名拥有最高的执行优先级，虽然系统中有 rm 命令，但是 Shell 仍然优先使用 rm 别名。另外，其他对象的优先级从高到低分别为关键字（如 if、functions 等）、函数、内置命令可执行文件和脚本。

2.4　小　　结

本章详细介绍了 Shell 编程环境的搭建，主要内容包括在不同的操作系统平台上搭建 Shell 执行环境、编辑器的选择、Shell 环境变量的设置方法和命令别名。本章的重点是掌握 Cygwin 的安装和配置方法，以及 vi 的常用命令。第 3 章将介绍 Shell 编程中的变量和引用。

2.5　习　　题

一、填空题

1．_____是一个非常优秀的 UNIX 模拟器。
2．vi 编辑器有 3 种模式，分别为_____、_____和_____。
3．Bourne Shell 的配置文件主要有两个，分别为_____和_____。

二、选择题

1．使用 vi 编辑器时，下面的（　　）快捷键可以进入编辑模式。
A．a　　B．i　　C．o　　D．J
2．当用户使用 vi 编辑器编辑文件时，使用（　　）命令仅保存文件，不退出编辑器。
A．:w　　B．wq　　C．wq!　　D．q

三、判断题

1．在同一台 Linux 系统中，可以同时安装多个 Shell。（　　）
2．为命令设置别名的主要目的是简化命令的输入。（　　）

四、操作题

1．练习使用 Cygwin 模拟器终端窗口查看文件列表。
2．练习使用 vi 编辑器编辑文件。

第 2 篇
Shell 编程核心技术

- 第 3 章　变量和引用
- 第 4 章　条件测试和判断语句
- 第 5 章　循环结构
- 第 6 章　函数
- 第 7 章　数组
- 第 8 章　正则表达式
- 第 9 章　文本处理
- 第 10 章　流编辑器
- 第 11 章　文本处理利器 awk 命令
- 第 12 章　文件操作
- 第 13 章　子 Shell 与进程处理

第 3 章　变量和引用

在任何程序设计语言中，变量是不可缺少的元素。正确、恰当地使用变量可以增加程序的可读性，提高程序的健壮性和灵活性。本章将从变量的基础知识开始，依次介绍什么是变量、变量的赋值和清空，以及变量的引用和替换。

本章涉及的主要知识点如下：

- 深入认识变量：主要介绍什么是变量、变量的命名、变量的类型、变量的有效范围，以及系统变量和用户自定义变量等。
- 变量的赋值和替换：主要介绍如何为变量赋值、如何引用变量的值以及如何清空变量的值。
- 引用和替换：主要介绍什么是引用、全引用、部分引用，以及如何进行命令替换和转义等。

3.1　深入理解变量

在程序设计语言中，变量是一个非常重要的概念，也是初学者在进行 Shell 程序设计之前必须掌握的一个非常基础的概念。只有理解变量的使用方法，才能设计出良好的程序。本节介绍 Shell 中的变量的相关知识。

3.1.1　什么是变量

顾名思义，变量是程序设计语言中一个可以变化的量。当然，可以变化的是变量的值。变量在几乎所有的程序设计语言中都有定义，并且其含义也大同小异。从本质上讲，变量就是在程序中保存用户数据的一块内存空间，而变量名就是这块内存空间的地址。

在程序的执行过程中，保存数据的内存空间可能会不断地发生变化，但是代表内存地址的变量名却保持不变。

由于变量的值是在计算机的内存中，所以当计算机被重新启动时，变量的值将会丢失。因此，对于需要长久保存的数据，应该将其写入磁盘中，避免存储在变量中。

3.1.2　变量的命名

对于初学者来说，可以简单地认为变量就是保存在计算机内存中的一系列的键值对。例如：

```
str="hello"
```

在上面的语句中，等号前面的部分就是键，等号后面的就是值。用户使用变量的目的就是通过键来存取不同的值。

在程序设计语言中，一般将上述语句中的键称为变量名。因此，用户是通过变量名来对变量所代表的值进行存取的。

在不同的程序设计语言中，对于变量名的要求也有所不同。在 Shell 中，变量名可以由字母、数字或者下画线组成，并且只能以字母或者下画线开头。对于变量名的长度，Shell 并没有做出明确的规定，因此可以使用任意长度的字符串作为变量名。但是，为了提高程序的可读性，建议使用相对较短的字符串作为变量名。

在一个设计良好的程序中，变量的命名很有讲究。通常情况下，应该选择有明确意义的英文单词作为变量名，尽量避免使用拼音或者毫无意义的字符串作为变量名，这样阅读程序的人通过变量名即可了解该变量的作用。

例如，下面的变量名都是非常好的选择：

```
PATH=/sbin
UID=100
JAVA_HOME="/usr/lib/jvm/jre-1.6.0-openjdk.x86_64/bin/../.."
SSHD=/usr/sbin/sshd
```

而下面的变量名的可读性相对较差：

```
a="123"
str1="hello"
```

这是因为程序的阅读者没有从变量名中获取到任何有用的信息。当变量较多的时候，有可能程序的设计者也不清楚这些变量的作用了，从而导致程序发生错误。

注意：在 Shell 语言中，变量名的大小写是敏感的。因此，大小写不同的两个变量名并不代表同一个变量。

3.1.3　变量的类型

与变量密切相关的有两个概念，其中一个是变量的类型，另外一个是变量的作用域。此处先介绍变量的类型，作用域（有效范围）将在后面介绍。

根据变量类型确定的时间，可以将程序设计语言分为两类，分别是静态类型语言和动态类型语言。其中，静态类型语言在程序的编译期间就确定变量类型，如 Java、C++和 Pascal。在这些语言中使用变量时，必须首先声明其类型。动态设计语言在程序执行过程中才确定变量的数据类型。常见的动态语言有 VBScript、PHP 及 Python 等。在这些语言中，变量的数据类型根据第一次赋值的数据类型来确定。

同样，根据是否强制要求类型定义，可以将程序设计语言分为强类型语言和弱类型语言。强类型语言要求在定义变量时必须明确指定其数据类型，如 Java 和 C++。在强类型语言中，数据类型的转换非常重要。与之相反，弱类型语言则不要求明确指定变量的数据类型，如 VBScript，可以将任意类型的数值赋给该变量，并且不需要执行类型的转换操作。

Shell 是一种动态类型语言和弱类型语言，即在 Shell 中，变量的数据类型无须进行显式地声明，变量的数据类型会根据不同的操作有所变化。准确地讲，Shell 中的变量是不分数据类型的，统一按照字符串进行存储。但是根据变量的上下文环境，允许程序执行一些

不同的操作，如字符串的比较和整数的加减等。

【例 3-1】　演示 Shell 变量的数据类型，代码如下：

```
#----------------------------/chapter3/ex3-1.sh-----------------
#! /bin/bash

#定义变量 x 并且赋值为 123
x=123
#变量 x 加 1
let "x += 1"
#输出变量 x 的值
echo "x = $x"
#显示空行
echo
#替换 x 中的 1 为 abc 并且将值赋给变量 y
y=${x/1/abc}
#输出变量 y 的值
echo "y = $y"
#声明变量 y
declare -i y
#输出变量 y 的值
echo "y = $y"
#变量 y 的值加 1
let "y += 1"
#输出变量 y 的值
echo "y = $y"
#显示空行
echo
#将字符串赋给变量 z
z=abc22
#输出变量 z 的值
echo "z = $z"
#替换变量 z 中的 abc 为数字 11 并且将值赋给变量 m
m=${z/abc/11}
#输出变量 m 的值
echo "m = $m"
#变量 m 加 1
let "m += 1"
#输出变量 m 的值
echo "m = $m"

echo
#将空串赋给变量 n
n=""
#输出变量 n 的值
echo "n = $n"
#变量 n 加 1
let "n += 1"
echo "n = $n"
echo
#输出空变量 p 的值
echo "p = $p"
# 变量 p 加 1
let "p += 1"
echo "p = $p"
```

为了便于介绍，首先来看程序的执行结果：

```
01  [root@linux chapter3]# ./ex3-1.sh
02  x = 124
03
04  y = abc24
05  y = abc24
06  y = 1
07
08  z = abc22
09  m = 1122
10  m = 1123
11
12  n =
13  n = 1
14
15  p =
16  p = 1
```

在例 3-1 中，第 5 行定义了变量 x，并且将一个整数值赋给该变量，因此变量 x 的数据类型为整数。第 7 行使用 let 语句将变量 x 的值加 1，从而变成了 124，输出结果的第 2 行正是程序第 9 行的 echo 语句的执行结果。

代码的第 13 行将变量 x 的值中的数字 1 替换成了字符串 abc，并且将替换后的结果赋给变量 y，因此变量 y 实际上是一个字符串。输出结果的第 4 行是代码第 15 行 echo 语句的执行结果。

代码第 17 行是使用 declare 语句声明整数变量 y，但是这个语句并不影响当前变量 y 的值，因此，输出结果的第 5 行是代码第 19 行 echo 语句的执行结果。

代码第 21 行是将变量 y 加 1，此时变量 y 的值为 abc24，这个值是一个含有字母和数字的字符串。为了能够执行加法运算，Shell 会自动进行数据类型转换，如果遇到含有非数字的字符串，则该字符串将被转换成整数 0，因此，在执行加 1 运算之后，变量 y 的值就变成 1。输出结果的第 6 行是代码第 23 行 echo 语句的执行结果。

代码第 27 行将一个含有字母的字符串赋给变量 z，然后在第 29 行输出该变量的值，得到输出结果的第 8 行。

代码第 31 行将变量 z 的值中的字母 abc 替换为整数 11，并且将替换后的结果赋给变量 m，在第 33 行输出变量 m 的值，即输出结果的第 9 行。

代码第 35 行将变量 m 的值加 1，由于此时 y 的值为 1122，这是一个完全由数字组成的字符串，所以 Shell 可以将其转换为整数，然后执行加法运算，得到结果为 1123。

代码第 41～52 行分别测试了在空串及没有定义变量的情况下，执行加法运算的执行结果。从上面的执行结果中可知，在这两种情况下，变量的值都会被转换为整数 0。

从上面的执行结果中可以看出，Shell 中的变量非常灵活，可以参与任何运算。实际上，在 Shell 中，一切变量都是字符串类型。

3.1.4　变量的定义

在 Shell 中，通常情况下可以直接使用变量，无须先进行定义。当第一次使用某个变量名时，实际上就同时定义了这个变量，在变量的作用域内都可以使用该变量。

【例 3-2】　通过直接使用变量来定义变量，代码如下：

```
01  #----------------------------/chapter3/ex3-2.sh-----------------
```

```
02  #! /bin/bash
03
04  #定义变量 a
05  a=1
06  #定义变量 b
07  b="hello"
08  #定义变量 c
09  c="hello world"
```

在上面的代码中，第 5 行定义了一个名称为 a 的变量，同时将一个数字赋给该变量。第 7 行定义了一个名称为 b 的变量，同时将一个字符串赋给该变量。第 9 行定义了一个变量 c，同时将一个包含空格的字符串赋给该变量。在 Shell 语言中，如果变量的值包含空格，则一定要使用引号引起来。

虽然通过以上方式可以非常方便地定义变量，但是对于变量的某些属性却不容易控制，如变量的类型和读写属性等。为了更好地控制变量的相关属性，bash 提供了一个名称为 declare 的命令来声明（declare）变量，该命令的基本语法如下：

```
declare attribute variable
```

其中，attribute 表示变量的属性，常用的属性如下：

- -p：显示（display）所有变量的值。
- -i：将变量定义为整数（integer）。然后就可以直接对表达式求值，结果只能是整数。如果求值失败或者不是整数，就将变量设置为 0。
- -r：将变量声明为只读（read）变量。只读变量不允许修改，也不允许删除。
- -a：将变量声明为数组（array）变量。但这没有必要。所有变量不必显式定义就可以用作数组。事实上，从某种意义上讲所有变量都是数组，而且赋值给没有下标的变量与赋值给下标为 0 的数组元素的结果相同。
- -f：显示所有自定义函数（function），包括名称和函数体。
- -x：将变量设置成环境变量并导出（export），方便随后的脚本和程序中使用。

参数 variable 表示变量名称。

注意：declare 命令又写作 typeset。

【例 3-3】 演示使用不同的方法声明变量，导致变量在不同的环境下表现不同，代码如下：

```
01  #----------------------------/chapter3/ex3-3.sh-----------------
02  #! /bin/bash
03
04  定义变量 x 并将一个算术式赋给它
05  x=6/3
06  echo "$x"
07  #定义变量 x 为整数
08  declare -i x
09  echo "$x"
10  #将算术式赋给变量 x
11  x=6/3
12  echo "$x"
13  #将字符串赋给变量 x
14  x=hello
15  echo "$x"
```

```
16  #将浮点数赋给变量 x
17  x=3.14
18  echo "$x"
19  #取消变量 x 的整数属性
20  declare +i x
21  #重新将算术式赋给变量 x
22  x=6/3
23  echo "$x"
24  #求表达式的值
25  x=$[6/3]
26  echo "$x"
27  #求表达式的值
28  x=$((6/3))
29  echo "$x"
30  #声明只读变量 x
31  declare -r x
32  echo "$x"
33  #尝试为只读变量赋值
34  x=5
35  echo "$x"
```

以上程序的执行结果如下：

```
01  [root@linux chapter3]# ./ex3-3.sh
02  6/3
03  6/3
04  2
05  0
06  ./ex3-3.sh: line 15: 3.14: syntax error: invalid arithmetic operator
    (error token is ".14")
07  0
08  6/3
09  2
10  2
11  2
12  ./ex3-3.sh: line 32: x: readonly variable
13  2
```

下面对比执行结果分析例 3-3 中的代码。第 5 行使用通常的方法定义了一个变量 x，并且将一个算术式作为初始值赋给该变量。第 6 行输出变量 x 的值。前面已经讲过，Shell 将所有数据都看作字符串来存储，因此在程序执行时，Shell 并不将 6/3 当成一个将被求值的算术式，而是作为一个普通的字符串，所以第 6 行直接输出了这个算术式本身，得到了输出结果的第 2 行。

代码第 8 行使用 declare 语句声明变量 x 为整数，但是程序并没对变量 x 重新赋值，因此第 9 行的 echo 语句的执行结果仍然得到算术式本身，即输出结果的第 3 行。

代码第 11 行对变量 x 重新赋值，将前面的算术表达式赋给它。因为当变量被声明为整数时可以直接参与算术运算，所以在第 12 行的 echo 语句中输出了算术式的值，即输出结果的第 4 行。

第 14 行尝试将一个字符串值赋给整数变量 x，并且在第 15 行使用 echo 语句输出 x 的值。在 Shell 中，如果变量被声明成整数，当把一个结果不是整数的表达式赋值给它时就会变成 0。因此，在输出结果中第 5 行的 0 是代码第 15 行 echo 语句的输出。

第 17 行将一个浮点数赋给变量 x，因为 bash 并不内置对浮点数的支持，所以得到了输出结果第 6 行的错误消息。此时，变量 x 的值变为 0，即在输出结果中第 7 行的 0。

第 20 行取消变量 x 的整数类型属性，第 22 行重新将算术式赋给变量 x，并且在第 23 行使用 echo 语句输出变量 x 的值。由于此时变量 x 已经不是整数变量，所以不能直接参与算术运算。因此，变量 x 的值仍然得到了算术式本身，即输出结果的第 8 行。

在 Shell 中，为了得到算术式的值，可以有两种方法，一种就是使用方括号，即第 25 行中的方式。输出结果的第 9 行正是此时变量 x 的值。另一种是使用圆括号，即第 28 行中的方式。从执行结果中可知，这两种方式都可以得到期望的结果。

第 31 行使用-r 选项声明了一个只读变量，第 34 行尝试为该变量重新赋值，因此得到输出结果第 12 行的错误消息。此时变量 x 的值仍然是 2，所以才有输出结果的第 13 行。

3.1.5　变量和引号

在 Shell 编程中，正确理解引号的作用非常重要。Shell 语言中一共有 3 种引号，分别为单引号（''）、双引号（""）和反引号（`）。这 3 种引号的作用是不同的，其中，由单引号括起来的字符视为普通字符；由双引号括起来的字符，除了“$”“\”“'”“"”这几个特殊字符并保留其特殊功能外，其余字符仍视为普通字符；由反引号括起来的字串被 Shell 解释为命令。在执行时，Shell 首先执行该命令，并以它的标准输出结果取代整个反引号（包括两个反引号）部分。关于单引号和双引号的作用，将在后面的引用部分介绍，下面举例说明反引号的使用方法。

【例 3-4】　演示反引号的使用方法，代码如下：

```
01  #-----------------------------/chapter3/ex3-4.sh------------------
02  #! /bin/bash
03
04  #输出当前目录
05  echo "current directory is `pwd`"
```

在上面的代码中，第 5 行中包含一个由反引号引起来的 Shell 命令 pwd。以上命令的执行结果如下：

```
[root@linux chapter3]# ./ex3-4.sh
current directory is /root/chapter3
```

从上面的执行结果中可以看出，代码的第 5 行在执行的过程中首先会执行 pwd 命令，用该命令的执行结果取代命令所在的位置，然后执行 echo 语句。

注意：反引号是键盘左上角波浪号“～”右边的那个符号。

3.1.6　变量的作用域

接下来介绍 Shell 语言中与变量密切相关的另外一个概念，即变量的作用域。与其他程序设计语言一样，Shell 中的变量也分为全局变量和局部变量两种。下面介绍这两种变量的作用域。

1．全局变量

通常认为，全局变量是使用范围较大的变量，它不局限于某个局部使用。在 Shell 语

言中，全局变量可以在脚本中定义，也可以在某个函数中定义。在脚本中定义的变量都是全局变量，其作用域为从被定义的地方开始，一直到 Shell 脚本结束或者被显式地删除。

【例 3-5】　演示全局变量的使用方法，代码如下：

```
#-----------------------------/chapter3/ex3-5.sh------------------
#! /bin/bash

#定义函数
func()
{
   #输出变量 v1 的值
   echo "$v1"
   #修改变量 v1 的值
   v1=200
}
#在脚本中定义变量 v1
v1=100
#调用函数
func
#输出变量 v1 的值
echo "$v1"
```

在上面的代码中，第 5～11 定义了名称为 func()的函数，第 8 行在函数内部输出全局变量 v1 的值，第 10 行修改全局变量 v1 的值为 200，第 13 行在脚本中，即函数外面定义了变量 v1，该变量是全局变量。第 15 行调用函数 func()，第 17 行重新输出修改后的变量 v1 的值。

程序的执行结果如下：

```
[root@linux chapter3]# ./ex3-5.sh
100
200
```

在上面的执行结果中，100 是第 8 行的 echo 语句的输出，从执行结果中可以看出，在函数 func()内部可以访问全局变量 v1。200 是第 17 行 echo 语句的输出结果，这是因为程序的第 10 行在函数 func()内部修改了变量 v1 的值。从例 3-5 的执行结果中可知，在脚本中定义的变量为全局变量，不仅可以在脚本中直接使用，而且可以在函数内部直接使用。

除了在脚本中定义全局变量之外，在函数内部定义的变量默认情况下也是全局变量。其作用域为从函数被调用时执行变量定义的地方开始，一直到 Shell 脚本结束或者被显式地删除为止。

【例 3-6】　演示在函数内部定义全局变量的方法，代码如下：

```
#-----------------------------/chapter3/ex3-6.sh------------------
#! /bin/bash

#定义函数
func()
{
   #在函数内部定义变量
   v2=200
}
#调用函数
func
#输出变量的值
echo "$v2"
```

在上面的代码中，第 5～9 行定义了名称为 func()的函数，其中，在第 8 行定义了一个名称为 v2 的变量。第 11 行调用 func()函数，第 13 行输出变量 v2 的值。

程序的执行结果如下：

```
[root@linux chapter3]# ./ex3-6.sh
200
```

之所以会得到 200，是因为在 Shell 中，默认情况下函数内部定义的变量也属于全局变量。因此，在代码的第 8 行定义的变量 v2 在函数外部仍然可以使用。

注意：函数的参数是局部变量。

2．局部变量

与全局变量相比，局部变量的使用范围较小，通常仅限于某个程序段访问，如函数内部。在 Shell 语言中，可以在函数内部通过 local 关键字定义局部变量。另外，函数的参数也是局部变量。

【例 3-7】　演示使用 local 关键字定义局部变量的方法。本例对例 3-6 中的代码稍作改动，使用 local 关键字定义变量 v2，代码如下：

```
01  #-----------------------------/chapter3/ex3-7.sh------------------
02  #! /bin/bash
03
04  #定义函数
05  func()
06  {
07     #使用 local 关键字定义局部变量
08     local v2=200
09  }
10  #调用函数
11  func
12  #输出变量的值
13  echo "$v2"
```

程序的执行结果如下：

```
[root@linux chapter3]# ./ex3-7.sh

```

从上面的执行结果中可知，由于在函数内部使用 local 关键字显式地定义了局部变量，所以在函数外面不能获得该变量的值。第 13 行的 echo 语句仅输出了空值。

注意：关于函数的详细介绍请参见第 6 章。

如果用户在函数外面定义了一个全局变量，同时在某个函数内部又存在相同名称的局部变量，则在调用该函数时，函数内部的局部变量会屏蔽函数外部定义的全局变量。也就是说，在出现同名的情况下，函数内部的局部变量会优先被使用。

【例 3-8】　演示全局变量和局部变量的区别。在本例中，定义两个名称都为 v1 的变量，其中一个为全局变量，另外一个为局部变量，然后比较这两个变量的值，代码如下：

```
01  #-----------------------------/chapter3/ex3-8.sh------------------
02  #! /bin/bash
03
04  #定义函数
```

```
05  func()
06  {
07     #输出全局变量 v1 的值
08     echo "global variable v1 is $v1"
09     #定义局部变量 v1
10     local v1=2
11     #输出局部变量 v1 的值
12     echo "local variable v1 is $v1"
13  }
14  #定义全局变量 v1
15  v1=1
16  #调用函数
17  func
18  #输出全局变量 v1 的值
19  echo "global variable v1 is $v1"
```

在上面的代码中，第 5～13 行定义了函数 func()。由于函数内部的局部变量是在第 10 行定义的，所以第 8 行的 echo 语句输出的是全局变量 v1 的值。第 10 行使用 local 关键字定义相同名称的局部变量 v1，并且赋值为 2。第 12 行输出局部变量 v1 的值。第 15 行定义全局变量 v1 并且赋值为 1。第 17 行调用函数 func()，第 19 行输出全局变量 v1 的值。

程序的执行结果如下：

```
[root@linux chapter3]# ./ex3-8.sh
global variable v1 is 1
local variable is 2
global variable is 1
```

从上面的执行结果中可以看出，在函数 func()中，第一次输出的是全局变量 v1 的值，第二次输出的是局部变量 v1 的值。虽然在函数内部修改了变量 v1 的值，但是这只影响局部变量 v1，所以在函数外部输出的仍然是全局变量的值。

注意：Shell 变量中的符号“$”表示取变量的值，只有在取值的时候才使用，定义和赋值时不需要使用符号“$”。另外，在 Shell 中，变量的原型为${var}，而常用的书写形式$var 是简写。在某些情况下，简写形式会导致程序执行错误。

3.1.7　系统变量

Shell 语言的系统变量主要是在对参数和命令返回值进行判断时使用，包括脚本和函数的参数，以及脚本和函数的返回值。Shell 语言中的系统变量并不多，但是十分有用，特别是在进行一些参数检测的时候。如表 3-1 列出了常用的系统变量。

表 3-1　Shell中常用的系统变量

变　量	说　明
$*n*	*n*是一个整数，从1开始，表示参数的位置。例如，$1表示第1个参数，$2表示第2个参数等
$#	命令行参数的个数
$0	当前Shell脚本的名称
$?	前一个命令或者函数的返回状态码
$*	以“参数1 参数2……”的形式将所有的参数通过一个字符串返回

续表

变　量	说　明
$@	以“参数1”“参数2”……的形式返回每个参数
$$	返回本程序的进程ID（PID）

【例 3-9】　演示常用系统变量的使用方法。在本例中通过 Shell 系统变量来获取不同的信息，代码如下：

```
01  #-----------------------------/chapter3/ex3-9.sh------------------
02  #! /bin/bash
03
04  #输出脚本的参数个数
05  echo "the number of parameters is $#"
06  #输出上一个命令的退出状态码
07  echo "the return code of last command is $?"
08  #输出当前的脚本名称
09  echo "the script name is $0"
10  #输出所有参数
11  echo "the parameters are $*"
12  #输出其中的几个参数
13  echo "\$1=$1;\$2=$2;\$11=$11"
```

在上面的代码中，第 5 行使用变量$#获取当前脚本的参数个数，第 7 行使用变量$?获取上一个脚本或者命令的返回状态码，第 9 行通过变量$0 获取当前脚本的名称，第 11 行通过变量$*以一个字符串的形式返回所有的参数值，第 13 行通过变量$n 输出其中几个参数的值。

程序的执行结果如下：

```
[root@linux chapter3]# ./ex3-9.sh a b c d e f g h i j k l m n
the number of parameters is 14
the return code of last command is 0
the script name is ./ex3-9.sh
the parameters are a b c d e f g h i j k l m n
$1=a;$2=b;$11=a1
```

在执行 ex3-9.sh 脚本文件时为该程序提供了 14 个参数，所以第 5 行的输出结果为 14。同时，由于第 5 行的 echo 语句执行成功，所以第 7 行的输出结果为 0。第 9 行的 echo 语句输出当前脚本文件的名称为 ex3-9.sh。第 11 行通过变量$*返回所有的参数的值。第 13 行分别通过变量$1、$2 和$11 获取第 1 个、第 2 个和第 11 个参数的值。从上面的执行结果中可知，第 1 个和第 2 个参数的值都已经正确获取，但是第 11 个参数的值却输出了“a1”，而非希望得到的 k。为什么会得到这种错误的结果呢？

原来在 Shell 语言中，当使用$n 的形式获取位置参数时，Shell 的变量名通常是一位数字，即 1～9。因此，在上面的程序中，当使用变量$11 获取第 11 个参数的值时，Shell 会将“$1”作为变量名，导致获取的实际上是第 1 个参数的值，而最后的“1”则是变量名$11 中的第 2 个“1”，这个数字会直接与前面的字符串连接在一起。

为了能够使 Shell 正确地知道哪些部分是变量名，可以使用花括号来界定变量名。下面将变量名$11 中的数字 11 用花括号括起来：

```
echo "\$1=$1;\$2=$2;\$11=${11}"
```

此时再次执行例 3-9 中的脚本文件，就可以得到正确的结果：

```
[root@linux chapter3]# ./ex3-9.sh a b c d e f g h i j k l m n
the number of parameters is 14
the return code of last command is 0
the script name is ./ex3-9.sh
the parameters are a b c d e f g h i j k l m n
$1=a;$2=b;$11=k
```

注意：例 3-9 中的反斜线“\”称为转义字符，用于将一些 Shell 中的特殊字符转换为普通字符，如“$”或者“’”等。

3.1.8　环境变量

Shell 中的环境变量是所有 Shell 程序都可以使用的变量。Shell 程序在运行时都会接收一组变量，这组变量就是环境变量。环境变量会影响所有脚本的执行结果。如表 3-2 列出了常用的 Shell 变量。

表 3-2　常用的Shell环境变量

变　量	说　明
PATH	命令搜索路径（path），以冒号为分隔符。注意与Windows下不同的是，当前目录不在系统路径里
HOME	用户主目录的路径名，即家（home）目录，是cd命令的默认参数
COLUMNS	定义了命令编辑模式下可使用的命令行的长度，单位是字符的列（column）数
HISTFILE	命令历史文件（history file）
HISTSIZE	命令历史（history）文件中最多可包含的命令条数（size）
HISTFILESIZE	命令历史文件（history file）中包含的最大行数（size）
IFS	定义Shell使用的分隔符
LOGNAME	当前的登录名（login name）
SHELL	Shell的全路径名
TERM	终端（terminal）类型
TMOUT	Shell自动退出的时间，即超时时间（timeout）单位为s。如果设为0，则禁止Shell自动退出
PWD	打印当前的工作目录（print working directory）

除了表 3-2 中列出的一些环境变量之外，还可以使用 set 命令列出所有的环境变量，具体如下：

```
[root@linux chapter3]# set | more
BASH=/bin/bash
BASHOPTS=checkwinsize:cmdhist:expand_aliases:extquote:force_fignore:
hostcomplete:interactive_comments:login_shell:progcomp:promptvars:
sourcepath
BASH_ALIASES=()
BASH_ARGC=()
BASH_ARGV=()
BASH_CMDS=()
BASH_LINENO=()
BASH_SOURCE=()
BASH_VERSINFO=([0]="4" [1]="1" [2]="2" [3]="1" [4]="release" [5]=
```

```
"x86_64-redhat-linux-gnu")
BASH_VERSION='4.1.2(1)-release'
COLORS=/etc/DIR_COLORS
COLUMNS=235
CVS_RSH=ssh
DIRSTACK=()
EUID=0
GROUPS=()
G_BROKEN_FILENAMES=1
HISTCONTROL=ignoredups
HISTFILE=/root/.bash_history
HISTFILESIZE=1000
HISTSIZE=1000
HOME=/root
HOSTNAME=linux
…
```

如果想要使用环境变量，则可以通过相应的变量名来获取。

【例 3-10】　通过环境变量获取与当前 Shell 有关的一些环境变量的值，代码如下：

```
01  #----------------------------/chapter3/ex3-10.sh-----------------
02  #! /bin/bash
03
04  #输出命令搜索路径
05  echo "commands path is $PATH"
06  #输出当前的登录名
07  echo "current login name is $LOGNAME"
08  #输出当前用户的主目录
09  echo "current user's home is $HOME"
10  #输出当前的 Shell
11  echo "current shell is $SHELL"
12  #输出当前的工作目录
13  echo "current path is $PWD"
```

在上面的代码中，第 5 行输出当前用户所设置的命令搜索路径，第 7 行输出当前用户的登录名，第 9 行输出当前用户的主目录，第 11 行输出当前 Shell 的全路径，第 13 行输出当前的工作路径。

程序的执行结果如下：

```
[root@linux chapter3]# ./ex3-10.sh
commands path is /usr/lib64/qt-3.3/bin:/usr/local/sbin:/usr/local/bin:/
sbin:/bin:/usr/sbin:/usr/bin:/root/bin:/usr/pgsql-9.2/bin
current login name is root
current user's home is /root
current shell is /bin/bash
current path is /root/chapter3
```

注意：按照惯例，Shell 中的环境变量全部使用大写字母表示。

3.2　变量的赋值和清空

在了解了变量的基础知识之后，接下来介绍 Shell 语言中变量的赋值和变量的销毁。

3.2.1　变量的赋值

在 Shell 语言中，通常情况下变量并不需要专门定义和初始化。一个没有初始化的 Shell 变量被认为是一个空字符串。可以通过变量的赋值操作来完成变量的声明并赋予其一个特定的值，还可以通过赋值语句为一个变量多次赋值，以改变其值。

在 Shell 中，变量的赋值使用以下语法：

```
variable_name=value
```

其中，varibale_name 表示变量名，关于变量名的命名规则，前面已经介绍过了，这里不再重复说明。value 表示将要赋予的变量值。一般情况下，Shell 将所有普通变量的值都看作字符串。如果 value 中包含空格、制表符和换行符，则必须用单引号或者双引号将其引起来。双引号内允许变量替换，而单引号则不可以。

中间的等于号“=”称为赋值符号，赋值符号的左右两边不能直接跟空格，否则 Shell 会将其视为命令。

例如，下面都是一些正确的赋值语句：

```
v1=Linux
v2='RedHat Linux'
v3="RedHat Linux $HOSTTYPE"
v4=12345
```

此外，Shell 允许只包含数字的变量值参与数值运算，如上面的 v4。另外，在上面的 v3 变量值中包含一个特殊符号“$”，这个符号的作用是取某个变量的值，具体将在后面的变量替换内容中介绍。

3.2.2　引用变量的值

在变量赋值完成之后，就需要使用变量的值。在 Shell 中，可以通过在变量名前面加上“$”来获取该变量的值。实际上，在前面的许多例子中已经多次使用这个符号来获取变量的值，为了让读者更加清楚 Shell 中的变量值的引用方法，下面进行详细介绍。

【例 3-11】　演示 Shell 变量的引用方法，代码如下：

```
01  #-----------------------------/chapter3/ex3-11.sh------------------
02  #!/bin/bash
03
04  v1=Linux
05  v2='RedHat Linux'
06  v3="RedHat Linux $HOSTTYPE"
07  v4=12345
08
09  #输出变量 v1 的值
10  echo "$v1"
11  #输出变量
12  echo "$v2"
13  #输出变量 v3 的值
14  echo "$v3"
15  #输出变量 v4 的值
16  echo "$v4"
```

在上面的代码中，第 4～7 行定义了 4 个变量，并且分别赋予了不同的初始值。第 10～16 输出变量 v1～v4 的值。该程序的执行结果如下：

```
[root@linux chapter3]# ./ex3-11.sh
Linux
RedHat Linux
RedHat Linux x86_64
12345
```

可以看出，上面的程序分别输出了 4 个变量的值。其中，v3 中的环境变量$HOSTTYPE 被具体的值取代。

另外，在 Shell 中，字符串是可以直接连接在一起的。如果对例 3-11 进行修改，将最后的变量 v4 的输出语句，即代码的第 13 行改成以下代码：

```
echo "$v4abc"
```

则程序在执行时会出现错误：

```
[root@linux chapter3]# ./ex3-11.sh
Linux
RedHat Linux
RedHat Linux x86_64

```

从上面的执行结果中可以看出，ex3-11.sh 脚本文件只正确输出了前 3 个变量的值，而最后一个变量的值变成了空字符串。

注意：在上面的执行结果中，变量 v4 的值也输出了，只是变成了空字符串，并非没有输出。

出现这种情况的原因在于，Shell 在进行解释代码时，遇到“$v4abc”这个字符串之后，并不知道具体的变量名到底是什么。因此，它会将整个字符串作为一个变量名来使用。在本程序中，变量 v4abc 当然是没有定义的，因此导致 ex3-11.sh 最后只输出了一个空行。

为了能够使 Shell 正确地界定变量名，避免混淆，在引用变量时可以使用花括号将变量名括起来，例如：

```
echo "${v4}abc"
```

那么以上程序就会得到正确的结果：

```
[root@linux chapter3]# ./ex3-11.sh
Linux
RedHat Linux
RedHat Linux x86_64
12345abc
```

强烈建议读者使用花括号将变量名进行明确地界定，因为这是一种非常正式、完整的书写方法，而省略花括号的形式只是一种简写。

3.2.3　清除变量

当某个 Shell 变量不再需要时可以将其清除。变量被清除后，其所代表的值也会消失。清除变量使用 unset 语句（unset 是 undo set 的简写，表示撤销设置），其基本语法如下：

```
unset variable_name
```

其中，参数 varibale_name 表示要清除的变量名称。

【例 3-12】　演示清除 Shell 变量的方法，并且观察清除前后变量值的变化情况，代码如下：

```
#----------------------------/chapter3/ex3-12.sh-----------------
#! /bin/bash

#定义变量 v1
v1="Hello world"
#输出 v1 的值
echo "$v1"
#清除变量
unset v1
echo "the value of v1 has been reset"
#再次输出变量的值
echo "$v1"
```

在上面的代码中，第 5 行定义了一个名称为 v1 的变量，第 7 行输出该变量的值。第 9 行使用 unset 语句将该变量清除，然后在第 12 行再次输出该变量的值。

程序的执行结果如下：

```
[root@linux chapter3]# ./ex3-12.sh
Hello world
the value of v1 has been reset
```

从上面的执行结果中可以看出，在执行第 9 行的 unset 语句之后，变量 v1 已经被清除掉了，因此第 12 行的 echo 语句仅输出了空值。

3.3　引用和替换

通常对于弱类型的程序设计语言来说，变量的功能都相对比较单薄。但是对于 Shell 来说，变量的功能却非常强大。为了增强变量的功能，Shell 对变量的使用方法进行了极大的扩展。本节将介绍引用和替换。

3.3.1　引用

所谓引用，是指将字符串用引用符号包裹起来，以防止其中的特殊字符被 Shell 解释为其他含义。特殊字符是指除了字面意思之外还可以解释为其他意思的字符。例如，在 Shell 中“$”符号本身的含义是美元符号，其 ASCII 码值为十进制 36。除了这个含义之外，前面已经讲过，“$”符号还可以用来获取某个变量的值，即变量替换。星号“*”也是一个特殊的字符，星号可以作为通配符使用。

【例 3-13】　演示星号通配符的使用方法，命令如下：

```
[root@linux chapter3]# ll ex*
-rwxr-xr-x  1   root        root    179     Jan  7 11:51        ex3-10.sh
-rwxr-xr-x  1   root        root    114     Jan  7 15:49        ex3-11.sh
-rwxr-xr-x  1   root        root    100     Jan  7 16:15        ex3-12.sh
…
```

在上面的 ll 命令中，参数 ex*表示列出以 ex 开头的所有文件。但是，如果使用双引号将其引用起来，则其含义会发生变化：

```
[root@linux chapter3]# ll "ex*"
ls: cannot access ex*: No such file or directory
```

从上面的执行结果中可以看出，当参数 ex*被双引号引起来之后，ll 命令会将其作为一个普通的文件名来对待。但是当前目录中并不存在名称为 ex*的文件，因此程序会给出没有该文件或者目录的提示信息。

对比两次的执行结果可以发现，其中起作用的是双引号，正是它改变了星号的意义，而这正是引用的目的。

在 Shell 中一共有 4 种引用符号，如表 3-3 所示。

表 3-3　常用的引用符号

引 用 符 号	说　　明
双引号	除了美元符号、单引号、反引号和反斜线之外，其他字符都保持字面意义
单引号	所有字符都保持字面意义
反引号	反引号中的字符串将被解释为Shell命令
反斜线	转义字符，屏蔽后的字符的特殊意义

注意：在 Linux 中，ll 命令是 ls -l 命令的别名。

3.3.2　全引用

在 Shell 语句中，当一个字符串被单引号引起来之后，其中的所有字符，除单引号本身之外都将被解释为字面意义，即字符本身的含义。这意味着被单引号引起来的所有字符都将被解释为普通的字符。因此，这种引用方式称为全引用。

【例 3-14】　演示全引用的使用方法，代码如下：

```
01  #-----------------------------/chapter3/ex3-14.sh------------------
02  #! /bin/bash
03
04  #定义变量 v1
05  v1="chunxiao"
06  #输出含有变量名的字符串
07  echo 'Hello, $v1'
```

在上面的代码中，第 5 行定义了一个字符串变量 v1，第 7 行使用 echo 语句输出一个含有变量名 v1 的用单引号引起来的字符串。前面已经讲过，单引号表示全引用，因此第 7 行会原封不动地输出单引号中的字符串。程序的执行结果如下：

```
[root@linux chapter3]# ./ex3-14.sh
Hello, $v1
```

3.3.3　部分引用

对于单引号来说，被其引起来的所有字符都将被解释为字面意义。而对于双引号则有所不同。如果使用双引号将字符串引起来，则其中所包含的字符除了美元符号（$）、反引

号（`）以及反斜线（\）之外的其他字符，都将被解释为字面意义，这称为部分引用。也就是说，在部分引用中，“$”“`”“\”仍然拥有特殊的含义。例如，“$”符号仍然可以用来引用变量的值。

【例 3-15】　演示部分引用的使用方法，将例 3-14 中的单引号改为双引号，代码如下：

```
#----------------------------/chapter3/ex3-15.sh-----------------
#! /bin/bash

#定义变量
v1="chunxiao"
#输出变量的值
echo "Hello, $v1"
```

以上代码的执行结果如下：

```
[root@linux chapter3]# ./ex3-15.sh
Hello, chunxiao
```

从上面的执行结果中可以看出，echo 语句中的变量名已经被变量的值所取代。

3.3.4　命令替换

所谓命令替换，是指在 Shell 程序中，将某个 Shell 命令的执行结果赋给某个变量。在 Bash 中，有两种语法可以进行命令替换，分别是使用反引号和圆括号，例如：

```
`shell_command`
$(shell_command)
```

以上两种语法是等价的，可以根据自己的习惯选择使用。

【例 3-16】　演示反引号的使用方法，代码如下：

```
#----------------------------/chapter3/ex3-16.sh-----------------
#! /bin/bash

#变量替换
v1=`pwd`
#输出变量的值
echo "current working directory is $v1"
```

在上面的代码中，第 5 行将 Shell 命令 pwd 的执行结果赋给变量 v1，然后在第 7 行输出该变量的值。

程序的执行结果如下：

```
[root@linux chapter3]# ./ex3-16.sh
current working directory is /root/chapter3
```

从上面的执行结果中可以看出，命令 pwd 的执行结果已经成功地替换了变量名 v1。

注意：Shell 会将反引号中的字符串当作 Shell 命令，如果输入了错误的命令，则会出现 command not found 的错误提示。

在上面的代码中使用了反引号，使用圆括号对代码进行相应的修改如下：

```
#! /bin/bash

v1=$(pwd)
echo "current working directory is $v1"
```

以上代码的执行结果与例 3-15 完全相同。

3.3.5　转义

顾名思义，转义的作用是转换某些特殊字符的意义。转义使用反斜线来表示，当反斜线后面的一个字符具有特殊的意义时，反斜线将会屏蔽该字符的特殊意义，使得 Shell 按照该字符的字面意义来解释。

例如，我们已经知道$符号是一个特殊的符号，通常加在变量名的前面，用来获取变量的值。在下面的两个命令中，第一个命令可以获得变量的值，而第二个命令则直接输出变量名：

```
[root@linux chapter3]# echo $SHELL
/bin/bash
[root@linux chapter3]# echo \$SHELL
$SHELL
```

为什么会得到这样的结果呢？这是因为在第二个命令的$符号前面使用了转义字符“\”，从而使得紧跟在后面的$符号失去了其特殊的作用，变成了一个普通的字符，即只表示$符号本身而已。

3.4　小　　结

本章详细介绍了 Shell 语言中的变量和引用的相关知识，主要内容包括变量含义、变量的命名规则、变量的定义、变量的作用域、系统变量和环境变量、变量赋值和清除、全引用和部分引用、命令替换及转义等。本章的重点是掌握变量的定义方法、变量的作用域、常用的系统变量和环境变量的使用方法，以及全引用和部分引用。第 4 章将介绍条件测试和判断语句。

3.5　习　　题

一、填空题

1．变量是程序设计语言中的_____。

2．在 Shell 中，变量名由_____、_____或者_____组成，并且只能以_____或者_____开头。

3．在 Shell 语言中共有 3 种引号，分别为_____、_____和_____。

二、选择题

1．在函数内部使用（　　）关键字定义局部变量。

A．func　　B．global　　C．local　　D．fun

2．在 Shell 中，通过在变量名前加上（　　）符号来获取变量的值。

A．#　　B．$　　C．@　　D．！

3．在 Shell 中，（　　）符号用来转换某些特殊字符的意义。

A．/　　B．\　　C．$　　D．！

三、判断题

1．在 Shell 语言中，变量名是区分大小写的。（　　）

2．Shell 中的变量分为全局变量和局部变量。其中，全局变量的范围大于局部变量。（　　）

四、操作题

1. 创建 Shell 脚本 test.sh，定义一个名为 user 的变量并赋值为 alice，然后输出变量 user 的值。

2．创建 Shell 脚本 test1.sh，执行命令 ls，显示当前的目录列表。