

Командная строка Linux и сценарии оболочки

**Библия
пользователя**

Второе издание

Linux[®] Command Line and Shell

Bible

Second Edition

**Richard Blum
Christine Bresnahan**



Wiley Publishing, Inc.

Командная строка Linux и сценарии оболочки

Библия пользователя

Второе издание

**Ричард Блум,
Кристина Бреснахэн**



Москва • Санкт-Петербург • Киев
2012

ББК 32.973.26-018.2.75

Б70

УДК 681.3.07

Компьютерное издательство “Диалектика”

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция *К.А. Птицына*

По общим вопросам обращайтесь в издательство “Диалектика” по адресу:

info@dialektika.com, http://www.dialektika.com

Блум, Ричард, Бреснахэн, Кристина.

Б70 Командная строка Linux и сценарии оболочек. Библия пользователя, 2-е изд. : Пер. с англ. — М. : ООО “И.Д. Вильямс”, 2012. — 784 с. : ил. — Парал. тит. англ. ISBN 978-5-8459-1780-5 (рус.)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Wiley US.

Copyright © 2012 by Dialektika Computer Publishing.

Original English language edition Copyright © 2011 by Wiley Publishing, Inc., Indianapolis, Indiana.

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600.

Wiley and the Wiley logo are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. Linux is a registered trademark of Linus Torvalds. All other trademarks are the property of their respective owners. Wiley Publishing, Inc. is not associated with any product or vendor mentioned in this book.

All rights reserved including the right of reproduction in whole or in part in any form. This translation is published by arrangement with Wiley Publishing, Inc.

Научно-популярное издание

Ричард Блум, Кристина Бреснахэн

Командная строка Linux и сценарии оболочек

Библия пользователя, 2-е издание

Литературный редактор *И.А. Попова*

Верстка *М.А. Удалов*

Художественный редактор *Е.П. Дынник*

Корректор *Л.А. Гордиенко*

Подписано в печать 22.06.2012. Формат 70х100/16.

Гарнитура Times. Печать офсетная.

Усл. печ. л. 63,21. Уч.-изд. л. 44,72.

Тираж 1000 экз. Заказ № 0000.

Первая Академическая типография “Наука”

199034, Санкт-Петербург, 9-я линия, 12/28

ООО “И. Д. Вильямс”, 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-1780-5 (рус.)

ISBN 978-1118-0044-25 (англ.)

© Компьютерное изд-во “Диалектика”, 2012,
перевод, оформление, макетирование

© by Wiley Publishing, Inc., Indianapolis, Indiana, 2011

Оглавление

ЧАСТЬ I. КОМАНДНАЯ СТРОКА LINUX	27
ГЛАВА 1. Основные сведения о командных интерпретаторах Linux	28
ГЛАВА 2. Получение доступа к командному интерпретатору	51
ГЛАВА 3. Основные команды интерпретатора bash	87
ГЛАВА 4. Дальнейшее описание команд интерпретатора bash	119
ГЛАВА 5. Использование переменных среды Linux	150
ГЛАВА 6. Основные сведения о правах доступа к файлам Linux	175
ГЛАВА 7. Управление файловыми системами	199
ГЛАВА 8. Установка программного обеспечения	221
ГЛАВА 9. Работа с редакторами	242
 ЧАСТЬ II. ОСНОВЫ РАБОТЫ СО СЦЕНАРИЯМИ	 269
ГЛАВА 10. Основы создания сценариев	270
ГЛАВА 11. Использование структурированных команд	297
ГЛАВА 12. Продолжение описания структурированных команд	323
ГЛАВА 13. Обработка ввода данных пользователем	352
ГЛАВА 14. Представление данных	379
ГЛАВА 15. Управление сценариями	401
 ЧАСТЬ III. УСОВЕРШЕНСТВОВАННЫЕ СЦЕНАРИИ КОМАНДНОГО ИНТЕРПРЕТАТОРА	 429
ГЛАВА 16. Создание функций	430
ГЛАВА 17. Написание сценариев для графических рабочих столов	453
ГЛАВА 18. Общие сведения о редакторах sed и gawk	480
ГЛАВА 19. Регулярные выражения	507
ГЛАВА 20. Дополнительные сведения о редакторе sed	532
ГЛАВА 21. Дополнительные сведения о редакторе gawk	559
ГЛАВА 22. Работа с другими командными интерпретаторами	588
 ЧАСТЬ IV. ДАЛЬНЕЙШЕЕ РАСШИРЕНИЕ СРЕДСТВ РАБОТЫ СО СЦЕНАРИЯМИ	 617
ГЛАВА 23. Работа с базами данных	618
ГЛАВА 24. Работа в Интернете	647
ГЛАВА 25. Использование электронной почты	669
ГЛАВА 26. Написание программ на основе сценариев	692
ГЛАВА 27. Более сложные сценарии командного интерпретатора	721
ПРИЛОЖЕНИЕ А. Краткое руководство по командам bash	755
ПРИЛОЖЕНИЕ Б. Краткое руководство по программам sed и gawk	763
Предметный указатель	774

Содержание

Посвящение	19
Об авторах	20
О техническом редакторе	20
Благодарности	21
ВВЕДЕНИЕ	22
Для кого предназначена книга	22
Структура книги	23
Принятые соглашения и обозначения	24
Минимальные требования	25
Направления дальнейшей работы	25
ЧАСТЬ I. КОМАНДНАЯ СТРОКА LINUX	27
ГЛАВА 1. Основные сведения о командных интерпретаторах Linux	28
Что такое Linux	28
Изучение ядра Linux	29
Программы GNU	37
Среда рабочего стола Linux	39
Дистрибутивы Linux	45
Основные дистрибутивы Linux	46
Дистрибутивы Linux категории LiveCD	48
Резюме	49
ГЛАВА 2. Получение доступа к командному интерпретатору	51
Эмуляция терминала	51
Графические возможности	52
Клавиатура	57
База данных terminfo	58
Консоль Linux	61
Терминал xterm	63
Параметры командной строки	63
Главное меню параметров xterm (Main Options)	65
Меню параметров VT (VT Options)	68
Меню шрифтов VT (VT Fonts)	70
Терминал Konsole	72
Параметры командной строки	72
Сеансы окон с вкладками	73
Профили	74
Строка меню	75
Терминал GNOME	80

Параметры командной строки	80
Вкладки	81
Строка меню	83
Резюме	86
ГЛАВА 3. Основные команды интерпретатора bash	87
Запуск командного интерпретатора	87
Приглашение к вводу информации командного интерпретатора	89
Руководство по командам bash	91
Навигация в файловой системе	92
Файловая система Linux	92
Переход по каталогам	95
Листинги файлов и каталогов	96
Основной формат листинга	97
Изменение формата представленной информации	99
Полный список параметров	100
Фильтрация вывода листинга	102
Обработка файлов	103
Создание файлов	103
Копирование файлов	104
Формирование ссылок на файлы	106
Переименование файлов	107
Удаление файлов	108
Обработка каталогов	109
Создание каталогов	110
Удаление каталогов	110
Просмотр содержимого файла	111
Просмотр статистических данных файла	111
Просмотр типа файла	112
Просмотр всего файла	112
Просмотр частей файла	116
Резюме	117
ГЛАВА 4. Дальнейшее описание команд интерпретатора bash	119
Отслеживание работы программ	119
Контроль над функционированием процессов	120
Отслеживание процессов в реальном времени	127
Останов процессов	130
Контроль над использованием места на диске	132
Монтирование носителей информации	132
Использование команды df	136
Использование команды du	137
Работа с файлами данных	138
Сортировка данных	138
Поиск данных	142

Сжатие данных	144
Архивирование данных	147
Резюме	149
ГЛАВА 5. Использование переменных среды Linux	150
Общее описание переменных среды	150
Глобальные переменные среды	151
Локальные переменные среды	153
Задание переменных среды	155
Задание локальных переменных среды	156
Задание глобальных переменных среды	157
Удаление переменных среды	158
Заданные по умолчанию переменные среды командного интерпретатора	159
Задание переменной среды PATH	163
Поиск системных переменных среды	164
Командный интерпретатор для входа в систему	165
Интерактивный командный интерпретатор	168
Неинтерактивный командный интерпретатор	171
Массивы переменных	171
Использование псевдонимов команд	173
Резюме	174
ГЛАВА 6. Основные сведения о правах доступа к файлам Linux	175
Безопасность Linux	175
Файл /etc/passwd	176
Файл /etc/shadow	178
Добавление нового пользователя	179
Удаление пользователя	182
Изменение пользователя	182
Использование групп Linux	186
Файл /etc/group	186
Создание новых групп	187
Внесение изменений в группы	188
Общие сведения о правах доступа к файлам	188
Использование символов для определения прав доступа к файлам	189
Заданные по умолчанию права доступа к файлу	190
Изменение параметров безопасности	192
Изменение разрешений	192
Изменение прав владения	194
Обеспечение совместного использования файлов	195
Резюме	197
ГЛАВА 7. Управление файловыми системами	199
Общие сведения о файловых системах Linux	199
Основные файловые системы Linux	200

Файловые системы с ведением журнала	201
Расширенные файловые системы Linux с ведением журнала	202
Работа с файловыми системами	204
Создание разделов	205
Создание файловой системы	208
Устранение нарушений в работе	209
Диспетчеры логических томов	211
Организация управления логическими томами	211
Применение программы LVM в системе Linux	212
Использование Linux LVM	214
Резюме	219
ГЛАВА 8. Установка программного обеспечения	221
Общие сведения об управлении пакетами	221
Системы на основе Debian	222
Управление пакетами с помощью команды aptitude	223
Установка пакетов программ с помощью инструмента aptitude	225
Обновление программного обеспечения с помощью команды aptitude	227
Удаление программного обеспечения с помощью команды aptitude	228
Репозитории aptitude	229
Системы на основе Red Hat	231
Получение списка установленных пакетов	231
Установка программного обеспечения с помощью yum	233
Обновление программного обеспечения с помощью yum	234
Удаление программного обеспечения с помощью yum	234
Возобновление нормальной работы при обнаружении нарушенных зависимостей	235
Репозитории yum	236
Установка из исходного кода	237
Резюме	241
ГЛАВА 9. Работа с редакторами	242
Редактор vim	242
Основные сведения о редакторе vim	243
Редактирование данных	245
Копирование и вставка	246
Поиск и замена	246
Редактор emacs	247
Работа с редактором emacs на терминале	248
Использование редактора emacs в среде X Window	253
Семейство редакторов KDE	254
Редактор KWrite	255
Редактор Kate	259
Редактор GNOME	261
Запуск программы gedit	262
Основные средства gedit	263

Задание параметров	264
Резюме	267
ЧАСТЬ II. ОСНОВЫ РАБОТЫ СО СЦЕНАРИЯМИ	269
ГЛАВА 10. Основы создания сценариев	270
Использование нескольких команд	270
Создание файла сценария	271
Отображение сообщений	273
Использование переменных	275
Переменные среды	275
Пользовательские переменные	276
Обратная одинарная кавычка	278
Перенаправление ввода и вывода	279
Перенаправление вывода	280
Перенаправление ввода	280
Каналы	282
Выполнение математических вычислений	284
Команда <code>expr</code>	285
Использование квадратных скобок	287
Способы выполнения вычислений с плавающей запятой	288
Выход из сценария	292
Проверка статуса выхода	292
Команда <code>exit</code>	293
Резюме	295
ГЛАВА 11. Использование структурированных команд	297
Работа с инструкцией <code>if-then</code>	298
Инструкция <code>if-then-else</code>	300
Уровень вложенности инструкций <code>if</code>	301
Команда <code>test</code>	301
Сравнение чисел	302
Сравнение строк	304
Сравнение файлов	309
Проверка того, предназначен ли файл для выполнения	314
Проверка права владения	315
Проверка с помощью составных условий	317
Дополнительные средства инструкции <code>if-then</code>	318
Применение двойных круглых скобок	318
Использование двойных квадратных скобок	319
Команда <code>case</code>	320
Резюме	322
ГЛАВА 12. Продолжение описания структурированных команд	323
Команда <code>for</code>	323
Чтение значений в списке	324

Чтение сложных значений в списке	325
Чтение списка из переменной	327
Чтение значений из команды	328
Изменение значения разделителя полей	329
Чтение каталога с использованием символов-заместителей	330
Команда <code>for</code> в стиле языка C	332
Команда <code>for</code> в стиле языка C	333
Использование нескольких переменных	334
Команда <code>while</code>	335
Основной формат <code>while</code>	335
Использование нескольких команд <code>test</code>	336
Команда <code>until</code>	338
Уровень вложенности циклов	339
Организация циклов на основе данных файла	341
Управление циклом	343
Команда <code>break</code>	343
Команда <code>continue</code>	346
Обработка вывода в цикле	349
Резюме	350
ГЛАВА 13. Обработка ввода данных пользователем	352
Параметры командной строки	352
Чтение параметров	353
Чтение имени программы	355
Проверка параметров	356
Специальные переменные параметров	357
Подсчет параметров	357
Захват всех данных	359
Применение сдвига	361
Работа с опциями	362
Поиск опций	362
Использование команды <code>getopt</code>	366
Дополнительные возможности команды <code>getopts</code>	369
Стандартизация параметров	371
Получение ввода данных от пользователя	372
Основные способы чтения данных	372
Выход по тайм-ауту	374
Чтение данных без повтора на экране	375
Чтение из файла	376
Резюме	377
ГЛАВА 14. Представление данных	379
Основные сведения о вводе и выводе	379
Стандартные дескрипторы файлов	380
Перенаправление вывода сообщений об ошибках	382

Перенаправление вывода в сценариях	384
Временные перенаправления	384
Постоянные перенаправления	385
Перенаправление ввода в сценариях	386
Создание собственного перенаправления	387
Создание дескрипторов выходных файлов	387
Перенаправление дескрипторов файлов	388
Создание дескрипторов входных файлов	389
Создание дескриптора файла для чтения-записи	390
Закрытие дескрипторов файлов	391
Получение перечня открытых дескрипторов файлов	392
Подавление вывода команды	394
Использование временных файлов	395
Создание локального временного файла	395
Создание временного файла в каталоге /tmp	396
Создание временного каталога	397
Ведение журналов сообщений	398
Резюме	399
ГЛАВА 15. Управление сценариями	401
Обработка сигналов	401
Дополнительные сведения о сигналах Linux	402
Выработка сигналов	402
Перехват сигналов	404
Перехват команды выхода из сценария	405
Удаление ловушки	406
Выполнение сценариев в фоновом режиме	407
Выполнение в фоновом режиме	407
Выполнение нескольких низкоприоритетных заданий	408
Выход из терминального сеанса	409
Выполнение сценариев без привязки к консоли	410
Управление заданиями	411
Просмотр заданий	411
Возобновление выполнения остановленных заданий	413
Определение приоритета процесса	414
Команда nice	414
Команда renice	415
Выполнение в заданное время	416
Планирование заданий с использованием команды at	416
Планирование регулярного выполнения сценариев	420
Запуск с момента загрузки	423
Запуск сценариев во время начальной загрузки	423
Запуск при открытии нового сеанса работы с командным интерпретатором	426
Резюме	426

ЧАСТЬ III. УСОВЕРШЕНСТВОВАННЫЕ СЦЕНАРИИ КОМАНДНОГО ИНТЕРПРЕТАТОРА	429
ГЛАВА 16. Создание функций	430
Основные сведения о применении функций в сценариях	430
Создание функций	431
Использование функций	431
Возврат значения	434
Статус выхода, заданный по умолчанию	434
Использование команды return	435
Использование вывода из функции	436
Использование переменных в функциях	437
Передача параметров в функцию	437
Обработка переменных в функции	439
Переменные типа массива и функции	442
Передача массивов в функции	442
Возврат массивов из функций	443
Рекурсивный вызов функций	444
Создание библиотеки	446
Использование функций в командной строке	448
Создание функций в командной строке	448
Определение функций в файле .bashrc	449
Резюме	451
ГЛАВА 17. Написание сценариев для графических рабочих столов	453
Создание текстовых меню	453
Создание компоновки меню	454
Создание функций меню	455
Добавление средств организации работы меню	456
Соединение описанных компонентов в одном сценарии	457
Использование команды select	458
Организация работы по такому же принципу, как в Windows	459
Пакет dialog	460
Параметры команды dialog	466
Использование команды dialog в сценарии	468
Применение графического режима	470
Среда KDE	471
Среда GNOME	474
Резюме	478
ГЛАВА 18. Общие сведения о редакторах sed и gawk	480
Работа с текстом	480
Редактор sed	481
Программа gawk	484
Основные сведения о редакторе sed	490

Более подробное описание опций подстановки	490
Использование адресов	492
Удаление строк	495
Вставка и добавление текста	497
Внесение изменений в строки	499
Команда <code>transform</code>	500
Дополнительные сведения о формировании выходных данных	501
Использование файлов при работе с редактором <code>sed</code>	503
Резюме	506
ГЛАВА 19. Регулярные выражения	507
Общее определение понятия регулярного выражения	507
Определение	507
Типы регулярных выражений	509
Определение шаблонов BRE	509
Обычный текст	509
Специальные символы	511
Символы обозначения точек привязки	512
Символ точки	514
Классы символов	515
Обращение классов символов	518
Использование диапазонов	518
Специальные классы символов	519
Звездочка	520
Расширенные регулярные выражения	521
Вопросительный знак	522
Знак “плюс”	522
Использование фигурных скобок	523
Символ канала	524
Выражения группирования	525
Регулярные выражения в действии	526
Подсчет количества файлов в каталоге	526
Проверка правильности номера телефона	527
Синтаксический анализ адреса электронной почты	529
Резюме	531
ГЛАВА 20. Дополнительные сведения о редакторе <code>sed</code>	532
Многострочные команды	532
Команда <code>next</code>	533
Многострочная команда <code>delete</code>	536
Многострочная команда <code>print</code>	537
Пространство хранения	538
Обращение команды	540
Изменение потока управления	542
Выполнение перехода	543

Проверка	544
Замена шаблона	545
Амперсанд	546
Замена отдельных слов	546
Использование редактора sed в сценариях	548
Использование оболочек	548
Перенаправление вывода sed	549
Создание программ sed	549
Строки с двойными интервалами	549
Файлы с двойными интервалами, которые могут уже содержать пустые строки	550
Нумерация строк в файле	551
Вывод последних строк	552
Удаление строк	553
Удаление дескрипторов HTML	555
Резюме	557
ГЛАВА 21. Дополнительные сведения о редакторе gawk	559
Использование переменных	559
Встроенные переменные	560
Пользовательские переменные	565
Работа с массивами	567
Определение переменных с типом массива	568
Обработка переменных с типом массива в цикле	568
Удаление переменных с типом массива	569
Использование шаблонов	570
Регулярные выражения	570
Оператор сопоставления	571
Математические выражения	572
Структурированные команды	572
Оператор if	573
Инструкция while	574
Инструкция do-while	576
Инструкция for	576
Форматированный вывод	577
Встроенные функции	580
Математические функции	580
Строковые функции	581
Функции работы со временем	583
Определяемые пользователем функции	584
Определение функции	584
Использование собственных функций	585
Создание библиотеки функций	585
Резюме	586

ГЛАВА 22. Работа с другими командными интерпретаторами	588
Общие сведения о командном интерпретаторе dash	588
Средства командного интерпретатора dash	590
Параметры командной строки командного интерпретатора dash	590
Переменные среды командного интерпретатора dash	591
Встроенные команды dash	594
Сценарная поддержка в командном интерпретаторе dash	595
Создание сценариев командного интерпретатора dash	595
Возможные причины возникновения нарушений в работе	596
Командный интерпретатор zsh	601
Компоненты командного интерпретатора zsh	602
Опции командного интерпретатора	602
Встроенные команды	606
Сценарная поддержка с помощью командного интерпретатора zsh	611
Математические выражения	611
Структурированные команды	613
Функции	613
Резюме	616
 ЧАСТЬ IV. ДАЛЬНЕЙШЕЕ РАСШИРЕНИЕ СРЕДСТВ РАБОТЫ СО СЦЕНАРИЯМИ	 617
ГЛАВА 23. Работа с базами данных	618
База данных MySQL	619
Установка MySQL	619
Клиентский интерфейс MySQL	621
Создание объектов базы данных MySQL	625
База данных PostgreSQL	628
Установка PostgreSQL	628
Интерфейс команд PostgreSQL	629
Создание объектов базы данных PostgreSQL	632
Работа с таблицами	634
Создание таблицы	634
Вставка и удаление данных	636
Выполнение запросов к данным	637
Использование базы данных в сценарии	638
Подключение к базе данных	639
Передача команд на сервер	641
Форматирование данных	644
Резюме	646
 ГЛАВА 24. Работа в Интернете	 647
Программа Lynx	647
Установка программы Lynx	648
Командная строка lynx	649
Файл конфигурации Lynx	656

Переменные среды Lwp	657
Перехват данных, поступающих из программы Lwp	658
Программа cURL	661
Установка программы cURL	661
Получение веб-ресурсов с помощью программы cURL	661
Работа в сети с помощью командного интерпретатора zsh	662
Модуль TCP	663
Подход на основе принципа “клиент/сервер”	663
Программирование “клиент/сервер” с применением командного интерпретатора zsh	664
Резюме	668
ГЛАВА 25. Использование электронной почты	669
Основы электронной почты Linux	669
Электронная почта в Linux	670
Агент пересылки сообщений	671
Агент доставки сообщений	672
Почтовый агент пользователя	673
Установка сервера	677
sendmail	678
Postfix	680
Отправка сообщений с помощью программы Mailx	684
Программа Mutt	687
Установка программы Mutt	687
Командная строка Mutt	688
Использование программы Mutt	689
Резюме	690
ГЛАВА 26. Написание программ на основе сценариев	692
Контроль над использованием места на диске	692
Обязательные функции	693
Создание сценария	695
Выполнение сценария	697
Выполнение резервного копирования	698
Архивирование файлов данных	698
Управление учетными записями пользователей	705
Обязательные функции	706
Создание сценария	713
Резюме	720
ГЛАВА 27. Более сложные сценарии командного интерпретатора	721
Текущий контроль статистических данных системы	721
Отчет с моментальным снимком системы	722
Отчет со статистическими данными системы	729
Отслеживание проблем, связанных с базой данных	737
Создание базы данных	737

Регистрация проблемы	740
Обновление сведений о проблеме	743
Поиск проблемы	749
Резюме	753
ПРИЛОЖЕНИЕ А. Краткое руководство по командам bash	755
Встроенные команды	755
Команды bash	757
Переменные среды	759
ПРИЛОЖЕНИЕ Б. Краткое руководство по программам sed и gawk	763
Редактор sed	763
Начало работы с редактором sed	763
Команды sed	764
Программа gawk	768
Формат команды gawk	768
Использование программы gawk	769
Переменные gawk	770
Средства программы gawk	772
Предметный указатель	774

Посвящение

*Посвящается Господу Богу всемогущему, “в коем
скрыты все сокровища мудрости и знаний”.*

Послание к колоссянам, 2:3

Об авторах

Ричард Блум имеет более чем 20-летний стаж работы системного и сетевого администратора в отрасли информационных технологий. Он опубликовал много книг по Linux и по программированию с открытым исходным кодом, занимался администрированием серверов UNIX, Linux, Novell и Microsoft, а также помогал проектировать и сопровождать сеть с 3500 пользователями на основе коммутаторов и маршрутизаторов Cisco. Он постоянно использовал серверы Linux и сценарии командного интерпретатора для осуществления автоматизированного текущего контроля сети и разрабатывал сценарии командного интерпретатора для большинства широко применяемых сред командного интерпретатора Linux. Рик выполняет функции инструктора оперативных вводных курсов по операционной системе Linux, которые включены в расписания занятий многих колледжей и университетов в США. Отвлекаясь от своих занятий с компьютером, Рик играет на электрогитаре в составе нескольких различных любительских церковных оркестров и проводит свободное время со своей женой Барбарой и двумя дочерьми, Кэйти Джейн и Джессикой.

Кристина Бреснахэн начала работать с компьютерами больше 25 лет тому назад, когда вступила в должность системного администратора в отрасли информационных технологий. В настоящее время она занимает должность адъюнкт-профессора в колледже Ivy Tech Community в Индианаполисе, шт. Индиана, и ведет курсы по системному администрированию Linux, безопасности Linux и Windows.

О техническом редакторе

Джек Кокс занимает должность старшего менеджера на предприятии CapTech в г. Ричмонд, шт. Вайоминг. Он имеет более чем 25-летний опыт работы в отрасли информационных технологий, охватывающий самые разные направления, включая мобильные вычисления, обработку транзакций, радиочастотную идентификацию (RFID-технологии), разработку на языке Java и шифрование. Джек проживает в Ричмонде со своей очаровательной женой и непоседами-детьми. Кроме профессиональной деятельности, в круг его интересов входят церковь, его дети и ближайшие родственники.

Благодарности

Прежде всего провозглашаю славу и благодарность Богу, который через своего сына, Иисуса Христа, открывает дорогу ко всем благим деяниям и дарит нам счастье вечной жизни.

Выражаю большую признательность потрясающему коллективу сотрудников издательства John Wiley & Sons за их самоотверженную работу над этим проектом. Хочу поблагодарить Мэри Джеймс (Mary James), редактора, ответственного за приобретение лицензий, которая предложила нам воспользоваться возможностью поработать над этой книгой. Кроме того, выражаю свою признательность Брайену Эррманну (Brian Herrmann), редактору проекта, который внимательно следил за подготовкой книги и помог сделать ее намного более презентабельной. Спасибо, Брайен, за вашу упорную работу и усердие. Технический редактор, Джек Кокс, взял на себя труд по двукратной проверке всего содержания книги, а также многое подсказал для того, чтобы улучшить ее. Благодарю Нэнси Рапопорт (Nancy Rapoport), технического редактора, за ее бесконечное терпение и стремление сделать подготовленный нами текст более удобочитаемым. Мы хотели бы также поблагодарить Кэрол Маккландон (Carole McClendon) из компании Waterside Productions, Inc, за то, что открыла для нас перспективу нового направления деятельности и помогла нам начать карьеру писателей.

Кристина хотела бы поблагодарить своего мужа, Тимоти, за его поддержку, терпение и готовность выслушать, даже если он понятия не имеет, о чем она ведет разговор.

Введение

Сердечно приветствуем вас со страниц второго издания книги *Командная строка Linux и сценарии оболочки. Библия пользователя*. Как и во всех книгах серии ...*Библия пользователя*, в ней можно найти и полезный на практике учебник, и сведения из области профессиональной работы, а также справочные и теоретические сведения, создающие необходимый контекст для изучаемого материала. Данная книга представляет собой достаточно полный источник информации по командам командной строки и командному интерпретатору Linux. К тому времени, как вы закончите работу над этой книгой, вы получите достаточно хорошую подготовку для того, чтобы самостоятельно разрабатывать собственные сценарии командного интерпретатора, позволяющие автоматизировать решение фактически любой задачи в системе Linux.

Для кого предназначена книга

Системный администратор в среде Linux получит большую пользу, узнав о том, как ведется разработка сценариев командного интерпретатора. В этой книге не рассматривается процесс настройки системы Linux, а речь идет о том, как автоматизировать некоторые рутинные административные задачи, которые становятся повседневными после того, как система вступает в период каждодневной эксплуатации. Именно в этой деятельности трудно обойтись без сценарной поддержки на основе команд командного интерпретатора, и данная книга станет для вас настоящим помощником. В книге показано, как с помощью сценариев командного интерпретатора автоматизировать любые административные задачи, начиная со сбора статистических данных системы для текущего контроля и заканчивая накоплением файлов данных для формирования отчетов, по которым могут быть приняты решения по улучшению работы.

Большую пользу книга принесет и тем, кто применяет систему Linux в личных целях. В наши дни в операционных системах доминирующее положение занимают готовые программы с графическим интерфейсом. В большинстве дистрибутивов Linux применяются рабочие столы, которые устроены так, что рядовому пользователю очень нелегко осуществить свои замыслы по упрощению собственной работы в системе Linux. Но иногда решение такой задачи становится буквально неотложным, поэтому пользователю приходится искать возможность работы с командами командного интерпретатора. Настоящая книга показывает, как получить доступ к приглашению командной строки Linux и продолжить работу после получения возможности воспользоваться этим интерфейсом. К тому же практика показывает, что многие простые задачи, такие как управление файлами, часто можно решить быстрее с помощью командной строки, чем с применением программы, имеющей пусть даже самый привлекательный графический интерфейс. Из командной строки можно вызвать на выполнение множество удобных команд, и в этой книге описано, как это сделать.

Структура книги

В настоящей книге приведены основные сведения о командной строке Linux и вслед за этим рассматриваются более сложные темы, в частности, касающиеся создания собственных сценариев командного интерпретатора. Книга разделена на пять частей, каждая из которых опирается на то, что было изложено в предыдущих частях.

В части I материал представлен на основе предположения о том, что у вас имеется работающая система Linux или вы готовы к тому, чтобы получить в свое распоряжение эту систему. В главе 1, “Командная строка Linux”, рассматриваются основные компоненты всей системы Linux и показано, какое место среди них занимает командный интерпретатор. После описания основ системы Linux в части I изложение переходит к перечисленным ниже темам.

- Использование пакета эмуляции терминала для получения доступа к командному интерпретатору (глава 2).
- Вводные сведения об основных командах командного интерпретатора (глава 3).
- Использование более сложных команд командного интерпретатора для получения информации о системе (глава 4).
- Применение переменных командного интерпретатора для манипулирования данными (глава 5).
- Основные сведения о файловой системе Linux и безопасности (глава 6).
- Работа с файловыми системами Linux из командной строки (глава 7).
- Установка и обновление программного обеспечения из командной строки (глава 8).
- Использование редакторов Linux для разработки сценариев командного интерпретатора (глава 9).

В части II работа над сценариями командного интерпретатора становится основной темой. В главах этой части рассматривается указанный ниже материал.

- Способы создания и выполнения сценариев командного интерпретатора (глава 10).
- Изменение хода выполнения программы в сценарии командного интерпретатора (глава 11).
- Выполнение разделов кода в цикле (глава 12).
- Обработка в сценариях данных, введенных пользователем (глава 13).
- Различные способы сохранения и отображения данных, полученных из сценария (глава 14).
- Управление способом и временем выполнения сценариев командного интерпретатора в системе (глава 15).

Часть III посвящена описанию более сложных областей программирования сценариев командного интерпретатора. В частности, в ней рассматриваются перечисленные ниже темы.

- Создание собственных функций, предназначенных для использования в нескольких сценариях (глава 16).
- Использование графического рабочего стола Linux для взаимодействия с пользователем сценария (глава 17).
- Применение команд Linux для фильтрации и синтаксического анализа файлов данных (глава 18).

- Использование регулярных выражений для определения данных (глава 19).
- Усовершенствованные способы управления данными в сценариях (глава 20).
- Формирование отчетов на основе произвольных данных (глава 21).
- Внесение изменений в сценарии командного интерпретатора в целях применения в других командных интерпретаторах Linux (глава 22).

В части IV, в которой представлены последние главы книги, показано, как использовать сценарии командного интерпретатора на практике. В этой части рассматриваются указанные ниже темы.

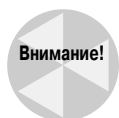
- Применение широко распространенных баз данных с открытым исходным кодом в сценариях командного интерпретатора (глава 23).
- Извлечение данных из текстового содержимого веб-сайтов и обмен данными между системами (глава 24).
- Использование электронной почты для отправки уведомлений и отчетов внешним пользователям (глава 25).
- Написание сценариев командного интерпретатора для автоматизации повседневных функций системного администрирования (глава 26).
- Использование всех средств, описанных в настоящей книге, для создания сценариев командного интерпретатора профессионального качества (глава 27).

Принятые соглашения и обозначения

В настоящей книге применяются многие способы организации материала и типографские соглашения, которые способствуют лучшему усвоению представленного в ней материала.

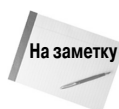
Предупреждения, примечания и советы

В предупреждениях авторы приводят сведения, которые обязательно должны привлечь внимание читателя.



Поскольку эти сведения очень важны, они располагаются в отдельных абзацах со специальной пиктограммой. Предупреждения позволяют узнать, о чем не следует забывать в процессе работы, будь то просто возможные причины неудобств или ситуации, потенциально опасные для данных или системы.

Для изложения дополнительных любопытных пояснений, относящихся к теме главы, авторы используют примечания.



Примечания предоставляют дополнительную, вспомогательную информацию, которая может оказаться полезной, но немного выделяется из потока излагаемых сведений.

Советы содержат сведения, позволяющие упростить выполняемую работу.



Советы позволяют взглянуть на излагаемый материал немного под другим углом зрения.

Минимальные требования

Книга *Командная строка Linux и сценарии оболочки. Библия пользователя* не нацелена на применение какого-то конкретного дистрибутива Linux, поэтому вы сможете изучать приведенные в ней сведения с использованием любой системы Linux, которая имеется в наличии. По большей части в книге речь идет о командном интерпретаторе `bash`, который применяется по умолчанию в большинстве систем Linux.

Направления дальнейшей работы

После завершения работы над книгой вы сможете успешно применять команды Linux в своей повседневной работе в системе Linux. Но следует помнить, что в мире Linux постоянно происходят изменения, поэтому рекомендуем вам оставаться в курсе новых разработок. Изменения часто касаются дистрибутивов Linux, в которых добавляются новые средства и исключаются устаревшие. Чтобы ваши знания о системе Linux всегда отражали современное состояние, старайтесь оставаться хорошо осведомленными. Найдите популярный сайт с форумами, посвященными Linux, и следите за всем, что происходит в мире Linux. Количество широко посещаемых новостных сайтов Linux, на которых предоставлена актуальная информация о новых достижениях в разработке Linux, таких как `Slashdot` и `Distrowatch`, достаточно велико.

Командная строка Linux

ЧАСТЬ



В этой части...

Глава 1

Основные сведения о командных интерпретаторах Linux

Глава 2

Получение доступа к командному интерпретатору

Глава 3

Основные команды командного интерпретатора bash

Глава 4

Дальнейшее описание команд командного интерпретатора bash

Глава 5

Использование переменных среды Linux

Глава 6

Основные сведения о правах доступа к файлам Linux

Глава 7

Управление файловыми системами

Глава 8

Установка программного обеспечения

Глава 9

Работа с редакторами

ГЛАВА

1

В этой главе...

Что такое Linux

Дистрибутивы Linux

Резюме

Основные сведения о командных интерпретаторах Linux

Прежде чем приступить к работе с командной строкой и командными интерпретаторами Linux, необходимо разобраться в том, что такое Linux, каково происхождение этой операционной системы и как она работает. В настоящей главе приведено общее описание того, что такое Linux, и показано, какое место занимают командный интерпретатор и командная строка во всем наборе программных средств Linux.

Что такое Linux

У тех, кто впервые сталкивается с Linux, возникает вопрос, почему у этой операционной системы так много различных версий. Рассматривая пакеты Linux, читатели наверняка уже сталкивались с такими терминами, как *дистрибутив*, *LiveCD* и *GNU*, и хотят в них разобраться. На первых порах приобщение к миру Linux может оказаться сложным. В начале этой главы мы раскроем некоторые тайны, окружающие систему Linux, прежде чем приступить к описанию команд и сценариев.

Для начала отметим, что система Linux состоит из следующих основных частей.

- Ядро Linux.
- Программы GNU.
- Среда графического рабочего стола.
- Прикладное программное обеспечение.

Каждая из этих четырех частей имеет в системе Linux определенное назначение. Причем каждая из них, отдельно взятая, не очень полезна. На рис. 1.1 приведена упрощенная схема того, как взаимодействуют указанные части, создавая полноценную систему Linux.

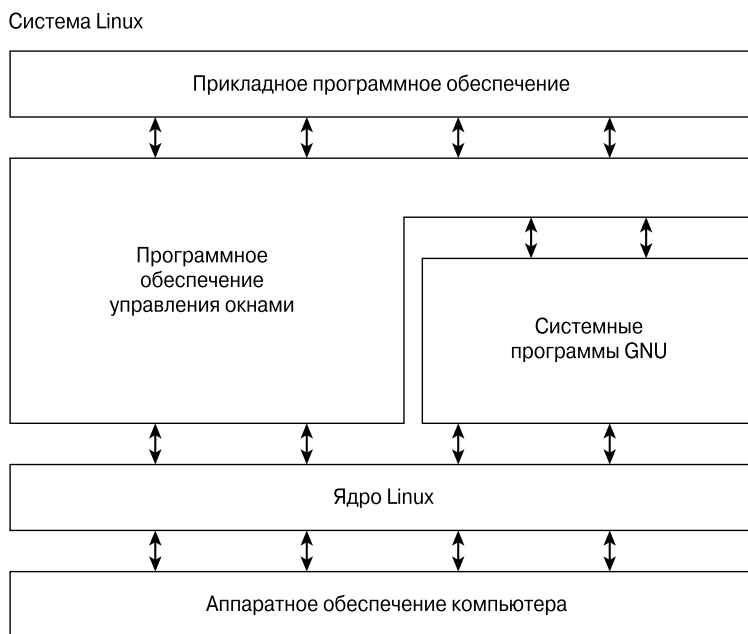


Рис. 1.1. Система Linux

В настоящем разделе подробно описаны эти четыре основные части и приведены общие сведения о том, как осуществляется их взаимодействие для создания полной системы Linux.

Изучение ядра Linux

Основой системы Linux является *ядро*. Ядро управляет всем аппаратным и программным обеспечением компьютерной системы, выделяя по мере необходимости аппаратные средства и выполняя программы, входящие в состав программного обеспечения, когда это потребуется.

Заинтересовавшись изучением мира Linux, разумеется, нельзя не узнать о таком имени, как Линус Торвальдс (Linus Torvalds). Именно Линус впервые создал программное обеспечение ядра Linux, еще будучи студентом Университета Хельсинки. Он поставил перед собой задачу создать копию системы Unix, которая в то время представляла собой популярную операционную систему, применяемую во многих университетах.

После разработки ядра Linux Линус Торвальдс представил его сообществу пользователей Интернета с просьбой направлять ему предложения об улучшении ядра. Это простое решение

вызвало целую революцию в мире компьютерных операционных систем. Вскоре Линус стал получать предложения и от студентов, и от профессиональных программистов со всего света.

Но с самого начала стало ясно, что предоставление любому желающему возможности изменять программный код ядра приведет к полному хаосу. Чтобы упростить работу, Линус стал главным координатором всех предложений по усовершенствованию. В конечном итоге именно Линус принимал решение о том, следует ли включать предложенный код в ядро. Этот подход до сих пор сохраняется в разработке кода ядра Linux, если не считать того, что теперь вместо самого Линуса, управляющего развитием кода ядра, эту задачу взяла на себя целая команда разработчиков.

Ядро, прежде всего, отвечает за выполнение четырех основных функций.

- Управление памятью системы.
- Управление программным обеспечением.
- Управление аппаратными средствами.
- Управление файловой системой.

В следующих разделах каждая из этих функций рассматривается более подробно.

Управление памятью системы

Одной из основных функций ядра операционной системы является управление памятью. Ядро не только управляет физической памятью, имеющейся на сервере, но и может также создавать и управлять виртуальной памятью, т.е. памятью, в основе которой лежит совместное применение программных и аппаратных средств.

Виртуальная память создается с использованием пространства на жестком диске, называемого *областью подкачки*. Ядро на время записывает содержимое неиспользуемых блоков фактически установленной на компьютере физической памяти в область подкачки и обозначает эти блоки как относящиеся к виртуальной памяти, а после того как записанные на диск блоки снова потребуются, возвращает их назад в физическую память. Благодаря этому система получает возможность распоряжаться большим объемом памяти, чем в действительности имеется на компьютере (рис. 1.2).

Блоки памяти группируются в объекты, называемые *страницами*. Ядро может найти каждую страницу памяти либо в физической памяти, либо в области подкачки. Кроме того, ядро сопровождает таблицу страниц памяти, в которой указано, какие страницы находятся в физической памяти и какие выгружены на диск.

Ядро следит за тем, какие страницы памяти используются, и автоматически переносит страницы памяти, к которым не осуществлялся доступ в течение определенного времени, в область подкачки (эта операция называется *выгрузкой*), даже если еще остается незанятый объем физической памяти. Если в программе возникает необходимость получить доступ к выгруженной странице памяти, ядро должно освободить для нее место в физической памяти, выгрузив другую страницу памяти, а затем загрузить требуемую страницу из области подкачки. Очевидно, что для осуществления такого процесса требуются определенные затраты времени, и это может вызвать замедление работы исполняемого процесса. Процесс выгрузки и загрузки страниц памяти для исполняющихся прикладных программ продолжается на протяжении всей работы системы Linux.

С текущим состоянием виртуальной памяти в конкретной системе Linux можно ознакомиться, рассмотрев специальный файл `/proc/meminfo`. Ниже приведен пример типичной записи файла `/proc/meminfo`.

Схема распределения памяти в системе Linux

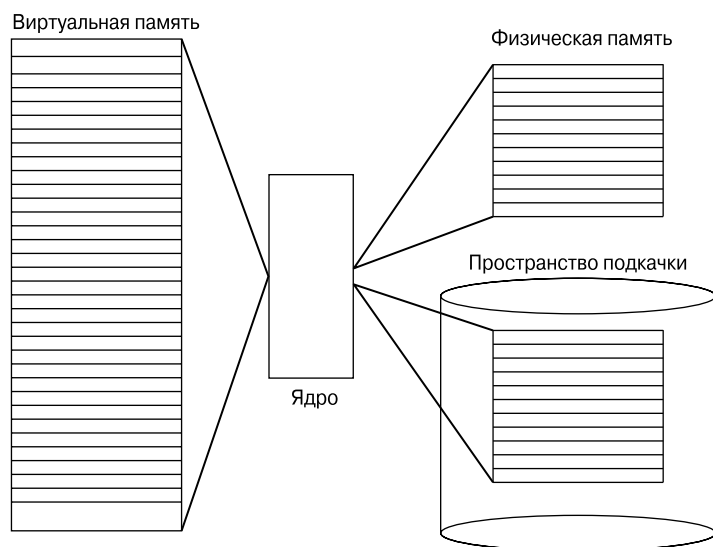


Рис. 1.2. Схема управления памятью системы Linux

```
rich@rich-desktop:~$ cat /proc/meminfo
MemTotal:        1026084 kB
MemFree:         666356 kB
Buffers:          49900 kB
Cached:          152272 kB
SwapCached:        0 kB
Active:           171468 kB
Inactive:         154196 kB
Active(anon):     131056 kB
Inactive(anon):   32 kB
Active(file):     40412 kB
Inactive(file):   154164 kB
Unevictable:      12 kB
Mlocked:          12 kB
HighTotal:        139208 kB
HighFree:         252 kB
LowTotal:         886876 kB
LowFree:          666104 kB
SwapTotal:        2781176 kB
SwapFree:         2781176 kB
Dirty:            588 kB
Writeback:        0 kB
AnonPages:        123500 kB
Mapped:           52232 kB
Shmem:            7600 kB
Slab:             17676 kB
SReclaimable:     9788 kB
```

```

SUnreclaim:      7888 kB
KernelStack:     2656 kB
PageTables:      5072 kB
NFS_Unstable:    0 kB
Bounce:          0 kB
WritebackTmp:    0 kB
CommitLimit:     3294216 kB
Committed_AS:    1234480 kB
VmallocTotal:    122880 kB
VmallocUsed:      7520 kB
VmallocChunk:    110672 kB
HardwareCorrupted: 0 kB
HugePages_Total: 0
HugePages_Free:  0
HugePages_Rsvd:  0
HugePages_Surv:  0
Hugepagesize:    4096 kB
DirectMap4k:     12280 kB
DirectMap4M:     897024 kB
rich@rich-desktop:~$

```

Строка `MemTotal`: показывает, что на данном сервере Linux имеется 1 Гбайт физической памяти. На рисунке также показано, что приблизительно 660 Мбайт в настоящее время не используется (`MemFree`). Кроме того, приведенный вывод показывает, что в этой системе имеется область подкачки объемом 2,5 Гбайт (`SwapTotal`).

По умолчанию для каждого процесса, исполняемого в системе Linux, отведены его собственные закрытые страницы памяти. Ни один процесс не может получить доступ к страницам памяти, используемым другим процессом. Ядро сопровождает свои собственные области памяти. В целях безопасности для всех прочих процессов исключена возможность получить доступ к памяти, используемой процессами ядра.

Но для обеспечения совместного доступа к данным можно создать страницы общей памяти. В области общей памяти чтение и запись могут осуществляться одновременно несколькими процессами. Ядро сопровождает области общей памяти и управляет ими, а также предоставляет отдельным процессам доступ к таким разделяемым областям.

Для просмотра применяемых в настоящее время в системе страниц общей памяти может применяться специальная команда `ipcs`. Ниже приведен типичный пример результатов выполнения команды `ipcs`.

```

# ipcs -m

----- Shared Memory Segments -----
key          shmid  owner    perms    bytes      nattch     status
0x00000000  0             rich      600      52228      6          dest
0x395ec51c  1             oracle    640      5787648   6
#

```

Каждый сегмент общей памяти имеет владельца, который создал этот сегмент. Помимо этого, каждый сегмент имеет стандартные разрешения Linux, которые определяют доступность сегмента для других пользователей. Для предоставления другим пользователям возможности получить доступ к сегменту общей памяти используется определенное значение ключа.

Управление программным обеспечением

В операционной системе Linux исполняемая программа именуется *процессом*. Процесс может выполняться на переднем плане и отображать свой вывод на дисплее или в фоновом режиме незаметно для посторонних глаз. Ядро осуществляет контроль над тем, как система Linux управляет всеми процессами, эксплуатируемыми в системе.

Ядро создает первый процесс, называемый процессом *init*, который обеспечивает запуск всех прочих процессов в системе. После запуска ядро загружает процесс *init* в виртуальную память. По мере загрузки ядром каждого следующего процесса происходит выделение для него уникальной области в виртуальной памяти для хранения данных и кода, используемых процессом.

В некоторых реализациях Linux предусмотрено использование таблицы процессов, в которой показаны процессы, запускаемые автоматически после загрузки. В одних системах Linux эта таблица обычно находится в специальном файле */etc/inittabs*.

А в других системах (таких как популярный дистрибутив Ubuntu Linux) используется папка */etc/init.d*, которая содержит сценарии запуска и останова отдельных приложений во время начальной загрузки. Сценарии запускаются с помощью записей в папках */etc/rcX.d*, где *X* обозначает режим работы.

В операционной системе Linux используется определенная система инициализации, которая вводит в действие конкретные режимы работы. Режим работы может использоваться для управления процессом инициализации в целях выполнения при запуске системы только процессов определенного типа в соответствии с тем, что задано в файле */etc/inittabs* или в папках */etc/rcX.d*. В операционной системе Linux предусмотрено пять режимов работы процесса инициализации.

В режиме работы 1 запускаются только основные процессы системы, а также один процесс операторского терминала. Этот режим называется *однопользовательским* и чаще всего применяется для сопровождения файловой системы в чрезвычайных обстоятельствах, когда приходится устранять последствия какого-то нарушения в работе. Очевидно, что в этом режиме в систему может войти только одно лицо (обычно администратор) для манипулирования данными.

Стандартным режимом работы процесса инициализации является режим 3, в котором запускается основная часть прикладного программного обеспечения, такого как программное обеспечение поддержки сети. Еще одним широко применяемым в Linux режимом работы является режим работы 5. В этом режиме работы система запускает графическое программное обеспечение X Window и предоставляет пользователю возможность войти в систему с применением графического окна рабочего стола.

Система Linux может управлять всеми системными функциональными средствами путем задания режима работы процесса инициализации. После перехода от режима работы 3 к режиму работы 5 в системе вместо доступа к программным средствам с помощью консоли начинает применяться развитая графическая система X Window.

В главе 4 будет показано, как используется команда *ps* для просмотра процессов, исполняемых в настоящее время в системе Linux. Ниже показано, какой вид имеют результаты выполнения команды *ps*.

```
$ ps ax
  PID TTY          STAT       TIME COMMAND
    1 ?            S           0:03   init
    2 ?            SW          0:00 [kflushd]
    3 ?            SW          0:00 [kupdate]
    4 ?            SW          0:00 [kpiod]
```

```

    5 ?      SW    0:00 [kswapd]
  243 ?     SW    0:00 [portmap]
  295 ?     S     0:00 syslogd
  305 ?     S     0:00 klogd
  320 ?     S     0:00 /usr/sbin/atd
  335 ?     S     0:00 crond
  350 ?     S     0:00 inetd
  365 ?     SW    0:00 [lpd]
  403 ttyS0  S     0:00 gpm -t ms
  418 ?     S     0:00 httpd
  423 ?     S     0:00 httpd
  424 ?     SW    0:00 [httpd]
  425 ?     SW    0:00 [httpd]
  426 ?     SW    0:00 [httpd]
  427 ?     SW    0:00 [httpd]
  428 ?     SW    0:00 [httpd]
  429 ?     SW    0:00 [httpd]
  430 ?     SW    0:00 [httpd]
  436 ?     SW    0:00 [httpd]
  437 ?     SW    0:00 [httpd]
  438 ?     SW    0:00 [httpd]
  470 ?     S     0:02 xfs -port -l
  485 ?     SW    0:00 [smbd]
  495 ?     S     0:00 nmbd -D
  533 ?     SW    0:00 [postmaster]
  538 tty1   SW    0:00 [mingetty]
  539 tty2   SW    0:00 [mingetty]
  540 tty3   SW    0:00 [mingetty]
  541 tty4   SW    0:00 [mingetty]
  542 tty5   SW    0:00 [mingetty]
  543 tty6   SW    0:00 [mingetty]
  544 ?     SW    0:00 [prefdm]
  549 ?     SW    0:00 [prefdm]
  559 ?     S     0:02 [kwm]
  585 ?     S     0:06 kikbd
  594 ?     S     0:00 kwmsound
  595 ?     S     0:03 kpanel
  596 ?     S     0:02 kfm
  597 ?     S     0:00 krootwm
  598 ?     S     0:01 kbgndwm
  611 ?     S     0:00 kcmlaptop -daemon
  666 ?     S     0:00 /usr/libexec/postfix/master
  668 ?     S     0:00 qmgr -l -t fifo -u
  787 ?     S     0:00 pickup -l -t fifo
  790 ?     S     0:00 telnetd: 192.168.1.2 [vt100]
  791 pts/0  S     0:00 login -- rich
  792 pts/0  S     0:00 -bash
  805 pts/0  R     0:00 ps ax
$

```

В первом столбце этого вывода показан *идентификатор процесса* (process ID — PID) для каждого исполняемого процесса. Отметим, что первым из этих процессов является уже знакомый нам процесс `init`, которому система Linux присваивает идентификатор PID 1. Всем прочим процессам, запускаемым после процесса `init`, присваиваются идентификаторы процессов в порядке возрастания номеров. Один и тот же идентификатор процесса не может быть присвоен двум разным процессам (хотя применявшиеся ранее числовые идентификаторы процессов могут повторно использоваться системой после завершения исходного процесса).

В третьем столбце показано текущее состояние процесса (S означает, что процесс простаивает — *sleeping*, SW говорит о том, что процесс простаивает и находится в состоянии ожидания — *sleeping and waiting*, а R указывает, что процесс работает — *running*). В последнем столбце показаны имена процессов. Квадратными скобками обозначены процессы, выгруженные из памяти в дисковую область подкачки в связи с отсутствием в них активности. Очевидно, что выгружена только часть процессов, но к большинству работающих процессов выгрузка не применяется.

Управление аппаратными средствами

Еще одной задачей, возложенной на ядро, является управление аппаратными средствами. Любое устройство, управляемое системой Linux, должно обмениваться данными с кодом драйвера, вставленного в код ядра. Код драйвера позволяет ядру передавать данные в устройство в прямом и обратном направлениях, действуя в качестве посредника между приложениями и аппаратными средствами. Предусмотрены два указанных ниже метода вставки кода драйвера в ядро Linux.

- Компиляция кода драйвера вместе с кодом ядра.
- Добавление модуля драйвера к ядру.

В ранних версиях Linux единственный способ вставки кода драйвера устройства состоял в повторной компиляции ядра. После добавления к системе каждого нового устройства приходилось повторно компилировать код ядра. Этот процесс становился все более трудоемким по мере того, как возрастало разнообразие аппаратных средств, поддерживаемых ядром Linux. К счастью, разработчики Linux изобрели лучший метод вставки кода драйвера в исполняемое ядро.

Программисты разработали понятие модулей ядра, позволяющих вставлять код драйвера в ядро, не прерывая его работу и не испытывая необходимости повторно компилировать ядро. Кроме того, модуль ядра может быть удален из ядра после завершения работы устройства, для которого предназначен данный модуль. Это способствует существенному упрощению использования аппаратных средств в системе Linux и расширению их номенклатуры.

В системе Linux конкретные устройства, входящие в состав аппаратных средств, идентифицируются как специальные файлы, называемые *файлами устройств*. Предусмотрено применение файлов устройств, относящихся к трем указанным ниже различным классам.

- Символьные.
- Блочные.
- Сетевые.

Символьные файлы устройств предназначены для устройств, которые могут обрабатывать данные только по одному символу одновременно. Большинство типов модемов и терминалов создаются как символьные файлы. Блочные файлы поддерживают устройства, способные обрабатывать данные одновременно в виде больших блоков, такие как дисковые накопители.

Сетевые типы файлов используются для устройств, в которых применяются сетевые пакеты для отправки и получения данных. К таким устройствам относятся сетевые платы и специальное петлевое устройство, которое позволяет системе Linux обмениваться данными с самой собой, используя общие сетевые программные протоколы.

В системе Linux создаются особые файлы, называемые *специальными файлами*, для каждого системного устройства. Вся связь с устройством осуществляется через специальный файл устройства. Каждый специальный файл обозначен уникальной парой номеров, которая идентифицирует его для ядра Linux. Эта пара номеров состоит из старшего и младшего номера устройства. Аналогичные устройства группируются под одним и тем же старшим номером устройства. Младший номер устройства используется для идентификации конкретного устройства в группе старшего номера устройства. Ниже приведены примеры нескольких файлов устройств на сервере Linux.

```
rich@rich-desktop: $ cd /dev
rich@rich-desktop:/dev$ ls -al sda* ttyS*
brw-rw---- 1 root disk      8,  0 2010-09-18 17:25 sda
brw-rw---- 1 root disk      8,  1 2010-09-18 17:25 sda1
brw-rw---- 1 root disk      8,  2 2010-09-18 17:25 sda2
brw-rw---- 1 root disk      8,  5 2010-09-18 17:25 sda5
crw-rw---- 1 root dialout  4, 64 2010-09-18 17:25 ttyS0
crw-rw---- 1 root dialout  4, 65 2010-09-18 17:25 ttyS1
crw-rw---- 1 root dialout  4, 66 2010-09-18 17:25 ttyS2
crw-rw---- 1 root dialout  4, 67 2010-09-18 17:25 ttyS3
rich@rich-desktop:/dev$
```

В различных дистрибутивах Linux работа с устройствами ведется с применением разных имен устройств. В рассматриваемом дистрибутиве устройство `sda` представляет собой первый жесткий диск ATA, а устройства `ttyS` соответствуют стандартным COM-портам персонального компьютера класса IBM. В этом листинге показаны все устройства `sda`, которые были созданы в типичной системе Linux. Не все эти устройства фактически используются, но они были созданы на тот случай, что потребуются администратору. Аналогичным образом в том же листинге показаны все созданные устройства `ttyS`.

В пятом столбце приведены старшие номера специальных файлов устройств. Обратите внимание на то, что все устройства `sda` имеют обозначенный одним и тем же номером 8 старший специальный файл устройства, а для всех устройств `ttyS` используется номер 4. В шестом столбце показан младший номер специального файла устройства. Каждое устройство, относящееся к определенному старшему номеру, имеет свой собственный уникальный младший номер специального файла устройства.

В первом столбце приведены разрешения для файла устройства. Первый символ разрешения указывает тип файла. Заслуживает внимания то, что все файлы жестких дисков ATA отмечены как блочные устройства (`b` — block), а файлы устройств COM-портов обозначены как символьные устройства (`c` — character).

Управление файловой системой

В отличие от некоторых других операционных систем, ядро Linux может поддерживать файловые системы различных типов для чтения и записи данных на жестких дисках. В самой системе Linux предусмотрено больше десяти собственных файловых систем, а кроме того, она позволяет осуществлять передачу данных в прямом и обратном направлении между файловыми системами, применяемыми в других операционных системах, таких как Microsoft Windows. Ядро должно быть откомпилировано с учетом поддержки всех тех типов файловых систем,

которые предназначены для использования в системе. В табл. 1.1 перечислены стандартные файловые системы, которые могут применяться в системе Linux для чтения и записи данных.

Таблица 1.1. Файловые системы Linux

Файловая система	Описание
ext	Расширенная (extended, сокращенно — ext) файловая система Linux — исходная файловая система Linux
ext2	Вторая расширенная файловая система, предоставляющая дополнительные возможности по сравнению с ext
ext3	Третья расширенная файловая система, в которой поддерживается ведение журналов
ext4	Четвертая расширенная файловая система, в которой поддерживаются дополнительные средства ведения журналов
hpfs	Высокопроизводительная файловая система OS/2
jfs	Файловая система IBM с ведением журналов
iso9660	Файловая система ISO 9660 (для CD-ROM)
minix	Файловая система MINIX
msdos	Файловая система Microsoft FAT16
ncp	Файловая система NetWare
nfs	Сетевая файловая система
ntfs	Поддержка для файловой системы Microsoft NT
proc	Доступ к системной информации
ReiserFS	Расширенная файловая система Linux, обеспечивающая лучшую производительность и восстановление информации на дисках
smb	Файловая система Samba SMB для сетевого доступа
sysv	Более старая файловая система Unix
ufs	Файловая система BSD
umsdos	Файловая система, подобная Unix, работа которой основана на применении файловой системы msdos
vfat	Файловая система Windows 95 (FAT32)
XFS	Высокопроизводительная 64-разрядная файловая система с ведением журналов

Все жесткие диски, доступ к которым предоставляется серверу Linux, должны быть отформатированы с использованием одного из типов файловых систем, перечисленных в табл. 1.1.

Ядро Linux взаимодействует с каждой файловой системой с использованием виртуальной файловой системы (Virtual File System — VFS). Таким образом ядро получает стандартный интерфейс для обмена данными с файловыми системами любых типов. Система VFS кеширует в памяти информацию, относящуюся к каждой смонтированной и используемой файловой системе.

Программы GNU

Кроме ядра, управляющего аппаратными устройствами, для компьютерной операционной системы нужны служебные программы для выполнения таких стандартных функций, как управление файлами и программами. К тому времени, как Линус создал ядро системы Linux, в его распоряжении не было ни одной системной служебной программы, или утилиты, которая могла бы эксплуатироваться в этой системе. К счастью для Линуса, в то же самое время,

когда он разрабатывал свою систему, определенная группа людей осуществляла в Интернете совместную работу в попытке разработать стандартный набор программ для компьютерной системы, которые могли бы имитировать популярную операционную систему Unix.

Организацией GNU (GNU представляет собой рекурсивное сокращение от GNU's Not Unix, или GNU — не Unix) был разработан полный набор служебных программ, аналогичных Unix, но в ее распоряжении не было базовой системы, в которой эти программы могли бы эксплуатироваться. Эти служебные программы были разработаны в рамках подхода к созданию программного обеспечения, получившего название разработки программ с открытым исходным кодом (open source software — OSS).

Концепция OSS позволяет программистам разрабатывать программное обеспечение, а затем выпускать в свет, не сопровождая обязательствами вносить за него лицензионную плату. Любой желающий может использовать это программное обеспечение, вносить в него изменения или встраивать в собственную систему, не будучи обязанным платить за лицензию. Объединение ядра Linux, созданного Линусом, с системными утилитами операционной системы, созданными организацией GNU, привело к появлению полноценной, работоспособной и бесплатной операционной системы.

Название *Linux* часто применяется без дополнительного уточнения для обозначения этой связки ядра Linux и программ GNU, но в Интернете можно увидеть, что некоторые педанты среди пользователей Linux именуют ее системой *GNU/Linux*, чтобы отдать должное организации GNU, которая внесла свой вклад в создание системы.

Основные программы GNU

Проект GNU разрабатывался главным образом в интересах системных администраторов Unix, которым требовалось предоставить доступ к среде, подобной Unix. Такая ориентация проекта привела к тому, что в состав вновь разработанных системных программ была перенесена значительная часть широко применяемых программ командной строки системы Unix. Основная совокупность таких программ, предусмотренных для использования в системе Linux, получила название пакета *coreutils*.

Пакет *coreutils* организации GNU состоит из трех частей.

- Программы обработки файлов.
- Программы манипулирования текстом.
- Программы управления процессами.

Каждая из этих трех основных групп программ содержит несколько утилит, которые являются весьма полезными для системных администраторов и программистов Linux. В настоящей книге подробно рассматривается каждая из программ, входящих в пакет *coreutils* GNU.

Командный интерпретатор

Командный интерпретатор GNU/Linux представляет собой специальную интерактивную программу. Эта программа обеспечивает для пользователей возможность запускать программы, управлять файлами в файловой системе, а также управлять процессами, которые исполняются в системе Linux. Основой командного интерпретатора является командная строка. Командная строка — это интерактивная часть командного интерпретатора. Она позволяет вводить текстовые команды, после чего они интерпретируются и выполняются ядром.

Командный интерпретатор поддерживает ряд внутренних команд, которые могут использоваться для управления такими операциями, как копирование, перемещение и переименование файлов, вывод сведений о программах, исполняемых в настоящее время в системе, а также

останов программ, эксплуатируемых в системе. Кроме выполнения внутренних команд, командный интерпретатор позволяет вводить имена программ в командной строке. Командный интерпретатор передает введенное имя программы ядру для запуска этой программы.

Можно также группировать команды командного интерпретатора в файлах для выполнения их как отдельной программы. Такие файлы принято называть *сценариями командного интерпретатора*. В сценарий командного интерпретатора можно поместить и выполнить в составе группы команд любую команду, которая может быть запущена на выполнение из командной строки. Благодаря этому появляются широкие возможности создания программ для часто используемых команд или процессов, для создания которых требуется группирование нескольких команд.

Для использования в системе Linux предусмотрено довольно большое количество командных интерпретаторов Linux. Различные командные интерпретаторы имеют разные характеристики, причем одни из них в большей степени подходят для создания сценариев, а другие более применимы для управления процессами. По умолчанию во всех дистрибутивах Linux используется командный интерпретатор `bash`, который был разработан в рамках проекта GNU в качестве замены стандартного командного интерпретатора Unix, называемого командным интерпретатором `Bourne` (по имени его создателя — программиста Борна). Имя командного интерпретатора `bash` представляет собой игру слов, поскольку расшифровывается как “снова командный интерпретатор Борна” (`Bourne again shell`).

В этой книге, кроме командного интерпретатора `bash`, будет описано еще несколько других популярных командных интерпретаторов. В табл. 1.2 перечислены различные командные интерпретаторы, которые будут здесь рассматриваться.

Таблица 1.2. Командные интерпретаторы Linux

Командный интерпретатор	Описание
<code>ash</code>	Простой, не требующий больших ресурсов командный интерпретатор, который может применяться в средах с небольшим объемом памяти, но имеет полную совместимость с командным интерпретатором <code>bash</code>
<code>korn</code>	Командный интерпретатор с поддержкой программирования, совместимый с командным интерпретатором <code>Bourne</code> , но обеспечивающий возможность применения таких дополнительных программных средств, как ассоциативные массивы и арифметика с плавающей точкой
<code>tcsh</code>	Командный интерпретатор, который позволяет включать конструкции языка программирования C в сценарии командного интерпретатора
<code>zsh</code>	Усовершенствованный командный интерпретатор, который включает многие средства программ <code>bash</code> , <code>tcsh</code> и <code>korn</code> , предоставляет дополнительные возможности программирования, а также позволяет применять общие файлы журналов и тематические приглашения к вводу информации

Большинство дистрибутивов Linux включает несколько командных интерпретаторов, хотя обычно в них только один командный интерпретатор выбран в качестве применяемого по умолчанию. Если применяемый вами дистрибутив Linux включает несколько командных интерпретаторов, рекомендуем провести эксперименты с различными командными интерпретаторами и определить, какой из них в наибольшей степени соответствует вашим потребностям.

Среда рабочего стола Linux

В первых версиях Linux (выпускавшихся в начале 1990-х годов) единственно доступным был простой текстовый интерфейс к операционной системе Linux. Он позволял администраторам запускать программы, управлять их работой и перемещать файлы в системе.

Однако в связи с распространением операционной системы Microsoft Windows пользователи компьютеров захотели иметь в своем распоряжении нечто большее, чем старый текстовый интерфейс. Такая тенденция стала стимулом к проведению большого объема работ в сообществе разработчиков программ с открытым исходным кодом, в результате чего и в Linux появились графические рабочие столы.

Замечательной особенностью Linux является способность этой операционной системы предоставлять широкий набор возможностей для выполнения любого действия, а графические рабочие столы оказались идеальным средством реализации этих возможностей. В системе Linux имеется большое количество графических рабочих столов, среди которых пользователь может выбрать для себя наиболее подходящий. В следующих разделах описаны некоторые наиболее широко применяемые из них.

Система X Window

Для создания видеосреды необходимы два основных устройства — видеоплата персонального компьютера и монитор. Чтобы можно было воспроизводить с помощью компьютера привлекательное графическое оформление, необходимо предусмотреть в программном обеспечении Linux средства взаимодействия с тем и другим устройством. Основой создания графического представления является программное обеспечение X Window.

Программное обеспечение X Window — это программа низкого уровня, которая работает непосредственно с видеоплатой и монитором в персональном компьютере и управляет теми средствами, с помощью которых приложения Linux могут создавать привлекательные окна и графику на экране компьютера.

Linux — не единственная операционная система, в которой используется программное обеспечение X Window; существуют версии этой программы, написанные для многих операционных систем. Что же касается Linux, то в этой системе есть только два пакета программ, которые могут реализовать функции X Window.

Среди этих двух пакетов раньше всего появился пакет программ XFree86, который в течение долгого времени был единственным пакетом X Window, предназначенным для Linux. Как следует из названия этого пакета, он представляет собой бесплатную версию программного обеспечения X Window с открытым исходным кодом.

Выход в свет более нового из этих двух пакетов, X.org, вызвал весьма значительный интерес в мире Linux, и теперь этот пакет является самым широко применяемым из них. Пакет X.org также представляет собой программную реализацию системы X Window с открытым исходным кодом, но обеспечивает поддержку большего количества новейших видеоплат, применяемых в наши дни.

Оба пакета действуют одинаковым образом, управляя тем, как Linux использует видеоплату, отображая графическое содержимое на мониторе. Для обеспечения этого необходимо вначале настроить пакет для работы в конкретной системе. Эта настройка должна происходить автоматически при установке Linux.

При первоначальной установке дистрибутива Linux вначале предпринимается попытка распознать модели видеоплаты и монитора, а затем создается файл конфигурации X Window, который содержит необходимую информацию. Во время установки можно заметить, что в какой-то момент инсталляционная программа передает в монитор инструкции, позволяющие определить поддерживаемые режимы создания видеоизображения. Иногда в связи с этим на мониторе в течение нескольких секунд отсутствует изображение. В наши дни количество различных типов видеоплат и мониторов весьма велико, поэтому процесс определения применимого режима может потребовать какого-то времени.

Основное программное обеспечение X Window обеспечивает создание среды поддержки графического отображения, но не более того. Разумеется, этого вполне достаточно для выполнения отдельных приложений, но не слишком удобно с точки зрения повседневной эксплуатации компьютера. В частности, еще нет среды рабочего стола, с помощью которого пользователи могут манипулировать файлами или запускать программы. Чтобы устранить этот недостаток, необходимо создать среду рабочего стола на основе программного обеспечения системы X Window.

Рабочий стол KDE

Рабочий стол среды K (K Desktop Environment — KDE) был разработан в 1996 году в рамках проекта с открытым исходным кодом, нацеленного на создание графического рабочего стола, который был бы аналогичен графической среде Microsoft Windows. Рабочий стол KDE включает все средства, с которыми привык работать пользователь Windows. На рис. 1.3 приведен пример рабочего стола KDE 4, который применяется в дистрибутиве openSUSE Linux.



Рис. 1.3. Рабочий стол KDE 4 в системе openSUSE Linux

Рабочий стол KDE позволяет помещать значки приложений и файлов в специальные области на рабочем столе. После одинарного щелчка на значке приложения система Linux запускает это приложение. Одинарный щелчок на значке файла приводит к тому, что программное обеспечение рабочего стола KDE пытается определить, какое приложение должно быть запущено для обработки этого файла.

В нижней части рабочего стола находится полоса, называемая *панелью*. Панель состоит из четырех частей.

- **Меню К.** Во многом аналогично меню Пуск (Start) системы Windows. Содержит ссылки, предназначенные для запуска установленных в системе приложений.
- **Ярлыки программ.** Содержат ссылки для быстрого запуска приложения непосредственно с панели.
- **Панель задач.** На ней расположены значки приложений, которые в настоящее время выполняются на рабочем столе.
- **Апплеты.** Это небольшие приложения, для которых предусмотрены значки на панели, способные изменяться в зависимости от того, какая информация поступает из приложения.

Все средства панели аналогичны тем, которые применяются в системе Windows. Кроме средств рабочего стола, в рамках проекта KDE создан широкий набор приложений, способных работать в среде KDE. Эти приложения приведены в табл. 1.3. (Обратите внимание на то, что в именах приложений KDE часто используется прописная буква “K”.)

Таблица 1.3. Приложения KDE

Приложение	Описание
amaroK	Программа воспроизведения аудиофайлов
digiKam	Программное обеспечение цифровой камеры
dolphin	Диспетчер файлов
K3b	Программное обеспечение для записи CD
Kaffeine	Проигрыватель видеофайлов
Kmail	Почтовый клиент
Koffice	Набор офисных приложений
Konqueror	Диспетчер файлов и веб-браузер
Kontact	Диспетчер личных сведений
Kopete	Клиент системы мгновенного обмена сообщениями

Это — лишь частичный список приложений, созданных в рамках проекта KDE. С рабочим столом KDE поставляется много больше приложений.

Рабочий стол GNOME

Еще одной широко применяемой средой рабочего стола Linux является GNOME (GNU Network Object Model Environment — среда сетевой объектной модели GNU). Среда GNOME была выпущена в 1999 году и стала применяемой по умолчанию средой рабочего стола для многих дистрибутивов Linux (к числу наиболее широко известных из них относится Red Hat Linux).

Создатели среды GNOME решили отступить от стандартного внешнего вида Microsoft Windows, но включили многие средства, с которыми привыкли работать большинство пользователей Windows, в частности следующие.

- Область рабочего стола для значков.
- Две области, оформленные в виде панелей.
- Перетаскиваемые значки.

На рис. 1.4 показан стандартный рабочий стол GNOME, используемый в дистрибутиве Ubuntu Linux.

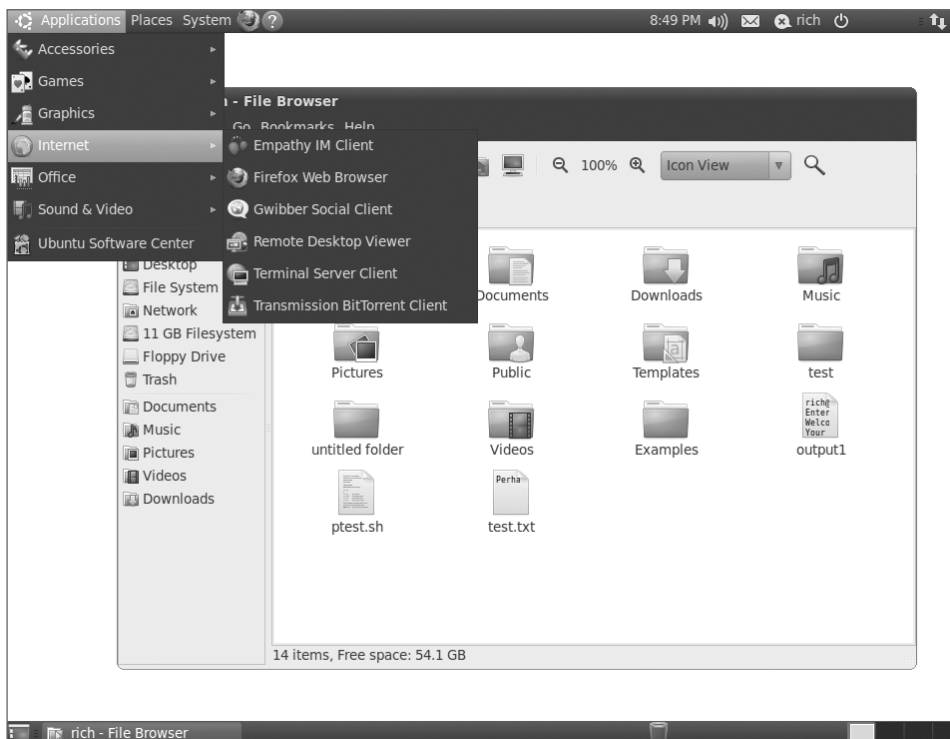


Рис. 1.4. Рабочий стол GNOME в системе Ubuntu Linux

Чтобы не отставать от KDE, разработчики GNOME создали большое количество графических приложений, входящих в состав рабочего стола GNOME (табл. 1.4).

Вполне очевидно, что рабочий стол GNOME также поддерживает весьма значительное количество приложений. Кроме всех этих приложений, большинство дистрибутивов Linux, в которых используется рабочий стол GNOME, включают также библиотеки KDE, позволяющие эксплуатировать приложения KDE на рабочем столе GNOME.

Таблица 1.4. Приложения GNOME

Приложение	Описание
epiphany	Веб-браузер
evince	Средство просмотра документов
gcalc-tool	Калькулятор
gedit	Текстовый редактор GNOME
gnome-panel	Панель рабочего стола для запуска приложений
gnome-nettool	Сетевой диагностический инструмент
gnome-terminal	Эмулятор терминала

Приложение	Описание
nautilus	Графический диспетчер файлов
nautilus-cd-burner	Программа записи CD
sound juicer	Инструмент для воспроизведения CD и извлечения содержимого дорожек дисков
tomboy	Программное обеспечение ведения заметок
totem	Программа воспроизведения мультимедийной информации

Другие рабочие столы

Недостатком среды графического рабочего стола является то, что для обеспечения ее надлежащего функционирования требуется значительное количество системных ресурсов. Разработчики первых версий Linux рекламировали и стремились подчеркнуть такую особенность этой операционной системы, как возможность ее эксплуатации на старых, менее мощных персональных компьютерах, на которых невозможно было применять более новые продукты для рабочего стола Microsoft. Однако по мере дальнейшего распространения рабочих столов KDE и GNOME ситуация изменилась, поскольку для эксплуатации рабочего стола KDE или GNOME требуется почти столько же памяти, сколько и для новейших версий среды рабочего стола Microsoft.

Однако это не должно вызывать разочарование у владельцев более старых персональных компьютеров. Разработчики Linux предприняли большие усилия для того, чтобы вернуть эту операционную систему к ее истокам. В частности, было создано значительное количество графических приложений для рабочего стола, не требующих больших затрат памяти, которые поддерживают основные необходимые функции и идеально работают на старых персональных компьютерах.

На основе этих графических рабочих столов разработано не столь значительное количество приложений, но в их число входят такие приложения, которые обеспечивают выполнение многих важных функций, таких как обработка текстов, работа с электронными таблицами, базами данных, рисунками и, безусловно, поддержка мультимедиа.

В табл. 1.5 перечислены некоторые варианты среды графического рабочего стола Linux с меньшими возможностями, которые могут использоваться на менее мощных персональных компьютерах и ноутбуках.

Таблица 1.5. Прочие графические рабочие столы Linux

Рабочий стол	Описание
fluxbox	Предельно простой рабочий стол, который не включает панель, а поддерживает только раскрывающееся меню для запуска приложений
xfce	Рабочий стол, аналогичный рабочему столу KDE, но с меньшим количеством графических средств, предназначенный для вариантов среды, не требующих значительных объемов памяти
JWM	Рабочий стол JWM (Joe's Window Manager) — весьма упрощенный рабочий стол, который идеально подходит для среды с малым объемом памяти и небольшим количеством свободного места на диске
fwm	Поддерживает некоторые дополнительные средства рабочего стола, такие как виртуальные рабочие столы и панели, но может работать в среде с малым объемом памяти
fwm95	Рабочий стол, созданный на основе fwm, который выглядит во многом аналогично рабочему столу Windows 95

Эти варианты среды графического рабочего стола не столь богаты своими возможностями, как рабочие столы KDE и GNOME, но вполне успешно поддерживают основные графические функциональные средства. На рис. 1.5 показано, как выглядит рабочий стол fluxbox, используемый в дистрибутиве antiX системы Puppy Linux.



Рис. 1.5. Внешний вид рабочего стола fluxbox в дистрибутиве Puppy Linux

Если в вашем распоряжении имеется какой-то старый персональный компьютер, попробуйте установить один из дистрибутивов Linux, в котором используется подобный рабочий стол, и посмотрите, что из этого выйдет. Вы можете быть приятно удивлены.

Дистрибутивы Linux

Итак, мы рассмотрели четыре основных компонента, необходимых для формирования полноценной системы Linux, и можем перейти к изучению вопроса, как эти компоненты сочетаются друг с другом, создавая систему Linux. К счастью, эта задача уже была успешно решена.

Полный пакет системных программ Linux называется *дистрибутивом*. Уже подготовлено весьма значительное количество различных дистрибутивов Linux, позволяющих удовлетворить почти любые требования к созданию вычислительной среды, которые только могут встретиться. Большинство дистрибутивов рассчитано на конкретные группы пользователей, таких как бизнесмены, любители мультимедийных развлечений, разработчики программ или обычные домашние пользователи. Каждый дистрибутив, разработанный в соответствии с определенными требованиями, включает пакеты программ, необходимые для поддержки таких специализи-

рованных функций, как редактирование аудио- и видеофайлов для работы с мультимедиа или поддержка интегрированной среды разработки (integrated development environment — IDE) для создания программ.

Различные дистрибутивы Linux принято подразделять на три категории.

- Полнофункциональные основные дистрибутивы Linux.
- Специализированные дистрибутивы.
- Тестовые дистрибутивы LiveCD.

Эти типы дистрибутивов Linux описаны в следующих разделах, а также приведены некоторые примеры дистрибутивов Linux, относящихся к каждой категории.

Основные дистрибутивы Linux

Каждый основной дистрибутив Linux включает ядро, один или несколько вариантов среды графического рабочего стола, а также почти все имеющиеся на данный момент приложения Linux, предварительно откомпилированные для применяемого ядра. Таким образом, основной дистрибутив позволяет получить сразу все необходимое для конкретной установки Linux. В табл. 1.6 перечислены некоторые из наиболее широко применяемых основных дистрибутивов Linux.

Таблица 1.6. Основные дистрибутивы Linux

Дистрибутив	Описание
Slackware	Один из первых разработанных наборов дистрибутивов Linux, который нравится компьютерным фанатам Linux
Red Hat	Коммерческий дистрибутив для деловых предприятий, используемый главным образом при развертывании серверов для Интернета
Fedora	Один из вариантов дистрибутива Red Hat, предназначенный для домашнего применения
Gentoo	Дистрибутив, разработанный для высокопрофессиональных пользователей Linux, который содержит только исходный код Linux
Mandriva	Дистрибутив, предназначенный главным образом для домашнего применения (который ранее носил название Mandrake)
openSuSe	Различные дистрибутивы для бизнеса и домашнего применения
Debian	Дистрибутив, чаще всего применяемый экспертами Linux, предназначенный для развертывания коммерческих продуктов Linux

Первые версии системы Linux поставлялись в виде дистрибутивов, оформленных как пакет гибких дисков. При этом для установки системы приходилось загружать все необходимые группы файлов, а затем копировать их на диски. Обычно требовалось двадцать или больше дисков, чтобы получить полный дистрибутив! Само собой разумеется, что работа с многочисленными дисками требовала больших усилий.

В настоящее время почти все домашние компьютеры комплектуются устройствами воспроизведения CD и DVD, поэтому дистрибутивы Linux выпускаются либо на нескольких CD, либо на одном DVD. Благодаря этому инсталляция Linux стала намного проще.

Тем не менее пользователи на первых порах часто сталкиваются с проблемами, устанавливая один из основных дистрибутивов Linux. Разработчикам Linux приходится включать в каждый отдельный дистрибутив большое количество разнообразного прикладного программного обеспечения, чтобы учесть почти любую ситуацию, в которой может потребоваться приме-

ние системы Linux. Поэтому дистрибутив включает все, начиная с высококачественных серверов баз данных с поддержкой Интернета и заканчивая обычными играми. При этом общее количество доступных приложений для Linux весьма велико, в связи с чем полный дистрибутив часто занимает от четырех и больше CD.

Безусловно, чем больше опций доступно в дистрибутиве, тем больший интерес он представляет для компьютерных фанатов Linux, но это может стать сложным препятствием для начинающих пользователей Linux. При установке большинства дистрибутивов пользователю приходится отвечать на целый ряд вопросов инсталляционной программы, которая пытается определить, какие приложения должны загружаться по умолчанию, какие аппаратные средства подключены к персональному компьютеру и как настроить аппаратные средства. Пользователи на первых порах часто сталкиваются с затруднениями, отвечая на эти вопросы. В результате такие пользователи либо устанавливают на своем компьютере слишком много программ, либо допускают противоположную ошибку и в дальнейшем обнаруживают, что компьютер не может выполнять те функции, которые им требуются.

К счастью, начинающие пользователи могут прибегнуть к значительно более простому способу установки Linux.

Специализированные дистрибутивы Linux

В последнее время все чаще появляются дистрибутивы Linux, относящиеся к новой категории. Эти дистрибутивы, как правило, базируются на одном из основных дистрибутивов, но содержат только такое подмножество приложений, которое применимо лишь в конкретной области.

Такие индивидуализированные дистрибутивы Linux не только предоставляют все необходимое специализированное программное обеспечение (например, только те офисные продукты, которые требуются для бизнес-пользователей), но и помогают начинающему пользователю Linux при установке системы, автоматически обнаруживая и настраивая обычно применяемые аппаратные средства. Благодаря этому установка Linux становится гораздо менее трудоемким процессом.

В табл. 1.7 перечислены некоторые из имеющихся специализированных дистрибутивов Linux и показано, к какой области они относятся.

Таблица 1.7. Специализированные дистрибутивы Linux

Дистрибутив	Описание
Xandros	Коммерческий пакет Linux, настроенный с учетом новых пользователей
SimplyMEPIS	Бесплатный дистрибутив для домашнего применения
Ubuntu	Бесплатный дистрибутив для работы в школе и дома
PCLinuxOS	Бесплатный дистрибутив для работы дома и в офисе
Mint	Бесплатный дистрибутив для использования в домашних развлечениях
dyne:bolic	Бесплатный дистрибутив, предназначенный для работы с приложениями аудио и MIDI
Puppy Linux	Небольшой бесплатный дистрибутив, который хорошо работает на персональных компьютерах старых типов

В этой таблице приведена лишь небольшая часть специализированных дистрибутивов Linux. Количество специализированных дистрибутивов Linux исчисляется буквально сотнями, причем в Интернете постоянно обнаруживаются все новые и новые дистрибутивы. В какой бы области ни работал пользователь, весьма велика вероятность того, что он найдет подходящий для себя дистрибутив Linux.

Многие из специализированных дистрибутивов Linux разработаны на основе дистрибутива Debian Linux. В них используются такие же инсталляционные файлы, как и в версии Debian, но поддерживается лишь небольшая часть полномасштабной системы Debian.

Дистрибутивы Linux категории LiveCD

Относительно новым явлением в мире Linux стало создание загружаемых дистрибутивов Linux на CD. Такие дистрибутивы позволяют ознакомиться с той или иной системой Linux, фактически не занимаясь ее установкой. Наиболее современные персональные компьютеры предоставляют возможность выполнять начальную загрузку с CD вместо стандартного жесткого диска. Разработчики дистрибутивов Linux воспользовались этим и создали дистрибутивы Linux на загружаемых CD, которые содержат типичную систему Linux (такие дистрибутивы получили название *Linux LiveCD*). В связи с ограниченным объемом отдельного CD в состав такой типичной системы невозможно включить все средства полной системы Linux, но часто достойно удивления то, насколько велико разнообразие программного обеспечения, которое разработчикам удается предоставить в распоряжение пользователей. Результатом становится то, что пользователи могут загружать свой персональный компьютер с CD и эксплуатировать дистрибутив Linux, не будучи вынужденными вообще что-либо устанавливать на жестком диске своего компьютера!

Это — превосходный способ проверки различных дистрибутивов Linux без нарушения работы используемого персонального компьютера. Достаточно лишь вставить CD в устройство чтения и выполнить начальную загрузку! Все программное обеспечение Linux будет эксплуатироваться непосредственно с помощью CD. Количество версий Linux LiveCD, которые можно загрузить из Интернета и записать на CD для испытательного прогона, весьма велико.

В табл. 1.8 перечислены некоторые доступные версии Linux LiveCD, которые нашли широкое применение.

Таблица 1.8. Дистрибутивы Linux LiveCD

Дистрибутив	Описание
Knoppix	Система Linux на немецком языке; первый из разработанных вариантов Linux LiveCD
SimplyMEPIS	Предназначена для начинающих пользователей Linux, которые применяют эту систему дома
PCLinuxOS	Полноценный дистрибутив Linux на LiveCD
Ubuntu	Международный проект Linux, предназначенный для поддержки многих национальных языков
Slax	Версия Linux LiveCD на основе Slackware Linux
Puppy Linux	Полнофункциональная система Linux, предназначенная для персональных компьютеров старых типов

Читатель может обнаружить в этой таблице знакомые названия. Для многих специализированных дистрибутивов Linux теперь выпущена также версия Linux LiveCD. Есть и такие дистрибутивы Linux LiveCD, как, например, Ubuntu, которые позволяют устанавливать дистрибутив Linux непосредственно с LiveCD. Это позволяет выполнить загрузку с CD, провести испытательный прогон дистрибутива Linux, определить, является ли он приемлемым, затем установить на жестком диске своего компьютера. Такая возможность является чрезвычайно удобной и понятной для пользователя.

Но, как и все хорошее, версии Linux LiveCD имеют свои недостатки. В частности, доступ ко всем приложениям осуществляется с помощью CD, поэтому работа замедляется, особенно если используются более старые, не столь быстродействующие компьютеры и устройства чте-

ния CD. Кроме того, уже записанные CD не допускают перезаписи, поэтому любые изменения, внесенные пользователем в систему Linux, удаляются после следующей перезагрузки.

Тем не менее в мире Linux LiveCD появились усовершенствования, позволяющие решить некоторые из этих проблем. К ним относятся следующие возможности.

- Копирование системных файлов Linux с CD в память.
- Копирование системных файлов в файлы на жестком диске.
- Сохранение параметров настройки системы во флеш-накопителе USB.
- Сохранение пользовательских настроек во флеш-накопителе USB.

Некоторые версии Linux LiveCD, например Puppy Linux, разработаны в целях сведения к минимуму количества системных файлов Linux. При загрузке с CD сценарии начальной загрузки LiveCD копируют системные файлы непосредственно в оперативную память. Это позволяет извлечь CD из компьютера сразу после загрузки Linux. Таким образом, приложения не только работают намного быстрее (поскольку при вызове из памяти быстродействие приложений значительно увеличивается), но и освобождается лоток устройства чтения CD, который может применяться для извлечения дорожек из звуковых CD или воспроизведения видео с DVD с помощью программного обеспечения, входящего в состав версии Puppy Linux.

В других версиях Linux LiveCD используется альтернативный метод, который также позволяет извлечь CD из лотка после загрузки. Этот метод предусматривает копирование основных файлов Linux на жесткий диск Windows в виде отдельного файла. После начальной загрузки с CD происходит поиск такого файла и чтение из него системных файлов. Данный метод, получивший название *закрепление*, используется в версии dyne:bolic Linux LiveCD. Разумеется, прежде чем выполнять начальную загрузку с CD, необходимо скопировать системный файл на жесткий диск.

Широко применяемый метод сохранения данных после сеанса работы с версией Linux LiveCD состоит в использовании обычного флеш-накопителя USB (такие накопители называют также флеш-дисками, или просто флешками). Почти любая версия Linux LiveCD способна распознавать вставленный флеш-накопитель USB (даже если эта флешка отформатирована для Windows), а также выполнять на нем операции чтения и записи. Это позволяет выполнить загрузку с помощью Linux LiveCD, воспользоваться приложениями Linux для создания файлов, сохранить эти файлы во флеш-накопителе, а затем получить к ним доступ из приложения Windows (или перенести на другой компьютер). Не правда ли, это великолепная возможность!

Резюме

В настоящей главе рассматривается система Linux и приведены основные сведения о ее работе. Основой системы является ядро Linux, которое обеспечивает взаимодействие памяти, программ и аппаратных средств. Еще одной важной частью системы Linux являются программы GNU. Командный интерпретатор Linux, описание которого является основной темой настоящей книги, входит в состав этих базовых программ GNU. Кроме того, в данной главе рассматривается последняя часть системы Linux — среда рабочего стола Linux. Система Linux применяется уже много лет, а в последнее время стала поддерживать несколько вариантов среды графического рабочего стола.

В настоящей главе рассматриваются также различные дистрибутивы Linux. В дистрибутиве Linux объединены различные части системы Linux в виде простого в использовании пакета, который можно легко установить на конкретном персональном компьютере. Дистрибутивы Linux подразделяются на такие категории, как полномасштабные дистрибутивы Linux, включающие

почти все приложения, которые можно только себе представить, и специализированные дистрибутивы Linux, в состав которых входят лишь приложения, предназначенные для выполнения специальных функций. Повальное увлечение версиями Linux LiveCD привело к созданию еще одной категории дистрибутивов Linux, которые позволяют легко провести испытательный прогон системы Linux без установки ее на жестком диске компьютера.

В следующей главе будет показано, что требуется для запуска интерфейса командной строки и приобщения к возможностям создания сценариев для командного интерпретатора. В ней будет показано, как обратиться к программе командного интерпретатора Linux из этой великолепной среды графического рабочего стола с привлекательным оформлением. В наши дни это не всегда достаточно просто.

Получение доступа к командному интерпретатору

В первых версиях Linux единственным доступным для работы в системе был командный интерпретатор. Поэтому системным администраторам, программистам и пользователям системы при вводе текстовых команд и просмотре текстового вывода приходилось пользоваться лишь операторским терминалом Linux. В наши дни, в связи с распространением различных вариантов среды графического рабочего стола с привлекательным оформлением, ситуация изменилась на противоположную, и стало сложно даже обнаружить приглашение командного интерпретатора в системе, чтобы в нем поработать. В настоящей главе описано создание среды командной строки и дан краткий обзор пакетов эмуляции терминала, с которыми приходится сталкиваться в различных дистрибутивах Linux.

Эмуляция терминала

Прежде, до столь широкого распространения графических рабочих столов, единственный способ взаимодействия с системой Unix заключался в использовании текстового *интерфейса командной строки* (command line interface — CLI), предоставляемого командным интерпретатором. Интерфейс командной строки допускает только текстовый ввод и способен отображать лишь текстовый и элементарный графический вывод.

В связи с этим ограничением невозможно было формировать на устройствах вывода результаты с достаточно

ГЛАВА

2

В этой главе...

Эмуляция терминала

База данных terminfo

Консоль Linux

Терминал xterm

Терминал Konsole

Терминал GNOME

Резюме

привлекательным оформлением. Чаще всего единственно необходимым условием для взаимодействия с системой Unix был простой терминал ввода-вывода. Причем обычно для создания терминала ввода-вывода было достаточно иметь лишь монитор и клавиатуру (хотя в дальнейшем на практике такие терминалы стали предоставлять дополнительные возможности благодаря использованию мыши), подключенные к системе Unix через кабель связи (обычно многопроводной последовательный кабель). Это простое сочетание устройств предоставляло легкий способ ввода текстовых данных в систему Unix и просмотра текстовых результатов.

Как известно, в современной среде Linux дела обстоят иначе. Почти в каждом дистрибутиве Linux используется среда графического рабочего стола того или иного типа. Однако для доступа к командному интерпретатору все еще требуется текстовый дисплей, обеспечивающий работу с интерфейсом командной строки. С этой проблемой сталкивается каждый, кому приходится обращаться непосредственно к системе. В связи с распространением всех этих новых и замечательных графических средств рабочего стола Linux иногда нелегко найти способ воспользоваться интерфейсом командной строки, предусмотренным в дистрибутиве Linux.

Один из вариантов перехода к интерфейсу командной строки состоит в том, чтобы вывести систему Linux из режима графического рабочего стола и поместить ее в текстовый режим. В текстовом режиме на мониторе разворачивается лишь простой интерфейс командной строки командного интерпретатора, точно так же, как и в дни, предшествующие появлению графических рабочих столов. Этот текстовый режим принято также называть режимом *консоли Linux*, поскольку он эмулирует стародавний аппаратный операторский терминал и предоставляет непосредственный интерфейс к системе Linux.

Альтернативой разворачиванию консоли Linux является применение *пакета эмуляции терминала*, вызываемого на выполнение из среды графического рабочего стола Linux. Пакет эмуляции терминала моделирует работу на терминале ввода-вывода, не выходя за пределы одного из графических окон на рабочем столе. На рис. 2.1 приведен пример выполнения программы эмулятора терминала в среде графического рабочего стола Linux.

Каждый пакет эмуляции терминала обладает способностью эмулировать терминалы ввода-вывода одного или нескольких конкретных типов. Всем, кто собирается работать с командным интерпретатором в системе Linux, к сожалению, приходится приобретать определенные знания об эмуляции терминала.

Зная о том, каковыми являются основные средства старых терминалов ввода-вывода, вы сможете принять решение о выборе конкретного типа эмуляции при работе с графическим эмулятором терминала и в полной мере воспользоваться всеми имеющимися средствами. Основные средства, предусмотренные в терминале ввода-вывода, подразделяются на две области: графические возможности и клавиатура. В настоящем разделе рассматриваются эти средства и описано, какое отношение они имеют к различным типам эмуляторов терминала.

Графические возможности

Наиболее важная часть эмуляции терминала состоит в определении способа отображения информации на мониторе. Сталкиваясь с термином “текстовый режим”, обычно никто не думает, что и в этом режиме нельзя обойтись без графики. Однако даже самые простые по своему устройству терминалы ввода-вывода поддерживают определенные операции манипулирования экраном (такие как очистка экрана и отображение текста в конкретном местоположении на экране).

В данном разделе рассматриваются графические средства, применение которых становится отличительным признаком каждого из разных типов терминалов, а также описано, из чего состоят пакеты эмуляции терминала.

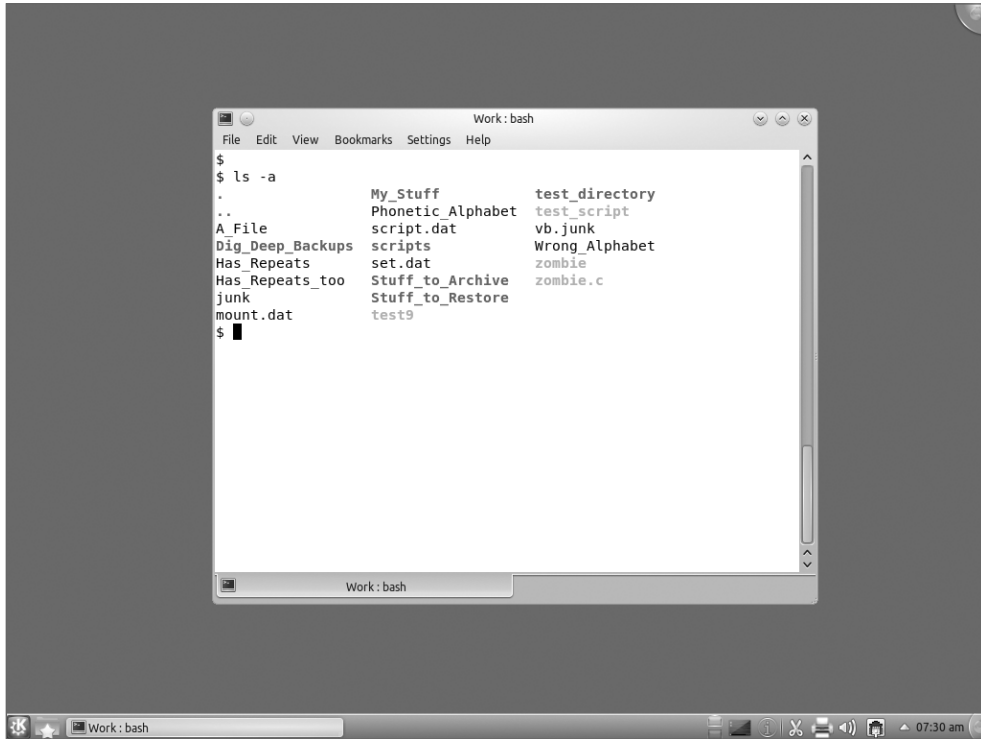


Рис. 2.1. Простой эмулятор терминала, эксплуатируемый на рабочем столе Linux

Кодировки

Все терминалы должны отображать символы на экране (в противном случае текстовый режим был бы довольно бесполезным). Сложность состоит в том, какие символы необходимо отображать и какие коды должна передавать система Linux для их отображения. *Кодировка* — это ряд двоичных команд, передаваемых системой Linux на монитор для отображения символов. Предусмотрено несколько кодировок, поддерживаемых различными пакетами эмуляции терминала, которые перечислены ниже.

- Кодировка ASCII. Стандартный американский код обмена информацией (American Standard Code for Information Interchange). Эта кодировка содержит английские символы, представленные в памяти с использованием 7-битового кода, и состоит из английских букв (в верхнем и нижнем регистре), цифр и специальных символов, общее количество которых составляет 128. Кодировка ASCII была утверждена Американским национальным институтом стандартов (American National Standards Institute — ANSI) под заголовком US-ASCII. Часто приходится видеть, что при описании эмуляторов терминала она упоминается как кодировка ANSI.
- Кодировка ISO-8859-1 (обычно называемая Latin-1). Расширение кодировки ASCII, разработанное Международной организацией по стандартизации (International Organization for Standardization — ISO). В ней используется 8-битовый код для поддержки стандартных символов ASCII, а также специальных символов иностранных алфавитов для боль-

шинства западноевропейских языков. Кодировка Latin-1 широко применяется в многонациональных пакетах эмуляции терминала.

- Кодировка ISO-8859-2. Кодировка ISO, которая поддерживает языковые символы восточноевропейских языков.
- Кодировка ISO-8859-6. Кодировка ISO, которая поддерживает языковые символы арабского языка.
- Кодировка ISO-8859-7. Кодировка ISO, которая поддерживает языковые символы греческого языка.
- Кодировка ISO-8859-8. Кодировка ISO, которая поддерживает языковые символы еврейского языка (иврит).
- Кодировка ISO-10646 (обычно называемая Юникодом): Двухбайтовая кодировка ISO, которая содержит коды для большинства языков с латинскими и другими алфавитами. Эта единственная кодировка содержит все коды, определенные во всех кодировках ряда ISO-8859-х. Кодировка Юникода в последнее время становится все более популярной среди разработчиков приложений с открытым исходным кодом.

Безусловно, кодировкой, которая в наши дни находит наиболее широкое распространение в англоязычных странах, является Latin-1. Все чаще применяется также кодировка Юникода, которая в один прекрасный день вполне может стать новым стандартом среди кодировок. Наиболее широко применяемые эмуляторы терминала позволяют выбирать кодировки, предназначенные для использования при эмуляции терминала.

Управляющие коды

Терминалы должны не только обладать способностью отображать символы, но и управлять специальными функциями на мониторе и клавиатуре, такими как задание местоположения курсора на экране. Эти функции осуществляются с использованием системы *управляющих кодов*. Управляющий код представляет собой специальный код, не используемый в кодировке, который передает терминалу сигнал на выполнение специальной операции, не приводящей к выводу символа на экран.

Широко применяемыми функциями управляющего кода являются возврат каретки (возврат курсора к началу строки), перевод строки (помещение курсора на следующую горизонтальную строку), горизонтальная табуляция (смещение курсора на заранее заданное количество пробелов), отработка действий клавиш со стрелками (вверх, вниз, влево и вправо) и действий клавиш перелистывания страниц вверх и вниз. Эти коды главным образом эмулируют функции, управляющие размещением курсора на мониторе, но есть также несколько других кодов, таких как очистка всего экрана и даже воспроизведение звука колокольчика (который эмулирует звонок колокольчика на пишущей машинке при достижении конца каретки).

Есть и такие управляющие коды, которые обеспечивают управление средствами связи терминалов ввода-вывода. Терминалы ввода-вывода подключаются к компьютерной системе через канал связи определенного типа, основой которого часто служит кабель последовательной передачи данных. Иногда возникает необходимость управления данными в канале связи, поэтому разработчики предусмотрели специальные управляющие коды, предназначенные для использования исключительно в области передачи данных. Разумеется, эти коды не обязательно могут потребоваться в современных эмуляторах терминала, но большинство из них поддерживает эти коды для обеспечения совместимости. Наиболее широко применяемыми кодами этой категории являются коды XON и XOFF, предназначенные для запуска и останова передачи данных на терминал соответственно.

Графика блочного режима

По мере роста популярности первых терминалов ввода-вывода их изготовители стали все чаще экспериментировать в области реализации элементарных графических возможностей. Безусловно, к числу “графических” терминалов ввода-вывода, наиболее часто применяемых в мире Unix, принадлежали терминалы корпорации DEC (Digital Equipment Corporation) ряда VT. Поворотный момент в развитии терминалов ввода-вывода наступил с выпуском устройства DEC VT100 в 1978 году. Терминал DEC VT100 был первым терминалом, поддерживающим полную кодировку ANSI, включая символы псевдографики блочного режима.

Кодировка ANSI содержит коды, которые позволяют мониторам отображать не только текст, но и простейшие графические символы, такие как T-образные, Г-образные, крестообразные знаки и отрезки прямых различной ориентации. Но терминалом, который далеко превзошел по своей популярности все прочие модели терминалов, применявшиеся при эксплуатации системы Unix в 1980-х годах, оказался DEC VT102, обновленная версия терминала VT100. Большинство современных программ эмуляции терминала все еще эмулируют работу дисплея VT102, поддерживая все коды ANSI, предназначенные для создания графики блочного режима.

Векторная графика

Вслед за этим компания Tektronix организовала выпуск популярного ряда терминалов, в которых был применен метод отображения, называемый *векторной графикой*. Создатели векторной графики отказались от применявшегося в терминалах DEC метода формирования графики блочного режима и перешли к формированию всех экранных изображений (включая символы) в виде ряда отрезков линий (получивших название *векторов*). Терминал Tektronix 4010 оказался в числе наиболее широко применяемых графических терминалов ввода-вывода из всех, которые когда-либо выпускались. Во многих пакетах эмуляции терминала все еще эмулируются его возможности.

В терминале 4010 для создания изображения происходит вычерчивание ряда векторов с использованием электронного луча, во многом аналогично тому, как формируется рисунок карандашом. В векторной графике не используются точки для рисования линий, поэтому она позволяет изображать геометрические формы с более высокой точностью по сравнению с большинством точечных графических терминалов. К этой возможности проявили особо большой интерес математики и ученые из других областей.

В современных эмуляторах терминала для эмуляции изобразительных возможностей векторной графики терминалов Tektronix 4010 используется программное обеспечение. Это средство все еще остается популярным среди специалистов, которым требуются точные графические изображения, а также среди тех пользователей, которые эксплуатируют приложения, основанные на применении процедур векторной графики для рисования сложных графиков и диаграмм.

Буферизация отображения

Ключом к организации успешной работы графического дисплея является способность терминала к буферизации данных. Для буферизации данных требуется применение дополнительной внутренней памяти в самом терминале для хранения символов, не отображаемых в настоящее время на мониторе.

В терминалах ряда DEC VT использовались следующие два типа буферизации данных.

- Буферизация данных в ходе их прокрутки за пределы основного окна экрана (этот способ принято называть *ведением журнала*).
- Буферизация полностью в отдельном окне экрана (это — *применение дополнительного экрана*).

Буферизация первого типа основана на использовании так называемой *области прокрутки* — определенной части памяти терминала, которая позволяет “запоминать” данные, которые вышли в результате прокрутки за пределы экрана. Стандартный терминал DEC VT102 имел область просмотра, состоящую из 25 строк символов. По мере того как на терминале отображается новая строка символов, происходит прокрутка вверх предшествующих ей строк. После достижения на терминале нижней строки отображения ввод следующей строки приводит к тому, что верхняя строка в результате прокрутки выходит за пределы отображения.

Внутренняя память терминала VT102 позволяла сохранять в нем последние 64 строки, которые вышли после прокрутки за пределы отображения. Пользователи имели возможность фиксировать текущее состояние отображения на экране и пользоваться клавишами со стрелками для прокрутки назад через предыдущие строки, которые перед этим были “убраны” из-за прокрутки вперед с экрана. Пакеты эмуляции терминала позволяют использовать для прокрутки правую линейку прокрутки или колесико прокрутки мыши для просмотра сохраненных данных без необходимости фиксировать отображение. Безусловно, для обеспечения полной совместимости эмуляции в большинстве пакетов эмуляции терминала предусмотрена также возможность фиксировать отображение на дисплее и использовать клавиши со стрелками и клавиши перелистывания страниц вверх и вниз для просмотра сохраненных данных.

Буферизация второго типа основана на применении так называемого *дополнительного экрана*. Как правило, на терминале вывод данных осуществляется непосредственно в обычную область отображения на мониторе. Но был разработан метод, позволяющий в примитивной форме реализовать анимацию путем применения двух областей экрана для хранения данных. Для этого использовались управляющие коды, которые служили командами для терминала, чтобы запись данных осуществлялась на дополнительном экране вместо текущего экрана отображения. Эти данные сохранялись в памяти. Другие управляющие коды передавали терминалу сигнал о том, что изображение на мониторе должно быть переключено с данных, отображаемых с помощью обычного экрана, на данные, относящиеся к дополнительному экрану, и это переключение происходило почти мгновенно. Путем сохранения подряд идущих страниц данных в области дополнительного экрана с их последующим отображением можно было грубо моделировать движущиеся графические изображения.

Терминалы, эмулирующие терминалы ряда VT, обладают способностью поддерживать оба метода буферизации данных — и область прокрутки, и дополнительный экран.

Цветное изображение

Программисты экспериментировали с различными способами представления данных даже в те дни, когда существовали лишь терминалы ввода-вывода с черно-белым (или скорее с черно-зеленым) изображением. Большинство терминалов поддерживало специальные управляющие коды, позволяющие формировать специальный текстовый вывод следующих типов.

- Полужирные символы.
- Подчеркнутые символы.
- Негативное изображение (черные символы на белом фоне).
- Мигание.
- Различные сочетания указанных средств.

В те дни для привлечения особого внимания пользователей программисты применяли вывод текста полужирным шрифтом, включали мигание или формировали негативное изображение текста. Просмотр такого текста очень утомляет зрение. А теперь применяются другие приемы, не менее утомительные!

После ввода в действие терминалов с цветным изображением программисты добавили специальные управляющие коды, с помощью которых текст отображается в различных цветах и оттенках. Кодировка ANSI включает управляющие коды, которые позволяют задавать конкретные цвета и для цвета текста (цвета переднего плана), и для цвета фона, отображаемого на мониторе. Большинство эмуляторов терминалов также поддерживает управляющие коды ANSI, относящиеся к цветному изображению.

Клавиатура

Назначение терминала состоит не только в формировании изображения на мониторе. Те, кому когда-либо приходилось сталкиваться с различными терминалами ввода-вывода, могли заметить, что на их клавиатуре имеется множество разных клавиш, кроме тех, которые служат для ввода символов. На клавиатуре обычного компьютера этих клавиш уже нет, поэтому разработчикам пакетов эмуляции терминала приходится решать сложную задачу эмуляции некоторых особых клавиш терминала ввода-вывода.

Дело в том, что в свое время разработчики клавиатуры для персонального компьютера отказались от мысли предусматривать клавиши, соответствующие всевозможным типам специальных клавиш, необходимых для терминалов ввода-вывода. Были и такие изготовители персональных компьютеров, которые проводили эксперименты по применению специальных клавиш для выполнения каких-то особых функций, но в конечном итоге состав клавиш клавиатуры персонального компьютера стал в большей степени стандартизированным.

Чтобы пакет эмуляции терминала мог полностью эмулировать терминал ввода-вывода конкретного типа, он должен предусматривать соответствие даже для всех тех клавиш терминала ввода-вывода, которые отсутствуют на клавиатуре персонального компьютера. При этом приходится применять такие средства обеспечения соответствия, которые часто приводят к путанице, особенно по той причине, что в различных системах используются разные управляющие коды для одних и тех же клавиш.

Ниже перечислены некоторые специальные клавиши, которые предусмотрены во многих пакетах эмуляции терминала.

- Прерывание. После нажатия клавиши прерывания в базовое устройство передается поток нулевых символов. Эта клавиша часто используется для прерывания программы, выполняемой в настоящее время в командном интерпретаторе.
- Блокировка прокрутки. Клавиша блокировки прокрутки, называемая также клавишей останова прокрутки, останавливает вывод на дисплее. В некоторых терминалах предусмотрена дополнительная память для хранения показанного ранее содержимого дисплея, чтобы пользователь мог выполнять прокрутку назад по строкам, выведенным за пределы изображения, в то время как включен режим блокировки прокрутки.
- Повторение. Клавиша повторения, будучи нажатой вместе с другой клавишей, вынуждает терминал снова и снова отправлять в базовое устройство значение этой другой клавиши.
- Возврат. Клавиша возврата (называемая также клавишей ввода) обычно используется для отправки в базовое устройство символа возврата каретки. Эта клавиша чаще всего применяется для указания на завершение ввода команды, которая должна быть обработана базовым устройством (теперь на клавиатуре персонального компьютера эта клавиша обозначается как <Enter>).
- Удаление. Клавиша удаления, которая на первый взгляд должна выполнять несложную функцию, вызывает затруднения у разработчиков пакетов эмуляции терминала. На не-

которых терминалах эта клавиша применяется для удаления символа в текущей позиции курсора, а на других после ее нажатия происходит удаление предыдущего символа. В целях устранения этого разногласия на клавиатурах персонального компьютера предусмотрены две клавиши удаления — `<Delete>` и `<Backspace>`.

- Клавиши со стрелками. Клавиши со стрелками обычно используются для перемещения курсора в определенное место, например, при просмотре или правке листинга.
- Функциональные клавиши. На терминалах ввода-вывода они представляли собой комбинации специализированных клавиш, с которыми можно было связывать уникальные функции в программах. Эти клавиши аналогичны клавишам `<F1>`–`<F12>` персонального компьютера. В терминалах ряда DEC VT фактически имеются два набора функциональных клавиш `<F1>`–`<F20>` и `<PF1>`–`<PF4>`.

Важнейшим элементом пакета эмуляции терминала является эмуляция клавиатуры. К сожалению, разработчики часто создают такие приложения, которые требуют от пользователя нажатия каких-то конкретных клавиш для выполнения определенных функций. Автору приходилось сталкиваться с многими пакетами связи, в которых применялись клавиши `<PF1>`–`<PF4>` старых моделей терминала DEC, которые обычно можно с трудом найти на клавиатуре, применяя пакет эмуляции терминала.

База данных terminfo

Как описано выше, пакеты эмуляции терминала способны эмулировать терминалы различных типов, поэтому должен быть предусмотрен способ передачи системе Linux указания о том, какой именно терминал должен быть эмулирован. Дело в том, что система Linux должна иметь сведения о составе управляющих кодов, применяемых при обмене данными с эмулятором терминала. Этой цели служат некоторая переменная среды (см. главу 5) и специальное множество файлов, которые имеют общее название *база данных terminfo*.

База данных terminfo представляет собой ряд файлов, определяющих характеристики различных терминалов, которые могут использоваться в системе Linux. Система Linux хранит данные terminfo для каждого типа терминала в виде отдельного файла в каталоге базы данных terminfo. В различных дистрибутивах может быть определено разное местоположение для этого каталога. Чаще всего этот каталог имеет расположение `/usr/share/terminfo`, `/etc/terminfo` или `/lib/terminfo`.

Как правило, каталог базы данных terminfo имеет подкаталоги, обозначенные буквами алфавита, поскольку это способствует упрощению организации содержащейся в нем информации (дело в том, что количество различных файлов terminfo обычно весьма велико). Отдельные файлы, относящиеся к конкретному терминалу, сохраняются в каталоге, обозначенном буквой, которая соответствует имени этого терминала. Например, в каталоге `/usr/share/terminfo/v` хранятся файлы эмуляторов терминала VT.

Каждый отдельный файл terminfo представляет собой двоичный файл, полученный в результате компиляции текстового файла. Сами текстовые файлы содержат кодовые слова, которые определяют функции экрана, связанные с управляющими кодами, необходимыми для реализации этих функций на терминале.

Поскольку файлы базы данных terminfo являются двоичными, сами коды в этих файлах нельзя просматривать непосредственно. Но можно воспользоваться командой `infocmp` для преобразования двоичных записей в текст. Ниже приведен пример использования данной команды.

```

$ infocmp vt100
# Reconstructed via infocmp from file: /lib/terminfo/v/vt100
vt100|vt100-am|dec vt100 (w/advanced video),
    am, msgr, xenl, xon,
    cols#80, it#8, lines#24, vt#3,
    acsc="aaffggjjkkllmmnnoppqrrssttuuvvwxxyyzz{|||}",
    bel=G, blink=\E[5m$<2>, bold=\E[1m$<2>,
    clear=\E[H\E[J$<50>, cr=M, csr=\E[%i%p1%d;%p2%dr,
    cub=\E[%p1%dD, cub1=H, cud=\E[%p1%dB, cud1=J,
    cuf=\E[%p1%dC, cuf1=\E[C$<2>,
    cup=\E[%i%p1%d;%p2%dH$<5>, cuu=\E[%p1%dA,
    cuul=\E[A$<2>, ed=\E[J$<50>, el=\E[K$<3>,
    ell=\E[1K$<3>,
    enacs=\E(B\E)0, home=\E[H, ht=I, hts=\EH, ind=J, ka1=\EOq,
    ka3=\EOs, kb2=\EOr, kbs=H, kc1=\EOp, kc3=\EOn, kcub1=\EOD,
    kcud1=\EOB, kcufl=\EOC, kcuul=\EOA, kent=\EOM, kf0=\EOy,
    kf1=\EOP, kf10=\EOx, kf2=\EOQ, kf3=\EOR, kf4=\EOS, kf5=\EOt,
    kf6=\EOu, kf7=\EOv, kf8=\EOl, kf9=\EOw, rc=\E8,
    rev=\E[7m$<2>, ri=\EM$<5>, rmacs=O, rmam=\E[?7l,
    rmkx=\E[?1l\E>, rmso=\E[m$<2>, rmul=\E[m$<2>,
    rs2=\E>\E[?3l\E[?4l\E[?5l\E[?7h\E[?8h,
    sc=\E7,
    sgr0=\E[m017$<2>, smacs=N, smam=\E[?7h, smkx=\E[?1h\E=,
    smso=\E[7m$<2>, smul=\E[4m$<2>, tbc=\E[3g,
$

```

Эта запись `terminfo` определяет имя терминала (в данном случае `vt100`) наряду со всеми прочими псевдонимами, которые могут быть связаны с этим именем терминала. Отметим, что в первой строке вывода показано местонахождение файла `terminfo`, из которого извлечены рассматриваемые значения.

Вслед за этим команда `infocmp` выводит конкретные сведения из определения терминала, а также управляющие коды, которые используются для эмуляции отдельных функций терминала. Некоторые функции могут быть включены или выключены. Если функция приведена в списке, то она включена согласно определению терминала (в качестве примера можно указать функцию `am`, которая определяет автоматическое выравнивание по правому полю). Другие функции должны определять конкретные последовательности управляющих кодов, применяемых для выполнения соответствующей задачи (таких как очистка, или стирание, отображения на мониторе). В табл. 2.1 приведен список некоторых функций, которые можно видеть в приведенном листинге файла определения `terminfo` терминала `vt100`.

Таблица 2.1. Коды функций `terminfo`

Код	Описание
<code>am</code>	Задание автоматического выравнивания по правому полю
<code>msgr</code>	Предохранение от перемещения курсора в режиме вставки
<code>xenl</code>	Пропуск символов обозначения конца строки после выхода за 80 столбцов
<code>xon</code>	Применение на терминале символов <code>XON/XOFF</code> для управления потоком данных
<code>cols#80</code>	Организация вывода по 80 столбцов в строке
<code>it#8</code>	Определение соответствия знака табуляции восьми пробелам

Код	Описание
lines#24	Организация вывода по 24 строки на экране
vt#3	Виртуальный терминал номер 3
bel	Управляющий код, используемый для эмуляции звонка
blink	Управляющий код, используемый для формирования мигающего текста
bold	Управляющий код, используемый для формирования полужирного текста
clear	Управляющий код, используемый для очистки экрана
cr	Управляющий код, используемый для ввода символа возврата каретки
csr	Управляющий код, используемый для смены области прокрутки
cub	Перемещение на один символ влево без стирания
cub1	Перемещение курсора назад на один пробел
cud	Перемещение курсора вниз на одну строку
cud1	Управляющий код для перемещения курсора вниз на одну строку
cuf	Перемещение на один символ вправо без стирания
cuf1	Управляющий код для перемещения курсор вправо на один пробел без стирания
cup	Управляющий код для перемещения в строку один, столбец два на дисплее
cuu	Перемещение курсора вверх на одну строку
cuu1	Управляющий код для перемещения курсора вверх на одну строку
ed	Очистка до конца экрана
el	Очистка до конца строки
el1	Очистка до начала строки.
enacs	Включение дополнительной кодировки
home	Управляющий код для перемещения курсора в исходную позицию — в строку один, столбец два (то же, что и cup)
ht	Символ табуляции
hts	Задать табуляцию в каждой строке в текущем столбце
ind	Выполнить прокрутку текста вверх
ka1	Левая верхняя клавиша на клавиатуре
ka3	Правая верхняя клавиша на клавиатуре
kb2	Центральная клавиша на клавиатуре
kbs	Клавиша <Backspace>
kc1	Левая нижняя клавиша на клавиатуре
kc3	Правая нижняя клавиша на клавиатуре
kcub1	Клавиша со стрелкой влево
kcud1	Управляющий код для клавиши со стрелкой вниз
kcuf1	Клавиша со стрелкой вправо
kcuu1	Клавиша со стрелкой вверх
kent	Клавиша <Enter> (ВВОД)
kf0	Функциональная клавиша <F0>

Код	Описание
kf1	Функциональная клавиша <F1>
kf10	Функциональная клавиша <F10>
rc	Восстановление позиции курсора в последней сохраненной позиции
rev	Режим негативного изображения
ri	Выполнение прокрутки текста вниз
rmacs	Конец дополнительной кодировки
rmam	Выключение автоматически заданных полей
rmkx	Выход из клавиатурного режима передачи
rmso	Выход из режима вставки
rmul	Выход из режима подчеркивания
rs2	Сброс
sc	Сохранение текущей позиции курсора
sgr	Определение атрибутов видеорежима
sgr0	Выключение всех атрибутов
smacs	Начало дополнительной кодировки
smam	Включение автоматически заданных полей
smkx	Запуск клавиатурного режима передачи
smso	Начало режима вставки
smul	Начало режима подчеркивания
tbc	Очистка всех табуляторов

В командном интерпретаторе Linux для определения того, какой параметр эмуляции терминала из базы данных `terminfo` должен использоваться для конкретного сеанса, служит переменная среды `TERM`. Если переменная среды `TERM` задана равной `vt100`, то командный интерпретатор определяет, что должны использоваться управляющие коды, связанные с записью базы данных `terminfo`, соответствующей `vt100`, для отправки управляющих кодов в эмулятор терминала. Чтобы определить значение переменной среды `TERM`, достаточно вызвать ее с помощью эхо-повтора из интерфейса командной строки:

```
$ echo $TERM
xterm
$
```

Этот пример показывает, что текущий тип терминала задан в соответствии с записью `xterm` в базе данных `terminfo`.

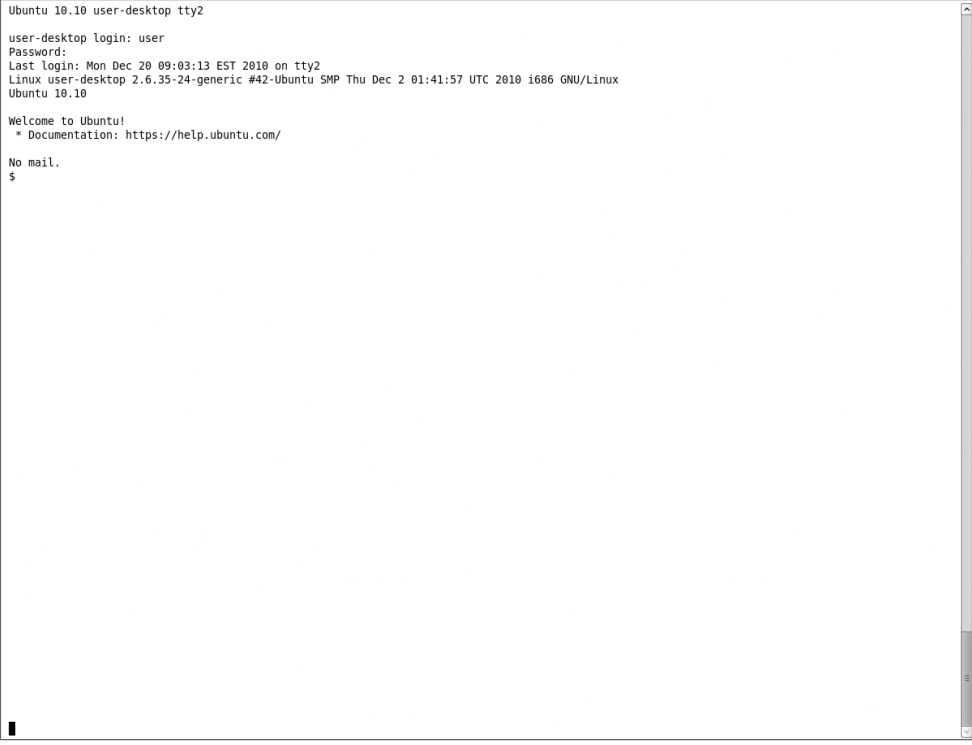
Консоль Linux

В первых версиях Linux после загрузки системы на мониторе появлялось приглашение к вводу информации регистрации, и на этом заканчивались возможности пользовательского интерфейса. Как уже было сказано выше, этот пользовательский интерфейс именуется консолью Linux. Консоль Linux была единственным местом, где можно было вводить команды для работы с системой.

В современных системах Linux после запуска автоматически создается несколько *виртуальных консолей*. Виртуальная консоль — это сеанс терминала, который выполняется в памяти в системе Linux. Вместо подключения нескольких терминалов ввода-вывода к персональному компьютеру в большинстве дистрибутивов Linux предусмотрен запуск семи (а иногда даже большего количества) виртуальных консолей, доступ к которым может быть получен с помощью одних и тех же устройств персонального компьютера — клавиатуры и монитора.

В большинстве дистрибутивов Linux можно получить доступ к виртуальной консоли с помощью простой комбинации клавиш. Обычно достаточно удерживать нажатыми клавиши <Ctl> и <Alt>, затем нажать одну из функциональных клавиш (<F1>—<F8>), чтобы перейти к виртуальной консоли, которую намечено использовать. Функциональная клавиша <F1> обеспечивает переход к виртуальной консоли 1, клавиша <F2> — к виртуальной консоли 2 и т.д.

В шести из всех виртуальных консолей используется полноэкранный текстовый эмулятор терминала для отображения текстового экрана регистрации, как показано на рис. 2.2.



```
Ubuntu 10.10 user-desktop tty2
user-desktop login: user
Password:
Last login: Mon Dec 20 09:03:13 EST 2010 on tty2
Linux user-desktop 2.6.35-24-generic #42-Ubuntu SMP Thu Dec 2 01:41:57 UTC 2010 i686 GNU/Linux
Ubuntu 10.10

Welcome to Ubuntu!
 * Documentation: https://help.ubuntu.com/

No mail.
$
```

Рис. 2.2. Экран регистрации консоли Linux

После регистрации пользователя со своим идентификатором и паролем происходит переход к интерфейсу командной строки командного интерпретатора `bash` системы Linux. Консоль Linux не предоставляет возможности выполнять какие-либо графические программы. Могут применяться лишь текстовые программы, вывод которых отображается на текстовых консолях Linux.

После регистрации на одной из виртуальных консолей можно оставить ее в активном состоянии и переключиться на другую виртуальную консоль, не прекращая начатого ранее

активного сеанса. Пользователь может переключаться между всеми виртуальными консолями, одновременно поддерживая несколько активных сеансов.

Первая или последние две виртуальные консоли обычно зарезервированы для графических рабочих столов X Window. В некоторых дистрибутивах для такого использования предназначена только одна виртуальная консоль, поэтому, скорее всего, придется вначале проверить все три комбинации клавиш, <Ctl+Alt+F1>, <Ctl+Alt+F7> и <Ctl+Alt+F8>, чтобы узнать, какие из них используются в данном конкретном дистрибутиве. Большинство дистрибутивов обеспечивает автоматическое переключение на одну из графических виртуальных консолей после завершения последовательности загрузки, чтобы предоставить пользователю возможность полноценной работы с графическим окном регистрации и графическим рабочим столом.

Организация работы по принципу регистрации в сеансе текстового виртуального терминала с последующим переключением на графический терминал обычно менее удобна. К счастью, предусмотрен лучший способ переключения между графическим и текстовым режимами в системе Linux. Он состоит в использовании пакетов эмуляции терминала и широко применяется для получения доступа к интерфейсу командной строки командного интерпретатора в сеансе работы с графическим рабочим столом. В следующих разделах описаны наиболее широко применяемые пакеты программ, которые обеспечивают эмуляцию терминала в графическом окне.

Терминал xterm

Раньше всех прочих пакетов был разработан пакет *xterm* эмуляции терминала X Window, который является также наиболее простым. Пакет *xterm* появился непосредственно со времени создания среды X Window и включен по умолчанию в большинство пакетов X Window.

Пакет *xterm* предоставляет и основной интерфейс эмуляции терминала VT102/220 с командной строкой, и графическую среду Tektronix 4014 (аналогичную среде 4010). Несмотря на то что *xterm* представляет собой полный пакет эмуляции терминала, он не требует для работы большого объема ресурсов (в частности, памяти). По этой причине пакет *xterm* все еще широко применяется в дистрибутивах Linux, предназначенных для эксплуатации на аппаратных платформах более старых выпусков. В некоторых вариантах среды графического рабочего стола, таких как Fluxbox, *xterm* используется как предусмотренный по умолчанию пакет эмуляции терминала.

Безусловно, пакет *xterm* не предлагает многих привлекательных функций, но весьма хорошо решает многие задачи, одной из которых является эмуляция терминала VT220. Более новые версии *xterm* обеспечивают даже эмуляцию управляющих кодов для работы с цветным изображением терминалов ряда VT, что позволяет использовать выделение цветом в выводе сценариев.

На рис. 2.3 показан внешний вид простого окна *xterm* на графическом рабочем столе Linux.

Пакет *xterm* позволяет управлять отдельными функциями с применением параметров командной строки и четырех простых графических меню. Эти функции рассматриваются в следующих разделах и там же описано, как изменять их параметры.

Параметры командной строки

Список параметров командной строки *xterm* весьма значителен. Предусмотрено много средств, которыми можно управлять для настройки функций эмуляции терминала, в частности, включать или выключать отдельные составляющие эмуляции VT.

В параметрах командной строки *xterm* используются знаки “плюс” (+) и “минус” (-) для указания на то, как установлено данное средство. Знак “плюс” указывает, что настройка рассматриваемого средства должна быть возвращена в значение по умолчанию. А знак “минус”

говорит о том, что для средства должно быть задано значение, отличное от предусмотренного по умолчанию. В табл. 2.2 перечислены некоторые из наиболее широко применяемых средств, которые можно задавать с помощью параметров командной строки.

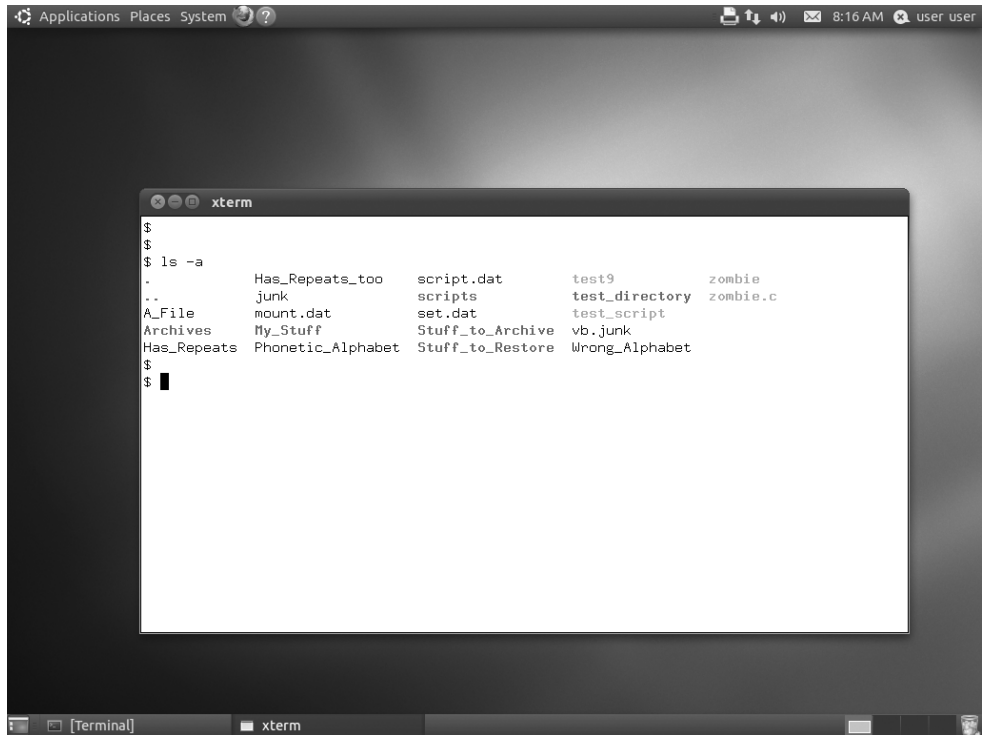


Рис. 2.3. Основное окно *xterm*

Таблица 2.2. Параметры командной строки <i>xterm</i>	
Параметр	Описание
132	По умолчанию <i>xterm</i> не допускает применения режима с 132 символами в строке
ah	Текстовый курсор всегда выделяется подсветкой
aw	Автоматический перенос строк включен
bc	Мигание текстового курсора включено
bg color	Определение цвета, используемого для фона
cm	Распознавание управляющих кодов изменения цвета ANSI отключено
fb font	Определение шрифта, используемого для полужирного текста
fg color	Определение цвета, используемого для текста переднего плана
fn font	Определение шрифта, используемого для текста
fw font	Определение шрифта, используемого для текста с широкими символами
hc color	Определение цвета, используемого для текста, выделенного подсветкой
j	Использование скачкообразной прокрутки, при которой происходит прокрутка одновременно на несколько строк

Параметр	Описание
<code>l</code>	Включение записи данных с экрана в файл журнала
<code>lf filename</code>	Определение имени файла, используемого для ведения журнала содержимого экрана
<code>mb</code>	Воспроизведение звука колокольчика на поле, когда курсор достигает конца строки
<code>ms color</code>	Определение цвета, используемого для текстового курсора
<code>name name</code>	Определение имени приложения, которое появляется в области заголовка
<code>rv</code>	Включение негативного изображения, при котором цвета фона и переднего плана сменяют друг друга
<code>sb</code>	Использование боковой линейки прокрутки для выполнения прокрутки по сохраненным данным прокрутки
<code>t</code>	Запуск программы <code>xterm</code> в режиме Tektronix
<code>tb</code>	Определение панели инструментов, которая должна отображаться в верхней части окна программы <code>xterm</code>

Следует учитывать, что не во всех реализациях `xterm` поддерживаются все эти параметры командной строки. Предусмотрена возможность определить, какие параметры реализованы в применяемой версии `xterm`, воспользовавшись параметром `-help` при запуске `xterm` в конкретной системе.

Главное меню параметров `xterm` (Main Options)

Главное меню `xterm` содержит элементы настройки, которые относятся одновременно к окнам VT102 и Tektronix. Можно получить доступ к главному меню, удерживая нажатой клавишу `<Ctrl>` и однократно щелкнув кнопкой мыши (левой кнопкой на мыши для правой руки или правой кнопкой на мыши — для левой) после перевода курсора в окно сеанса `xterm`. Внешний вид главного меню `xterm` в одном из дистрибутивов показан на рис. 2.4.

В главном меню `xterm` имеются четыре раздела, которые описаны ниже.

Команды событий X

В разделе команд событий X доступны средства, которые позволяют управлять взаимодействием программы `xterm` с окном X Window.

- Панель инструментов (Toolbar). Если в применяемой версии `xterm` поддерживается панель инструментов, то в меню предусмотрен соответствующий элемент, позволяющий включать или отключать отображение панели инструментов в окне `xterm` (эта команда меню аналогична параметру командной строки `tb`).
- Безопасная клавиатура (Secure Keyboard). Эта команда позволяет ограничить применение определенных комбинаций клавиш исключительно конкретным окном `xterm`. Это удобно при вводе пароля, поскольку гарантирует, что введенные данные нельзя будет перехватить в другом окне.
- Разрешение на передачу событий (Allow SendEvents). Эта команда разрешает принимать в окне `xterm` сообщения о событиях X Window, сформированных в других приложениях X Window.
- Перерисовка окна (Redraw Window). Эта команда указывает системе X Window, что необходимо обновить содержимое окна `xterm`.

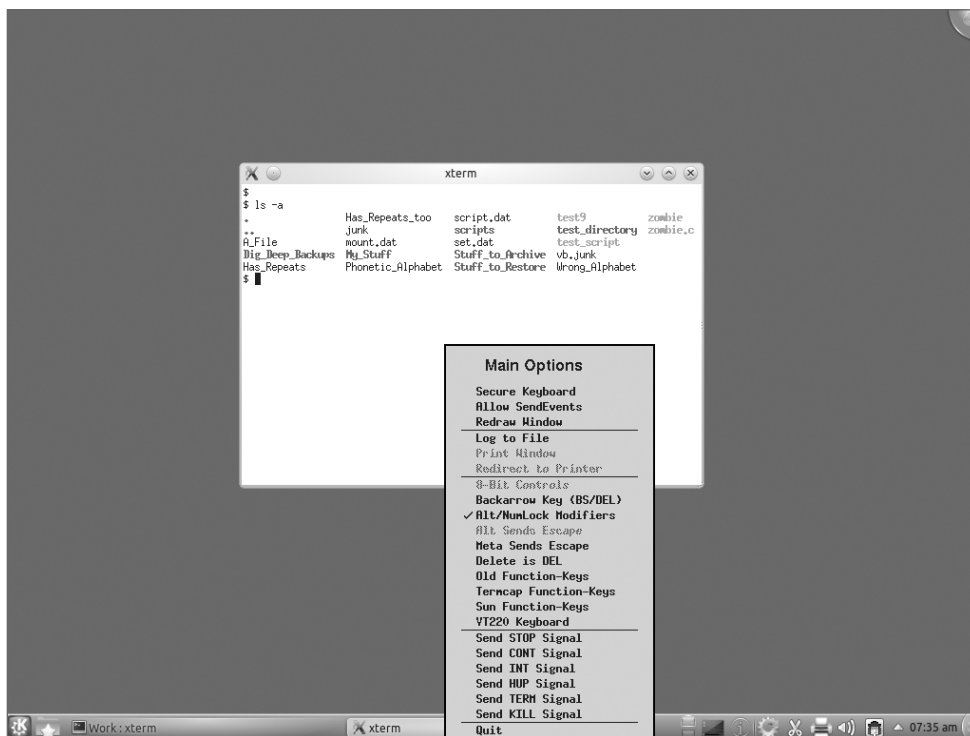


Рис. 2.4. Главное меню параметров xterm (Main Options)

Еще раз отметим, что не все эти функции могут поддерживаться в конкретной применяемой реализации xterm. Если они не поддерживаются, то не появляются в меню или выделяются серым цветом.

Перехват вывода

Пакет xterm позволяет перехватывать данные, отображаемые в окне, а затем либо регистрировать эти данные в файле, либо отправлять их на текущий принтер, определенный в системе X Window. Ниже перечислены элементы меню, относящиеся к этому разделу.

- **Регистрация в файле (Log to file).** Эта команда обеспечивает передачу всех данных, отображаемых в окне xterm, в текстовый файл.
- **Печать содержимого окна (Print window).** Эта команда обеспечивает отправку всех данных, отображаемых в текущем окне, на применяемый по умолчанию принтер X Window.
- **Перенаправление на принтер (Redirect to printer).** Эта команда также обеспечивает отправку всех данных, отображаемых в окне xterm, на применяемый по умолчанию принтер X Window. Данная функция должна быть выключена в целях прекращения печати чрезмерного объема данных.

Дело в том, что это средство перехвата может создавать излишнюю нагрузку на систему, если в области отображения используются графические или управляющие символы (например, для вывода текста с выделением цветом). Причем в файле журнала сохраняются или передаются на принтер все символы, отправленные на дисплей, включая управляющие символы.

При использовании средства печати xterm предполагается, что в системе X Window определен текущий принтер, применяемый по умолчанию. Если же ни один принтер не определен, то соответствующий этому средству элемент меню будет выделен серым цветом.

Настройки клавиатуры

Раздел с настройками клавиатуры содержит функции, позволяющие индивидуализировать применяемый в программе xterm способ отправки символов, введенных с клавиатуры, в базовую систему.

- 8-битовые управляющие коды (8-bit controls). Эта команда обеспечивает передачу 8-битовых управляющих кодов, применяемых в терминалах VT220, вместо 7-битовых управляющих кодов ASCII.
- Клавиша с обратной стрелкой (Back arrow key). Эта команда выполняет прямое и обратное переключение функции клавиши с обратной стрелкой между отправкой символа возврата на позицию (Backspace) или символа удаления (Delete).
- Модификаторы Alt/Numlock (Alt/Numlock Modifiers). Эта команда управляет тем, должны ли нажатия клавиши <Alt> или <Numlock> приводить к изменению поведения цифровой клавиатуры персонального компьютера.
- Передача экранирующего кода клавишей <Alt> (Alt Sends Escape). Эта команда позволяет указать, что при нажатии клавиши <Alt> вместе с другой клавишей должна происходить передача экранирующего управляющего кода.
- Передача экранирующего кода метаклавишей (Meta sends Escape). Эта команда позволяет управлять тем, должна ли происходить при нажатии функциональных клавиш передача двухсимвольного управляющего кода, включая экранирующий управляющий код.
- Клавиша <Delete> служит как обычная клавиша удаления (Delete is DEL). Эта команда позволяет указать, что клавиша <Delete> клавиатуры персонального компьютера предназначена для передачи символа удаления (Delete) вместо символа возврата на позицию (Backspace).
- Старые версии функциональных клавиш (Old Function keys). Эта команда указывает, что функциональные клавиши клавиатуры персонального компьютера должны эмулировать функциональные клавиши терминала DEC VT100.
- Функциональные клавиши Termcap (Termcap Function keys). Эта команда указывает, что функциональные клавиши клавиатуры персонального компьютера должны эмулировать функциональные клавиши Termcap системы Berkley Unix.
- Функциональные клавиши Sun (Sun Function keys). Эта команда указывает, что функциональные клавиши клавиатуры персонального компьютера должны эмулировать функциональные клавиши рабочей станции Sun.
- Клавиатура VT220 (VT220 keyboard). Эта команда указывает, что функциональные клавиши клавиатуры персонального компьютера должны эмулировать функциональные клавиши терминала DEC VT220.

Вполне очевидно, что настройка параметров клавиатуры часто зависит от того, в какой среде и (или) в каком конкретном приложении применяется эмулятор терминала. Многое также зависит от личных предпочтений пользователя. Очень часто выбор определенных параметров клавиатуры продиктован тем, являются ли они наиболее удобными в работе конкретного пользователя.

Меню параметров VT (VT Options)

Меню параметров VT определяет функции, используемые программой xterm при эмуляции терминала VT102. Для доступа к меню параметров VT необходимо удерживать нажатой клавишу <Ctrl> и щелкнуть второй кнопкой. Как правило, второй кнопкой мыши является средняя кнопка мыши. Если используется мышь с двумя кнопками, то в большинстве конфигураций X Window в системе Linux щелчок средней кнопкой мыши эмулируется одновременным щелчком левой и правой кнопками. На рис. 2.5 показан внешний вид меню параметров VT.

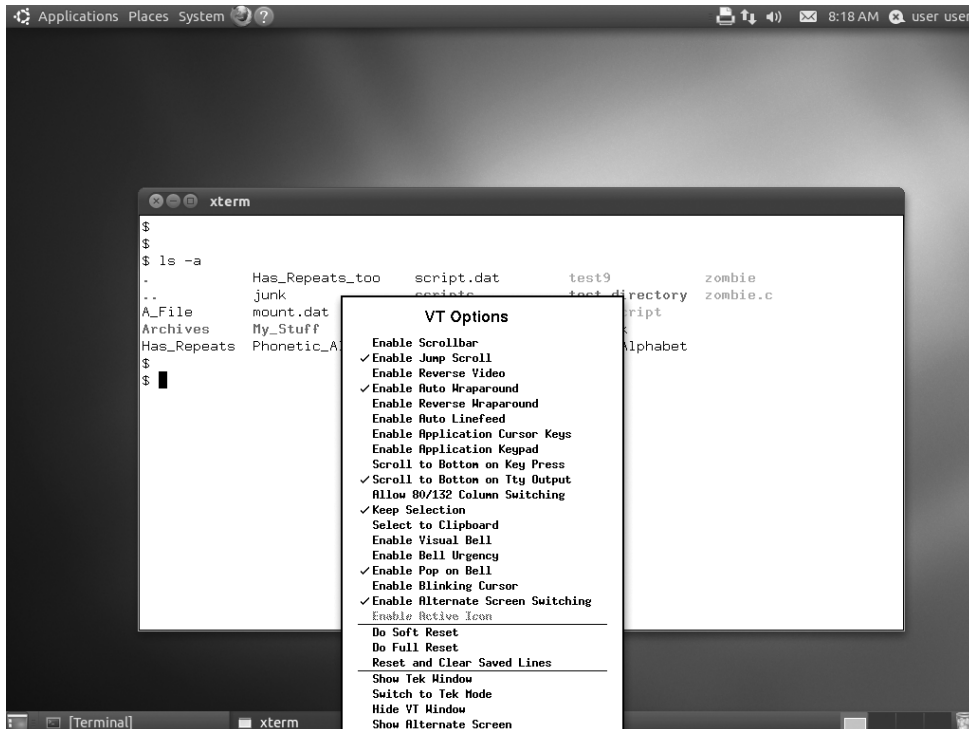


Рис. 2.5. Меню параметров VT (VT Options) программы xterm

Как показано на рис. 2.5, многие функции VT, которые можно задавать с применением параметров командной строки, могут быть также заданы с помощью меню параметров VT. В меню параметров показан весьма значительный список доступных параметров. Команды настройки параметров VT разделены на три набора команд, описанных в следующих разделах.

Средства VT

К командам управления средствами VT относятся команды, определяющие функции, с помощью которых программа xterm реализует эмуляцию VT102/220. Эти команды перечислены ниже.

- Включение линейки прокрутки (Enable Scrollbar).
- Включение линейки скачкообразной прокрутки (Enable Jump Scrollbar).

- Включение негативного изображения (Enable Reverse Video).
- Включение автоматического переноса строк (Enable Auto Wraparound).
- Включение обратного переноса строк (Enable Reverse Wraparound).
- Включение автоматического перевода строки (Enable Auto Linefeed).
- Включение клавиш курсора приложения (Enable Application Cursor Keys).
- Включение клавиатуры приложения (Enable Application Keypad).
- Прокрутка вниз при нажатии клавиши (Scroll to Bottom on Keypress).
- Прокрутка вниз при выводе на терминал (Scroll to Bottom on TTY Output).
- Ввод в действие переключения между 80 и 132 столбцами (Allow 80/132 Column Switching).
- Выборка в буфер обмена (Select to Clipboard).
- Включение визуального управляемого звонка (Enable Visual Bell).
- Включение безотлагательности звонка (Enable Bell Urgency).
- Включение создания окна при получении сигнала звонка (Enable Pop on Bell).
- Включение мигающего курсора (Enable Blinking Cursor).
- Включение переключения на дополнительный экран (Enable Alternate Screen Switching).
- Включение активного значка (Enable Active Icon).

Каждую из этих функций можно включать или отключать, щелкая на соответствующем ей элементе в меню. Включенная функция отмечена стоящей рядом с ней галочкой.

Команды VT

Раздел команд VT предназначен для передачи конкретных команд сброса в окно эмуляции xterm. К ним относятся команды, перечисленные ниже.

- Выполнить программный сброс (Do Soft Reset).
- Выполнить полный сброс (Do Full Reset).
- Выполнить сброс и очистку сохраненных строк (Reset and Clear Saved Lines).

Программный сброс предусматривает передачу управляющего кода для сброса области экрана. Эта команда может применяться, если в какой-то программе неправильно задана область прокрутки. Полный сброс приводит к очистке экрана, сбросу всех установленных позиций табуляции и переустановке всех функций терминала, установленных в ходе сеанса, в начальное состояние. Команда выполнения сброса и очистки сохраненных строк обеспечивает полный сброс, а также удаляет содержимое файла журнала области прокрутки.

Текущие команды экрана

Раздел с текущими командами экрана позволяет передавать в эмулятор xterm команды, определяющие то, какой экран является в настоящее время активным.

- Отображение окна Tek (Show Tek Window). Позволяет отобразить окно терминала Tektronix наряду с окном терминала VT100.
- Переключение на окно Tek (Switch to Tek Window). Обеспечивает сокрытие окна терминала VT100 и отображение окна терминала Tektronix.
- Сокрытие окна VT (Hide VT Window). Обеспечивает сокрытие окна терминала VT100 в то время, когда отображается окно терминала Tektronix.

- Отображение дополнительного экрана (Show Alternate Screen). Позволяет отобразить данные, которые в настоящее время хранятся в области дополнительного экрана VT100.

Эмулятор терминала xterm предоставляет возможность запуска либо в режиме терминала VT100 (по умолчанию), либо в режиме терминала Tektronix (при использовании параметра командной строки `t`). После запуска в том или ином режиме эту область меню можно использовать для переключения в ходе сеанса на другой режим.

Меню шрифтов VT (VT Fonts)

Меню шрифтов VT задает начертание шрифта, используемое в окне эмуляции VT100/220. Для того чтобы получить доступ к этому меню, следует удерживать нажатой клавишу `<Ctrl>` и однократно щелкнуть третьей кнопкой мыши (правой кнопкой на мыши для правой руки или левой кнопкой — для левой) после перевода курсора в окно сеанса xterm. Внешний вид меню шрифтов VT показан на рис. 2.6.

Меню шрифтов VT, рассматриваемое ниже, содержит три раздела с вариантами выбора.

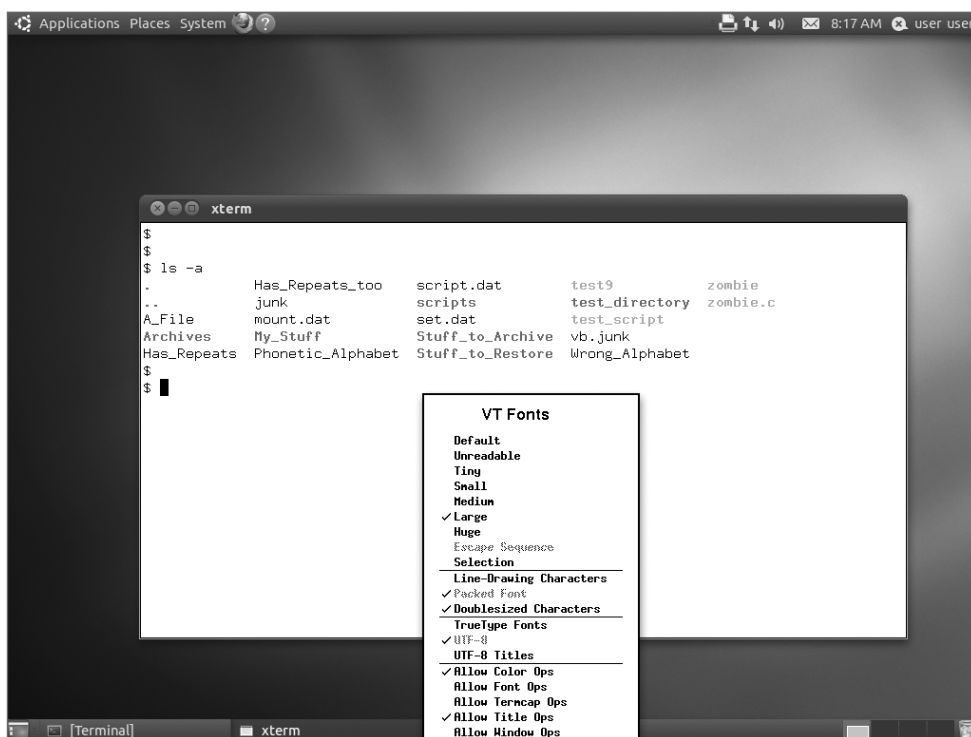


Рис. 2.6. Меню шрифтов VT программы xterm

Задание шрифта

Параметры меню задания шрифта определяют размер шрифта, используемого в окне xterm. Возможные размеры приведены ниже.

- Размер по умолчанию (Default).
- Нечитабельный (Unreadable).

- Мельчайший (Tiny).
- Мелкий (Small).
- Средний (Medium).
- Крупный (Large).
- Сверхкрупный (Huge).
- Управляющая последовательность (Escape Sequence).
- Выбор (Selection).

Шрифт, применяемый по умолчанию, представляет собой шрифт стандартного размера, который используется для отображения текста в текущей рамке X Window. Нечитабельный шрифт в основном соответствует своему названию. При его использовании размеры окна хтерт уменьшаются до такой степени, что им фактически невозможно пользоваться. Однако это удобно, если требуется до предела свернуть окно хтерт на рабочем столе, не удаляя его полностью с экрана. Крупный и сверхкрупный варианты выбора шрифта позволяют вывести на экран шрифты увеличенного размера для пользователей со слабым зрением.

Команда меню для выбора управляющей последовательности задает в качестве применяемого шрифта тот шрифт, который был установлен последним по времени с помощью управляющего кода выбора шрифта терминала VT100. Команда выбора позволяет сохранить текущий шрифт со специальным именем шрифта.

Отображение шрифта

Раздел меню с командами отображения шрифта предназначен для определения типа символов, используемых при выводе текста. В этом разделе представлены три варианта.

- Символы вычерчивания линий (Line Drawing Characters). Команда выбора символов вычерчивания линий служит для системы Linux указанием, что линии следует формировать с использованием символов линий в составе псевдографики ANSI, а не символов линий из выбранного шрифта.
- Упакованный шрифт (Packed Font). Команда выбора упакованного шрифта указывает системе Linux, что должен использоваться упакованный шрифт.
- Символы удвоенного размера (Doublesized characters) Эта команда указывает системе Linux, что шрифт должен быть масштабирован с удвоением по отношению к его обычному размеру.

Команда выбора символов рисования линий позволяет определить, какого рода графические средства должны использоваться при формировании графики в текстовом режиме. При этом могут применяться либо символы, предусмотренные в выбранном источнике шрифта, либо символы, представленные управляющими кодами DEC VT100.

Задание шрифта

В этом разделе меню содержатся следующие команды, позволяющие указать тип шрифта, используемого для вывода символов.

- Шрифты TrueType (TrueType Fonts).
- Шрифты UTF-8 (UTF-8 Fonts).
- Шрифты заголовков UTF-8 (UTF-8 Titles).

Шрифты TrueType широко применяются в различных вариантах графической среды. В отличие от моноширинных шрифтов, в которых для каждого символа отводится одинаковое пространство на строке, шрифты TrueType обеспечивают размещение символов с учетом их собственных размеров. Это означает, что буква *i* занимает на строке меньше места, чем буква *m*. Шрифт UTF-8 позволяет на время перейти к использованию кодировки Юникода при работе с приложениями, которые не поддерживают буквы, отличные от латинских. Команда выбора шрифта заголовков позволяет выводить в окне `xterm` заголовки с применением кодировки UTF-8.

Терминал Konsole

В рамках проекта создания рабочего стола KDE был разработан собственный пакет эмуляции терминала, называемый *Konsole*. Пакет *Konsole* включает основные средства программы `xterm`, а также предусматривает применение большого количества дополнительных функций, благодаря которым он становится сравнимым по своим возможностям с приложениями Windows. В настоящем разделе описаны функции терминала *Konsole* и показано, как их использовать.

Параметры командной строки

Во многих дистрибутивах Linux предусмотрен метод запуска приложений непосредственно из системы меню графического рабочего стола. Если в применяемом дистрибутиве эта возможность не поддерживается применительно к терминалу *Konsole*, то его можно запустить вручную с помощью следующего формата:

```
konsole parameters
```

Полностью аналогично `xterm`, в пакете *Konsole* используются параметры командной строки *parameters* в целях задания набора функций, применяемого в создаваемых сеансах. В табл. 2.3 приведены применимые параметры командной строки *Konsole*.

Таблица 2.3. Параметры командной строки Konsole

Параметр	Описание
<code>-e command</code>	Выполнение команды вместо командного интерпретатора
<code>--keytab file</code>	Использование указанного файла ключей для определения сопоставлений клавиш
<code>--keytabs</code>	Формирование списка всех имеющихся файлов <code>keytab</code>
<code>--ls</code>	Запуск сеанса <i>Konsole</i> с экраном регистрации
<code>--name name</code>	Задание имени, которое появляется в области заголовка <i>Konsole</i>
<code>--noclose</code>	Предотвращение закрытия окна <i>Konsole</i> после закрытия последнего сеанса
<code>--noframe</code>	Запуск окна <i>Konsole</i> без рамки
<code>--nohist</code>	Предотвращение сохранения программой <i>Konsole</i> журнала прокрутки в сеансах
<code>--nomenubar</code>	Запуск окна <i>Konsole</i> без стандартных параметров строки меню
<code>--noresize</code>	Предотвращение изменения размеров области окна <i>Konsole</i>
<code>--notabbar</code>	Запуск окна <i>Konsole</i> без стандартной области вкладок для сеансов
<code>--noxft</code>	Запуск окна <i>Konsole</i> без поддержки замены более мелких шрифтов псевдонимами этих шрифтов
<code>--profile file</code>	Запуск окна <i>Konsole</i> с настройками, сохраненными в указанном файле

Параметр	Описание
--profiles	Формирование списка всех имеющихся профилей Konsole
--schema name	Запуск окна Konsole с использованием указанного имени схемы или файла
--schemata	Формирование списка схем, доступных в Konsole
-T title	Задание заголовка окна Konsole
--type type	Запуск сеанса Konsole с использованием указанного типа
--types	Формирование списка всех доступных типов сеансов Konsole
--vt_sz CxL	Определение количества столбцов (C) и строк (L) на терминале
dir --workdir	Определение рабочего каталога для Konsole, в котором должны храниться временные файлы

Сеансы окон с вкладками

После запуска программы Konsole обнаруживается такая ее особенность, как окно с вкладками, в котором каждая вкладка соответствует открытому сеансу эмуляции терминала. Таковой является предусмотренная по умолчанию организация применения сеансов с помощью окон с вкладками, и обычно в этом состоит использование стандартного интерфейса командной строки командного интерпретатора `bash`. Программа Konsole дает пользователю возможность иметь в своем распоряжении одновременно несколько активных вкладок. Вкладки могут размещаться в верхней или в нижней области окна и позволяют легко переключаться между сеансами. Это — превосходная возможность для программистов, которым требуется, допустим, редактировать код, который представлен на одной вкладке, чередуя это с проверкой кода с помощью другой вкладки окна с вкладками. Дело в том, что можно очень легко переключаться в любой последовательности между различными активными вкладками в Konsole. На рис. 2.7 показано окно Konsole с тремя активными вкладками.

Как и эмулятор терминала `xterm`, терминал Konsole предоставляет простое меню, доступ к которому можно получить с помощью щелчка правой кнопкой мыши в активной области вкладок. После щелчка правой кнопкой мыши в области вкладок открывается меню со следующими параметрами.

- Копирование (Copy). Эта команда обеспечивает копирование выбранного текста в буфер обмена.
- Вставка (Paste). С помощью этой команды выполняется вставка содержимого буфера обмена в выбранную область.
- Очистка области прокрутки и сброс (Clear Scrollback & Reset). С помощью этой команды можно удалить весь текст из текущей вкладки и выполнить сброс терминала.
- Открытие диспетчера файлов (Open File Manager). Эта команда позволяет открыть применяемый по умолчанию диспетчер файлов рабочего стола KDE, Dolphin, в текущем рабочем каталоге.
- Смена профиля (Change Profile). Эта команда позволяет сменить профиль для текущей вкладки.
- Редактирование текущего профиля (Edit Current Profile). Эта команда позволяет отредактировать профиль текущей вкладки.
- Отображение строки меню (Show Menu Bar). Эта команда дает возможность включить или отключить отображение строки меню.

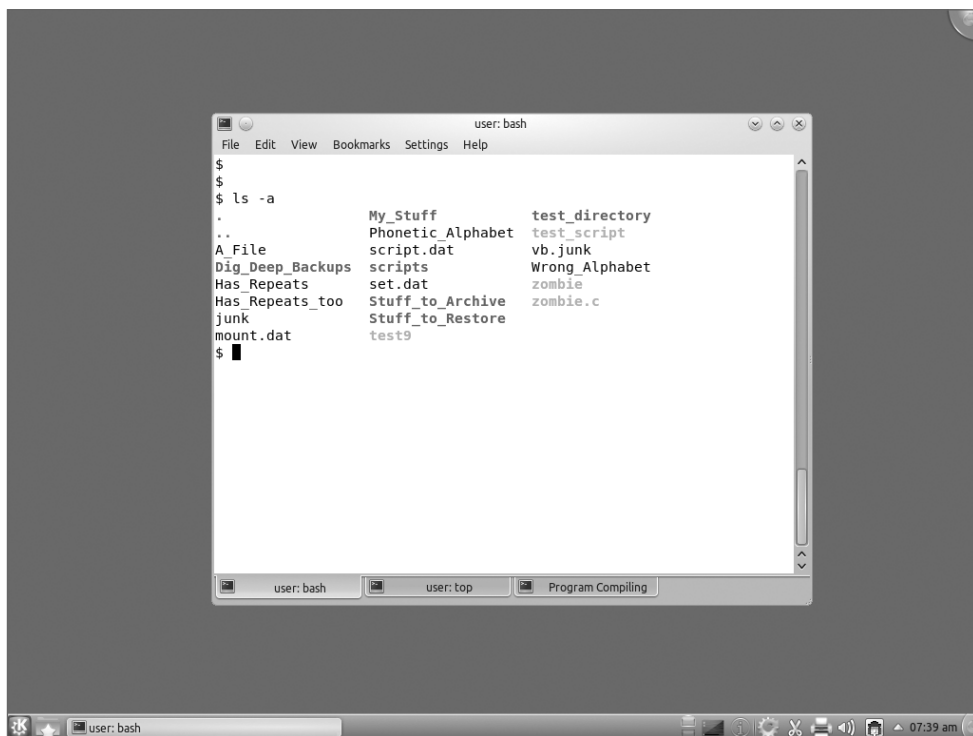


Рис. 2.7. Эмулятор терминала Konsole с тремя активными сеансами

- Кодировка символов (Character Encoding). Эта команда позволяет выбрать кодировку, используемую для передачи и отображения символов.
- Закрытие вкладки (Close Tab). С помощью этой команды можно завершить текущий сеанс окна с вкладками. Если закрываемая вкладка является последней вкладкой в окне Konsole, то закрывается сама программа Konsole.

В программе Konsole предусмотрен еще один удобный способ получения доступа к новому меню вкладок: достаточно удерживать нажатой клавишу <Ctrl> и щелкнуть правой кнопкой мыши в области вкладок.

После внесения изменений в окне с вкладками можно сохранить эти изменения для использования в дальнейшем с помощью профиля.

Профили

В программе Konsole предусмотрен мощный метод, который обеспечивает сохранение и повторное использование параметров сеанса окна с вкладками, основанный на применении профилей. Если запуск программы Konsole производится впервые, то параметры сеанса окна с вкладками извлекаются из применяемого по умолчанию профиля Shell. Эти параметры обеспечивают настройку таких опций, как используемый командный интерпретатор, цветовые схемы и т.д. После внесения изменений в текущий сеанс окна с вкладками можно сохранить эти изменения как новый профиль. Такая возможность позволяет применять многие настройки

для вкладок, например, организовать сеанс окна с вкладками, в котором используется командный интерпретатор, отличный от `bash`.

Профили могут также применяться для автоматизации трудоемких задач, таких как регистрация в другой системе. Предусмотрена возможность определять произвольное количество профилей и использовать необходимые профили при открытии тех или иных сеансов окон с вкладками. Для создания нового профиля можно воспользоваться командой редактирования текущего профиля, которая упоминалась выше при описании простого меню Konsole. Чтобы переключиться с текущего профиля на другой профиль, можно воспользоваться командой смены профиля простого меню. Описанные выше возможности поддерживаются также командами строки меню.

По умолчанию в программе Konsole строка меню используется для предоставления дополнительных функциональных средств, с помощью которых можно модифицировать и сохранять вкладки и профили Konsole.

Строка меню

В предусмотренной по умолчанию установке Konsole используется строка меню, позволяющая легко просматривать и изменять параметры и применяемые средства на вкладках. Строка меню состоит из шести элементов, как описано в следующих разделах.

Файл (File)

Элемент строки меню **Файл** позволяет открыть новую вкладку в текущем или новом окне. Он содержит следующие элементы.

- Новая вкладка (New Tab). Открывает новую вкладку Konsole в текущем окне терминала с использованием заданного по умолчанию профиля Shell (Командный интерпретатор).
- Новое окно (New Window). Запускает новое окно терминала для размещения новой вкладки Konsole.
- Список определенных профилей (List of defined profiles). Переключает на новый профиль в текущем сеансе вкладки.
- Открытие диспетчера файлов (Open File Manager). Открывает диспетчер файлов в текущем рабочем каталоге.
- Закрытие вкладки (Close Tab). Закрывает текущую вкладку.
- Завершение (Quit). Завершает работу приложения Konsole.

При первоначальном запуске Konsole единственным профилем, перечисленным в **Списке определенных профилей** (List-of-Defined-Profiles), является Shell (Командный интерпретатор). По мере дальнейшего создания и сохранения дополнительных профилей их имена также появляются в списке.

Редактирование (Edit)

Элемент строки меню **Edit** обеспечивает возможность применять операции обработки текста в сеансе, а также предусматривает выполнение некоторых дополнительных функций, как указано ниже.

- Копирование (Copy). Копирование выбранного текста (который выделен подсветкой с помощью мыши) в буфер обмена системы.
- Вставка (Paste). Вставка текста, находящегося в настоящее время в буфере обмена системы, в текущем местоположении курсора. Если текст содержит символы обозначения конца строки, эти символы обрабатываются командным интерпретатором.

- Переименование вкладки (Rename Tab). Изменение текущего имени вкладки. В дополнение к тексту могут использоваться следующие обозначения.
 - %#. Номер сеанса.
 - %D. Текущий каталог (абсолютное имя).
 - %d. Текущий каталог (относительное имя).
 - %p. Имя программы.
 - %u. Имя пользователя.
 - %w. Заголовок окна, заданный командным интерпретатором.
- Перенаправление ввода (Copy Input To). Эта команда указывает, что текст, введенный в текущей вкладке, должен быть перенаправлен в одну или несколько вкладок в открытых в настоящее время окнах терминала.
- Передача по ZModem (ZModem Upload). Эта команда позволяет передать файл в систему с использованием протокола ZModem.
- Очистка и сброс (Clear and Reset). Отправляет управляющий код для сброса эмулятора терминала и очищает текущее окно сеанса.

Программа Konsole предоставляет превосходный метод отслеживания назначения вкладок. С помощью команды **Переименовать вкладку** (Rename Tab) можно присвоить вкладке имя, соответствующее ее профилю. Это помогает следить за тем, какие функции выполняют те или иные вкладки.

Представление (View)

Элемент строки меню **Представление** содержит элементы, позволяющие управлять отдельными сеансами в окне Konsole. В число предусмотренных при этом команд входят следующие.

- Раздельное представление (Split View). Управляет отображением в текущем окне эмуляции терминала. Возможные способы представления можно задать с помощью следующих команд.
 - Разделение на левую и правую панели (Split View Left/Right). Разбивает текущее представление на два идентичных экрана, расположенных рядом.
 - Разделение на верхнюю и нижнюю панели (Split View Top/Bottom). Разбивает текущее представление на два идентичных экрана, расположенных один над другим.
 - Закрытие активного (Close Active). Объединяет разбитое в настоящее время текущее окно терминала снова в одно окно.
 - Закрытие других (Close Others). Объединяет разбитые в настоящее время окна терминала, отличные от текущего, снова в одно окно.
 - Развертывание представления (Expand View). Корректирует размеры активной панели разбитого на несколько панелей окна терминала, предоставив ей большую часть экрана.
 - Свертывание представления (Shrink View). Корректирует размеры активной панели разбитого на несколько панелей окна терминала, предоставив ей меньшую часть экрана.
- Отсоединение представления (Detach View). С помощью этой команды можно удалить текущую вкладку из окна Konsole и запустить новое окно Konsole с использованием

данной текущей вкладки. Такая возможность предоставляется, если открыта больше чем одна активная вкладка.

- Отображение строки меню (Show Menu Bar). Включает или отключает отображение строки меню.
- Полноэкранный режим (Full Screen Mode). Включает или отключает заполнение окном терминала всей области отображения монитора.
- Отслеживание паузы (Monitor for Silence). Включает или отключает специальный внешний вид значка, при котором на вкладке не появляется какой-либо новый текст в течение 10 секунд. Это позволяет переключиться на другую вкладку во время ожидания прекращения вывода из приложения, например, при компиляции большой программы.
- Отслеживание активности (Monitor for Activity). Включает или отключает специальный внешний вид значка, при котором на вкладке выводится текст. Это позволяет переключиться на другую вкладку во время ожидания вывода из приложения.
- Кодировка символов (Character Encoding). Выбирает кодировку, используемую для передачи и отображения символов.
- Увеличение размера текста (Increase Text Size). Увеличивает размер шрифта текста.
- Уменьшение размера текста (Decrease Text Size). Уменьшает размер шрифта текста.

Функция **Раздельное представление (Split View)** в программе Konsole отслеживает текущее количество открытых вкладок в раздельном представлении. Например, если в окне терминала имеются три вкладки и происходит разбиение представления, каждое представление будет иметь три вкладки.

Обратная прокрутка (Scrollback)

Программа Konsole сохраняет для каждой вкладки область журнала, которую принято называть *буфером обратной прокрутки*. Область журнала содержит текстовый вывод, состоящий из строк, которые вышли в результате прокрутки за пределы области просмотра эмулятора терминала. По умолчанию в буфере обратной прокрутки сохраняется последняя 1000 строк вывода. В меню **Обратная прокрутка (Scrollback)** предусмотрены команды для работы с этим буфером.

- Поиск в выводе (Search Output). Открывает диалоговое окно в нижней части текущей вкладки. Диалоговое окно **Найти (Find)** позволяет указать программе Konsole, какой конкретно текст должен быть найден в буфере прокрутки. Это окно позволяет указать, должен ли учитываться регистр, задать регулярные выражения и определить направление поиска.
- Найти следующее (Find Next). Находит следующее вхождение текста, согласующегося со строкой поиска, в более близкой по времени части журнала буфера обратной прокрутки.
- Найти предыдущее (Find Previous). Находит следующее вхождение текста, согласующегося со строкой поиска, в более дальней по времени части журнала буфера обратной прокрутки.
- Сохранить вывод (Save Output). Сохраняет содержимое буфера обратной прокрутки в текстовом файле или в HTML-файле.
- Параметры обратной прокрутки (Scrollback Options). Управляет функционированием буфера обратной прокрутки. Ниже перечислены возможные варианты работы буфера.

- Отсутствие обратной прокрутки (No Scrollback). Отключает буфер обратной прокрутки.
- Фиксация обратной прокрутки (Fixed Scrollback). Задаёт размер буфера обратной прокрутки (количество строк). По умолчанию применяется размер, равный 1000 строк.
- Отсутствие ограничения обратной прокрутки (Unlimited Scrollback). Сохраняет в буфере обратной прокрутки неограниченное количество строк.
- Сохранение в текущий профиль (Save to Current Profile). Сохраняет параметры буфера обратной прокрутки в составе параметров текущего профиля.
- Очистка области прокрутки и сброс (Clear Scrollback & Reset). Удаляет содержимое буфера обратной прокрутки и выполняет сброс в окне терминала.

Предусмотрена возможность просматривать буфер обратной прокрутки следующим образом: используя линейку прокрутки в области просмотра; нажимая клавишу <Shift> и клавишу со стрелкой вверх для выполнения построчной прокрутки; нажимая клавишу перелистывания страницы вверх для постраничной прокрутки (на 24 строки).

Закладки (Bookmarks)

Элементы меню **Закладки** предоставляют возможность управлять закладками, установленными в окне Konsole. Закладка даёт возможность сохранять данные о местоположении в каталоге активного сеанса, а затем возвращаться в прежнее место в том же или новом сеансе. Во время работы часто приходится раскрывать один за другим большое количество каталогов, чтобы найти что-либо в системе Linux, а затем выходить на прежний уровень в дереве каталогов, после чего бывает трудно вспомнить, как снова пройти этот проделанный путь. Закладки позволяют решить эту проблему. Таким образом, после того, как найдено нужное место в дереве каталогов, достаточно установить новую закладку. Если потребуется снова вернуться в это место, можно найти соответствующую ему закладку в меню **Закладки**, после чего автоматически выполняется смена каталога от имени пользователя для перехода в нужное ему место. Элементы меню для работы с закладками включают следующие команды.

- Добавление закладки (Add Bookmark). Создает новую закладку, соответствующую текущему местоположению в дереве каталогов.
- Установка закладок на вкладки как папки (Bookmark Tabs as Folder). Создает по одной закладке для всех текущих вкладок окна терминала.
- Создание папки закладок (New Bookmark Folder). Создает новую папку для хранения закладок.
- Редактирование закладок (Edit Bookmarks). Позволяет редактировать существующие закладки.
- Создание списка закладок (A list of your bookmarks). Включает в один список все созданные закладки.

Не существует ограничения на число закладок, которое может быть сохранено в программе Konsole, но если количество закладок слишком велико, это может привести к путанице. По умолчанию все закладки в области закладок находятся на одном и том же уровне. Предусмотрена возможность более удобно организовать распределение закладок, создавая новые папки закладок и перемещая отдельные закладки в эти папки с помощью команды меню **Редактирование закладок**.

Настройки (Settings)

Область строки меню **Настройки** позволяет настраивать профили и управлять ими, а также предусматривать немного более широкие функциональные возможности для текущего сеанса работы с вкладками. Эта область включает следующие команды.

- Смена профиля (Change Profile). Применяет выбранный профиль к текущей вкладке.
- Редактирование текущего профиля (Edit Current Profile). Открывает диалоговое окно, в котором можно изменить широкий перечень параметров профиля.
- Управление профилями (Manage Profiles). Позволяет определить какой-то конкретный профиль в качестве профиля по умолчанию. Кроме того, с ее помощью можно создавать и удалять профили. К тому же эта команда управляет последовательностью отображения профилей в меню **Файл** (File).
- Настройка комбинаций клавиш (Configure Shortcuts). Определяет комбинации клавиш для команд Konsole.
- Настройка уведомлений (Configure Notifications). Применяется для определения действий, связанных с конкретными событиями сеанса.
- Настройка Konsole (Configure Konsole). Создает определяемые пользователем схемы и сеансы Konsole.

Область **Настройка уведомлений** предоставляет весьма широкие возможности. Она позволяет связывать пять конкретных событий, которые могут происходить в сеансе, с шестью разными действиями. При возникновении одного из этих событий выполняется одно или несколько определенных действий.

Параметры **Редактирование текущего профиля** (Edit Current Profile) представляют собой мощное инструментальное средство, обеспечивающее расширенное управление функциями работы с профилями. Это диалоговое окно предоставляет способ создания и сохранения разнообразных профилей для последующего использования. На рис. 2.8 показано основное диалоговое окно **Редактирование текущего профиля** (Edit Current Profile).

Диалоговое окно редактирования текущего профиля состоит из шести областей с вкладками.

- **Общее** (General). Эта область позволяет задать имя профиля, значок, начальный каталог файловой системы и т.д. Можно также определить команду, выполняемую после открытия вкладки. Эта команда, как правило, указывает на командный интерпретатор `bash`, `/bin/bash`, но может также представлять собой обычную команду командного интерпретатора, такую как `top`.
- **Вкладки** (Tabs). Эта команда определяет формат названия вкладки и позицию полос прокрутки вкладки.
- **Внешний вид** (Appearance). Окно **Внешний вид** содержит такие элементы, как цветовая схема вкладки и настройки шрифтов.
- **Прокрутка** (Scrolling). В число настроек входят размер буфера обратной прокрутки и местоположение полосы прокрутки в окне.
- **Ввод** (Input). В этой области можно определить так называемые *привязки клавиш*, которые указывают, какие коды символов должны передаваться в программу эмуляции терминала при нажатии определенных комбинаций клавиш.
- **Дополнительные настройки** (Advanced). Это окно позволяет задавать некоторые дополнительные настройки. К ним относятся функции терминала, кодировка символов, команды работы с мышью и средства курсора.

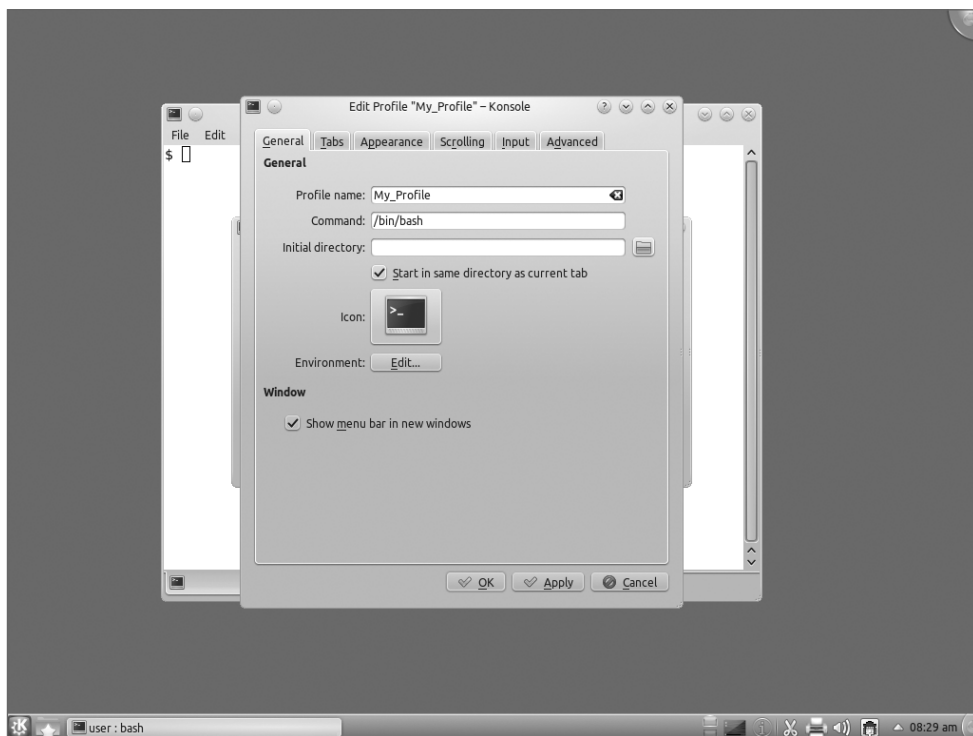


Рис. 2.8. Диалоговое окно редактирования текущего профиля программы Konsole

Справка (Help)

Элемент меню Справка позволяет получить доступ к полному руководству по программе Konsole (если при установке дистрибутива Linux были развернуты руководства KDE), к средству Совет дня (Tip of the day), которое показывает интересные, но малоизвестные способы упрощения работы и рекомендации при каждом запуске программы Konsole, а также к стандартному диалоговому окну О программе Konsole (About Konsole).

Терминал GNOME

Как и следовало ожидать, в проекте рабочего стола GNOME предусмотрена собственная программа эмуляции терминала. Пакет программ терминала GNOME предоставляет во многом такие же возможности, как Konsole и xterm. В настоящем разделе приведено краткое описание различных функций настройки и использования терминала GNOME.

Параметры командной строки

В программе терминала GNOME также предусмотрен широкий набор параметров командной строки, которые позволяют определить поведение терминала GNOME при его запуске. В табл. 2.4 перечислены имеющиеся параметры.

Таблица 2.4. Параметры командной строки терминала GNOME

Параметр	Описание
<code>-e command</code>	Выполнение программы, указанной данным параметром, в применяемом по умолчанию окне терминала
<code>-x</code>	Выполнение всего содержимого командной строки, которое следует за этим параметром, в применяемом по умолчанию окне терминала
<code>--window</code>	Открытие нового окна с заданным по умолчанию окном терминала. Предусмотрена возможность добавления нескольких параметров <code>--window</code> для запуска нескольких окон
<code>--window-with-profile=</code>	Открытие нового окна с указанным профилем. Этот параметр также может быть задан несколько раз в командной строке
<code>--tab</code>	Открытие нового терминала с вкладками в последнем открытом окне терминала
<code>--tab-with-profile=</code>	Открытие нового терминала с вкладками в последнем открытом окне терминала с использованием указанного профиля
<code>--role=</code>	Задание роли для последнего указанного окна
<code>--show-menubar</code>	Включение строки меню в верхней части окна терминала
<code>--hide-menubar</code>	Отключение строки меню в верхней части окна терминала
<code>--full-screen</code>	Отображение полностью развернутого окна терминала
<code>--geometry=</code>	Задание параметра геометрии X Window
<code>--disable-factory</code>	Отмена регистрации на сервере имен активизации
<code>--use-factory</code>	Регистрация на сервере имен активизации
<code>--startup-id=</code>	Задание идентификатора для протокола уведомления о запуске Linux
<code>-t, --title=</code>	Задание заголовка для окна терминала
<code>--working-directory=</code>	Задание применяемого по умолчанию рабочего каталога для окна терминала
<code>--zoom=</code>	Задание коэффициента масштабирования терминала
<code>--active</code>	Задание последней указанной вкладки терминала в качестве активной вкладки

Параметры командной строки терминала GNOME позволяют автоматически задавать использование многих средств терминала GNOME при его запуске. Тем не менее большинство из этих средств можно также задать из окна терминала GNOME после его развертывания.

Вкладки

По аналогии с терминалом Konsole, на терминале GNOME для каждого сеанса применяется отдельная *вкладка*, кроме того, вкладки используются для отслеживания нескольких сеансов, работающих в одном и том же окне. На рис. 2.9 показано окно терминала GNOME с тремя активными вкладками сеансов.

Предусмотрена возможность щелкнуть правой кнопкой мыши в окне вкладки для вызова на экран контекстного меню, позволяющего выполнять определенные действия в сеансе работы с вкладкой, как описано ниже.

- Открытие терминала (Open Terminal). Открывает новое окно терминала GNOME с заданным по умолчанию сеансом работы с вкладкой.
- Открытие вкладки (Open Tab). Открывает новую вкладку сеанса в существующем окне терминала GNOME.

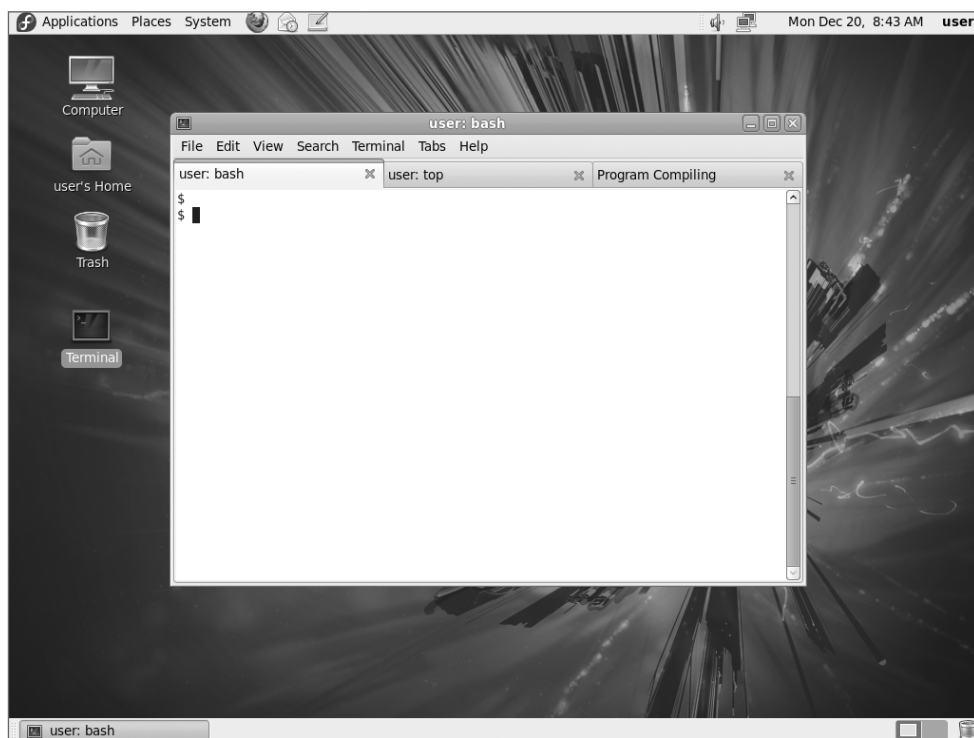


Рис. 2.9. Терминал GNOME с тремя активными сеансами

- **Заккрытие вкладки (Close Tab) или закрытие окна (Close Window).** Если открыто несколько вкладок, отображается элемент меню **Заккрытие вкладки**, которое позволяет закрыть текущую вкладку сеанса. Если открыта только одна вкладка, отображается команда меню **Заккрытие окна**, с помощью которой можно закрыть все окно терминала GNOME.
- **Копирование (Copy).** Копирует текст, выделенный подсветкой в текущей вкладке сеанса, в буфер обмена.
- **Вставка (Paste).** Вставляет данные из буфера обмена в текущей позиции курсора на текущей вкладке сеанса.
- **Профили (Profiles).** Позволяет изменить профиль текущего сеанса работы с вкладками или отредактировать текущий профиль вкладки.
- **Отображение строки меню (Show Menubar).** Включает или отключает отображение строки меню.
- **Методы ввода (Input Methods).** Изменяет текущий метод ввода путем перехода к другому символьному преобразованию или полностью выключает его.

С помощью контекстного меню можно быстрее получить доступ к часто выполняемым действиям, которые представлены в стандартной строке меню в окне терминала.

Строка меню

Но основной объем операций с терминалом GNOME выполняется с помощью строки меню, которая содержит все параметры конфигурации и настройки, необходимые для подготовки терминала GNOME к работе в требуемом режиме. Различные элементы строки меню рассматриваются в следующих разделах.

Файл (File)

Элемент меню **File** (Файл) содержит элементы, позволяющие создавать вкладки терминала и управлять ими следующим образом.

- **Открытие терминала (Open Terminal).** Запуск нового сеанса командного интерпретатора в новом окне терминала GNOME.
- **Открытие вкладки (Open Tab).** Запуск нового сеанса командного интерпретатора на новой вкладке в существующем окне терминала GNOME.
- **Новый профиль (New Profile).** Эта команда позволяет настраивать сеанс вкладки и сохранять настройку в виде профиля, который можно в дальнейшем выбрать для использования.
- **Сохранение содержимого (Save Contents).** С помощью этой команды можно сохранить содержимое буфера обратной прокрутки в текстовом файле.
- **Закрытие вкладки (Close Tab).** Закрытие текущей вкладки в окне.
- **Закрытие окна (Close Window).** Данная команда позволяет прекратить текущий сеанс терминала GNOME и закрыть все активные вкладки.

Доступ к большинству элементов меню **File** можно также получить, щелкнув правой кнопкой мыши в области вкладки сеанса. Элемент **New Profile** (Новый профиль) дает возможность настроить параметры вкладки сеанса и сохранить эти параметры для дальнейшего использования.

При использовании элемента **New Profile** необходимо вначале предоставить имя для нового профиля, чтобы открыть диалоговое окно **Editing Profile** (Редактирование профиля), которое показано на рис. 2.10.

В этой области, которая, в свою очередь, состоит из следующих шести областей, можно уточнить характеристики эмуляции терминала сеанса.

- **Общее (General).** Задаются общие параметры, такие как шрифт, звуковой сигнал и элементы строки меню.
- **Заголовок и команда (Title and Command).** Задается заголовок для вкладки сеанса (отображаемый на вкладке) и указывается, начинается ли сеанс с выполнения специальной команды или с вызова командного интерпретатора.
- **Цвета (Colors).** Задаются цвета переднего плана и фона, используемые на вкладке сеанса.
- **Фон (Background).** Данная область позволяет определить фоновое изображение для вкладки сеанса или сделать фон прозрачным, чтобы рабочий стол был виден через вкладку сеанса.
- **Прокрутка (Scrolling).** Эта область управляет тем, должна ли быть создана область прокрутки и какой размер она должна иметь.
- **Совместимость (Compatibility).** Здесь можно указать, какие управляющие коды передаются в систему при нажатии клавиш `<Backspace>` и `<Delete>`.

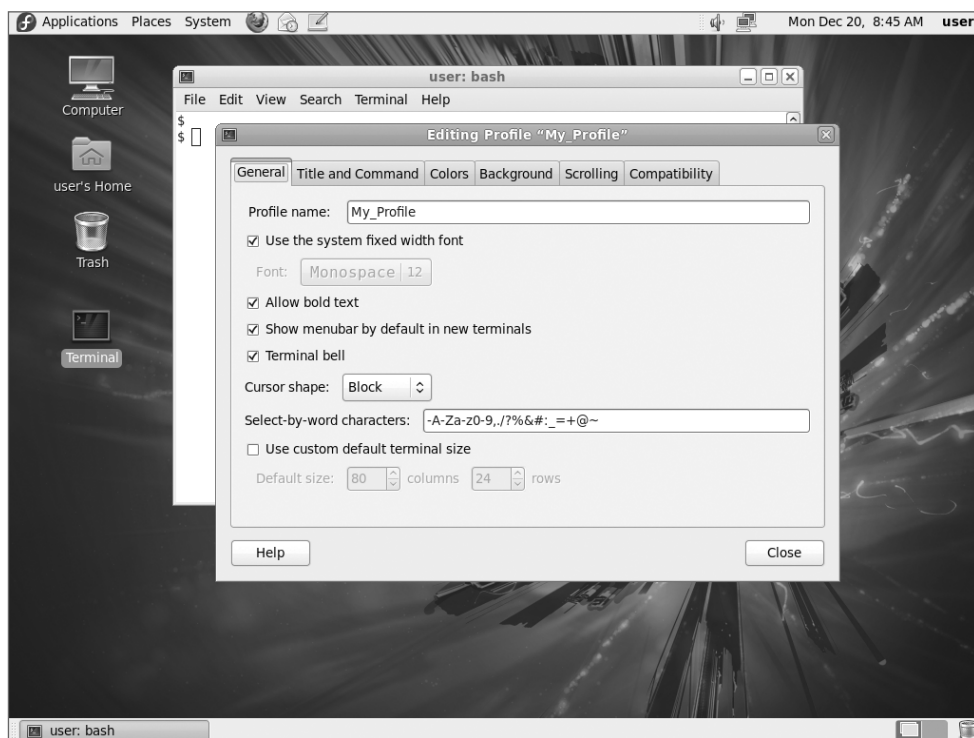


Рис. 2.10. Диалоговое окно **Editing Profile** терминала GNOME

После настройки профиля можно задать этот параметр при открытии нового сеанса работы с вкладками.

Редактирование (Edit)

Элемент меню **Edit** (Редактирование) содержит элементы, с помощью которых можно обрабатывать текст на вкладках. Предусмотрена возможность копировать и вставлять текст в любом месте в окне вкладки с помощью мыши. Это позволяет легко скопировать текст вывода из командной строки в буфер обмена, а затем перенести его в редактор. Можно также вставлять в окно сеанса работы с вкладкой текст из другого приложения GNOME.

- Копирование (Copy). Копирование выбранного текста в буфер обмена GNOME.
- Вставка (Paste). Вставка текста из буфера обмена GNOME в окно сеанса работы с вкладкой.
- Выделение всего (Select All). Выборка всего содержимого буфера обратной прокрутки.
- Профили (Profiles). Добавление, удаление или изменение профилей в терминале GNOME.
- Комбинации клавиш (Keyboard Shortcuts). Создание комбинации клавиш для ускорения доступа к средствам терминала GNOME.
- Приоритеты профиля (Profile Preferences). С помощью этой команды можно легко отредактировать профиль, используемый для текущей вкладки сеанса.

Эта функция редактирования профиля представляет собой чрезвычайно мощное инструментальное средство настройки нескольких профилей с последующей сменой профилей при переходе от одного сеанса к другому.

Представление (View)

Меню View (Представление) содержит элементы, позволяющие управлять внешним видом окна сеанса работы с вкладками.

- Отображение строки меню (Show Menubar). Включает или отключает отображение строки меню.
- Полноэкранное изображение (Full Screen). Увеличивает размеры окна терминала GNOME до размеров рабочего стола.
- Увеличение (Zoom In). Увеличивает размер шрифта в окне вкладки.
- Уменьшение (Zoom Out). Уменьшает размер шрифта в окне вкладки.
- Обычный размер (Normal Size). С помощью этой команды можно вернуться к применяемому по умолчанию размеру шрифта в окне вкладки.

Если строка меню оказалась скрытой, можно ее снова вывести на экран. Для этого достаточно щелкнуть правой кнопкой мыши в окне любого сеанса работы с вкладками и установить отметку Show Menubar (Показывать строку меню).

Терминал (Terminal)

Меню Terminal содержит элементы, позволяющие управлять средствами эмуляции терминала в сеансе работы с вкладками. К ним относятся команды, перечисленные ниже.

- Смена профиля (Change Profile). Переключает на другой настроенный профиль на вкладке сеанса.
- Задание заголовка (Set Title). Задает заголовок на вкладке сеанса, чтобы можно было легко определить, что это за сеанс.
- Задание кодировки символов (Set Character Encoding). Выбирает кодировку, используемую для передачи и отображения символов.
- Сброс (Reset). С помощью этого элемента меню можно отправить управляющий код сброса в систему Linux.
- Сброс и очистка (Reset and Clear). Позволяет отправить управляющий код сброса в систему Linux и удалить весь текст, отображаемый в настоящее время в области вкладки.
- Список размеров окон (Window Size List). Открывает список размеров, которые можно задать для текущего окна терминала GNOME. Достаточно выбрать необходимые размеры, и размеры окна будут откорректированы автоматически.

Что касается кодировок символов, то на выбор предоставляется большой список имеющихся наборов символов. Это особенно удобно, если приходится работать не на английском языке.

Вкладки

Меню Tabs (Вкладки) содержит элементы, позволяющие управлять местоположением вкладок и выбирать, какая вкладка является активной. Данное меню отображается, если открыто несколько сеансов работы с вкладками.

- Следующая вкладка (Next Tab). Активизирует следующую вкладку в списке.

- Предыдущая вкладка (Previous Tab). Активизирует предыдущую вкладку в списке.
- Сдвиг вкладки влево (Move Tab to the Left). Меняет местами текущую и предыдущую вкладки.
- Сдвиг вкладки вправо (Move Tab to the Right). Меняет местами текущую и следующую вкладки.
- Отсоединение вкладки (Detach Tab). Удаляет вкладку из текущего окна и открывает новое окно терминала GNOME с использованием данной вкладки.
- Список вкладок (The Tab List). Выводит список сеансов работы с вкладками, выполняющихся в настоящее время в окне терминала. Выберите вкладку, чтобы быстро перейти к требуемому сеансу.

Этот раздел позволяет управлять вкладками, что может оказаться необходимым, если работа ведется с несколькими одновременно открытыми вкладками.

Справка (Help)

Элемент меню Help (Справка) предоставляет доступ к полному руководству по терминалу GNOME, с помощью которого можно изучать отдельные элементы и средства, предусмотренные в терминале GNOME.

Резюме

Чтобы приступить к изучению команд командной строки Linux, необходимо получить доступ к командной строке. В мире графических интерфейсов эта задача иногда становится сложной. В настоящей главе изложены некоторые соображения, которые необходимо учитывать при получении доступа к командной строке Linux из среды графического рабочего стола. В начале главы рассматривается тематика эмуляции терминала и показано, какими функциями необходимо воспользоваться, чтобы обеспечить взаимодействие системы Linux с применяемым пакетом эмуляции терминала, а также отображение текста и графики должным образом.

В частности, рассматриваются три типа эмуляторов терминала. Первым пакетом эмулятора терминала, доступным для Linux, был `xterm`. Он эмулирует и терминалы VT102, и терминалы Tektronix 4014. В рамках проекта рабочего стола KDE создан пакет эмуляции терминала Konsole. Этот пакет характеризуется наличием нескольких привлекательных средств, таких как возможность открывать несколько сеансов в одном и том же окне, использовать и сеансы консоли, и сеансы `xterm`, а также иметь полный контроль над параметрами эмуляции терминала.

Наконец, в главе описан пакет эмуляции терминала GNOME, разработанный в рамках проекта рабочего стола GNOME. Терминал GNOME также позволяет открывать несколько сеансов терминала в одном окне, а кроме того, предоставляет удобный способ настройки многих функций терминала.

В следующей главе приведено вступительное описание команд командной строки Linux. В ней рассматриваются команды, необходимые для перехода по файловой системе Linux, а также для создания, удаления файлов и управления ими.

Основные команды интерпретатора bash

По умолчанию во всех дистрибутивах Linux используется командный интерпретатор bash из состава утилит GNU. В настоящей главе рассматриваются основные возможности командного интерпретатора bash и кратко описано, как работать с файлами и каталогами Linux, используя основные команды, предоставляемые командным интерпретатором bash. Читатели, которые уже хорошо освоили работу с файлами и каталогами в среде Linux, вполне могут пропустить эту главу и перейти к изучению главы 4, в которой представлены более сложные команды bash.

Запуск командного интерпретатора

Командный интерпретатор bash в составе утилит GNU — это программа, которая предоставляет интерактивный доступ к системе Linux. Командный интерпретатор функционирует как обычная программа, которая, как правило, запускается после каждой регистрации пользователя на терминале. При этом система может запускать разные командные интерпретаторы в зависимости от настройки, связанной с конкретным идентификатором пользователя.

Файл `/etc/passwd` содержит список всех учетных записей пользователей системы, а также некоторые основ-

ГЛАВА

3

В этой главе...

Запуск командного интерпретатора

Приглашение к вводу информации командного интерпретатора

Руководство по командам bash

Навигация в файловой системе

Листинги файлов и каталогов

Обработка файлов

Обработка каталогов

Просмотр содержимого файла

Резюме

ные сведения о конфигурации, относящиеся к каждому пользователю. Ниже приведен пример типичной записи из файла `/etc/passwd`.

```
rich:x:501:501:Rich Blum:/home/rich:/bin/bash
```

Каждая запись содержит семь полей данных, отделенных одно от другого двоеточием. Система использует данные этих полей для предоставления пользователю конкретных возможностей. Эти поля перечислены ниже.

- Имя пользователя.
- Пароль пользователя (или местозаполнитель, если пароль хранится в другом файле).
- Идентификационный номер пользователя, присвоенный пользователю в системе.
- Идентификационный номер группы, присвоенный пользователю в системе.
- Полное имя пользователя.
- Применяемый по умолчанию исходный каталог пользователя.
- Применяемая по умолчанию программа командного интерпретатора пользователя.

Большая часть этих полей будет рассматриваться более подробно в главе 6. А на данный момент достаточно отметить, что для пользователя указана применяемая программа командного интерпретатора.

В большинстве систем Linux при запуске среды интерфейса командной строки (command line interface — CLI) для взаимодействия с пользователем применяется предусмотренная по умолчанию программа командного интерпретатора `bash`. В программе `bash` также используются параметры командной строки для изменения типа запускаемого командного интерпретатора. В табл. 3.1 перечислены параметры командной строки, предусмотренные в программе `bash`, которые определяют, какой командный интерпретатор должен использоваться.

Таблица 3.1. Параметры командной строки `bash`

Параметр	Описание
<code>-c string</code>	Чтение команд из строки <code>string</code> и их обработка
<code>-r</code>	Запуск ограниченного командного интерпретатора, который вынуждает пользователя ограничиваться основным каталогом
<code>-i</code>	Запуск интерактивного командного интерпретатора, позволяющего принимать входные данные от пользователя
<code>-s</code>	Чтение команд из стандартного устройства ввода

По умолчанию командный интерпретатор `bash` при запуске автоматически обрабатывает команды из файла `.bashrc`, находящегося в исходном каталоге пользователя. Во многих дистрибутивах Linux этот файл служит также для загрузки общего файла, содержащего команды и настройки для всех пользователей в системе. Этот общий файл обычно хранится в системе под именем `/etc/bashrc`. В данном файле часто бывают заданы переменные среды (см. главу 5), используемые в различных приложениях.

Приглашение к вводу информации командного интерпретатора

Сразу после запуска пакета эмуляции терминала или входа в систему с консоли Linux пользователь получает *приглашение к вводу информации*, которое относится к интерфейсу командной строки командного интерпретатора. Это приглашение к вводу информации можно рассматривать как путь доступа к командному интерпретатору. Ввод команд командного интерпретатора осуществляется на строке, которая следует за приглашением.

По умолчанию символом приглашения к вводу информации для командного интерпретатора `bash` является знак доллара (`$`). Этот символ указывает, что командный интерпретатор ожидает ввода текста пользователем. Формат приглашения к вводу информации, используемый командным интерпретатором, можно изменить. В различных дистрибутивах Linux используются разные форматы приглашения. В рассматриваемой нами системе Ubuntu Linux приглашение командного интерпретатора `bash` выглядит следующим образом:

```
rich@user-desktop:$
```

В системе Fedora Linux приглашение выглядит так:

```
[rich@testbox]$
```

Предусмотрена возможность настраивать приглашение к вводу информации в целях получения с его помощью основных сведений о применяемой среде. В первом примере показаны следующие три фрагмента информации, содержащиеся в приглашении к вводу информации.

- Имя пользователя, который запустил командный интерпретатор.
- Номер текущей виртуальной консоли.
- Текущий каталог (знак тильды (`~`) сокращенно обозначает исходный каталог пользователя).

Второй пример предоставляет аналогичную информацию, если не считать того, что в нем используется имя хоста вместо номера виртуальной консоли. Предусмотрены две переменные среды, которые управляют форматом приглашения к вводу информации командной строки.

- `PS1`. Управляет форматом заданного по умолчанию приглашения к вводу информации командной строки.
- `PS2`. Управляет форматом приглашения к вводу информации командной строки второго уровня.

Командный интерпретатор использует заданное по умолчанию приглашение к вводу информации `PS1` для начального ввода данных в командном интерпретаторе. А после ввода команды, которая требует дополнительных сведений, командный интерпретатор отображает приглашение к вводу информации второго уровня, заданное переменной среды `PS2`.

Для отображения текущих настроек приглашений к вводу информации можно воспользоваться командой `echo`:

```
rich@ user-desktop:$ echo $PS1
${debian_chroot:+($debian_chroot)}\u@\h:\w$
rich@ user-desktop:$ echo $PS2
>
rich@ user-desktop:$
```

Формат переменных среды, относящихся к приглашениям к вводу информации, является довольно сложным. В командном интерпретаторе используются специальные символы для обозначения элементов приглашения к вводу информации командной строки. В табл. 3.2 показаны специальные символы, которые можно использовать в строке приглашения к вводу информации.

Таблица 3.2. Символы приглашения к вводу информации командного интерпретатора bash

Символ	Описание
\a	Символ звукового сигнала
\d	Дата в формате "день_недели_месяц_число"
\e	Экранирующий символ ASCII
\h	Локальное имя хоста
\H	Полное доменное имя хоста
\j	Количество заданий, выполняемых в настоящее время под управлением командного интерпретатора
\l	Имя без расширения терминального устройства командного интерпретатора
\n	Символ обозначения конца строки ASCII
\r	Символ возврата каретки ASCII
\s	Имя командного интерпретатора
\t	Текущее время в 24-часовом формате HH:MM:SS (чч:мм:сс)
\T	Текущее время в 12-часовом формате HH:MM:SS (чч:мм:сс)
\@	Текущее время в 12-часовом формате с указанием времени дня, am/pm (до полудня/после полудня)
\u	Имя пользователя, которое относится к текущему пользователю
\v	Версия командного интерпретатора bash
\V	Уровень выпуска командного интерпретатора bash
\w	Текущий рабочий каталог
\W	Имя без расширения текущего рабочего каталога
!\	Номер данной команды в журнале командного интерпретатора bash
\#	Номер команды, присвоенный данной команде
\\$	Знак доллара, если это — обычный пользователь, или знак дизеса, если это — пользователь root
\nnn	Символ, соответствующий восьмеричному значению nnn
\\	Наклонная черта влево
\[Начало последовательности управляющих кодов
\]	Конец последовательности управляющих кодов

Обратите внимание на то, что все специальные символы приглашения к вводу информации начинаются с наклонной черты влево (\). Именно это позволяет отличить символ приглашения к вводу информации от обычного текста в приглашении. В предыдущем примере приглашение к вводу информации содержало и символы приглашения, и обычные символы (знак "коммерческого at", @, и квадратные скобки). В применяемом приглашении к вводу информации можно определить любую комбинацию символов приглашения. Чтобы создать новый формат приглашения к вводу информации, достаточно присвоить новую строку переменной PS1:

```
[rich@testbox]$ PS1="[\t][\u]\$ "
```

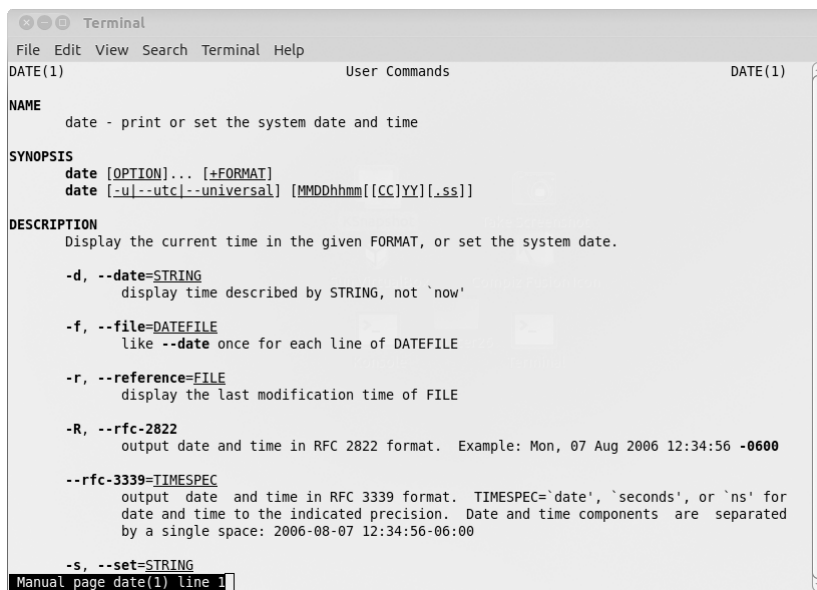
```
[14:40:32][rich]$
```

Это новое приглашение командного интерпретатора теперь показывает текущее время и имя пользователя. Новое определение переменной PS1 будет действовать только на протяжении текущего сеанса работы с командным интерпретатором. После очередного запуска командного интерпретатора опять загрузится предусмотренное по умолчанию определение для приглашения командного интерпретатора. В главе 5 будет показано, как изменить заданное по умолчанию приглашение командного интерпретатора для всех сеансов командного интерпретатора.

Руководство по командам bash

Большинство дистрибутивов Linux включает интерактивное руководство, предназначенное для поиска информации о командах командного интерпретатора, а также о многих других программах GNU, включенных в дистрибутив. Рекомендуется ознакомиться с этим руководством, поскольку оно является неоценимым источником сведений о работе с программами, особенно если требуется освоить работу с различными параметрами командной строки.

Доступ к справочным руководствам, хранимым в системе Linux, предоставляет команда `man`. После ввода команды `man`, за которой следует имя конкретной программы, открывается запись интерактивного руководства, относящаяся к этой программе. На рис. 3.1 приведен пример поиска справочного руководства для команды `date`.



```
Terminal
File Edit View Search Terminal Help
DATE(1) User Commands DATE(1)

NAME
date - print or set the system date and time

SYNOPSIS
date [OPTION]... [+FORMAT]
date [-u|--utc|--universal] [MMDDhhmm[[CC]YY][.ss]]

DESCRIPTION
Display the current time in the given FORMAT, or set the system date.

-d, --date=STRING
    display time described by STRING, not 'now'

-f, --file=DATEFILE
    like --date once for each line of DATEFILE

-r, --reference=FILE
    display the last modification time of FILE

-R, --rfc-2822
    output date and time in RFC 2822 format. Example: Mon, 07 Aug 2006 12:34:56 -0600

--rfc-3339=TIMESPEC
    output date and time in RFC 3339 format. TIMESPEC='date', 'seconds', or 'ns' for
    date and time to the indicated precision. Date and time components are separated
    by a single space: 2006-08-07 12:34:56-06:00

-s, --set=STRING

Manual page date(1) line 1
```

Рис. 3.1. Отображение страниц справочного руководства для команды `date` системы Linux

В справочном руководстве различные сведения о команде распределены по отдельным разделам, которые приведены в табл. 3.3.

Предусмотрена возможность просматривать справочные руководства, нажимая клавишу пробела или используя клавиши со стрелками для прокрутки вперед и назад текста справочного руководства (при условии, что применяемый пакет эмуляции терминала поддерживает

функции клавиш со стрелками). После завершения работы со страницами справочного руководства нажмите клавишу <q>, чтобы выйти из программы.

Таблица 3.3. Формат справочного руководства Linux

Раздел	Описание
Name (Имя)	Отображает имя команды и содержит краткое описание
Synopsis (Краткий обзор)	Показан формат команды
Описание	Содержит описание всех параметров команды
Author (Автор)	Предоставляет информацию о том, кто разработал команду
Reporting bugs (Отправка сообщений об ошибках)	Предоставляет информацию о том, куда следует сообщать об обнаруженных ошибках
Copyright (Авторское право)	Содержит сведения о состоянии авторских прав на код команды
See Also (См. также)	Предоставляет ссылки на все прочие аналогичные команды

Для просмотра информации о командном интерпретаторе `bash` можно открыть справочное руководство по этой программе, используя следующую команду:

```
$ man bash
```

Откроется окно, которое позволяет выполнить пошаговый просмотр всего справочного руководства, относящегося к командному интерпретатору `bash`. Это особенно удобно при написании сценариев, поскольку не приходится обращаться к книгам или интернет-сайтам для поиска конкретных форматов команд. Данное руководство всегда доступно в открытом сеансе.

Навигация в файловой системе

После открытия командного интерпретатора можно увидеть в его приглашении, что запуск сеанса командного интерпретатора обычно влечет за собой переход пользователя в его исходный каталог. Но во многих случаях возникает необходимость выйти за пределы исходного каталога и выполнять различные действия в других областях в системе Linux. В настоящем разделе описано, как перейти в другой каталог с помощью команд командного интерпретатора. Однако, прежде чем приступить к изучению этой темы, необходимо вкратце рассмотреть, что представляет собой файловая система Linux, чтобы понять, в чем состоит переход по ее каталогам.

Файловая система Linux

Те, кто только осваивают систему Linux, могут почувствовать растерянность, узнав о том, как в ней обозначаются файлы и каталоги, особенно если перед этим приходилось знакомиться лишь с организацией работы с файлами в операционной системе Microsoft Windows. Прежде чем приступить к изучению системы Linux, необходимо понять, как она устроена.

Первое отличие, которое можно заметить, состоит в том, что в именах путей Linux не используются имена дисков. В мире Windows имена путей к файлам всегда содержат буквенные обозначения дисководов, установленных на персональном компьютере. В системе Windows каждый физический дисковый накопитель обозначается буквой, а каждый диск имеет собственную структуру каталогов, используемую для получения доступа к хранящимся на нем файлам.

Например, в Windows часто можно видеть примерно такое обозначение пути к файлам:

```
c:\Users\Rich\Documents\test.doc
```


Это обозначение указывает, что файл `test.doc` находится в каталоге `Documents`, который непосредственно находится в каталоге `Rich`. Каталог `Rich` расположен под каталогом `Users`, который находится в разделе жесткого диска с присвоенной ему буквой `C` (обычно таковым является первый жесткий диск на персональном компьютере).

Путь к файлу `Windows` позволяет точно узнать, в каком физическом разделе диска содержится файл `test.doc`. Например, если файл должен быть сохранен на флеш-диске, то он может быть обозначен буквой диска `J`. После щелчка на значке диска `J`, находящемся на рабочем столе, автоматически открывается доступ к файлу `J:\test.doc`. Этот путь указывает, что данный файл находится в корне диска, которому присвоено буквенное обозначение `J`.

В системе `Linux` используется другой метод обозначения файлов. Все файлы `Linux` хранятся в единой структуре каталогов, называемой *виртуальным каталогом*. Виртуальный каталог определяет пути к файлам, находящимся на всех запоминающих устройствах, которые установлены на персональном компьютере, объединяя их в единую структуру каталогов.

Структура виртуального каталога `Linux` включает единственный основной каталог, называемый *корнем*. Каталоги и файлы, находящиеся под корневым каталогом, перечисляются с учетом пути к каталогу, используемого для их достижения, аналогично тому, как это происходит в системе `Windows`.



Заслуживает внимания то, что в системе `Linux` для обозначения каталогов в пути к файлу используется косая черта (`/`) вместо обратной косой черты (`\`). Символ обратной косой черты в `Linux` обозначает экранирующий символ и при его использовании в обозначениях путей к файлам вызывает многочисленные проблемы. Но у тех, кто привык работать в среде `Windows`, может выработаться устойчивая привычка использовать обратную косую черту в обозначениях путей.

При этом, например, применяемый в системе `Linux` путь к файлу `/home/rich/Documents/test.doc` указывает лишь то, что файл `test.doc` находится в каталоге `Documents`, расположенном в каталоге `rich`, который содержится в каталоге `home`. Это обозначение не предоставляет никакой информации о том, на каком физическом диске персонального компьютера хранится этот файл.

Сложность, связанная с использованием виртуального каталога `Linux`, заключается в том, что в него входят структуры каталогов всех запоминающих устройств на компьютере. Первый жесткий диск, установленный в персональном компьютере `Linux`, именуется *корневым диском*. Корневой диск содержит ядро виртуального каталога. От него исходят все остальные части виртуального каталога.

На корневом диске система `Linux` создает специальные каталоги, называемые *точками монтирования*. Точки монтирования — это каталоги в виртуальном каталоге, за которыми закрепляются дополнительные запоминающие устройства.

Монтирование запоминающих устройств в виртуальном каталоге приводит к появлению файлов и каталогов этих запоминающих устройств под точками монтирования в каталогах монтирования, без учета того, что они фактически могут находиться на других дисках.

На корневом диске часто физически располагаются системные файлы, в то время как пользовательские файлы хранятся на другом диске (рис. 3.2).

Как показывает рис. 3.2, на персональном компьютере имеются два жестких диска. Один жесткий диск связан с корнем виртуального каталога (что обозначено отдельной косой чертой). Другие жесткие диски могут быть смонтированы в определенных местах виртуальной структуры каталогов. В данной примере второй жесткий диск смонтирован в местоположении `/home`, которое представляет собой то место, где находятся пользовательские каталоги.

Структура файловой системы Linux

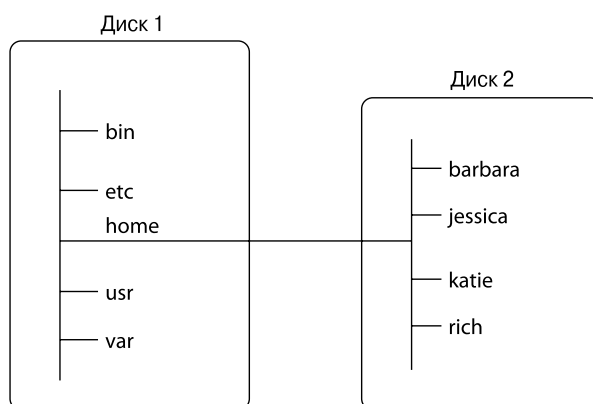


Рис. 3.2. Файловая архитектура системы Linux

Структура файловой системы Linux была разработана на основе файловой архитектуры Unix. К сожалению, файловая архитектура Unix за истекшие годы в связи с появлением различных разновидностей Unix потеряла былое единообразие. Складывается такое впечатление, будто нельзя найти такие две системы Unix или Linux, в которых применялась бы полностью одинаковая структура файловой системы. Тем не менее предусмотрено несколько общепринятых имен каталогов, которые предназначены для выполнения примерно одних и тех же функций. В табл. 3.4 перечислены некоторые из наиболее широко применяемых имен подкаталогов виртуального каталога Linux.

Таблица 3.4. Общепринятые имена каталогов Linux

Каталог	Назначение
/	Корень виртуального каталога. Обычно принято не размещать в корне какие-либо файлы
/bin	Каталог двоичных файлов, в котором хранится большое количество программ пользовательского уровня, относящихся к категории утилит GNU
/boot	Загрузочный каталог, в котором хранятся загрузочные файлы
/dev	Каталог устройств, в котором Linux создает специальные файлы устройств
/etc	Каталог файлов конфигурации системы
/home	Исходный каталог, в котором Linux создает пользовательские каталоги
/lib	Библиотечный каталог, в котором хранятся файлы библиотек системы и приложений
/media	Мультимедийный каталог — общее место для точек монтирования, используемых для сменных носителей
/mnt	Каталог монтирования — еще одно общепринятое место для точек монтирования, используемых для сменных носителей
/opt	Дополнительный каталог, который часто служит для хранения дополнительных пакетов программ
/root	Корневой исходный каталог
/sbin	Системный каталог двоичных файлов, в котором хранится много административных программ на уровне GNU
/tmp	Временный каталог, в котором могут создаваться и уничтожаться временные рабочие файлы

Каталог	Назначение
/usr	Каталог программ, установленных пользователем
/var	Каталог с изменчивым содержимым, предназначенный для часто изменяющихся файлов, таких как файлы журналов

После запуска нового приглашения командного интерпретатора открывается сеанс с переходом в исходный каталог пользователя, представляющий собой уникальный каталог, который предназначен для работы только с определенной учетной записью пользователя. При создании учетной записи пользователя система обычно назначает уникальный каталог исключительно для этой учетной записи (см. главу 6).

В мире Windows для перемещения по структуре каталогов широко применяется графический интерфейс. А для перехода по виртуальному каталогу в приглашении интерфейса командной строки применяется команда `cd`, которую необходимо изучить.

Переход по каталогам

В файловой системе Linux для перехода в другой каталог в сеансе командного интерпретатора используется команда смены каталога `cd` (change directory). Формат команды `cd` является относительно несложным:

```
cd destination
```

Команда `cd` может принимать единственный параметр назначения, *destination*, указывающий имя каталога, в который необходимо перейти. Если в команде `cd` не указано назначение, то происходит переход в исходный каталог пользователя.

Но параметр назначения может быть выражен с использованием двух различных способов:

- в виде абсолютного пути к файлу;
- как относительный путь к файлу.

В следующих разделах описаны различия между этими двумя методами обозначения путей.

Абсолютный путь к файлу

Для ссылки на имя каталога, входящего в состав виртуального каталога, может, прежде всего, использоваться обозначение *абсолютного пути к файлу*. Абсолютный путь к файлу точно определяет, где находится каталог в структуре виртуального каталога, начиная от корня виртуального каталога и кончая именем самого каталога, что приводит к получению полного имени каталога.

Таким образом, для ссылки на каталог `apache`, содержащийся в каталоге `lib`, который в свою очередь содержится в каталоге `usr`, можно использовать следующий абсолютный путь к файлу:

```
/usr/lib/NetworkManager
```

Зная абсолютный путь к файлу, можно не сомневаться в том, что поиск объекта, на который он указывает, будет успешным. Чтобы перейти в конкретное местоположение в файловой системе с использованием абсолютного пути к файлу, достаточно указать это полное имя пути в команде `cd`:

```
rich@testbox[]$cd /etc
rich@testbox[etc]$
```

Приглашение к вводу информации показывает, что новым каталогом для командного интерпретатора после выполнения этой команды `cd` теперь является `/etc`. С помощью абсо-

лютного пути к файлу можно перейти на любой уровень во всей структуре виртуального каталога Linux:

```
rich@testbox[]$ cd /usr/lib/NetworkManager
rich@testbox[NetworkManager]$
```

Но если требуется обеспечить лишь работу, допустим, в пределах собственной структуры исходного каталога, то применение абсолютных путей к файлу часто становится излишне трудоемким. Например, если уже произошел переход в каталог `/home/rich`, то кажется довольно неудобным применение примерно такой команды:

```
cd /home/rich/Documents
```

лишь для перехода в подкаталог Documents. К счастью, существует более простое решение.

Относительные пути к файлу

Относительные пути к файлу позволяют указывать путь к требуемому файлу относительно текущего местоположения в файловой системе, не вынуждая начинать с корня. Обозначение относительного пути к файлу не начинается с косой черты, которая указывает на корневой каталог.

Вместо этого обозначение относительного пути к файлу начинается либо с имени каталога (если происходит переход к каталогу, расположенному ниже текущего каталога), либо со специального символа, указывающего местоположение относительно текущего каталога. Для этого используются два специальных символа:

- точка (`.`), которая обозначает текущий каталог;
- двойная точка (`..`), которая представляет родительский каталог.

Символ двойной точки становится чрезвычайно удобным, если требуется выполнить переход по иерархии каталогов. Например, если текущим является подкаталог Documents исходного каталога и требуется перейти в каталог Desktop, который также расположен ниже исходного каталога, эту задачу можно решить следующим образом:

```
rich@testbox[Documents]$ cd ../Desktop
rich@testbox[Desktop]$
```

Символ двойной точки обеспечивает переход вверх на один уровень в структуре исходного каталога, а затем часть относительного пути `/Desktop` позволяет снова перейти на более низкий уровень, но уже в каталог Desktop. Для перемещения по структуре каталогов можно столько раз использовать символ двойной точки, сколько потребуется. Например, если необходимо перейти в каталог `/etc` из текущего каталога, каковым является исходный каталог (`/home/rich`), то можно ввести следующее:

```
rich@testbox[]$ cd ../../etc
rich@testbox[etc]$
```

Разумеется, в подобных случаях фактически приходится применять больший объем ввода, чтобы воспользоваться относительным путем к файлу, по сравнению с непосредственным указанием абсолютного пути к файлу, `/etc`!

Листинги файлов и каталогов

Одной из наиболее важных функций командного интерпретатора является предоставление возможности получения сведений о том, какие файлы имеются в системе. Инструментом, по-

звляющим достичь этой цели, является команда `ls` (сокращение от `list` — список). В настоящем разделе рассматривается команда `ls` и приведено описание всех параметров, применяемых для форматирования данных, которые может предоставить эта команда.

Основной формат листинга

Команда `ls`, применяемая в наиболее простой форме, отображает файлы и каталоги, находящиеся в текущем каталоге:

```
$ ls
4rich Desktop Download Music Pictures store store.zip test
backup Documents Drivers myprog Public store.sql Templates Videos
```

Заслуживает внимания то, что команда `ls` формирует листинг в алфавитном порядке (это касается столбцов, а не строк). Если для работы используется эмулятор терминала, который поддерживает цвет, то с помощью команды `ls` можно также сформировать вывод, в котором записи разных типов обозначены различными цветами. Этим средством управляет переменная среды `LS_COLORS`. В различных дистрибутивах Linux эта переменная среды задается в зависимости от возможностей эмулятора терминала.

Если же применяемый эмулятор терминала не поддерживает цвет, то в команде `ls` можно задать параметр `-F`, позволяющий сформировать вывод, в котором проще отличить файлы от каталогов. Применение параметра `-F` приводит к получению следующего вывода:

```
$ ls -F
4rich/      Documents/ Music/      Public/      store.zip  Videos/
backup.zip  Download/  myprog*    store/       Templates/
Desktop/    Drivers/   Pictures/   store.sql    test
$
```

В выводе, сформированном с помощью параметра `-F`, каталоги обозначены косой чертой, что позволяет проще находить их в листинге. Кроме того, данный параметр обеспечивает применение звездочки для обозначения исполняемых файлов (что можно видеть выше на примере файла `myprog`), а это позволяет проще находить файлы, которые могут быть вызваны на выполнение в системе.

Тем не менее применение команды `ls` в основной форме может привести к некоторой путанице. Она показывает файлы и каталоги, содержащиеся в текущем каталоге, но не обязательно все. В системе Linux часто используются *скрытые файлы* для хранения информации о конфигурации. Скрытые файлы в системе Linux — это файлы с именами файлов, начинающимися с точки. Такие файлы не появляются в создаваемом по умолчанию листинге `ls` (именно поэтому они называются скрытыми).

Для отображения скрытых файлов наряду с обычными файлами и каталогами используется параметр `-a`. Пример применения параметра `-a` с командой `ls` приведен на рис. 3.3.

Очевидно, что применение указанного параметра приводит к существенному изменению листинга. В исходном каталоге пользователя, который зарегистрировался в системе с помощью графического рабочего стола, обнаруживается большое количество скрытых файлов конфигурации. Данный конкретный пример относится к пользователю, который зарегистрировался в сеансе работы с рабочим столом GNOME. Следует также отметить, что в данном случае обнаруживаются три файла, имена которых начинаются с подстроки `.bash`. Это — скрытые файлы, которые используются в среде командного интерпретатора `bash`. Данная тема рассматривается более подробно в главе 5.

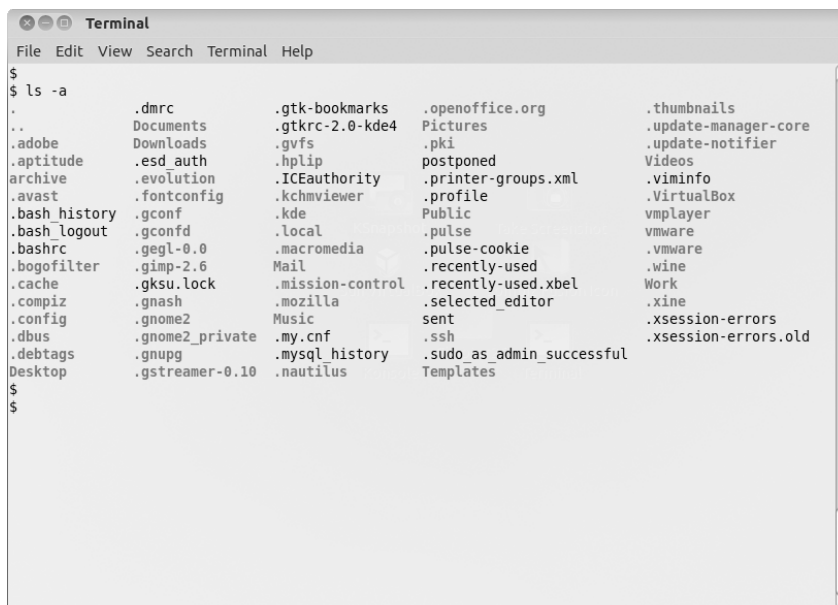


Рис. 3.3. Использование параметра -a в команде ls

Может также применяться еще один важный параметр команды `ls` — параметр `-R`, который позволяет узнать, какие файлы находятся в подкаталогах текущего каталога. Если количество подкаталогов достаточно велико, то листинг, формируемый с применением указанного параметра, может оказаться весьма длинным. Ниже приведен простой пример результатов, полученных при использовании параметра `-R`.

```
$ ls -F -R
.:
file1 test1/  test2/

./test1:
myprog1*  myprog2*

./test2:
$
```

Заслуживает внимания то, что параметр `-R` обеспечивает получение в первую очередь содержимого текущего каталога, каковым является файл (`file1`) и два каталога (`test1` и `test2`). Вслед за этим, в соответствии с назначением параметра `-R`, происходит переход по содержанию этих двух каталогов и отображение всех файлов, содержащихся в каждом из них. В каталоге `test1` обнаруживаются два файла (`myprog1` и `myprog2`), а в каталоге `test2` отсутствуют какие-либо файлы. Если бы в каталогах `test1` и `test2` находились каталоги более низкого уровня, то в связи с применением параметра `-R` происходил бы дальнейший перебор и этих подкаталогов. Вполне очевидно, что при наличии достаточно развитой структуры каталогов объем листинга может стать весьма значительным.

Изменение формата представленной информации

Приведенные выше листинги показывают, что команда `ls` в своей основной форме представляет лишь наиболее важные сведения о каждом файле. Для получения в листинге дополнительных сведений служит еще один широко известный параметр, `-l`. Параметр `-l` (сокращение от `long` — длинный) позволяет получить так называемый длинный формат листинга, в котором приведена подробная информация о каждом файле в каталоге:

```
$ ls -l
total 2064
drwxrwxr-x  2 rich rich    4096 2010-08-24 22:04 4rich
-rw-r--r--  1 rich rich 1766205 2010-08-24 15:34 backup.zip
drwxr-xr-x  3 rich rich    4096 2010-08-31 22:24 Desktop
drwxr-xr-x  2 rich rich    4096 2009-11-01 04:06 Documents
drwxr-xr-x  2 rich rich    4096 2009-11-01 04:06 Download
drwxrwxr-x  2 rich rich    4096 2010-07-26 18:25 Drivers
drwxr-xr-x  2 rich rich    4096 2009-11-01 04:06 Music
-rwxr--r--  1 rich rich     30 2010-08-23 21:42 myprog
drwxr-xr-x  2 rich rich    4096 2009-11-01 04:06 Pictures
drwxr-xr-x  2 rich rich    4096 2009-11-01 04:06 Public
drwxrwxr-x  5 rich rich    4096 2010-08-24 22:04 store
-rw-rw-r--  1 rich rich  98772 2010-08-24 15:30 store.sql
-rw-r--r--  1 rich rich 107507 2010-08-13 15:45 store.zip
drwxr-xr-x  2 rich rich    4096 2009-11-01 04:06 Templates
drwxr-xr-x  2 rich rich    4096 2009-11-01 04:06 Videos
[rich@testbox]$
```

В каждой строке листинга в длинном формате содержатся сведения о различных файлах и каталогах, имеющихся в данном каталоге. Такой листинг, кроме имени файла, показывает другую полезную информацию. В первой строке вывода содержатся сведения об общем количестве блоков данных, относящихся к текущему каталогу. Вслед за этим происходит вывод отдельных строк, каждая из которых включает следующую информацию о каждом файле (или каталоге).

- Тип файла, такой как каталог (`d`), файл (`-`), символьное устройство (`c`) или блочное устройство (`b`).
- Разрешения для файла (см. главу 6).
- Количество жестких ссылок на файл (см. раздел “Формирование ссылок на файлы” ниже в этой главе).
- Имя пользователя владельца файла.
- Имя группы файлов, к которой принадлежит этот файл.
- Размер файла в байтах.
- Время последнего изменения файла.
- Имя файла или каталога.

Параметр `-l` — это мощное инструментальное средство, без которого трудно обойтись. Применение данного параметра позволяет получить почти всю информацию, которая может потребоваться для работы с любым файлом или каталогом в системе.

Полный список параметров

В команде `ls` предусмотрено также много других параметров, которые могут потребоваться при решении задач управления файлами. Вызвав на выполнение команду `man`, относящуюся к команде `ls`, можно увидеть, что список параметров, предназначенных для изменения формата вывода команды `ls`, занимает несколько страниц.

В команде `ls` используются параметры командной строки двух типов:

- однобуквенные (короткие) параметры;
- параметры, состоящие из полных слов (длинные параметры).

Однобуквенным параметрам всегда предшествует один знак тире. Параметры, состоящие из полных слов, являются более описательными и обозначаются двойными тире. Некоторые параметры имеют и однобуквенную версию, и версию из полных слов, а другие имеют только один тип. В табл. 3.5 перечислена часть наиболее широко применяемых параметров, с помощью которых можно получить требуемые данные, работая с командой `ls` командного интерпретатора `bash`.

Таблица 3.5. Некоторые широко применяемые параметры команды `ls`

Однобуквенный	Состоящий из полных слов	Описание
-a	--all	Не пропускать записи, начинающиеся с точки
-A	--almost-all	Не выводить в листинг файлы, обозначенные одной и двумя точками (. и ..)
	--author	Выводить имя автора каждого файла
-b	--escape	Выводить восьмеричные значения вместо непечатаемых символов
	--block-size=size	Вычислять размеры блоков с использованием байта как единицы измерения размера блока
-B	--ignore-backups	Не включать в листинг записи с символом тильды (~), которые используются для обозначения резервных копий
-c		Сортировать по времени последнего изменения
-C		Формировать перечень записей по столбцам
	--color=when	Определить условие использования цвета, when (всегда использовать (always), никогда не использовать, (never) или форматировать автоматически (auto))
-d	--directory	Выводить в листинг записи с обозначениями каталогов вместо их содержимого и не разадресовывать символические ссылки
-F	--classify	Присоединять к записям обозначения типов файлов
	--file-type	Присоединять обозначения типов файлов к записям, относящимся только к некоторым типам файлов (но не к исполняемым файлам)
	--format=word	Определить формат вывода как обозначенный одним из ключевых слов, word: поперечный (across), с разделением запятыми (commas), горизонтальный (horizontal), длинный (long), в один столбец (single-column), подробный (verbose) или вертикальный (vertical)
-g		Вывести в листинг полную информацию о файле за исключением владельца файла

Однобуквенный	Состоящий из полных слов	Описание
	<code>--group-directories-first</code>	Вывести в листинг все каталоги и лишь затем файлы
<code>-G</code>	<code>--no-group</code>	Не отображать в длинном листинге имена групп
<code>-h</code>	<code>--human-readable</code>	Выводить данные о размерах с использованием обозначения К для килобайтов (1024 байта), М для мегабайтов и Г для гигабайтов
	<code>--si</code>	То же, что и <code>-h</code> , но использовать для вычисления кратных единиц измерения коэффициент 1000 вместо 1024
<code>-i</code>	<code>--inode</code>	Отображать индексный номер (так называемый индексный узел) для каждого файла
<code>-l</code>		Сформировать листинг в длинном формате
<code>-L</code>	<code>--dereference</code>	Отображать информацию об исходном файле для связанного файла
<code>-n</code>	<code>--numeric-uid-gid</code>	Отображать вместо имен числовые идентификаторы пользователя и группы
<code>-o</code>		Не отображать в длинном листинге имена владельцев
<code>-r</code>	<code>--reverse</code>	Применять обратный порядок сортировки при отображении файлов и каталогов
<code>-R</code>	<code>--recursive</code>	Рекурсивно выводить содержимое подкаталогов
<code>-s</code>	<code>--size</code>	Выводить размер каждого файла в блоках
<code>-S</code>	<code>--sort=size</code>	Сортировать вывод по размерам файлов
<code>-t</code>	<code>--sort=time</code>	Сортировать вывод по времени модификации файлов
<code>-u</code>		Отображать время последнего доступа к файлу вместо времени последней модификации
<code>-U</code>	<code>--sort=none</code>	Не сортировать выходной листинг
<code>-v</code>	<code>--sort=version</code>	Сортировать вывод по версиям файлов
<code>-x</code>		Выводить записи по строкам, а не по столбцам
<code>-X</code>	<code>--sort=extension</code>	Сортировать вывод по расширениям файлов

По желанию можно одновременно задавать несколько параметров. Параметры с двойными тире должны быть перечислены отдельно, а параметры с одинарным тире могут быть объединены в одну строку, которая следует за одним тире. Часто применяется сочетание параметров, в котором параметр `-a` служит для перечисления всех файлов, а параметр `-i` — для указания *индексного узла* (inode) каждого файла, а также сочетание параметра `-l`, с помощью которого формируется длинный листинг, и параметра `-s`, позволяющего узнать размер каждого файла в блоках. Индексный узел файла или каталога представляет собой уникальный идентификационный номер, присваиваемый ядром каждому объекту в файловой системе. Объединение всех этих параметров позволяет получить один из удобных форматов листинга с помощью сочетания, которое принято задавать как `-sail` (это — легко запоминающееся обозначение, которое переводится как “парус”):

```
$ ls -sail
total 2360
301860      8 drwx----- 36 rich rich    4096 2010-09-03 15:12 .
 65473      8 drwxr-xr-x   6 root root    4096 2010-07-29 14:20 ..
```

```

360621      8 drwxrwxr-x  2 rich rich    4096 2010-08-24 22:04 4rich
301862      8 -rw-r--r--  1 rich rich      124 2010-02-12 10:18 .bashrc
361443      8 drwxrwxr-x  4 rich rich    4096 2010-07-26 20:31 .ccache
301879      8 drwxr-xr-x  3 rich rich    4096 2010-07-26 18:25 .config
301871      8 drwxr-xr-x  3 rich rich    4096 2010-08-31 22:24 Desktop
301870      8 -rw-----  1 rich rich       26 2009-11-01 04:06 .dmrc
301872      8 drwxr-xr-x  2 rich rich    4096 2009-11-01 04:06 Download
360207      8 drwxrwxr-x  2 rich rich    4096 2010-07-26 18:25 Drivers
301882      8 drwx-----  5 rich rich    4096 2010-09-02 23:40 .gconf
301883      8 drwx-----  2 rich rich    4096 2010-09-02 23:43 .gconfd
360338      8 drwx-----  3 rich rich    4096 2010-08-06 23:06 .gftp

```

В данном случае в дополнение к обычной информации, выводимой при использовании параметра `-l`, в каждой строке добавляются два числа. Первое число в этом листинге представляет собой номер индексного узла файла или каталога. Второе число — это размер файла в блоках. Третий элемент является символическим обозначением типа файла наряду с разрешениями файла. (Данная тема рассматривается более подробно в главе 6.)

За этим следуют такие данные: количество жестких ссылок на файл (эта тема обсуждается ниже, в разделе “Формирование ссылок на файлы”), владелец файла, группа, к которой принадлежит файл, размер файла (в байтах), отметка времени, по умолчанию показывающая время последней модификации, и наконец, фактическое имя файла.

Фильтрация вывода листинга

Как показывают приведенные примеры, по умолчанию команда `ls` перечисляет все файлы в каталоге. Полученный при этом объем информации иногда оказывается слишком велик, особенно если задача состоит в получении сведений лишь об одном файле.

К счастью, команда `ls` предоставляет также возможность определить фильтр в командной строке. В этой команде фильтр используется для определения того, какие файлы или каталоги должны быть отображены в выводе.

Фильтр работает как простая строка сопоставления с текстом. Для этого достаточно включить фильтр после любых параметров командной строки, к которым он должен быть применен:

```

$ ls -l myprog
-rwxr--r-- 1 rich rich 30 2007-08-23 21:42 myprog
$

```

Если в качестве фильтра указано имя конкретного файла, то команда `ls` отображает информацию только об этом файле. Иногда может оказаться неизвестным точное имя искомого файла. Команда `ls` распознает также следующие стандартные символы-шаблоны и использует их для сопоставления с шаблонами в фильтре:

- вопросительный знак (?) представляет один символ;
- звездочка (*) представляет от нуля и более символов.

Вопросительный знак может использоваться для замены одного и только одного символа в любой позиции в строке фильтра. Например:

```

$ ls -l mypro?
-rw-rw-r-- 1 rich rich  0 2010-09-03 16:38 myprob
-rwxr--r-- 1 rich rich 30 2010-08-23 21:42 myprog
$

```

Фильтр `myprob?` был сопоставлен с двумя файлами в каталоге. Аналогичным образом для сопоставления с нулем и большим количеством символов может использоваться звездочка:

```
$ ls -l myprob*
-rw-rw-r-- 1 rich rich 0 2010-09-03 16:38 myprob
-rw-rw-r-- 1 rich rich 0 2010-09-03 16:40 myproblem
$
```

В данном случае звездочка сопоставляется с нулем символов в файле `myprob` и с тремя символами в файле `myproblem`.

Фильтры представляют собой мощное средство поиска файлов, применимое в тех случаях, когда точно не известны имена файлов.

Обработка файлов

Командный интерпретатор `bash` предоставляет большое количество команд манипулирования файлами в файловой системе `Linux`. В настоящем разделе приведено краткое описание основных команд, которые применяются для работы с файлами из интерфейса командной строки во всех ситуациях, когда требуется осуществлять обработку файлов.

Создание файлов

Время от времени возникают ситуации, в которых требуется создать пустой файл. Например, иногда функционирование приложений основано на том, что в системе уже имеется файл журнала, в котором они могли бы выполнять операции записи. В подобных случаях при отсутствии требуемого файла можно использовать команду `touch`, позволяющую легко создать пустой файл:

```
$ touch test1
$ ls -il test1
1954793 -rw-r--r-- 1 rich rich 0 Sep 1 09:35 test1
$
```

Команда `touch` создает новый файл с указанным именем, а в качестве владельца файла назначает имя текущего пользователя. В рассматриваемой команде `ls` использовался параметр `-il`, поэтому первая запись в листинге показывает номер индексного узла, присвоенный файлу. Каждый файл в файловой системе `Linux` имеет уникальный номер индексного узла.

Обратите внимание на то, что этот файл имеет размер, равный нулю, поскольку команда `touch` создает только пустые файлы. Команда `touch` может также использоваться для изменения значений времени доступа и модификации для существующего файла без изменения содержимого этого файла:

```
$ touch test1
$ ls -l test1
-rw-r--r-- 1 rich rich 0 Sep 1 09:37 test1
$
```

Теперь время модификации файла `test1` обновлено и отличается от исходного значения времени. Если требуется изменить только время доступа, то можно воспользоваться параметром `-a`. Для изменения только времени модификации служит параметр `-m`. По умолчанию в команде `touch` используется текущее время. Но можно указать другое значение времени, используя параметр `-t` с конкретной отметкой времени:

```
$ touch -t 201112251200 test1
$ ls -l test1
-rw-r--r--  1 rich      rich      0 Dec 25  2011 test1
$
```

Теперь время модификации для рассматриваемого файла установлено на будущее время, которое значительно опережает текущее значение времени.

Копирование файлов

Системным администраторам в своей работе приходится также повседневно заниматься копированием файлов и каталогов из одного местоположения в файловой системе в другое. Такая возможность предоставляется командой `cp`.

В наиболее простой форме в команде `cp` используются два параметра, обозначающие исходный и целевой объекты:

```
cp source destination
```

Если оба параметра, *source* и *destination*, представляют собой имена файлов, то команда `cp` копирует исходный файл, создавая новый файл с именем файла, указанного в качестве целевого. Новый файл создается с нуля и имеет обновленные значения времени создания и модификации файла:

```
$ cp test1 test2
$ ls -il
total 0
1954793 -rw-r--r--  1 rich      rich      0 Dec 25  2011 test1
1954794 -rw-r--r--  1 rich      rich      0 Sep  1 09:39 test2
$
```

Для нового файла `test2` показан другой номер индексного узла, а это свидетельствует о том, что полученный файл является полностью новым. Можно также заметить, что в качестве времени модификации для файла `test2` показано время его создания. Если целевой файл уже существует, то команда `cp` выводит приглашение с вопросом, должна ли быть выполнена перезапись этого файла:

```
$ cp test1 test2
cp: overwrite 'test2'? y
$
```

Если ответ отличается от `y`, операция копирования файла отменяется. Можно также скопировать файл в один из существующих каталогов:

```
$ cp test1 dir1
$ ls -il dir1
total 0
1954887 -rw-r--r--  1 rich      rich      0 Sep  6 09:42 test1
$
```

Новый файл теперь располагается в каталоге `dir1` и имеет то же имя, что и исходный. Во всех этих примерах использовались относительные имена путей, но с тем же успехом для обозначения исходного и целевого объектов можно применять абсолютные имена путей.

Чтобы скопировать файл в текущий каталог, в котором ведется работа, можно использовать символ точки:

```
$ cp /home/rich/dir1/test1 .
cp: overwrite './test1'?
```

Как и большинство других команд, команда `cp` позволяет воспользоваться многочисленными параметрами командной строки, которые предназначены для решения многих разных задач. Эти параметры приведены в табл. 3.6.

Таблица 3.6. Параметры команды `cp`

Параметр	Описание
<code>-a</code>	Запись файлов в архив с сохранением их атрибутов
<code>-b</code>	Создание резервной копии каждого существующего целевого файла вместо его перезаписи
<code>-d</code>	Сохранение значений атрибутов
<code>-f</code>	Принудительная перезапись существующих целевых файлов без вывода приглашения для подтверждения
<code>-i</code>	Вывод приглашения перед перезаписью целевых файлов
<code>-l</code>	Создание ссылок на файлы вместо копирования файлов
<code>-p</code>	Сохранение атрибутов файлов, если это возможно
<code>-r</code>	Рекурсивное копирование файлов
<code>-R</code>	Рекурсивное копирование каталогов
<code>-s</code>	Создание символической ссылки вместо копирования файла
<code>-S</code>	Переопределение средства резервного копирования
<code>-u</code>	Копирование исходного файла, только если он имеет более новые значения даты и времени по сравнению с целевым файлом (проведение обновления)
<code>-v</code>	Применение режима подробного вывода с объяснением всего происходящего
<code>-x</code>	Ограничение сферы действия операции копирования текущей файловой системой

Параметр `-p` используется для сохранения таких же значений времени доступа к файлу или модификации файла, как в исходном файле, для скопированного файла.

```
$ cp -p test1 test3
$ ls -il
total 4
1954886 drwxr-xr-x  2 rich    rich      4096 Sep  1 09:42 dir1/
1954793 -rw-r--r--  1 rich    rich        0 Dec 25  2011 test1
1954794 -rw-r--r--  1 rich    rich        0 Sep  1 09:39 test2
1954888 -rw-r--r--  1 rich    rich        0 Dec 25  2011 test3
$
```

Итак, даже притом, что файл `test3` является полностью новым, он имеет те же отметки времени, что и исходный файл `test1`.

Параметр `-R` предоставляет весьма значительные возможности. Он позволяет рекурсивно копировать содержимое всего каталога с помощью одной команды:

```
$ cp -R dir1 dir2
$ ls -l
total 8
drwxr-xr-x  2 rich    rich      4096 Sep  6 09:42 dir1/
drwxr-xr-x  2 rich    rich      4096 Sep  6 09:45 dir2/
-rw-r--r--  1 rich    rich        0 Dec 25  2011 test1
```

```
-rw-r--r-- 1 rich rich 0 Sep 6 09:39 test2
-rw-r--r-- 1 rich rich 0 Dec 25 2011 test3
$
```

Теперь каталог `dir2` является полной копией каталога `dir1`. В применяемых командах `ср` также можно использовать символы-шаблоны:

```
$ cp -f test* dir2
$ ls -al dir2
total 12
drwxr-xr-x 2 rich rich 4096 Sep 6 10:55 ./
drwxr-xr-x 4 rich rich 4096 Sep 6 10:46 ../
-rw-r--r-- 1 rich rich 0 Dec 25 2011 test1
-rw-r--r-- 1 rich rich 0 Sep 6 10:55 test2
-rw-r--r-- 1 rich rich 0 Dec 25 2011 test3
$
```

Эта команда копирует все файлы с именами, начинающимися со строки `test`, в каталог `dir2`. Был включен также параметр `-f` для принудительной перезаписи файла `test1`, который уже находился в каталоге, без вывода приглашения для подтверждения.

Формирование ссылок на файлы

При изучении параметров команды `ср` можно заметить, что определенная часть параметров предназначена для формирования ссылок на файлы. Ссылки на файлы представляют собой одну из привлекательных особенностей файловых систем Linux. Если возникает необходимость поддерживать в системе две (или большее количество) копии одного и того же файла, то вместо создания отдельных физических копий можно использовать одну физическую копию и сформировать для нее несколько виртуальных копий, называемых *ссылками*. Ссылка — это местозаполнитель в каталоге, который указывает, где в действительности находится файл. В Linux применяются ссылки на файлы двух различных типов:

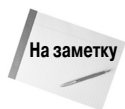
- гибкие, или символические, ссылки;
- жесткие ссылки.

При формировании жесткой ссылки создается отдельный файл, содержащий информацию об исходном файле и его местонахождении. Обращение к файлу жесткой ссылки полностью аналогично обращению к исходному файлу:

```
$ cp -l test1 test4
$ ls -il
total 16
1954886 drwxr-xr-x 2 rich rich 4096 Sep 1 09:42 dir1/
1954889 drwxr-xr-x 2 rich rich 4096 Sep 1 09:45 dir2/
1954793 -rw-r--r-- 2 rich rich 0 Sep 1 09:51 test1
1954794 -rw-r--r-- 1 rich rich 0 Sep 1 09:39 test2
1954888 -rw-r--r-- 1 rich rich 0 Dec 25 2011 test3
1954793 -rw-r--r-- 2 rich rich 0 Sep 1 09:51 test4
$
```

С помощью параметра `-l` создана жесткая ссылка `test4` для файла `test1`. В листинге файлов обнаруживается, что номера индексных узлов файлов `test1` и `test4` являются одинаковыми, а это указывает на то, что в действительности это один индексный узел, относящийся

к одному и тому же файлу. Следует также отметить, что количество ссылок (третий элемент в листинге) теперь показывает, что оба файла имеют по две ссылки.



Предусмотрена возможность создавать жесткие ссылки только между файлами, находящимися на одном и том же физическом носителе. Не может быть создана жесткая ссылка между файлами, расположенными под отдельными точками монтирования. В этом случае приходится использовать символические ссылки.

С другой стороны, параметр `-s` обеспечивает создание гибких, или символических, ссылок:

```
$ cp -s test1 test5
$ ls -il test*
total 16
1954793 -rw-r--r--  2 rich    rich    6 Sep  1 09:51 test1
1954794 -rw-r--r--  1 rich    rich    0 Sep  1 09:39 test2
1954888 -rw-r--r--  1 rich    rich    0 Dec 25 2011 test3
1954793 -rw-r--r--  2 rich    rich    6 Sep  1 09:51 test4
1954891 lrwxrwxrwx  1 rich    rich    5 Sep  1 09:56 test5
$ -> test1
$
```

В этом листинге файлов многое заслуживает особого внимания. Во-первых, можно отметить, что новый файл `test5` имеет другой номер индексного узла по сравнению с файлом `test1`, а это означает, что данные два файла рассматриваются в системе Linux как отдельные файлы. Во-вторых, новый файл имеет меньший размер, чем исходный. Связанный файл должен хранить лишь информацию об исходном файле, но не фактические данные этого файла. В области имен файлов листинга показана связь между этими двумя файлами.



Если возникает необходимость сформировать ссылки между файлами, то вместо команды `cp` можно также использовать команду `ln`. По умолчанию команда `ln` создает жесткие ссылки. Если же потребуется создать символическую ссылку, то все равно придется использовать параметр `-s`.

При копировании связанных файлов необходимо соблюдать осторожность. Если команда `cp` используется для копирования файла, связанного ссылкой с другим исходным файлом, то фактически эта операция копирования приводит к созданию еще одной копии исходного файла. Непродуманное использование подобных операций быстро приводит к путанице. Вместо копирования связанного файла можно создать просто еще одну ссылку на исходный файл. Предусмотрена возможность формирования многочисленных ссылок на один и тот же файл без каких-либо проблем. Однако не следует создавать символические ссылки на другие файлы, связанные символическими ссылками. Это приводит к созданию цепочек ссылок, что может не только стать причиной путаницы, но и легко привести к разрыву связей, порождая всевозможные проблемы.

Переименование файлов

В мире Linux операция переименования файлов рассматривается как *перемещение*. Для перемещения файлов и каталогов из одного местоположения в другое применяется команда `mv` (сокращение от `move`):

```
$ mv test2 test6
$ ls -il test*
1954793 -rw-r--r--  2 rich    rich    6 Sep  1 09:51 test1
```

```

1954888 -rw-r--r-- 1 rich rich 0 Dec 25 2011 test3
1954793 -rw-r--r-- 2 rich rich 6 Sep 1 09:51 test4
1954891 lrwxrwxrwx 1 rich rich 5 Sep 1 09:56 test5
$ -> test1
1954794 -rw-r--r-- 1 rich rich 0 Sep 1 09:39 test6
$

```

Заслуживает внимания то, что в данном случае перемещение файла привело к изменению имени файла, а номер индексного узла и значение отметки времени остались теми же. Перемещение файла с символическими ссылками становится причиной определенной проблемы:

```

$ mv test1 test8
$ ls -il test*
total 16
1954888 -rw-r--r-- 1 rich rich 0 Dec 25 2011 test3
1954793 -rw-r--r-- 2 rich rich 6 Sep 1 09:51 test4
1954891 lrwxrwxrwx 1 rich rich 5 Sep 1 09:56 test5 -> test1
1954794 -rw-r--r-- 1 rich rich 0 Sep 1 09:39 test6
1954793 -rw-r--r-- 2 rich rich 6 Sep 1 09:51 test8
[rich@test2 clsc]$ mv test8 test1

```

Для файла `test4`, в котором используется жесткая ссылка, по-прежнему сохраняется существующий номер индексного узла, что вполне соответствует ожиданиям. Однако файл `test5` теперь указывает на неправильно заданный файл и больше не является допустимой ссылкой.

Команду `mv` можно также использовать для перемещения каталогов:

```
$ mv dir2 dir4
```

Все содержимое каталога остается неизменным. Единственным атрибутом каталога, который изменяется, является имя каталога. Таким образом, команда `mv` действует намного быстрее по сравнению с командой `cp`.

Удаление файлов

В ходе эксплуатации системы Linux в какой-то момент неизбежно возникает необходимость удалять существующие файлы. Это может быть связано с потребностью очистки файловой системы или деинсталлирования пакета программ, но при этом так или иначе проводится уничтожение файлов.

В мире Linux уничтожение файлов принято называть *удалением*. В командном интерпретаторе `bash` для удаления файлов служит команда `rm` (сокращение от `remove`). В своей основной форме команда `rm` довольно проста:

```

$ rm -i test2
rm: remove 'test2'? y
$ ls -l
total 16
drwxr-xr-x 2 rich rich 4096 Sep 1 09:42 dir1/
drwxr-xr-x 2 rich rich 4096 Sep 1 09:45 dir2/
-rw-r--r-- 2 rich rich 6 Sep 1 09:51 test1
-rw-r--r-- 1 rich rich 0 Dec 25 2011 test3
-rw-r--r-- 2 rich rich 6 Sep 1 09:51 test4
lrwxrwxrwx 1 rich rich 5 Sep 1 09:56 test5 -> test1
$

```


Заслуживают внимания приглашения, которые выводит эта команда, позволяющие подтвердить, что файл действительно должен быть удален. В командном интерпретаторе `bash` не предусмотрена “мусорная корзина”, или область восстановления. Файл после его удаления исчезает навсегда.

В связи с удалением файлов, на который имеются ссылки, возникают интересные нюансы:

```
$ rm test1
$ ls -l
total 12
drwxr-xr-x  2 rich    rich    4096 Sep  1 09:42 dir1/
drwxr-xr-x  2 rich    rich    4096 Sep  1 09:45 dir2/
-rw-r--r--  1 rich    rich      0 Dec 25  2011 test3
-rw-r--r--  1 rich    rich     6 Sep  1 09:51 test4
lrwxrwxrwx  1 rich    rich     5 Sep  1 09:56 test5 -> test1
$ cat test4
hello
$ cat test5
cat: test5: No such file or directory
$
```

Произошло удаление файла `test1`, который имел жесткую ссылку на файл `test4` и символическую ссылку на файл `test5`. Рассмотрим, что произошло в связи с этим. Оба файла, обозначенные ссылками, все еще присутствуют в листинге, даже несмотря на то, что файл `test1` уже исчез (хотя на цветном терминале имя файла `test5` теперь выделено красным цветом). Просмотр содержимого файла `test4`, который представлял собой жесткую ссылку, показывает, что это содержимое файла еще присутствует в системе. А при попытке просмотра содержимого файла `test5`, который представлял собой символическую ссылку, командный интерпретатор `bash` указывает, что этот файл больше не существует.

Напомним, что для файла жесткой ссылки используется тот же номер индексного узла, что и для исходного файла. Файл жесткой ссылки сохраняет этот номер индексного узла до тех пор, пока не произойдет удаление последнего файла, связанного с ним жесткой ссылкой, а это означает, что данные по-прежнему сохраняются! Файл символической ссылки позволяет лишь определить, что исходный файл, применявшийся при его создании, больше не существует, поэтому нет и объекта, на который он в свое время указывал. Об этой важной особенности ссылок всегда следует помнить, работая со связанными файлами.

Еще одним средством команды `rm`, которое может применяться при удалении большого количества файлов, позволяющим избежать необходимости отвечать на многочисленные приглашения с подтверждениями, является использование параметра `-f` для принудительного удаления. Но применение этого параметра требует удвоенного внимания!



Как и при копировании файлов, при использовании команды `rm` можно задавать символы-шаблоны. И снова призываем вас проявлять осмотрительность, поскольку любая операция удаления, даже если она приводит к таким результатам, на которые вы не рассчитывали, влечет за собой безвозвратное уничтожение файлов!

Обработка каталогов

В Linux значительное количество команд может применяться для работы и с файлами, и с каталогами (в качестве примера можно назвать команду `cp`), а некоторые команды применимы только для каталогов. Для создания нового каталога необходимо воспользоваться

конкретной командой, которая рассматривается в этом разделе. Определенные нюансы возникают и при удалении каталогов, поэтому соответствующая команда также рассматривается в настоящем разделе.

Создание каталогов

Задача создания нового каталога в Linux — не слишком сложная; достаточно лишь выполнить команду `mkdir`:

```
$ mkdir dir3
$ ls -il
total 16
1954886 drwxr-xr-x  2 rich  rich  4096 Sep  1 09:42 dir1/
1954889 drwxr-xr-x  2 rich  rich  4096 Sep  1 10:55 dir2/
1954893 drwxr-xr-x  2 rich  rich  4096 Sep  1 11:01 dir3/
1954888 -rw-r--r--   1 rich  rich      0 Dec 25 2011 test3
1954793 -rw-r--r--   1 rich  rich     6 Sep  1 09:51 test4
$
```

Система создает новый каталог и присваивает ему новый номер индексного узла.

Удаление каталогов

Задача удаления каталогов может оказаться сложной, и на это есть определенные причины. Приступая к удалению каталогов, следует помнить, что при этом обнаруживаются многие предпосылки появления различных неприятностей. Безусловно, при этом командный интерпретатор `bash` в максимально возможной степени пытается предотвратить непредвиденные катастрофы. Основной командой удаления каталогов является `rmdir`:

```
$ rmdir dir3
$ rmdir dir1
rmdir: dir1: Directory not empty
$
```

По умолчанию действие команды `rmdir` заключается в том, что она удаляет лишь пустые каталоги. Поскольку в каталоге `dir1` имеется файл, команда `rmdir` отказывается удалить этот каталог. Разумеется, можно обеспечить удаление непустых каталогов, для чего предназначен параметр `--ignore-fail-on-non-empty`.

Одним из союзников пользователя, который может помочь при обработке каталогов, является команда `rm`.

При попытке применить эту команду без параметров, как для работы с файлами, можно испытать определенное разочарование:

```
$ rm dir1
rm: dir1: is a directory
$
```

Однако, если действительно требуется удалить каталог, можно воспользоваться параметром `-r` для рекурсивного удаления файлов в каталоге, а затем самого каталога:

```
$ rm -r dir2
rm: descend into directory 'dir2'? y
rm: remove 'dir2/test1'? y
rm: remove 'dir2/test3'? y
```

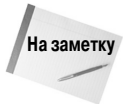
```
rm: remove 'dir2/test4'? y
rm: remove directory 'dir2'? y
$
```

Такой вариант вызова команды, безусловно, является применимым, но не совсем удобным. Обратите внимание на то, что все еще остается необходимость проверять правильность удаления каждого файла. Если в каталоге имеется большое количество файлов и подкаталогов, это может потребовать чрезмерных затрат сил.

Окончательным решением может стать то, чтобы отбросить осторожность и задать команду удаления всего каталога, его подкаталогов и всего содержимого, каковой является команда `rm` с обоими параметрами, `-r` и `-f`:

```
$ rm -rf dir2
$
```

На этом существование каталога заканчивается. Никаких предупреждений, никаких восклицательных знаков; после завершения операции лишь в очередной раз появляется приглашение командного интерпретатора. Безусловно, это — инструмент, чрезвычайно опасный в использовании, особенно в том случае, если работа ведется в учетной записи пользователя `root`. Используйте данную команду удаления как можно реже, и только после троекратной проверки того, что намечено уничтожение именно тех файлов и каталогов, которые должны быть удалены.



В последнем примере заслуживает внимания то, что два указанных параметра командной строки объединены с использованием одного тире. Это — одна из возможностей командного интерпретатора `bash`, который позволяет соединять однобуквенные параметры командной строки в целях сокращения объема ввода.

Просмотр содержимого файла

Выше был приведен большой объем сведений о том, как работать с файлами, но не были описаны команды просмотра содержимого файлов. Предусмотрено несколько команд, позволяющих заглянуть в файлы, не открывая их в редакторе (см. главу 11). В настоящем разделе рассматриваются некоторые имеющиеся команды, которые позволяют изучать файлы.

Просмотр статистических данных файла

В этой главе уже было показано, как используется команда `ls` для получения большого объема полезных сведений о файлах. Однако иногда все еще возникает необходимость получения дополнительной информации, которую не предоставляет команда `ls` (или, по крайней мере, не позволяет получить в одном вызове на выполнение).

Полный обзор статуса файла в файловой системе предоставляет команда `stat`:

```
$ stat test10
  File: "test10"
  Size: 6                Blocks: 8                Regular File
Device: 306h/774d       Inode: 1954891        Links: 2
Access: (0644/-rw-r--r--)  Uid: ( 501/ rich)  Gid: ( 501/ rich)
Access: Sat Sep  1 12:10:25 2010
Modify: Sat Sep  1 12:11:17 2010
Change: Sat Sep  1 12:16:42 2010
$
```

В результатах команды `stat` показано почти все, что может потребоваться узнать о рассматриваемом файле, вплоть до старшего и младшего номеров устройства, на котором хранится файл.

Просмотр типа файла

Несмотря на то что информация, предоставляемая командой `stat`, столь обширна, в ней все же отсутствуют сведения о типе файла. Это важно, поскольку, прежде чем отправить на устройство вывода файл объемом, допустим, в 1000 байтов, обычно следует выяснить, к какому типу относится этот файл. При попытке вывести двоичный файл на мониторе появится бессмысленный набор знаков и может даже произойти блокировка эмулятора терминала.

Поэтому следует всегда помнить о существовании команды `file`, которая вызывает небольшую удобную программу. Эта команда позволяет лишь заглянуть “внутрь” файла и определить, к какому типу он относится:

```
$ file test1
test1: ASCII text
$ file myscript
myscript: Bourne shell script text executable
$ file myprog
myprog: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), not stripped
$
```

Команда `file` классифицирует файлы на следующие три категории.

- **Текстовые файлы.** Это — файлы, содержащие символы, которые могут быть выведены на печать.
- **Исполняемые файлы.** Файлы, которые могут быть выполнены в системе.
- **Файлы данных.** Файлы, содержащие неподходящие для печати двоичные символы, которые не могут быть также вызваны на выполнение в системе.

В первом примере показан текстовый файл. Команда `file` позволила не только определить, что файл содержит текст, но и установить формат символического кода текста. Во втором примере показан текстовый файл сценария. Хотя этот файл является текстовым, он вместе с тем представляет собой файл сценария, поэтому может быть вызван на выполнение (или, проще говоря, выполнен) в системе. В последнем примере показана двоичная исполняемая программа. Команда `file` определяет, для какой платформы была откомпилирована программа и какие типы библиотек для нее требуются. Эта возможность становится особенно удобной, если приходится сталкиваться с двоичной исполняемой программой, полученной из неизвестного источника.

Просмотр всего файла

Иногда возникает необходимость просматривать все содержимое большого текстового файла. Для этого в Linux предусмотрены три команды.

Команда `cat`

Команда `cat` представляет собой удобный инструмент для отображения всего содержимого текстового файла:

```
$ cat test1
hello

This is a test file.

That we'll use to      test the cat command.
$
```

Безусловно, этот вывод не представляет собой что-то особенное; в нем находится лишь содержимое текстового файла. Как и при использовании многих других команд, при вызове команды `cat` можно задать некоторые параметры.

Параметр `-n` обеспечивает нумерацию всех выводимых строк:

```
$ cat -n test1
 1 hello
 2
 3 This is a test file.
 4
 5
 6 That we'll use to      test the cat command.
$
```

Такая возможность становится удобной, например, при просмотре сценариев. Если требуется пронумеровать только строки, в которых имеется текст, то можно воспользоваться параметром `-b`:

```
$ cat -b test1
 1 hello

 2 This is a test file.

 3 That we'll use to      test the cat command.
$
```

Для сжатия нескольких пустых строк в одну пустую строку служит параметр `-s`:

```
$ cat -s test1
hello

This is a test file.

That we'll use to      test the cat command.
$
```

Наконец, если нежелательно появление в выводе символов табуляции, можно воспользоваться параметром `-T`:

```
$ cat -T test1
hello

This is a test file.

That we'll use to^^Itest the cat command.
$
```

Параметр `-T` заменяет все знаки табуляции в тексте комбинацией символов `^^I`.

Но команда `cat` при попытке ее применения для просмотра больших файлов вызывает лишь раздражение. Текст обрабатываемого ею файла быстро пробегает по экрану монитора без остановки. К счастью, предусмотрен простой способ решения этой проблемы.

Команда more

Основной недостаток команды `cat` состоит в том, что после ее запуска пользователь теряет контроль над происходящим. Для решения этой проблемы разработчики создали команду `more`. Команда `more` также отображает текстовые файлы, но останавливается после вывода на экран каждой страницы данных. Типичный пример экрана при вызове команды `more` показан на рис. 3.4.

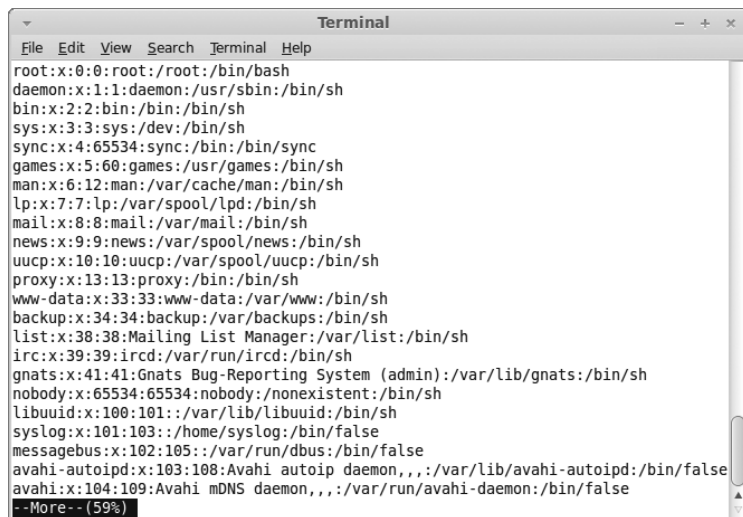


Рис. 3.4. Использование команды `more` для отображения текстового файла

Обратите внимание на то, что в нижней части экрана на рис. 3.4 команда `more` выводит приглашение, показывающее, что продолжается работа в программе `more`, и позволяющее узнать, какая часть текстового файла уже просмотрена. Это — приглашение команды `more`. Получив такое приглашение, можно ввести одну из нескольких подкоманд, приведенных в табл. 3.7.

Таблица 3.7. Подкоманды команды `more`

Подкоманда	Описание
<code>h</code>	Отображение справочного меню
клавиша пробела	Отображение следующего экрана текста из файла
<code>z</code>	Отображение следующего экрана текста из файла
<code>ENTER</code>	Отображение еще одной строки текста из файла
<code>d</code>	Отображение полуэкрана (11 строк) текста из файла
<code>q</code>	Выход из программы
<code>s</code>	Пропуск в прямом направлении одной строки текста
<code>f</code>	Пропуск в прямом направлении одного экрана текста
<code>b</code>	Пропуск в обратном направлении одного экрана текста
<code>/expression</code>	Поиск текста в файле с помощью выражения <code>expression</code>
<code>n</code>	Поиск следующего вхождения последнего указанного выражения

Подкоманда	Описание
'	Переход к первому вхождению указанного выражения
!cmd	Выполнение команды командного интерпретатора
v	Запуск редактора vi в текущей строке
CTRL-L	Перерисовка экрана в текущем местоположении в файле
=	Отображение номера текущей строки в файле
.	Повторение предыдущей команды

Команда `more` предоставляет лишь некоторые элементарные возможности перемещения по текстовому файлу. Если потребуются более сложные операции при просмотре файла, то можно воспользоваться командой `less`.

Команда less

Безусловно, по имени команды `less` (что в переводе означает “меньше”) трудно догадаться, что она обеспечивает большие возможности по сравнению с командой `more` (что означает “больше”); фактически имя команды `less` представляет собой игру слов, а сама она является более развитой версией команды `more` (имя команды `less` возникло от крылатых слов “less is more” — чем меньше, тем больше). Команда `less` позволяет воспользоваться некоторыми весьма удобными средствами прямой и обратной прокрутки текстового файла, а также предоставляет весьма развитые функции поиска.

Кроме того, команда `less` способна отображать содержимое файла по частям, прежде чем будет прочитан весь файл. Такой способностью не обладают ни команда `cat`, ни команда `more`, а это серьезный недостаток, особенно если задача состоит в просмотре чрезвычайно больших файлов.

Вообще говоря, команда `less` действует аналогично команде `more`, отображая по одному экрану текста из файла одновременно. На рис. 3.5 показана команда `less` в действии.

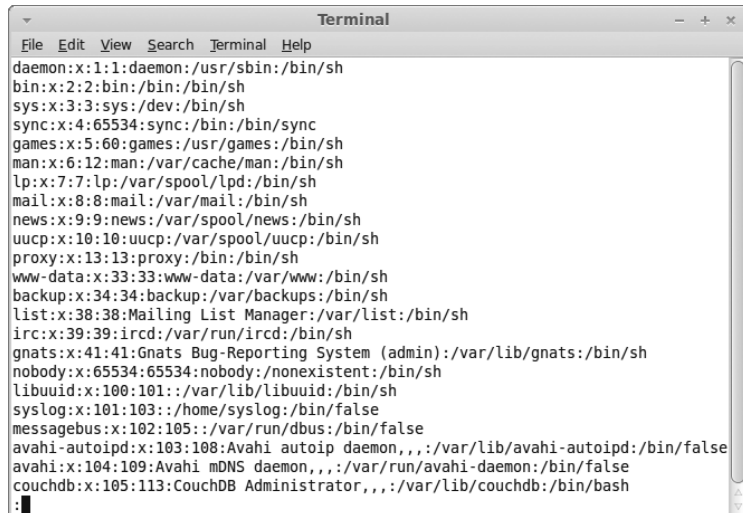


Рис. 3.5. Просмотр файла с использованием команды `less`

Заслуживает внимания тот факт, что команда `less` в своем приглашении предоставляет дополнительные сведения, показывая общее количество строк в файле и диапазон строк, отображаемых в настоящее время. Команда `less` поддерживает тот же набор подкоманд, что и команда `more`, но с весьма значительным дополнением. Для ознакомления со всеми доступными подкомандами команды `less` обратитесь к справочному руководству по этой команде. Расширению набора возможностей команды `less` способствует то, что она распознает нажатия клавиш со стрелками вверх и вниз, а также клавиш перелистывания страниц вверх и вниз (но для этого необходимо, чтобы используемый терминал был определен должным образом). Итак, эта команда предоставляет пользователю полный контроль над просмотром файла.

Просмотр частей файла

Часто возникает такая ситуация, что данные, с которыми необходимо ознакомиться, расположены либо в самом начале текстового файла, либо в самом его конце. Если даже требуется рассмотреть информацию лишь в начале большого файла, то при использовании команды `cat` или `more` все равно приходится ожидать окончания загрузки всего файла, прежде чем приступить к его изучению. Если же информация, подлежащая просмотру, располагается в самом конце файла (что часто бывает при изучении таких файлов, как файлы журналов), то приходится пропускать тысячи строк текста лишь для того, чтобы добраться до последних нескольких записей. К счастью, в системе Linux имеются специальные команды, позволяющие решить обе эти проблемы.

Команда `tail`

Команда `tail` отображает последнюю группу строк в файле. По умолчанию эта команда показывает последние 10 строк в файле, но эту установку можно изменить с помощью параметров командной строки, которые показаны в табл. 3.8.

Таблица 3.8. Параметры командной строки команды `tail`

Параметр	Описание
<code>-c bytes</code>	Отображать последние байты файла в количестве <code>bytes</code>
<code>-n lines</code>	Отображать последние строки файла в количестве <code>lines</code>
<code>-f</code>	Поддерживать программу <code>tail</code> в активном состоянии и продолжать отображение новых строк по мере их добавления к файлу
<code>--pid=PID</code>	При наличии параметра <code>-f</code> следить за появлением новых строк в файле до завершения процесса с идентификатором <code>PID</code>
<code>-s sec</code>	При наличии параметра <code>-f</code> переходить в состояние простоя между итерациями на время <code>sec</code> в секундах
<code>-v</code>	Всегда отображать выходные заголовки с указанием имени файла
<code>-q</code>	Никогда не отображать выходные заголовки с указанием имени файла

Параметр `-f` представляет собой весьма удобное средство команды `tail`. Он позволяет просматривать файл одновременно с тем, как происходит использование этого файла другими процессами. Команда `tail` остается активной и продолжает отображать все новые и новые строки по мере их появления в текстовом файле. В этом состоит один из лучших способов отслеживания системных файлов журналов в режиме реального времени.

Команда head

Не будучи столь необычной по своим функциям, как команда `tail` (хвост), команда `head` (голова) выполняет то, о чем можно судить по ее имени, — отображает первую группу строк в начале файла. По умолчанию эта команда отображает первые 10 строк текста. Аналогично команде `tail`, команда `head` поддерживает параметры `-с` и `-n`, с помощью которых можно откорректировать объем отображаемой информации.

Начало файла, как правило, не изменяется, поэтому команда `head` не поддерживает функции параметра `-f`. Команда `head` предоставляет удобный способ просмотра одного лишь начала файла, если не совсем ясно, что содержит файл, поскольку при ее использовании не приходится просматривать весь файл.

Резюме

В настоящей главе приведены основные сведения о работе в файловой системе Linux с помощью приглашения командного интерпретатора. Вначале было проведено общее обсуждение командного интерпретатора `bash` и показано, как проводится работа с командным интерпретатором. В интерфейсе командной строки (command line interface — CLI) используется строка приглашения для обозначения состояния готовности командного интерпретатора к приему введенных команд. Предусмотрена возможность настраивать строку приглашения, чтобы можно было видеть в ней полезную информацию о системе и применяемом идентификаторе входа в систему, и даже дату и время.

Командный интерпретатор `bash` предоставляет доступ к широкому набору программ, которые можно использовать для создания и управления файлами. Прежде чем приступить к работе с файлами, необходимо понять, как организовано хранение файлов в системе Linux. В настоящей главе приведены основные сведения о виртуальном каталоге Linux и показано, каким образом система Linux предоставляет доступ к поддерживаемым устройствам. После описания файловой системы Linux в этой главе показано, как использовать команду `cd` для перемещения по виртуальному каталогу.

После изложения сведений о том, как перейти в определенный каталог, в данной главе демонстрируется использование команды `ls` для формирования списков файлов и подкаталогов. Команда `ls` поддерживает многочисленные параметры, которые позволяют определять формат ее вывода. В частности, командой `ls` можно воспользоваться для получения значительной части необходимой информации о файлах и каталогах.

Полезным средством создания пустых файлов и изменения значений времени доступа или модификации существующих файлов является команда `touch`. В этой главе обсуждается также использование команды `cp` для копирования существующих файлов из одного местоположения в другое. Кроме того, описан процесс создания ссылок на файлы вместо их копирования, который предоставляет удобный способ размещения одного и того же файла в двух разных местоположениях без создания отдельной копии. Эту задачу позволяют решить команда `cp`, а также команда `ln`.

Далее приведены сведения о переименовании файлов (эту операцию в системе Linux принято называть *перемещением*) с использованием команды `mv` и показано, как проводится уничтожение файлов (эту операцию принято называть *удалением*) с помощью команды `rm`. Было также показано, как решать аналогичные задачи с каталогами, используя команды `mkdir` и `rmdir`.

Наконец, в настоящей главе описано, как просматривать содержимое файлов. Команды `cat`, `more` и `less` предоставляют простые способы просмотра всего содержимого файла, а команды `tail` и `head` превосходно подходят для ознакомления с небольшими частями файлов, конечными и начальными.

В следующей главе продолжается обсуждение команд командного интерпретатора `bash`. В ней будут рассматриваться более сложные команды, находящиеся в распоряжении системного администратора, без которых нельзя обойтись при решении задач управления системой Linux.

Дальнейшее описание команд интерпретатора bash

ГЛАВА

4

В этой главе...

Отслеживание работы программ

Контроль над использованием места на диске

Работа с файлами данных

Резюме

В главе 3 приведены основные сведения о том, как осуществляется переход по файловой системе Linux, и о работе с файлами и каталогами. Управление файлами и каталогами — важная функция командного интерпретатора Linux, однако ознакомления только с этой функцией недостаточно для работы по программированию сценариев. В данной главе приведены команды сопровождения системы Linux и показано, как приступить к изучению функционирования системы Linux с использованием команд командной строки. После этого показано несколько удобных команд, которые можно применять для работы с файлами данных в системе.

Отслеживание работы программ

Одна из наиболее трудоемких задач, стоящих перед системным администратором Linux, состоит в отслеживании процессов, действующих в системе. Эта задача еще более усложнилась в наши дни в связи с применением графических рабочих столов, для обеспечения функционирования

которых используется большое количество программ. В системе в любой момент времени работает много служебных программ.

К счастью, предусмотрено несколько утилит с интерфейсом командной строки, которые могут помочь специалисту в его нелегкой деятельности. В этом разделе рассматривается несколько основных инструментов, которые необходимо освоить, чтобы научиться управлять программами в системе Linux.

Контроль над функционированием процессов

Программу, исполняемую в системе, принято называть *процессом*. Для исследования этих процессов применяется команда `ps`, которая представляет собой буквально незаменимый универсальный инструмент. С помощью команды `ps` можно получить большой объем информации обо всех программах, функционирующих в системе.

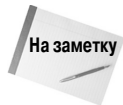
К сожалению, широкие возможности команды влекут за собой сложность ее использования (что выражается в наличии многочисленных параметров), поэтому команда `ps`, по-видимому, является одной из самых трудных для освоения. Большинство системных администраторов подбирает для себя определенное подмножество параметров этой команды, позволяющее получить необходимую информацию, а затем неизменно использует только эти параметры.

Но, несмотря на сказанное, в действительности команда `ps` в основной форме не предоставляет такой уж большой объем информации:

```
$ ps
  PID TTY          TIME CMD
 3081 pts/0    00:00:00 bash
 3209 pts/0    00:00:00 ps
$
```

Эти результаты нельзя назвать чрезвычайно интересными. По умолчанию команда `ps` показывает только процессы, которые принадлежат текущему пользователю и выполняются на текущем терминале. В данном случае приведены сведения только об эксплуатируемом командном интерпретаторе `bash` (напомним: командный интерпретатор — это всего лишь еще одна программа, функционирующая в системе), а также, разумеется, о самой команде `ps`.

Выходные данные команды в основной форме показывают идентификатор процесса (process ID — PID) программы, терминал (TTY), на котором она выполняется, и процессорное время, использованное процессом.



Одной из сложных особенностей команды `ps` (и одной из причин того, почему ее так сложно освоить) является существование в свое время двух версий этой команды. Каждая версия имела свой собственный набор параметров командной строки, которые применялись для управления тем, какая информация должна быть отображена и как это должно быть сделано. Недавно разработчики Linux объединили обе версии команды `ps` в отдельную программу `ps` (а также, безусловно, внесли результаты своей собственной работы над этой командой).

Команда `ps` в составе утилит GNU, которая используется в системах Linux, поддерживает три различных типа параметров командной строки, как описано ниже.

- Параметры в стиле Unix, которым предшествует тире.
- Параметры в стиле BSD, которым не предшествует тире.
- Длинные параметры GNU, которым предшествует двойное тире.

Эти три типа параметров рассматриваются в следующих разделах, там же приведены примеры того, как они работают.

Параметры в стиле Unix

Параметры в стиле Unix впервые были разработаны для первоначальной версии команды `ps`, предназначенной для систем AT&T Unix, которые были созданы в корпорации Bell Labs (табл. 4.1).

Таблица 4.1. Параметры команды `ps` в стиле Unix

Параметр	Описание
<code>-A</code>	Показывать все процессы
<code>-N</code>	Показывать такие данные, как если бы указанные параметры были заменены противоположными
<code>-a</code>	Показывать все процессы, кроме заголовков сеансов и процессов, выполняемые вне терминалов
<code>-d</code>	Показывать все процессы, кроме заголовков сеансов
<code>-e</code>	Показывать все процессы
<code>-C <i>cmslist</i></code>	Показывать процессы, содержащиеся в списке <i>cmslist</i>
<code>-G <i>grplist</i></code>	Показывать процессы с идентификаторами групп, перечисленными в списке <i>grplist</i>
<code>-U <i>userlist</i></code>	Показывать процессы, принадлежащие пользователям с идентификаторами, перечисленными в списке <i>userlist</i>
<code>-g <i>grplist</i></code>	Показывать процессы по сеансам или по идентификаторам групп, перечисленным в списке <i>grplist</i>
<code>-p <i>pidlist</i></code>	Показывать процессы с идентификаторами процессов, перечисленными в списке <i>pidlist</i>
<code>-s <i>sesslist</i></code>	Показывать процессы с идентификатором сеансов, перечисленными в списке <i>sesslist</i>
<code>-t <i>ttylist</i></code>	Показывать процессы с идентификаторами терминалов, перечисленными в списке <i>ttylist</i>
<code>-u <i>userlist</i></code>	Показывать процессы по действительным идентификаторам пользователей, перечисленным в списке <i>userlist</i>
<code>-F</code>	Использовать сверхполный вывод
<code>-O <i>format</i></code>	Отображать конкретные столбцы, перечисленные в списке <i>format</i> , наряду со столбцами, заданными по умолчанию
<code>-M</code>	Отображать информацию о безопасности, касающуюся процесса
<code>-c</code>	Показывать дополнительную информацию планировщика о процессе
<code>-f</code>	Отображать листинг в полном формате
<code>-j</code>	Показывать информацию о задании
<code>-l</code>	Отображать длинный листинг
<code>-o <i>format</i></code>	Отображать только конкретные столбцы, перечисленные в списке <i>format</i>
<code>-y</code>	Не показывать флаги процессов
<code>-Z</code>	Отображать информацию контекста безопасности
<code>-H</code>	Отображать процессы в иерархическом формате (показывать также родительские процессы)
<code>-n <i>namelist</i></code>	Определить значения, которые должны отображаться в столбце <code>WCHAN</code>
<code>-w</code>	Использовать широкий выходной формат, предназначенный для дисплеев с неограниченной шириной вывода

Параметр	Описание
-L	Показывать потоки процесса
-V	Отображать версию команды ps

Очевидно, что количество этих параметров весьма велико, но следует помнить, что на этом они не исчерпываются! Ключом к успешному использованию команды `ps` является не запоминание всех возможных параметров, а освоение тех, которые представляются наиболее полезными. Большинство системных администраторов Linux выбирают собственные наборы обычно применяемых параметров, которые они всегда могут вспомнить, приступая к получению требуемой информации. Например, если возникает необходимость просматривать все процессы, работающие в системе, можно воспользоваться комбинацией параметров `-ef` (команда `ps` позволяет применять сочетания однобуквенных параметров, как в данном примере):

```
$ ps -ef
UID          PID  PPID  C  STIME TTY          TIME CMD
root           1      0  0  11:29 ?        00:00:01 init [5]
root           2      0  0  11:29 ?        00:00:00 [kthreadd]
root           3      2  0  11:29 ?        00:00:00 [migration/0]
root           4      2  0  11:29 ?        00:00:00 [ksoftirqd/0]
root           5      2  0  11:29 ?        00:00:00 [watchdog/0]
root           6      2  0  11:29 ?        00:00:00 [events/0]
root           7      2  0  11:29 ?        00:00:00 [khelper]
root          47      2  0  11:29 ?        00:00:00 [kblockd/0]
root          48      2  0  11:29 ?        00:00:00 [kacpid]
68            2349      1  0  11:30 ?        00:00:00 hald
root          2489      1  0  11:30 tty1      00:00:00 /sbin/mingetty tty1
root          2490      1  0  11:30 tty2      00:00:00 /sbin/mingetty tty2
root          2491      1  0  11:30 tty3      00:00:00 /sbin/mingetty tty3
root          2492      1  0  11:30 tty4      00:00:00 /sbin/mingetty tty4
root          2493      1  0  11:30 tty5      00:00:00 /sbin/mingetty tty5
root          2494      1  0  11:30 tty6      00:00:00 /sbin/mingetty tty6
root          2956      1  0  11:42 ?        00:00:00 /usr/sbin/httpd
apache        2958    2956  0  11:42 ?        00:00:00 /usr/sbin/httpd
apache        2959    2956  0  11:42 ?        00:00:00 /usr/sbin/httpd
root          2995      1  0  11:43 ?        00:00:00 auditd
root          2997    2995  0  11:43 ?        00:00:00 /sbin/audispd
root          3078    1981  0  12:00 ?        00:00:00 sshd: rich [priv]
rich          3080    3078  0  12:00 ?        00:00:00 sshd: rich@pts/0
rich          3081    3080  0  12:00 pts/0    00:00:00 -bash
rich          4445    3081  3  13:48 pts/0    00:00:00 ps -ef
$
```

В целях экономии места в этом выводе удалена весьма значительная часть строк, но все равно эти данные показывают, насколько большое количество процессов применяется в системе Linux. В этом примере используются два параметра: параметр `-e`, который показывает все процессы, работающие в системе, и параметр `-f`, обеспечивающий вывод в широком формате, с включением нескольких полезных столбцов информации.

- **UID.** Пользователь, ответственный за запуск рассматриваемого процесса.

- PID. Идентификатор рассматриваемого процесса.
- PPID. Значение PID родительского процесса (если процесс запущен другим процессом).
- C. Загрузка процессора на протяжении времени существования процесса.
- STIME. Системное время запуска процесса.
- TTY. Терминальное устройство, с которого был запущен процесс.
- TIME. Совокупное значение процессорного времени, затраченного на выполнение процесса.
- CMD. Имя программы, которая была запущена.

В результате выводится довольно значительный объем информации, но многие из этих сведений как раз и представляют интерес для системных администраторов. Чтобы получить еще более подробную информацию, можно воспользоваться параметром `-l` (сокращение от `long`), который формирует вывод в длинном формате:

```
$ ps -l
F S  UID PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY      TIME    CMD
0 S  500 3081  3080  0  80   0 -  1173 wait pts/0    00:00:00 bash
0 R  500 4463  3081  1  80   0 -  1116 -   pts/0    00:00:00 ps
$
```

Обратите внимание на следующие дополнительные столбцы, которые появляются при использовании параметра `-l`.

- F. Флаги системы, присвоенные процессу ядром.
- S. Состояние процесса (O — выполняющий на процессоре; S — простаивающий; R — работоспособный, ждущий запуска; Z — зависший в связи с тем, что после завершения процесса не обнаружен его родительский объект; T — процесс остановлен).
- PRI. Приоритет процесса (чем выше числовое значение, тем ниже приоритет).
- NI. Величина изменения приоритета, которая используется для определения окончательных значений приоритетов.
- ADDR. Адрес процесса в памяти.
- SZ. Приблизительный объем области подкачки, который может потребоваться при загрузке процесса.
- WCHAN. Имя функции ядра, которая перевела процесс в состояние простоя.

Прежде чем продолжить описание, отметим, что есть еще один чрезвычайно удобный параметр, о котором всегда следует помнить, `-H`. Параметр `-H` позволяет организовать вывод данных о процессах в иерархическом формате, который показывает, какие процессы были начаты раньше других процессов. Ниже приведен пример листинга, сформированного при использовании параметров `-efH`.

```
$ ps -efH
UID      PID    PPID  C STIME TTY      TIME    CMD
root     3078    1981  0 12:00 ?        00:00:00 sshd: rich [priv]
rich     3080    3078  0 12:00 ?        00:00:00 sshd: rich@pts/0
rich     3081    3080  0 12:00 pts/0    00:00:00 -bash
rich     4803    3081  1 14:31 pts/0    00:00:00 ps -efH
```

Обратите внимание на сдвиг в выводе столбца `CMD`. Он показывает иерархию выполняемых процессов, складывающуюся так, что сначала происходит запуск процесса `sshd` пользо-

вателем root. Это серверный сеанс командного интерпретатора Secure Shell (SSH), в котором происходит *прослушивание* (так принято называть прием по сети) дистанционных запросов на установление соединений SSH. Таким образом, этот сеанс стал результатом подключения дистанционно расположенного терминала к системе, поэтому в основном процессе SSH произошел запуск процесса терминала (`pts/0`), который, в свою очередь, запустил командный интерпретатор `bash`.

Из командного интерпретатора была вызвана на выполнение команда `ps`, которая показана в виде процесса, дочернего по отношению к процессу `bash`. В многопользовательской системе команда `ps` представляет собой чрезвычайно полезный инструмент, который позволяет изучать причины появления процессов, вышедших из-под контроля, или предпринимать попытки определения того, какому идентификатору пользователя или терминалу они принадлежат.

Параметры в стиле BSD

Выше были описаны параметры в стиле Unix, а в настоящем разделе рассматриваются параметры в стиле BSD. Дистрибутив программного обеспечения Беркли (Berkeley Software Distribution — BSD) — это версия Unix, разработанная (что и можно было предположить) в Калифорнийском университете, г. Беркли. Он обладает многими тонкими отличиями от системы AT&T Unix, и само существование этих отличий стало причиной существенных разногласий в сообществе пользователей Unix, которые наблюдались в прошедшие годы. Параметры команды `ps` в стиле BSD приведены в табл. 4.2.

Таблица 4.2. Параметры команды `ps` в стиле BSD

Параметр	Описание
<code>T</code>	Показывать все процессы, ассоциированные с этим терминалом
<code>a</code>	Показывать все процессы, ассоциированные с любым терминалом
<code>g</code>	Показывать все процессы, включая заголовки сеансов
<code>r</code>	Показывать только выполняющиеся процессы
<code>x</code>	Показывать все процессы, даже те, для которых не назначено терминальное устройство
<code>U userlist</code>	Показывать процессы, принадлежащие пользователям с идентификаторами, перечисленными в списке <i>userlist</i>
<code>p pidlist</code>	Показывать процессы со значениями PID, перечисленными в списке <i>pidlist</i>
<code>t ttylist</code>	Показывать процессы, которые ассоциированы с терминалами, перечисленными в списке <i>ttylist</i>
<code>O format</code>	Показать список конкретных столбцов, перечисленных в списке <i>format</i> , которые должны отображаться наряду со стандартными столбцами
<code>X</code>	Отображать данные в формате регистра
<code>Z</code>	Включить в выходные данные информацию о безопасности
<code>j</code>	Показывать информацию о задании
<code>l</code>	Использовать длинный формат
<code>o format</code>	Отображать только столбцы, указанные в списке <i>format</i>
<code>s</code>	Использовать формат сигнала
<code>u</code>	Применять формат, ориентированный на пользователя
<code>v</code>	Использовать формат виртуальной памяти
<code>N namelist</code>	Определить значения, предназначенные для использования в столбце <code>WCAN</code>

Параметр	Описание
O order	Определить порядок отображения информационных столбцов
S	Суммировать относящуюся к дочерним процессам числовую информацию, такую как использование процессора и памяти, в привязке к родительскому процессу
c	Отображать истинное имя команды (имя программы, которая использовалась для запуска процесса)
e	Отображать все переменные среды, используемые командой
f	Отображать процессы в иерархическом формате, показывая, какие процессы были запущены теми или иными процессами
h	Не отображать информацию заголовка
k sort	Определить один или несколько столбцов для использования при сортировке выходных данных
n	Использовать числовые значения для идентификаторов пользователя и группы, наряду с информацией WCHAN
w	Формировать широкий вывод для более широких терминалов
H	Отображать потоки, как если бы они были процессами
m	Отображать потоки после относящихся к ним процессов
L	Показать список всех спецификаторов формата
V	Отображать версию ps

Вполне очевидно, что параметры типа Unix и BSD имеют много общего. Большая часть информации, предоставляемая параметрами одного типа, может быть также получена с помощью параметров другого типа. Чаще всего выбор типа параметров пользователем бывает основан на том, какой формат является для него более удобным (примером может служить ситуация, в которой пользователь перед переходом на систему Linux работал в среде BSD).

При использовании параметров в стиле BSD команда ps автоматически изменяет формат вывода в целях моделирования формата BSD. Ниже приведен пример использования параметра l.

```
$ ps l
F  UID  PID PPID PRI  NI  VSZ  RSS WCHAN  STAT TTY      TIME COMMAND
0  500  3081 3080  20   0 4692 1432 wait   Ss   pts/0    0:00 -bash
0  500  5104 3081  20   0 4468  844 -       R+   pts/0    0:00 ps l
$
```

Заслуживает внимания то, что многие выходные столбцы остаются такими же, как при использовании параметров в стиле Unix, однако есть также несколько отличающихся столбцов, как показано ниже.

- VSZ. Объем памяти в килобайтах, распределенный для процесса.
- RSS. Объем физической памяти, используемой процессом, который не был выгружен.
- STAT. Двухсимвольный код состояния, представляющий текущее состояние процесса.

Многие системные администраторы охотно используют параметр l в стиле BSD, поскольку он предоставляет более подробные сведения о коде состояния для процессов (столбец STAT). Этот двухсимвольный код более точно определяет, что именно происходит с процессом, чем односимвольный вывод в стиле Unix.

Для первого символа используются те же значения, что и для выходного столбца S в стиле Unix, которые показывают, находится ли процесс в состоянии простоя, выполнения или ожидания. Второй символ дополнительно определяет статус процесса следующим образом.

- <. Процесс выполняется с высоким приоритетом.
- N. Процесс выполняется с низким приоритетом.
- L. Для процесса заблокированы страницы в памяти.
- s. Процесс является заполнителем сеанса.
- l. Процесс является многопоточковым.
- +: Процесс выполняется на переднем плане.

Простой пример, приведенный выше, показывает, что процесс, запущенный командой `bash`, переходит в состояние простоя, но является заполнителем сеанса (основным процессом в данном сеансе), тогда как команда `ps` выполняется в системе на переднем плане.

Длинные параметры GNU

В конечном итоге разработчики программ GNU приложили собственные усилия к созданию новой, улучшенной команды `ps`, а также добавили еще несколько параметров к общей совокупности параметров. Часть длинных параметров GNU копирует существующие параметры в стиле BSD или Unix, тогда как другие предоставляют новые возможности. В табл. 4.3 перечислены длинные параметры GNU, которые имеются в распоряжении пользователя.

Таблица 4.3. Параметры команды `ps` в стиле GNU

Параметр	Описание
<code>--deselect</code>	Показывать все процессы, кроме перечисленных в командной строке
<code>--Group <i>grplist</i></code>	Показывать процессы, идентификаторы группы которых перечислены в списке <i>grplist</i>
<code>--User <i>userlist</i></code>	Показывать процессы, идентификаторы пользователя которых перечислены в списке <i>userlist</i>
<code>--group <i>grplist</i></code>	Показывать процессы, действительные идентификаторы группы которых перечислены в списке <i>grplist</i>
<code>--pid <i>pidlist</i></code>	Показывать процессы, идентификаторы процесса которых перечислены в списке <i>pidlist</i>
<code>--ppid <i>pidlist</i></code>	Показывать процессы, идентификаторы родительского процесса которых перечислены в списке <i>pidlist</i>
<code>--sid <i>sidlist</i></code>	Показывать процессы, идентификаторы сеанса которых перечислены в списке <i>sidlist</i>
<code>--tty <i>ttylist</i></code>	Показывать процессы, идентификатор терминального устройства которых перечислен в списке <i>ttylist</i>
<code>--user <i>userlist</i></code>	Показывать процессы, эффективный идентификатор пользователя которых перечислен в списке <i>userlist</i>
<code>--format <i>format</i></code>	Отображать только столбцы, указанные в списке <i>format</i>
<code>--context</code>	Отображать дополнительную информацию о безопасности
<code>--cols <i>n</i></code>	Задать ширину экрана, соответствующую <i>n</i> столбцам
<code>--columns <i>n</i></code>	Задать ширину экрана, соответствующую <i>n</i> столбцам
<code>--cumulative</code>	Включить информацию об остановленных дочерних процессах

Параметр	Описание
--forest	Отображать процессы в виде иерархического листинга, показывающего родительские процессы
--headers	Повторять заголовки столбцов на каждой странице вывода
--no-headers	Не отображать заголовки столбцов
--lines <i>n</i>	Задать высоту экрана в строках дисплея, равную <i>n</i>
--rows <i>n</i>	Задать высоту экрана в строках таблицы, равную <i>n</i>
--sort <i>order</i>	Определить один или несколько столбцов для использования при сортировке выходных данных
--width <i>n</i>	Задать ширину экрана в столбцах, равную <i>n</i>
--help	Отображать справочную информацию
--info	Отображать отладочную информацию
--version	Отображать версию программы <code>ps</code>

Предусмотрена возможность объединять длинные параметры GNU с параметрами в стиле Unix или BSD, чтобы получить в результате настройки наиболее подходящий формат отображения. В составе длинных параметров GNU предусмотрено одно действительно великолепное средство — параметр `--forest`. Он обеспечивает отображение информации о процессе в виде иерархической структуры, но с использованием символов ASCII, в результате чего формируются визуально привлекательные графики:

```

1981 ?      00:00:00 sshd
3078 ?      00:00:00  \_ sshd
3080 ?      00:00:00    \_ sshd
3081 pts/0   00:00:00      \_ bash
16676 pts/0  00:00:00        \_ ps

```

С помощью этого формата можно буквально за мгновение разобраться во взаимосвязях дочерних и родительских процессов!

Отслеживание процессов в реальном времени

Команда `ps` очень хорошо подходит для сбора информации о процессах, работающих в системе, но имеет один недостаток. Дело в том, что команда `ps` может отображать только информацию на какой-то конкретный момент времени. Если же требуется определить тенденции, характеризующие поведение процессов, которые часто выгружаются из памяти и снова загружаются, это нелегко сделать с помощью команды `ps`.

Вместо этого для поиска решения данной проблемы можно применить команду `top`. Команда `top` отображает информацию о процессах, так же, как и команда `ps`, но выполняет это в оперативном режиме. На рис. 4.1 приведен снимок с экрана, на котором показана команда `top` в действии.

В первом разделе этого вывода показана общая информация о системе. В первой строке можно видеть текущее время, продолжительность пребывания системы в рабочем состоянии, количество зарегистрированных пользователей и среднюю загрузку системы.

Средняя загрузка отображается в виде трех чисел: усредненных данных по загрузке в течение 1 мин, 5 мин и 15 мин. Чем выше эти значения, тем большую загрузку испытывает система.

Нередко приходится видеть, что значение загрузки за 1 мин становится высоким при коротких всплесках активности. Если же высоким является значение загрузки за 15 мин, это может свидетельствовать о том, что система работает в неблагоприятных условиях.

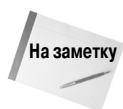
```

top - 16:04:38 up 1 min, 2 users, load average: 0.82, 0.52, 0.20
Tasks: 179 total, 1 running, 178 sleeping, 0 stopped, 0 zombie
Cpu(s): 0.5%us, 1.3%sy, 0.0%ni, 97.8%id, 0.3%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 1026084k total, 433076k used, 593008k free, 50440k buffers
Swap: 2781176k total, 0k used, 2781176k free, 191008k cached

  PID USER      PR  NI  VIRT  RES  SHR  S  %CPU  %MEM    TIME+  COMMAND
  952 root        20   0 35924   22m 7576 S   1   2.3   0:03.99 Xorg
1432 root        20   0 15056  1868 1516 S   0   0.2   0:00.25 prl_wmouse_d
1527 rich        20   0 78512   17m 13m S   0   1.7   0:00.43 nautilus
1668 rich        20   0 64568   15m 11m S   0   1.5   0:01.25 gnome-terminal
   1 root         20   0  2804   1656 1200 S   0   0.2   0:00.61 init
   2 root         20   0     0     0   0 S   0   0.0   0:00.00 kthreadd
   3 root         RT    0     0     0   0 S   0   0.0   0:00.00 migration/0
   4 root         20   0     0     0   0 S   0   0.0   0:00.01 ksoftirqd/0
   5 root         RT    0     0     0   0 S   0   0.0   0:00.00 watchdog/0
   6 root         RT    0     0     0   0 S   0   0.0   0:00.00 migration/1
   7 root         20   0     0     0   0 S   0   0.0   0:00.00 ksoftirqd/1
   8 root         RT    0     0     0   0 S   0   0.0   0:00.00 watchdog/1
   9 root         20   0     0     0   0 S   0   0.0   0:00.00 events/0
  10 root         20   0     0     0   0 S   0   0.0   0:00.04 events/1
  11 root         20   0     0     0   0 S   0   0.0   0:00.00 cpuset
  12 root         20   0     0     0   0 S   0   0.0   0:00.00 khelper
  13 root         20   0     0     0   0 S   0   0.0   0:00.00 netns
  14 root         20   0     0     0   0 S   0   0.0   0:00.00 async/mgr
  15 root         20   0     0     0   0 S   0   0.0   0:00.00 pm
  17 root         20   0     0     0   0 S   0   0.0   0:00.00 sync_supers

```

Рис. 4.1. Вывод команды `top` в ходе ее выполнения



На заметку

Одна из сложностей в системном администрировании Linux состоит в определении того, насколько допустимо то или иное повышенное значение средней загрузки. Это значение зависит от задач, обычно эксплуатируемых в системе, и от конфигурации аппаратных средств. Загрузка, высокая для одной системы, может оказаться допустимой для другой. Как правило, если значение средней загрузки начинает выходить за пределы величины, равной 2, то процессы в системе выполняются не так быстро, как хотелось бы.

Во второй строке показана общая информация о процессах (называемых *задачами*, tasks, в программе `top`), в частности, о количестве процессов в различных состояниях: выполняющиеся, простаивающие, остановленные и зависшие (завершенные, но не получившие ответа от своего родительского процесса).

В следующей строке показана общая информация о процессоре. В выводе программы `top` сведения о загрузке процессора подразделяются на несколько категорий в зависимости от того, каковым является владелец процесса (при этом проводится различие между пользовательскими и системными процессами) и каково состояние процесса (работающий, простаивающий или ждущий).

За этим следуют две строки, которые позволяют уточнить сведения о статусе памяти системы. В первой строке показан статус физической памяти в системе со сведениями о том, чему равен общий объем памяти, какой объем используется в настоящее время и какой остается свободным. Во второй строке со сведениями о памяти показан статус области подкачки, предусмотренной для управления памятью в системе (если таковая имеется), где приведена аналогичная информация.

Наконец, в следующем разделе приведен подробный список процессов, выполняющихся в настоящее время, который включает некоторые столбцы с информацией, во многом аналогичные соответствующим столбцам в выводе команды `ps`, как показано ниже.

- **PID.** Идентификатор рассматриваемого процесса.
- **USER.** Имя пользователя, соответствующее владельцу процесса.
- **PR.** Приоритет процесса.
- **NI.** Величина изменения приоритета процесса.
- **VIRT.** Общий объем виртуальной памяти, используемой процессом.
- **RES.** Объем физической памяти, используемой процессом.
- **SHR.** Объем памяти, совместно используемой процессом с другими процессами.
- **S.** Статус процесса (D — находящийся в прерываемом простое, R — выполняющийся; S — простаивающий; T — отслеживаемый или остановленный; Z — зависший).
- **%CPU.** Доля процессорного времени, используемого процессом.
- **%MEM.** Доля доступной физической памяти, используемой процессом.
- **TIME+.** Общее количество процессорного времени, затраченного процессом с момента запуска.
- **COMMAND:** Имя процесса, заданное в командной строке (запущенная программа).

По умолчанию после запуска команды `top` сортировка процессов осуществляется с учетом значения **%CPU**. От этого порядка сортировки можно перейти к другому с использованием одной из нескольких интерактивных подкоманд во время выполнения команды `top`. Каждая интерактивная подкоманда представляет собой отдельный символ, который можно ввести по ходу выполнения команды `top`, что приводит к изменению поведения программы. Эти подкоманды приведены в табл. 4.4.

Таблица 4.4. Интерактивные подкоманды команды `top`

Команда	Описание
l	Переключиться на отображение состояния отдельного процессора и SMP (Symmetric Multiprocessor — симметричная многопроцессорная система)
b	Переключиться на отображение важных числовых значений в таблицах полужирным шрифтом
I	Переключиться на режим Irix/Solaris
z	Выполнить настройку цвета для таблицы
l	Переключиться на отображение строки информации о средней загрузке
t	Переключиться на отображение строки информации о процессоре
m	Переключиться на отображение строк информации MEM и SWAP
f	Добавить или удалить те или иные информационные столбцы
o	Изменить порядок отображения информационных столбцов
F или O	Выбрать поле, по которому должны быть отсортированы процессы (по умолчанию применяется %CPU)
< или >	Переместить поле сортировки на один столбец влево (<) или вправо (>)
r	Переключиться на обычный или обратный порядок сортировки
h	Переключиться на отображение потоков
c	Переключиться на отображение для процессов имени команды или полной командной строки (включая параметры)

Команда	Описание
i	Переключиться на отображение простаивающих процессов
S	Переключиться на отображение совокупного процессорного времени или относительного процессорного времени
x	Переключиться на выделение подсветкой поля сортировки
y	Переключиться на выделение подсветкой работающих задач
z	Переключиться на режим цветного или монохромного отображения
b	Переключиться на режим отображения полужирным шрифтом для режимов x и y
u	Показывать процессы для конкретного пользователя
n или #	Задать количество отображаемых процессов
k	Уничтожить конкретный процесс (подкоманда доступна только владельцу процесса или пользователю root)
r	Изменить (renice) приоритет конкретного процесса (подкоманда доступна только владельцу процесса или пользователю root)
d или s	Изменить интервал обновления (значение по умолчанию — три секунды)
W	Записать текущие значения параметров в файл конфигурации
q	Выйти из команды top

Очевидно, что предусмотрены широкие возможности управления выводом команды top. Использование этого инструмента часто позволяет находить процессы, нарушающие нормальную работу, которые действуют в системе. Безусловно, после обнаружения такого процесса следующей задачей становится прекращение его работы, описанию чего посвящен следующий раздел.

Останов процессов

Одной из крайне важных задач системного администратора является выполнение своевременного и правильного останова процессов. Иногда процесс приостанавливается и требует лишь небольшого вмешательства для продолжения своей работы или полного останова. Бывают и такие ситуации, что процесс становится неуправляемым, безмерно поглощает процессорное время и не может быть прекращен обычными способами. И в том и в другом случае для управления процессом необходимо воспользоваться определенной командой. В системе Linux используется такой же способ организации межпроцессного взаимодействия, как и в системе Unix.

В Linux процессы обмениваются друг с другом данными с использованием *сигналов*. Сигнал процесса — это стандартное сообщение, распознаваемое процессом, после получения которого процесс может выбрать, игнорировать ли его или выполнить соответствующее ему действие. Применяемые в процессе способы обработки сигналов закладываются разработчиками при написании программы. В большинстве качественно написанных приложений предусмотрена способность получать стандартные сигналы процесса Unix и действовать согласно этим сигналам. Указанные сигналы приведены в табл. 4.5.

В системе Linux предусмотрены две описанные ниже команды, которые позволяют отправлять сигналы работающим процессам.

Таблица 4.5. Сигналы процесса Linux

Сигнал	Имя	Описание
1	HUP	Отбой
2	INT	Прерывание
3	QUIT	Прекратить выполнение
9	KILL	Безоговорочно завершить
11	SEGV	Нарушение сегмента
15	TERM	Завершить, если возможно
17	STOP	Безоговорочно остановить, но не завершать
18	TSTP	Остановить или приостановить, но продолжить работу в фоновом режиме
19	CONT	Возобновить выполнение после сигналов STOP или TSTP

Команда kill

Команда `kill` дает возможность отправлять сигналы процессам с учетом их идентификатора процесса (process ID — PID). По умолчанию команда `kill` отправляет сигнал `TERM` всем процессам с идентификаторами процессов, перечисленными в командной строке. К сожалению, предусмотрена возможность использовать лишь PID процесса вместо имени команды, с помощью которой он был запущен, поэтому иногда задача правильного выполнения команды `kill` становится затруднительной.

Чтобы иметь право передать сигнал процессу, необходимо либо быть владельцем процесса, либо зарегистрироваться как пользователь `root`.

```
$ kill 3940
-bash: kill: (3940) - Operation not permitted
$
```

Сигнал `TERM` указывает процессу, что должна быть выполнена процедура корректного прекращения выполнения. К сожалению, если при этом приходится иметь дело с процессом, вышедшем из-под контроля, то этот запрос, скорее всего, будет проигнорирован. Если потребуется завершить выполнение процесса принудительно, можно воспользоваться параметром `-s`, позволяющим указывать другие сигналы (с использованием имени или номера сигнала).

Общепринятая процедура состоит в том, что вначале следует попытаться применить сигнал `TERM`. Если этот сигнал будет проигнорирован процессом, то предпринимается попытка отправить сигнал `INT` или `HUP`. Если программа распознает такой сигнал, то попытается корректно завершить выполняемые в ней операции, прежде чем произвести останов. Сигналом, определяющим наивысшую степень принудительного завершения, является сигнал `KILL`. После получения этого сигнала процесс немедленно прекращает выполнение. Это может привести к искажению информации в файлах.

Как показывает следующий пример, с командой `kill` не связан какой-либо вывод:

```
# kill -s HUP 3940
#
```

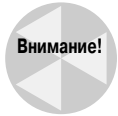
Для определения того, оказалась ли результативной эта команда, необходимо еще раз вызвать на выполнение команду `ps` или `top`, чтобы узнать, удалось ли остановить процесс, нарушающий нормальную работу.

Команда killall

Команда `killall` представляет собой мощный способ останова процессов с использованием их имен, а не числовых значений PID. Команда `killall` позволяет также использовать символы-шаблоны, поэтому становится весьма полезным инструментом при восстановлении нормальной работы системы, в которой возникли недопустимые процессы:

```
# killall http*  
#
```

В этом примере показано уничтожение всех процессов, имена которых начинаются с `http`, в частности служб `httpd` для веб-сервера Apache.



Будучи зарегистрированным в качестве пользователя `root`, следует использовать команду `killall` с исключительной осторожностью. Дело в том, что можно легко потерять контроль над символами-шаблонами и непреднамеренно остановить важные системные процессы. Это может привести к повреждению файловой системы.

Контроль над использованием места на диске

Еще одна важная задача системного администратора состоит в отслеживании использования дисковой памяти в системе. Независимо от того, эксплуатируется ли обычный компьютер с рабочим столом Linux или крупный сервер Linux, необходимо знать, каковы характеристики пространства, доступного для конкретных приложений.

Предусмотрен целый ряд команд командной строки, которые могут использоваться для управления средой, создаваемой носителями информации в системе Linux. В настоящем разделе рассматриваются основные команды, с которыми приходится сталкиваться каждому системному администратору во время работы.

Монтирование носителей информации

Как описано в главе 3, функционирование файловой системы Linux основано на том, что все содержимое носителей информации объединяется в общем виртуальном каталоге. Прежде чем появится возможность использовать в системе новый носитель информации, его содержимое необходимо включить в виртуальный каталог. Соответствующая операция называется *монтированием*.

В современном мире, в котором широко применяются графические рабочие столы, большинство дистрибутивов Linux обладает способностью автоматически монтировать конкретные типы *сменных носителей*. Устройство со сменными носителями информации само рассматривается как носитель, который (что само собой разумеется) можно легко удалить из персонального компьютера; к этой категории относятся компакт-диски, гибкие диски и флеш-накопители с интерфейсом USB.

Если применяемый дистрибутив не обеспечивает автоматическое монтирование и размонтирование сменных носителей, то эту операцию приходится непосредственно выполнять пользователю. В настоящем разделе рассматриваются команды командной строки Linux, позволяющие управлять устройствами со сменными носителями информации.

Команда mount

Не правда ли, немного неожиданно, что команда, используемая для монтирования носителей информации, носит имя `mount`! А если говорить серьезно, то по умолчанию команда `mount` отображает список устройств с носителями информации, смонтированными в настоящее время в системе:

```
$ mount
/dev/mapper/VolGroup00-LogVol00 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
/dev/sda1 on /boot type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
sunrpc on /var/lib/nfs/rpc_pipefs type rpc_pipefs (rw)
/dev/sdb1 on /media/disk type vfat
(rw,nosuid,nodev,uhelper=hal,shortname=lower,uid=503)
$
```

Команда `mount` предоставляет информацию следующих четырех категорий.

- Имя файла устройства с носителем информации.
- Точка монтирования в виртуальном каталоге, где смонтирован носитель информации.
- Тип файловой системы.
- Статус доступа к смонтированному носителю информации.

Последняя запись в предыдущем примере показывает флеш-накопитель USB, который автоматически смонтирован программами рабочего стола GNOME в точке монтирования `/media/disk`. Обозначение типа файловой системы `vfat` показывает, что этот носитель первоначально отформатирован на персональном компьютере Microsoft Windows.

Чтобы иметь возможность монтировать устройства с носителями информации в виртуальном каталоге вручную, необходимо зарегистрироваться как пользователь `root`. Ниже приведена основная команда монтирования вручную устройства с носителем информации.

```
mount -t type device directory
```

Параметр `type` определяет тип файловой системы, с помощью которой был отформатирован диск. Количество различных типов файловых систем, распознаваемых в Linux, чрезвычайно велико. Если устройства со сменными носителями информации применяются для работы и в ОС Windows, и в ОС Linux, то, по всей вероятности, на этих устройствах будут применяться типы, перечисленные ниже.

- `vfat`. Файловая система Windows с длинными именами файлов.
- `ntfs`. Усовершенствованная файловая система Windows, используемая в версиях Windows NT, XP и Vista.
- `iso9660`. Стандартная файловая система компакт-дисков.

Флеш-накопители USB и дискеты чаще всего бывают отформатированными с применением файловой системы `vfat`. Если потребуется смонтировать CD с данными, то необходимо будет воспользоваться файловой системой типа `iso9660`.

Следующие два параметра определяют местонахождение файла устройства для конкретного устройства с носителем информации, а также расположение точки монтирования в виртуальном

каталоге. Например, чтобы вручную смонтировать флеш-накопитель USB на устройстве `/dev/sdb1` в местоположении `/media/disk`, необходимо воспользоваться следующей командой:

```
mount -t vfat /dev/sdb1 /media/disk
```

После того как устройство с носителем информации будет смонтировано в виртуальном каталоге, пользователь `root` получает полный доступ к этому устройству, однако доступ других пользователей является ограниченным. Предусмотрена возможность управлять предоставлением доступа к устройству путем определения прав доступа к каталогу (о чем речь пойдет в главе 6).

В табл. 4.6 приведены параметры, которые могут потребоваться в случае использования некоторых дополнительных функций команды `mount`.

Таблица 4.6. Параметры команды `mount`

Параметр	Описание
<code>-a</code>	Монтировать все файловые системы, указанные в файле <code>/etc/fstab</code>
<code>-f</code>	Применение этого параметра приводит к тому, что команда <code>mount</code> моделирует монтирование устройства, но не монтирует его фактически
<code>-F</code>	При использовании с параметром <code>-a</code> одновременно монтируются все файловые системы
<code>-v</code>	Задать подробный режим с объяснением всех шагов, необходимых для монтирования устройства
<code>-I</code>	Не использовать вспомогательные файлы файловой системы из подкаталога <code>/sbin/mount.filesystem</code>
<code>-l</code>	Добавить метки файловой системы для файловых систем <code>ext2</code> , <code>ext3</code> , или <code>XFS</code> автоматически
<code>-n</code>	Монтировать устройство, не регистрируя его в смонтированном файле устройства <code>/etc/mstab</code>
<code>-p num</code>	Для монтирования с шифрованием считать ключевую фразу из дескриптора файла <code>num</code>
<code>-s</code>	Пропускать параметры монтирования, не поддерживаемые файловой системой
<code>-r</code>	Монтировать устройство как допускающее только чтение
<code>-w</code>	Монтировать устройство как предназначенное для чтения и записи (значение по умолчанию)
<code>-L label</code>	Монтировать устройство с указанной меткой <code>label</code>
<code>-U uuid</code>	Монтировать устройство с указанным идентификатором <code>uuid</code>
<code>-O</code>	При использовании с параметром <code>-a</code> ограничить набор применяемых файловых систем
<code>-o</code>	Добавить конкретные параметры для файловой системы

Параметр `-o` позволяет монтировать файловую систему с разделенным запятыми списком дополнительных параметров. К числу широко используемых параметров относятся следующие.

- `ro`. Монтировать устройство как допускающее только чтение.
- `rw`. Монтировать устройство как предназначенное для чтения и записи.
- `user`. Разрешить монтировать файловую систему обычному пользователю.
- `check=none`. Монтировать файловую систему, не выполняя проверку целостности.
- `loop`. Монтировать файл.

В наши дни среди пользователей системы Linux получил широкое распространение способ передачи содержимого CD в виде файлов с расширением `.iso`. Файл `.iso` представляет собой полный образ CD, оформленный в виде одного файла. Возможность создать новый CD на основе файла `.iso` предусмотрена в большинстве пакетов программ записи CD. Одна из особенностей команды `mount` состоит в том, что она позволяет монтировать файл `.iso` непосред-

ственно в виртуальный каталог Linux, не требуя предварительной записи содержимого этого файла на CD. Эта операция осуществляется при использовании параметра `-o` с опцией `loop`:

```
$ mkdir mnt
$ su
Password:
# mount -t iso9660 -o loop MEPIS-KDE4-LIVE-DVD_32.iso mnt
# ls -l mnt
total 16
-r--r--r-- 1 root root 702 2007-08-03 08:49 about
dr-xr-xr-x 3 root root 2048 2007-07-29 14:30 boot
-r--r--r-- 1 root root 2048 2007-08-09 22:36 boot.catalog
-r--r--r-- 1 root root 894 2004-01-23 13:22 cdrom.ico
-r--r--r-- 1 root root 5229 2006-07-07 18:07 MCWL
dr-xr-xr-x 2 root root 2048 2007-08-09 22:32 mepis
dr-xr-xr-x 2 root root 2048 2007-04-03 16:44 OSX
-r--r--r-- 1 root root 107 2007-08-09 22:36 version
# cd mnt/boot
# ls -l
total 4399
dr-xr-xr-x 2 root root 2048 2007-06-29 09:00 grub
-r--r--r-- 1 root root 2392512 2007-07-29 12:53 initrd.gz
-r--r--r-- 1 root root 94760 2007-06-14 14:56 memtest
-r--r--r-- 1 root root 2014704 2007-07-29 14:26 vmlinuz
#
```

После монтирования файла образа CD `.iso` с помощью команды `mount` содержимое диска становится доступным, как если бы это был настоящий CD, и появляется возможность переходить по его файловой системе.

Команда `umount`

Чтобы исключить из виртуального каталога содержимое устройства со сменным носителем информации, его ни в коем случае нельзя просто удалять из системы. Вместо этого следует вначале всегда применять к нему операцию *размонтирования*.



Система Linux не позволяет выталкивать из устройства смонтированный CD. Если возникают затруднения при извлечении диска из устройства, по всей вероятности, это означает, что он все еще остается смонтированным в виртуальном каталоге. Необходимо вначале его размонтировать, а затем снова попытаться вытолкнуть.

Для размонтирования (`umount`) устройств используется команда `umount` (это не опечатка; в команде действительно отсутствует первая буква “n” слова `unmount`, что иногда приводит к путанице). Команда `umount` имеет довольно простой формат вызова:

```
umount [directory | device ]
```

Команда `umount` предоставляет возможность указывать устройство с носителем информации либо по местоположению устройства, либо по имени смонтированного каталога. Если на устройстве какой-либо программой открыт хотя бы один файл, то система не позволит размонтировать устройство.

```
[root@testbox mnt]# umount /home/rich/mnt
umount: /home/rich/mnt: device is busy
```

```

umount: /home/rich/mnt: device is busy
[root@testbox mnt]# cd /home/rich
[root@testbox rich]# umount /home/rich/mnt
[root@testbox rich]# ls -l mnt
total 0
[root@testbox rich]#

```

В данном примере приглашение к вводу команд все еще указывает на один из каталогов в пределах структуры файловой системы, поэтому попытка размонтировать файл образа с помощью команды `umount` окончилась неудачей. После того как в приглашении к вводу команд произойдет переход за пределы файловой системы файла образа, с помощью команды `umount` будет успешно размонтирован файл образа.

Использование команды `df`

Иногда возникает необходимость узнать, каков объем свободного пространства на каждом отдельном устройстве. С помощью команды `df` можно легко определить характеристики всех смонтированных дисков:

```

$ df
Filesystem            1K-blocks    Used Available Use% Mounted on
/dev/sda2              18251068   7703964   9605024   45% /
/dev/sda1              101086     18680     77187    20% /boot
tmpfs                  119536         0    119536     0% /dev/shm
/dev/sdb1              127462    113892     13570    90% /media/disk
$

```

Команда `df` показывает каждую смонтированную файловую систему, которая содержит данные. Как показывают результаты выполнения команды `mount`, приведенные выше, некоторые смонтированные устройства используются для внутренних потребностей системы. Команда отображает сведения, которые указаны ниже.

- Местоположение устройства.
- Количество 1024-байтовых блоков данных, которые могут на нем храниться.
- Количество используемых 1024-байтовых блоков.
- Количество доступных 1024-байтовых блоков.
- Относительное количество используемого пространства в процентных долях.
- Точка монтирования, в которой смонтировано устройство.

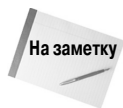
В команде `df` предусмотрено большое количество параметров командной строки, большая часть которых обычно не используется. Одним из широко применяемых параметров является параметр `-h`, который показывает место на диске в формате, удобном для восприятия человеком, с указанием суффикса `M` для обозначения мегабайтов или `G` — для гигабайтов:

```

$ df -h
Filesystem            Size  Used Avail Use% Mounted on
/dev/sdb2             18G   7.4G   9.2G   45% /
/dev/sda1             99M   19M   76M   20% /boot
tmpfs                 117M    0   117M    0% /dev/shm
/dev/sdb1             125M  112M   14M   90% /media/disk
$

```

При этом пользователь избавлен от необходимости расшифровывать крайне неудобные для восприятия значения в виде количества блоков, поскольку все объемы на диске отображаются с использованием “обычных” размеров. Команда `df` представляет собой бесценный инструмент устранения неполадок, возникающих в связи с нехваткой дискового пространства в системе.



Следует учитывать, что в работающей системе Linux всегда присутствуют процессы, выполняющиеся в фоновом режиме, которые обеспечивают обработку файлов. Значения, полученные с помощью команды `df`, отражают результаты измерения системой Linux текущих значений на данный момент времени. Иногда обнаруживается, что какой-то процесс, функционирующий в системе, создал или удалил файл, но еще не освободил этот файл. Связанное с этим значение не учитывается при вычислении объема свободного пространства.

Использование команды `du`

Команда `df` позволяет легко определить, не достигнуто ли на диске состояние нехватки свободного пространства. Системный администратор должен определить, что делать в связи с возникновением этой проблемы.

Для этого можно воспользоваться еще одной полезной командой — `du`. Команда `du` показывает использование дискового пространства применительно к конкретному каталогу (по умолчанию таковым является текущий каталог). С помощью указанной команды можно быстро определить, нет ли таких файлов, которые со всей очевидностью бесполезно расходуют дисковое пространство в системе.

По умолчанию команда `du` отображает все файлы, каталоги и подкаталоги, расположенные под текущим каталогом, а также показывает, сколько дисковых блоков занимает каждый файл или каталог. Если это один из часто встречающихся каталогов в системе, то полученный при этом листинг может иметь весьма значительный объем. Ниже приведена часть листинга, полученного с использованием команды `du`.

```
$ du
484    ./gstreamer-0.10
8      ./Templates
8      ./Download
8      ./ccache/7/0
24     ./ccache/7
368    ./ccache/a/d
384    ./ccache/a
424    ./ccache
8      ./Public
8      ./gphpedit/plugins
32     ./gphpedit
72     ./gconfd
128    ./nautilus/metafiles
384    ./nautilus
72     ./bittorrent/data/metainfo
20     ./bittorrent/data/resume
144    ./bittorrent/data
152    ./bittorrent
8      ./Videos
8      ./Music
16     ./config/gtk-2.0
```

```
40      ./config
8       ./Documents
```

Число слева от каждой строки указывает количество дисковых блоков, которые занимает соответствующий файл или каталог. Заслуживает внимания то, что листинг начинается с конца каталога и переходит вверх по файлам и подкаталогам, содержащимся в пределах каталога.

Иногда не имеет смысла применять команду `du` отдельно. Разумеется, неплохо иметь возможность узнать, какой объем дискового пространства занимает каждый отдельный файл и каталог, но применение этих данных становится неоправданным, если приходится перелистывать одну за другой страницу с информацией, прежде чем удастся найти искомые сведения.

Поэтому предусмотрено несколько параметров командной строки, которые можно использовать с командой `du` в целях получения более удобных для восприятия сведений, как указано ниже.

- `-c`. Сформировать общий итог по всем перечисленным файлам.
- `-h`. Выводить значения размеров в форме, удобной для восприятия человеком, с использованием суффикса К для обозначения килобайтов, М — мегабайтов и G — гигабайтов.
- `-s`. Суммировать данные по каждому параметру.

На следующем этапе устранения нарушений в работе системный администратор должен воспользоваться некоторыми командами манипулирования файлами, чтобы обеспечить управление большими объемами данных. Именно этому посвящен следующий раздел.

Работа с файлами данных

Если приходится иметь дело со слишком большим количеством данных, часто становятся затруднительными обработка информации и извлечение из нее пользы. Как показывают примеры применения команды `du`, приведенные в предыдущем разделе, при работе с системными командами часто может неожиданно возникнуть перегрузка по данным.

В системе Linux предусмотрено несколько инструментов с интерфейсом командной строки, позволяющих управлять большими объемами данных. Следующий раздел содержит описание нескольких основных команд, которые должен изучить каждый системный администратор (а также любой пользователь, повседневно работающий в системе Linux), чтобы уметь воспользоваться ими для облегчения своей работы.

Сортировка данных

Одной из широко применяемых функций, необходимость в использовании которой часто возникает при работе с большими объемами данных, является команда `sort`. Она выполняет то, что следует из ее имени, — сортирует данные.

По умолчанию команда `sort` сортирует строки данных в текстовом файле на основании стандартных правил сортировки для языка, указанного как применяемый по умолчанию в текущем сеансе.

```
$ cat file1
one
two
three
four
five
$ sort file1
```

```
five
four
one
three
two
$
```

На первый взгляд все довольно просто. Однако в действительности при использовании команды `sort` возникают определенные сложности. Рассмотрим следующий пример:

```
$ cat file2
1
2
100
45
3
10
145
75
$ sort file2
1
10
100
145
2
3
45
75
$
```

Если это была попытка отсортировать числа в порядке возрастания числовых значений, то она оказалась неудачной. По умолчанию команда `sort` интерпретирует цифры как символы и выполняет стандартную лексикографическую сортировку, что приводит к получению вывода, который может оказаться далеким от ожидаемого. Чтобы решить эту проблему, необходимо воспользоваться параметром `-n`, который служит для команды `sort` указанием на то, что строки цифр должны рассматриваться как числа, а не символы, а сортировка должна проводиться с учетом их числовых значений:

```
$ sort -n file2
1
2
3
10
45
75
100
145
$
```

Теперь полученный результат намного лучше! Еще одним часто применяемым параметром является `-M`, который обеспечивает сортировку по месяцам. Файлы журнала Linux обычно содержат отметку времени в начале строки, которая указывает, когда произошло событие:

```
Sep 13 07:10:09 textbox smartd[2718]: Device: /dev/sda, opened
```

Если сортировка файла, в котором присутствуют даты в виде отметок времени, проводится с использованием предусмотренных по умолчанию параметров, то результат приобретает примерно такой вид:

```
$ sort file3
Apr
Aug
Dec
Feb
Jan
Jul
Jun
Mar
May
Nov
Oct
Sep
$
```

Вряд ли это то, что требовалось. Если же используется параметр `-M`, то команда сортировки распознает трехсимвольное обозначение месяца и проводит сортировку должным образом:

```
$ sort -M file3
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
$
```

Как показано в табл. 4.7, предусмотрено еще несколько удобных параметров команды `sort`.

Таблица 4.7. Параметры команды `sort`

<i>Одинарное тире</i>	<i>Двойное тире</i>	<i>Описание</i>
<code>-b</code>	<code>--ignore-leading-blanks</code>	Пропускать начальные пробелы при сортировке
<code>-C</code>	<code>--check = quiet</code>	Не сортировать, но и не сообщать, если данные находятся в не-отсортированном порядке
<code>-c</code>	<code>--check</code>	Не сортировать, но проверять, отсортированы ли уже входные данные. Сообщать, если данные не отсортированы
<code>-d</code>	<code>--dictionary-order</code>	Учитывать только пробелы и алфавитно-цифровые символы; не учитывать специальные символы
<code>-g</code>	<code>--general-numeric-sort</code>	Использовать общее числовое значение при определении порядка сортировки

Одинарное тире	Двойное тире	Описание
-f	--ignore-case	По умолчанию прописные буквы рассматриваются в порядке сортировки как предшествующие строчным. Этот параметр указывает, что регистр букв не должен учитываться
-i	--ignore-nonprinting	Пропускать при сортировке непечатаемые символы
-k	--key = <i>POS1</i> , <i>POS2</i>	Сортировать с проведением сравнения, начиная с позиции <i>POS1</i> и заканчивая позицией <i>POS2</i> , если она указана
-M	--month-sort	Сортировать в последовательности месяцев в году, используя трехсимвольные названия месяцев
-m	--merge	Объединить два уже отсортированных файла данных
-n	--numeric-sort	Сортировать по числовым значениям строк
-o	--output = <i>file</i>	Записать результаты в указанный файл
-R	--random-sort	Сортировать по случайному хешу ключей
	--random-source = <i>FILE</i>	Указать файл для случайного выбора байтов, используемых параметром -R
-r	--reverse	Сортировать в обратном порядке (по убыванию, а не по возрастанию)
-S	--buffer-size = <i>SIZE</i>	Задать используемый объем памяти
-s	--stable	Отключить функцию сравнения, применяемую на крайний случай
-T	--temporary-direction = <i>DIR</i>	Указать местоположение, предназначенное для хранения временных рабочих файлов
-t	--field-separator = <i>SEP</i>	Указать символ, используемый для разграничения позиций ключей
-u	--unique	При использовании с параметром -c проверять на наличие строгого упорядочения; без параметра -c выводить только первое вхождение двух аналогичных строк
-z	--zero-terminated	Завершать все строки нуль-символом вместо символа перевода на новую строку

Параметры -k и -t могут применяться при сортировке данных, состоящих из полей, таких как пароли в файле /etc/passwd. Параметр -t предназначен для указания символического значения разделителя полей, а параметр -k указывает, по какому полю должна быть проведена сортировка. Например, чтобы отсортировать файл паролей на основе числовых идентификаторов пользователей, достаточно выполнить следующую команду:

```
$ sort -t ':' -k 3 -n /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
```

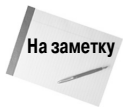
```
news:x:9:13:news:/etc/news:
uucp:x:10:14:uucp:/var/spool/uucp:/sbin/nologin
operator:x:11:0:operator:/root:/sbin/nologin
games:x:12:100:games:/usr/games:/sbin/nologin
gopher:x:13:30:gopher:/var/gopher:/sbin/nologin
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
```

После этого данные становятся идеальным образом отсортированными по третьему полю, которое содержит числовые значения идентификаторов пользователей.

Параметр `-n` превосходно подходит для сортировки числовых данных, наподобие тех, что имеются в выводе команды `du`:

```
$ du -sh * | sort -nr
1008k  mrtg-2.9.29.tar.gz
972k   bldgl
888k   fbs2.pdf
760k   Printtest
680k   rsync-2.6.6.tar.gz
660k   code
516k   fig1001.tiff
496k   test
496k   php-common-4.0.4pl1-6mdk.i586.rpm
448k   MesaGLUT-6.5.1.tar.gz
400k   plp
```

Обратите внимание на то, что при этом параметр `-r` сортирует значения в порядке убывания, поэтому можно легко определить, какие файлы занимают основной объем пространства в рассматриваемом каталоге.



В данном примере использовалась команда канала (`|`) для перенаправления выходных данных команды `du` на вход команды `sort`. Эта тема рассматривается более подробно в главе 10.

Поиск данных

В процессе работы часто возникает необходимость найти в большом файле определенную строку данных, скрывающуюся глубоко в середине этого файла. Предусмотрена возможность вместо личного визуального просмотра всего файла применить для поиска требуемой информации команду `grep`. Формат командной строки для команды `grep` является таковым:

```
grep [options] pattern [file]
```

Команда `grep` выполняет поиск в своих входных данных или в указанном файле тех строк, которые содержат символы, сопоставляемые с заданным шаблоном. После этого команда `grep` выводит строки, содержащие данные, которые согласуются с шаблоном.

Ниже приведены два простых примера применения команды `grep` к файлу `file1`, который использовался в разделе “Сортировка данных”.

```
$ grep three file1
three
$ grep t file1
two
three
$
```

В первом примере в файле `file1` производится поиск текста, сопоставляемого с шаблоном `three`. Затем команда `grep` выводит строку, содержащую сопоставленный шаблон. В следующем примере в файле `file1` происходит поиск текста, согласующегося с шаблоном `t`. В этом случае обнаруживаются две строки, сопоставляемые с указанным шаблоном, и обе эти строки отображаются.

Команда `grep` нашла очень широкое применение, поэтому со времени ее создания подверглась существенным доработкам со стороны многих разработчиков. В частности, к команде `grep` было добавлено много дополнительных функций. Изучение справочного руководства по команде `grep` показывает, насколько значительными в действительности являются ее возможности.

Например, если требуется получить результаты, обратные по отношению к заданному условию поиска (вывести строки, не сопоставляющиеся с шаблоном), то можно воспользоваться параметром `-v`:

```
$ grep -v t file1
one
four
five
$
```

Чтобы определить номера строк, в которых обнаружены сопоставляющиеся шаблоны, достаточно задать параметр `-n`:

```
$ grep -n t file1
2:two
3:three
$
```

Если же требуется лишь узнать, в каком количестве строк содержатся данные, согласующиеся с шаблоном, то можно применить параметр `-c`:

```
$ grep -c t file1
2
$
```

Если возникает необходимость задать больше одного шаблона согласования, то можно воспользоваться параметром `-e` для указания отдельно каждого шаблона:

```
$ grep -e t -e f file1
two
three
four
five
$
```

В этом примере отображаются строки, содержащие подстроку `t` или `f`.

По умолчанию в команде `grep` для сопоставления с шаблонами используются основные регулярные выражения в стиле Unix. Регулярное выражение в стиле Unix состоит из специальных символов, которые определяют, как должен осуществляться поиск данных для определения соответствия с шаблонами.

Более подробные сведения о регулярных выражениях приведены в главе 19.

Ниже приведен простой пример использования регулярного выражения при поиске данных с помощью команды `grep`.

```
$ grep [tf] file1
two
```

```
three
four
five
$
```

Квадратные скобки в регулярном выражении указывают, что команда `grep` должна провести поиск согласований, которые содержат символ *t* или *f*. Без применения этого регулярного выражения в команде `grep` осуществлялся бы поиск текста, который сопоставляется со строкой `tf`.

На основе `grep` была разработана команда `egrep`, позволяющая задавать расширенные регулярные выражения в стиле POSIX, в которых увеличено количество символов, применяемых для составления шаблонов соответствия (и эта тема более подробно рассматривается в главе 19). Еще одной версией `grep` является команда `fgrep`, которая позволяет задавать шаблоны соответствия в виде списка постоянных строковых значений, разделенных символами конца строки. Это позволяет поместить список искомых строк в файл, а затем задать этот список в команде `fgrep` для поиска строк в более крупном файле.

Сжатие данных

Любой пользователь, имеющий достаточно продолжительный опыт работы в системе Microsoft Windows, без сомнения, имел дело с файлами `zip`. Файлы `zip` постепенно нашли такое широкое применение, что в конечном итоге корпорация Microsoft включила это средство в операционную систему Windows XP. Программа `zip` позволяет легко осуществлять упаковку больших файлов (и текстовых, и исполняемых) с получением меньших по размеру файлов, для которых не требуется столь много дискового пространства.

В системе Linux также содержится множество средств сжатия файлов. Безусловно, на первый взгляд это кажется привлекательным, но приводит к путанице и хаосу, когда приходится загружать файлы. Средства сжатия файлов, предусмотренные для Linux, перечислены в табл. 4.8.

Таблица 4.8. Средства сжатия файлов Linux

Программа	Расширение файла	Описание
<code>bzip2</code>	<code>.bz2</code>	Использовать алгоритм сжатия текста с блочной сортировкой Барроуза-Уилера (Burrows-Wheeler) и кодирование по алгоритму Хаффмена (Huffman)
<code>compress</code>	<code>.Z</code>	Исходное средство сжатия файлов Unix; начинает полностью выходить из употребления
<code>gzip</code>	<code>.gz</code>	Программа сжатия из проекта GNU; использует кодирование по алгоритму Лемпеля-Зива (Lempel-Ziv)
<code>zip</code>	<code>.zip</code>	Версия Unix программы PKZIP для Windows

Средство сжатия файлов `compress` предусмотрено не во всех версиях системы Linux. После загрузки файла с расширением `.Z` может потребоваться установить пакет с программой сжатия (называемой `ncompress` во многих дистрибутивах Linux), используя методы установки программ, рассматриваемые в главе 8, а затем распаковать этот файл с помощью команды `uncompress`.

Программа `bzip2`

Все более широкое распространение находит программа `bzip2` — относительно новый пакет сжатия, который особенно хорошо подходит для работы с большими двоичными файлами. В пакете `bzip2` содержатся приведенные ниже программы.

- Программа `bzip2`, предназначенная для упаковки файлов.
- Программа `bzcat`, применяемая для отображения содержимого упакованных текстовых файлов.
- Программа `bunzip2`, которая служит для распаковки упакованных файлов с расширением `.bz2`
- Программа `bzip2recover`, с помощью которой можно предпринять попытку восстановить поврежденные сжатые файлы.

По умолчанию команда `bzip2` пытается упаковать исходный файл и заменить его сжатым файлом, для которого используется то же имя файла и расширение `.bz2`:

```
$ ls -l myprog
-rwxrwxr-x 1 rich rich 4882 2007-09-13 11:29 myprog
$ bzip2 myprog
$ ls -l my*
-rwxrwxr-x 1 rich rich 2378 2007-09-13 11:29 myprog.bz2
$
```

Программа `myprog` первоначально имела размер 4882 байта, а после сжатия с помощью `bzip2` стала занимать 2378 байтов. Следует также отметить, что команда `bzip2` автоматически переименовала исходный файл, присвоив ему расширение `.bz2`, что может указывать, какой способ сжатия использовался для упаковки данного файла.

Для распаковки этого файла достаточно воспользоваться командой `bunzip2`:

```
$ bunzip2 myprog.bz2
$ ls -l myprog
-rwxrwxr-x 1 rich rich 4882 2007-09-13 11:29 myprog
$
```

Как показывают приведенные выше результаты, распакованный файл возвратился к своему исходному размеру. После упаковки текстового файла становится невозможным использование стандартных команд `cat`, `more` или `less` для просмотра данных. Вместо этого необходимо применить команду `bzcat`:

```
$ bzcat test.bz2
This is a test text file.
The quick brown fox jumps over the lazy dog.
This is the end of the test text file.
$
```

Команда `bzcat` отображает текст, представленный в сжатом файле, без распаковки самого файла.

Программа `gzip`

Наиболее широко применяемым средством сжатия файлов в системе Linux является программа `gzip`. Пакет `gzip` создан в рамках проекта GNU, разработчики которого стремились создать бесплатную версию оригинальной программы упаковки из системы Unix. Этот пакет включает следующие файлы.

- Программа `gzip`, предназначенная для упаковки файлов.
- Программа `gzcat`, которая служит для отображения содержимого упакованных текстовых файлов.
- Программа `gunzip`, применяемая для распаковки файлов.

Эти программы функционируют полностью аналогично программе bzip2:

```
$ gzip myprog
$ ls -l my*
-rwxrwxr-x 1 rich rich 2197 2007-09-13 11:29 myprog.gz
$
```

Команда `gzip` упаковывает файлы, которые указаны в командной строке. Предусмотрена также возможность указывать несколько имен файлов и даже использовать символы-шаблоны для упаковки сразу нескольких файлов:

```
$ gzip my*
$ ls -l my*
-rwxr--r-- 1 rich rich 103 Sep 6 13:43 myprog.c.gz
-rwxr-xr-x 1 rich rich 5178 Sep 6 13:43 myprog.gz
-rwxr--r-- 1 rich rich 59 Sep 6 13:46 myscript.gz
-rwxr--r-- 1 rich rich 60 Sep 6 13:44 myscript0.gz
$
```

Команда `gzip` упаковывает все файлы в каталоге, имя которого сопоставляется с шаблоном, заданным символом-заместителем.

Программа zip

Программа `zip` совместима с популярным пакетом PKZIP, который создан Филом Катцем (Phil Katz) для операционных систем MS-DOS и Windows. В пакете `zip` для Linux имеются четыре программы.

- Программа `zip`, которая создает сжатый файл, содержащий перечисленные файлы и каталоги.
- Программа `zipcloak`, формирующая зашифрованный, упакованный файл, содержащий перечисленные файлы и каталоги.
- Программа `zipnote`, которая извлекает комментарии из файла `zip`.
- Программа `zipsplit`, позволяющая разбить файл `zip` на меньшие файлы заданного размера (эта программа в свое время использовалась для копирования больших файлов `zip` на дискеты).
- Программа `unzip`, которая извлекает файлы и каталоги из упакованного файла `zip`.

Для просмотра всех параметров, предусмотренных для программы `zip`, достаточно ввести в командной строке имя этой программы без параметра:

```
$ zip
Copyright (C) 1990-2005 Info-ZIP
Type 'zip -L' for software license.
Zip 2.31 (March 8th 2005). Usage:
zip [-options] [-b path] [-t mmdyyy] [-n suffixes] [zipfile list]
[-xi list]

The default action is to add or replace zipfile entries from list,
which can include the special name - to compress standard input.
If zipfile and list are omitted, zip compresses stdin to stdout.
-f freshen: only changed files  -u update: only changed or new files
-d delete entries in zipfile    -m move into zipfile (delete files)
-r recurse into directories    -j junk directory names
```

```

-0 store only
-l compress faster
-q quiet operation
-c add one-line comments
-@ read names from stdin
-x exclude the following names
-F fix zipfile (-FF try harder)
-A adjust self-extracting exe
-T test zipfile integrity
-y store symbolic links as the link instead of the referenced file
-R PKZIP recursion (see manual)
-e encrypt
$
-l convert LF to CR LF
-9 compress better
-v verbose operation
-z add zipfile comment
-o make file as old as latest entry
-i include only the following names
-D do not add directory entries
-J junk zipfile prefix (unzipsfx)
-X eXclude eXtra file attributes
-n don't compress these suffixes

```

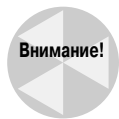
Сильной стороной программы `zip` является ее способность упаковывать целые каталоги с файлами в отдельный сжатый файл. Благодаря этому данная программа становится идеальной для архивирования отдельных структур каталогов:

```

$ zip -r testzip test
adding: test/ (stored 0%)
adding: test/test1/ (stored 0%)
adding: test/test1/myprog2 (stored 0%)
adding: test/test1/myprog1 (stored 0%)
adding: test/myprog.c (deflated 39%)
adding: test/file3 (deflated 2%)
adding: test/file4 (stored 0%)
adding: test/test2/ (stored 0%)
adding: test/file1.gz (stored 0%)
adding: test/file2 (deflated 4%)
adding: test/myprog.gz (stored 0%)
$

```

В этом примере создается файл `zip` с именем `testzip.zip`, после чего проводится рекурсивная обработка всего каталога `test` с добавлением каждого найденного файла и каталога в файл `zip`. В полученных результатах заслуживает внимания то, что не все файлы удалось упаковать в файл `zip`. Программа `zip` автоматически определяет наилучший тип сжатия, применимый для каждого отдельного файла.



Если в команде `zip` используется средство рекурсивной обработки, то файлы сохраняются в файл `zip` в виде исходной структуры каталогов. Файлы, находящиеся в подкаталогах, сохраняются в файл `zip` в таких же подкаталогах. При извлечении файлов необходимо соблюдать осторожность; дело в том, что команда `unzip` восстанавливает всю структуру каталогов в новом месте. Иногда это приводит к путанице, если и без этого на диске имеется много подкаталогов и файлов.

Архивирование данных

Безусловно, команда `zip` превосходно подходит для сжатия и архивирования данных в виде отдельного файла, но сама программа `zip` в мире Unix и Linux не рассматривается как стандартная. В системах Unix и Linux в качестве инструмента архивирования гораздо более широко применяется команда `tar`.

Команда `tar` первоначально использовалась для записи файлов на ленточное устройство в целях архивирования. Но эта команда позволяет также записывать вывод в файл, и этот способ стал широко применяться для архивирования данных в Linux.

Формат команды `tar` приведен ниже.

```
tar function [options] object1 object2 ...
```

Параметры команды `tar` определяют функции, которые должны быть выполнены этой командой, как показано в табл. 4.9.

Таблица 4.9. Функции команды `tar`

Функция	Параметр в длинном формате	Описание
-A	--concatenate	Присоединить существующий файл архива <code>tar</code> к другому существующему файлу архива <code>tar</code>
-c	--create	Создать новый файл архива <code>tar</code>
-d	--diff	Проверить различия между файлом архива <code>tar</code> и файловой системой
	--delete	Провести удаление из существующего <code>tar</code> -файла архива
-r	--append	Присоединить файлы к концу существующего файла архива <code>tar</code>
-t	--list	Сформировать список содержимого существующего <code>tar</code> -файла архива
-u	--update	Присоединить к существующему <code>tar</code> -файлу архива файлы, более новые по сравнению с файлами с теми же именами в существующем архиве
-x	--extract	Извлечь файлы из существующего файла архива

При определении каждой функции используются *параметры*, которые регламентируют выполнение конкретных операций с `tar`-файлом архива. В табл. 4.10 перечислены общие параметры, применяемые с командой `tar`.

Таблица 4.10. Параметры команды `tar`

Параметр	Описание
-C <i>dir</i>	Перейти в указанный каталог
-f <i>file</i>	Вывести результаты в файл (или на устройство) <i>file</i>
-j	Перенаправить вывод в команду <code>bzip2</code> для сжатия
-p	Сохранить все права доступа к файлу
-v	Выводить имена файлов по мере их обработки
-z	Перенаправить вывод в команду <code>gzip</code> для сжатия

Эти параметры обычно объединяются для создания следующих сценариев. Прежде всего необходимо создать файл архива с использованием следующей команды:

```
tar -cvf test.tar test/ test2/
```

С помощью приведенной выше команды был создан файл архива с именем `test.tar`, который включает содержимое двух каталогов, `test` и `test2`. После этого команда

```
tar -tf test.tar
```

формирует список содержимого `tar`-файла `test.tar` (но не извлекает его). Наконец, команда

```
tar -xvf test.tar
```


извлекает содержимое tar-файла `test.tar`. Если tar-файл был создан исходя из определенной структуры каталогов, то воссоздается вся эта структура каталогов, начиная с текущего каталога.

Вполне очевидно, что использование команды `tar` может стать основой простого способа создания файлов архива на основе целого ряда структур каталогов. Поэтому команда `tar` в мире Linux стала основой общепринятого метода распространения файлов исходного кода для приложений с открытым исходным кодом.



После загрузки программного обеспечения с открытым исходным кодом часто обнаруживается, что имена полученных файлов оканчиваются расширением `.tgz`. Это — tar-файлы, сжатые программой `gzip`, которые могут быть извлечены с помощью команды `tar -zxvf filename.tgz`.

Резюме

В настоящей главе приведено описание некоторых более сложных команд `bash`, используемых системными администраторами и программистами в Linux. Важным средством определения статуса системы являются команды `ps` и `top`, которые позволяют получить сведения о том, какие приложения работают в системе и какой объем ресурсов они расходуют.

В наши дни широко применяются сменные носители информации, поэтому для системных администраторов одним из направлений деятельности стало монтирование и размонтирование носителей на запоминающих устройствах. Команда `mount` позволяет смонтировать физическое запоминающее устройство для включения содержимого установленного в нем носителя в структуру виртуального каталога Linux. Для удаления устройства из системы используется команда `umount`.

Наконец, в этой главе приведено описание различных программ, применяемых для обработки данных. Программа `sort` позволяет легко отсортировать большие файлы данных, что способствует лучшей организации данных, а программа `grep` дает возможность быстро просматривать крупные объемы данных при поиске конкретной информации. В системе Linux предусмотрено несколько различных средств сжатия файлов, включая программы `bzip2`, `gzip` и `zip`. Каждая из этих программ позволяет упаковывать большие файлы, что способствует экономии пространства в файловой системе. Программа `tar` в системе Linux широко применяется для архивирования структур каталогов с сохранением в отдельном файле, который может быть легко перенесен в другую систему.

В следующей главе рассматриваются переменные среды Linux. Переменные среды позволяют получить доступ к информации о системе из сценариев, а также предоставляют удобный способ сохранения данных в сценариях.

ГЛАВА

5

В этой главе...

Общее описание переменных среды

Задание переменных среды

Удаление переменных среды

Заданные по умолчанию переменные среды командного интерпретатора

Задание переменной среды PATH

Поиск системных переменных среды

Массивы переменных

Использование псевдонимов команд

Резюме

Использование переменных среды Linux

Переменные среды Linux помогают должным образом определить пользовательский интерфейс командного интерпретатора Linux. Но начинающие пользователи Linux могут столкнуться с трудностями при их освоении. Переменные среды используются во многих программах и сценариях для получения сведений о системе, а также сохранения временных данных и информации о конфигурации. Задание значений переменных среды в системе Linux осуществляется во многих программах, и важно знать, каковы эти программы. В настоящей главе кратко описано все, что связано с переменными среды Linux, и показано, как использовать эти переменные и даже как создать свои собственные. Данная глава завершается связанным по теме разделом, в котором описано определение и использование псевдонимов в сеансах командного интерпретатора.

Общее описание переменных среды

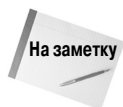
В командном интерпретаторе `bash` предусмотрено средство, называемое *переменными среды*, предназначенное для сохранения информации о сеансе командного интерпретатора и рабочей среде (отсюда происходит название “переменные среды”). Кроме того, это средство позволяет сохранять в памяти данные, к которым может быть легко предоставлен доступ из любых программ или сценариев, вызванных на выполнение из командного интерпретатора.

Это — удобный способ хранения на постоянной основе таких данных, которые уточняют характеристики учетной записи пользователя, системы, командного интерпретатора или любых других объектов, которые должны оставаться доступными для использования.

В командном интерпретаторе `bash` предусмотрено применение переменных среды двух типов:

- глобальные переменные;
- локальные переменные.

В этом разделе описаны переменные среды обоих типов и показано, как просматривать и использовать их значения.



Безусловно, в командном интерпретаторе `bash` предусмотрено применение единого набора переменных среды, но в тех или иных дистрибутивах Linux часто можно обнаружить дополнительные, собственные переменные среды. Примеры переменных среды, приведенные в настоящей главе, могут быть немного иными, в зависимости от конкретного дистрибутива. Если вам придется столкнуться с переменными среды, не описанными в данной главе, обратитесь к документации по применяемому дистрибутиву Linux.

Глобальные переменные среды

Доступ к глобальным переменным среды может быть получен из сеанса командного интерпретатора и из любых дочерних процессов, запущенных командным интерпретатором. С другой стороны, локальные переменные доступны только в том сеансе работы с командным интерпретатором, в котором они созданы. Поэтому глобальные переменные среды находят свое применение в приложениях, запускающих дочерние процессы, для которых требуется информация от родительского процесса.

Система Linux задает несколько глобальных переменных среды при запуске каждого сеанса `bash` (дополнительные сведения о том, какие переменные при этом определяются, приведены в разделе “Поиск системных переменных среды” ниже в этой главе). Имена системных переменных среды всегда состоят полностью из прописных букв, что позволяет проводить различие между ними и обычными пользовательскими переменными среды.

Для просмотра значений глобальных переменных среды используется команда `printenv`:

```
$ printenv
ORBIT_SOCKETDIR=/tmp/orbit-user
HOSTNAME=localhost.localdomain
IMSETTINGS_INTEGRATE_DESKTOP=yes
TERM=xterm
SHELL=/bin/bash
XDG_SESSION_COOKIE=787b3cf537971ef8462260960000006b-1284670942.
440386-1012435051
HISTSIZE=1000
GTK_RC_FILES=/etc/gtk/gtkrc:/home/user/.gtkrc-1.2-gnome2
WINDOWID=29360131
GNOME_KEYRING_CONTROL=/tmp/keyring-Egjauk
IMSETTINGS_MODULE=none
USER=user
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:
bd=40;33;01:cd=40;33;01:or=40;31;01:mi=01;05;37;41:su=37;41:sg=30;43
...
```

```

SSH_AUTH_SOCK=/tmp/keyring-Egjauk/ssh
SESSION_MANAGER=local/unix:@/tmp/.ICE-unix/1331,unix/unix:
🖱️/tmp/.ICE-unix/1331
USERNAME=user
DESKTOP_SESSION=gnome
MAIL=/var/spool/mail/user
PATH=/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/bin:/usr/bin:
/bin:/usr/local/sbin:/usr/sbin:/sbin:/home/user/bin
QT_IM_MODULE=xim
PWD=/home/user/Documents
XMODIFIERS=@im=none
GDM_KEYBOARD_LAYOUT=us
LANG=en_US.utf8
GNOME_KEYRING_PID=1324
GDM_LANG=en_US.utf8
GDMSESSION=gnome
SSH_ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass
HISTCONTROL=ignoredups
HOME=/home/user
SHLVL=2
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
LOGNAME=user
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-
ifKtHnDKnU,guid=6f061c07be822c134f12956b00000081
LESSOPEN=|/usr/bin/lesspipe.sh %s
WINDOWPATH=1
DISPLAY=:0.0
G_BROKEN_FILENAMES=1
XAUTHORITY=/var/run/gdm/auth-for-user-GIU7sE/database
COLORTERM=gnome-terminal
_=/usr/bin/printenv
OLDPWD=/home/user
$

```

Вполне очевидно, что количество глобальных переменных среды, установленных для командного интерпретатора `bash`, весьма велико. Большинство из них устанавливается системой в течение процесса регистрации.

Для отображения значений отдельных переменных среды предназначена команда `echo`. При ссылке на переменную среды необходимо помещать знак доллара перед именем переменной среды:

```

$ echo $HOME
/home/user
$

```

Как уже было сказано, глобальные переменные среды доступны также в дочерних процессах, работающих под управлением текущего сеанса командного интерпретатора:

```

$ bash
$ echo $HOME
/home/user
$

```

В этом примере после запуска нового экземпляра командного интерпретатора с использованием команды `bash` отображается текущее значение переменной среды `HOME`, которую задает система при регистрации пользователя в основном интерфейсе командного интерпретатора. Вполне очевидно, что это значение доступно также в дочернем процессе командного интерпретатора.

Локальные переменные среды

Локальные переменные среды, как следует из их названия, доступны только в локальном процессе, в котором они определены. Но из того, что эти переменные именуются локальными, не следует, что они менее важны, чем глобальные переменные среды. Фактически система Linux от имени пользователя определяет также по умолчанию определенный набор стандартных локальных переменных среды.

Задача просмотра списка локальных переменных среды является довольно сложной. К сожалению, нет такой команды, которая отображала бы только локальные переменные среды. Команда `set` отображает весь набор переменных среды, относящихся к конкретному процессу. Но при этом отображаются также глобальные переменные среды.

Ниже приведен типичный пример вывода команды `set`.

```
$ set
BASH=/bin/bash
BASHOPTS=checkwinsize:cmdhist:expand_aliases:extquote:
force_ignore:hostcomplete:interactive_comments:
progcomp:promptvars:sourcepath
BASH_ALIASES=()
BASH_ARGC=()
BASH_ARGV=()
BASH_CMDS=()
BASH_LINENO=()
BASH_SOURCE=()
BASH_VERSIONINFO=([0]="4" [1]="1" [2]="7" [3]="1"
[4]="release" [5]="i386-redhat-linux-gnu")
BASH_VERSION='4.1.7(1)-release'
COLORS=/etc/DIR_COLORS
COLORTERM=gnome-terminal
COLUMNS=80S
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-
ifKtHnDKnU,guid=6f061c07be822c134f12956b00000081
DESKTOP_SESSION=gnome
DIRSTACK=()
DISPLAY=:0.0
EUID=500
GDMSESSION=gnome
GDM_KEYBOARD_LAYOUT=us
GDM_LANG=en_US.utf8
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
GNOME_KEYRING_CONTROL=/tmp/keyring-Egjauk
GNOME_KEYRING_PID=1324
GROUPS=()
GTK_RC_FILES=/etc/gtk/gtkrc:/home/user/.gtkrc-1.2-gnome2
G_BROKEN_FILENAMES=1
```

```

HISTCONTROL=ignoredups
HISTFILE=/home/user/.bash_history
HISTFILESIZE=1000
HISTSIZE=1000
HOME=/home/user
HOSTNAME=localhost.localdomain
HOSTTYPE=i386
IFS=$' \t\n'
IMSETTINGS_INTEGRATE_DESKTOP=yes
IMSETTINGS_MODULE=none
LANG=en_US.utf8
LESSOPEN='|/usr/bin/lesspipe.sh %s'
LINES=24
LOGNAME=user
LS_COLORS='rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:
bd=40;33;01:cd=40;33;01:or=40;31;01:mi=01;05;37;41:su=37;41:sg=30;43
...
MACHTYPE=i386-redhat-linux-gnu
MAIL=/var/spool/mail/user
MAILCHECK=60
OLDPWD=/home/user
OPTERR=1
OPTIND=1
ORBIT_SOCKETDIR=/tmp/orbit-user
OSTYPE=linux-gnu
PATH=/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/bin:/usr/bin:
/bin:/usr/local/sbin:/usr/sbin:/sbin:/home/user/bin
PIPESTATUS=( [0]="0")
PPID=1674
PROMPT_COMMAND='echo -ne "\033]0;${USER}@${HOSTNAME%%.*}:
${PWD/#$HOME/}"; echo -ne "\007"'
PS1='[\u@\h \W]\$ '
PS2='> '
PS4='+ '
PWD=/home/user/Documents
QT_IM_MODULE=xim
SESSION_MANAGER=local/unix:@/tmp/.ICE-unix/1331,unix/unix:
/tmp/.ICE-unix/1331
SHELL=/bin/bash
SHELLOPTS=braceexpand:emacs:hashall:histexpand:history:interactive-
comments:monitor
SHLVL=2
SSH_ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass
SSH_AUTH_SOCK=/tmp/keyring-Egjauk/ssh
TERM=xterm
UID=500
USER=user
USERNAME=user
WINDOWID=29360131
WINDOWPATH=1

```

```

XAUTHORITY=/var/run/gdm/auth-for-user-GIU7sE/database
XDG_SESSION_COOKIE=787b3cf537971ef8462260960000006b-1284670942.
440386-1012435051
XMODIFIERS=@im=none
_printenv
colors=/etc/DIR_COLORS
__udisks ()
{
    local IFS='
';
    local cur="${COMP_WORDS[COMP_CWORD]}";
    if [ "${COMP_WORDS[${COMP_CWORD} - 1]}" = "--show-info" ]; then
        COMPREPLY=( $(compgen -W "$(udisks --enumerate-device-files)"
-- $cur) );
    else
        if [ "${COMP_WORDS[${COMP_CWORD} - 1]}" =
"--inhibit-polling" ]; then
        ...
            fi;
        fi;
    fi;
fi
}
command_not_found_handle ()
{
    runcnf=1;
    retval=127;
    [ ! -S /var/run/dbus/system_bus_socket ] && runcnf=0;
    [ ! -x /usr/sbin/packagekitd ] && runcnf=0;
    if [ $runcnf -eq 1 ]; then
        /usr/libexec/pk-command-not-found $1;
        retval=$?;
    else
        echo "bash: $1: command not found";
    fi;
    return $retval
}

$

```

Необходимо отметить, что все глобальные переменные среды, отображаемые с помощью команды `printenv`, появляются также в выводе команды `set`. Однако последняя команда выводит довольно большое количество дополнительных переменных среды. Это и есть локальные переменные среды.

Задание переменных среды

Предусмотрена возможность задавать собственные переменные среды непосредственно из командного интерпретатора `bash`. В настоящем разделе показано, как создавать собственные

переменные среды и ссылаться на них из интерактивного интерфейса командного интерпретатора или из сценария командного интерпретатора.

Задание локальных переменных среды

После запуска командного интерпретатора `bash` (или вызова на выполнение сценария командного интерпретатора) появляется возможность создавать локальные переменные, доступ к которым может быть получен в каждом процессе, работающем под управлением командного интерпретатора. Предусмотрена возможность присвоить переменной среды числовое или строковое значение, используя оператор присваивания значения переменной, представляющий собой знак равенства, `"="`:

```
$ test=testing
$ echo $test
testing
$
```

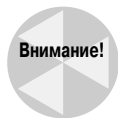
Очевидно, что это весьма несложно! После этого в любое время, когда потребуется обратиться к значению переменной среды `test`, достаточно сослаться на эту переменную по имени `$test`.

Если потребуется присвоить переменной среды строковое значение, содержащее пробелы, то необходимо применить одинарные кавычки для обозначения начала и конца строки:

```
$ test=testing a long string
-bash: a: command not found
$ test='testing a long string'
$ echo $test
testing a long string
$
```

Если бы не было этих одинарных кавычек, то командный интерпретатор `bash` принял бы предположение, что следующая буква после пробела является началом очередной команды, подлежащей обработке. Необходимо учитывать, что для составления имен определяемых локальных переменных среды должны использоваться строчные буквы, тогда как во всех системных переменных среды, включая уже описанные в этой главе, применяются прописные буквы.

Это — стандартное соглашение в командном интерпретаторе `bash`. При создании новых переменных среды рекомендуется (но не требуется) использовать строчные буквы. Это поможет отличить свои собственные переменные среды от заранее заданных системных переменных среды.



Исключительно важным требованием при определении переменной среды является то, что между именем переменной, знаком равенства, `"="`, и значением не должно быть пробелов. Если в операторе присваивания значения переменной присутствуют какие-либо пробелы, то командный интерпретатор `bash` трактует значение как отдельную команду:

```
$ test2 = test
-bash: test2: command not found
$
```

Сразу после задания локальной переменной среды она становится доступной для использования во всех процессах командного интерпретатора. Но если в этом командном интерпретаторе будет запущен другой командный интерпретатор, то в дочернем командном интерпретаторе эти локальные переменные среды окажутся недоступными:


```
$ bash
$ echo $test

$ exit
exit
$ echo $test
testing a long string
$
```

В настоящем примере показано, что был выполнен запуск дочернего командного интерпретатора. Вполне очевидно, что переменная среды `test` недоступна в дочернем командном интерпретаторе (попытка ее вывода приводит к получению пустого значения). А после завершения работы с дочерним командным интерпретатором и возврата к исходному командному интерпретатору обнаруживается, что эта локальная переменная среды по-прежнему доступна.

Аналогичным образом, если локальная переменная среды будет задана в дочернем процессе, то сразу после завершения дочернего процесса эта локальная переменная среды станет недоступной:

```
$ bash
$ test=testing
$ echo $test
testing
$ exit
exit
$ echo $test

$
```

После возврата в родительский командный интерпретатор переменная среды `test`, заданная в дочернем командном интерпретаторе, больше не существует.

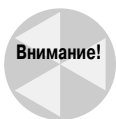
Задание глобальных переменных среды

Любая глобальная переменная среды является доступной во всех дочерних процессах, запущенных процессом, в котором задана глобальная переменная среды. Для создания глобальной переменной среды применяется способ, состоящий в создании локальной переменной среды и последующем экспорте ее в глобальную среду.

Последняя операция выполняется с использованием команды `export`:

```
$ echo $test
testing a long string
$ export test
$ bash
$ echo $test
testing a long string
$
```

После экспорта локальной переменной среды `test` был запущен дочерний процесс командного интерпретатора, в котором выведено значение переменной среды `test`. На этот раз переменная среды со своим значением оказалась доступной, поскольку была преобразована в глобальную с помощью команды `export`.



Следует учитывать, что при экспорте локальной переменной среды не следует использовать знак доллара для ссылки на имя переменной.

Удаление переменных среды

Безусловно, если имеется возможность создать новую переменную среды, необходимо также предусмотреть возможность удаления существующей переменной среды. Это можно выполнить с помощью команды `unset`:

```
$ echo $test
testing
$ unset test
$ echo $test

$
```

Ссылаясь на переменную среды в команде `unset`, следует помнить, что в этой команде знак доллара не должен использоваться.

Задача удаления глобальных переменных среды немного сложнее. Если сброс значения глобальной переменной среды с помощью команды `unset` осуществляется в дочернем процессе, то результат выполнения этой операции обнаруживается только в дочернем процессе. Эта глобальная переменная среды все еще остается доступной в родительском процессе:

```
$ test=testing
$ export test
$ bash
$ echo $test
testing
$ unset test
$ echo $test

$ exit
exit
$ echo $test
testing
$
```

В рассматриваемом примере задана локальная переменная среды с именем `test`, после чего произведен экспорт этой переменной в целях ее преобразования в глобальную переменную среды. После этого запускается дочерний процесс командного интерпретатора и проверяется, остается ли доступной глобальная переменная среды `test`. Затем, в то время как все еще продолжается работа в дочернем командном интерпретаторе, происходит вызов команды `unset` для удаления глобальной переменной среды `test`, после чего осуществляется выход из дочернего командного интерпретатора. После возврата в исходный родительский командный интерпретатор проводится проверка значения переменной среды `test`, которая показывает, что эта переменная по-прежнему остается действительной.

Заданные по умолчанию переменные среды командного интерпретатора

Ряд конкретных переменных среды командный интерпретатор `bash` использует по умолчанию для определения системной среды. Можно всегда рассчитывать на то, что эти переменные установлены в применяемой системе Linux. Кроме того, командный интерпретатор `bash` происходит от созданного ранее командного интерпретатора Unix Bourne, поэтому включает переменные среды, первоначально определенные в этом командном интерпретаторе.

В табл. 5.1 показаны переменные среды, предоставляемые командным интерпретатором `bash`, которые совместимы с оригинальным командным интерпретатором Unix Bourne.

Таблица 5.1. Переменные командного интерпретатора `bash`, совместимые с Bourne

Переменная	Описание
<code>CDPATH</code>	Разделенный двоеточиями список каталогов, используемый в качестве пути поиска файлов для команды <code>cd</code>
<code>HOME</code>	Исходный каталог текущего пользователя
<code>IFS</code>	Список символов, предназначенных для использования в качестве разделителей полей, которые применяются командным интерпретатором для разбиения текстовых строк
<code>MAIL</code>	Имя файла для почтового ящика текущего пользователя. Командный интерпретатор <code>bash</code> проверяет этот файл для определения наличия вновь поступившей почты
<code>MAILPATH</code>	Разделенный двоеточиями список из нескольких имен файлов, предназначенных для использования в качестве почтового ящика текущего пользователя. Командный интерпретатор <code>bash</code> проверяет каждый файл из этого списка для определения наличия вновь поступившей почты
<code>OPTARG</code>	Значение параметра последнего аргумента, обработанного командой <code>getopts</code>
<code>OPTIND</code>	Значение индекса параметра последнего аргумента, обработанного командой <code>getopts</code>
<code>PATH</code>	Разделенный двоеточиями список каталогов, в котором командный интерпретатор ведет поиск двоичных файлов команд
<code>PS1</code>	Строка приглашения первичного интерфейса командной строки командного интерпретатора
<code>PS2</code>	Строка приглашения вторичного интерфейса командной строки командного интерпретатора

Безусловно, наиболее важной переменной среды в этом списке является `PATH`. После ввода текста команды в интерфейсе командной строки командного интерпретатора должен быть проведен поиск в системе программы, соответствующей этой команде. Переменная среды `PATH` определяет каталоги, в которых должен осуществляться поиск команд. Один из примеров переменной среды `PATH` в системе Linux может выглядеть следующим образом:

```
$ echo $PATH
/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/bin:/usr/bin:
/bin:/usr/local/sbin:/usr/sbin:/sbin:/home/user/bin

$
```

Этот пример показывает, что командный интерпретатор проводит поиск команд в шести каталогах. Каталоги в переменной среды `PATH` разделены двоеточиями. В переменной `PATH` отсутствует какое-либо обозначение конца списка каталогов. Поэтому можно добавлять дополнительные каталоги к списку `PATH`, просто вводя еще одно двоеточие и указывая новый

каталог. Кроме того, переменная среды `PATH` показывает, в каком порядке должен проводиться поиск команд.

Кроме применяемых по умолчанию переменных среды `Bourne`, командный интерпретатор `bash` предоставляет несколько собственных переменных, как показано в табл. 5.2.

Таблица 5.2. Переменные среды командного интерпретатора `bash`

<i>Переменная</i>	<i>Описание</i>
<code>BASH</code>	Полное имя пути, применяемое при исполнении программы текущего экземпляра командного интерпретатора <code>bash</code>
<code>BASH_ALIASES</code>	Ассоциативный массив псевдонимов, заданных в настоящее время
<code>BASH_ARGC</code>	Массив переменных, содержащий ряд параметров, передаваемых в подпрограмму или сценарий командного интерпретатора
<code>BASH_ARCV</code>	Массив переменных, который содержит параметры, передаваемые в подпрограмму или сценарий командного интерпретатора
<code>BASH_CMDS</code>	Ассоциативный массив с указанием местоположений команд, выполняемых командным интерпретатором
<code>BASH_COMMAND</code>	Команда командного интерпретатора, которая в настоящее время выполняется или должна быть выполнена
<code>BASH_ENV</code>	Если значение переменной <code>BASH_ENV</code> задано, то в каждом сценарии <code>bash</code> предпринимается попытка выполнить файл запуска, определенный этой переменной, перед переходом к дальнейшей работе
<code>BASH_EXECUTION_STRING</code>	Команды, переданные с использованием параметра <code>bash -c</code>
<code>BASH_LINENO</code>	Массив переменных, содержащий номер строки исходного кода функции командного интерпретатора, выполняющейся в настоящее время
<code>BASH_REMATCH</code>	Предназначенный только для чтения массив переменных, содержащий шаблоны и относящиеся к ним вспомогательные шаблоны для положительных согласований с использованием оператора сравнения регулярных выражений (<code>=</code>)
<code>BASH_SOURCE</code>	Массив переменных, содержащий имя файла исходного кода функции командного интерпретатора, выполняющейся в настоящее время
<code>BASH_SUBSHELL</code>	Текущий уровень вложенности дочерней среды командного интерпретатора. Начальное значение равно 0
<code>BASH_VERSION</code>	Номер версии текущего экземпляра командного интерпретатора <code>bash</code>
<code>BASH_VERSINFO</code>	Массив переменных, который содержит отдельные номера версий текущего экземпляра командного интерпретатора <code>bash</code> — старший номер версии и младший номер версии
<code>BASH_XTRACEFD</code>	Параметр <code>BASH_XTRACEFD</code> , будучи установленным равным дескриптору действительного файла (0, 1, 2), включает перенаправление вывода трассировки, сформированного с помощью параметра отладки <code>"set -x"</code> . Такая возможность часто используется для отправки вывода трассировки в отдельный файл
<code>BASHOPTS</code>	Список параметров командного интерпретатора <code>bash</code> , которые включены в настоящее время
<code>BASHPID</code>	Идентификатор процесса текущего процесса <code>bash</code>
<code>COLUMNS</code>	Параметр <code>COLUMNS</code> содержит значение ширины окна терминала, используемого для текущего экземпляра командного интерпретатора <code>bash</code>
<code>COMP_CWORD</code>	Индекс в переменной <code>COMP_WORDS</code> , который содержит значение текущей позиции курсора

Переменная	Описание
COMP_LINE	Текущая командная строка
COMP_POINT	Индекс текущей позиции курсора относительно начала текущей команды
COMP_KEY	Последнее нажатие клавиши, которое использовалось для вызова текущего завершения функции командного интерпретатора
COMP_TYPE	Целочисленное значение, представляющее тип предпринятой попытки завершения, которая привела к вызову функции завершения командного интерпретатора
COMP_WORDBREAKS	Символы разделителей слов библиотеки Readline, применяемые для завершения вводимых слов
COMP_WORDS	Массив переменных, содержащий отдельные слова из текущей командной строки
COMPREPLY	Массив переменных, который содержит возможные коды завершения, сформированные функцией командного интерпретатора
DIRSTACK	Массив переменных, который содержит текущее содержимое стека каталогов
EMACS	Параметр EMACS, будучи заданным равным "t", указывает, что выполняется команда из буфера командного интерпретатора emacs и редактирование строк отключено
EUID	Действительный числовой идентификатор текущего пользователя
FCEDIT	Заданный по умолчанию редактор, используемый командой <code>fc</code>
FIGNONE	Разделенный двоеточиями список суффиксов, которые следует пропускать при выполнении операции завершения имени файла
FUNCNAME	Имя выполняющейся в настоящее время функции командного интерпретатора
GLOBIGNORE	Разделенный двоеточиями список шаблонов, определяющих набор имен файлов, которые следует пропускать при дополнении имени файла
GROUPS	Массив переменных, содержащий список групп, членом которых является текущий пользователь
histchars	Параметр, содержащий до трех символов, которые управляют разворачиванием журнала
HISTCMD	Номер текущей команды в журнале
HISTCONTROL	Параметр, управляющий тем, какие команды должны быть введены в список журнала командного интерпретатора
HISTFILE	Имя файла, предназначенного для сохранения списка журнала командного интерпретатора (по умолчанию — <code>.bash_history</code>)
HISTFILESIZE	Максимальное количество строк, предназначенных для сохранения в файле журнала
HISTIGNORE	Разделенный двоеточиями список шаблонов, используемый для принятия решения о том, какие команды пропускаются при включении в файл журнала
HISTSIZE	Максимальное количество команд, хранимых в файле журнала
HOSTFILE	Параметр, содержащий имя файла, который должен быть считан при необходимости дополнения в командном интерпретаторе имени хоста в процессе его ввода
HOSTNAME	Имя текущего хоста
HOSTTYPE	Строка, описывающая компьютер, на котором работает командный интерпретатор <code>bash</code>

Переменная	Описание
IGNOREEOF	Количество последовательных символов признака конца файла EOF, которые должны быть получены командным интерпретатором для завершения работы. Если этот параметр не задан, то по умолчанию применяется значение 1
INPUTRC	Имя файла инициализации Readline (значение по умолчанию — <code>.inputrc</code>)
LANG	Категория региональной установки для командного интерпретатора
LC_ALL	Параметр, переопределяющий переменную <code>LANG</code> , которая задает категорию региональной установки
LC_COLLATE	Параметр, который задает порядок сортировки, используемый при сортировке строковых значений
LC_CTYPE	Параметр, определяющий интерпретацию символов, используемых в расширениях имен файлов и в шаблонах согласования
LC_MESSAGES	Параметр, который определяет региональную установку, используемую при интерпретации строк в двойных кавычках, которым предшествует знак доллара
LC_NUMERIC	Параметр, определяющий региональную установку, которая используется при форматировании чисел
LINENO	Номер строки в сценарии, выполняющемся в настоящее время
LINES	Параметр, который определяет количество строк, имеющихся на терминале
MACHTYPE	Строка, которая задает тип системы в формате “система–компания–ЦП”
MAILCHECK	Параметр, от которого зависит, с какой периодичностью (в секундах) командный интерпретатор должен проводить проверку на наличие новой почты (значение по умолчанию — 60)
OLDPWD	Предыдущий рабочий каталог, который использовался в командном интерпретаторе
OPTERR	Если для параметра <code>OPTERR</code> задано значение 1, командный интерпретатор <code>bash</code> отображает сообщения об ошибках, сформированные командой <code>getopts</code>
OSTYPE	Строка, определяющая операционную систему, в которой работает командный интерпретатор
PIPESTATUS	Массив переменных, содержащий список значений статуса выхода из процессов в данном процессе переднего плана
POSIXLY_CORRECT	Если этот параметр задан, командный интерпретатор <code>bash</code> начинает свою работу в режиме POSIX
PPID	Идентификатор родительского процесса командного интерпретатора <code>bash</code>
PROMPT_COMMAND	Если этот параметр задан, он определяет команду, выполняемую перед отображением первичного приглашения к вводу информации
PROMPT_DIRTRIM	Целое число, используемое для указания количества отображаемых заключительных имен каталогов при использовании строковых экранирующих кодов приглашения <code>\w</code> и <code>\W</code> . Удаленные имена каталогов заменяются одним набором многоточий
PS3	Приглашение к вводу информации, используемое для команды <code>select</code>
PS4	Приглашение к вводу информации, отображаемое перед повтором командной строки, если используется параметр <code>bash -x</code>
PWD	Текущий рабочий каталог

Переменная	Описание
RANDOM	Параметр, который возвращает случайное число от 0 до 32767. Присваивание значения этой переменной равносильно заданию начального значения для генератора случайных чисел
REPLY	Заданная по умолчанию переменная для команды <code>read</code>
SECONDS	Продолжительность времени (в секундах), истекшего с момента запуска командного интерпретатора. Присваивание значения этому параметру приводит к переустановке таймера в это значение
SHELL	Полное имя пути к командному интерпретатору <code>bash</code>
SHELLOPTS	Разделенный двоеточиями список включенных параметров командного интерпретатора <code>bash</code>
SHLVL	Параметр, который указывает уровень командного интерпретатора, увеличивающийся на единицу после каждого запуска нового командного интерпретатора <code>bash</code>
TIMEFORMAT	Формат, указывающий способ отображения значений времени в командном интерпретаторе
TMOUТ	Значение, определяющее продолжительность времени (в секундах) ожидания ввода командами <code>select</code> и <code>read</code> . По умолчанию предусмотрено значение нуль, которое задает неопределенно долгое время ожидания
TMPDIR	Имя каталога, в котором командный интерпретатор <code>bash</code> создает временные файлы для собственного использования
UID	Реальный числовой идентификатор текущего пользователя

Заслуживает внимания то, что при использовании команды `set` отображаются не все заданные по умолчанию переменные среды. Причина этого состоит в том, что, несмотря на наличие этих заданных по умолчанию переменных среды, не все они обязаны содержать значение.

Задание переменной среды PATH

Создается впечатление, будто переменная среды `PATH` вызывает наибольшее количество проблем при организации работы систем Linux. Эта переменная определяет, в каких каталогах командный интерпретатор ищет команды, введенные в командной строке. Если поиск команды оканчивается неудачей, командный интерпретатор отображает следующее сообщение об ошибке:

```
$ myprog
-bash: myprog: command not found
$
```

Проблема состоит в том, что устанавливаемые приложения часто помещают свои исполняемые программы в каталогах, не указанных в переменной среды `PATH`. Важная задача системного администратора состоит в обеспечении того, чтобы применяемая переменная среды `PATH` включала все каталоги, в которых находятся развернутые приложения.

Предусмотрена возможность добавлять новые каталоги поиска к существующей переменной среды `PATH` без переопределения содержимого этой переменной с нуля. Отдельные каталоги, перечисленные в переменной среды `PATH`, разделены двоеточиями. Чтобы откорректировать значение этой переменной, достаточно лишь задать ссылку на исходное значение `PATH` и добавить к строке все необходимые новые каталоги. Соответствующая последовательность действий выглядит примерно так:

```
$ echo $PATH
/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/bin:/usr/bin:
/bin:/usr/local/sbin:/usr/sbin:/sbin:/home/user/bin

$ PATH=$PATH:/home/user/test
$ echo $PATH
/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/bin:/usr/bin:
/bin:/usr/local/sbin:/usr/sbin:/sbin:/home/user/bin:/home/user/test
$ myprog
The factorial of 5 is 120.
$
```

После добавления каталога к переменной среды `PATH` появляется возможность вызывать на выполнение программу, независимо от того, где находится текущий каталог в структуре виртуального каталога:

```
[user@localhost ]$ cd /etc
[user@localhost etc]$ myprog
The factorial of 5 is 120
[user@localhost etc]$
```

Кроме того, программисты широко применяют один прием, позволяющий вызывать программу из текущего каталога, который заключается в добавлении символа одинарной точки в применяемую переменную среды `PATH`. Символ одинарной точки представляет текущий каталог (см. главу 3):

```
[user@localhost ]$ PATH=$PATH:.
[user@localhost ]$ cd test2
[user@localhost test2]$ myprog2
The factorial of 6 is 720
[user@localhost test2]$
```

В следующем разделе показано, как вносить постоянные изменения в переменные среды, чтобы иметь возможность всегда успешно вызывать на выполнение необходимые программы.

Поиск системных переменных среды

Система Linux использует переменные среды для указания собственных параметров, которые могут потребоваться в программах и сценариях. Благодаря этому предоставляется удобный способ получения системной информации, необходимой для тех или иных программ. Сложность состоит в определении того, как задаются значения этих переменных среды.

После того как пользователь запускает командный интерпретатор `bash`, регистрируясь в системе Linux, применяемый по умолчанию экземпляр `bash` проверяет несколько файлов на наличие в них команд запуска. Эти файлы именуются *файлами запуска*. Состав файлов запуска, обрабатываемых командным интерпретатором `bash`, зависит от применяемого пользователем способа запуска этого командного интерпретатора. Предусмотрены три способа запуска командного интерпретатора `bash`.

- Запуск в качестве заданного по умолчанию командного интерпретатора для входа в систему во время регистрации.
- Запуск в виде интерактивного командного интерпретатора, который не представляет собой командный интерпретатор для входа в систему.

- Запуск в целях применения как неинтерактивного командного интерпретатора для выполнения сценария.

В следующих разделах рассматриваются файлы запуска, выполняемые командным интерпретатором `bash` при использовании каждого из этих способов запуска.

Командный интерпретатор для входа в систему

При регистрации пользователя в системе Linux командный интерпретатор `bash` запускается как предназначенный для входа в систему. Командный интерпретатор для входа в систему ищет четыре различных файла запуска в целях выполнения содержащихся в них команд. Ниже указана последовательность, в который командный интерпретатор `bash` обрабатывает эти файлы.

- `/etc/profile`.
- `$HOME/.bash_profile`.
- `$HOME/.bash_login`.
- `$HOME/.profile`.

Основным применяемым по умолчанию файлом запуска для командного интерпретатора `bash` в системе является файл `/etc/profile`. Этот файл запуска исполняется при регистрации каждого пользователя в системе. Остальные три файла запуска определяются конкретно для каждого пользователя и могут быть настроены в соответствии с требованиями того или иного пользователя. Рассмотрим эти файлы более подробно.

Файл `/etc/profile`

Файл `/etc/profile` — это основной применяемый по умолчанию файл запуска для командного интерпретатора `bash`. Каждый раз при регистрации пользователя в системе Linux командный интерпретатор `bash` выполняет команды, заданные в файле запуска `/etc/profile`. В различных дистрибутивах Linux этот файл содержит разные команды. В рассматриваемой системе Linux он выглядит следующим образом:

```
$ cat /etc/profile
# /etc/profile

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

# It's NOT good idea to change this file unless you know what you
# are doing. Much better way is to create custom.sh shell script in
# /etc/profile.d/ to make custom changes to environment. This will
# prevent need for merging in future updates.

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

```

        fi
    esac
}

if [ -x /usr/bin/id ]; then
    if [ -z "$EUID" ]; then
        # ksh workaround
        EUID='id -u'
        UID='id -ru'

        fi
        USER="'id -un'"
        LOGNAME=$USER
        MAIL="/var/spool/mail/$USER"
    fi

    # Path manipulation
    if [ "$EUID" = "0" ]; then
        pathmunge /sbin
        pathmunge /usr/sbin
        pathmunge /usr/local/sbin
    else
        pathmunge /usr/local/sbin after
        pathmunge /usr/sbin after
        pathmunge /sbin after
    fi

    HOSTNAME='/bin/hostname 2>/dev/null'
    HISTSIZE=1000
    if [ "$HISTCONTROL" = "ignorespace" ] ; then
        export HISTCONTROL=ignoreboth
    else
        export HISTCONTROL=ignoredups
    fi

    export PATH USER LOGNAME MAIL HOSTNAME HISTSIZE HISTCONTROL

    for i in /etc/profile.d/*.sh ; do
        if [ -r "$i" ]; then
            if [ "$PS1" ]; then
                . $i
            else
                . $i >/dev/null 2>&1
            fi
        fi
    done

    unset i
    unset pathmunge
    $

```

Большинство команд и сценариев, которые можно найти в данном файле, рассматриваются более подробно в главе 10. А в данный момент необходимо отметить, что в этом файле запуска

предусмотрено определение переменных среды. В частности, заслуживает внимания строка экспорта переменных в нижней части файла:

```
export PATH USER LOGNAME MAIL HOSTNAME HISTSIZE HISTCONTROL
```

Применение этой строки позволяет обеспечить, чтобы определяемые переменные среды были доступны во всех дочерних процессах, запускаемых из командного интерпретатора входа в систему.

В этом файле профиля используется еще одно сложное средство. В нем задан оператор `for`, который проходит по циклу по всем файлам каталога `/etc/profile.d`. (Операторы `for` описаны более подробно в главе 12.) Каталог `/etc/profile.d` представляет собой то место, в котором система Linux располагает относящиеся к приложениям файлы запуска, выполняемые командным интерпретатором при регистрации пользователя. В рассматриваемой системе Linux в каталоге `profile.d` находятся следующие файлы:

```
$ ls -l /etc/profile.d
total 72
-rw-r--r--. 1 root root 1133 Apr 28 10:43 colorls.csh
-rw-r--r--. 1 root root 1143 Apr 28 10:43 colorls.sh
-rw-r--r--. 1 root root 192 Sep 9 2004 glib2.csh
-rw-r--r--. 1 root root 192 Dec 12 2005 glib2.sh
-rw-r--r--. 1 root root 58 May 31 06:23 gnome-ssh-askpass.csh
-rw-r--r--. 1 root root 70 May 31 06:23 gnome-ssh-askpass.sh
-rw-r--r--. 1 root root 184 Aug 25 11:36 krb5-workstation.csh
-rw-r--r--. 1 root root 57 Aug 25 11:36 krb5-workstation.sh
-rw-r--r--. 1 root root 1741 Jun 24 15:20 lang.csh
-rw-r--r--. 1 root root 2706 Jun 24 15:20 lang.sh
-rw-r--r--. 1 root root 122 Feb 7 2007 less.csh
-rw-r--r--. 1 root root 108 Feb 7 2007 less.sh
-rw-r--r--. 1 root root 837 Sep 2 05:24 PackageKit.sh
-rw-r--r--. 1 root root 2142 Aug 10 16:41 udisks-bash-completion.sh
-rw-r--r--. 1 root root 74 Mar 25 19:24 vim.csh
-rw-r--r--. 1 root root 248 Mar 25 19:24 vim.sh
-rw-r--r--. 1 root root 161 Nov 27 2007 which2.csh
-rw-r--r--. 1 root root 169 Nov 27 2007 which2.sh
$
```

Заслуживает внимания то, что эти файлы главным образом связаны с конкретными приложениями в системе. Для большинства приложений создаются два файла запуска: один из них предназначен для командного интерпретатора `bash` (в нем используется расширение `.sh`), а другой — для командного интерпретатора `C` (он имеет расширение `.csh`).

Файлы `lang.csh` и `lang.sh` предназначены для осуществления попыток определить предусмотренную по умолчанию языковую кодировку, которая используется в системе, и задать соответствующим образом переменную среды `LANG`.

Файлы запуска \$HOME

Все остальные три файла запуска предназначены для осуществления той же функции — для предоставления ориентированных на каждого пользователя файлов запуска, которые служат для определения переменных среды, необходимых для того или иного пользователя. В большинстве дистрибутивов Linux используется только один из этих трех файлов запуска:

- `$HOME/.bash_profile`.

- \$HOME/.bash_login.
- \$HOME/.profile.

Обратите внимание на то, что имена всех этих трех файлов начинаются с точки, что равносильно определению их как скрытых файлов (они не отображаются в обычном листинге команды `ls`). Эти файлы находятся в каталоге `HOME` каждого пользователя, поэтому пользователи имеют возможность редактировать свои файлы и добавлять собственные переменные среды, активизируемые после запуска любого сеанса командного интерпретатора `bash`.

Рассматриваемая система Linux содержит следующий файл `.bash_profile`:

```
$ cat .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
$
```

В этом файле запуска `.bash_profile` вначале проверяется, присутствует ли в каталоге `HOME` еще один файл запуска, который носит имя `.bashrc` (этот файл будет рассматриваться ниже, в разделе “Интерактивный командный интерпретатор”). Если этот файл имеется в каталоге, файл запуска `.bash_profile` выполняет содержащиеся в нем команды. Затем файл запуска добавляет в переменную среды `PATH` каталог, предоставляя общее местоположение для размещения исполняемых файлов в каталоге `HOME`.

Интерактивный командный интерпретатор

Если запуск командного интерпретатора `bash` происходит без регистрации в системе (например, как при вводе отдельно взятой команды `bash` в приглашении интерфейса командной строки), то запускается так называемый *интерактивный командный интерпретатор*. Интерактивный командный интерпретатор действует иначе по сравнению с командным интерпретатором входа в систему, но все равно предоставляет приглашение интерфейса командной строки, в котором можно вводить команды.

Если команда `bash` запускается в качестве интерактивного командного интерпретатора, то не осуществляет обработку файла `/etc/profile`. Вместо этого данная команда проводит проверку на наличие файла `.bashrc` в каталоге `HOME` пользователя.

В рассматриваемом дистрибутиве Linux этот файл выглядит следующим образом:

```
$ cat .bashrc

# .bashrc
# Source global definitions
if [ -f /etc/bashrc ]; then
    . /etc/bashrc
fi
```

```
# User specific aliases and functions
$
```

Файл `.bashrc` обеспечивает выполнение двух функций. Во-первых, с его помощью проводится проверка на наличие общего файла `bashrc` в каталоге `/etc`. Во-вторых, этот файл предоставляет для пользователя место, в котором он может задавать персональные псевдонимы (об этом речь пойдет ниже, в разделе “Использование псевдонимов команд”) и приватные функции сценариев (рассматриваемые в главе 16).

Общий файл запуска `/etc/bashrc` выполняется в системе для всех, кто запускает интерактивный сеанс командного интерпретатора. В рассматриваемом дистрибутиве Linux этот файл выглядит следующим образом:

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

# System wide functions and aliases
# Environment stuff goes in /etc/profile

# It's NOT good idea to change this file unless you know what you
# are doing. Much better way is to create custom.sh shell script in
# /etc/profile.d/ to make custom changes to environment. This will
# prevent need for merging in future updates.

# By default, we want this to get set.
# Even for non-interactive, non-login shells.
# Current threshold for system reserved uid/gids is 200
# You could check uidgid reservation validity in
# /usr/share/doc/setup-*/uidgid file
if [ $UID -gt 199 ] && [ "`id -gn`" = "`id -un`" ]; then
    umask 002
else
    umask 022
fi

# are we an interactive shell?
if [ "$PS1" ]; then
    case $TERM in
        xterm*)
            if [ -e /etc/sysconfig/bash-prompt-xterm ]; then
                PROMPT_COMMAND=/etc/sysconfig/bash-prompt-xterm
            else
                PROMPT_COMMAND='echo -ne
"\033]0;${USER}@${HOSTNAME%.*}:
${PWD/#$HOME/}"; echo -ne "\007"'
            fi
            ;;
        screen)
            if [ -e /etc/sysconfig/bash-prompt-screen ]; then
                PROMPT_COMMAND=/etc/sysconfig/bash-prompt-screen
            else
                PROMPT_COMMAND='echo -ne
"\033_${USER}@${HOSTNAME%.*}:
${PWD/#$HOME/}"; echo -ne "\033\\\"'
```

```

        fi
        ;;
    *)
        [ -e /etc/sysconfig/bash-prompt-default ] &&
        PROMPT_COMMAND=/etc/sysconfig/bash-prompt-default
        ;;
    esac
    # Turn on checkwinsize
    shopt -s checkwinsize
    [ "$PS1" = "\\s-\\v\\\\"$ " ] && PS1="[\\u@\\h \\W]\\\\"$ "
    # You might want to have e.g. tty in prompt (e.g. more
    # virtual machines) and console windows
    # If you want to do so, just add e.g.
    # if [ "$PS1" ]; then
    #     PS1="[\\u@\\h:\\l \\W]\\\\"$ "
    # fi
    # to your custom modification shell script in
    # /etc/profile.d/ directory
fi

if ! shopt -q login_shell ; then # We're not a login shell
    # Need to redefine pathmunge, it get's undefined at the
    # end of /etc/profile
    pathmunge () {
        case "${PATH}:" in
            *:"$1":*)
                ;;
            *)
                if [ "$2" = "after" ] ; then
                    PATH=$PATH:$1
                else
                    PATH=$1:$PATH
                fi
            esac
        }

    # Only display echos from profile.d scripts if we are no login
    # shell and interactive - otherwise just process them to set
    # envvars
    for i in /etc/profile.d/*.sh; do
        if [ -r "$i" ]; then
            if [ "$PS1" ]; then
                . $i
            else
                . $i >/dev/null 2>&1
            fi
        fi
    done

    unset i
    unset pathmunge

```

```
fi
# vim:ts=4:sw=4
$
```

В применяемом по умолчанию файле задается несколько переменных среды, но заслуживает внимания то, что в нем не используется команда `export` для преобразования их в глобальные. Следует помнить, что файл запуска интерактивного командного интерпретатора выполняется каждый раз при запуске нового экземпляра интерактивного командного интерпретатора. Это означает, что в каждом дочернем командном интерпретаторе этот файл запуска интерактивного командного интерпретатора выполняется автоматически.

Кроме того, необходимо отметить, что в файле `/etc/bashrc` выполняются также характерные для приложений файлы запуска, находящиеся в каталоге `/etc/profile.d`.

Неинтерактивный командный интерпретатор

Наконец, командный интерпретатор последнего типа представляет собой неинтерактивный командный интерпретатор. Это — командный интерпретатор, запускаемый системой для выполнения сценариев командного интерпретатора. Его отличительной особенностью является отсутствие приглашения интерфейса командной строки, которое во всех прочих случаях всегда выводится для пользователя. Но так или иначе, может оказаться, что при каждом запуске пользователем сценария в своей системе должны быть выполнены определенные команды запуска.

Чтобы можно было найти применимое решение в этой ситуации, в командном интерпретаторе `bash` предусмотрена переменная среды `BASH_ENV`. При запуске в командном интерпретаторе неинтерактивного процесса командного интерпретатора осуществляется проверка на наличие в этой переменной среды имени файла запуска, который должен быть выполнен. Если это имя задано, командный интерпретатор выполняет команды, указанные в соответствующем файле. В рассматриваемом дистрибутиве Linux это значение переменной среды по умолчанию не задано.

Массивы переменных

Одним из действительно великолепных особенностей переменных среды является то, что они могут использоваться как *массивы*. Массив представляет собой переменную, которая позволяет хранить несколько значений. На эти значения можно ссылаться либо отдельно, либо в целом, как ко всему массиву.

Чтобы задать несколько значений для переменной среды, достаточно заключить эти значения в круглые скобки, разделив их пробелами:

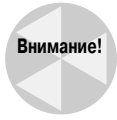
```
$ mytest=(one two three four five)
$
```

Очевидно, что в этом нет ничего особенного. При попытке вывести значение всего массива как обычной переменной среды можно испытать разочарование:

```
$ echo $mytest
one
$
```

Отображается только первое значение в массиве. Чтобы сослаться на отдельный элемент массива, необходимо воспользоваться числовым значением индекса, который представляет место этого элемента в массиве. Это числовое значение должно быть заключено в квадратные скобки:

```
$ echo ${mytest[2]}
three
$
```



Отсчет значений индексов в массивах переменных среды начинается с нуля, что часто приводит к путанице.

Поэтому для отображения всего значения переменной типа массива в качестве значения индекса используется символ-шаблон звездочки:

```
$ echo ${mytest[*]}
one two three four five
$
```

Предусмотрена также возможность изменить значение в отдельной позиции индекса:

```
$ mytest[2]=seven
$ echo ${mytest[*]}
one two seven four five
$
```

Кроме того, можно воспользоваться командой `unset`, чтобы удалить отдельное значение в массиве, но при этом следует соблюдать осторожность, поскольку такая операция удаления становится причиной некоторых сложностей. Рассмотрим следующий пример:

```
$ unset mytest[2]
$ echo ${mytest[*]}
one two four five
$
$ echo ${mytest[2]}

$ echo ${mytest[3]}
four
$
```

В этом примере используется команда `unset` для удаления значения в позиции индекса 2. После вывода массива создается впечатление, что следующие за этой позицией значения индекса просто уменьшились на единицу. Однако если будут отдельно выведены данные из позиции индекса 2, то обнаружится, что это местоположение пусто.

Наконец, можно удалить весь массив, для чего достаточно задать отдельно взятое имя массива в команде `unset`:

```
$ unset mytest
$ echo ${mytest[*]}

$
```

Иногда применение массивов переменных приводит лишь к усложнению работы, поэтому такие массивы используются в программировании сценариев командного интерпретатора не слишком часто. Возникают также трудности при переносе массивов переменных в другие варианты среды командного интерпретатора, а это — важный недостаток, если приходится выполнять большой объем программирования сценариев командного интерпретатора для различных командных интерпретаторов. Массивы используются в нескольких системных пере-

менных среды `bash` (таких как `BASH_VERSION`), однако обычно пользователю не приходится сталкиваться с ними достаточно часто.

Использование псевдонимов команд

С формальной точки зрения псевдонимы команд нельзя считать переменными среды командного интерпретатора, но псевдонимы во многом действуют аналогично переменным. Как правило, *псевдонимы команд* применяются в целях создания псевдонимов для широко применяемых команд (включая параметры этих команд), что позволяет свести к минимуму необходимый объем ввода.

Во многих дистрибутивах Linux чаще всего уже заданы некоторые общие псевдонимы команд от имени пользователя. Для просмотра списка активных псевдонимов применяется команда `alias` с параметром `-p`:

```
$ alias -p
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
alias ls='ls --color=auto'
alias vi='vim'
alias which='alias | /usr/bin/which --tty-only --read-alias --show-dot
--show-tilde'
$
```

Обратите внимание на то, что в рассматриваемом дистрибутиве Linux один из псевдонимов использовался для переопределения стандартной команды `ls`. При этом автоматически предусмотрен параметр `--color`, который указывает, что данный терминал поддерживает выделение вывода цветом.

Предусмотрена возможность создавать собственные псевдонимы с помощью команды `alias`:

```
$ alias li='ls -il'
$ li
total 32
75 drwxr-xr-x. 2 user user 4096 Sep 16 13:11 Desktop
79 drwxr-xr-x. 2 user user 4096 Sep 20 15:40 Documents
76 drwxr-xr-x. 2 user user 4096 Sep 15 13:30 Downloads
80 drwxr-xr-x. 2 user user 4096 Sep 15 13:30 Music
81 drwxr-xr-x. 2 user user 4096 Sep 15 13:30 Pictures
78 drwxr-xr-x. 2 user user 4096 Sep 15 13:30 Public
77 drwxr-xr-x. 2 user user 4096 Sep 15 13:30 Templates
82 drwxr-xr-x. 2 user user 4096 Sep 15 13:30 Videos
$
```

После определения значения псевдонима появляется возможность использовать его в командном интерпретаторе в любое время, в том числе в сценариях командного интерпретатора.

Псевдонимы команд действуют аналогично локальным переменным среды. Они являются допустимыми только для процесса командного интерпретатора, в котором определены:

```
$ alias li='ls -il'
$ bash
$ li
bash: li: command not found
$
```

Но читатель, разумеется, уже знает решение этой проблемы. При запуске нового экземпляра интерактивного командного интерпретатора всегда происходит считывание файла запуска `$HOME/.bashrc` командным интерпретатором `bash`. Этот файл превосходно подходит для размещения команд `alias` (как и сказано в комментариях к файлу `.bashrc`).

Резюме

В настоящей главе приведено подробное описание переменных среды Linux. Доступ к глобальным переменным среды может быть получен из любого дочернего процесса, запуск которого был осуществлен процессом, определившим эти переменные. Локальные переменные среды доступны только в том процессе, в котором они определены.

В системе Linux глобальные и локальные переменные среды используются для хранения информации о системной среде. Доступ к этой информации может быть получен из интерфейса командной строки командного интерпретатора, а также из сценариев командного интерпретатора. В командном интерпретаторе `bash` используются системные переменные среды, которые были впервые определены при создании его прототипа, командного интерпретатора Unix Bourne, кроме того, предусмотрено большое количество новых переменных среды. Переменная среды `PATH` определяет шаблон поиска, который применяется командным интерпретатором `bash` для обнаружения местонахождения команды, вызванной на выполнение. Пользователи имеют возможность модифицировать переменную среды `PATH`, добавляя обозначения собственных каталогов или символ текущего каталога, чтобы упростить вызов программ на выполнение.

Кроме того, пользователи могут создавать собственные глобальные и локальные переменные среды для личного использования. После создания переменной среды она остается доступной в течение всего времени работы сеанса командного интерпретатора.

При запуске командного интерпретатора `bash` происходит выполнение нескольких файлов запуска. Эти файлы запуска могут содержать определения переменных среды, предназначенных для задания значений стандартных переменных среды при каждом сеансе работы с программой `bash`. При входе пользователя в систему Linux командный интерпретатор `bash` обращается к файлу запуска `/etc/profile`, а также к трем локальным файлам запуска, предусмотренным для каждого пользователя, — `$HOME/.bash_profile`, `$HOME/.bash_login` и `$HOME/.profile`. Пользователи могут настраивать эти файлы и включать в них определения переменных среды и сценарии запуска для личного использования.

Командный интерпретатор `bash` обеспечивает также работу с массивами переменных среды. Эти переменные среды позволяют задавать несколько значений в одной переменной. К этим значениям можно обращаться либо по отдельности, применяя в ссылке значение индекса, либо в целом, задавая в качестве ссылки имя всего массива переменных среды.

Наконец, в этой главе приведено описание использования псевдонимов команд. Псевдонимы команд не являются переменными среды, но действуют во многом аналогично переменным среды. Они позволяют определять псевдонимы для команд, включая параметры этих команд. Поэтому, чтобы избавиться от необходимости вводить длинную строку, состоящую из команды и ее параметров, можно просто присвоить это строковое значение краткому псевдониму, а затем использовать его по мере необходимости в сеансе командного интерпретатора.

В следующей главе будет подробно описана вся тематика, касающаяся прав доступа к файлам Linux. По-видимому, это — наиболее трудная тема для начинающих пользователей Linux. Тем не менее написание качественных сценариев командного интерпретатора невозможно без понимания того, как действуют права доступа к файлам; кроме того, без этого понимания нельзя успешно работать в системе Linux.

Основные сведения о правах доступа к файлам Linux

Ни одну операционную систему нельзя считать совершенной, если в ней не предусмотрена та или иная форма обеспечения безопасности данных. В системе должен существовать механизм, позволяющий защищать файлы от несанкционированного просмотра или модификации. В системе Linux унаследован предусмотренный в Unix способ защиты, основанный на назначении прав доступа к файлам, который позволяет отдельным пользователям и группам обращаться к файлам только после проверки ряда параметров безопасности, относящихся к каждому файлу и каталогу. В настоящей главе описано, как использовать систему безопасности файлов Linux для защиты данных, когда это необходимо, и предоставления совместного доступа к данным, когда это желательно.

Безопасность Linux

Ядром системы безопасности Linux является *учетная запись пользователя*. Каждому лицу, получающему доступ к системе Linux, должна быть присвоена уникальная учетная запись пользователя. Набор разрешений на объекты в системе, предоставляемый пользователю, зависит от того, в какой учетной записи пользователя он зарегистрировался.

Пользовательские разрешения отслеживаются с применением *идентификатора пользователя* (user ID — UID), который присваивается учетной записи при ее создании.

ГЛАВА

6

В этой главе...

Безопасность Linux

Использование групп Linux

Общие сведения о правах доступа к файлам

Изменение параметров безопасности

Обеспечение совместного использования файлов

Резюме

Идентификатор пользователя — это числовое значение, уникальное для каждого пользователя. Однако пользователи регистрируются в системе Linux не с применением идентификатора пользователя. Вместо этого пользователь при входе в систему должен указать свое *регистрационное имя* — алфавитно-цифровую текстовую строку, состоящую не более чем из восьми символов, которая применяется пользователем для регистрации в системе (наряду с паролем, связанным с регистрационным именем).

В системе Linux используются специальные файлы и программы для отслеживания и управления учетными записями пользователей в системе. Прежде чем приступить к описанию прав доступа к файлам, необходимо кратко рассмотреть, как ведется обработка учетных записей пользователей в системе Linux. В настоящем разделе приведено описание файлов и программ, необходимых для работы с учетными записями пользователей. С этой проблематикой необходимо ознакомиться, чтобы знать, как использовать эти файлы и программы, устанавливая права доступа к файлам.

Файл /etc/passwd

В системе Linux для сопоставления регистрационного имени с соответствующим значением идентификатора пользователя служит специальный файл — /etc/passwd, который содержит ряд фрагментов информации о пользователе. Ниже приведен пример типичного файла /etc/passwd в системе Linux.

```
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
news:x:9:13:news:/etc/news:
uucp:x:10:14:uucp:/var/spool/uucp:/sbin/nologin
operator:x:11:0:operator:/root:/sbin/nologin
games:x:12:100:games:/usr/games:/sbin/nologin
gopher:x:13:30:gopher:/var/gopher:/sbin/nologin
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
nobody:x:99:99:Nobody:/:/sbin/nologin
rpm:x:37:37:/:/var/lib/rpm:/sbin/nologin
vcsa:x:69:69:virtual console memory owner:/dev:/sbin/nologin
mailnull:x:47:47:/:/var/spool/mqueue:/sbin/nologin
smmsp:x:51:51:/:/var/spool/mqueue:/sbin/nologin
apache:x:48:48:Apache:/var/www:/sbin/nologin
rpc:x:32:32:Rpcbind Daemon:/var/lib/rpcbind:/sbin/nologin
ntp:x:38:38:/:etc/ntp:/sbin/nologin
nscd:x:28:28:NSCD Daemon:/:/sbin/nologin
tcpdump:x:72:72:/:/sbin/nologin
dbus:x:81:81:System message bus:/:/sbin/nologin
avahi:x:70:70:Avahi daemon:/:/sbin/nologin
hsqldb:x:96:96:/:/var/lib/hsqldb:/sbin/nologin
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
```

```
rpcuser:x:29:29:RPC Service User:/var/lib/nfs:/sbin/nologin
nfsnobody:x:65534:65534:Anonymous NFS User:/var/lib/nfs:/sbin/nologin
haldaemon:x:68:68:HAL daemon:/sbin/nologin
xfs:x:43:43:X Font Server:/etc/X11/fs:/sbin/nologin
gdm:x:42:42:/:/var/gdm:/sbin/nologin
rich:x:500:500:Rich Blum:/home/rich:/bin/bash
mama:x:501:501:Mama:/home/mama:/bin/bash
katie:x:502:502:katie:/home/katie:/bin/bash
jessica:x:503:503:Jessica:/home/jessica:/bin/bash
mysql:x:27:27:MySQL Server:/var/lib/mysql:/bin/bash
$
```

Учетная запись пользователя `root` принадлежит администратору системы Linux и всегда имеет присвоенный ей идентификатор пользователя, UID 0. Вполне очевидно, что в системе Linux создается много учетных записей пользователей для выполнения различных функций, которые не принадлежат реальным пользователям. Эти учетные записи именуются *системными учетными записями*. Системная учетная запись — это специальная учетная запись, которую используют службы, эксплуатируемые в системе, для получения доступа к ресурсам системы. Все службы, работающие в фоновом режиме, должны быть зарегистрированы в системе Linux под той или иной учетной записью пользователя системы.

На начальных этапах развития Linux, когда еще не уделялось столь значительное внимание безопасности данных, эти службы часто регистрировались просто с применением учетной записи пользователя `root`. Но, к сожалению, если происходил взлом защиты одной из этих служб посторонним лицом, не имеющим полномочий для работы в системе, этот человек немедленно получал доступ ко всей системе с правами пользователя `root`. В современных версиях Linux такая возможность исключена, поскольку в них почти все службы, работающие на сервере Linux в фоновом режиме, имеют свою собственную учетную запись пользователя и регистрируются с помощью этой учетной записи. Таким образом, даже если какому-либо взломщику защиты удастся взять на себя контроль над службой, остается надежда на то, что он не сможет получить доступ ко всей системе.

В системе Linux идентификаторы пользователей с числовыми значениями меньше 500 зарезервированы для системных учетных записей. Есть и такие службы, которые не могут работать должным образом, если им не назначены какие-то конкретные идентификаторы пользователей. В большинстве систем Linux при создании учетных записей для обычных пользователей происходит назначение первого свободного идентификатора пользователя, начиная с 500 (хотя это правило не распространяется на все дистрибутивы Linux).

Рассматривая файл `/etc/passwd`, можно заметить, что в нем приведены не только регистрационное имя и идентификатор пользователя, но и большой объем дополнительных данных. Поля файла `/etc/passwd` содержат следующую информацию:

- регистрационное имя пользователя;
- пароль пользователя;
- числовой идентификатор учетной записи пользователя;
- числовой идентификатор группы (group ID — `GID`), к которой относится учетная запись пользователя;
- текстовое описание учетной записи пользователя (поле с текстовым описанием называется полем комментария);
- местоположение каталога `HOME` для пользователя;
- командный интерпретатор, применяемый по умолчанию для пользователя.

В каждом поле пароля, которое имеется в файле `/etc/passwd`, задано значение `x`. Это не означает, что все учетные записи пользователей имеют один и тот же пароль. В первых версиях Linux файл `/etc/passwd` содержал зашифрованные версии паролей пользователей. Но со временем начал расширяться круг программ, которым требовался доступ к файлу `/etc/passwd` для получения сведений о пользователях, поэтому хранение паролей в этом файле стало небезопасным. Кроме того, со временем появилось программное обеспечение, которое позволяло относительно легко расшифровывать засекреченные пароли, поэтому злоумышленники получили стимул для осуществления попыток взлома паролей пользователей, хранящихся в файле `/etc/passwd`. Разработчикам Linux пришлось пересмотреть с нуля всю политику хранения паролей.

В настоящее время в большинстве систем Linux пароли пользователей хранятся в отдельном файле (который носит имя теневого файла, или файла *shadow*, и расположен в каталоге `/etc/shadow`). Доступ к этому файлу разрешен только специальным программам (таким как программы регистрации).

Вполне очевидно, что файл `/etc/passwd` представляет собой обычный текстовый файл. Можно использовать любой текстовый редактор, чтобы выполнять ручную функции управления пользователями (например, добавлять, изменять или удалять учетные записи пользователей), корректируя непосредственно файл `/etc/passwd`. Однако это чрезвычайно опасная практика. Если структура файла `/etc/passwd` будет искажена, то система не сможет его прочитать, поэтому возможность регистрации потеряют все пользователи (даже пользователь `root`). Вместо этого следует использовать намного более безопасные стандартные программы управления пользователями Linux, позволяющие осуществлять все необходимые функции управления пользователями.

Файл `/etc/shadow`

Файл `/etc/shadow` предоставляет намного больший контроль над тем, как должно происходить управление паролями в системе Linux. К файлу `/etc/shadow` имеет доступ только пользователь `root`, поэтому данный файл защищен гораздо лучше по сравнению с файлом `/etc/passwd`.

В файле `/etc/shadow` содержится по одной записи для каждой учетной записи пользователя в системе. Записи выглядят следующим образом:

```
rich:$1$.FfcK0ns$f1UgiyHQ25wrB/hykCn020:11627:0:99999:7:::
```

В каждой записи файла `/etc/shadow` имеются девять полей с такими данными:

- регистрационное имя, соответствующее регистрационному имени в файле `/etc/passwd`;
- зашифрованный пароль;
- количество дней, отсчитываемое с 1 января 1970 года, которое позволяет определить дату последнего изменения пароля;
- минимальное количество дней, по истечении которого может быть изменен пароль;
- количество дней до того, как должен быть изменен пароль;
- количество дней до истечения срока действия пароля, после чего пользователь получит предупреждение в связи с необходимостью сменить пароль;
- количество дней после истечения срока действия пароля, по прошествии которых учетная запись будет отключена;

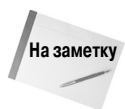
- дата (хранится как количество дней с 1 января 1970 года), после которой учетная запись пользователя была отключена;
- поле, зарезервированное для дальнейшего использования.

Ввод в действие системы теневых паролей позволил обеспечить в системе Linux гораздо более детализированный контроль над паролями пользователей. Современная система паролей позволяет управлять тем, как часто пользователь должен сменять свой пароль и когда должна быть отключена учетная запись, если смена пароля не произошла.

Добавление нового пользователя

Основным инструментом, который служит для добавления нового пользователя в систему Linux, является команда `useradd`. Она предоставляет простой способ создания новой учетной записи пользователя и развертывания одновременно с этим структуры каталогов `HOME` для пользователя. В команде `useradd` применяется сочетание системных значений по умолчанию и параметров командной строки для определения учетной записи пользователя должным образом. Для просмотра системных значений по умолчанию, предусмотренных в конкретном дистрибутиве Linux, достаточно ввести команду `useradd` с параметром `-D`:

```
# /usr/sbin/useradd -D
GROUP=100
HOME=/home
INACTIVE=-1
EXPIRE=
SHELL=/bin/bash
SKEL=/etc/skel
CREATE_MAIL_SPOOL=yes
#
```



В некоторых дистрибутивах Linux программы управления пользователями и группами Linux находятся в каталоге `/usr/sbin`, который может быть не указан в переменной среды `PATH`. Если так обстоят дела в применяемом дистрибутиве Linux, то можно либо добавить этот каталог к переменной среды `PATH`, либо всегда задавать абсолютный путь к файлу программы для ее выполнения.

Параметр `-D` показывает, какие значения по умолчанию используются в команде `useradd`, если соответствующие параметры не заданы в командной строке при создании новой учетной записи пользователя. В рассматриваемом примере показаны следующие значения по умолчанию:

- новый пользователь добавляется к общей группе с идентификатором группы 100;
- для нового пользователя создается учетная запись `HOME` в каталоге `/home/loginname`;
- учетная запись не отключается по истечении срока действия пароля;
- новая учетная запись не настраивается на истечение срока действия в заданную дату;
- для новой учетной записи в качестве командного интерпретатора, заданного по умолчанию, используется командный интерпретатор `bash`;
- при создании учетной записи пользователя система копирует содержимое каталога `/etc/skel` в каталог `HOME` пользователя;
- система создает файл в каталоге `mail`, чтобы можно было получать почту с помощью учетной записи пользователя.

В этом списке выполняемых функций особый интерес представляет предпоследняя операция. Она означает, что команда `useradd` позволяет администратору создавать заданную по умолчанию конфигурацию каталога `HOME`, а затем применять ее в качестве шаблона для создания каталога `HOME` нового пользователя. Таким образом, обеспечивается возможность автоматически помещать в каталог `HOME` каждого нового пользователя предусмотренные по умолчанию файлы для работы в системе. В системе Ubuntu Linux в каталоге `/etc/skel` находятся следующие файлы:

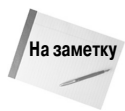
```
$ ls -al /etc/skel
total 32
drwxr-xr-x  2 root root  4096 2010-04-29 08:26 .
drwxr-xr-x 135 root root 12288 2010-09-23 18:49 ..
-rw-r--r--  1 root root   220 2010-04-18 21:51 .bash_logout
-rw-r--r--  1 root root  3103 2010-04-18 21:51 .bashrc
-rw-r--r--  1 root root   179 2010-03-26 08:31 examples.desktop
-rw-r--r--  1 root root   675 2010-04-18 21:51 .profile
$
```

Читатель должен был запомнить эти файлы, которые описаны в главе 5. Это — стандартные файлы запуска для среды командного интерпретатора `bash`. Система автоматически копирует эти предусмотренные по умолчанию файлы в каталог `HOME` каждого нового создаваемого пользователя.

Правильность выполнения данной операции можно проверить, создав новую учетную запись пользователя с применением заданных по умолчанию параметров системы, а затем рассмотрев содержимое каталога `HOME` нового пользователя:

```
# useradd -m test
# ls -al /home/test
total 24
drwxr-xr-x  2 test test  4096 2010-09-23 19:01 .
drwxr-xr-x  4 root root  4096 2010-09-23 19:01 ..
-rw-r--r--  1 test test   220 2010-04-18 21:51 .bash_logout
-rw-r--r--  1 test test  3103 2010-04-18 21:51 .bashrc
-rw-r--r--  1 test test   179 2010-03-26 08:31 examples.desktop
-rw-r--r--  1 test test   675 2010-04-18 21:51 .profile
#
```

По умолчанию команда `useradd` не создает каталог `HOME`, но указанием на необходимость создания каталога `HOME` для этой команды служит параметр командной строки `-m`. Как показывает этот пример, командой `useradd` действительно был создан новый каталог `HOME` с использованием файлов, содержащихся в каталоге `/etc/skel`.



Для выполнения команд администрирования учетных записей пользователей, приведенных в данной главе, необходимо либо зарегистрироваться с помощью специальной учетной записи пользователя `root`, либо воспользоваться командой `sudo` для выполнения этих команд в качестве владельца учетной записи пользователя `root`.

Если возникает необходимость переопределить применяемые по умолчанию значения или операции при создании нового пользователя, это можно сделать с помощью параметров командной строки (табл. 6.1).

Вполне очевидно, что при создании новой учетной записи пользователя есть возможность переопределить все системные значения по умолчанию исключительно с применением пара-

метров командной строки. Однако, если будет обнаружено, что постоянно возникает необходимость переопределять какое-то значение, то легче просто поменять системное значение по умолчанию.

Таблица 6.1. Параметры командной строки useradd

Параметр	Описание
-c <i>comment</i>	Добавить текст к полю комментария нового пользователя
-d <i>home_dir</i>	Задать для исходного каталога, HOME, другое имя, отличное от регистрационного имени
-e <i>expire_date</i>	Задать дату в формате YYYY-MM-DD (ГГГГ-ММ-ДД), в которую истечет срок действия учетной записи
-f <i>inactive_days</i>	Задать количество дней после истечения срока действия пароля, когда учетная запись будет отключена. Значение 0 указывает, что учетная запись должна быть отключена сразу после истечения срока действия пароля; применение значения -1 приводит к отмене отключения
-g <i>initial_group</i>	Задать имя группы или идентификатор группы регистрации пользователя
-G <i>group . . .</i>	Задать одну или несколько дополнительных групп, к которым принадлежит пользователь
-k	Скопировать содержимое каталога /etc/skel в каталог HOME пользователя (при этом должен быть также задан параметр -m)
-m	Создать каталог HOME пользователя
-M	Не создавать каталог HOME пользователя (этот параметр используется, если настройка по умолчанию предусматривает создание такового)
-n	Создать новую группу с использованием имени, совпадающего с регистрационным именем пользователя
-r	Создать системную учетную запись
-p <i>passwd</i>	Задать применяемый по умолчанию пароль для учетной записи пользователя
-s <i>shell</i>	Задать применяемый по умолчанию командный интерпретатор входа в систему
-u <i>uid</i>	Задать уникальный идентификатор пользователя для учетной записи

Предусмотрена возможность изменить применяемое по умолчанию системное значение для нового пользователя путем задания параметра -D, наряду с параметром, представляющим значение, которое подлежит изменению. Эти параметры приведены в табл. 6.2.

Таблица 6.2. Значения параметров по умолчанию, которые могут быть изменены в команде useradd

Параметр	Описание
-b <i>default_home</i>	Изменить местоположение, в котором создаются каталоги HOME пользователей
-e <i>expiration_date</i>	Изменить дату истечения срока хранения новых учетных записей
-f <i>inactive</i>	Изменить количество дней после истечения срока действия пароля до отключения учетной записи
-g <i>group</i>	Изменить применяемое по умолчанию имя группы (или идентификатор группы)
-s <i>shell</i>	Изменить применяемый по умолчанию командный интерпретатор входа в систему

Таким образом, задача изменения указанных значений по умолчанию является совсем не сложной:

```
# useradd -D -s /bin/tSCH
# useradd -D
GROUP=100
HOME=/home
INACTIVE=-1
EXPIRE=
SHELL=/bin/tSCH
SKEL=/etc/skel
CREATE_MAIL_SPOOL=yes
#
```

После этого в команде `useradd` для всех новых учетных записей пользователей, создаваемых в системе, будет использоваться командный интерпретатор `tSCH` в качестве заданного по умолчанию командного интерпретатора входа в систему.

Удаление пользователя

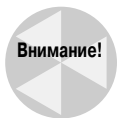
Если когда-либо потребуется удалить пользователя из системы, то достаточно применить команду `userdel`. По умолчанию команда `userdel` удаляет только сведения о пользователе из файла `/etc/passwd` и не удаляет какие-либо файлы, которыми владеет учетная запись в системе.

Если же используется параметр `-r`, то команда `userdel` удаляет каталог `HOME` пользователя наряду с каталогом `mail` этого пользователя. Тем не менее в системе могут все еще оставаться другие файлы, принадлежащие удаленной учетной записи пользователя. В некоторых вариантах среды это может приводить к возникновению проблем.

Ниже приведен пример использования команды `userdel` для удаления существующей учетной записи пользователя.

```
# /usr/sbin/userdel -r test
# ls -al /home/test
ls: cannot access /home/test: No such file or directory
#
```

После выполнения этой команды с параметром `-r` старый каталог пользователя `/home/test` больше не существует.



В среде с большим количеством пользователей необходимо соблюдать осторожность, когда приходится применять параметр `-r`. Посторонний человек не может знать, хранятся ли в каталоге `HOME` пользователя важные файлы, которые применяются какими-то другими пользователями или программами. Всегда проводите проверку перед удалением каталога `HOME` пользователя!

Изменение пользователя

В системе Linux предусмотрено несколько различных программ, предназначенных для корректировки данных, относящихся к существующим учетным записям пользователей (табл. 6.3).

Таблица 6.3. Программы модификации учетных записей пользователей

Программа	Описание
<code>usermod</code>	Программа редактирования полей учетной записи пользователя, которая позволяет также определять принадлежность к первичной и вторичной группам
<code>passwd</code>	Программа изменения пароля для существующего пользователя

Программа	Описание
chpasswd	Программа чтения файла, в котором в виде отдельных пар заданы регистрационные имена и пароли и обновления паролей
chage	Программа изменения даты истечения срока действия пароля
chfn	Программа изменения содержимого комментария к учетной записи пользователя
chsh	Программа изменения заданного по умолчанию командного интерпретатора для учетной записи пользователя

Каждая из этих программ (подробно они рассматриваются в следующих разделах) предоставляет возможность воспользоваться конкретной функцией для модификации сведений об учетных записях пользователей.

Программа usermod

Программа `usermod` имеет наиболее широкую область действия среди всех программ модификации учетных записей пользователей. Она предоставляет возможность вносить изменения в большинство полей файла `/etc/passwd`. Для этого достаточно воспользоваться лишь теми параметрами командной строки, которые соответствуют значению, подлежащему изменению. Эти параметры в основном совпадают с параметрами команды `useradd` (например, параметр `-s` вносит изменения в поле комментария, `-e` служит для изменения продолжительности срока действия, а `-g` позволяет изменить заданную по умолчанию группу регистрации). Однако эта команда предоставляет также ряд дополнительных параметров, которые могут оказаться необходимыми, включая следующие:

- параметр `-l` позволяет изменить регистрационное имя учетной записи пользователя;
- параметр `-L` дает возможность заблокировать учетную запись, чтобы пользователь не мог зарегистрироваться;
- параметр `-p` служит для изменения пароля учетной записи;
- параметр `-U` обеспечивает разблокирование учетной записи, чтобы пользователь мог входить в систему.

Особенно удобным является параметр `-L`. Он позволяет блокировать учетную запись, чтобы пользователь не мог зарегистрироваться, без удаления учетной записи и данных пользователя. Чтобы в дальнейшем вернуть эту учетную запись в нормальное состояние, достаточно воспользоваться параметром `-U`.

Команды `passwd` и `chpasswd`

Один из удобных способов смены только пароля для пользователя состоит в применении команды `passwd`:

```
# passwd test
Changing password for user test.
New UNIX password:
Retype new UNIX password:
passwd: all authentication tokens updated successfully.
#
```

Команда `passwd`, вызванная без параметров, позволяет сменить свой собственный пароль. Любой пользователь в системе имеет право сменить свой собственный пароль, но только пользователь `root` может сменить пароль любого другого пользователя.

Параметр `-e` предоставляет удобный способ вынудить пользователя произвести смену пароля при очередном входе в систему. Это позволяет администратору задать легко запоминающийся пароль пользователя, а затем вынудить пользователя вместо этого пароля задать для себя более сложный, хотя и доступный для запоминания.

Иногда возникает необходимость осуществить массовую смену паролей для большого количества пользователей в системе. В этом случае значительную помощь может оказать команда `chpasswd`, которая считывает список пар, состоящих из регистрационного имени и пароля (разделенных двоеточием), со стандартного ввода, автоматически шифрует пароли и задает их для учетных записей пользователей. Можно также воспользоваться командой перенаправления для передачи файла с парами `userid:password` в эту команду:

```
# chpasswd < users.txt
#
```

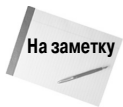
Программы `chsh`, `chfn` и `chage`

Программы `chsh`, `chfn` и `chage` специально предназначены для выполнения конкретных функций модификации учетных записей. Команда `chsh` позволяет быстро сменить заданный по умолчанию командный интерпретатор входа в систему для пользователя. При этом необходимо задавать полное имя пути для вызова командного интерпретатора, а не просто имя программы командного интерпретатора:

```
# chsh -s /bin/csh test
Changing shell for test.
Shell changed.
#
```

Команда `chfn` предоставляет стандартный способ сохранения информации в поле комментариев файла `/etc/passwd`. Команда `chfn` не вставляет в поле комментария произвольно сформированный текст, в котором, допустим, содержатся имена или псевдонимы, и не оставляет его пустым, а использует конкретную информацию, предоставляемую командой `finger` Unix, для сохранения информации в поле комментария. Команда `finger` позволяет легко находить сведения о пользователях в системе Linux:

```
# finger rich
Login: rich                               Name: Rich Blum
Directory: /home/rich                     Shell: /bin/bash
On since Thu Sep 20 18:03 (EDT) on pts/0 from 192.168.1.2
No mail.
No Plan.
#
```



Тем не менее многие системные администраторы Linux отключают команду `finger` в своих системах по соображениям безопасности.

Если команда `chfn` вызывается без параметров, то выводит запросы для пользователя на получение соответствующих значений, которые должны быть введены в поле комментария:

```
# chfn test
Changing finger information for test.
```

```

Name []: Ima Test
Office []: Director of Technology
Office Phone []: (123)555-1234
Home Phone []: (123)555-9876

Finger information changed.
# finger test
Login: test                               Name: Ima Test
Directory: /home/test                     Shell: /bin/csh
Office: Director of Technology            Office Phone: (123)555-1234
Home Phone: (123)555-9876
Never logged in.
No mail.
No Plan.
#

```

После предоставления этих данных рассматриваемая запись в файле `/etc/passwd` будет выглядеть следующим образом:

```

# grep test /etc/passwd
test:x:504:504:Ima Test,Director of Technology,(123)555-
1234,(123)555-9876:/home/test:/bin/csh
#

```

Вся информация команды `finger` сохраняется в удобном виде в записи файла `/etc/passwd`.

Наконец, команда `chage` позволяет управлять процессом смены паролей для учетных записей пользователей в связи с устареванием. В этой команде применяется несколько параметров, позволяющих отдельно задавать необходимые значения (табл. 6.4).

Таблица 6.4. Параметры команды `chage`

Параметр	Описание
<code>-d</code>	Задать количество дней после последнего изменения пароля
<code>-E</code>	Задать дату истечения срока годности пароля
<code>-I</code>	Задать количество дней неприятия мер после истечения пароля, по прошествии которых учетная запись блокируется
<code>-m</code>	Задать минимальное количество дней между сменами пароля
<code>-W</code>	Задать количество дней до истечения пароля, когда появляется предупреждающее сообщение

Значения даты в команде `chage` могут быть выражены с использованием одного из двух методов:

- дата в формате `YYYY-MM-DD`;
- числовое значение, представляющее количество дней с 1 января 1970 года.

Одной из привлекательных особенностей команды `chage` является то, что она позволяет задавать дату истечения срока действия для учетной записи. С применением этого средства можно создавать временные учетные записи пользователей, срок действия которых автоматически истекает в заданную дату, что позволяет избежать необходимости следить за их своевременным удалением! Учетные записи с истекшим сроком действия аналогичны заблокированным учетным записям. Учетная запись все еще существует, но пользователь не может войти в систему с ее помощью.

Использование групп Linux

Учетные записи пользователей превосходно подходят для управления безопасностью применительно к отдельным пользователям, но они не столь удобны, когда требуется обеспечить совместную работу с ресурсами для групп пользователей. Для достижения указанной цели в системе Linux реализована еще одна концепция безопасности, известная как *группы*.

Для групп назначаются разрешения, которые позволяют нескольким пользователям совместно работать в рамках общих разрешений на такие объекты в системе, как файлы, каталоги или устройства (дополнительные сведения по этой теме приведены ниже, в разделе “Общие сведения о правах доступа к файлам”).

Между разными дистрибутивами Linux имеются определенные различия в отношении того, какие правила определения принадлежности к группам по умолчанию в них используются. В некоторых дистрибутивах Linux создается только одна группа, в которой в качестве членов содержатся все учетные записи пользователей. Работая в таком дистрибутиве Linux, необходимо соблюдать осторожность, поскольку может оказаться, что файлы одного пользователя доступны для чтения всем другим пользователям в системе. В других дистрибутивах создается отдельная учетная запись для каждого пользователя, что обеспечивает намного более высокий уровень безопасности.

Каждая группа имеет уникальный идентификатор группы, который, как и идентификатор пользователя, представляет собой уникальное числовое значение в системе. Наряду с идентификатором группы каждая группа имеет уникальное имя группы. Для работы с группами предусмотрено несколько программ, с помощью которых можно создавать собственные группы в системе Linux и управлять ими. В настоящем разделе описано, как сохраняется информация групп и как использовать программы работы с группами для создания новых и модификации существующих групп.

Файл `/etc/group`

Полностью аналогично учетным записям пользователей, сведения о группах хранятся в системе в отдельном файле. Информация о каждой группе, применяемой в системе, содержится в файле `/etc/group`. Ниже приведено несколько примеров из типичного файла `/etc/group` в системе Linux.

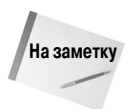
```
root:x:0:root
bin:x:1:root,bin,daemon
daemon:x:2:root,bin,daemon
sys:x:3:root,bin,adm
adm:x:4:root,adm,daemon
rich:x:500:
mama:x:501:
katie:x:502:
jessica:x:503:
mysql:x:27:
test:x:504:
```

Как и идентификаторы пользователей, идентификаторы групп присваиваются с помощью специального формата. Группам, которые предназначены для системных учетных записей, присваиваются идентификаторы групп с числовым значением меньше 500, а группам для пользователей — идентификаторы групп, начинающиеся с 500. В файле `/etc/group` используются следующие четыре поля:

- имя группы;
- пароль группы;
- идентификатор группы;
- список учетных записей пользователя, которые принадлежат к группе.

Пароль группы позволяет не члену группы стать на время членом группы, используя пароль. Такая возможность применяется не слишком часто, но она существует.

Ни в коем случае не следует добавлять пользователей к группам, редактируя файл `/etc/group`. Вместо этого необходимо использовать команду `usermod` (см. выше раздел “Безопасность Linux”) для добавления учетных записей пользователей к группе. Для того чтобы добавить пользователей к различным группам, требуется вначале создать эти группы.



При этом может стать источником заблуждений наличие списка учетных записей пользователей. В частности, можно обнаружить, что в перечне групп встречаются такие группы, которые не имеют в своем списке пользователей ни одного пользователя. Но это не означает, что данные группы не имеют членов. Если в учетной записи пользователя в файле `/etc/passwd` некоторая группа указана как заданная по умолчанию, то эта учетная запись пользователя не указана в файле `/etc/group` в качестве члена. Это правило многие годы остается источником путаницы для несметного количества системных администраторов!

Создание новых групп

Для создания новых групп в системе применяется команда `groupadd`:

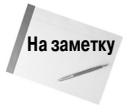
```
# /usr/sbin/groupadd shared
# tail /etc/group
haldaemon:x:68:
xfs:x:43:
gdm:x:42:
rich:x:500:
mama:x:501:
katie:x:502:
jessica:x:503:
mysql:x:27:
test:x:504:
shared:x:505:
#
```

По умолчанию после создания новой группы в нее еще не назначен ни один пользователь. Команда `groupadd` не предоставляет возможности добавлять учетные записи пользователей к группам. Вместо этого для добавления новых пользователей предназначена команда `usermod`:

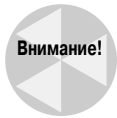
```
# /usr/sbin/usermod -G shared rich
# /usr/sbin/usermod -G shared test
# tail /etc/group
haldaemon:x:68:
xfs:x:43:
gdm:x:42:
rich:x:500:
mama:x:501:
katie:x:502:
```

```
jessica:x:503:
mysql:x:27:
test:x:504:
shared:x:505:rich, test
#
```

Теперь в группу shared входят два члена, test и rich. Параметр -G команды usermod присоединяет новую группу к списку групп для учетной записи пользователя.



Если изменение состава групп пользователей осуществляется по отношению к учетной записи, пользователь которой в настоящее время зарегистрирован в системе, то данный пользователь должен выйти из системы, а затем снова войти, чтобы это изменение вступило в силу.



Назначая группы для учетных записей пользователей, необходимо соблюдать осторожность. Например, при использовании параметра -g группа с указанным именем заменяет заданную по умолчанию группу для учетной записи пользователя. Параметр -G позволяет добавить группу к списку групп, к которым принадлежит пользователь, оставляя неизменной группу, заданную по умолчанию.

Внесение изменений в группы

Как показывает файл /etc/group, объем информации о группах, подлежащей изменению, не слишком велик. Команда groupmod позволяет изменить идентификатор группы (с использованием параметра -g) или имя группы (с использованием параметра -n) применительно к существующей группе:

```
# /usr/sbin/groupmod -n sharing shared
# tail /etc/group
haldaemon:x:68:
xfs:x:43:
gdm:x:42:
rich:x:500:
mama:x:501:
katie:x:502:
jessica:x:503:
mysql:x:27:
test:x:504:
sharing:x:505:test,rich
#
```

Если изменяется имя группы, идентификатор группы и список членов группы остаются неизменными, а другим становится только имя группы. В связи с тем, что все определения прав доступа основаны на идентификаторе группы, допускается изменять имя группы настолько часто, насколько это потребуется, не оказывая отрицательного влияния на безопасность файла.

Общие сведения о правах доступа к файлам

Усвоение изложенных выше сведений о пользователях и группах позволяет приступить к подробному изучению сложной темы, касающейся прав доступа к файлам, которая впервые

была затронута при описании команды `ls`. В настоящем разделе описано, как расшифровать сведения об этих разрешениях и по какому принципу формируются разрешения.

Использование символов для определения прав доступа к файлам

Как было описано в главе 3, команда `ls` позволяет получать сведения о правах доступа к файлам, относящихся к файлам, каталогам и устройствам в системе Linux:

```
$ ls -l
total 68
-rw-rw-r-- 1 rich rich  50 2010-09-13 07:49 file1.gz
-rw-rw-r-- 1 rich rich  23 2010-09-13 07:50 file2
-rw-rw-r-- 1 rich rich  48 2010-09-13 07:56 file3
-rw-rw-r-- 1 rich rich  34 2010-09-13 08:59 file4
-rwxrwxr-x 1 rich rich 4882 2010-09-18 13:58 myprog
-rw-rw-r-- 1 rich rich  237 2010-09-18 13:58 myprog.c
drwxrwxr-x 2 rich rich 4096 2010-09-03 15:12 test1
drwxrwxr-x 2 rich rich 4096 2010-09-03 15:12 test2
$
```

Первое поле в выводе этого листинга представляет собой код, который описывает разрешения для файлов и каталогов. Первый символ в поле определяет тип объекта:

- `-` — обозначает файлы;
- `d` — обозначает каталоги;
- `l` — обозначает ссылки;
- `c` — обозначает символьные устройства;
- `b` — обозначает блочные устройства;
- `n` — обозначает сетевые устройства.

За этим следуют три набора по три символа. Каждый набор из трех символов задает три разрешения на доступ:

- `r` — определяет разрешение на чтение для объекта;
- `w` — определяет разрешение на запись для объекта;
- `x` — определяет разрешение на выполнение для объекта.

Если разрешение отсутствует, то в соответствующей позиции появляется тире. Три набора символов связывают следующие три уровня безопасности для объекта:

- владелец объекта;
- группа, которая владеет объектом;
- все прочие пользователи в системе.

Расшифровка этих сведений приведена на рис. 6.1.

Легче всего можно раскрыть эту тему, взяв конкретный пример, и декодировать права доступа к файлу одно за другим:

```
-rwxrwxr-x 1 rich rich 4882 2010-09-18 13:58 myprog
```

Права доступа к файлам в Linux

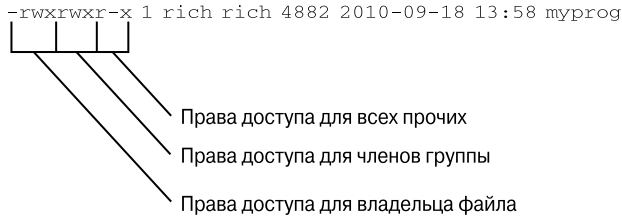


Рис. 6.1. Права доступа к файлам Linux

Файл `myprog` имеет следующие наборы разрешений:

- набор разрешений `rwx` задан для владельца файла (в качестве него определено регистрационное имя `rich`);
- набор разрешений `rwx` задан для владельца группы файлов (в качестве него определена группа с именем `rich`);
- набор разрешений `r-x` задан для всех остальных пользователей в системе.

Эти разрешения указывают, что пользователь с регистрационным именем `rich` может читать, писать и выполнять файл (учитывая то, что он имеет полные права на файл). Аналогичным образом читать, писать и выполнять этот файл могут также члены группы `rich`. Но все прочие пользователи, не принадлежащие к группе `rich`, могут только читать и выполнять файл; вместо символа `w` стоит тире, которое указывает, что на этом уровне безопасности разрешения на запись не назначены.

Заданные по умолчанию права доступа к файлу

Представляет интерес вопрос о том, как формируются эти права доступа к файлу. Ответ на данный вопрос состоит в том, что для этого используется команда `umask`, которая определяет пользовательскую маску. Команда `umask` позволяет задать применяемые по умолчанию разрешения для любого создаваемого файла или каталога:

```
$ touch newfile
$ ls -al newfile
-rw-r--r-- 1 rich rich 0 Sep 20 19:16 newfile
$
```

С помощью выполненной в этом примере команды `touch` был создан файл с использованием заданных по умолчанию разрешений, которые присвоены учетной записи пользователя, применяющиеся автором. Команда `umask` показывает и задает разрешения, применяемые по умолчанию:

```
$ umask
0022
$
```

К сожалению, предусмотренные в команде `umask` настройки не совсем очевидны, а при попытке точно установить, как она работает, ситуация становится еще более запутанной. Первая цифра представляет специальное средство безопасности, называемое *битом фиксации*. До-

полнительные сведения об этом будут приведены ниже, в разделе “Обеспечение совместного доступа к файлам”.

Следующие три цифры представляют восьмеричные значения `umask` для файла или каталога. Чтобы понять, как работает команда `umask`, необходимо вначале ознакомиться с восьмеричными обозначениями параметров безопасности.

Формирование *восьмеричных обозначений* параметров безопасности осуществляется так: берутся три значения разрешений `rwX` и преобразуются в трехбитовое двоичное значение, представленное отдельным восьмеричным значением. В двоичном представлении каждая позиция представляет собой двоичную цифру (бит). Таким образом, если единственным заданным разрешением является разрешение на чтение, то значение прав доступа приобретает вид `r--`, соответствующее двоичному значению 100, а это значение преобразуется в восьмеричное значение 4. В табл. 6.5 приведены все возможные комбинации, с которыми приходится сталкиваться при использовании разрешений.

Таблица 6.5. Коды прав доступа к файлам Linux

Разрешения	Двоичное значение	Восьмеричное значение	Описание
---	000	0	Разрешения отсутствуют
--x	001	1	Разрешение только на выполнение
-w-	010	2	Разрешение только на запись
-wx	011	3	Разрешения на запись и выполнение
r--	100	4	Разрешение только на чтение
r-x	101	5	Разрешения на чтение и выполнение
rw-	110	6	Разрешения на чтение и запись
rwX	111	7	Разрешения на чтение, запись и выполнение

Восьмеричная система обозначения предусматривает получение восьмеричных кодов разрешений и перечисление их в последовательности, соответствующей трем уровням безопасности (пользователь, группа и все прочие). Таким образом, восьмеричное обозначение прав доступа 664 показывает наличие разрешений на чтение и запись для пользователя и группы, а для всех прочих — разрешения только на чтение.

Однако даже после изучения восьмеричных обозначений, когда становится известен способ определения разрешений, знакомство с конкретными значениями `umask` вызывает еще большую путаницу. Ознакомление с заданной по умолчанию пользовательской маской, `umask`, в системе Linux автора показывает, что она имеет восьмеричное обозначение 0022, но созданный автором файл имеет восьмеричное обозначение разрешений 644. Как возникла такая ситуация?

Дело в том, что значение пользовательской маски является именно таковым — маской. Пользовательская маска маскирует разрешения, которые не должны быть назначены на том или ином уровне безопасности. Теперь необходимо заняться восьмеричными вычислениями, чтобы до конца понять, что происходит.

Значение пользовательской маски вычитается из полного набора разрешений для объекта. Полным набором разрешений для файла является режим 666 (который соответствует разрешениям на чтение и запись для всех), а для каталога это значение равно 777 (разрешения на чтение, запись и выполнение для всех).

Таким образом, в данном примере создание файла начинается с задания разрешений 666, после чего применяется пользовательская маска 022 и в результате остаются только права доступа к файлу 644.

Значение пользовательской маски обычно устанавливается в файле запуска `/etc/profile` (см. главу 5). С помощью команды `umask` можно задать другое значение пользовательской маски, применяемой по умолчанию:

```
$ umask 026
$ touch newfile2
$ ls -l newfile2
-rw-r-----  1 rich      rich          0 Sep 20 19:46 newfile2
$
```

Если будет задано значение пользовательской маски 026, то по умолчанию права доступа к файлу будут приобретать значение 640, поэтому для каждого нового файла права доступа сведутся к разрешению только на чтение для членов группы, а все остальные пользователи в системе не получают никаких разрешений на работу с файлом.

Значение пользовательской маски применяется также при создании новых каталогов:

```
$ mkdir newdir
$ ls -l
drwxr-x--x  2 rich      rich          4096 Sep 20 20:11 newdir/
$
```

Поскольку заданные по умолчанию разрешения для каталога определены как 777, после создания каталога результирующие разрешения, которые обусловлены применением пользовательской маски, являются иными по сравнению с вновь созданным файлом. После вычитания значения пользовательской маски 026 из значения 777 остается тройка восьмеричных цифр 751, которые и определяют права доступа к каталогу.

Изменение параметров безопасности

Иногда возникает необходимость изменить параметры безопасности существующего файла или каталога. Для этого в Linux предусмотрено несколько различных программ. В настоящем разделе показано, как изменить существующие разрешения заданного по умолчанию владельца и заданные по умолчанию настройки группы для файла или каталога.

Изменение разрешений

Изменять параметры безопасности для файлов и каталогов позволяет команда `chmod`, которая имеет следующий формат:

```
chmod options mode file
```

Параметр `mode` дает возможность задавать параметры безопасности с применением восьмеричных или символических обозначений. Задача определения параметров с помощью восьмеричных обозначений является весьма несложной; достаточно лишь задать стандартный восьмеричный код, состоящий из трех цифр, который устанавливает для файла необходимые параметры:

```
$ chmod 760 newfile
$ ls -l newfile
-rwxr-----  1 rich      rich          0 Sep 20 19:16 newfile
$
```

Эти восьмеричные обозначения прав доступа автоматически применяются к указанному файлу. С другой стороны, применение символических обозначений для разрешений является более сложным.

В команде `chmod` вместо задания обычной строки, представляющей собой три набора по три символа, применяется другой подход. Ниже показан формат, применяемый для задания разрешений с помощью символических обозначений.

```
[ugoa...][[+=)] [rwxXstugo...]
```

Вполне очевидно, что этот формат также является достаточно удобным. Первая группа символов определяет, к какому объекту относятся новые разрешения:

- символ `u` (сокращение от `user`) применяется для обозначения пользователя;
- символ `g` (сокращение от `group`) применяется для обозначения группы;
- символ `o` (сокращение от `others`) применяется для обозначения других (всех остальных объектов);
- символ `a` (сокращение от `all`) применяется для обозначения всех вышеупомянутых объектов.

После этого вводится символ, который указывает, следует ли сложить данное разрешение с существующими разрешениями (+), вычесть разрешение из существующих разрешений (-) или задать разрешение равным указанному значению (=).

Наконец, третий символ представляет собой разрешение, используемое для корректировки прав доступа. Необходимо отметить, что при этом используются не только обычные значения `rwx`, но и дополнительные значения. Эти дополнительные значения таковы:

- символ `X` применяется для присваивания прав на выполнение, только если объект представляет собой каталог или если для него уже назначено право на выполнение;
- символ `s` предназначен для задания идентификатора пользователя или идентификатора группы при выполнении;
- символ `t` служит для сохранения текста программы;
- символ `u` позволяет задать в качестве разрешений разрешения владельца;
- символ `g` позволяет задать в качестве разрешений разрешения группы;
- символ `o` позволяет задать в качестве разрешений разрешения других объектов.

Пример применения описанного способа корректировки разрешений приведен ниже.

```
$ chmod o+r newfile
$ ls -l newfile
-rwxrw-r--  1 rich      rich          0 Sep 20 19:16 newfile
$
```

Запись `o+r` служит для добавления разрешения на чтение ко всем разрешениям всех уровней безопасности, которые уже заданы.

```
$ chmod u-x newfile
$ ls -l newfile
-rw-rw-r--  1 rich      rich          0 Sep 20 19:16 newfile
$
```

Запись `u-x` удаляет право на выполнение, которое перед этим имел пользователь. Еще раз отметим, что для команды `ls` заданы такие параметры, чтобы в ее выводе было показано, имеет ли файл разрешение на выполнение, путем добавления звездочки к имени файла.

Предусмотрены также *дополнительные* параметры, которые позволяют вносить дополнительные уточнения в операции, выполняемые командой `chmod`. Параметр `-R` указывает, что внесение изменений в разрешения для файлов и каталогов должно проводиться рекурсивно. В задаваемых именах файлов можно использовать символы-шаблоны, что позволяет изменять разрешения сразу для нескольких файлов с помощью одной команды.

Изменение прав владения

Иногда возникает необходимость сменить владельца файла, например, в связи с тем, что какой-то сотрудник увольняется из организации или разработчик создает приложение, для которого перед передачей в эксплуатацию необходимо задать в качестве владельца системную учетную запись. В системе Linux для решения этой задачи предусмотрены две команды. Команда `chown` позволяет без особых затруднений сменить владельца файла, а команда `chgrp` дает возможность определить для файла другую заданную по умолчанию группу.

Команда `chown` имеет следующий формат:

```
chown options owner[.group] file
```

Для обозначения нового владельца файла можно указать регистрационное имя или числовой идентификатор пользователя:

```
# chown dan newfile
# ls -l newfile
-rw-rw-r-- 1 dan      rich          0 Sep 20 19:16 newfile
#
```

Очевидно, что это несложно. Команда `chown` позволяет также одновременно сменить и пользователя *owner*, и группу *group* файла *file*:

```
# chown dan.shared newfile
# ls -l newfile
-rw-rw-r-- 1 dan      shared        0 Sep 20 19:16 newfile
#
```

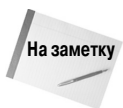
А самый сложный вариант применения этой команды состоит в том, чтобы сменить только заданную по умолчанию группу для файла:

```
# chown .rich newfile
# ls -l newfile
-rw-rw-r-- 1 dan      rich          0 Sep 20 19:16 newfile
#
```

Наконец, если в рассматриваемой системе Linux используются отдельные имена групп, которые совпадают с регистрационными именами пользователей, то можно сменить и то и другое с помощью одной команды:

```
# chown test. newfile
# ls -l newfile
-rw-rw-r-- 1 test     test          0 Sep 20 19:16 newfile
#
```

В команде `chown` применяются немного другие дополнительные параметры. Параметр `-R` позволяет вносить изменения рекурсивно, проходя по подкаталогам и файлам, для чего используются символы-шаблоны. Кроме того, параметр `-h` дополнительно позволяет изменить право владения для любых файлов, которые символически связаны с данным файлом.



Сменить владельца файла может только пользователь root. Задать другую группу по умолчанию для файла может любой пользователь, но при условии, что он является членом обеих групп — и текущей, и будущей группы для файла.

Если требуется сменить только заданную по умолчанию группу для файла или каталога, то можно воспользоваться удобной командой `chgrp`:

```
$ chgrp shared newfile
$ ls -l newfile
-rw-rw-r-- 1 rich shared 0 Sep 20 19:16 newfile
$
```

После этого любой член группы `shared` получает возможность выполнять операции записи в файл. В этом состоит один из способов обеспечения совместного использования файлов в системе Linux. Однако организация совместного доступа к файлам для группы пользователей в системе может оказаться более сложной. В следующем разделе описано, как решить эту задачу.

Обеспечение совместного использования файлов

Читатель мог уже обнаружить, что одним из способов организации совместного доступа к файлам в системе Linux является создание групп. Но если для работы пользователей должна быть предоставлена такая среда, в которой полностью обеспечен совместный доступ к файлам, задача становится более сложной.

Как уже было показано в разделе “Общие сведения о правах доступа к файлам”, при создании нового файла система Linux присваивает новому файлу такие разрешения, в которых учитываются заданные по умолчанию идентификатор пользователя и идентификатор группы. Чтобы предоставить доступ к вновь созданному файлу другим пользователям, необходимо либо сменить права доступа для группы безопасности, относящейся ко всем пользователям, либо назначить файл в другую группу по умолчанию, которая включает других пользователей.

И тот и другой способ может оказаться сложным в крупномасштабной среде, если в ней необходимо создавать и предоставлять совместный доступ к документам большому количеству пользователей. К счастью, предусмотрен простой способ решения этой проблемы.

В системе Linux для каждого файла и каталога хранятся три дополнительных информационных бита, которые указаны ниже.

- Установленный идентификатор пользователя (Set User ID — SUID). При исполнении файла пользователем прогон этой программы осуществляется на основе разрешений владельца файла.
- Установленный идентификатор группы (Set Group ID — SGID). Применительно к файлу выполнение программы осуществляется на основе разрешений для группы файла. Что касается каталога, то для новых файлов, создаваемых в каталоге, в качестве заданной по умолчанию группы используется группа каталога.
- Бит фиксации. Файл, для которого установлен бит фиксации, остается (фиксируется) в памяти после завершения процесса.

Бит SGID является важным средством обеспечения совместного доступа к файлам. Путем включения бита SGID можно принудительно задать, чтобы все файлы, вновь создаваемые в общем каталоге, принадлежали группе этого каталога, а также группе отдельного пользователя.

Для задания бита SGID применяется команда `chmod`. Этот бит добавляется перед началом стандартного трехзначного восьмеричного значения (что приводит к преобразованию его в четырехзначное восьмеричное значение). Если же используются символические обозначения, то можно дополнительно задать символ `s`.

При работе с восьмеричными обозначениями необходимо учитывать правильное расположение битов, которое показано в табл. 6.6.

Таблица 6.6. Восьмеричные значения битов SUID, SGID и бита фиксации в команде `chmod`

Двоичное значение	Восьмеричное значение	Описание
000	0	Все биты очищены
001	1	Бит фиксации установлен
010	2	Бит SGID установлен
011	3	Биты SGID и фиксации установлены
100	4	Бит SUID установлен
101	5	Биты SUID и фиксации установлены
110	6	Биты SUID и SGID установлены
111	7	Все биты установлены

Таким образом, чтобы создать общий каталог, в котором всегда происходит задание группы каталога для всех новых файлов, достаточно лишь установить бит SGID для этого каталога:

```
$ mkdir testdir
$ ls -l
drwxrwxr-x  2 rich      rich          4096 Sep 20 23:12 testdir/
$ chgrp shared testdir
$ chmod g+s testdir
$ ls -l
drwxrwsr-x  2 rich      shared        4096 Sep 20 23:12 testdir/
$ umask 002
$ cd testdir
$ touch testfile
$ ls -l
total 0
-rw-rw-r--  1 rich      shared          0 Sep 20 23:13 testfile
$
```

Первый шаг состоит в создании каталога, предназначенного для совместного использования, с помощью команды `mkdir`. После этого применяется команда `chgrp`, чтобы задать вместо предусмотренной по умолчанию группы для каталога такую группу, в состав которой входят пользователи, нуждающиеся в совместном использовании файлов. Наконец, для каталога устанавливается бит SGID с той целью, чтобы применительно ко всем файлам, создаваемым в каталоге, использовалось общее имя группы в качестве заданной по умолчанию группы.

Такая среда может функционировать должным образом лишь при том условии, что для всех членов группы значения пользовательской маски будут установлены на создание файлов, в которых разрешена запись для членов группы. В предыдущем примере показано, как значение `umask` изменено на 002, в результате чего создаваемые файлы становятся предназначенными для записи в группе.

После завершения всех этих операций любой член группы получает возможность перейти в общий каталог и создать новый файл. Как и следовало ожидать, для каждого нового файла

используется заданная по умолчанию группа каталога, а не заданная по умолчанию группа учетной записи пользователя. После этого любой пользователь в группе совместного доступа может обратиться к данному файлу.

Резюме

В настоящей главе приведено описание команд командной строки, которые применяются для управления средствами безопасности доступа Linux в конкретной системе. В Linux для защиты доступа к файлам, каталогам и устройствам используется система идентификаторов пользователей и идентификаторов групп. В Linux информация об учетных записях пользователей хранится в файле `/etc/passwd`, а информация о группах — в файле `/etc/group`. Каждому пользователю присваивается уникальный числовой идентификатор пользователя, наряду с текстовым регистрационным именем, для обозначения его в системе. Группам также присваиваются уникальные числовые идентификаторы групп и текстовые имена групп. В группу можно включить одного или нескольких пользователей, чтобы разрешить совместный доступ к ресурсам системы.

Для управления учетными записями пользователей и групп предусмотрен целый ряд команд. Команда `useradd` позволяет создавать новые учетные записи пользователей, а команда `groupadd` дает возможность создавать новые учетные записи групп. Для изменения существующей учетной записи пользователя применяется команда `usermod`. Аналогичным образом, команда `groupmod` предназначена для изменения информации учетной записи группы.

В Linux используется сложная система битовых обозначений для определения разрешений на доступ к файлам и каталогам. Для защиты каждого файла предусмотрены три уровня безопасности: уровень владельца файла, уровень заданной по умолчанию группы с доступом к файлу и еще один уровень, который относится ко всем прочим пользователям в системе. Каждый уровень безопасности определен тремя битами доступа: доступ для чтения, доступ для записи и доступ для выполнения. Сочетание этих трех битов часто обозначается символами `rwX`, которые сокращенно представляют чтение, запись и выполнение (`read`, `write`, `execute`). Если какое-либо разрешение не дано, вместо соответствующего ему символа ставится тире (например, обозначение `r--` показывает, что разрешено только чтение).

Вместо этих символических обозначений часто применяются восьмеричные обозначения, в которых каждые три бита разрешений объединяются в одно восьмеричное значение, а все три восьмеричные значения представляют три уровня безопасности. Для определения заданных по умолчанию параметров безопасности для файлов и каталогов, создаваемых в системе, используется команда `umask`. Как правило, предусмотренное по умолчанию значение пользовательской маски системный администратор задает в файле `/etc/profile`, но предусмотрена возможность в любое время воспользоваться командой `umask`, чтобы сменить для себя применяемое значение пользовательской маски.

Для изменения параметров безопасности, относящихся к файлам и каталогам, предназначена команда `chmod`. Задавать другие разрешения для файлов и каталогов может только владелец этих файлов или каталогов. Тем не менее изменить параметры безопасности для любого файла или каталога в системе может также пользователь `root`. Для смены заданного по умолчанию владельца и группы файла могут использоваться команды `chown` и `chgrp`.

Наконец, в заключение этой главы приведено описание того, как использовать заданный бит идентификатора группы (SGID) для создания общего каталога. Применение бита SGID приводит к тому, что для всех новых файлов или каталогов, создаваемых в конкретном каталоге, принудительно используется заданное по умолчанию имя группы родительского каталога,

а не имя группы создающего их пользователя. Благодаря этому может быть реализован простой способ обеспечения совместного доступа к файлам для пользователей системы.

Таким образом, наш читатель стал настоящим специалистом по работе с правами доступа к файлам, поэтому теперь мы можем приступить к подробному описанию того, как обращаться с реальной файловой системой Linux. В следующей главе показано, как создавать новые разделы в системе Linux из командной строки и как их форматировать для дальнейшего включения в виртуальный каталог Linux.

Управление файловыми системами

При подготовке к работе в системе Linux необходимо принять ряд решений, в частности, о том, какие файловые системы должны использоваться для запоминающих устройств. Большинство дистрибутивов Linux обычно предлагает заданную по умолчанию файловую систему во время установки, и многие начинающие пользователи Linux просто подтверждают этот выбор по умолчанию, не уделяя этому вопросу большого внимания.

Разумеется, предусмотренный по умолчанию вариант файловой системы, как правило, является вполне приемлемым, но иногда не мешает также ознакомиться с другими возможными вариантами. В настоящей главе рассматриваются различные варианты файловых систем, применяемых в Linux, и показано, как создать эти системы и управлять ими из командной строки Linux.

Общие сведения о файловых системах Linux

Как было описано в главе 3, в Linux *файловые системы* используются для хранения файлов и папок на запоминающих устройствах. Данные хранятся на жестком диске в виде двоичных нулей и единиц, в приложении работа с данными осуществляется путем доступа к файлам и папкам, а файловая система в Linux обеспечивает связь между этими двумя представлениями данных.

ГЛАВА

7

В этой главе...

Общие сведения о файловых системах Linux

Работа с файловыми системами

Диспетчеры логических томов

Резюме

В Linux поддерживается целый ряд файловых систем, позволяющих управлять файлами и папками. Каждая файловая система реализует структуру виртуального каталога на запоминающем устройстве, но при этом в каждой из них применяется немного иной подход. В настоящем разделе рассматриваются преимущества и недостатки файловых систем, наиболее широко применяемых в среде Linux.

Основные файловые системы Linux

В первых версиях Linux использовалась простая файловая система, которая имитировала функциональные возможности файловой системы Unix. В данном разделе описано, как происходило развитие этой файловой системы.

Файловая система ext

Файловая система, первоначально введенная в ОС Linux, именовалась *расширенной файловой системой* (или сокращенно ext). Это была файловая система для Linux, подобная таковой для Unix, в которой для работы с физическими устройствами использовались виртуальные каталоги, а данные хранились в блоках фиксированной длины на физических устройствах.

В файловой системе ext используются структуры, именуемые *индексными узлами*, для отслеживания информации о файлах, хранимых в виртуальном каталоге. В системе индексных узлов на каждом физическом устройстве создается отдельная таблица, называемая *таблицей индексных узлов*, для хранения информации о файлах. Для каждого файла, хранимого в виртуальном каталоге, предусмотрена отдельная запись в таблице индексных узлов. Само название этой файловой системы как *расширенной* определяется тем, что в ней для отслеживания каждого файла хранятся дополнительные данные, которые состоят из следующих фрагментов данных:

- имя файла;
- размер файла;
- владелец файла;
- группа, к которой принадлежит файл;
- разрешения на доступ для файла;
- указатели на каждый дисковый блок, который содержит данные из файла.

В системе Linux сопровождаются ссылки на каждый индексный узел в таблице индексных узлов, для которых используются уникальные номера (называемые *номерами индексных узлов*), присваиваемые файловой системой в ходе создания файлов данных. В файловой системе номера индексных узлов служат для идентификации файлов, что позволяет обойтись без использования полного имени и пути к файлу.

Файловая система ext2

Исходная файловая система ext характеризовалась наличием существенных ограничений, в частности, размеры файлов в ней ограничивались лишь значением 2 Гбайт. Поэтому вскоре после появления первых версий Linux файловая система ext была доработана в целях создания второй расширенной файловой системы, получившей имя *ext2*.

Как и можно было предположить, в файловой системе ext2 дополнены основные возможности файловой системы ext, но применяемая структура хранения файлов осталась неизменной. В файловой системе ext2 расширен формат таблицы индексных узлов для отслеживания дополнительной информации о каждом файле в системе.

В таблицу индексных узлов ext2 добавлены значения времени создания, изменения и последнего доступа для файлов, для того чтобы системные администраторы могли следить за доступом к файлам в системе. Кроме того, в файловой системе ext2 увеличен максимально допустимый размер файла до 2 Тбайт (затем в последующих версиях ext2 это значение возросло до 32 Тбайт), поэтому появилась возможность обеспечить работу с большими файлами, обычно применяемыми в серверах баз данных.

Кроме расширения таблицы индексных узлов, в файловой системе ext2 изменился также способ хранения файлов в виде блоков данных. Одним из недостатков файловой системы ext было то, что при записи файла на физическое устройство, как правило, происходило рассеивание блоков, используемых для хранения данных, по всему устройству (это явление получило название *фрагментации*). Фрагментация файлов, хранимых в виде блоков данных, может привести к снижению производительности файловой системы, поскольку приходится затрачивать больше времени для поиска на запоминающем устройстве всех блоков, относящихся к конкретному файлу, и доступа к ним.

Применение файловой системы ext2 способствует сокращению фрагментации, поскольку при сохранении файла выделение для него дисковых блоков происходит в виде групп блоков. Поскольку блоки данных, относящихся к файлу, сгруппированы, в файловой системе не возникает необходимость проводить поиск по всему физическому устройству блоков данных для чтения файла.

Файловая система ext2 применялась по умолчанию в дистрибутивах Linux в течение многих лет, но при этом приходилось постоянно учитывать ее ограничения. Применение таблицы индексных узлов вполне себя оправдывало, поскольку эта таблица позволяет отслеживать в файловой системе дополнительные сведения о файлах, но само ее существование может стать причиной возникновения в системе неустранимых проблем. Дело в том, что при выполнении в файловой системе каждой операции сохранения или обновления файла приходится вносить изменения в таблицу индексных узлов с учетом новых сведений о файлах. Но в связи с этим возникает такой недостаток, что это действие не всегда является неразрывным.

Если в компьютерной системе происходит какой-то сбой в промежутке между сохранением файла и обновлением таблицы индексных узлов, то нарушается синхронизация между таблицей и файловой системой. Файловая система ext2 приобрела печальную известность тем, что может быть легко повреждена из-за аварийных отказов системы и прекращения подачи электроэнергии. Если операция сохранения файла с данными на физическом устройстве будет выполнена вполне приемлемо, то при неудачном завершении операции записи в таблицу индексных узлов в файловой системе ext2 не будет даже сведений о том, что этот файл существует!

Такая ситуация вскоре вынудила разработчиков к исследованию других возможных вариантов создания файловых систем Linux.

Файловые системы с ведением журнала

Файловые системы с ведением журнала позволили подняться на более высокий уровень безопасности работы с данными в системе Linux. Вместо записи данных непосредственно на запоминающее устройство с последующим обновлением таблицы индексных узлов в файловых системах с ведением журнала запись изменений в файле сначала происходит во временный файл (называемый *журналом*). После того как данные успешно записываются на запоминающее устройство и в таблицу индексных узлов, происходит удаление записи журнала.

Если в системе происходит аварийный останов или прекращается подача электроэнергии до того, как удастся записать данные на запоминающее устройство, то после возобновления работы в файловой системе с ведением журнала достаточно лишь быстро прочитать файл журнала и обработать все оставшиеся незафиксированные данные.

В Linux, как правило, применяются три различных метода ведения журнала, обеспечивающие разные уровни защиты (табл. 7.1).

Таблица 7.1. Методы ведения журнала в файловой системе

Метод	Описание
Режим данных	Ведение журнала предусмотрено и для индексных узлов, и для данных файлов. Малый риск потери данных, но низкая производительность
Упорядоченный режим	В журнал записываются только данные индексного узла, но не удаляются до успешной записи данных файла. Достигается удачный компромисс между производительностью и безопасностью
Режим отложенной записи	В журнал записываются только данные индексного узла и не ведется контроль над тем, как происходит запись данных файла. Более высокий риск потери данных, но все же меньшая опасность этого, чем при отказе от использования журнала

Метод ведения журнала с режимом данных, безусловно, является наиболее безопасным с точки зрения защиты данных, но характеризуется также самой низкой производительностью. Все данные, подлежащие сохранению на запоминающем устройстве, должны быть записаны дважды, один раз — в журнале, а второй раз — на устройстве физической памяти. Это может стать причиной снижения производительности, особенно в тех системах, в которых применяется значительное количество операций записи данных.

За эти годы в Linux появилось несколько различных файловых систем с ведением журнала. В следующих разделах описаны файловые системы с ведением журнала, наиболее широко применяемые в Linux.

Расширенные файловые системы Linux с ведением журнала

Той же группе разработчиков, которой были созданы файловые системы `ext` и `ext2` в составе проекта Linux, удалось также создать несколько версий файловых систем с ведением журнала. Эти файловые системы с ведением журнала совместимы с файловой системой `ext2`, поэтому позволяют легко проводить прямое и обратное преобразования между ними. В настоящее время применяются две отдельные файловые системы с ведением журнала на основе файловой системы `ext2`.

Файловая система `ext3`

Средства поддержки файловой системы `ext3` были добавлены к ядру Linux в 2001 году, и вплоть до недавнего времени эта файловая система применялась по умолчанию почти во всех дистрибутивах Linux. В этой файловой системе используется такая же структура таблицы индексных узлов, как и в файловой системе `ext2`, но на каждом запоминающем устройстве дополнительно используется файл журнала для внесения в него на время тех данных, которые сохраняются на запоминающем устройстве.

По умолчанию в файловой системе `ext3` применяется упорядоченный метод режима ведения журнала, при котором в файл журнала записывается только информация индексного узла, но удаление этой информации не происходит до тех пор, пока не будет успешно выполнена запись блоков данных на запоминающее устройство. Предусмотрена возможность выбрать другой метод ведения журнала, используемый в файловой системе `ext3`, указав либо режим записи данных, либо режим отложенной записи с применением простого параметра командной строки при создании файловой системы.

Безусловно, с внедрением файловой системы ext3 удалось добавить основные средства ведения журнала к файловой системе Linux, но все же в ней недоставало некоторых важных функций. Например, файловая система ext3 не предоставляет никакой возможности восстановления случайно удаленных файлов, в ней отсутствуют встроенные средства сжатия данных (хотя и есть обновление, которое может быть установлено отдельно, предоставляющее эту возможность), к тому же эта файловая система ext3 не поддерживает шифрование файлов. По этим причинам разработчики, участвующие в проекте Linux, решили продолжить работу над усовершенствованием файловой системы ext3.

Файловая система ext4

Файловая система *ext4* (как и можно было предположить) стала результатом работ по усовершенствованию файловой системы ext3. Поддержка файловой системы ext4 в ядре Linux была официально введена в 2008 году, а теперь эта файловая система используется по умолчанию в наиболее широко применяемых дистрибутивах Linux, таких Fedora и Ubuntu.

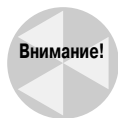
Кроме поддержки сжатия и шифрования, файловая система ext4 поддерживает также средство, именуемое *экстентами*. Благодаря экстентам появляется возможность выделять пространство на запоминающем устройстве в виде нескольких блоков и сохранять в таблице индексных узлов только сведения о местоположении начального блока. Это позволяет экономить пространство в таблице индексных узлов, поскольку отпадает необходимость сохранять перечень всех блоков данных, используемых для хранения данных файла.

Кроме того, в файловой системе ext4 предусмотрены средства *предварительного распределения блоков*. Если требуется зарезервировать на запоминающем устройстве пространство для файла, который, как известно, будет увеличиваться в размерах, то файловая система ext4 позволяет заранее выделить для файла все блоки, намеченные для использования, а не только блоки, в которых располагаются имеющиеся данные файла. В файловой системе ext4 зарезервированные блоки данных заполняются нулями и учитывается то, что эти блоки нельзя выделять для какого-либо другого файла.

Файловая система Рейзера

В 2001 году Ганс Рейзер (Hans Reiser) создал первую файловую систему с ведением журнала для Linux, которая получила название *ReiserFS*. Файловая система с ведением журнала ReiserFS поддерживает только режим отложенной записи, в котором в файл журнала записываются только данные таблицы индексных узлов. В связи с тем, что в журнале фиксируются только данные таблицы индексных узлов, файловая система ReiserFS является одной из самых быстросуществующих файловых систем с ведением журнала в Linux.

В файловой системе ReiserFS реализованы две привлекательные функции: возможность изменять размеры существующей файловой системы без прекращения ее активной эксплуатации и применение так называемой методики *размещения в последнем блоке*, которая обеспечивает запись данных одного файла в пустом пространстве последнего блока данных другого файла. Возможность изменять размеры активной файловой системы становится очень важной, если возникает необходимость расширить уже созданную файловую систему для приспособления ее к большим объемам данных.

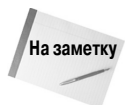


Поскольку Ганс Рейзер столкнулся с судебным преследованием, в настоящее время статус файловой системы ReiserFS находится под вопросом. Несмотря на то что проект ReiserFS относится к категории разработок с открытым исходным кодом, Ганс был ведущим разработчиком этого проекта, а сейчас находится в тюрьме. В последнее время в части файловой системы ReiserFS не проводятся какие-либо разработки и не ясно,

возьмет ли кто-либо еще на себя ответственность за дальнейшее продолжение этого проекта. Учитывая то, что в настоящее время имеются возможности применения других файловых систем с ведением журнала, в большинстве новых установок Linux файловая система ReiserFS не используется. Однако на практике все еще могут встретиться существующие системы Linux, в которых она применяется.

Файловая система с ведением журнала

По-видимому, старейшей из всех существующих файловых систем с ведением журнала является *JFS* (Journaled File System — файловая система с ведением журнала), которая была разработана корпорацией IBM в 1990 году для собственной версии Unix, AIX Unix. Но эта файловая система была перенесена в среду Linux только после появления ее второй версии.



Официальным названием второй версии файловой системы JFS, присвоенным корпорацией IBM, является JFS2, но в большинстве систем Linux она именуется просто как JFS.

В файловой системе JFS используется упорядоченный метод ведения журнала, который предусматривает хранение в журнале только данных таблицы индексных узлов, но удаления записей этой таблицы не происходит до тех пор, пока не завершается операция сохранения фактических данных файла на запоминающем устройстве. Этот метод позволяет достичь компромисса между быстродействием ReiserFS и целостностью данных, достигаемой в режиме сохранения данных метода с ведением журнала.

В файловой системе JFS предусмотрено размещение файлов на основе экстенгов, при котором выделяется группа блоков для каждого файла, сохраняемого на запоминающем устройстве. Благодаря применению такого метода удастся уменьшить фрагментацию на запоминающем устройстве.

Файловая система JFS чаще всего встречается в основном в версиях Linux, предлагаемых корпорацией IBM, но нельзя гарантировать, что с ней не придется столкнуться при изучении тех или иных версий Linux.

Файловая система XFS

Файловая система с ведением журнала *XFS* представляет собой еще одну файловую систему, первоначально созданную для коммерческой системы Unix, которая проложила свой путь в мир Linux. Система XFS была создана в корпорации SGI (Silicon Graphics Incorporated) в 1994 году для собственной коммерческой системы IRIX Unix. А для среды Linux аналогичная система была создана для общего использования в 2002 году.

В файловой системе XFS применяется режим ведения журнала с отложенной записью, который обеспечивает высокую производительность, но создает определенный риск того, что фактические данные могут оказаться не записанными в файл журнала. Файловая система XFS также обеспечивает оперативное изменение размеров файловой системы, аналогично файловой системе ReiserFS, если не считать того, что файловая система XFS позволяет только увеличивать, но не уменьшать размеры.

Работа с файловыми системами

В Linux предусмотрено несколько различных программ, которые обеспечивают возможность работы с файловыми системами из командной строки. Эти программы предоставляют удобные возможности добавления новых файловых систем или изменения существующих

файловых систем путем ввода данных с клавиатуры. В настоящем разделе приведено краткое описание команд, предназначенных для обеспечения взаимодействия с файловыми системами из среды командной строки.

Создание разделов

Для размещения файловой системы необходимо прежде всего создать *раздел* на запоминающем устройстве. Раздел может занимать весь диск или часть диска, содержащую определенный фрагмент виртуального каталога.

Для организации разделов на любом запоминающем устройстве, установленном в системе, служит программа `fdisk`. Команда `fdisk` вызывает на выполнение программу, работающую в интерактивном режиме, которая позволяет вводить команды и проходить с их помощью по этапам создания раздела на жестком диске.

При запуске команды `fdisk` необходимо указать имя устройства для запоминающего устройства, на котором должен быть выделен раздел:

```
$ sudo fdisk /dev/sdc
[sudo] password for rich:
Device contains neither a valid DOS partition table, nor Sun, SGI or
OSF disklabel
Building a new DOS disklabel with disk identifier 0x4beedc66.
Changes will remain in memory only, until you decide to write them.
After that, of course, the previous content won't be recoverable.

Warning: invalid flag 0x0000 of partition table 4 will be corrected
by w(rite)

WARNING: DOS-compatible mode is deprecated. It's strongly recommended
        To switch off the mode (command 'c') and change display
        units to sectors (command 'u').
```

Command (m for help):

Если выделение разделов на запоминающем устройстве происходит впервые, то `fdisk` выводит предупреждение, которое указывает, что на устройстве нет таблицы разделов.

В интерактивной командной строке `fdisk` используются однобуквенные команды для передачи команде `fdisk` инструкций по выполнению очередных операций. В табл. 7.2 приведены команды, которые могут применяться в командной строке `fdisk`.

Таблица 7.2. Команды `fdisk`

Команда	Описание
a	Включить или отключить флаг, указывающий, является ли раздел загрузочным
b	Изменить метку диска, используемую в системах BSD Unix
c	Включить или отключить флаг совместимости с DOS
d	Удалить раздел
l	Вывести список доступных типов разделов
m	Показать параметры команды
n	Добавить новый раздел
o	Создать таблицу разделов DOS

Команда	Описание
p	Вывести на дисплей текущую таблицу разделов
q	Завершить работу без сохранения изменений
s	Создать новую метку диска для систем Sun Unix
t	Изменить системный идентификатор раздела
u	Изменить используемые единицы хранения
v	Проверить таблицу разделов
w	Записать таблицу разделов на диск
x	Усовершенствованные функции

Безусловно, на первый взгляд создается впечатление, что освоить все команды из этого списка просто невозможно, но в действительности в повседневной работе обычно применяется лишь несколько основных команд.

Для начала можно вывести на экран подробные сведения о запоминающем устройстве с помощью команды p:

```
Command (m for help): p
```

```
Disk /dev/sdc: 5368 MB, 5368946688 bytes
255 heads, 63 sectors/track, 652 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x4beedc66
```

Device	Boot	Start	End	Blocks	Id	System
--------	------	-------	-----	--------	----	--------

```
Command (m for help):
```

В выводе этой команды показано, что данное запоминающее устройство имеет объем памяти 5368 Мбайт (5 Гбайт). Вслед за сведениями о запоминающем устройстве отображается листинг, в котором показаны существующие разделы на этом устройстве. В данном примере в листинге не показано ни одного раздела, и это означает, что на устройстве еще не определены разделы.

После этого приступим к решению задачи создания нового раздела на запоминающем устройстве. Для этого предназначена команда n:

```
Command (m for help): n
```

```
Command action
```

```
  e   extended
```

```
  p   primary partition (1-4)
```

```
p
```

```
Partition number (1-4): 1
```

```
First cylinder (1-652, default 1): 1
```

```
Last cylinder, +cylinders or +size{K,M,G} (1-652, default 652): +2G
```

```
Command (m for help):
```

Разделы могут создаваться в качестве *основного* или *расширенного раздела*. Основные разделы могут быть непосредственно отформатированы с созданием файловой системы, тогда

как расширенные предназначены исключительно для развертывания на них других основных разделов. Необходимость в использовании расширенных разделов обусловлена тем, что каждое отдельное запоминающее устройство может содержать только четыре раздела. Но это ограничение можно преодолеть, создавая несколько расширенных разделов, а затем определяя основные разделы в расширенных разделах. В данном примере показано, как создать на запоминающем устройстве первичный раздел, присвоить ему номер раздела 1, а затем выделить для него часть пространства запоминающего устройства объемом 2 Гбайт. Чтобы ознакомиться с полученными результатами, снова воспользуемся командой `p`:

```
Command (m for help): p
```

```
Disk /dev/sdc: 5368 MB, 5368946688 bytes
255 heads, 63 sectors/track, 652 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x4beedc66
```

Device	Boot	Start	End	Blocks	Id	System
/dev/sdc1		1	262	2104483+	83	Linux

```
Command (m for help):
```

Очевидно, что теперь в выводе этой команды присутствует раздел на запоминающем устройстве (с именем `/dev/sdc1`). Запись `Id` определяет тип этого раздела с точки зрения системы Linux. Программа `fdisk` позволяет создавать разделы многих разных типов. Для ознакомления со списком всех допустимых типов предназначена команда `l`. Применяемым по умолчанию типом является тип 83, который определяет файловую систему Linux. Если же потребуется создать раздел для другой файловой системы (например, раздел NTFS для Windows), достаточно лишь выбрать соответствующий тип раздела.

Этот процесс можно повторить для выделения оставшегося пространства на запоминающем устройстве другому разделу Linux. После создания всех необходимых разделов следует применить команду `w`, чтобы сохранить изменения на запоминающем устройстве:

```
Command (m for help): w
```

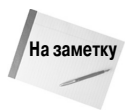
```
The partition table has been altered!
```

```
Calling ioctl() to re-read partition table.
```

```
Syncing disks.
```

```
$
```

Теперь, после создания раздела на запоминающем устройстве, можно приступить к его форматированию в целях определения на нем файловой системы Linux.



Иногда самой трудной задачей при создании нового раздела диска становится поиск предназначенного для этого физического диска в системе Linux. В Linux используется стандартный формат для присваивания имен устройств жестким дискам, и для успешного поиска необходимого устройства нужно быть знакомым с этим форматом. Для IDE более старого поколения в Linux используется обозначение `/dev/hdx`, где `x` — буква диска, присваиваемая в порядке обнаружения дисков (буква `a` присваивается первому обнаруженному в системе диску, `b` — второму и т.д.). Для более новых типов дисков SATA, а также для дисков SCSI в системе Linux используется обозначение `/dev/sdx`, где `x`, аналогичным образом, представляет собой букву диска, присваиваемую в порядке обнаружения дисков (и в этом случае буква `a` относится к первому диску, `b` — ко второму).

и т.д.). Всегда рекомендуется проводить тщательную проверку, чтобы быть уверенным в том, что в команде диск указан правильно, прежде чем приступить к форматированию раздела!

Создание файловой системы

Возможность хранения данных в разделе диска появляется только после форматирования этого раздела с созданием файловой системы, которая могла бы использоваться в Linux. Применительно к каждому типу файловой системы используется отдельная программа командной строки для форматирования разделов. Программы, применяемые для различных файловых систем, рассматриваемых в настоящей главе, приведены в табл. 7.3.

Таблица 7.3. Программы с командной строкой, предназначенные для создания файловых систем

Программа	Назначение
mkfs	Создание файловой системы ext
mke2fs	Создание файловой системы ext2
mkfs.ext3	Создание файловой системы ext3
mkfs.ext4	Создание файловой системы ext4
mkreiserfs	Создание файловой системы ReiserFS
jfs_mkfs	Создание файловой системы JFS
mkfs.xfs	Создание файловой системы XFS

С каждой командой создания файловой системы связан широкий набор параметров командной строки, которые позволяют точно настраивать способ создания файловой системы в разделе. Для ознакомления со всеми имеющимися параметрами командной строки можно воспользоваться командой `man`, которая позволяет вывести на экран справочное руководство, относящееся к требуемой команде создания файловой системы (см. главу 2). Все команды создания файловой системы позволяют сформировать файловую систему по умолчанию с указанием лишь одной команды, без параметров:

```
$ sudo mkfs.ext4 /dev/sdc1
[sudo] password for rich:
mke2fs 1.41.11 (14-Mar-2010)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
Stride=0 blocks, Stripe width=0 blocks
131648 inodes, 526120 blocks
26306 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=541065216
17 block groups
32768 blocks per group, 32768 fragments per group
7744 inodes per group
Superblock backups stored on blocks:
    32768, 98304, 163840, 229376, 294912
```

```

Writing inode tables: done
Creating journal (16384 blocks): done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 34 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
$

```

Во вновь созданной файловой системе используется тип файловой системы ext4, который относится к файловой системе с ведением журнала в Linux. Обратите внимание на то, что в состав процесса создания файловой системы входило создание нового журнала.

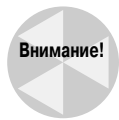
Следующим шагом после создания файловой системы для раздела является ее монтирование в одной из точек монтирования в виртуальном каталоге, поскольку лишь после этого появится возможность сохранять данные в новой файловой системе. Предусмотрена возможность смонтировать новую файловую систему в любом подкаталоге виртуального каталога, где должно быть предоставлено дополнительное пространство.

```

$ sudo mkdir /mnt/testing
$ sudo mount -t ext4 /dev/sdb1 /mnt/testing
$ ls -al /mnt/testing
total 24
drwxr-xr-x 3 root root 4096 2010-09-25 19:25 .
drwxr-xr-x 3 root root 4096 2010-09-25 19:38 ..
drwx----- 2 root root 16384 2010-09-25 19:25 lost+found
$

```

С помощью команды `mkdir` создается точка монтирования в виртуальном каталоге, а команда `mount` служит для добавления нового раздела жесткого диска в точку монтирования. После этого можно приступить к сохранению новых файлов и папок в новом разделе!



Этот метод монтирования файловой системы предназначен лишь для временного монтирования файловой системы. Эта файловая система не будет автоматически монтироваться после каждой перезагрузки системы Linux. Чтобы вынудить Linux автоматически монтировать новую файловую систему во время начальной загрузки, необходимо добавить эту новую файловую систему к файлу `/etc/fstab`.

В настоящей главе показано, как работать с файловыми системами, расположенными на физических запоминающих устройствах. Кроме этого, в Linux предусмотрено несколько различных способов создания логических запоминающих устройств для файловых систем. Сведения о том, как использовать логическое запоминающее устройство для создания файловых систем, приведены в следующем разделе.

Устранение нарушений в работе

Даже в самых совершенных файловых системах с ведением журнала могут возникать сбои, например, если неожиданно возникает нарушение электропитания или вышедшее из-под контроля приложение блокирует систему, в то время как происходит доступ к файлу. К счастью, предусмотрено несколько утилит командной строки, которые способны оказать помощь при восстановлении нормальной работы файловой системы.

Для каждой файловой системы разработана собственная команда восстановления, предназначенная для работы именно с ней. Иными словами, по мере того как увеличивалось количество файловых систем, доступных в среде Linux, происходило расширение набора отдельных

команд, которые требовалось изучать для работы с этими файловыми системами, а это влекло за собой появление дополнительных сложностей. К счастью, была создана программа с общим внешним интерфейсом, которая позволяет определить файловую систему на запоминающем устройстве и применить соответствующую команду восстановления файловой системы с учетом типа восстанавливаемой файловой системы.

Для проверки и восстановления файловых систем Linux любого типа, включая все описанные ранее в данной главе файловые системы ext, ext2, ext3, ext4, ReiserFS, JFS и XFS, предназначена команда `fsck`, имеющая следующий формат:

```
fsck options filesystem
```

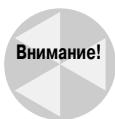
В командной строке вызова этой команды можно указать несколько *файловых систем*, подлежащих проверке. Файловые системы могут быть указаны с использованием имени устройства, точки монтирования в виртуальном каталоге или специального значения UUID в системе Linux, присвоенного файловой системе.

В команде `fsck` используется файл `/etc/fstab` для автоматического определения типа файловой системы на запоминающем устройстве, на котором обычно монтируется эта система. Если при обычных обстоятельствах запоминающее устройство остается не смонтированным (например, если еще только происходит создание файловой системы на новом запоминающем устройстве), то необходимо задавать параметр командной строки `-t` для указания типа файловой системы. Другие применимые параметры командной строки приведены в табл. 7.4.

Таблица 7.4. Параметры командной строки `fsck`

Параметр	Описание
<code>-a</code>	Автоматически восстанавливать файловую систему при обнаружении ошибок
<code>-A</code>	Проверить все файловые системы, перечисленные в файле <code>/etc/fstab</code>
<code>-C</code>	Отобразить индикатор хода работы для файловых систем, которые поддерживают это средство (только ext2 и ext3)
<code>-N</code>	Не проводить проверку, а только показать, какие проверки были бы выполнены
<code>-r</code>	Вывести приглашение для подтверждения того, что должны быть внесены исправления при обнаружении ошибок
<code>-R</code>	Пропустить корневую файловую систему, если используется параметр <code>-A</code>
<code>-s</code>	Если должно быть проверено несколько файловых систем, проверять их по одной
<code>-t</code>	Указать тип файловой системы, подлежащей проверке
<code>-T</code>	Не показывать информацию заголовка при запуске
<code>-V</code>	Формировать подробный вывод в течение проверки
<code>-y</code>	Автоматически восстанавливать файловую систему при обнаружении ошибок

Вполне очевидно, что некоторые из этих параметров командной строки являются избыточными. Причина их появления обусловлена попытками реализовать общий внешний интерфейс для поддержки нескольких различных команд, о чем было сказано выше. Некоторые отдельные команды восстановления файловой системы имеют дополнительные применимые параметры. Если возникает необходимость в проведении дополнительной проверки на наличие ошибок, то можно ознакомиться со справочными руководствами к отдельным инструментам исправления файловой системы, чтобы узнать, какие расширенные параметры относятся к этой файловой системе.



Команда `fsck` может быть выполнена лишь применительно к размонтированным файловым системам. При работе с большинством файловых систем на компьютере достаточно просто размонтировать соответствующую файловую систему, провести ее проверку, а затем повторно смонтировать после завершения этого этапа. Однако корневая файловая система содержит все основные программы Linux и файлы журналов, поэтому ее нельзя размонтировать, не останавливая работу системы.

Именно в этих обстоятельствах вполне может пригодиться версия LiveCD системы Linux! Достаточно просто загрузить систему с LiveCD, а затем выполнить команду `fsck` применительно к корневой файловой системе!

Диспетчеры логических томов

Если применяемые файловые системы созданы на основе стандартных разделов на жестких дисках, то после исчерпания свободного пространства попытка ввести дополнительный объем в существующую файловую систему может оказаться весьма трудоемкой. Раздел может быть расширен только за счет доступного пространства на том же физическом жестком диске, где находится этот раздел. Если же на этом жестком диске больше нет свободного места, то остается лишь один выход: приобрести жесткий диск большего объема и вручную переместить существующую файловую систему на новый диск.

Вместо этого был бы намного удобнее способ, позволяющий динамически присоединять дополнительное пространство к существующей файловой системе, например, путем добавления раздела с другого жесткого диска к этой файловой системе. В Linux предусмотрен пакет программ LVM (Logical Volume Manager — диспетчер логических томов), позволяющий осуществлять именно этот способ. Программы LVM предоставляют возможность легко манипулировать дисковым пространством в системе Linux, не сталкиваясь с необходимостью перестраивать все файловые системы.

Организация управления логическими томами

В основе управления логическими томами лежит применяемый в этой системе способ работы с физическими разделами жестких дисков, установленными в системе. В терминологии управления логическими томами жесткие диски именуются *физическими томами* (physical volume — PV). Каждый физический том отображается на конкретный физический раздел, созданный на жестком диске.

Многочисленные физические тома, рассматриваемые как элементы системы, объединяются в пулы, создавая *группы томов* (volume group — VG). В системе управления логическими томами группа томов рассматривается как один физический жесткий диск, но в действительности группа томов может состоять из нескольких физических разделов, распределенных по нескольким жестким дискам. Группа томов предоставляет платформу для создания логических разделов, которые фактически содержат файловую систему.

Заключительным уровнем в этой структуре является *логический том* (logical volume — LV). Логические тома создают среду определения разделов для Linux, в которой создаются файловые системы, действуя полностью аналогично физическим разделам жестких дисков с точки зрения средств поддержки запоминающих устройств в системе Linux. В системе Linux логический том рассматривается точно так же, как физический раздел. Предусмотрена возможность отформатировать логический том с созданием любой из стандартных файловых систем Linux, а затем добавить эту файловую систему к виртуальному каталогу Linux в точке монтирования.

Основная организация типичной среды управления логическими томами Linux показана на рис. 7.1.



Рис. 7.1. Среда управления логическими томами

Группа томов, показанная на рис. 7.1, охватывает три отдельных физических жестких диска, занимая пять отдельных физических разделов. В группу томов входят два отдельных логических тома. В системе Linux каждый логический том рассматривается как полностью соответствующий физическому разделу. Каждый логический том может быть отформатирован с созданием файловой системы ext4, а затем смонтирован в конкретном местоположении в виртуальном каталоге.

Обратите внимание на то, что на рис. 7.1 третий физический жесткий диск имеет неиспользуемый раздел. С помощью средств управления логическими томами можно без затруднений в дальнейшем присоединить этот неиспользуемый раздел к существующей группе томов, а затем либо воспользоваться им для создания нового логического тома, либо добавить его для увеличения объема к одному из существующих логических томов, если потребуется увеличить объем доступного пространства.

Аналогичным образом, если на компьютере будет установлен новый жесткий диск, локальная система управления томами предоставит возможность добавить его к существующей группе томов, а затем отвести большее пространство для одного из существующих логических томов или создать новый логический том, который можно будет монтировать отдельно. Очевидно, что такой способ работы с файловыми системами, требующими расширения объема, является намного более удобным!

Применение программы LVM в системе Linux

Программа Linux LVM была разработана Хайнцем Мауэльсхагеном (Heinz Mauelshagen) и передана в распоряжение сообщества Linux в 1998 году. Эта программа позволяет управлять всей средой управления логическими томами с использованием простых команд командной строки Linux.

Имеются две версии программы Linux LVM.

- LVM1. Оригинальный пакет LVM, выпущенный в 1998 году и применяемый только в версии 2.4 ядра Linux. Этот пакет предоставляет лишь основные средства управления логическими томами.
- LVM2. Модифицированный вариант LVM, доступный в версии 2.6 ядра Linux. Этот пакет предоставляет дополнительные возможности по сравнению со стандартными средствами LVM1.

В большинстве современных дистрибутивов Linux, основанных на версии 2.6 ядра Linux, предусмотрена поддержка пакета LVM2. Кроме стандартных средств управления логическими томами, LVM2 предоставляет также другие привлекательные средства, которые могут применяться в конкретной системе Linux.

Моментальные снимки

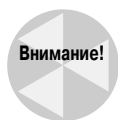
Исходная версия Linux LVM позволяет создать копию существующего логического тома на другом устройстве, не выводя этот логический том из активного состояния. Это средство принято называть средством создания *моментального снимка*. Моментальные снимки предоставляют великолепную возможность создания резервных копий важных данных, которые не могут быть заблокированы на время выполнения операции резервного копирования в связи с наличием требований к высокому уровню доступности. Обычные методы резервного копирования, как правило, требуют блокировки файлов, копируемых на резервные носители информации. Средства создания моментальных снимков позволяют продолжать эксплуатацию важных веб-серверов или серверов баз данных в процессе получения копии. К сожалению, версия LVM1 позволяет создавать лишь моментальные снимки, предназначенные только для чтения. Возможность выполнять операции записи по отношению к моментальному снимку после его создания не предусмотрена.

С другой стороны, версия LVM2 дает возможность создать моментальный снимок активного логического тома, предназначенный для чтения и записи. После получения копии для чтения и записи можно отключить исходный логический том и смонтировать моментальный снимок в качестве его замены. Это средство является буквально неоценимым при создании способов быстрого возобновления после сбоя или при проведении экспериментов с приложениями, вносящими изменения в такие данные, которые могут потребовать восстановления при возникновении каких-либо нарушений в работе.

Полосовое распределение

Еще одним привлекательным средством, предоставляемым программой LVM2, является *полосовое распределение*. Средства полосового распределения позволяют создавать логические тома, распределенные по нескольким физическим жестким дискам. При записи файла на логическом томе в программе Linux LVM происходит распределение блоков данных из этого файла по нескольким жестким дискам. Каждый очередной подряд идущий блок данных записывается на следующий жесткий диск.

Полосовое распределение позволяет повысить производительность операций с дисками, поскольку система Linux может одновременно записывать несколько блоков данных из файла на нескольких жестких дисках, не ожидая завершения записи на одном жестком диске для перемещения головки чтения-записи в очередное местоположение перед выводом следующего блока. Благодаря этому усовершенствованию ускоряется также чтение файлов в режиме последовательного доступа, поскольку программа LVM может читать данные одновременно с нескольких жестких дисков.



Полосовое распределение LVM не следует путать с полосовым распределением RAID. При полосовом распределении LVM не формируется запись контроля четности, необходимая для создания отказоустойчивой среды. В действительности применение полосового распределения LVM может привести к повышению вероятности потери файлов из-за отказа жесткого диска. Сбой даже одного диска может привести к тому, что несколько логических томов станут недоступными.

Зеркальное отображение

Одно лишь то, что файловая система устанавливается с использованием LVM, не означает, что больше не могут произойти нарушения в ее работе. Точно так же как физические разделы, логические тома LVM восприимчивы к прекращению подачи электроэнергии и аварийным отказам дисков. А после того как возникают повреждения в файловой системе, всегда есть вероятность, что ее не удастся восстановить.

Немного большую уверенность придает процесс создания моментальных снимков LVM, поскольку он позволяет в любое время создать резервную копию логического тома, но для некоторых вариантов среды этого может быть недостаточно. В системах, по отношению к которым выполняется большой объем операций изменения в данных, таких как серверы баз данных, может оказаться, что со времени получения последнего моментального снимка были сохранены сотни или даже тысячи записей.

Решение этой проблемы состоит в использовании средств создания *зеркальной копии* LVM. Зеркальная копия — это полная копия логического тома, которая обновляется в режиме реального времени. Во время создания зеркального логического тома программа LVM синхронизирует исходный логический том с зеркальной копией. Для выполнения этой операции может потребоваться определенное время, в зависимости от размера исходного логического тома.

После завершения исходного этапа синхронизации программа LVM выполняет по две операции записи в ответ на каждую команду в процессе записи в файловой системе: одну операцию — в основном логическом томе и вторую — в копии с зеркальным отображением. Как и можно было предположить, применение этого процесса приводит к снижению производительности операций записи в системе. Тем не менее, если исходный логический том по каким-то причинам становится поврежденным, у вас под рукой всегда есть полная актуальная копия!

Использование Linux LVM

Выше были описаны основные функции пакета Linux LVM, а в настоящем разделе показано, как реализовать эти функции, чтобы наилучшим образом использовать дисковое пространство в системе. В пакете Linux LVM предусмотрены только программы командной строки, предназначенные для создания всех компонентов системы управления логическими томами и контроля над ними. Некоторые дистрибутивы Linux включают графические интерфейсы для этих команд командной строки, но, чтобы научиться полностью контролировать применяемую среду LVM, следует хорошо освоить работу непосредственно с этими командами.

Определение физических томов

Первым шагом в этом процессе является преобразование физических разделов на жестком диске в экстенды физических томов, используемые в программе Linux LVM. В этом может помочь уже известная нам команда `fdisk`. После создания основного раздела Linux необходимо изменить тип этого раздела с помощью команды `t`:

```
Command (m for help): t
Selected partition 1
Hex code (type L to list codes): 8e
Changed system type of partition 1 to 8e (Linux LVM)

Command (m for help): p

Disk /dev/sdc: 2147 MB, 2147992064 bytes
255 heads, 63 sectors/track, 261 cylinders
```

```
Units = cylinders of 16065 * 512 = 8225280 bytes
Disk identifier: 0x4bc26133
```

Device	Boot	Start	End	Blocks	Id	System
/dev/sdc1		1	261	2096451	8e	Linux LVM

Command (m for help):

Разделу должен быть присвоен тип 8e, который показывает, что раздел должен использоваться в составе системы Linux LVM, а не как непосредственно доступная файловая система, работа с которой была показана выше на примере раздела типа 83.

Следующий шаг состоит в использовании этого раздела для создания действительного физического тома. Для этого предназначена команда `pvcreate`:

```
$ sudo pvcreate /dev/sdc1
Physical volume "/dev/sdc1" successfully created
$
```

Команда `pvcreate` определяет физический том, используемый в качестве объекта типа PV. Она просто отмечает раздел как физический том для системы Linux LVM. Для ознакомления со списком созданных физических томов можно воспользоваться командой `pvdisk`, если возникает необходимость проверить ход работы по подготовке системы Linux LVM.

Создание групп томов

Очередным шагом в этом процессе является создание одной или нескольких групп томов из ранее подготовленных физических томов. Не установлены какие-либо правила в отношении того, сколько групп томов должно быть создано для конкретной системы; можно ввести все имеющиеся физические тома в одну группу томов или создать несколько групп томов, применяя существующие физические тома в различных сочетаниях.

Чтобы создать группу томов из командной строки, необходимо вызвать на выполнение команду `vgcreate`. Для команды `vgcreate` необходимо задать несколько параметров командной строки, которые определяют имя группы томов, а также имена физических томов, используемых для создания группы томов:

```
$ sudo vgcreate Vol1 /dev/sdc1
Volume group "Vol1" successfully created
$
```

Очевидно, что эти выходные данные не содержат большого объема сведений! Если есть необходимость получить дополнительные данные об этой вновь созданной группе томов, то можно воспользоваться командой `vgdisplay`:

```
$ sudo vgdisplay
--- Volume group ---
VG Name                Vol1
System ID
Format                  lvm2
Metadata Areas          1
Metadata Sequence No    1
VG Access                read/write
VG Status                resizable
MAX LV                  0
Cur LV                  0
Open LV                  0
```

```

Max PV          0
Cur PV         1
Act PV         1
VG Size        2.00 GB
PE Size        4.00 MB
Total PE       511
Alloc PE / Size 0 / 0
Free PE / Size  511 / 2.00 GB
VG UUID        CyilHZ-Y840-8TUn-Wvti-4S6Q-bNHT-C113IO

```

\$

В этом примере создается группа томов с именем `Vol1` с применением физического тома, созданного на основе раздела `/dev/sdc1`.

Теперь, после создания одной или нескольких групп томов, можно приступить к определению логического тома.

Создание логических томов

Именно логические тома используются в системе Linux для эмуляции физического раздела и размещения файловой системы. В системе Linux логический том рассматривается как полностью аналогичный физическому разделу, что позволяет определить файловую систему на логическом томе, а затем смонтировать эту файловую систему в виртуальном каталоге.

Для создания логического тома используется команда `lvcreate`. Как показано выше, при выполнении некоторых команд Linux LVM обычно можно обойтись без использования параметров командной строки, но команда `lvcreate` требует ввода по крайней мере некоторых параметров. Применимые параметры командной строки приведены в табл. 7.5.

Таблица 7.5. Параметры команды `lvcreate`

Параметр	Длинное имя параметра	Описание
-c	--chunksize	Указать размер фрагмента логического тома моментального снимка
-C	--contiguous	Задать или переустановить политику смежного размещения
-i	--stripes	Указать количество элементов полосового распределения
-I	--stripsize	Указать размер каждого элемента полосового распределения
-l	--extents	Указать количество логических экстенгов, которые должны быть выделены для нового логического тома, или процентную долю используемых логических экстенгов
-L	--size	Указать, какой объем пространства на диске должен быть выделен для нового логического тома
-	--minor	Указать младший номер устройства
-m	--mirrors	Создать логический том с зеркальным отображением
-M	--persistent	Преобразовать младший номер в постоянный
-n	--name	Указать имя нового логического тома
-p	--permission	Задать разрешения на чтение и на запись для логического тома
-r	--readahead	Задать количество секторов упреждающего чтения
-R	--regionsize	Указать размер, применяемый для разделения областей зеркального отображения

Параметр	Длинное имя параметра	Описание
-s	--snapshot	Создать логический том моментального снимка
-Z	--zero	Задать нулевые значения для первого участка данных нового логического тома длиной 1 КБайт

Безусловно, на первый взгляд эти параметры командной строки могут показаться устрашающими по своей сложности, но в большинстве ситуаций можно обойтись минимальным количеством параметров:

```
$ sudo lvcreate -l 100%FREE -n lvtest Vol1
Logical volume "lvtest" created
$
```

Для ознакомления с подробными сведениями о созданном логическом томе можно воспользоваться командой `lvdisplay`:

```
$ sudo lvdisplay
--- Logical volume ---
LV Name                /dev/Vol1/lvtest
VG Name                Vol1
LV UUID                usDxti-pAEj-fEIZ-3kWV-LNAu-PFNx-2LGqNv
LV Write Access        read/write
LV Status              available
# open                 0
LV Size                2.00 GB
Current LE             511
Segments              1
Allocation             inherit
Read ahead sectors     auto
- currently set to    256
Block device           253:2
$
```

Вот теперь полученные результаты становятся более наглядными! Заслуживает внимания то, что имя группы томов (`Vol1`), которое использовалось для обозначения группы томов, теперь послужило в качестве имени при создании нового логического тома.

Параметр `-l` определяет, какая часть доступного пространства в указанной группе томов должна применяться для логического тома. Следует учитывать, что это значение может быть указано в процентах от объема свободного пространства в группе томов. В данном примере для нового логического тома было задействовано все свободное пространство (100%).

Параметр `-l` можно использовать для задания объема как процентной доли от доступного пространства, а параметр `-L` — для указания реального размера в байтах, килобайтах (KB), мегабайтах (MB) или гигабайтах (GB). Параметр `-n` позволяет указать имя для логического тома (в данном примере ему присвоено имя `lvtest`).

Создание файловой системы

После выполнения команды `lvcreate` логический том становится существующим, но еще не имеет файловой системы. Для создания файловой системы необходимо воспользоваться

программой командной строки, предназначенной для формирования соответствующей файловой системы:

```
$ sudo mkfs.ext4 /dev/Vol1/lvtest
mke2fs 1.41.9 (22-Aug-2009)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
130816 inodes, 523264 blocks
26163 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=536870912
16 block groups
32768 blocks per group, 32768 fragments per group
8176 inodes per group
Superblock backups stored on blocks:
    32768, 98304, 163840, 229376, 294912

Writing inode tables: done
Creating journal (8192 blocks): done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 21 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
$
```

После создания новой файловой системы можно смонтировать полученный том в виртуальном каталоге с применением стандартной команды `mount` системы Linux, полностью аналогично тому, как происходит монтирование физического раздела. Единственное различие состоит в том, что при этом используется специальное обозначение пути, в котором идентифицируется логический том:

```
$ sudo mount /dev/Vol1/lvtest test
$ cd test
$ ls -al
total 24
drwxr-xr-x.  3 root root  4096 2010-09-17 17:36 .
drwx-----. 33 rich rich  4096 2010-09-17 17:37 ..
drwx-----.  2 root root 16384 2010-09-17 17:36 lost+found
$
```

Заслуживает внимания то, что обозначения путей, используемые в обеих командах, `mkfs.ext4` и `mount`, немного необычны. Вместо пути к физическому разделу в этих обозначениях путей используются имя группы томов, а также имя логического тома. После монтирования этой файловой системы может осуществляться доступ к новой области в виртуальном каталоге.

Внесение изменений в систему LVM

Как уже было сказано, преимуществом применения Linux LVM является возможность динамически вносить изменения в файловые системы, поэтому следует полагать, что есть такие инструменты, которые позволяют выполнять именно это. В системе Linux действительно

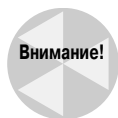
предусмотрены некоторые инструменты, с помощью которых можно изменять существующую конфигурацию управления логическими томами.

Безусловно, не во всех версиях Linux предусмотрены возможности использовать удобный графический интерфейс для управления средой Linux LVM, но даже в этом случае не все потеряно. В настоящей главе уже были описаны некоторые программы командной строки для работы с Linux LVM. Кроме них есть много других программ командной строки, которые могут применяться в целях управления средой LVM после ее установки. Широко применяемые команды, доступные в пакете Linux LVM, перечислены в табл. 7.6.

Таблица 7.6. Команды Linux LVM

Команда	Функция
<code>vgchange</code>	Активизировать или деактивизировать группу томов
<code>vgremove</code>	Удалить группу томов
<code>vgextend</code>	Добавить физические тома к группе томов
<code>vgreduce</code>	Удалить физические тома из группы томов
<code>lvextend</code>	Увеличить размер логического тома
<code>lvreduce</code>	Уменьшить размер логического тома

Применение этих программ командной строки позволяет получить полный контроль над средой Linux LVM.



Тем не менее, увеличивая или уменьшая размер логического тома вручную, необходимо соблюдать осторожность. Для переноса результатов изменения размера в файловую систему, хранимую в логическом томе, необходимо вручную внести исправления. Для большинства файловых систем предусмотрены программы командной строки, позволяющие переформатировать файловую систему. В качестве примера можно привести программу `resize2fs` для файловых систем `ext2` и `ext3`.

Резюме

Чтобы успешно работать с запоминающими устройствами в Linux, необходимо немного знать о файловых системах. Знания о том, как создавать файловые системы и работать с ними из командной строки, особенно востребованы при эксплуатации системы Linux. В настоящей главе описывалось, как осуществлять поддержку файловых систем из командной строки Linux.

Одним из отличий системы Linux от Windows является то, что в ней поддерживается гораздо больше различных методов хранения файлов и папок. Для каждого способа организации файловой системы предусмотрены различные средства, поэтому те или иные файловые системы становятся наиболее подходящими в разных ситуациях. Кроме того, в файловых системах различных типов используются разные команды для работы с запоминающими устройствами.

Прежде чем приступить к установке файловой системы на запоминающем устройстве, необходимо подготовить это устройство. Для выделения разделов на запоминающих устройствах в целях подготовки к развертыванию файловой системы применяется команда `fdisk`. Еще на этапе выделения разделов на запоминающем устройстве необходимо определить, какая файловая система будет на нем использоваться.

После создания раздела на запоминающем устройстве в нем можно установить одну из нескольких применимых файловых систем. Наиболее широко применяемыми файловыми системами в Linux являются `ext3` и `ext4`. Обе эти файловые системы предоставляют средства

файловой системы с ведением журнала, что способствует уменьшению количества ошибок и проблем, обычно возникающих при аварийном завершении работы системы Linux.

При создании файловых систем непосредственно в разделе запоминающего устройства приходится сталкиваться с одним ограничением. Оно обусловлено тем, что нет возможности легко изменить размер файловой системы после исчерпания свободного места на диске. Тем не менее Linux поддерживает средства управления логическими томами, которые обеспечивают создание виртуальных разделов, охватывающих несколько запоминающих устройств. Благодаря этому появляется возможность легко увеличивать объем пространства в существующей файловой системе, не перестраивая ее полностью. Пакет Linux LVM предоставляет команды командной строки для создания логических томов, распределяемых по нескольким запоминающим устройствам в целях формирования на них файловых систем.

В этой главе и предшествующих ей главах были подробно описаны основные команды командной строки Linux, и теперь мы можем приступить к написанию программ сценариев командного интерпретатора. Однако, прежде чем приступить к написанию кода, необходимо обсудить еще один компонент, применяемый в этой работе, — редакторы. Готовясь к созданию сценариев командного интерпретатора, необходимо освоить среду, в которой станут рождаться будущие шедевры. В следующей главе речь пойдет о том, как устанавливать пакеты программ из командной строки в различных средах Linux и управлять ими.

Установка программного обеспечения

В старых версиях Linux при решении задач установки программного обеспечения приходилось сталкиваться с определенными сложностями. К счастью, со временем разработчики Linux намного упростили эту работу благодаря включению комплектов программного обеспечения в предварительно собранные пакеты, установка которых осуществляется гораздо проще. Тем не менее для установки пакетов программ пользователю все равно приходится выполнять определенные действия, особенно если эта задача должна быть выполнена из командной строки. В настоящей главе рассматриваются различные системы управления пакетами, предусмотренные в Linux, и утилиты командной строки, предназначенные для установки, управления и удаления программного обеспечения.

Общие сведения об управлении пакетами

Данная глава посвящена описанию средств управления пакетами Linux, но вначале в ней излагаются некоторые основы управления пакетами. В каждом из ведущих дистрибутивов Linux в той или иной форме применяются системы управления пакетами (Package Management System — PMS) для управления установкой программных приложений и библиотек. В PMS используется база данных, которая позволяет контролировать следующее:

ГЛАВА

8

В этой главе...

Общие сведения об
управлении пакетами

Системы на основе Debian

Системы на основе Red Hat

Установка из исходного кода

Резюме

- какие пакеты программ установлены в системе Linux;
- какие файлы установлены для каждого пакета;
- версии каждого из установленных пакетов программ.

Пакеты программ хранятся на серверах, называемых *репозиториями*, а доступ к ним предоставляется по Интернету с помощью программ PMS, эксплуатируемых в локальной системе Linux. Программы PMS можно использовать для поиска новых пакетов программ и даже обновлений к пакетам программ, уже установленным в системе.

Пакет программ часто имеет так называемые *зависимости*, под которыми подразумеваются другие пакеты, которые должны быть установлены в первую очередь, для того чтобы рассматриваемое программное обеспечение могло функционировать должным образом. Программы PMS обнаруживают эти зависимости и предлагают установить все необходимые дополнительные пакеты программ перед установкой пакета, который потребовался пользователю.

Недостатком систем PMS является то, что для них не создана отдельная стандартная программа. Например, все команды командного интерпретатора `bash`, которые рассматривались до сих пор в настоящей книге, работают примерно одинаково, независимо от используемого дистрибутива Linux, а в отношении средств управления пакетами программ дела обстоят иначе.

Между программами PMS и связанными с ними командами обнаруживаются существенные различия в разных дистрибутивах Linux. Двумя основными программами PMS, которые обычно применяются в мире Linux и считаются базовыми, являются `dpkg` и `rpm`.

В дистрибутивах на основе Debian, таких как Ubuntu и Linux Mint, в качестве базовой для программ PMS используется команда `dpkg`. Она взаимодействует непосредственно с PMS в системе Linux и используется для установки, управления и удаления пакетов программ.

В дистрибутивах на основе Red Hat, в частности, таких, как Fedora, openSUSE и Mandriva, в качестве базовой для PMS используется команда `rpm`. Аналогично команде `dpkg`, команда `rpm` позволяет получать список установленных пакетов, устанавливать новые пакеты и удалять существующее программное обеспечение.

Следует учитывать то, что эти две команды представляют собой ядро соответствующих систем PMS, а не сами PMS. Для многих дистрибутивов Linux, в которых используются средства команды `dpkg` или `rpm`, подготовлены дополнительные специализированные программы PMS, которые позволяют намного упростить работу с этими основными командами. В следующих разделах описаны различные вспомогательные команды PMS, с которыми приходится сталкиваться, работая с большинством широко применяемых дистрибутивов Linux.

Системы на основе Debian

Ядром семейства инструментов PMS для дистрибутивов на основе Debian является команда `dpkg`. К другим инструментам, включаемым в эту систему PMS, относятся следующие:

- `apt-get`;
- `apt-cache`;
- `aptitude`.

Безусловно, наибольшее применение среди этих утилит командной строки находит `aptitude`, и не без оснований. Инструмент `aptitude` по существу представляет собой сервер, обслуживающий клиентские запросы и для инструментов `apt`, и для команды `dpkg`. Следует учитывать, что `dpkg` — это инструмент PMS, тогда как `aptitude` — полная система управления пакетами.

Применение команды `aptitude` в командной строке позволяет избежать проблем, часто возникающих при установке программного обеспечения, таких как пропущенные программные зависимости, нестабильная системная среда, а также многих ненужных затруднений. В настоящем разделе речь пойдет о том, как работать с инструментом `aptitude` из командной строки Linux.

Управление пакетами с помощью команды `aptitude`

Системные администраторы Linux часто сталкиваются с необходимостью определения того, какие пакеты уже установлены в системе. К счастью, команда `aptitude` имеет удобный интерактивный интерфейс, который позволяет упростить решение этой задачи.

В приглашении командного интерпретатора введите `aptitude` и нажмите клавишу <ВВОД>, чтобы перейти к полноэкранному режиму `aptitude`, как показано на рис. 8.1.

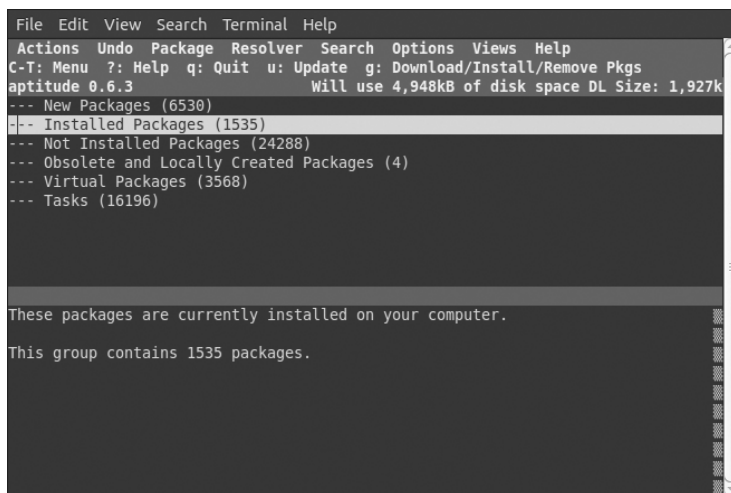


Рис. 8.1. Главное окно `aptitude`

Для перехода по командам меню в окне `aptitude` используются клавиши со стрелками. Чтобы ознакомиться с тем, какие пакеты установлены в системе, выберите элемент меню **Installed Packages** (Установленные пакеты). На дисплее появится список пакетов программ, распределенный по нескольким группам, таким как редакторы и т.д. За каждой группой следует число в круглых скобках, которое указывает количество пакетов, содержащихся в каждой группе.

Воспользуйтесь клавишами со стрелками, чтобы выделить одну из групп подсветкой, и нажмите клавишу <ВВОД> для просмотра подгрупп пакетов. После этого будет показан список отдельных пакетов с их именами и номерами версий. Нажимая клавишу <ВВОД> после перехода к отдельным пакетам, можно получить очень подробные дополнительные сведения, такие как описание пакета, начальная страница разработчика, размер, ответственные за разработку пакета и т.д.

Закончив просмотр установленных пакетов, нажмите клавишу <q>, чтобы завершить работу с экраном. Вслед за этим снова произойдет переход к окну, в котором можно использовать клавиши со стрелками и нажимать клавишу <ВВОД> для открытия или закрытия групп и под-

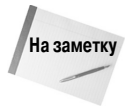
групп пакетов. По завершении просмотра пакетов достаточно только несколько раз нажать клавишу <q>, после чего появится всплывающее окно Really quit Aptitude? (Выйти из программы Aptitude?).

Если все пакеты, установленные в системе, хорошо известны и требуется лишь быстро получить подробные сведения о конкретном пакете, то нет необходимости открывать интерактивный интерфейс aptitude. Можно использовать aptitude как отдельную команду в командной строке:

```
aptitude show ИМЯ_пакета
```

Ниже приведен пример отображения подробных сведений о пакете grub2-theme-mint.

```
$ aptitude show grub2-theme-mint
Package: grub2-theme-mint
New: yes
State: installed
Automatically installed: no
Version: 1.0.3
Priority: optional
Section: misc
Maintainer: Clement Lefebvre <root@linuxmint.com>
Uncompressed Size: 442k
Description: Grub2 theme for Linux Mint
  Grub2 theme for Linux Mint
$
```



Команда `aptitude show` не указывает, установлен ли пакет в системе. Она лишь отображает подробные сведения о пакете, полученные из репозитория программного обеспечения.

Однако `aptitude` не позволяет получить определенный фрагмент необходимых сведений — листинг всех файлов, связанных с конкретным пакетом программ. Чтобы получить этот список, необходимо прибегнуть непосредственно к инструменту `dpkg`:

```
dpkg -L ИМЯ_пакета
```

Ниже приведен пример использования `dpkg` для получения списка всех файлов, установленных в пакете grub2-theme-mint.

```
$
$ dpkg -L grub2-theme-mint
/.
/boot
/boot/boot
/boot/boot/grub
/boot/boot/grub/linuxmint.png
/boot/grub
/boot/grub/linuxmint.png
/usr
/usr/share
/usr/share/doc
/usr/share/doc/grub2-theme-mint
/usr/share/doc/grub2-theme-mint/changelog.gz
```

```
/usr/share/doc/grub2-theme-mint/copyright
/etc
/etc/grub.d
/etc/grub.d/06_mint_theme
$
```

Можно также решить обратную задачу — найти пакет, к которому принадлежит конкретный файл:

```
dpkg --search absolute_file_name
```

Необходимо учитывать, что для решения этой задачи необходимо использовать абсолютную ссылку на файл:

```
$
$ dpkg --search /boot/grub/linuxmint.png
grub2-theme-mint: /boot/grub/linuxmint.png
$
```

В выводе данной команды показано, что файл `linuxmint.png` был установлен в составе пакета `grub2-theme-mint`.

Установка пакетов программ с помощью инструмента aptitude

Выше было показано, как получить сведения о пакете программ в конкретной системе, а в настоящем разделе описан процесс установки пакетов программ. Прежде всего необходимо определить имя пакета, подлежащего установке. Как найти конкретный пакет программ? Для этого следует использовать команду `aptitude` с параметром поиска:

```
aptitude search имя_пакета
```

Привлекательной стороной этого способа поиска является то, что он не требует вставки символов-заместителей вокруг параметра *имя_пакета*. В данной команде подразумевается использование символов-заместителей. Ниже приведен пример применения команды `aptitude` для поиска пакета программ `wine`.

```
$
$ aptitude search wine
p  gnome-wine-icon-theme      - red variation of the GNOME- ...
v  libkwineffectsl-api        -
p  libkwineffectsla           - library used by effects...
p  q4wine                     - Qt4 GUI for wine (W.I.N.E)
p  shiki-wine-theme           - red variation of the Shiki- ...
p  wine                       - Microsoft Windows Compatibility ...
p  wine-dev                   - Microsoft Windows Compatibility ...
p  wine-gecko                  - Microsoft Windows Compatibility ...
p  wine1.0                     - Microsoft Windows Compatibility ...
p  wine1.0-dev                 - Microsoft Windows Compatibility ...
p  wine1.0-gecko               - Microsoft Windows Compatibility ...
p  wine1.2                     - Microsoft Windows Compatibility ...
p  wine1.2-dbg                 - Microsoft Windows Compatibility ...
p  wine1.2-dev                 - Microsoft Windows Compatibility ...
```

```

p   wine1.2-gecko          - Microsoft Windows Compatibility ...
p   winefish               - LaTeX Editor based on Bluefish
$

```

Заслуживает внимания то, что в этом листинге перед каждым именем пакета приведен символ `p` или `i`. Если в списке присутствуют символы `i` и `u`, это означает, что данный пакет в настоящее время установлен в конкретной системе. Если показаны символы `p` и `v`, это означает, что пакет доступен, но не установлен. На основании приведенных выше результатов можно сделать вывод, что в данной системе пакет `wine` в настоящее время не установлен, но может быть получен из репозитория программного обеспечения.

Как показано ниже, задача установки пакета программ в системе путем получения его из репозитория с помощью команды `aptitude` решается очень просто.

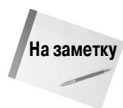
```
aptitude install ИМЯ_пакета
```

Как только имя пакета программ будет установлено с применением вызова команды, обеспечивающего поиск, достаточно указать это имя в следующем вызове команды `aptitude` с параметром установки:

```

$
$ sudo aptitude install wine
The following NEW packages will be installed:
  cabextract{a} esound-clients{a} esound-common{a}
  ↵ gnome-exe-thumbnailer
  {a}
  icoutils{a} imagemagick{a} libaudio2{a} libaudiofile0{a}
  ↵ libcddt4{a}
  libesd0{a} libgraph4{a} libgvc5{a} libilmbase6{a}
  ↵ libmagickcore3-extra
  {a}
  libmpg123-0{a} libnetpbm10{a} libopenall{a} libopenexr6{a}
  libpathplan4{a} libxdot4{a} netpbm{a} ttf-mscorefonts-installer{a}
  ttf-symbol-replacement{a} winbind{a} wine wine1.2{a} wine1.2-gecko{a}
0 packages upgraded, 27 newly installed, 0 to remove and
  ↵ 0 not upgraded.
Need to get 0B/27.6MB of archives. After unpacking 121MB will be used.
Do you want to continue? [Y/n/?] Y
Preconfiguring packages ...
...
All done, no errors.
All fonts downloaded and installed.
Updating fontconfig cache for /usr/share/fonts/truetype/msttcorefonts
Setting up winbind (2:3.5.4dfsg-1ubuntu7) ...
  * Starting the Winbind daemon winbind
  [ OK ]
Setting up wine (1.2-0ubuntu5) ...
Setting up gnome-exe-thumbnailer (0.6-0ubuntu1) ...
Processing triggers for libc-bin ...
ldconfig deferred processing now taking place
$

```



В приведенном выше примере перед вызовом команды `aptitude` была вызвана команда `sudo`, которая позволяет выполнять другие команды от имени пользователя `root`. Поэтому команду `sudo` можно использовать для решения административных задач, таких как установка программного обеспечения.

Чтобы проверить, установлена ли программа должным образом, можно снова воспользоваться параметром `search`. На сей раз перед именем пакета программ `wine` должно присутствовать обозначение `i u`, которое указывает, что этот пакет установлен.

Заслуживает также внимания то, что перед этим пакетом появились дополнительные пакеты с таким же обозначением `i u`. Появление этих пакетов обусловлено тем, что команда `aptitude` автоматически разрешает все необходимые зависимости пакета от имени пользователя и устанавливает связанные с этим дополнительные пакеты библиотек и программ. Это замечательное средство включено во многие системы управления пакетами.

Обновление программного обеспечения с помощью команды `aptitude`

Таким образом, команда `aptitude` позволяет избежать многих проблем при установке программного обеспечения, но задача координации обновления сразу нескольких пакетов с зависимостями остается сложной. Для безопасного обновления всех пакетов программ в системе с переходом на новые их версии, появившиеся в репозитории, следует использовать параметр `safe-upgrade`:

```
aptitude safe-upgrade
```

Необходимо учитывать, что для этой команды не требуется указывать в качестве параметра какое-либо имя пакета программ. Причина этого состоит в том, что сам параметр `safe-upgrade` обновляет все установленные пакеты с применением наиболее современных версий, имеющихся в репозитории, а это позволяет добиться намного более стабильной работы системы.

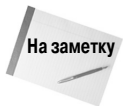
Ниже приведен пример вывода результатов выполнения команды `aptitude safe-update`.

```
$
$ sudo aptitude safe-upgrade
The following packages will be upgraded:
  evolution evolution-common evolution-plugins gsfonts libevolution
  xserver-xorg-video-geode
6 packages upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
Need to get 9,312kB of archives. After unpacking 0B will be used.
Do you want to continue? [Y/n/?] Y
Get:1 http://us.archive.ubuntu.com/ubuntu/ maverick/main
  libevolution i386 2.30.3-1ubuntu4 [2,096kB]
...
Preparing to replace xserver-xorg-video-geode 2.11.9-2
(using .../xserver-xorg-video-geode_2.11.9-3_i386.deb) ...
Unpacking replacement xserver-xorg-video-geode ...
Processing triggers for man-db ...
Processing triggers for desktop-file-utils ...
Processing triggers for python-gmenu ...
...
Current status: 0 updates [-6].
$
```

Для обновления программ можно также использовать следующие параметры, менее ограничительные по сравнению с безопасным обновлением:

- `aptitude full-upgrade;`
- `aptitude dist-upgrade.`

Эти параметры выполняют ту же задачу, обновляя все пакеты программ с переходом к новейшим версиям. Отличие этих параметров от `safe-upgrade` заключается в том, что они не проверяют зависимости между пакетами. Однако при этом снова приходится сталкиваться с решением проблемы учета зависимостей пакетов, которая может оказаться очень трудоемкой. Если в действительности зависимости для различных пакетов точно не известны, то следует неизменно пользоваться параметром `safe-upgrade`.



Разумеется, вызов на выполнение команды `aptitude` с параметром `safe-upgrade` необходимо осуществлять регулярно для постоянной поддержки системы в актуальном состоянии. Но особенно важно учитывать эту рекомендацию после новой установки дистрибутива. Как правило, после последнего полного выпуска дистрибутива и до текущего момента накапливается большое количество исправлений для средств безопасности и обновлений.

Удаление программного обеспечения с помощью команды `aptitude`

Избавляться от ненужных пакетов программ с помощью `aptitude` так же легко, как и устанавливать или обновлять пакеты. При этом единственным важным решением, которое должно быть принято администратором, является то, следует ли оставлять или уничтожать файлы данных и конфигурации, которые относятся к удаляемому программному обеспечению.

Чтобы удалить пакет программ, но не файлы данных и конфигурации, следует воспользоваться параметром `remove` команды `aptitude`. А для удаления пакета программ вместе с относящимися к нему файлами данных и конфигурации применяется параметр `purge`:

```
$ sudo aptitude purge wine
[sudo] password for user:
The following packages will be REMOVED:
  cabextract{u} esound-clients{u} esound-common{u}
↳ gnome-exe-thumbnailer
{u}
  icoutils{u} imagemagick{u} libaudio2{u} libaudiofile0{u} libcdt4{u}
  libesd0{u} libgraph4{u} libgvc5{u} libilmbase6{u}
↳ libmagickcore3-extra
{u}
  libmpeg123-0{u} libnetpbm10{u} libopenall{u} libopenexr6{u}
  libpathplan4{u} libxdot4{u} netpbm{u} ttf-mscorefonts-installer{u}
  ttf-symbol-replacement{u} winbind{u} wine{p} wine1.2{u} wine1.2-gecko
{u}
0 packages upgraded, 0 newly installed, 27 to remove and 6 not
↳ upgraded.
Need to get 0B of archives. After unpacking 121MB will be freed.
Do you want to continue? [Y/n/?] Y
(Reading database ... 120968 files and directories currently
↳ installed.)
```



```

Removing ttf-mscorefonts-installer ...
...
Processing triggers for fontconfig ...
Processing triggers for ureadahead ...
Processing triggers for python-support ...

$

```

Чтобы узнать, успешно ли удален пакет, можно снова воспользоваться параметром `aptitude search`. Если перед именем пакета стоит символ `c`, это означает, что программное обеспечение удалено, но файлы конфигурации не были уничтожены в этой системе. Предшествующий символ `p` говорит о том, что файлы конфигурации были также удалены.

Репозитории aptitude

Применяемые по умолчанию местоположения репозитариев программного обеспечения для `aptitude` определяются при установке конкретного дистрибутива Linux. Сведения об этих местоположениях репозитариев сохраняются в файле `/etc/apt/sources.list`.

Чаше всего не приходится сталкиваться с необходимостью добавлять или удалять тот или иной репозиторий программного обеспечения, поэтому пользователь не должен вносить изменения в этот файл. Но следует учитывать, что команда `aptitude` осуществляет выборку программного обеспечения только из этих репозитариев. Кроме того, команда `aptitude` проверяет лишь эти репозитории при поиске программного обеспечения для установки или обновления. Если возникает необходимость включить какие-то дополнительные репозитории программного обеспечения для PMS, то следует воспользоваться указанным выше файлом.



Разработчики дистрибутива Linux прилагают большие усилия к тому, чтобы полностью исключить возможность возникновения конфликтов между версиями пакетов, добавляемыми к репозиториям. Поэтому обычно наиболее безопасно проводить установку или обновление пакетов программ из репозитория, предлагаемого разработчиками дистрибутива. Даже если стало известно о появлении более новой версии в каком-то другом месте, все равно следует отложить переход на нее до появления этой версии в репозитории применяемого дистрибутива Linux.

Ниже приведен пример файла `sources.list` из системы Ubuntu.

```

$
$ cat /etc/apt/sources.list
#deb cdrom:[Ubuntu 10.10 _Maverick Meerkat_ - Alpha i386
(20100921.1)]/ maverick main restricted
# See http://help.ubuntu.com/community/UpgradeNotes for how to upgrade to
# newer versions of the distribution.

deb http://us.archive.ubuntu.com/ubuntu/ maverick main restricted
deb-src http://us.archive.ubuntu.com/ubuntu/ maverick main restricted

## Major bug fix updates produced after the final release of the
## distribution.
deb http://us.archive.ubuntu.com/ubuntu/ maverick-updates main
restricted
deb-src http://us.archive.ubuntu.com/ubuntu/ maverick-updates main
restricted

```

```

...
## This software is not part of Ubuntu, but is offered by third-party
## developers who want to ship their latest software.
deb http://extras.ubuntu.com/ubuntu maverick main
deb-src http://extras.ubuntu.com/ubuntu maverick main

deb http://security.ubuntu.com/ubuntu maverick-security main restricted
deb-src http://security.ubuntu.com/ubuntu maverick-security main
restricted
deb http://security.ubuntu.com/ubuntu maverick-security universe
deb-src http://security.ubuntu.com/ubuntu maverick-security universe
deb http://security.ubuntu.com/ubuntu maverick-security multiverse
deb-src http://security.ubuntu.com/ubuntu maverick-security multiverse
$

```

Прежде всего заслуживает внимания в этом файле наличие большого количества полезных комментариев и предупреждений. Для указания источников репозитория используется следующая структура:

```
deb (или deb-src) address distribution_name package_type_list
```

Значение `deb` или `deb-src` указывает тип пакета программ. Значение `deb` говорит о том, что источник содержит откомпилированные программы, а значение `deb-src` указывает, что в этом источнике находится исходный код.

Запись *address* указывает веб-адрес репозитория программного обеспечения. Запись *distribution_name* содержит имя версии дистрибутива данного конкретного репозитория программного обеспечения. В рассматриваемом примере дистрибутив имеет имя `maverick`. Из этого не обязательно следует, что эксплуатируемый дистрибутив представляет собой `Maverick Meercat` версии `Ubuntu`; это лишь означает, что в данном дистрибутиве `Linux` используется репозиторий программного обеспечения `Maverick Meercat` для `Ubuntu`! С другой стороны, в файле `sources.list` дистрибутива `Linux Mint` можно найти сочетание различных репозиториях программного обеспечения `Ubuntu` и `Linux Mint`.

Наконец, запись *package_type_list* может состоять больше чем из одного слова и указывать, какого типа пакеты имеются в этом репозитории. Например, часто встречаются такие значения, как `main` (основной), `restricted` (ограниченный), `universe` (общеприменимый) или `partner` (партнерский).

Если вдруг возникнет необходимость добавить какой-то репозиторий программного обеспечения к файлу применяемых источников, можно попытаться сделать это самостоятельно, но весьма велика вероятность того, что это приведет к возникновению проблем. Тем не менее на сайтах репозиториях программного обеспечения и на различных сайтах разработчиков пакетов приведена тщательно проверенная строка текста, которую можно скопировать с соответствующего веб-сайта и вставить в свой файл `sources.list`. При этом лучше всего не набирать эту строку вручную, а использовать операции копирования и вставки.

Инструмент `aptitude` как внешний интерфейс предоставляет глубоко продуманные параметры командной строки для работы программы `dpkg` в дистрибутивах на основе `Debian`. Теперь перейдем к рассмотрению программы `rpm`, применяемой в дистрибутивах на основе `Red Hat`, и рассмотрим различные внешние интерфейсы этой программы.

Системы на основе Red Hat

Как и для дистрибутивов на основе Debian, для систем на основе Red Hat предусмотрено несколько различных инструментов с пользовательским интерфейсом. Наиболее широко применяемые инструменты перечислены ниже.

- `yum`. Используется в Red Hat и Fedora.
- `urpm`. Используется в Mandriva.
- `zypper`. Используется в openSUSE.

Все эти инструменты с пользовательским интерфейсом основаны на программе командной строки `rpm`. В следующем разделе описано, как управлять пакетами программ с использованием различных инструментов на основе `rpm`. Основное внимание будет уделено программе `yum`, но будут также приведены определенные сведения о программах `zypper` и `urpm`.

Получение списка установленных пакетов

Чтобы узнать, какие пакеты в настоящее время установлены в конкретной системе, введите в приглашении командного интерпретатора следующую команду:

```
yum list installed
```

Скорее всего, эта информация за одно мгновение пролетит перед вашими глазами на экране дисплея, поэтому лучше всего перенаправить результаты формирования листинга программного обеспечения в файл. После этого можно воспользоваться командой `more` или `less` (или редактором с пользовательским графическим интерфейсом), чтобы не спеша просмотреть полученный список:

```
yum list installed > installed_software
```

Для получения списка установленных пакетов в дистрибутиве openSUSE или Mandriva можно воспользоваться командами, приведенными в табл. 8.1. К сожалению, инструмент `urpm`, применяемый в Mandriva, в настоящее время не позволяет получить листинг установленных программ. Поэтому приходится обращаться к инструменту `rpm`, лежащему в его основе.

Таблица 8.1. Получение списка установленного программного обеспечения с применением `zypper` и `urpm`

Дистрибутив	Инструмент, обслуживающий клиентские запросы	Команда
Mandriva	<code>urpm</code>	<code>rpm -qa > installed_software</code>
openSUSE	<code>zypper</code>	<code>zypper search -I > installed_software</code>

Инструмент `yum` как средство получения подробных сведений о конкретном пакете программ действительно великолепен. Он не только предоставляет очень подробные описания пакетов, но и с помощью еще одной простой команды позволяет узнать, установлен ли пакет:

```
# yum list xterm
Loaded plugins: langpacks, presto, refresh-packagekit
Adding en_US to language list
Available Packages
xterm.i686 261-2.fc14 fedora
#
```

```
# yum list installed xterm
Loaded plugins: refresh-packagekit
Error: No matching Packages to list
#
```

Команды, предназначенные для получения подробных сведений о пакете программ с помощью `urpm` и `zypper`, приведены в табл. 8.2. Можно также получить еще более детализированный набор сведений о пакете из репозитория с помощью параметра `info` команды `zypper`.

Таблица 8.2. Способы просмотра различных сведений о пакете с помощью `zypper` и `urpm`

Тип расшифровки	Инструмент, обслуживающий клиентские запросы	Команда
Информация о пакете	<code>urpm</code>	<code>urpmq -i имя_пакета</code>
Установлен ли пакет	<code>urpm</code>	<code>rpm -q имя_пакета</code>
Информация о пакете	<code>zypper</code>	<code>zypper search -s имя_пакета</code>
Установлен ли пакет	<code>zypper</code>	Аналогичная команда, но обеспечивает поиск обозначения <code>i</code> в столбце <code>Status</code> (Состояние)

Наконец, чтобы узнать, к какому пакету программ относится конкретный файл в файловой системе, можно снова обратиться к универсальному инструменту `yum`! Достаточно лишь ввести следующую команду:

```
yum provides file_name
```

Ниже приведен пример осуществления попытки определить, к какому пакету программ относится файл конфигурации `/etc/yum.conf`.

```
#
# yum provides /etc/yum.conf
Loaded plugins: langpacks, presto, refresh-packagekit
Adding en_US to language list
yum-3.2.28-5.fc14.noarch : RPM installer/updater
Repo : fedora
Matched from:
Filename : /etc/yum.conf

yum-3.2.28-5.fc14.noarch : RPM installer/updater

Repo : installed
Matched from:
Other : Provides-match: /etc/yum.conf

#
```

В этом примере программой `yum` была проведена проверка двух отдельных репозитариев — `fedora` и `installed`. Из обоих репозитариев был получен ответ, который означает, что данный файл представлен в пакете программ `yum`!

Установка программного обеспечения с помощью yum

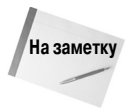
Задача установки пакетов программ при использовании инструмента yum становится предельно простой. Ниже приведена основная команда для установки из репозитория пакета программ, всех необходимых для него библиотек, а также зависимостей этого пакета.

```
yum install имя_пакета
```

Рассмотрим следующий пример установки пакета xterm:

```
$ su -
Password:
# yum install xterm
Loaded plugins: langpacks, presto, refresh-packagekit
Adding en_US to language list
fedora/metalink | 20 kB 00:00
fedora | 4.3 kB 00:00
fedora/primary_db | 11 MB 01:57
updates/metalink | 16 kB 00:00
updates | 4.7 kB 00:00
updates/primary_db | 3.1 MB 00:30
Setting up Install Process
Resolving Dependencies
--> Running transaction check
---> Package xterm.i686 0:261-2.fc14 set to be installed
...
Installed:
xterm.i686 0:261-2.fc14

Complete!
#
#
```



В приведенном выше листинге команде yum предшествует команда su, позволяющая переключиться на работу в учетной записи пользователя root. В этой системе Linux приглашение со знаком # указывает, что текущий пользователь зарегистрирован в качестве пользователя root. Как правило, следует переходить к учетной записи пользователя root только на короткое время, для выполнения таких административных задач, как установка и обновление программного обеспечения. Еще один способ перехода к использованию учетной записи пользователя root состоит в применении команды sudo.

Предусмотрена также возможность загрузить установочный файл rpm вручную и установить его с использованием yum. Такой способ установки называется *локальной установкой*. Основной применяемой для этого командой является следующая:

```
yum localinstall имя_пакета.rpm
```

Приведенное выше описание вполне позволяет убедиться в том, что одним из достоинств инструмента yum является то, что в нем используются очень логичные и понятные команды.

В табл. 8.3 показано, как выполнять установку пакетов с помощью urpmi и zypper. Следует учитывать, что пользователи, не зарегистрированные в учетной записи root, при вызове на выполнение инструмента urpmi получают сообщение об ошибке "command not found" (команда не найдена).

Таблица 8.3. Установка программного обеспечения с помощью zypper и urpm

Инструмент, обслуживающий клиентские запросы	Команда
urpm	urpmi имя_пакета
zypper	zypper install имя_пакета

Обновление программного обеспечения с помощью yum

В большинстве дистрибутивов Linux после достаточно продолжительной работы в пользовательском графическом интерфейсе появляются небольшие изящные значки уведомлений, которые указывают, что требуется обновление, после чего можно легко решить эту задачу. Что же касается интерфейса командной строки, то при его использовании приходится прилагать немного больше усилий.

Для просмотра списка всех имеющихся обновлений для установленных пакетов введите следующую команду:

```
yum list updates
```

При этом всегда бывает приятно отсутствие ответа на данную команду, поскольку это означает, что нет необходимости в обновлениях! Тем не менее, если будет обнаружено, что какой-то конкретный пакет программ нуждается в обновлении, введите следующую команду:

```
yum update имя_пакета
```

Если же потребуются обновить все пакеты, перечисленные в списке обновлений, достаточно лишь ввести следующую команду:

```
yum update
```

Команды, применяемые для обновления пакетов программ в дистрибутивах Mandriva и openSUSE, перечислены в табл. 8.4. Если используется инструмент urpm, то произойдет автоматическое пополнение базы данных репозитория, а также обновление пакетов программ.

Таблица 8.4. Обновление программного обеспечения с помощью zypper и urpm

Инструмент, обслуживающий клиентские запросы	Команда
urpm	urpmi --auto-update --update
zypper	zypper update

Удаление программного обеспечения с помощью yum

Инструмент yum предоставляет также простой способ удаления программного обеспечения, которое больше не требуется в конкретной системе. Как и в случае применения инструмента aptitude, необходимо принять решение о том, следует ли сохранять файлы данных и конфигурации, которые относятся к удаляемому пакету программ.

Чтобы удалить только пакет программ и оставить нетронутыми все файлы данных и конфигурации, можно воспользоваться следующей командой:

```
yum remove имя_пакета
```

Для удаления программного обеспечения и всех его файлов служит параметр `erase`:

```
yum erase имя_пакета
```

Как показано в табл. 8.5, задача удаления программного обеспечения с помощью `urpm` и `zypper` решается столь же просто. Оба эти инструмента выполняют функцию, аналогичную параметру `erase` программы `yum`.

Таблица 8.5. Удаление программного обеспечения с помощью `zypper` и `urpm`

Инструмент, обслуживающий клиентские запросы	Команда
<code>urpm</code>	<code>urpme имя_пакета</code>
<code>zypper</code>	<code>zypper remove имя_пакета</code>

Очевидно, что пакеты PMS намного упрощают задачу управления программным обеспечением, но и при их использовании могут возникать проблемы. Например, иногда при работе с ними обнаруживаются непредвиденные результаты. К счастью, для исправления ситуации можно сделать многое.

Возобновление нормальной работы при обнаружении нарушенных зависимостей

Иногда после загрузки нескольких пакетов программ обнаруживается, что программная зависимость для одного пакета переопределена в результате установки другого пакета. Такая ситуация известна под названием *нарушенной зависимости*.

Если подобная ситуация обнаруживается в системе, вначале необходимо выполнить команду

```
yum clean all
```

После этого следует воспользоваться параметром `update` команды `yum`. А иногда выходом из положения становится просто очистка всех файлов, которые находятся в неподходящих местоположениях.

Если это не позволяет решить проблему, попытайтесь выполнить следующую команду:

```
yum deplist имя_пакета
```

Эта команда показывает библиотечные зависимости для всех пакетов и позволяет узнать, какие пакеты программ предоставляют эти зависимости. После получения сведений о том, какие библиотеки требуются для пакета, можно просто их установить. Ниже приведен пример определения зависимостей для пакета `xterm`.

```
# yum deplist xterm
Loaded plugins: langpacks, presto, refresh-packagekit
Adding en_US to language list
Finding dependencies:
package: xterm.i686 261-2.fc14
dependency: libutempter.so.0
provider: libutempter.i686 1.1.5-4.fc12
dependency: rtld(GNU_HASH)
provider: glibc.i686 2.12.90-17
provider: glibc.i686 2.12.90-21
dependency: libc.so.6(GLIBC_2.4)
provider: glibc.i686 2.12.90-17
```

```

provider: glibc.i686 2.12.90-21
...
dependency: /bin/sh
provider: bash.i686 4.1.7-3.fc14
dependency: libICE.so.6
provider: libICE.i686 1.0.6-2.fc13
dependency: libXmu.so.6
provider: libXmu.i686 1.0.5-2.fc13
dependency: libc.so.6(GLIBC_2.3)
provider: glibc.i686 2.12.90-17
provider: glibc.i686 2.12.90-21
dependency: libXaw.so.7
provider: libXaw.i686 1.0.6-4.fc12
dependency: libX11.so.6
provider: libX11.i686 1.3.4-3.fc14
dependency: libc.so.6(GLIBC_2.2)
provider: glibc.i686 2.12.90-17
provider: glibc.i686 2.12.90-21
#

```

Если и это не позволяет устранить данную проблему, можно прибегнуть еще к одному, последнему средству:

```
yum update --skip-broken
```

Параметр `--skip-broken` позволяет просто пропустить пакет с нарушенной зависимостью и обновить остальные пакеты программ. Возможно, это не поможет устранить нарушения в работе пакета, но, по крайней мере, все остальные пакеты в системе будут успешно обновлены!

В табл. 8.6 перечислены команды, с помощью которых можно попытаться восстановить нарушенные зависимости, обнаруженные с помощью `urpm` и `zypper`. При использовании инструмента `zypper` может применяться лишь одна команда для проверки и исправления нарушенной зависимости. С другой стороны, если при использовании `urpm` параметр `clean` не позволяет достичь требуемых результатов, то можно, по крайней мере, пропустить пакет, при обновлении которого возникают нарушения в работе. Для этого необходимо добавить имя пакета, нарушающего нормальную работу, в файл `/etc/urpmi/skip.list`.

Таблица 8.6. Восстановление нарушенных зависимостей с помощью `zypper` и `urpm`

Инструмент с клиентским интерфейсом	Команда
<code>urpm</code>	<code>urpmi --clean</code>
<code>zypper</code>	<code>zypper verify</code>

Репозитории `yum`

Для инструмента `yum`, точно так же, как и для систем `aptitude`, предусмотрены репозитории программного обеспечения, применяемые при установке программ. В большинстве ситуаций эти заранее подготовленные репозитории позволяют полностью выполнить все требования, возникающие при установке программ. Причем, даже если в какой-то момент возникнет необходимость установить программное обеспечение из другого репозитория, следует предварительно учесть приведенные ниже соображения.



Предусмотрительный системный администратор всегда придерживается утвержденных репозитариев. *Утвержденными репозитариями* называются те репозитории, применение которых одобрено на официальном сайте разработчиков дистрибутива. Занявшись добавлением неутвержденных репозитариев, можно потерять гарантию стабильности. Дело в том, что при этом создаются предпосылки возникновения нарушенных зависимостей.

Для ознакомления с тем, какие репозитории в настоящее время применяются для выборки программного обеспечения, введите следующую команду:

```
yum repolist
```

Если же не удастся найти репозитарий, в котором находится требуемое программное обеспечение, то следует выполнить небольшое редактирование файла конфигурации. Файлы определения репозитариев `yum` находятся в каталоге `/etc/yum.repos.d`. Необходимо добавить в этот файл URL-адрес, соответствующий репозиторию, и получить доступ ко всем требуемым ключам шифрования.

На хорошо организованных сайтах репозитариев, таких как `rpmfusion.org` (<http://rpmfusion.org>), можно найти описание всех шагов, необходимых для использования этих сайтов. Иногда эти сайты репозитариев предлагают файл `rpm`, который можно загрузить и установить с помощью команды `yum localinstall`. В результате установки данного файла `rpm` вся работа по подготовке репозитария к использованию будет выполнена автоматически. Это очень удобно!

В пакете `urpm` для обозначения репозитариев применяется термин *носители информации* (*media*). В табл. 8.7 приведены команды, применяемые для просмотра носителей информации `urpm` и репозитариев `zypper`. Работая с этими двумя инструментами с внешним интерфейсом, не приходится редактировать файл конфигурации. Вместо этого, чтобы добавить носитель информации или репозитарий, достаточно ввести одну лишь команду.

Таблица 8.7. Репозитории `zypper` и `urpm`

Действие	Инструмент, обслуживающий клиентские запросы	Команда
Отображение репозитария	<code>urpm</code>	<code>urpmq --list-media</code>
Добавление репозитария	<code>urpm</code>	<code>urpmi.addmedia path_name</code>
Отображение репозитария	<code>zypper</code>	<code>zypper repos</code>
Добавление репозитария	<code>zypper</code>	<code>zypper addrepo path_name</code>

В системах обоих типов (основанных и на Debian, и на Red Hat) для упрощения процесса управления программным обеспечением используются системы управления пакетами. Но иногда приходится отказываться от удобств, связанных с применением систем управления пакетами, и решать несколько более трудную задачу установки программного обеспечения непосредственно из исходного кода. Речь об этом пойдет в следующем разделе.

Установка из исходного кода

В главе 4 рассматривались пакеты в виде *tar-файлов* и было описано, как создавать эти пакеты с использованием команды командной строки `tar` и как их распаковывать. До появления таких привлекательных инструментов, как `rpm` и `dpkg`, администраторам постоянно приходилось заниматься распаковкой и установкой программного обеспечения из *tar-файлов*.

Но и в наши дни программистам, использующим программную среду с открытым исходным кодом, иногда приходится сталкиваться с программным обеспечением, собранным в виде tar-файла. В настоящем разделе рассматривается процесс распаковки и установки пакетов программ из tar-файлов.

В качестве примера будет использоваться пакет программ sysstat. Программа sysstat представляет собой чрезвычайно интересный пакет программ, который предоставляет целый ряд инструментов контроля над системой.

Прежде всего необходимо загрузить tar-файл sysstat для конкретной системы Linux. Безусловно, можно легко найти готовый пакет sysstat на том или ином сайте Linux, но обычно лучше непосредственно перейти на сайт с исходным кодом этой программы. В рассматриваемом случае таковым является веб-сайт <http://sebastien.godard.pagesperso-orange.fr/>.

После щелчка на ссылке **Download (Загрузка)** происходит переход к странице, которая содержит файлы для загрузки. Ко времени написания этой книги текущая версия программы имела номер 9.1.5, а файл дистрибутива носил имя `sysstat-9.1.5.tar.gz`.

Щелкните на соответствующей ссылке, чтобы загрузить файл с исходным кодом в систему Linux. Сразу после загрузки файла его можно распаковать.

Для распаковки программного tar-файла используется стандартная команда `tar`:

```
#
# tar -zxvf sysstat-9.1.5.tar.gz
sysstat-9.1.5/
sysstat-9.1.5/sar.c
sysstat-9.1.5/iostat.c
sysstat-9.1.5/sadc.c
sysstat-9.1.5/sa.h
sysstat-9.1.5/iconfig
sysstat-9.1.5/CHANGES
sysstat-9.1.5/COPYING
sysstat-9.1.5/CREDITS
sysstat-9.1.5/sa2.in
sysstat-9.1.5/README
sysstat-9.1.5/crontab.sample
sysstat-9.1.5/nls/
...
sysstat-9.1.5/nfsiostat.c
sysstat-9.1.5/sysstat-9.1.5.lsm
sysstat-9.1.5/cifsioestat.c
sysstat-9.1.5/nfsiostat.h
sysstat-9.1.5/cifsioestat.h
sysstat-9.1.5/sysstat-9.1.5.spec
#
```

После того как tar-файл будет распакован, а содержащиеся в нем файлы должным образом размещены в каталоге `sysstat-9.1.5`, можно перейти в этот каталог и продолжить работу.

Сначала воспользуемся командой `cd` для перехода в новый каталог, а затем рассмотрим перечень его содержимого:

```
$ cd sysstat-9.1.5
$ ls
activity.c      INSTALL        prf_stats.h    sysconfig.in
build           ioconf.c       pr_stats.c     sysstat-9.1.5.lsm
```

```

CHANGES          ioconf.h      pr_stats.h      sysstat-9.1.5.spec
cifsiostat.c      iostat.c      rd_stats.c      sysstat.cron.daily.in
cifsiostat.h      iostat.h      rd_stats.h      sysstat.cron.d.in
common.c          Makefile.in   README         sysstat.cron.hourly.in
common.h          man           sa1.in         sysstat.in
configure         mpstat.c      sa2.in         sysstat.ioconf
configure.in      mpstat.h      sa_common.c    sysstat.sysconfig.in
contrib          nfsiostat.c   sadc.c         TODO
COPYING          nfsiostat.h   sadf.c         version.in
CREDITS          nls           sadf.h         xml
crontab.sample   pidstat.c     sa.h
FAQ              pidstat.h     sar.c
iconfig          prf_stats.c   sa_wrap.c
$

```

В листинге каталога, созданного подобным образом, обычно обнаруживается файл README или AAAREADME. С этим файлом следует обязательно ознакомиться. В нем приведены инструкции, относящиеся к текущей версии, которые необходимо выполнить, чтобы довести до конца установку программного обеспечения.

Согласно рекомендации, содержащейся в файле README для пакета sysstat, следующим шагом должно быть выполнение команды `configure sysstat` в конкретной системе. При этом выполняется проверка того, имеются ли в данной системе Linux все библиотеки, от которых зависит пакет, и есть ли необходимый компилятор для компиляции исходного кода:

```

# ./configure
Check programs:
.
checking for gcc... gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
...
checking for ANSI C header files... (cached) yes
checking for dirent.h that defines DIR... yes
checking for library containing opendir... none required
checking ctype.h usability... yes
checking ctype.h presence... yes
checking for ctype.h... yes
checking errno.h usability... yes
...
Check library functions:
.
checking for strchr... yes
checking for strcspn... yes
checking for strspn... yes
checking for strstr... yes
checking for sensors support... yes
...
Check configuration:
...
config.status: creating Makefile

```

```

Sysstat version:      9.1.5
Installation prefix:  /usr/local
rc directory:         /etc/rc.d
Init directory:       /etc/rc.d/init.d
Configuration directory: /etc/sysconfig
Man pages directory:  /usr/local/man
Compiler:             gcc
Compiler flags:       -g -O2

```

#

Если же какие-либо обязательные требования к установке не соблюдаются, на шаге применения команды `configure` будут получены сообщения об ошибках с объяснением того, что еще должно быть сделано.

На следующем этапе должны быть созданы все необходимые двоичные файлы с использованием команды `make`. Команда `make` обеспечивает компиляцию исходного кода, после чего редактор связей создает окончательные версии исполняемых файлов для пакета. Как и команда `configure`, команда `make` вырабатывает большой объем вывода по мере того, как осуществляются шаги компиляции и редактирования связей для всех файлов исходного кода:

```

# make
...
gcc -o nfsiostat -g -O2 -Wall -Wstrict-prototypes -pipe
-O2 nfsiostat.o librdstats.a libsyscom.a -s
gcc -o cifsioestat.o -c -g -O2 -Wall -Wstrict-prototypes -pipe
-O2 -DSA_DIR="/var/log/sa\"
-DSADC_PATH="/usr/local/lib/sa/sadc\" cifsioestat.c
gcc -o cifsioestat -g -O2 -Wall -Wstrict-prototypes -pipe
-O2 cifsioestat.o librdstats.a libsyscom.a -s
#

```

А после завершения работы команды `make` в каталоге появится практически применимая программа `sysstat`! Однако вызов программы на выполнение из того каталога, в котором осуществлена сборка ее исполняемого файла, связан с определенными неудобствами. Поэтому программу следует установить там, где обычно принято размещать программное обеспечение в системе Linux. Для этого необходимо зарегистрироваться в учетной записи пользователя `root` (или воспользоваться командой `sudo`, если этот вариант является предпочтительным в применении дистрибутиве Linux), а затем воспользоваться параметром `install` команды `make`:

```

# make install
mkdir -p /usr/local/man/man1
mkdir -p /usr/local/man/man8
rm -f /usr/local/man/man8/sa1.8*
install -m 644 -g man man/sa1.8 /usr/local/man/man8
rm -f /usr/local/man/man8/sa2.8*
install -m 644 -g man man/sa2.8 /usr/local/man/man8
rm -f /usr/local/man/man8/sadc.8*
install -m 644 -g man man/sadc.8 /usr/local/man/man8
rm -f /usr/local/man/man1/sar.1*
...
install -m 644 sysstat.sysconfig /etc/sysconfig/sysstat
install -m 644 CHANGES /usr/local/share/doc/sysstat-9.1.5

```

```
install -m 644 COPYING /usr/local/share/doc/sysstat-9.1.5
install -m 644 CREDITS /usr/local/share/doc/sysstat-9.1.5
install -m 644 README /usr/local/share/doc/sysstat-9.1.5
install -m 644 FAQ /usr/local/share/doc/sysstat-9.1.5
install -m 644 *.lsm /usr/local/share/doc/sysstat-9.1.5
#
```

Теперь пакет `sysstat` окончательно установлен в системе! Безусловно, описанные выше шаги немного затруднительнее по сравнению с установкой пакета программ с помощью PMS, но в целом установка с использованием программных `tar`-файлов не является такой уж сложной.

Резюме

В настоящей главе описано, как работать с системами управления пакетами программ (Package Management System — PMS) при установке, обновлении или удалении программного обеспечения из командной строки. Безусловно, в большинстве дистрибутивов Linux предусмотрены привлекательные инструменты с графическим пользовательским интерфейсом для управления пакетами программ, но не исключена возможность заниматься управлением пакетами из командной строки.

В дистрибутивах Linux на основе Debian для доступа к интерфейсу с системой PMS из командной строки используется программа `dpkg`. Внешним интерфейсом для программы `dpkg` служит программа `aptitude`. Программа `aptitude` предоставляет простые параметры командной строки для работы с пакетами программ в формате `dpkg`.

В дистрибутивах Linux на основе Red Hat применяется программа `rpm`, но для работы в командной строке служат другие инструменты пользовательского интерфейса. В дистрибутивах Red Hat и Fedora для установки пакетов программ и управления ими используется программа `yum`. В дистрибутиве openSUSE для управления программным обеспечением используется программа `zypper`, а в дистрибутиве Mandriva — программа `urpm`.

В завершении главы был описан процесс установки пакетов программ, которые распространяются только в виде `tar`-файлов с исходным кодом. Команда `tar` позволяет распаковать файлы исходного кода, содержащиеся в `tar`-файле, после чего с помощью команд `configure` и `make` можно окончательно собрать исполняемую программу из исходного кода.

В следующей главе рассматриваются различные редакторы, предусмотренные в дистрибутивах Linux. Прежде чем приступить к работе над сценариями командного интерпретатора, необходимо ознакомиться с редакторами, которые позволяют выполнять эту работу!

ГЛАВА

9

В этой главе...

Редактор vim

Редактор emacs

Семейство редакторов KDE

Редактор GNOME

Резюме

Работа с редакторами

Прежде чем приступать к освоению работы по написанию сценариев для командного интерпретатора, необходимо узнать, как использовать по крайней мере один текстовый редактор в Linux. Чем больше вам будет известно о нюансах применения всех привлекательных средств редакторов, таких как поиск, вырезка и вставка текста, тем в большей степени вы будете подготовлены к решению задач разработки сценариев командного интерпретатора. В настоящей главе рассматриваются основные текстовые редакторы, которые могут встретиться в мире Linux.

Редактор vim

При работе в режиме командной строки часто возникает необходимость применения средств хотя бы одного текстового редактора, который работает на консоли Linux. Одним из редакторов, который применялся в системах Unix с самых первых версий, является редактор vi. В нем используется графический режим консоли для эмуляции окна редактирования текста, что позволяет просматривать строки файла, переходить из одного места файла в другое, а также вставлять, редактировать и заменять текст.

Возможно, правы те, кто считает vi самым сложным редактором в мире (по крайней мере, такого мнения придерживаются люди, которые его буквально ненавидят), но этот редактор предоставляет так много средств, что заслуженно играет роль основного инструмента для администраторов Unix в течение многих десятилетий.

Разработчики проекта GNU, перенося редактор vi в мир программного обеспечения с открытым исходным кодом, решили внести в него некоторые усовершенствования.

В связи с этим в новой версии появились значительные отличия по сравнению с исходным редактором `vi`, применяющимся в мире Unix, поэтому разработчики переименовали свою версию в “`vi improved`”, или сокращенно `vim`.

Для упрощения работы почти во всех дистрибутивах Linux создаются псевдонимы (см. главу 5) для этого редактора, обозначаемые как `vi`, которые указывают на программу `vim`:

```
$ alias vi
alias vi='vim'
$
```

В настоящем разделе приведены основные сведения об использовании редактора `vim` для внесения изменений в текстовые файлы сценариев командного интерпретатора.

Основные сведения о редакторе `vim`

Редактор `vim` работает с данными в буфере, находящемся в памяти. Для запуска редактора `vim` достаточно ввести команду `vim` (или `vi`, если определен псевдоним) и имя файла, в который необходимо внести изменения:

```
$ vim myprog.c
```

Если запуск `vim` происходит без указания имени файла или если указанный файл не существует, то `vim` открывает новую буферную область для редактирования. Если же в командной строке указан существующий файл, то `vim` считывает все содержимое файла в буферную область и подготавливает файл для редактирования, как показано на рис. 9.1.

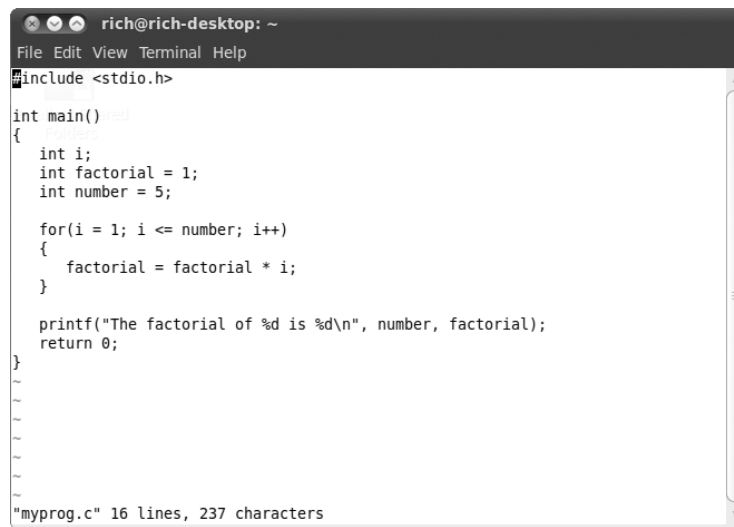


Рис. 9.1. Главное окно `vim`

Редактор `vim` распознает тип терминала, используемый для проведения сеанса (см. главу 2), и переходит в полноэкранный режим, что позволяет работать во всем окне терминала как в области редактирования.

В начальном окне редактирования `vim` отображается содержимое файла (если таковое имеется) наряду со строкой сообщения, расположенной вдоль нижнего края окна. Если содер-

жимое файла занимает не весь экран, редактор vim обозначает знаками тильды (~) те строки в окне, которые не относятся к файлу (что можно видеть на рис. 9.1).

В строке сообщения в нижней части окна приведены сведения о редактируемом файле, которые изменяются в зависимости от статуса файла, и показаны параметры конфигурации, применяемые по умолчанию в данной конкретной инсталляции vim. Если файл новый, появляется сообщение [New File].

Редактор vim имеет два режима работы:

- нормальный режим;
- режим вставки.

Сразу после открытия файла для редактирования (или после начала работы с новым файлом) редактор vim переходит в *нормальный* режим, в котором он интерпретирует коды введенных комбинаций клавиш как команды (более подробные сведения об этом приведены ниже).

А в режиме *вставки* редактор vim вставляет символ, соответствующий каждой нажатой клавише, в текущем местоположении курсора в буфере, показанном на экране. Для перехода в режим вставки следует нажать клавишу <i>. Чтобы выйти из режима вставки и возвратиться в нормальный режим, необходимо нажать клавишу <Esc> на клавиатуре.

В нормальном режиме можно перемещать курсор в текстовой области с использованием клавиш со стрелками (такая возможность предоставляется, только если применяемый тип терминала распознан редактором vim должным образом). Если же оказалось, что установленное терминальное соединение не поддерживает все необходимые функции и клавиши со стрелками не определены, то это не означает, что все потеряно. В число команд vim входят команды перемещения курсора, включая следующие:

- команда h, предназначенная для перемещения влево на один символ;
- команда j для перемещения вниз на одну строку (на следующую строку в тексте);
- команда k, позволяющая перейти вверх на одну строку (на предыдущую строку в тексте);
- команда l, обеспечивающая переход вправо на один символ.

Безусловно, построчное перемещение курсора в больших текстовых файлах является слишком непродуктивным. К счастью, в редакторе vim предусмотрено несколько команд, позволяющих ускорить переход по содержимому файла:

- команда PageDown (или <Ctrl+F>), которая указывает, что должен быть выполнен переход вперед на один экран с данными;
- команда PageUp (или <Ctrl+B>), предназначенная для перехода назад на один экран с данными;
- команда G для перемещения к последней строке в буфере;
- команда num G, позволяющая перейти к строке с номером num в буфере;
- команда gg, обеспечивающая переход к первой строке в буфере.

Кроме того, в нормальном режиме редактор vim предоставляет доступ к специальной функции, называемой *режимом командной строки*. В режиме командной строки пользователю предоставляется интерактивная командная строка, в которой он может вводить дополнительные команды для управления другими действиями в редакторе vim. Для перехода в режим командной строки следует нажать клавишу со знаком двоеточия в нормальном режиме. Курсор перемещается в строку сообщения, и появляется двоеточие как приглашение к вводу команды.

В число команд командной строки входят следующие несколько команд, предназначенных для сохранения буфера в файл и выхода из программы vim:

- команда `q`, позволяющая сразу же завершить работу, если не было внесено никаких изменений в данные в буфере;
- команда `q!`, с помощью которой можно завершить работу и отказаться от каких-либо изменений, внесенных в данные буфера;
- команда `w filename`, предназначенная для сохранения файла под другим именем файла;
- команда `wq`, обеспечивающая сохранение данных буфера в файле и завершение работы.

После ознакомления только с этими несколькими основными командами vim вполне можно понять, почему некоторые люди испытывают к редактору vim непримиримую ненависть. Дело в том, что для полноценного использования vim необходимо изучить большое количество непонятных, трудно запоминающихся команд. Однако, хорошо освоив даже небольшое количество основных команд vim, вы получите возможность быстро редактировать файлы непосредственно из командной строки, независимо от того, в какой среде работаете. Кроме того, после получения устойчивых навыков ввода команд с клавиатуры привычка набирать и данные, и команды редактирования становится второй натурой, поэтому программисты даже чувствуют себя немного не в своей тарелке, когда снова приходится прибегать к использованию мыши!

Редактирование данных

В процессе работы в режиме вставки можно вставлять данные в буфер, но иногда приходится также добавлять или удалять данные после того, как эти данные уже были введены в буфер. В нормальном режиме редактора vim предусмотрено несколько команд редактирования данных в буфере. В табл. 9.1 перечислены некоторые общие команды редактирования для vim.

Таблица 9.1. Команды редактирования vim

Команда	Описание
x	Удалить символ в текущей позиции курсора
dd	Удалить строку в текущей позиции курсора
dw	Удалить слово в текущей позиции курсора
d\$	Провести удаление до конца строки от текущей позиции курсора
J	Удалить символ обозначения конца строки, находящийся в конце строки в текущей позиции курсора (что приводит к соединению строк)
u	Отменить предыдущую команду редактирования
a	Добавить данные после текущей позиции курсора
A	Добавить данные до конца строки, начиная с текущей позиции курсора
r char	Заменить отдельный символ в текущей позиции курсора символом <i>char</i>
R text	Перезаписывать данные в текущей позиции курсора, заменяя текстом <i>text</i> , пока не будет нажата клавиша <Escape>

Некоторые команды редактирования позволяют также использовать числовой модификатор, который указывает кратность выполнения команды. Например, применение команды `2x` приводит к удалению двух символов, начиная с текущей позиции курсора, а команда `5dd` удаляет пять строк, начиная со строки, в которой в настоящее время находится курсор.



Прибегая к использованию клавиши <Backspace> или <Delete> клавиатуры персонального компьютера в редакторе vim, необходимо соблюдать осторожность. Редактор vim обычно распознает нажатие клавиши <Delete> как команду, функционально эквивалентную команде `x`, которая удаляет символ в текущем местоположении курсора. С другой стороны, обычно редактор vim не распознает нажатие клавиши <Backspace>.

Копирование и вставка

Стандартным средством современных редакторов является возможность вырезать или копировать данные, а затем вставлять их в другое место в документе. Редактор vim предоставляет способ выполнения этой операции.

Задача вырезки и вставки решается относительно просто. В табл. 9.1 уже были показаны команды, которые позволяют удалять данные из буфера. Однако редактор vim при удалении данных фактически не уничтожает эти данные, а передает на хранение в отдельный регистр. Затем удаленные данные можно получить с помощью команды `p`.

Например, можно удалить строку текста с помощью команды `dd`, затем переместить курсор в то место в буфере, в котором снова должна появиться эта строка, после чего применить команду `p`. Команда `p` вставляет текст после строки в текущей позиции курсора. Аналогичную операцию можно проделать с помощью любой команды, которая удаляет текст.

Задача копирования текста немного сложнее. Командой копирования в редакторе vim является `y` (сокращение от `yank` — копирование в память). С командой `y` можно использовать такой же второй символ, как и с командой `d` (например, вариант команды `yw` означает, что в память должно быть скопировано слово, а команда `y$` указывает, что в память должен быть скопирован текст до конца строки). После копирования текста в память можно перевести курсор к тому месту, где должен быть помещен текст, после чего задать команду `p`. Теперь скопированный в память текст появится в новом месте.

Сложность применения копирования в память заключается в том, что нельзя следить за происходящим, поскольку копирование текста не затрагивает сам текст, скопированный в память. Вы не знаете наверняка, что было скопировано в память, пока не вставите полученную копию где-нибудь в другом месте. Однако в редакторе vim предусмотрено еще одно средство, которое упрощает применение операции копирования в память.

Это — *визуальный режим*, предусматривающий выделение текста подсветкой при перемещении курсора. Визуальный режим можно использовать, выбирая текст, который должен быть скопирован в память для последующей вставки. Чтобы перейти в визуальный режим, переместите курсор в то место, где должно начаться копирование в память, и введите команду `v`. Вы заметите, что текст, начиная с текущей позиции курсора, теперь выделяется подсветкой. После этого перемещайте курсор, чтобы охватить текст, который должен быть скопирован в память (можно даже переходить на нижние строки, чтобы скопировать в память больше чем одну строку текста). По мере перемещения курсора редактор vim выделяет подсветкой текст в области копирования в память. Вслед за окончанием выделения подсветкой текста, который должен быть скопирован, нажмите клавишу <u> для вызова команды копирования в память. Теперь, когда текст отправлен в регистр, достаточно переместить курсор в то место, где он должен быть вставлен, и ввести команду `p`.

Поиск и замена

Команда поиска vim позволяет легко проводить поиск данных в буфере. Чтобы ввести строку для поиска, нажмите клавишу косой черты (</>). Курсор перейдет в строку сообщения,

и редактор vim выведет косую черту. Наберите текст, который должен быть найден, и нажмите клавишу <ВВОД>. Редактор vim в ответ выполнит одно из трех действий.

- Если искомое слово находится дальше в буфере по сравнению с текущим местоположением курсора, то происходит переход к первому ближайшему местоположению, где находится такой же текст.
- Если искомое слово не расположено вслед за текущим местоположением курсора, то после достижения конца файла происходит переход к первому местоположению в файле, где имеется текст (на что указывает также сообщение).
- Если поиск до конца файла или с начала файла оканчивается неудачей, то выводится сообщение об ошибке с указанием, что текст не найден в файле.

Чтобы продолжить поиск того же слова, нажмите клавишу косой черты, а затем — клавишу <ВВОД>; для продолжения поиска можно также ввести команду n (сокращение от next — далее).

Команда замены позволяет быстро заменять одно слово другим в тексте (подставлять вместо одного слова другое). Чтобы воспользоваться командой замены, необходимо перейти в режим командной строки. Команда замены имеет следующий формат:

```
:s/old/new/
```

Редактор vim переходит к первому вхождению текста old и заменяет его текстом new. Команда замены имеет несколько показанных ниже модификаций, которые позволяют заменять больше одного вхождения текста.

- Команда :s/old/new/g обеспечивает замену всех вхождений old в строке.
- Команда :n,ms/old/new/g указывает, что должны быть заменены все вхождения old в строках с номерами от n до m.
- Команда :%s/old/new/g позволяет заменить все вхождения old во всем файле.
- Команда :%s/old/new/gc требует заменить все вхождения old во всем файле, но выводить приглашения для подтверждения каждой замены.

Вполне очевидно, что vim, в отличие от других текстовых редакторов с командной строкой, содержит довольно много дополнительных функций. Редактор vim включен в каждый дистрибутив Linux, поэтому рекомендуется овладеть по меньшей мере основами работы с ним, чтобы всегда иметь возможность вносить изменения в сценарии, независимо от того, какие задачи вы решаете или какие еще редакторы имеете в своем распоряжении.

Редактор emacs

Редактор emacs — чрезвычайно популярный текстовый редактор, который был разработан еще до создания операционной системы Unix. Программисты в свое время настолько привыкли к этому редактору, что перенесли его в среду Unix, а теперь он перенесен также в среду Linux. Редактор emacs начал свое существование как редактор текста на терминале, во многом аналогично vi, но уже довольно давно был заменен графической версией.

Редактор emacs все еще обеспечивает работу в таком же режиме, как и его предшественник, предназначенный для терминала, но в настоящее время предоставляет также возможность использовать графическое окно X Window для редактирования текста в графической среде. Как правило, после запуска редактора emacs из командной строки программа редактора определяет, имеется ли доступный сеанс X Window, и в случае положительного ответа запускается

в графическом режиме. Если таковой сеанс не существует, редактор запускается в терминальном режиме.

В настоящем разделе приведено описание редакторов emacs обоих типов, и для терминального, и для графического режима, чтобы читатель мог узнать, как использовать тот и другой, если есть такое желание (или необходимость).

Работа с редактором emacs на терминале

Версию редактора emacs для терминального режима можно рассматривать как еще один редактор, в котором предусмотрено большое количество клавиатурных команд, позволяющих выполнять функции редактирования. В редакторе emacs используются комбинации клавиш, которые включают клавишу <Control> (клавиша <Ctrl> на клавиатуре персонального компьютера) и клавишу <Meta>. В большинстве пакетов эмуляции терминала персонального компьютера клавиша <Meta> отображается на клавишу <Alt> персонального компьютера. В официально утвержденной документации по emacs клавиша <Ctrl> сокращенно обозначается как <C->, а клавиша <Meta> — как <M->. Таким образом, для указания на то, что нужно ввести комбинацию клавиш <Ctrl-x>, в документах приведено сокращение <C-x>. В данной главе применяется тот же принцип, чтобы не возникала путаница.

Основы emacs

Чтобы приступить к редактированию файла с помощью emacs, введите в командной строке следующую команду:

```
$ emacs myprog.c
```

Откроется окно emacs для работы в терминальном режиме с кратким введением и экраном справки. Не следует беспокоиться по поводу того, что текст выбранного вами файла не появился на экране; как только вы нажмете любую клавишу, редактор emacs загрузит файл в активный буфер и отобразит текст, как показано на рис. 9.2.

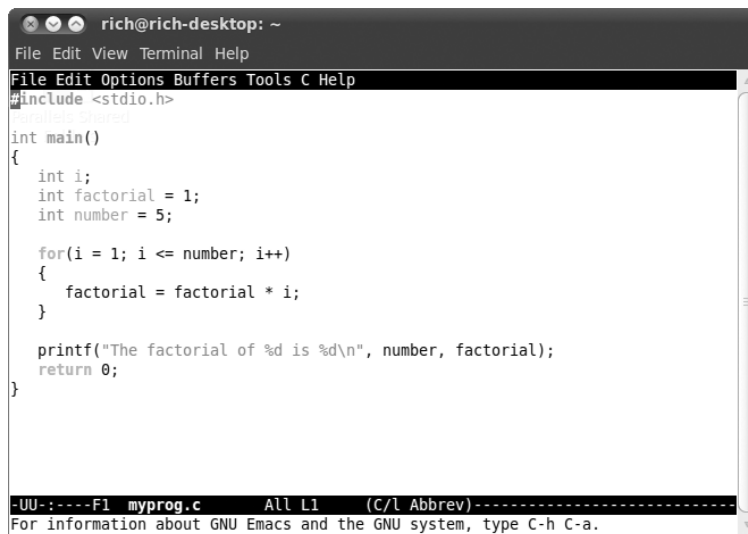
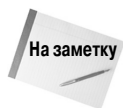


Рис. 9.2. Редактирование файла с использованием редактора emacs в терминальном режиме

Можно видеть, что в верхней части окна терминального режима отображается типичная строка меню. К сожалению, эта строка меню предназначена для использования не в терминальном режиме, а лишь в графическом.



Если у вас имеется графический рабочий стол, но вы предпочитаете использовать emacs в терминальном режиме, а не в режиме X Window, то дополнительно задайте параметр `-nw` в командной строке.

В отличие от редактора `vim`, в котором приходится входить и выходить из режима вставки для переключения между вводом команд и вставкой текста, в редакторе `emacs` имеется только один режим. После ввода любого символа, который может быть выведен на печать, редактор `emacs` вставляет его в текущей позиции курсора. А после ввода команды `emacs` выполняет эту команду.

Для перемещения курсора в области буфера можно использовать клавиши со стрелками, а также клавиши `<PageUp>` и `<PageDown>`. Эта возможность предоставляется лишь при том условии, что программа редактора `emacs` правильно распознает применяемый эмулятор терминала. В противном случае для перемещения курсора можно воспользоваться командами, приведенными ниже.

- Команда `C-p`, предназначенная для продвижения вверх на одну строку (предыдущую строку в тексте).
- Команда `C-b`, позволяющая перейти влево (назад) на один символ.
- Команда `C-f`, с помощью которой можно перейти вправо (вперед) на один символ.
- Команда `C-n`, обеспечивающая перемещение вниз на одну строку (следующую строку в тексте).

Предусмотрены также следующие команды, предназначенные для перемещения курсора в тексте на более значительные расстояния.

- Команда `M-f` позволяет перейти вправо (вперед) к следующему слову.
- Команда `M-b` перемещает курсор влево (назад) к предыдущему слову.
- Команда `C-a` перемещает курсор в начало текущей строки.
- Команда `C-e` позволяет перейти в конец текущей строки.
- Команда `M-a` обеспечивает перемещение в начало текущего предложения.
- Команда `M-e` перемещает курсор в конец текущего предложения.
- Команда `M-v` обеспечивает переход назад на один экран данных.
- Команда `C-v` позволяет перейти вперед на один экран данных.
- Команда `M-<` указывает, что должен быть выполнено перемещение на первую строку текста.
- Команда `M->` дает возможность перейти в последнюю строку текста.

Необходимо также освоить несколько следующих команд, с помощью которых можно снова сохранить буфер редактора в файле и выйти из программы `emacs`.

- Команда `C-x C-s` служит для сохранения текущего содержимого буфера в файл.
- Команда `C-z` позволяет выйти из редактора `emacs`, но оставить его работающим в текущем сеансе, чтобы можно было снова к нему вернуться.
- Команда `C-x C-c` указывает, что нужно выйти из `emacs` и остановить эту программу.

Заслуживает внимания то, что для двух из этих средств требуются по две клавиатурные команды. Команда C-x называется *расширенной командой*. Применение этой команды лежит в основе создания еще одного крупного набора команд, предназначенных для работы.

Редактирование данных

Редактор emacs довольно удачно справляется с операциями вставки и удаления текста в буфере. Чтобы вставить текст, переместите курсор в то место, где должен быть вставлен текст, и приступите к вводу текста. Для удаления текста в редакторе emacs предназначены клавиша <Backspace>, позволяющая удалить символ перед текущей позицией курсора, и клавиша <Delete>, которая удаляет символ в текущем местоположении курсора.

В редакторе emacs предусмотрены также команды уничтожения текста. Разница между удалением и уничтожением текста состоит в том, что текст после уничтожения помещается редактором emacs во временной области, из которой его можно извлечь (см. раздел “Копирование и вставка”). А удаленный текст исчезает навсегда.

Для уничтожения текста в буфере предусмотрено несколько команд.

- Команда M-Backspace позволяет уничтожить слово перед текущей позицией курсора.
- Команда M-d служит для уничтожения слова после текущей позиции курсора.
- Команда C-k применяется для уничтожения текста от текущей позиции курсора до конца строки.
- Команда M-k предназначена для уничтожения текста от текущей позиции курсора до конца предложения.

Кроме того, в редакторе emacs предусмотрен привлекательный способ уничтожения большого объема текста. Для этого достаточно переместить курсор в начало области, в которой должен быть уничтожен текст, и нажать клавиши <C-@> или <C-ПРОБЕЛ>. После этого необходимо переместить курсор в конец области, где находится уничтожаемый текст, и нажать командные клавиши <C-w>. Весь текст между этими двумя местоположениями будет уничтожен.

Если окажется так, что уничтожение текста произошло по ошибке, то можно воспользоваться командой C-u, которая отменяет действие команды уничтожения и возвращает данные в то состояние, в котором они находились до уничтожения.

Копирование и вставка

Выше было показано, как вырезать данные в области буфера emacs, а в данном разделе речь пойдет о том, как вставить эти данные в другом месте. К сожалению, у тех, кто привык работать с редактором vim, могут возникать недоразумения после перехода к выполнению этого процесса с использованием редактора emacs.

Этому способствует такое неудачное совпадение, что в редакторе emacs *копированием в память* (yanking) называется вставка данных. А в редакторе vim термином *yanking* обозначается непосредственно копирование в память, и об этом приходится постоянно помнить, если в ходе работы используются оба редактора.

После уничтожения данных с применением одной из команд уничтожения переместите курсор в то местоположение, где должны быть вставлены данные, и задайте команду C-y. Это приводит к копированию в память текста из временной области и вставке его в текущей позиции курсора. Команда C-y копирует в память текст, сохраненный во временной области после выполнения последней команды уничтожения. Если же было выполнено несколько команд уничтожения, то можно циклически переходить по полученным с их помощью данным с использованием команды M-y.

Чтобы выполнить копирование и вставку текста, достаточно лишь скопировать его назад в память в том же местоположении, в котором он был уничтожен, а затем перейти в новое местоположение и снова воспользоваться командой `C-y`. Допускается выполнять копирование текста назад в память столько раз, сколько потребуется.

Поиск и замена

Поиск текста в редакторе `emacs` выполняется с помощью команд `C-r` и `C-s`. Команда `C-s` осуществляет прямой поиск в области буфера от текущей позиции курсора до конца буфера, а команда `C-r` выполняет обратный поиск в области буфера от текущей позиции курсора до начала буфера.

После ввода команды `C-s` или `C-r` в нижней строке экрана появляется приглашение к вводу информации с запросом ввести текст для поиска. Редактор `emacs` может выполнять операции поиска двух типов.

При *инкрементном* поиске редактор `emacs` выполняет поиск текста в оперативном режиме одновременно с тем, как происходит ввод искомого слова. После ввода первой буквы редактор выделяет подсветкой все вхождения этой буквы в буфере. А вслед за вводом второй буквы выделяются подсветкой все вхождения двухбуквенной комбинации в тексте, и такой ввод может продолжаться до тех пор, пока не будет найден искомый текст.

При *неинкрементном* поиске требуется нажать клавишу `<ВВОД>` после вызова команды `C-r` или `C-s`. Это приводит к блокированию вывода запроса поиска в нижней строке текстовой области, что позволяет полностью ввести искомый текст перед началом поиска.

Чтобы заменить существующую текстовую строку новой текстовой строкой, необходимо воспользоваться командой `M-x`. Для выполнения этой команды требуется также задать текстовую команду с параметрами.

Текстовой командой является команда `replace-string`. После ввода этой команды нажмите клавишу `<ВВОД>`, и редактор `emacs` выведет запрос для указания существующей текстовой строки, подлежащей замене. Вслед за вводом этой строки снова нажмите клавишу `<ВВОД>`, и редактор `emacs` выведет запрос для указания новой строки, которая должна заменить существующий текст.

Использование буферов в редакторе emacs

Если в редакторе `emacs` развернуто несколько областей буферов, то он позволяет редактировать сразу несколько файлов. Предусмотрена возможность загружать в каждый буфер по одному файлу и переключаться с одного буфера на другой в ходе редактирования.

Чтобы загрузить новый файл в буфер, работая в редакторе `emacs`, воспользуйтесь последовательностью комбинаций клавиш `<C-x C-f>`. Это — режим поиска файла в редакторе `emacs`. В этом режиме открывается нижняя строка в окне и пользователю предоставляется возможность ввести имя файла, к редактированию которого он желает приступить. Если имя или местоположение файла не известно, достаточно нажать клавишу `<ВВОД>`. Это приводит к вызову средства просмотра файла в окне редактирования, как показано на рис. 9.3.

С помощью этого окна можно перейти к файлу, который необходимо отредактировать. Чтобы перейти на более высокий уровень в каталоге, переведите курсор на запись с двойной точкой и нажмите клавишу `<ВВОД>`. Для перехода вниз по каталогу переведите курсор на запись каталога и нажмите клавишу `<ВВОД>`. После того как будет найден файл, предназначенный для редактирования, остается только нажать клавишу `<ВВОД>`, и редактор `emacs` загрузит его в новую область буфера.

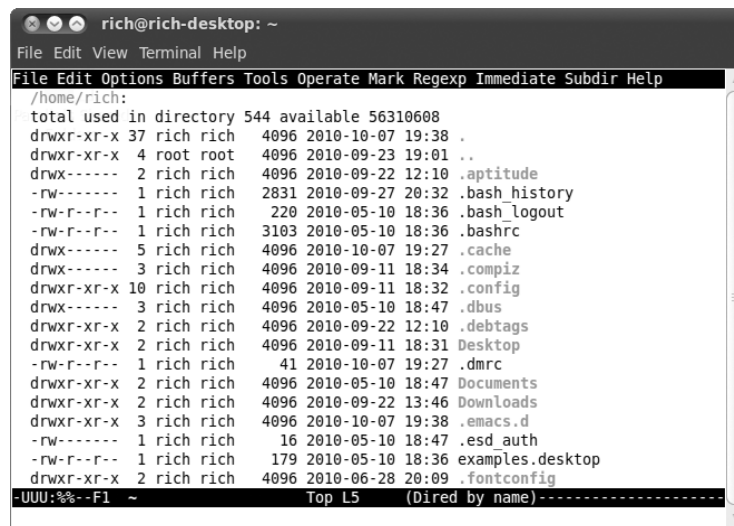


Рис. 9.3. Браузер режима поиска файла, применяемый в редакторе emacs

Предусмотрена возможность просматривать список активных областей буферов, нажимая комбинацию расширенной команды `C-x C-b`. Редактор emacs разбивает окно редактора на две части и отображает список буферов в нижней части окна. В дополнение к главному буферу редактирования редактор emacs всегда предоставляет еще два таких буфера:

- рабочая область, обозначаемая как `*scratch*`;
- область сообщений с обозначением `*Messages*`.

Рабочая область позволяет вводить команды языка программирования LISP, а также оставлять для себя заметки. В области сообщений отображаются сообщения, формируемые редактором emacs в процессе работы. Если в ходе выполнения программы emacs возникают какие-либо ошибки, они отображаются в области сообщений.

Предусмотрены следующие два способа переключения на другую область буфера в окне.

- Команда `C-x o` позволяет переключиться на окно со списком буферов. После этого можно переместиться к требуемой области буфера с помощью клавиш со стрелками и клавишу `<ВВОД>`.
- Команда `C-x b` предусматривает ввод имени области буфера, на которую необходимо переключиться.

Если будет выбран вариант с переключением с помощью окна со списком буферов, то редактор emacs откроет область буфера в новой области окна. Редактор emacs позволяет открывать несколько окон в одном сеансе. В следующем разделе показано, как управлять несколькими окнами в редакторе emacs.

Использование окон в редакторе emacs, работающем в терминальном режиме

Редактор emacs для работы в терминальном режиме был разработан задолго до того, как появилась идея применения графических окон. Но в то время такое нововведение представ-

ляло собой большой шаг вперед, поскольку обеспечивалась возможность поддерживать одновременно несколько окон редактирования с помощью основного окна emacs.

Предусмотрена возможность разбивать окно редактирования emacs на несколько окон с помощью одной из следующих двух команд.

- Команда C-x 2 разбивает окно по горизонтали на два окна.
- Команда C-x 3 разбивает окно по вертикали на два окна.

Для перехода от одного окна к другому служит команда C-x o. Следует отметить, что после создания нового окна редактор emacs использует в этом новом окне область буфера из исходного окна. После перехода в новое окно можно воспользоваться командой C-x C-f для загрузки нового файла или вызвать одну из тех команд, которые предназначены для переключения на другую область буфера в новом окне.

Чтобы закрыть окно, необходимо перейти к нему и вызвать на выполнение команду C-x 0 (0 означает ноль). Если потребуется закрыть все окна, кроме того, в котором вы работаете, воспользуйтесь командой C-x 1 (1 — это единица).

Использование редактора emacs в среде X Window

После вызова на выполнение редактора emacs в среде X Window (такой как рабочий стол KDE или GNOME) его запуск происходит в графическом режиме, как показано на рис. 9.4.

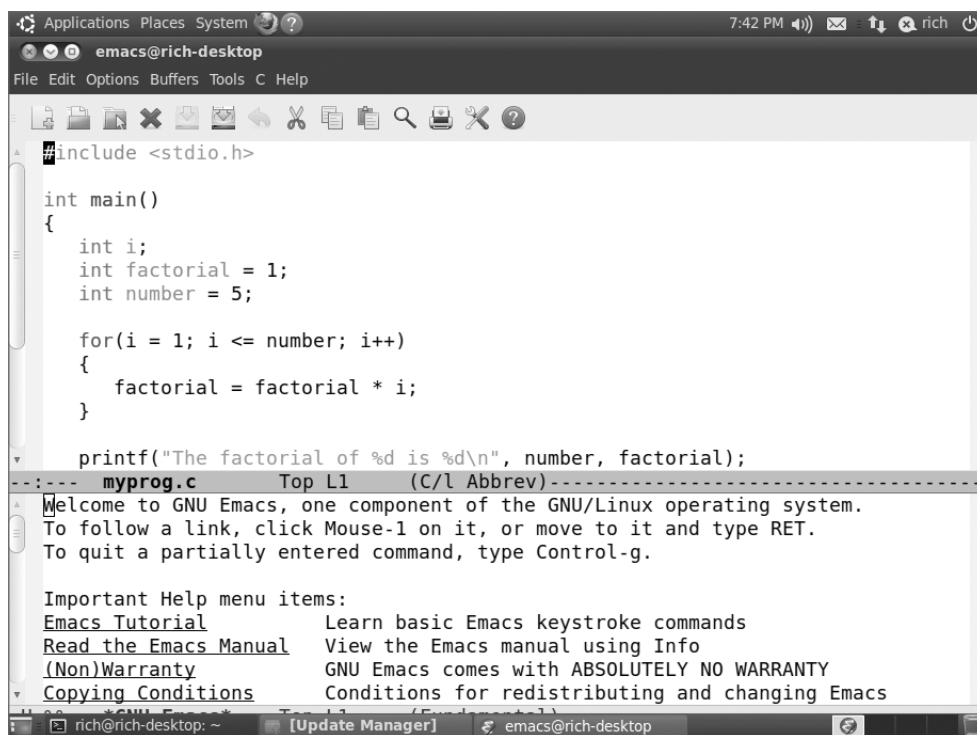


Рис. 9.4. Графическое окно редактора emacs

После успешного овладения приемами работы с редактором emacs в терминальном режиме не составляет особого труда перейти к работе в режиме X Window. Все клавиатурные команды становятся доступными как элементы строки меню. Строка меню emacs содержит следующие элементы.

- **File** (Файл). Этот элемент меню позволяет открывать файлы в окнах, создавать новые окна, закрывать окна, сохранять буфера и выводить содержимое буферов на печать.
- **Edit** (Редактирование). Позволяет вырезать и копировать выбранный текст в буфер обмена, вставлять данные буфера обмена в текущей позиции курсора, выполнять поиск и замену текста.
- **Options** (Параметры). Предоставляет настройки для многих других средств emacs, таких как выделение подсветкой, перенос по словам, определение типа курсора и задание шрифтов.
- **Buffers** (Буфера). Позволяет сформировать списки текущих доступных буферов и легко переключаться между областями буферов.
- **Tools** (Инструменты). Обеспечивает доступ к таким дополнительным функциям emacs, как доступ к интерфейсу командной строки, проверка орфографии, сравнение текста в разных файлах (что принято называть поиском различий), отправка сообщений электронной почты, работа с календарем и калькулятором.
- **Help** (Справка). Предоставляет доступ к оперативному руководству по emacs, что позволяет легко получить справку по конкретным функциям emacs.

В дополнение к обычным графическим элементам строки меню emacs часто встречается еще один отдельный элемент, характерный для того типа файла, который находится в буфере редактора. На рис. 9.4 показано, что в окне открыта программа на языке C, поэтому в редакторе emacs предусмотрен элемент меню C, позволяющий применять дополнительные настройки для выделения подсветкой синтаксиса C, а также компилировать, выполнять и отлаживать код из командной строки.

Графическое окно emacs может служить примером того, как происходил перенос более старых приложений для консоли в мир графических средств. В настоящее время, когда во многих дистрибутивах Linux предусмотрены графические рабочие столы (даже на серверах, где они фактически не требуются), графические редакторы все чаще применяются вместо редакторов всех прочих типов. В обеих популярных версиях среды рабочего стола Linux (KDE и GNOME) также предусмотрены графические текстовые редакторы, специально предназначенные для той или иной среды, которые рассматриваются в остальной части этой главы.

Семейство редакторов KDE

Перед пользователями дистрибутивов Linux, в которых применяется рабочий стол KDE (см. главу 1), открывается широкий выбор, если возникает необходимость выполнить работу в текстовом редакторе. В проекте KDE официально поддерживаются два текстовых редактора.

- **Kwrite**. Пакет редактирования текста с применением одного экрана.
- **Kate**. Полнофункциональный, многоэкранный пакет редактирования текста.

Оба эти редактора представляют собой графические текстовые редакторы, которые содержат много дополнительных функций. Редактор Kate предоставляет более развитые функции, а также позволяет воспользоваться дополнительными, привлекательными средствами, которые

не часто встречаются в стандартных текстовых редакторах. В настоящем разделе описаны каждый из этих редакторов, а также некоторые средства, которые можно использовать для упрощения работы в области редактирования сценариев командного интерпретатора.

Редактор KWrite

Основным редактором для среды KDE является KWrite. Он предоставляет простые возможности редактирования текста, такие как редактирование с учетом стилей, а также поддерживает выделение цветом синтаксических конструкций в коде и редактирование кода. Применяемое по умолчанию окно редактирования KWrite показано на рис. 9.5.

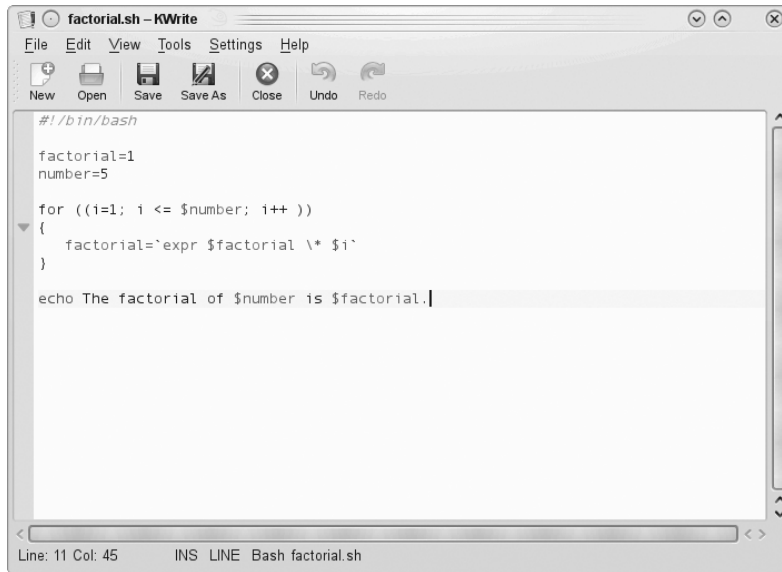


Рис. 9.5. Применяемое по умолчанию окно KWrite, в котором редактируется программа сценария командного интерпретатора

Об этом нельзя судить на основании рис. 9.5, но редактор KWrite распознает несколько типов языков программирования и позволяет использовать цветовую разметку для проведения различий между константами, функциями, комментариями и т.д. Кроме того, следует отметить, что для оператора цикла `for` предусмотрен значок, который связывает открывающие и закрывающие фигурные скобки. Этот значок называется *маркером сворачивания*. Щелкнув на подобном значке, можно свернуть целую функцию в одну строку. Это средство становится очень полезным, когда приходится работать с большими приложениями.

Окно редактирования KWrite предоставляет все возможности вырезки и вставки с использованием мыши и клавиш со стрелками. Как и любой текстовый процессор, этот редактор позволяет выделять подсветкой, вырезать (или копировать) текст в любом месте области буфера и вставлять его в любом другом месте.

Чтобы приступить к редактированию файла с использованием KWrite, можно либо выбрать KWrite в системе меню KDE на рабочем столе (некоторые дистрибутивы Linux даже создают для этого редактора значок на панели), либо запустить программу редактора в приглашении командной строки:

```
$ kwrite factorial.sh
```

Команда `kwrite` имеет несколько описанных ниже параметров командной строки, которые можно использовать для настройки способа запуска редактора.

- Параметр `--stdin` вынуждает редактор KWrite читать данные со стандартного устройства ввода данных, а не из файла.
- Параметр `--encoding` указывает, какой тип кодировки символов используется для файла.
- Параметр `--line` позволяет указать, с какого номера строки должен начаться вывод файла в окне редактора.
- Параметр `--column` указывает, с какого номера столбца начинается вывод файла в окне редактора.

В редакторе KWrite предусмотрены и строка меню, и панель инструментов в верхней части окна редактирования, что позволяет выбирать необходимые средства и вносить изменения в параметры конфигурации редактора KWrite.

Строка меню содержит следующие элементы.

- **File** (Файл). Позволяет загружать, сохранять, печатать и экспортировать текст из файлов.
- **Edit** (Правка). Дает возможность управлять текстом в области буфера.
- **View** (Представление). Управляет тем, как выглядит текст в окне редактора.
- **Bookmarks** (Закладки). Служит для обработки указателей, позволяющих возвращаться к конкретным местоположениям в тексте (может потребоваться разрешить использование этой функции с помощью параметров настройки).
- **Tools** (Инструменты). Содержит специализированные средства управления текстом.
- **Settings** (Настройки). Служит для определения способа обработки текста в редакторе.
- **Help** (Справка). Предназначен для предоставления сведений о редакторе и его командах.

В элементе строки меню **Edit** предусмотрены команды, с помощью которых можно решать любые задачи редактирования текста. Поэтому достаточно лишь выбирать нужные элементы в строке меню **Edit**, как показано в табл. 9.2, не сталкиваясь с необходимостью запоминать загадочные клавиатурные команды (которые, кстати, также поддерживаются редактором KWrite).

Таблица 9.2. Элементы меню Edit (Правка) редактора KWrite

Элемент	Описание
Undo (Отмена)	Выполняет по отношению к последнему действию или операции обратное действие или операцию
Redo (Повтор)	Выполняет действие, обратное по отношению к последнему действию отмены
Cut (Вырезка)	Удаляет выбранный текст и помещает его в буфер обмена
Copy (Копировать)	Копирует выбранный текст в буфер обмена
Copy as HTML (Копирование как HTML)	Копирует выбранный текст в буфер обмена как код HTML
Paste (Вставить)	Вставляет текущее содержимое буфера обмена в текущей позиции курсора
Select All (Выделить все)	Выделяет весь текст в редакторе
Deselect (Отменить выделение)	Отменяет выделение всего текста, который выбран в настоящее время

Элемент	Описание
Overwrite Mode (Режим замены)	Переключает режим вставки на режим замены, в котором происходит замена текста вновь введенным текстом, а не вставка нового текста
Find (Найти)	Выводит диалоговое окно Find Text (Поиск текста), которое позволяет настраивать текстовый поиск
Find Next (Найти следующее)	Повторяет последнюю операцию поиска в прямом направлении в области буфера
Find Previous (Найти предыдущее)	Повторяет последнюю операцию поиска в обратном направлении в области буфера
Replace (Заменить)	Выводит диалоговое окно Replace With (Заменить на), которое позволяет настраивать текстовый поиск и замену
Find Selected (Найти выбранное)	Находит следующее вхождение выбранного текста
Find Selected Backwards (Найти выбранное сзади)	Находит предыдущее вхождение выбранного текста
Go to Line (Перейти к строке)	Выводит диалоговое окно Goto (Перейти), которое позволяет ввести номер строки. Курсор перемещается к указанной строке

Средство Find (Поиск) имеет два режима: обычный режим, который позволяет выполнять простые операции текстового поиска, и дополненный режим поиска и замены, позволяющий в случае необходимости решать более сложные задачи поиска и замены. Для переключения между этими двумя режимами предназначена зеленая стрелка в разделе Find, как показано на рис. 9.6.

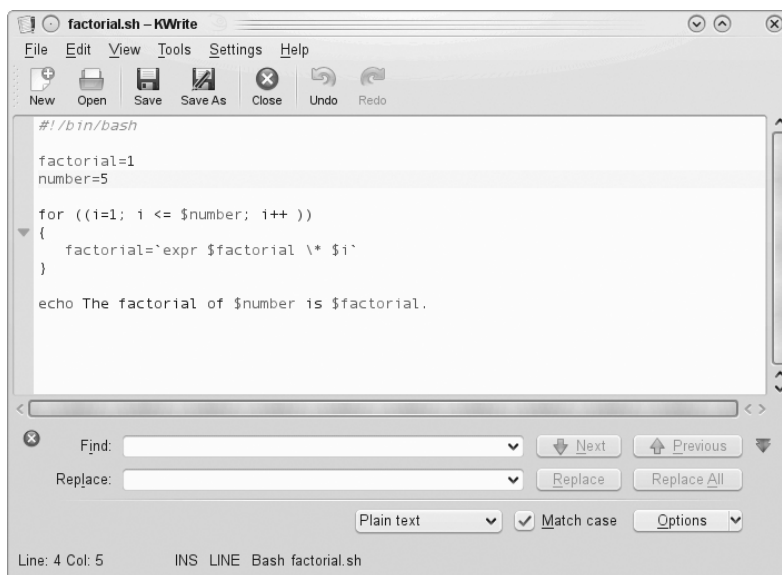


Рис. 9.6. Раздел Find редактора KWrite

Дополненный режим поиска позволяет задавать для поиска не только символьные строки, но и регулярные выражения (о чем речь пойдет в главе 19). Предусмотрены также некоторые другие параметры, которые можно использовать для настройки поиска, позволяющие, напри-

мер, указать, должен ли проводиться поиск с учетом регистра и следует ли рассматривать при поиске лишь целые слова, а не произвольные строки.

Элемент строки меню **Tools** (Инструменты) предоставляет несколько удобных средств для работы с текстом в области буфера. Инструменты, предусмотренные в редакторе Kwrite, перечислены в табл. 9.3.

Таблица 9.3. Инструменты KWrite

Инструмент	Описание
Read Only Mode (Режим доступа только для чтения)	Блокирует текст, чтобы во время работы в редакторе нельзя было вносить какие-либо изменения
Encoding (Кодировка)	Задаёт кодировку для набора символов, используемого в тексте
Spelling (Орфография)	Запускает программу проверки орфографии с начала текста
Spelling (from cursor) (Орфография (с позиции курсора))	Запускает программу проверки орфографии с текущей позиции курсора
Spellcheck Selection (Проверка орфографии в выбранном тексте)	Запускает программу проверки орфографии только на выбранном участке текста
Indent (Прямой отступ)	Увеличивает отступ абзаца на единицу
Unindent (Обратный отступ)	Уменьшает отступ абзаца на единицу
Clean Indentation (Очистка отступов)	Возвращает все отступы абзацев к исходным настройкам
Align (Выровнять)	Принудительно возвращает текущую строку или выбранные строки к заданным по умолчанию стандартным настройкам
Uppercase (Верхний регистр)	Преобразует символы в выбранном тексте или в текущей позиции курсора в символы верхнего регистра
Lowercase (Нижний регистр)	Преобразует символы в выбранном тексте или в текущей позиции курсора в символы нижнего регистра
Capitalize (Преобразовать в прописные)	Преобразовывает первую букву выбранного текста или слова в текущей позиции курсора в прописную
Join Lines (Соединить строки)	Объединяет в одну строку выбранные строки или строку в текущей позиции курсора и следующую строку
Word Wrap Document (Выполнить перенос по словам)	Обеспечивает перенос на новую строку в тексте. Если длина строки текста превышает ширину окна редактора, вывод строки текста продолжается на следующей строке экрана

Очевидно, что этот с виду простой текстовый редактор предоставляет весьма широкий набор инструментов!

Меню **Settings** (Настройки) позволяет получить доступ к диалоговому окну **Configure Editor** (Настройка редактора), которое показано на рис. 9.7.

В левой части диалогового окна **Configuration** (Конфигурация) находятся значки, позволяющие пользователю выбрать средство KWrite, подлежащее настройке. После выбора значка в правой части диалогового окна отображаются значения параметров конфигурации для соответствующего средства.

Средство **Appearance** (Внешний вид) позволяет задавать некоторые функции, которые управляют отображением текста в окне текстового редактора. Здесь можно разрешить перенос по словам, ввести нумерацию строк (что очень удобно для программистов) и обозначить папки маркерами. Средство **Fonts & Colors** (Шрифты и цвета) позволяет настраивать полную цветовую схему для редактора, определяя, какие цвета должны применяться для обозначения всех категорий текстовых конструкций в коде программы.

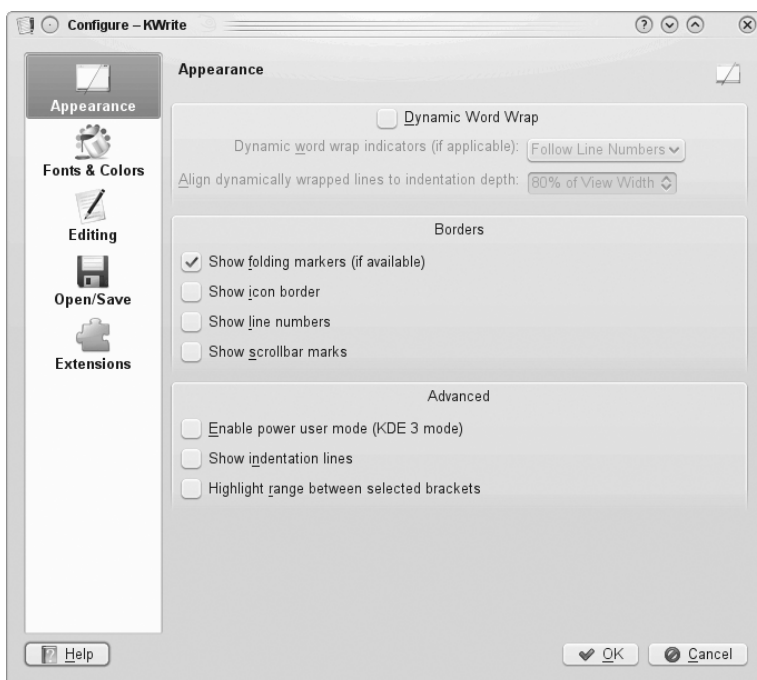


Рис. 9.7. Диалоговое окно Configure Editor редактора KWrite

Редактор Kate

Редактор Kate — ведущий редактор для проекта KDE. В нем используется такой же текстовый редактор, как и в приложении KWrite (поэтому аналогичным является и большинство предусмотренных в нем средств), но в тот же отдельный пакет включено много дополнительных средств.

Первое, что бросается в глаза после запуска редактора Kate, — диалоговое окно, показанное на рис. 9.8, а не само окно редактора!

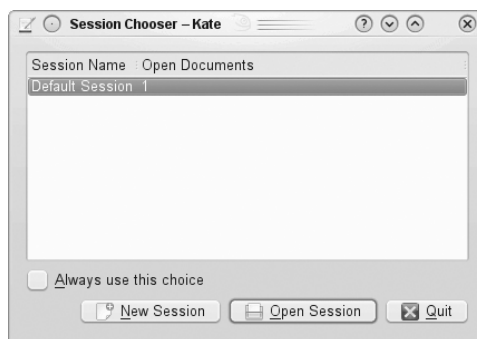


Рис. 9.8. Диалоговое окно сеанса Kate

Редактор Kate обрабатывает файлы в сеансах. Предусмотрена возможность открывать в сеансах несколько файлов, а также иметь несколько сохраненных сеансов. После запуска редактор Kate предоставляет возможность выбрать сеанс, к которому следует возвратиться. После закрытия сеанса Kate редактор запоминает документы, которые были открыты, и отображает их при следующем запуске Kate. Это позволяет легко управлять файлами из нескольких проектов путем использования отдельных рабочих пространств для каждого проекта.

После выбора сеанса открывается основное окно редактирования Kate, которое показано на рис. 9.9.

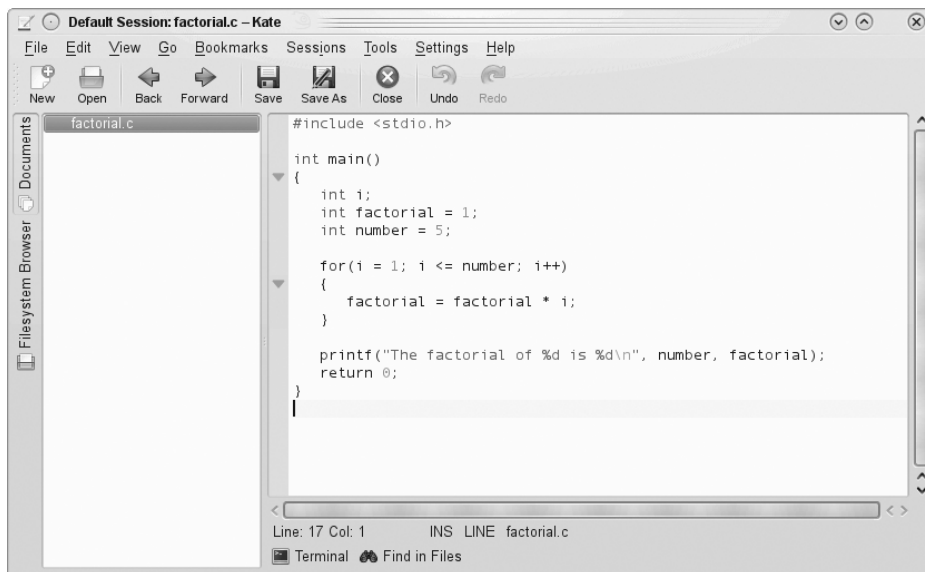


Рис. 9.9. Основное окно редактирования Kate

В области окна слева отображаются документы, открытые в настоящее время в сеансе. Предусмотрена возможность переключаться с одного документа на другой, для чего достаточно щелкнуть на имени документа. Чтобы вызвать на редактирование новый файл, перейдите на вкладку Filesystem Browser (Обозреватель файловой системы), находящуюся слева. После этого в левой области окна открывается полноценный графический обозреватель файловой системы, который позволяет находить нужные файлы с помощью графического интерфейса.

Великолепным средством редактора Kate является встроенное окно терминала, которое показано на рис. 9.10.

Запуск встроенного эмулятора терминала в программе Kate осуществляется после перехода на вкладку с обозначением терминала в нижней части окна текстового редактора (применяется эмулятор терминала Konsole среды KDE). Вызов этого средства на выполнение приводит к разбиению текущего окна редактирования по горизонтали и созданию нового окна, в котором работает терминал Konsole. Это окно позволяет вводить команды командной строки, запускать программы или проверять параметры настройки системы, не выходя из редактора! Чтобы закрыть окно терминала, достаточно ввести команду exit в командной строке.

Как показывает это средство работы с терминалом, редактор Kate обладает также способностью поддерживать несколько окон. Элемент меню Window (Окно) предоставляет следующие возможности.

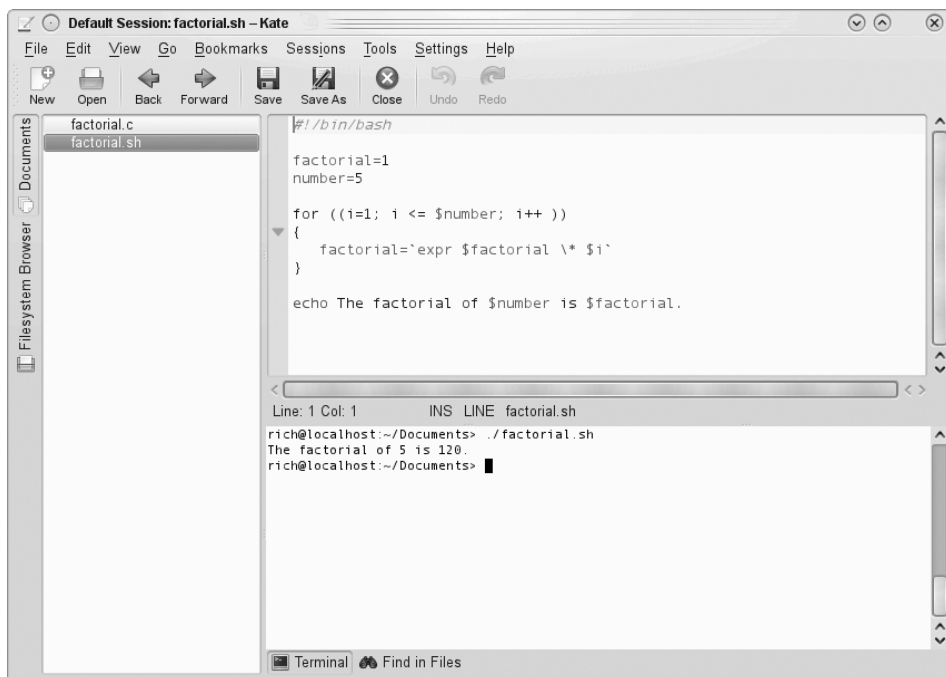


Рис. 9.10. Встроенное окно терминала Kate

- Создание нового окна Kate с использованием текущего сеанса.
- Разбиение текущего окна по вертикали для создания нового окна.
- Разбиение текущего окна по горизонтали для создания нового окна.
- Закрытие текущего окна.

Чтобы задать значения параметров конфигурации в Kate, разверните элемент строки меню **Settings** (Настройки) и выберите элемент **Configure Kate** (Настройка Kate). Появится диалоговое окно **Configuration** (Конфигурация), которое показано на рис. 9.11.

Заслуживает внимания то, что область **Editor settings** (Настройки редактора) полностью совпадает с той же областью, которая относится к редактору KWrite. Это связано с тем, что в обоих этих редакторах применяется одно и то же ядро текстового редактора. Область **Application settings** (Параметры приложения) позволяет задавать настройки для различных элементов меню Kate, например, относящихся к управлению сеансами (как показано на рис. 9.11), формированию списка документов и применению обозревателя файловой системы. Кроме того, редактор Kate поддерживает внешние сменные приложения, которые могут быть активизированы в этой области.

Редактор GNOME

Пользователи, работающие в системе Linux с применением среды рабочего стола GNOME, могут также воспользоваться графическим текстовым редактором, предусмотренным в этой среде. Это — текстовый редактор **gedit**, представляющий собой довольно простой текстовый

редактор, в котором разработчики предусмотрели также несколько дополнительных функций, которые увеличивают его привлекательность. В настоящем разделе приведено краткое описание средств gedit и показано, как использовать этот редактор при программировании сценариев командного интерпретатора.

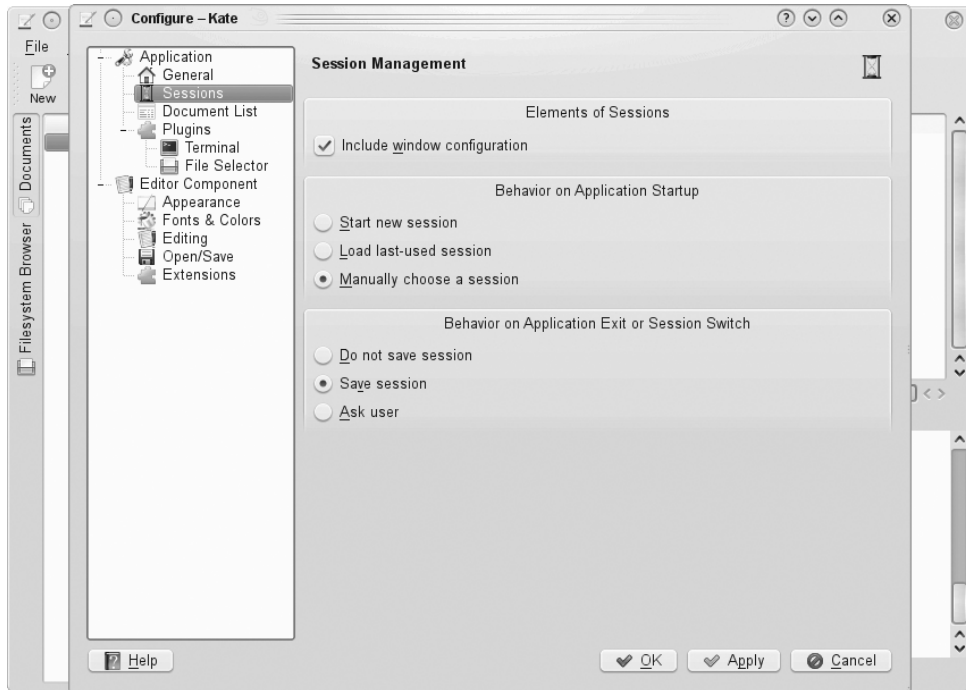


Рис. 9.11. Диалоговое окно со значениями параметров конфигурации Kate

Запуск программы gedit

В большинстве вариантов среды рабочего стола GNOME редактор gedit можно открыть, развернув элемент меню **Accessories Panel** (Панель вспомогательных программ). Если в этом меню редактор gedit отсутствует, его можно запустить из приглашения командной строки:

```
$ gedit factorial.sh myprog.c
```

После запуска программы gedit для редактирования нескольких файлов происходит загрузка всех этих файлов в отдельные буфера, а имя каждого файла отображается в заголовке одной из вкладок окна с вкладками в главном окне редактора (рис. 9.12).

В левой области главного окна редактора gedit отображаются имена документов, которые в настоящее время загружены в редактор. В правой области показано окно с вкладками, которое содержит текст из буфера. При проведении указателя мыши над каждой вкладкой появляется диалоговое окно, в котором показаны полное имя пути файла, тип MIME и кодировка набора символов, используемого в файле.

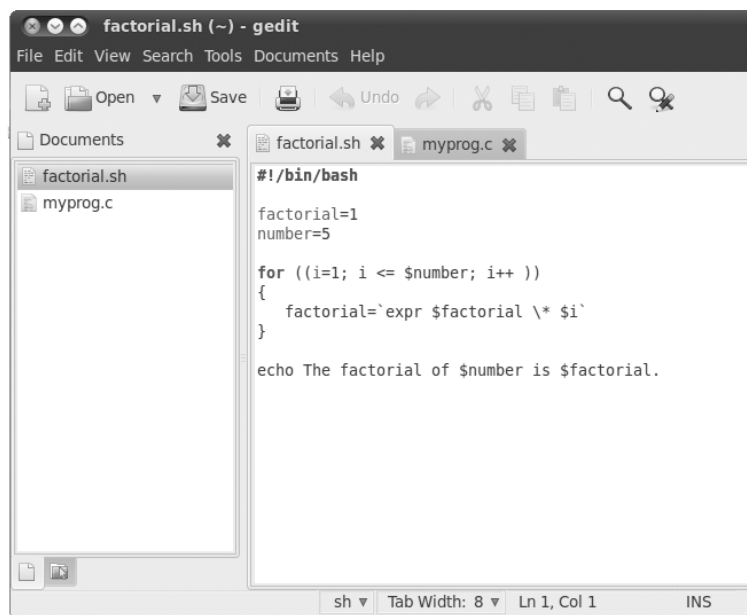


Рис. 9.12. Главное окно редактора gedit

Основные средства gedit

В дополнение к окнам редактора в программе gedit используются строка меню и панель инструментов, которые позволяют задавать применяемые средства и определять значения параметров. Панель инструментов предоставляет быстрый доступ к элементам строки меню. Предусмотренные в редакторе элементы строки меню перечислены ниже.

- **File (Файл).** Обеспечивает обработку новых файлов, сохранение существующих файлов и вывод содержимого файлов на печать.
- **Edit (Правка).** Дает возможность управлять текстом в активной области буфера и задавать параметры редактора.
- **View (Вид).** Предназначен для определения функций редактора, которые управляют отображением данных в окне, и задания режима выделения текста подсветкой.
- **Search (Поиск).** Служит для поиска и замены текста в активной области буфера редактора.
- **Tools (Инструменты).** Предназначен для получения доступа к сменным инструментам, установленным в редакторе gedit.
- **Documents (Документы).** Позволяет управлять файлами, открытыми в областях буферов.
- **Help (Справка).** Предоставляет доступ к полному руководству по программе gedit.

Очевидно, что в этом редакторе предусмотрен примерно такой же набор функций, как и в других редакторах. Меню **File** предоставляет доступ к параметру **Open Location** (Открыть доступ), который позволяет открывать файлы, полученные из сети с использованием стандартного формата URI (Uniform Resource Identifier — универсальный идентификатор ресурса), ко-

торый широко применяется в Интернете. Этот формат позволяет указать протокол, используемый для получения доступа к файлу (такой как HTTP или FTP), сервер, на котором находится файл, и полный путь на сервере для получения доступа к файлу.

В меню **Edit** предусмотрены стандартные функции вырезки, копирования и вставки, а также еще одна удобная функция, которая позволяет легко вставлять в текст значения даты и времени в нескольких форматах. Меню **Search** предоставляет доступ к стандартной функции поиска, после вызова которой открывается диалоговое окно, в котором можно ввести искомый текст, а также задать параметры поиска (указать, должен ли учитываться регистр, проводится ли поиск путем сопоставления с целым словом или строкой, и определить направление поиска). Предусмотрено также средство инкрементного поиска, которое работает в оперативном режиме, отыскивая текст одновременно с тем, как пользователь вводит символы искомой строки.

Задание параметров

Меню **Edit** содержит элемент **Preferences** (Параметры), после щелчка на котором открывается диалоговое окно **gedit Preferences** (Параметры gedit), которое показано на рис. 9.13.

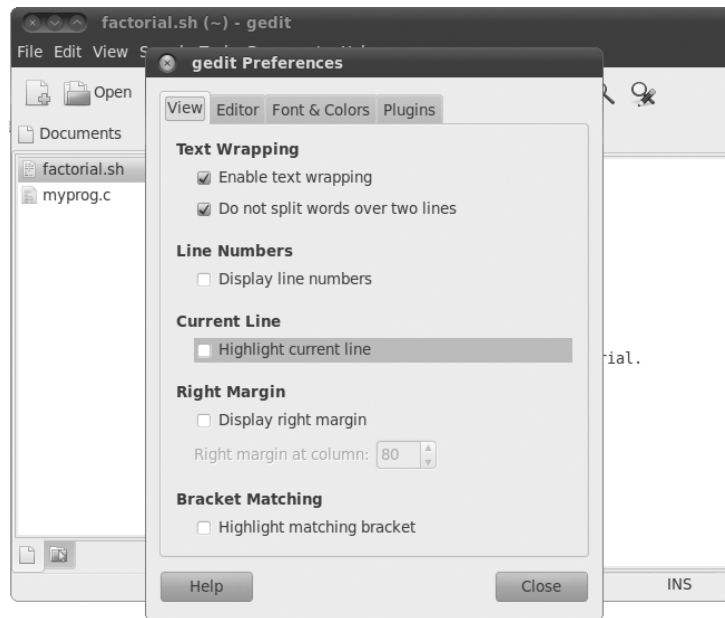


Рис. 9.13. Диалоговое окно **gedit Preferences**

Диалоговое окно **gedit Preferences** предназначено для настройки параметров, управляющих работой редактора **gedit**. Оно содержит пять областей с вкладками, предназначенных для определения средств и функций редактора, которые описаны ниже.

Вкладка **View** (Вид)

Позволяет определить параметры, управляющие тем, как программа **gedit** отображает текст в окне редактора.

- Вкладка **Text Wrapping** (Перенос текста по словам). Управляет отображением длинных строк текста в редакторе. После выбора параметра **Enabling text wrapping** (Разрешить перенос строк текста по словам) части строк текста, длина которых превышает ширину экрана, переносятся на следующие строки экрана редактора. Параметр **Do Not Split Words Over Two Lines** (Не выполнять разбивку слов на две строки) отменяет автоматическую вставку дефисов в длинных словах, в результате которой две части одного слова оказываются на двух строках экрана.
- Вкладка **Line Numbers** (Нумерация строк). С ее помощью можно предусмотреть отображение номеров строк в левой части окна редактора.
- Вкладка **Current Line** (Текущая строка). Позволяет предусмотреть выделение подсвеченной строки, в которой в настоящее время находится курсор, тем самым упрощая поиск позиции курсора в тексте.
- Вкладка **Right Margin** (Правое поле). Дает возможность определить правое поле и задать количество столбцов в окне редактора. Значение по умолчанию составляет 80 столбцов.
- Вкладка **Bracket Matching** (Согласование скобок). С ее помощью можно включить функцию выделения подсвечкой парных скобок в коде программы, что позволяет легче находить соответствующие друг другу скобки в операторах `if-then`, в циклах `for` и `while` и в других программных конструкциях, в которых применяются парные скобки.

Функции нумерации строк и согласования скобок создают для программистов такую удобную среду отладки кода, которая не часто встречается в текстовых редакторах.

Вкладка Editor (Редактор)

Позволяет задавать параметры, определяющие то, как ведется обработка в редакторе `gedit` знаков табуляции и отступов, а также указывать, как должно проводиться сохранение файлов.

- Параметр **Tab Stops** (Табуляторы). Указывает количество позиций, на которое должен сместиться курсор после нажатия клавиши `<Tab>`. Значение по умолчанию — восемь. В определение этой функции также включен флажок, который обеспечивает вставку соответствующего количества пробелов вместо одного знака табуляции.
- Параметр **Automatic Indentation** (Автоматический отступ). Если этот параметр включен, редактор `gedit` автоматически обозначает отступом все последующие строки текста в абзацах и программных конструкциях (таких как операторы `if-then` и циклы).
- Параметр **File Saving** (Сохранение файла). Позволяет воспользоваться двумя дополнительными возможностями сохранения файлов. Таковыми являются, во-первых, создание резервной копии файла после его открытия в окне редактирования и, во-вторых, автоматическое сохранение файла через заранее установленные промежутки времени.

Средство автоматического сохранения — это великолепный способ обеспечить сохранение вносимых изменений на регулярной основе. Это позволяет предотвратить случайную потерю результатов работы при аварийном отказе или прекращении подачи электроэнергии.

Вкладка Font & Colors (Шрифт и цвет)

Как и следует из ее названия, позволяет настраивать следующие два набора свойств текста.

- Элемент **Font** (Шрифт). Позволяет выбрать заданный по умолчанию шрифт `Monospace` 10 или указать определяемые пользователем шрифт и размер шрифта в диалоговом окне.

- Элемент Color Scheme (Цветовая схема). Позволяет выбрать цветовую схему, применяемую по умолчанию, для текста, фона, выбранного текста и так далее, или указать определяемый пользователем цвет для каждой категории.

Как правило, цвета, заданные по умолчанию для редактора gedit, согласуются со стандартной темой рабочего стола GNOME, выбранной в настоящее время. Если для рабочего стола будет принята другая тема, изменятся и эти цвета.

Вкладка Plugins (Подключаемые модули)

Предоставляет возможность настройки подключаемых модулей, применяемых в редакторе gedit. Подключаемые модули — это отдельные программы, способные взаимодействовать с программой gedit, предоставляя дополнительные функциональные средства. Вкладка Plugins показана на рис. 9.14.

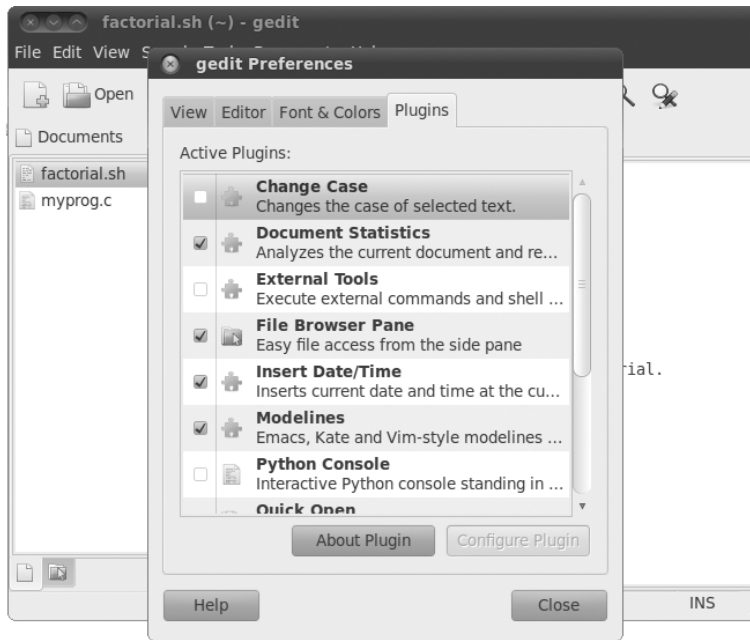


Рис. 9.14. Вкладка *Plugins Preferences* (Параметры подключаемых модулей) программы *gedit*

Для редактора gedit предусмотрен целый ряд подключаемых модулей, но не все они устанавливаются по умолчанию. В табл. 9.4 перечислены подключаемые модули, которые в настоящее время предусмотрены в редакторе gedit.

Таблица 9.4. Подключаемые модули gedit

Подключаемый модуль	Описание
Change Case (Смена регистра)	Преобразует регистр символов в выбранном тексте на противоположный
Document Statistics (Статистические данные документа)	Позволяет получить сведения о количестве слов, строк, символов и непечатаемых символов

Подключаемый модуль	Описание
External Tools (Внешние инструменты)	Предоставляет в редакторе среду командного интерпретатора, позволяющую выполнять команды и сценарии
File Browser Pane (Панель обозревателя файлов)	Предоставляет простое средство просмотра файлов, с применением которого легче решается задача выбора файлов для редактирования
Insert Date/Time (Вставка даты и времени)	Позволяет вставить значения текущей даты и времени в нескольких разных форматах в текущей позиции курсора
Modelines (Моделайны — строки управления параметрами)	Предоставляет возможность открывать строки сообщений в стиле emacs в нижней части окна редактора
Python Console (Консоль Python)	Позволяет открыть в нижней части окна редактора интерактивную консоль для ввода команд на языке программирования Python
Quick Open (Быстрое открытие)	Открывает файлы непосредственно в окне редактирования gedit
Snippets (Фрагменты)	Обеспечивает возможность сохранения часто используемых фрагментов текста для упрощения их выборки и вставки в любом месте в тексте
Sort (Сортировка)	Быстро сортирует весь файл или выбранный текст
Spell Checker (Программа проверки орфографии)	Предоставляет словарь, с помощью которого можно проверить орфографию в текстовом файле
Tag List (Список тегов)	Предоставляет список часто используемых строк, которые можно легко вставлять в текст

Подключаемые модули, которые разрешены для использования, обозначены флажком рядом с именем. Некоторые подключаемые модули, такие как **External Tools** (Внешние инструменты), предоставляют также дополнительные средства настройки после их выбора. Это позволяет задать комбинацию клавиш для запуска терминала, в котором gedit отображает вывод, а также команду запуска сеанса командного интерпретатора.

К сожалению, не все подключаемые модули устанавливаются в одном и том же месте в строке меню программы gedit. Одни из них появляются в элементе строки меню **Tools** (к ним относятся подключаемые модули **Spell Checker** и **External Tools**), тогда как другие находятся в элементе строки меню **Edit** (в частности, подключаемые модули **Change Case** и **Insert Date/Time**).

Резюме

При создании сценариев командного интерпретатора приходится прибегать к использованию текстового редактора того или иного типа. Для среды Linux предусмотрен целый ряд широко применяемых текстовых редакторов. Редактор vi, наиболее широко применяемый в мире Unix, был перенесен в Linux под именем vim. Он позволяет решать простые задачи редактирования текста с помощью консоли, для чего служит полноэкранный графический режим, не предусматривающий применение достаточно развитых графических функций. Редактор vim предоставляет много дополнительных средств редактора, таких как поиск и замена текста.

Из среды Unix в мир Linux пробился еще один популярный редактор — emacs. В Linux предусмотрены две версии emacs, для консоли и для графического режима X Window, поэтому emacs может служить мостом между старыми и новыми технологиями редактирования текста. Редактор emacs позволяет использовать несколько областей буферов, что дает возможность редактировать одновременно несколько файлов.

В рамках проекта KDE созданы два редактора, предназначенные для использования на рабочем столе KDE. Редактор KWrite — это простой редактор, предоставляющий основные средства редактирования текста, наряду с несколькими дополнительными функциями, такими как выделение цветом синтаксической конструкции в коде программы, нумерация строк и сворачивание кода. Редактор Kate предоставляет больше дополнительных функций для программистов. Одним из великолепных средств Kate является встроенное окно терминала. Пользователь может открыть сеанс работы с командной строкой непосредственно в редакторе Kate, не открывая отдельное окно эмулятора терминала. Кроме того, редактор Kate позволяет открывать несколько файлов, выводя содержимое каждого из них в отдельные окна.

В рамках проекта GNOME также предусмотрен простой текстовый редактор для программистов. Редактор gedit обладает основным набором функций текстового редактора, а также предоставляет некоторые дополнительные функции, такие как выделение цветом синтаксических конструкций в коде и нумерация строк, но в целом в задачу его разработчиков входило создание редактора, который сам по себе не отягощен ненужными функциями. Для расширения возможностей редактора gedit разработчики создали подключаемые модули, позволяющие предусмотреть для него дополнительные функции. В число предусмотренных в настоящее время подключаемых модулей входят программа проверки орфографии, эмулятор терминала и средство просмотра файлов.

Эта глава завершает ряд подготовительных глав, посвященных работе с командной строкой в Linux. В следующей части книги основное внимание уделено ознакомлению с возможностями сценариев для командного интерпретатора. Со следующей главы начинается рассмотрение вопросов создания файлов сценариев для командного интерпретатора и описания того, как вызывать эти файлы на выполнение в системе Linux. Кроме того, далее будут показаны основы сценариев для командного интерпретатора, что позволит читателю создавать простые программы, связывая друг с другом отдельные команды в сценарий, который может быть вызван на выполнение.

Основы работы со сценариями

ЧАСТЬ



В этой части...

Глава 10

Основы создания сценариев

Глава 11

Использование структурированных команд

Глава 12

Продолжение описания структурированных команд

Глава 13

Обработка ввода данных пользователем

Глава 14

Представление данных

Глава 15

Управление сценариями

ГЛАВА

10

В этой главе...

Использование нескольких команд

Создание файла сценария

Отображение сообщений

Использование переменных

Перенаправление ввода и вывода

Каналы

Выполнение математических вычислений

Выход из сценария

Резюме

Основы создания сценариев

После ознакомления в предыдущих главах с основными сведениями о системе Linux и командной строке мы готовы приступить к написанию кода. В настоящей главе приведены основные принципы написания сценариев командного интерпретатора. Читатель должен ознакомиться с этими фундаментальными понятиями, поскольку лишь после этого он сможет приступить к созданию собственных шедевров — сценариев командного интерпретатора.

Использование нескольких команд

В предыдущих главах было показано, как работать с интерфейсом командной строки (command line interface — CLI) — приглашением командного интерпретатора, вводя отдельные команды и просматривая результаты команд. По сравнению с этим основной особенностью сценариев командного интерпретатора является возможность вводить несколько команд и последовательно обрабатывать результаты, сформированные каждой командой, причем предоставляется даже возможность передавать результаты от одной команды к другой. Командный интерпретатор позволяет составлять цепочки команд так, что их выполнение происходит за один шаг.

Если требуется, чтобы две команды были выполнены вместе, их можно ввести в одной и той же строке приглашения командного интерпретатора, разделив точкой с запятой:

```
$ date ; who
Mon Feb 21 15:36:09 EST 2011
Christine tty2                2011-02-21 15:26
```

```
Samantha tty3      2011-02-21 15:26
Timothy  tty1      2011-02-21 15:26
user     tty7       2011-02-19 14:03 (:0)
user     pts/0      2011-02-21 15:21 (:0.0)
```

\$

Поздравляем! Вы только что написали сценарий командного интерпретатора. В этом простом сценарии используются только две команды командного интерпретатора `bash`. Вначале выполняется команда `date`, которая показывает текущие значения даты и времени, а за ней следует вывод команды `who`, который позволяет узнать, какие пользователи в настоящее время зарегистрированы в системе. Используя этот способ, можно соединять в цепочку любое необходимое количество команд, вплоть до предельного значения количества знаков в командной строке, равного 255 символам.

Безусловно, применение данного способа является весьма удобным при создании небольших сценариев, но он имеет серьезный недостаток, который заключается в том, что приходится снова и снова набирать весь текст цепочки команд в командной строке, когда возникает необходимость вызвать ее на выполнение. Поэтому была предусмотрена возможность объединять команды в простой текстовый файл, чтобы не приходилось каждый раз вводить команды вручную в командной строке. И если возникла необходимость выполнить эти команды, достаточно вызвать на выполнение текстовый файл.

Создание файла сценария

Чтобы поместить команды командного интерпретатора в текстовый файл, вначале необходимо воспользоваться текстовым редактором (см. главу 9) для создания файла, а затем ввести команды в этот файл.

При создании файла сценария командного интерпретатора необходимо в первой строке файла указать используемый командный интерпретатор. Ниже приведен формат применяемой для этого строки.

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

Во всех остальных строках сценария командного интерпретатора *знак дизеля* (`#`), называемый также *знаком фунта*, рассматривается как признак начала строки комментария. Строка комментария в сценарии командного интерпретатора не обрабатывается командным интерпретатором. Но первая строка файла сценария командного интерпретатора трактуется особым образом, и знак дизеля, за которым следует восклицательный знак, сообщает командному интерпретатору, какой командный интерпретатор должен быть вызван для выполнения сценария. (Это действительно так. Вы можете, например, работать в командном интерпретаторе `bash`, а для выполнения своего сценария применить другой командный интерпретатор.)

Вслед за строкой с указанием командного интерпретатора необходимо приступить к построчному вводу команд в файл, вставляя после каждой строки символ возврата каретки. Как уже было сказано, предусмотрена также возможность вводить комментарии, обозначая их с помощью знака дизеля. Рассмотрим следующий пример:

```
#!/bin/bash
# Этот сценарий отображает дату и показывает, кто зарегистрирован в системе
date
who
```

В нем находятся все элементы сценария, о которых было сказано выше. В сценариях командного интерпретатора по желанию также можно использовать точку с запятой и помещать несколько команд в одной и той же строке, но обычно принято отводить для каждой команды отдельную строку. Командный интерпретатор обрабатывает команды в том порядке, в каком они приведены в файле.

Кроме того, обратите внимание на то, что включена еще одна строка, начинающаяся со знака диэза, и добавлен комментарий. Строки, которые начинаются со знака диэза (кроме первой строки, начинающейся со знаков `#!`), не интерпретируются командным интерпретатором. Это — великолепный способ оставлять для самого себя пояснения действий, выполняемых в сценарии, чтобы можно было легко восстановить ход своих мыслей, вернувшись, допустим, к этому сценарию через два года.

Сохраните этот сценарий в файле `test1`, после чего мы будем почти готовы к выполнению следующих шагов. Дело в том, что нужно еще кое-что сделать, прежде чем появится возможность вызвать на выполнение этот вновь созданный файл сценария командного интерпретатора.

Если же попытка выполнить этот файл будет предпринята немедленно, то полученные результаты вас разочаруют:

```
$ test1
bash: test1: command not found
$
```

Первое препятствие, с которым приходится сталкиваться, состоит в том, что командному интерпретатору `bash` требуется как-то указать, где находится файл сценария. Как было описано в главе 5, в командном интерпретаторе для поиска команд используется переменная среды `PATH`. Достаточно ознакомиться со значением переменной среды `PATH`, как сразу становится понятной причина возникшей проблемы:

```
$ echo $PATH
/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/bin:/usr/bin
:/bin:/usr/local/sbin:/usr/sbin:/sbin:/home/user/bin $
```

Переменная среды `PATH` задана так, что поиск команд осуществляется только в нескольких заданных каталогах. Чтобы обеспечить для командного интерпретатора возможность найти сценарий `test1`, необходимо воспользоваться одним из следующих двух способов.

- Добавить обозначение каталога, в котором находится наш файл сценария командного интерпретатора, в переменную среды `PATH`.
- Задать абсолютный или относительный путь к файлу в качестве ссылки на созданный нами файл сценария командного интерпретатора в приглашении к вводу информации.



В некоторых дистрибутивах Linux принято добавлять каталог `$HOME/bin` к переменной среды `PATH`. Тем самым в каталоге `HOME` каждого пользователя создается место для хранения исполняемых файлов, в котором командный интерпретатор может их находить и вызывать на выполнение.

В данном примере воспользуемся вторым способом и передадим командному интерпретатору точное указание на местонахождение файла сценария. Напомним, что для формирования ссылки на файл в текущем каталоге можно использовать оператор одинарной точки в командном интерпретаторе:

```
$ ./test1
bash: ./test1: Permission denied
$
```

После этого командный интерпретатор успешно найдет файл сценария командного интерпретатора, однако обнаружится очередная проблема. Командный интерпретатор указывает, что вы не имеете разрешение на выполнение файла. Достаточно кратко ознакомиться со сведениями о правах доступа к файлу, чтобы понять, что происходит:

```
$ ls -l test1
-rw-r--r--  1 user      user          73 Sep 24 19:56 test1
$
```

При создании нового файла `test1` для этого файла были определены по умолчанию разрешения согласно действующему значению `umask`. Значение переменной `umask` задано равным `022` (см. главу 6), поэтому системой был создан файл с разрешениями для владельца файла только для чтения и записи.

Следующий шаг состоит в том, чтобы предоставить владельцу файла разрешение выполнять файл с использованием команды `chmod` (см. главу 6):

```
$ chmod u+x test1
$ ./test1
Mon Feb 21 15:38:19 EST 2011
Christine tty2      2011-02-21 15:26
Samantha tty3       2011-02-21 15:26
Timothy  tty1        2011-02-21 15:26
user     tty7         2011-02-19 14:03 (:0)
user     pts/0         2011-02-21 15:21 (:0.0) $
```

На этот раз мы добились успеха, поскольку соблюдены все условия, необходимые для беспрепятственного вызова на выполнение нового файла сценария командного интерпретатора!

Отображение сообщений

Команды командного интерпретатора чаще всего вырабатывают собственный вывод, который отображается на консоли при выполнении сценария. Тем не менее нередко возникает необходимость в добавлении своих собственных текстовых сообщений, которые могли бы помочь пользователю сценария следить за тем, какие действия осуществляются в ходе выполнения сценария. Для этого предусмотрена команда `echo`. Команда `echo` может отобразить простую текстовую строку, которая введена вслед за этой командой:

```
$ echo This is a test
This is a test
$
```

Следует учитывать, что по умолчанию не требуется использовать кавычки для обозначения начала и конца отображаемой строки. Но иногда выходные данные могут претерпеть искажения, если в строке используются кавычки:

```
$ echo Let's see if this'll work
Let's see if this'll work
$
```

В команде `echo` допускается применять для обозначения текстовых строк одинарные или двойные кавычки. Если же кавычки должны быть заданы и в тексте строки, то кавычки одного типа должны использоваться в тексте, а кавычки другого типа — служить для обозначения начала и конца строки:

```
$ echo "This is a test to see if you're paying attention"
This is a test to see if you're paying attention
$ echo 'Rich says "scripting is easy".'
Rich says "scripting is easy".
$
```

После этого все кавычки отображаются в выходных данных должным образом.

Допускается возможность добавлять инструкции `echo` в любых местах сценариев командного интерпретатора, где возникает необходимость вывести дополнительные сведения:

```
$ cat test1
#!/bin/bash
# Этот сценарий отображает дату и показывает, кто зарегистрирован
# в системе
echo The time and date are:
date
echo "Let's see who's logged into the system:"
who
$
```

После вызова этого сценария на выполнение формируется следующий вывод:

```
$ ./test1
The time and date are:
Mon Feb 21 15:41:13 EST 2011
Let's see who's logged into the system:
Christine tty2          2011-02-21 15:26
Samantha tty3           2011-02-21 15:26
Timothy  tty1           2011-02-21 15:26
user     tty7           2011-02-19 14:03 (:0)
user     pts/0          2011-02-21 15:21 (:0.0)
$
```

В данном случае полученный результат нас вполне устраивает, но иногда возникает необходимость оставить текстовую строку, выведенную командой `echo`, на той же строке, где будет находиться вывод следующей команды. Для этого можно задать в инструкции `echo` параметр `-n`. В рассматриваемом сценарии достаточно просто заменить первую строку с инструкцией `echo` следующей строкой:

```
echo -n "The time and date are: "
```

На сей раз строка должна быть выделена кавычками, поскольку необходимо обеспечить наличие пробела в конце строки, повторяемой с помощью команды `echo`. Вывод следующей команды начинается точно с того места, где заканчивается вывод повторяемой строки. Теперь результаты выполнения сценария будут выглядеть следующим образом:

```
$ ./test1
The time and date are: Mon Feb 21 15:42:23 EST 2011
Let's see who's logged into the system:
Christine tty2          2011-02-21 15:26
Samantha tty3           2011-02-21 15:26
Timothy  tty1           2011-02-21 15:26
user     tty7           2011-02-19 14:03 (:0)
user     pts/0          2011-02-21 15:21 (:0.0)
$
```

Это как раз то, что требуется! Особенно важную роль команды `echo` играют в сценариях командного интерпретатора, которые должны взаимодействовать с пользователем. Эта команда применяется также во многих других ситуациях, особенно если возникает необходимость отобразить значения переменных сценария. Эта тема рассматривается в следующем разделе.

Использование переменных

Разумеется, такие сценарии командного интерпретатора, которые состоят из отдельных команд, также важны, но иногда возникают ситуации, когда этого становится недостаточно. Например, зачастую обнаруживается необходимость включить другие данные в команды командного интерпретатора в целях обработки информации. Эту задачу можно решить с использованием *переменных*. Переменные позволяют сохранять на время информацию в сценарии командного интерпретатора для последующего применения в других командах в сценарии. В настоящем разделе показано, как использовать переменные в сценариях командного интерпретатора.

Переменные среды

В предыдущих главах уже были показаны в действии переменные Linux определенного типа. Речь идет о переменных среды, предусмотренных в системе Linux, которые рассматривались в главе 5. Доступ к значениям этих переменных может быть также получен в сценариях командного интерпретатора.

Командный интерпретатор поддерживает переменные среды, которые отслеживают конкретную системную информацию, такую как имя системы, имя пользователя, зарегистрированного в системе, идентификатор пользователя в системе (называемый просто *идентификатором пользователя* — `UID`), применяемый по умолчанию исходный каталог пользователя и путь поиска файлов, используемый командным интерпретатором для поиска программ. Чтобы ознакомиться с полным списком активных переменных среды, к которым может быть получен доступ, достаточно воспользоваться командой `set`:

```
$ set
BASH=/bin/bash
...
HOME=/home/Samantha
HOSTNAME=localhost.localdomain
HOSTTYPE=i386
IFS=$' \t\n'
IMSETTINGS_INTEGRATE_DESKTOP=yes
IMSETTINGS_MODULE=none
LANG=en_US.utf8
LESSOPEN='|/usr/bin/lesspipe.sh %s'
LINES=24
LOGNAME=Samantha
...
```

Значения этих переменных среды можно вставлять в сценарии, для чего необходимо указать имя переменной среды, которому предшествует знак доллара. Такой способ доступа к переменным среды демонстрируется в следующем сценарии:

```
$ cat test2
#!/bin/bash
# отображение информации о пользователе.
echo "User info for userid: $USER"
echo UID: $UID
echo HOME: $HOME
$
```

Переменные среды `$USER`, `$UID` и `$HOME` использовались для отображения интересующей нас информации о зарегистрированном пользователе. Вывод этого сценария должен выглядеть примерно так:

```
$chmod u+x test2
$ ./test2
User info for userid: Samantha
UID: 1001
HOME: /home/Samantha
$ $
```

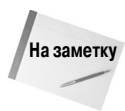
Обратите внимание на то, что переменные среды в командах `echo` заменяются их текущими значениями при выполнении сценария. Заслуживает также внимания то, что мы смогли поместить переменную системы `$USER` в двойные кавычки в первой строке, а сценарий командного интерпретатора, несмотря на это, был выполнен в соответствии с намеченной нами целью. Тем не менее данный метод вывода символов форматирования текста имеет определенный недостаток. Рассмотрим, что происходит в следующем примере:

```
$ echo "The cost of the item is $15"
The cost of the item is 5
```

Очевидно, что полученный результат вряд ли соответствует ожидаемому. Каждый раз, когда в сценарии обнаруживается знак доллара в строке, заключенной в кавычки, предполагается наличие ссылки на переменную. В этом примере командный интерпретатор при обработке сценария пытался отобразить значение переменной `$1` (которая не определена), а затем цифру 5. Чтобы буквально отобразить знак доллара, необходимо сопроводить его префиксом в виде обратной косой черты:

```
$ echo "The cost of the item is \$15"
The cost of the item is $15
```

Этот результат гораздо лучше. В результате введения обратной косой черты командный интерпретатор смог обработать сценарий так, что знак доллара был принят за действительный знак доллара, а не за обозначение переменной. В следующем разделе показано, как создавать собственные переменные в своих сценариях.



Часто можно также видеть, что для ссылок на переменные используется формат `${variable}`. В таком случае дополнительные фигурные скобки вокруг имени переменной *variable* применяются для того, чтобы проще было определить имя переменной, перед которой стоит знак доллара.

Пользовательские переменные

В сценариях командного интерпретатора можно не только использовать переменные среды, но также задавать и включать свои собственные переменные. Задание переменных позволяет на время сохранять данные и использовать их во всем сценарии, в результате чего сценарий

командного интерпретатора в большей степени становится похожим на настоящую компьютерную программу.

Имена пользовательских переменных могут представлять собой любую текстовую строку длиной до 20 символов, состоящую из букв, цифр и символов подчеркивания. В именах пользовательских переменных учитывается регистр, поэтому переменная `Var1` рассматривается как отличная от переменной `var1`. Забывая об этом простом правиле, начинающие программисты на языках сценариев часто допускают трудно диагностируемые ошибки.

Значения присваиваются пользовательским переменным с помощью знака равенства (`=`). Между именем переменной, знаком равенства и значением не должно быть ни одного пробела (неопытные программисты иногда забывают и об этом правиле). Ниже приведено несколько примеров присваивания значений пользовательским переменным.

```
var1=10
var2=-57
var3=testing
var4="still more testing"
```

При обработке сценариев командного интерпретатора тип данных, используемый для представления значения переменной, определяется автоматически. Переменные, которые определены в сценарии командного интерпретатора, сохраняют свои значения на протяжении всего времени выполнения сценария и уничтожаются после завершения сценария.

Для ссылки на пользовательские переменные, как и на переменные системы, применяется знак доллара:

```
$ cat test3
#!/bin/bash
# проверка переменных
days=10
guest="Katie"
echo "$guest checked in $days days ago"
days=5
guest="Jessica"
echo "$guest checked in $days days ago"
$
```

Выполнение этого сценария приводит к получению следующего вывода:

```
$ chmod u+x test3
$ ./test3
Katie checked in 10 days ago
Jessica checked in 5 days ago
$
```

При каждой ссылке на переменную происходит возврат значения, которое присвоено переменной в настоящее время. Следует помнить, что при ссылке на значение переменной знак доллара должен быть приведен, а если переменная указана в целях присваивания ей значения, то знак доллара не используется. Ниже приведен пример, позволяющий пояснить сказанное.

```
$ cat test4
#!/bin/bash
# Присваивание значения одной переменной другой

value1=10
value2=$value1
```

```
echo The resulting value is $value2
$
```

Если же значение переменной, в данном случае `value1`, применяется в операторе присваивания, знак доллара все еще должен использоваться. При выполнении этого кода формируется следующий вывод:

```
$ chmod u+x test4
$ ./test4
The resulting value is 10
$
```

Если мы забудем привести знак доллара и оформим строку присваивания `value2` примерно так:

```
value2=value1
```

то получим следующий вывод:

```
$ ./test4
The resulting value is value1
$
```

Командный интерпретатор интерпретирует имя переменной без знака доллара как обычную текстовую строку, а это, очевидно, не входило в наши намерения.

Обратная одинарная кавычка

В качестве одного из наиболее полезных средств сценариев командного интерпретатора можно назвать непритязательный символ обратной кавычки, который в мире Linux обычно именуется обратной одинарной кавычкой (`'`). Следует всегда помнить, что это — не тот обычный символ одинарной кавычки, который используется в составе строк. Символ обратной одинарной кавычки в основном используется лишь в сценариях командного интерпретатора, поэтому многие даже не представляют себе, где этот символ находится на клавиатуре. Но специалисты по Linux должны знать, где найти этот символ, поскольку он является крайне важным элементом многих сценариев командного интерпретатора.



На клавиатуре персонального компьютера символ обратной одинарной кавычки обычно можно ввести с помощью той же клавиши, что и символ тильды (`~`).

Символы обратной одинарной кавычки позволяют присвоить вывод команды командного интерпретатора переменной. На первый взгляд кажется, что сказанное выше не представляет собой что-то особенное, но фактически эта операция является одним из важных строительных блоков в программировании сценариев.

В символы обратной одинарной кавычки должна быть заключена вся команда командной строки:

```
testing=`date`
```

Командный интерпретатор выполняет команду, заключенную в обратные одинарные кавычки, и присваивает полученный вывод переменной `testing`. Ниже приведен пример создания переменной с использованием вывода одной из обычных команд командного интерпретатора.

```
$ cat test5
#!/bin/bash
```

```
# использование обратной одинарной кавычки
testing='date'
echo "The date and time are: " $testing
$
```

Переменная `testing` получает вывод из команды `date`, после чего полученные данные используются в инструкции `echo` для их отображения. Вызов этого сценария командного интерпретатора на выполнение приводит к получению следующих результатов:

```
$ chmod u+x test5
$ ./test5
The date and time are: Mon Jan 31 20:23:25 EDT 2011
$
```

Разумеется, этот пример не так уж оригинален (тех же результатов можно было бы достичь столь же легко, поместив команду в инструкцию `echo`), но важно то, что после перехвата вывода команды и присваивания его переменной появляется возможность проводить с полученными данными самые разные манипуляции.

Ниже приведен часто цитируемый пример использования обратной одинарной кавычки для перехвата значения текущей даты и дальнейшего применения этого значения для создания уникального имени файла в сценарии.

```
#!/bin/bash
# копирование листинга каталога /usr/bin в файл журнала
today='date +%y%m%d'
ls /usr/bin -al > log.$today
```

Переменной `today` присваивается отформатированный вывод команды `date`. Этот прием широко используется для извлечения информации о текущей дате и формирования имен файлов журналов. Формат `+%y%m%d` служит для команды `date` указанием на то, что дата должна быть представлена как состоящая из последних двух цифр обозначения года, номера месяца и числа месяца:

```
$ date +%y%m%d
110131
$
```

В сценарии это значение присваивается переменной, которая затем используется в составе строки формирования имени файла. Сам файл содержит перенаправленный вывод листинга каталога (речь об этом пойдет ниже, в разделе “Перенаправление ввода и вывода”). После выполнения этого сценария в текущем каталоге должен появиться новый файл:

```
-rw-r--r--  1 user  user      769 Jan 31 10:15 log.110131
```

Имя файла журнала, созданного в каталоге, сформировано с использованием значения переменной `$today` в составе имени файла. Сам файл журнала содержит листинг каталога `/usr/bin`. Если бы этот сценарий был выполнен на следующий день, то файл журнала получил бы имя `log.110201`. Иными словами, в каждый новый день создается новый файл.

Перенаправление ввода и вывода

Иногда возникает необходимость сохранить вывод команды, а не просто показать его на мониторе. В командном интерпретаторе `bash` предусмотрено несколько разных операторов, которые позволяют *перенаправить* вывод команды в альтернативное местоположение (например,

в файл). Перенаправление применяется как для ввода, так и для вывода; в последнем случае, например, содержимое файла может быть перенаправлено в команду в качестве входных данных. В настоящем разделе описано, что должно быть сделано для использования перенаправления в конкретных сценариях командного интерпретатора.

Перенаправление вывода

Наиболее простым типом перенаправления является передача вывода команды в файл. В командном интерпретаторе `bash` для этого используется знак “больше” (`>`):

```
command > outputfile
```

Все, что должно было появиться на мониторе в качестве вывода команды, вместо этого сохраняется в указанном выходном файле:

```
$ date > test6
$ ls -l test6
-rw-r--r--  1 user      user      29 Feb 10 17:56 test6
$ cat test6
Thu Feb 10 17:56:58 EDT 2011
$
```

В операторе перенаправления создается файл `test6` (с использованием заданного по умолчанию значения `umask`), затем происходит перенаправление вывода из команды `date` в файл `test6`. Если выходной файл уже существует, то оператор перенаправления перезаписывает существующий файл с применением данных нового файла:

```
$ who > test6
$ cat test6
user      pts/0      Feb 10 17:55
$
```

Теперь содержимое файла `test6` включает вывод команды `who`.

Иногда возникает такая необходимость, чтобы вместо перезаписи содержимого файла вывод команды добавлялся к существующему файлу. Подобная ситуация, например, встречается при использовании файла журнала для документирования каких-то действий, происходящих в системе. В этом случае для добавления данных могут использоваться два знака “больше” (`>>`):

```
$ date >> test6
$ cat test6
user      pts/0      Feb 10 17:55
Thu Feb 10 18:02:14 EDT 2011
$
```

Файл `test6` все еще содержит данные, первоначально полученные в результате предыдущего выполнения команды `who`, но в дополнение к этому появился вновь полученный вывод команды `date`.

Перенаправление ввода

Операция перенаправления ввода является обратной по отношению к перенаправлению вывода. При перенаправлении вывода берется вывод команды и перенаправляется в файл, а при перенаправлении ввода берется содержимое файла и перенаправляется в команду.

Для обозначения перенаправления ввода используется знак “меньше” (<):

```
command < inputfile
```

Проще всего можно запомнить правила обращения с символами перенаправления так, что команда всегда должна быть приведена в командной строке в первую очередь, а за ней следует знак перенаправления, который “указывает” направление передачи данных. Знак “меньше” служит указанием того, что данные передаются от входного файла в команду.

Ниже приведен пример использования перенаправления ввода в сочетании с командой `wc`.

```
$ wc < test6
      2      11      60
$
```

Команда `wc` подсчитывает количество элементов текста в данных. По умолчанию она возвращает три значения:

- количество строк в тексте;
- количество слов в тексте;
- количество байтов в тексте.

Перенаправив текстовый файл в команду `wc`, можно быстро определить количество строк, слов и байтов в файле. Данный пример показывает, что в файле `test6` находится текст, состоящий из 2 строк, 11 слов и 60 байтов.

Предусмотрен еще один метод перенаправления ввода, называемый *перенаправлением ввода через буфер*. Этот метод позволяет задавать данные для перенаправления ввода в командной строке, а не в файле. На первый взгляд может показаться непонятным, для чего это нужно, но данный процесс имеет несколько областей применения (в частности, такие области, о которых речь пойдет ниже, в разделе “Выполнение математических вычислений”).

Для обозначения перенаправления ввода через буфер служит удвоенный знак “меньше” (<<). Кроме этого символа, необходимо задать текстовый маркер, который обозначает начало и конец данных, используемых для ввода. В качестве текстового маркера можно задать любое строковое значение, но оно должно быть одинаковым и в начале, и в конце данных:

```
command << marker
data
marker
```

Если перенаправление ввода через буфер используется в командной строке, то командный интерпретатор запрашивает вводимые данные с помощью вторичного приглашения к вводу информации, которое определено в переменной среды `PS2` (см. главу 5). Ниже приведен пример применения такого перенаправления ввода.

```
$ wc << EOF
> test string 1
> test string 2
> test string 3
> EOF
      3      9      42
$
```

Вторичное приглашение к вводу информации появляется в качестве запроса для дальнейшего ввода данных в одной строке за другой, пока не будет введено строковое значение текстового маркера. После этого команда `wc` выполняет подсчет количества строк, слов и байтов в данных, заданных с помощью перенаправления ввода через буфер.

Каналы

Иногда возникает необходимость передавать выход одной команды на вход другой. Для этого можно использовать перенаправление, но это связано с применением довольно сложных конструкций:

```
$ rpm -qa > rpm.list
$ sort < rpm.list
abrt-1.1.14-1.fc14.i686
abrt-addon-ccpp-1.1.14-1.fc14.i686
abrt-addon-kerneloops-1.1.14-1.fc14.i686
abrt-addon-python-1.1.14-1.fc14.i686
abrt-desktop-1.1.14-1.fc14.i686
abrt-gui-1.1.14-1.fc14.i686
abrt-libs-1.1.14-1.fc14.i686
abrt-plugin-bugzilla-1.1.14-1.fc14.i686
abrt-plugin-logger-1.1.14-1.fc14.i686
abrt-plugin-runapp-1.1.14-1.fc14.i686
acl-2.2.49-8.fc14.i686
...
```

Рассмотрим такой пример. Команда `rpm` управляет пакетами программ, установленными в системе с использованием системы управления пакетами (Package Management system — RPM) Red Hat наподобие системы Fedora, как показано выше. Будучи вызванной с параметрами `-qa`, эта команда формирует список существующих установленных пакетов, но не обязательно в каком-то конкретном порядке. Если требуется найти конкретный пакет или группу пакетов, то при использовании вывода команды `rpm` в непосредственном виде данная задача может стать затруднительной.

В данном случае вывод был перенаправлен из команды `rpm` в файл `rpm.list` с помощью перенаправления в стандартное устройство вывода. После завершения команды появляется файл `rpm.list`, содержащий список всех установленных пакетов программ в рассматриваемой системе. Затем было использовано перенаправление входа для передачи содержимого файла `rpm.list` в команду `sort`, которая выполняет сортировку имен пакетов в алфавитном порядке.

Очевидно, что этот метод достижения поставленной цели оказался применимым, но еще раз отметим, что его нельзя назвать самым простым методом получения требуемой информации. Вместо перенаправления вывода команды в файл можно перенаправить этот вывод непосредственно в другую команду. Такой процесс называется передачей данных по каналу.

Как и обратная одинарная кавычка (`'`), символ передачи данных по каналу не часто применяется в других целях, кроме написания сценариев командного интерпретатора. Этот символ выглядит как две вертикальные черты, расположенные друг над другом. Однако на печати этот символ канала часто отображается как одна вертикальная черта (`|`). При работе на клавиатуре для персонального компьютера для ввода этого символа обычно применяется та же клавиша, где представлен символ обратной косой черты (`\`). Символ канала помещается между командами для перенаправления вывода из одной команды в другую:

```
command1 | command2
```

Не следует рассматривать передачу данных по каналу как последовательное выполнение двух команд, связанных символом канала. Фактически система Linux выполняет обе команды

одновременно, незаметно для постороннего взгляда соединяя в системе выход одной команды с входом другой. Как только первая команда вырабатывает какой-то вывод, он немедленно передается во вторую команду. Для передачи этих данных не используются какие-либо промежуточные файлы или области буферов.

Итак, применяя передачу данных по каналу, мы можем легко перенаправить с помощью символа канала вывод команды `rpm` непосредственно в команду `sort` для получения требуемых результатов:

```
$ rpm -qa | sort
abrt-1.1.14-1.fc14.i686
abrt-addon-ccpp-1.1.14-1.fc14.i686
abrt-addon-kerneloops-1.1.14-1.fc14.i686
abrt-addon-python-1.1.14-1.fc14.i686
abrt-desktop-1.1.14-1.fc14.i686
abrt-gui-1.1.14-1.fc14.i686
abrt-libs-1.1.14-1.fc14.i686
abrt-plugin-bugzilla-1.1.14-1.fc14.i686
abrt-plugin-logger-1.1.14-1.fc14.i686
abrt-plugin-runapp-1.1.14-1.fc14.i686
acl-2.2.49-8.fc14.i686
...
```

Но даже очень зоркий и внимательный читатель не способен уследить за выводом, формируемым командой в достаточно большом объеме. Функция передачи данных по каналу действует в режиме реального времени, поэтому непосредственно после начала выработки данных в команде `rpm` команда `sort` приступает к делу, сортируя эту информацию. К тому времени, как команда `rpm` завершает вывод данных, в команде `sort` заканчивается сортировка данных и начинается их отображение на мониторе.

Какие-либо ограничения на количество операторов передачи данных по каналу, которые могут использоваться в инструкции, не предусмотрены. По мере необходимости можно продолжать передачу по каналу вывода одних команд в другие команды в целях дальнейшего усовершенствования выполняемой операции.

В данном случае необходимо добиться того, чтобы вывод команды `sort` не пробегал по экрану так быстро, поэтому можно воспользоваться одной из команд страничного отображения текста (такой как `less` или `more`), которые позволяют останавливать вывод после формирования каждого экрана данных:

```
$ rpm -qa | sort | more
```

В этой последовательности команд выполняется команда `rpm`, ее вывод передается по каналу в команду `sort`, вывод которой отправляется с помощью канала в команду `more` для отображения с остановкой после развертывания каждого экрана с информацией. Вот теперь мы можем воспользоваться паузами и с удобством читать то, что находится на дисплее, перед продолжением, как показано на рис. 10.1.

Чтобы наш сценарий стал еще более интересным, можем наряду с передачей данных по каналу воспользоваться перенаправлением, чтобы сохранить полученный вывод в файле:

```
$ rpm -qa | sort > rpm.list
$ more rpm.list
abrt-1.1.14-1.fc14.i686
abrt-addon-ccpp-1.1.14-1.fc14.i686
abrt-addon-kerneloops-1.1.14-1.fc14.i686
```

```

abrt-addon-python-1.1.14-1.fc14.i686
abrt-desktop-1.1.14-1.fc14.i686
abrt-gui-1.1.14-1.fc14.i686
abrt-libs-1.1.14-1.fc14.i686
abrt-plugin-bugzilla-1.1.14-1.fc14.i686
abrt-plugin-logger-1.1.14-1.fc14.i686
abrt-plugin-runapp-1.1.14-1.fc14.i686
acl-2.2.49-8.fc14.i686
...

```

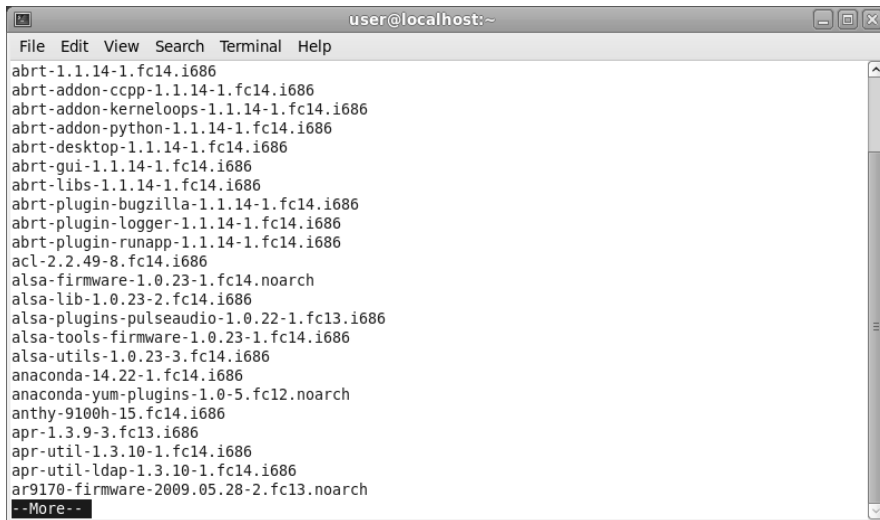


Рис. 10.1. Использование передачи данных по каналу для отправки данных в команду more

Как и следовало ожидать, данные в файле `rpm.list` теперь отсортированы!

Передача по каналу результатов команд, вырабатывающих большой объем вывода, в команду `more` является одним из наиболее широко применяемых способов использования оператора канала. Такая конструкция особенно часто встречается при работе с командой `ls`, как показано на рис. 10.2.

Команда `ls -l` вырабатывает длинный листинг всех файлов в каталоге. Если количество файлов в каталоге достаточно велико, то объем листинга может стать весьма значительным. Перенаправляя вывод в команду `more`, можно обеспечить принудительный останов вывода по завершении отображения каждого очередного экрана данных.

Выполнение математических вычислений

Еще одним важным средством любого языка программирования является способность манипулировать числами. К сожалению, в сценариях командного интерпретатора эта задача решается довольно сложно. Предусмотрены два способа выполнения математических операций в сценариях командного интерпретатора, которые рассматриваются ниже.

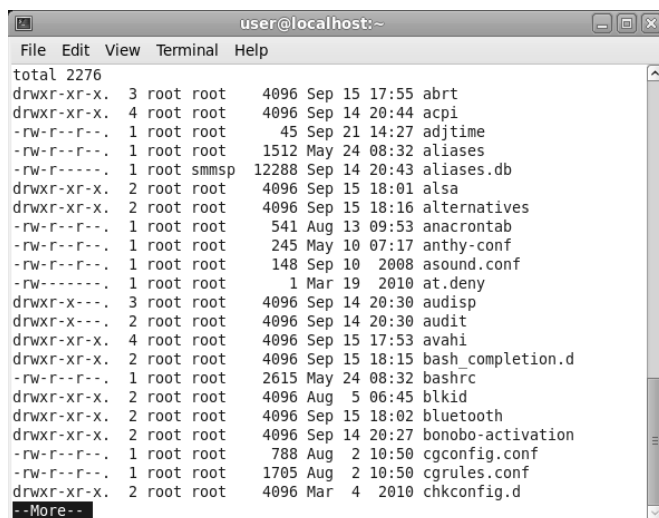


Рис. 10.2. Использование команды `more` в сочетании с командой `ls`

Команда `expr`

Первоначально в командном интерпретаторе Bourne была предусмотрена специальная команда, предназначенная для обработки математических выражений. Команда `expr` позволяла обрабатывать выражения, заданные в командной строке, но для этого применялись чрезвычайно сложные конструкции:

```
$ expr 1 + 5
6
```

Команда `expr` распознает некоторые математические и строковые операторы, которые показаны в табл. 10.1.

Таблица 10.1. Операторы команды `expr`

Оператор	Описание
<code>ARG1 ARG2</code>	Возвращает <code>ARG1</code> , если ни один из аргументов не имеет значения <code>null</code> или <code>0</code> ; в противном случае возвращает <code>ARG2</code>
<code>ARG1 & ARG2</code>	Возвращает <code>ARG1</code> , если ни один из аргументов не имеет значения <code>null</code> или <code>0</code> ; в противном случае возвращает <code>0</code>
<code>ARG1 < ARG2</code>	Возвращает <code>1</code> , если <code>ARG1</code> меньше <code>ARG2</code> ; в противном случае возвращает <code>0</code>
<code>ARG1 <= ARG2</code>	Возвращает <code>1</code> , если <code>ARG1</code> меньше или равен <code>ARG2</code> ; в противном случае возвращает <code>0</code>
<code>ARG1 = ARG2</code>	Возвращает <code>1</code> , если <code>ARG1</code> равен <code>ARG2</code> ; в противном случае возвращает <code>0</code>
<code>ARG1 != ARG2</code>	Возвращает <code>1</code> , если <code>ARG1</code> не равен <code>ARG2</code> ; в противном случае возвращает <code>0</code>
<code>ARG1 >= ARG2</code>	Возвращает <code>1</code> , если <code>ARG1</code> больше или равен <code>ARG2</code> ; в противном случае возвращает <code>0</code>
<code>ARG1 > ARG2</code>	Возвращает <code>1</code> , если <code>ARG1</code> больше <code>ARG2</code> ; в противном случае возвращает <code>0</code>
<code>ARG1 + ARG2</code>	Возвращает арифметическую сумму <code>ARG1</code> и <code>ARG2</code>
<code>ARG1 - ARG2</code>	Возвращает арифметическую разность <code>ARG1</code> и <code>ARG2</code>

Оператор	Описание
ARG1 * ARG2	Возвращает арифметическое произведение ARG1 и ARG2
ARG1 / ARG2	Возвращает арифметическое частное от деления ARG1 на ARG2
ARG1 % ARG2	Возвращает арифметический остаток от деления ARG1 на ARG2
STRING : REGEXP	Возвращает результат сопоставления с шаблоном, если REGEXP сопоставляется с шаблоном в STRING
match STRING REGEXP	Возвращает результат сопоставления с шаблоном, если REGEXP сопоставляется с шаблоном в STRING
substr STRING POS LENGTH	Возвращает подстроку длиной LENGTH символов, начиная с позиции POS (с отсчетом от 1)
index STRING CHARS	Возвращает позицию в строке STRING, в которой найдена подстрока CHARS; в противном случае возвращает 0
length STRING	Возвращает числовое значение длины строки STRING
+ TOKEN	Интерпретирует TOKEN как строку, даже если это — ключевое слово
(EXPRESSION)	Возвращает значение EXPRESSION

Очевидно, что эти стандартные операторы в команде `expr` действуют вполне приемлемо, но при их использовании в сценарии или в командной строке возникают проблемы. Многие операторы команды `expr` имеют в командном интерпретаторе другие значения (в качестве примера можно указать звездочку). Применение таких операторов в команде `expr` приводит к получению непредсказуемых результатов:

```
$ expr 5 * 2
expr: syntax error
$
```

Для решения этой проблемы приходится использовать экранирующий символ командного интерпретатора (обратную косую черту) для обозначения всех символов, которые могут быть неправильно обработаны командным интерпретатором, и только после этого передавать полученное выражение в команду `expr`:

```
$ expr 5 \* 2
10
$
```

Вот теперь мы в самом деле получили нечто недоступное для восприятия! Столь же громоздкие конструкции возникают при использовании команды `expr` в сценариях командного интерпретатора:

```
$ cat test6
#!/bin/bash
# Пример использования команды expr
var1=10
var2=20
var3=`expr $var2 / $var1`
echo The result is $var3
```

Чтобы присвоить результат обработки математического выражения переменной, мы были вынуждены использовать символ обратной одинарной кавычки для извлечения вывода из команды `expr`:

```
$ chmod u+x test6
$ ./test6
The result is 2
$
```

К счастью, в командном интерпретаторе `bash` реализовано определенное усовершенствование, связанное с выполнением математических операций, как будет показано в следующем разделе.

Использование квадратных скобок

Командный интерпретатор `bash` продолжает поддерживать команду `expr`, поскольку он должен оставаться совместимым с командным интерпретатором `Bourne`. Однако в `bash` предусмотрен намного более простой способ выполнения математических вычислений. В командном интерпретаторе `bash` при использовании операции присваивания результата математических вычислений переменной можно включить математическое выражение в конструкцию, состоящую из знака доллара и квадратных скобок (`${ operation }`):

```
$ var1=$((1 + 5))
$ echo $var1
6
$ var2 = `${var1} * 2`
$ echo $var2
12
$
```

Благодаря использованию квадратных скобок проведение математических вычислений в командном интерпретаторе становится намного проще, чем при использовании команды `expr`. Тот же метод применим и в сценариях командного интерпретатора:

```
$ cat test7
#!/bin/bash
var1=100
var2=50
var3=45
var4=$((var1 * (var2 - var3)))
echo The final result is $var4
$
```

Выполнение этого сценария приводит к получению следующего вывода:

```
$ chmod u+x test7
$ ./test7
The final result is 500
$
```

К тому же заслуживает внимания то, что при использовании метода на основе квадратных скобок для вычисления математических выражений не приходится беспокоиться о том, что знак умножения или какие-либо другие символы будут неправильно обработаны командным интерпретатором. Командный интерпретатор определяет, что данный символ не является специальным, поскольку он заключен в квадратные скобки.

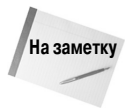
Тем не менее при осуществлении математических вычислений в сценарии командного интерпретатора `bash` приходится учитывать одно серьезное ограничение. Рассмотрим следующий пример:

```
$ cat test8
#!/bin/bash
var1=100
var2=45
var3=$((var1 / $var2))
echo The final result is $var3
$
```

Теперь попытаемся выполнить эти команды и посмотрим, что происходит:

```
$ chmod u+x test8
$ ./test8
The final result is 2
$
```

Математические операторы в командном интерпретаторе `bash` поддерживают только целочисленную арифметику. Это — серьезнейшее ограничение, если приходится выполнять хотя бы какие-то практически значимые математические вычисления.



Полная поддержка арифметических операций с плавающей запятой предусмотрена в командном интерпретаторе `z` (`zsh`). Если должен быть составлен такой сценарий командного интерпретатора, в котором требуется выполнять вычисления с плавающей запятой, то можно рассмотреть возможность использования командного интерпретатора `z` (рассматриваемого в главе 22).

Способы выполнения вычислений с плавающей запятой

Предусмотрено несколько решений, позволяющих преодолеть ограничение командного интерпретатора `bash`, связанное с поддержкой только целых чисел. Наиболее широко применяемое решение предусматривает использование встроенного калькулятора `bash`, называемого `bc`.

Основные сведения о калькуляторе `bc`

Применяемый в интерпретаторе `bash` калькулятор фактически поддерживает язык программирования, который позволяет вводить выражения для работы с плавающей запятой в командной строке, а затем интерпретировать эти выражения, вычислять их и возвращать результаты. Калькулятор `bash` распознает следующие синтаксические элементы:

- числа (целые и с плавающей запятой);
- переменные (простые переменные и массивы);
- комментарии (строки, начинающиеся со знака диэза, или парные скобки языка `C`, `/* */`);
- выражения;
- программные инструкции (наподобие инструкций `if-then`);
- функции.

Доступ к калькулятору `bash` можно получить из приглашения командного интерпретатора с помощью команды `bc`:

```
$ bc
bc 1.06.95
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software
```

```

Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type 'warranty'.
12 * 5.4
64.8
3.156 * (3 + 5)
25.248
quit
$

```

Этот пример начинается с ввода выражения $12 * 5.4$. Калькулятор `bash` возвращает ответ. После этого вычисляется каждое последующее выражение, заданное для калькулятора, и отображается результат. Для выхода из калькулятора `bash` необходимо ввести команду `quit`.

Управление функционированием средств арифметики с плавающей запятой осуществляется с помощью встроенной переменной `scale`. В качестве значения этой переменной следует задать требуемое количество десятичных позиций, которые должны отображаться в ответах, поскольку в противном случае может оказаться, что полученные данные не соответствуют требованиям:

```

$ bc -q
3.44 / 5
0
scale=4
3.44 / 5
.6880
quit
$

```

По умолчанию переменная `scale` имеет значение нуль. До тех пор пока значение `scale` не установлено, калькулятор `bash` выдает ответы с нулевым количеством десятичных позиций. После задания значения переменной `scale`, равного четырем, калькулятор `bash` отображает ответы с четырьмя десятичными позициями. Для подавления крупной заставки калькулятора `bash` с приветственным сообщением применяется параметр командной строки `-q`.

Кроме обычных числовых значений, калькулятор `bash` позволяет также пользоваться переменными:

```

$ bc -q
var1=10
var1 * 4
40
var2 = var1 / 5
print var2
2
quit
$

```

После определения значения переменной эту переменную можно использовать на протяжении всего сеанса работы с калькулятором `bash`. С помощью инструкции `print` можно вывести значения переменных и чисел на экран.

Использование калькулятора `bc` в сценариях

Настоящий раздел посвящен использованию калькулятора `bash` при выполнении арифметических вычислений с плавающей запятой в сценариях командного интерпретатора. Напомним,

что выше был описан удобный в работе символ обратной одинарной кавычки. Следует добавить к сказанному, что символ обратной одинарной кавычки можно использовать для выполнения команды `bc` и присваивания ее вывода переменной! Для этого предназначен следующий основной формат:

```
variable='echo "options; expression" | bc`
```

Первая часть, `options`, позволяет задавать переменные. Если необходимо задать несколько переменных, их следует отделить друг от друга точкой с запятой. Параметр `expression` определяет математическое выражение, которое должно быть вычислено с помощью `bc`. Ниже приведен краткий пример применения такой конструкции в сценарии.

```
$ cat test9
#!/bin/bash
var1='echo " scale=4; 3.44 / 5" | bc`
echo The answer is $var1
$
```

В этом примере задается значение переменной `scale`, равное четырем десятичным позициям, а затем задается конкретное выражение для вычисления. Вызов этого сценария приводит к получению следующего вывода:

```
$ chmod u+x test9
$ ./test9
The answer is .6880
$
```

Теперь мы знаем, что в наших руках есть превосходный инструмент! В качестве значения выражения можно не ограничиваться применением только чисел. Можно также использовать переменные, определенные в сценарии командного интерпретатора:

```
$ cat test10
#!/bin/bash
var1=100
var2=45
var3='echo "scale=4; $var1 / $var2" | bc`
echo The answer for this is $var3
$
```

В этом сценарии определены две переменные, которые используются в выражении, передаваемом в команду `bc`. Не забывайте задавать знаки доллара для указания на то, что должны применяться значения переменных, а не сами переменные. Этот сценарий формирует следующий вывод:

```
$ ./test10
The answer for this is 2.2222
$
```

Разумеется, после присваивания значения переменной эту переменную можно использовать еще в одном вычислении:

```
$ cat test11
#!/bin/bash
var1=20
var2=3.14159
var3='echo "scale=4; $var1 * $var1" | bc`
```

```
var4='echo "scale=4; $var3 * $var2" | bc'
echo The final result is $var4
$
```

Этот метод работает вполне приемлемо в коротких вычислениях, но иногда приходится выполнять с числовыми данными более сложные манипуляции. Если предстоящую задачу нельзя свести лишь к нескольким вычислениям, то возникает путаница при попытке перечислить одно за другим несколько выражений в одной и той же командной строке.

Для этой проблемы предусмотрено решение. Команда `bc` распознает перенаправление ввода, что позволяет перенаправить в команду `bc` для обработки целый файл. Однако и это может привести к путанице, поскольку приходится сохранять обрабатываемые выражения в файле.

Наилучший метод состоит в использовании перенаправления ввода через буфер, что позволяет передавать данные в команду непосредственно из командной строки. В сценарии командного интерпретатора вывод присваивается переменной:

```
variable='bc << EOF
options
statements
expressions
EOF
\
```

Текстовая строка `EOF` указывает начало и конец данных, перенаправляемых через буфер. Следует помнить, что необходимость в использовании символов обратной одинарной кавычки все еще остается, поскольку вывод команды `bc` должен быть присвоен переменной.

После этого можно поместить все отдельные элементы вызова калькулятора `bash` в отдельных строках файла сценария. Ниже приведен пример использования этого метода в сценарии.

```
$ cat test12
#!/bin/bash

var1=10.46
var2=43.67
var3=33.2
var4=71

var5='bc << EOF
scale = 4
a1 = ( $var1 * $var2)
b1 = ($var3 * $var4)
a1 + b1
EOF
\

echo The final answer for this mess is $var5
$
```

При этом благодаря возможности размещения каждого параметра и выражения на отдельной строке сценарий становится более простым и удобным для чтения и сопровождения. Строка `EOF` указывает начало и конец данных, перенаправляемых в команду `bc`. Безусловно, необходимо использовать символы обратной одинарной кавычки для задания команды, которую следует присвоить переменной.

В этом примере заслуживает также внимания то, что в нем применяется возможность присваивать значения переменным в ходе работы с калькулятором `bash`. Важно помнить, что все

переменные, которые определены в процессе использования калькулятора `bash`, остаются допустимыми только на протяжении сеанса работы с калькулятором `bash` и не могут использоваться в самом сценарии командного интерпретатора.

Выход из сценария

Во всех приведенных выше примерах сценариев не уделялось внимание корректному завершению работы сценария. В них после выполнения последней команды просто происходил выход из сценария. Однако предусмотрен более правильный способ завершения работы.

В каждой команде, выполняемой в командном интерпретаторе, используется так называемый *статус выхода* для передачи командному интерпретатору указания, чем закончилась обработка этой команды. Статус выхода — это целочисленное значение от 0 до 255, передаваемое командой командному интерпретатору после завершения своего выполнения. Это значение можно перехватывать и использовать в своих сценариях.

Проверка статуса выхода

В Linux предусмотрена специальная переменная `$?` , в которой хранится значение статуса выхода последней выполненной команды. Значение переменной `$?` необходимо рассматривать или использовать сразу после выполнения команды, статус выхода которой должен быть проверен. В следующем примере показано, как изменяется значение статуса выхода последней команды, выполненной командным интерпретатором:

```
$ date
Sat Jan 15 10:01:30 EDT 2011
$ echo $?
0
$
```

В соответствии с принятым соглашением статус выхода успешно завершенной команды равен нулю. Если команда завершается с ошибкой, то статус выхода принимает значение положительного целого числа:

```
$ asdfg
-bash: asdfg: command not found
$ echo $?
127
$
```

В данном случае недопустимая команда возвратила статус выхода, равный 127. В отношении того, какие значения должен принимать статус выхода при возникновении тех или иных ошибок, в Linux практически нет какого-либо общепринятого соглашения. Однако при этом можно руководствоваться некоторыми рекомендациями, как показано в табл. 10.2.

Таблица 10.2. Коды статуса выхода Linux

Код	Описание
0	Успешное завершение команды
1	Общая неизвестная ошибка
2	Неправильное употребление команды командного интерпретатора

Код	Описание
126	Команда не может быть выполнена
127	Команда не найдена
128	Недопустимый аргумент выхода
128+x	Неисправимая ошибка с сигналом x системы Linux
130	Выполнение команды завершено нажатием клавиш <Ctrl+C>
255	Значение статуса выхода не соответствует допустимому диапазону

Значение статуса выхода, равное 126, указывает, что для пользователя не заданы надлежащие разрешения на выполнение команды:

```
$ ./myprog.c
-bash: ./myprog.c: Permission denied
$ echo $?
126
$
```

Еще одна широко распространенная ошибка, с которой сталкиваются практически все, состоит в использовании недопустимого параметра для вызова команды:

```
$ date %t
date: invalid date '%t'
$ echo $?
1
$
```

При этом формируется общий код статуса выхода, равный 1, который указывает, что в команде произошла неизвестная ошибка.

Команда exit

По умолчанию выход из сценария командного интерпретатора происходит с установкой для него статуса выхода последней команды в этом сценарии:

```
$ ./test6
The result is 2
$ echo $?
0
$
```

Предусмотрена возможность вместо этого сформировать и вернуть код статуса выхода, выбранный самим программистом. Для указания статуса выхода при завершении сценария применяется команда `exit`:

```
$ cat test13
#!/bin/bash
# проверка статуса выхода
var1=10
var2=30
var3=$(( $var1 + var2 ))
echo The answer is $var3
```

```
exit 5
$
```

При проверке статуса выхода сценария будет получено значение, заданное в качестве параметра команды `exit`:

```
$ chmod u+x test13
$ ./test13
The answer is 40
$ echo $?
5
$
```

При определении параметра команды `exit` можно также использовать переменные:

```
$ cat test14
#!/bin/bash
# проверка статуса выхода
var1=10
var2=30
var3=$(( $var1 + var2 ))
exit $var3
$
```

После вызова данной команды на выполнение сформируется следующий статус выхода:

```
$ chmod u+x test14
$ ./test14
$ echo $?
40
$
```

Однако при использовании этого средства необходимо соблюдать осторожность, поскольку значения кодов статуса выхода не могут превышать 255. Рассмотрим, что происходит в следующем примере:

```
$ cat test14b
#!/bin/bash
# проверка статуса выхода
var1=10
var2=30
var3=$(( $var1 * var2 ))
echo The value is $var3
exit $var3
$
```

После вызова на выполнение данного сценария будет получен следующий результат:

```
$ ./test14b
The value is 300
$ echo $?
44
$
```

Значение кода статуса выхода усечено так, чтобы он не выходил за пределы диапазона от 0 до 255. В командном интерпретаторе для осуществления этого действия применяется арифметика вычисления остатка от деления (вычисления по модулю). *Модуль* значения — это остаток

от деления. Полученное число представляет собой остаток после деления заданного числа на 256. В случае применения значения 300 (в качестве значения результата) остаток составляет 44, и именно это значение отображается в качестве кода статуса выхода.

В главе 11 будет показано, как использовать инструкцию `if-then` для проверки статуса ошибки, возвращенного командой, для определения того, была ли команда выполнена успешно.

Резюме

Сценарии командного интерпретатора `bash` позволяют соединять отдельные команды в цепочки команд, которые образуют сценарий. Наиболее простой способ создания сценария состоит в том, что в командной строке размещаются сразу несколько команд, разделенных точками с запятой. Командный интерпретатор последовательно выполняет одну команду за другой, отображая вывод каждой команды на мониторе.

Предусмотрена также возможность создавать файлы сценариев командного интерпретатора, помещая в один файл несколько команд, которые должны быть выполнены в командном интерпретаторе. В файле сценария командного интерпретатора должно быть указано, какой командный интерпретатор следует применять для выполнения сценария. Для этого предназначена первая строка файла сценария, в которой должен быть указан символ `#!`, а затем — полный путь к командному интерпретатору.

В сценарии командного интерпретатора можно ссылаться на значения переменных среды, задавая знак доллара перед именем переменной. Можно также определять собственные переменные для использования в сценарии, присваивать им значения и даже результаты выполнения команды (с использованием символа обратной одинарной кавычки). Чтобы воспользоваться значением переменной в сценарии, необходимо поместить знак доллара перед именем переменной.

В командном интерпретаторе `bash` предусмотрено стандартное средство, позволяющее перенаправлять и ввод, и вывод команд. Вывод любой команды можно перенаправить так, чтобы он вместо отображения на мониторе был записан в файл. Для этого используется знак “больше”, за которым следует имя файла, предназначенного для перехвата вывода. Кроме того, можно присоединять выходные данные к существующему файлу с использованием двух знаков “больше”. Знак “меньше” служит для перенаправления ввода в команду. Предусмотрена возможность перенаправлять ввод из файла в любую команду.

Команда канала `Linux`, обозначаемая символом прерванной вертикальной черты, позволяет перенаправлять выход одной команды непосредственно на вход другой команды. Система `Linux` выполняет обе команды одновременно, передавая выходные данные первой команды на вход второй команды без использования каких-либо файлов перенаправления.

В командном интерпретаторе `bash` предусмотрен целый ряд способов выполнения математических операций в сценариях командного интерпретатора. Один из простых способов выполнения операций целочисленной арифметики состоит в использовании команды `expr`. В командном интерпретаторе `bash` можно также выполнять основные математические вычисления, заключая математические выражения в квадратные скобки, которым предшествует знак доллара. Для выполнения вычислений с помощью операций арифметики с плавающей запятой необходимо использовать команды калькулятора `bc`, перенаправляя ввод из встроенных данных и сохраняя вывод в пользовательской переменной.

Наконец, в настоящей главе рассматривалась тема использования статуса выхода в сценариях командного интерпретатора. Каждая команда, выполняемая в командном интерпретаторе, вырабатывает статус выхода. Статус выхода — это целочисленное значение от 0 до 255, кото-

рое указывает, завершена ли команда успешно, а если нет, сообщает возможную причину неудачи. Статус выхода 0 свидетельствует о том, что команда завершена успешно. Чтобы задать конкретное значение статуса выхода, которое станет доступным после завершения сценария, можно воспользоваться командой `exit` в сценарии командного интерпретатора.

В настоящей главе рассматривались сценарии командного интерпретатора, в которых обработка команд происходила последовательно, от одной к другой. В следующей главе будет показано, как можно использовать некоторые логические средства управления потоком данных для изменения последовательности выполнения команд в сценарии.

Использование структурирован- ных команд

В сценариях командного интерпретатора, представленных в главе 10, командный интерпретатор обрабатывал каждую отдельную команду в сценарии командного интерпретатора в том порядке, в котором команды представлены в этом сценарии. Это вполне приемлемо при осуществлении последовательных действий, когда требуется обработать все команды в надлежащем порядке. Однако такая организация работы приемлема не во всех программах.

Для многих программ требуется обеспечить в том или ином виде логическое управление потоком данных, передаваемых между командами в сценарии. Это означает, что командный интерпретатор должен выполнять определенные команды при одной совокупности условий, но может также переходить к обработке других команд, если условия окажутся иными. Предусмотрен целый ряд команд, которые позволяют пропускать в сценарии ряд команд или обрабатывать команды в цикле после проверки условий, определяемых значениями переменных или результатами других команд. В целом такие команды именуются *структурированными командами*.

Структурированные команды позволяют изменять ход выполнения инструкций в программе, выполняя одни команды при одних условиях и пропуская другие при других условиях. Количество структурированных команд, предусмотренных в командном интерпретаторе `bash`, весьма велико, поэтому будем рассматривать их отдельно. В настоящей главе рассматривается инструкция `if-then`.

ГЛАВА

11

В этой главе...

Работа с инструкцией `if-then`

Инструкция `if-then-else`

Уровень вложенности
инструкций `if`

Команда `test`

Проверка с помощью
составных условий

Дополнительные средства
инструкции `if-then`

Команда `case`

Резюме

Работа с инструкцией if-then

К наиболее важным типам структурированных команд относится инструкция if-then, имеющая следующий формат:

```
if command
then
    commands
fi
```

Для тех, кто использовал инструкцию if-then в других языках программирования, этот формат может показаться довольно сложным. Во многих других языках программирования объектом, расположенным вслед за инструкцией if, является выражение, значение которого вычисляется для определения того, равно ли оно TRUE или FALSE. Инструкция if командного интерпретатора bash функционирует иначе.

В инструкции if командного интерпретатора bash выполняется команда *command*, заданная в строке if. Если статус выхода команды (см. главу 10) равен нулю (команда завершена успешно), выполняются команды *commands*, перечисленные в разделе then. Если статус выхода команды имеет значение, отличное от нуля, команды в разделе then не выполняются и командный интерпретатор bash переходит к следующей команде в сценарии.

Ниже приведен простой пример, позволяющий пояснить сказанное.

```
$ cat test1
#!/bin/bash
# проверка инструкции if
if date
then
    echo "it worked"
fi
```

В этом сценарии в строке if используется команда date. После успешного завершения команды инструкция echo должна отобразить текстовую строку. Вызов этого сценария на выполнение из командной строки приводит к получению следующих результатов:

```
$ ./test1
Sat Jan 23 14:09:24 EDT 2011
it worked
$
```

Командный интерпретатор выполнил команду date, заданную в строке if. Статус выхода этой команды был равен нулю, поэтому была также выполнена инструкция echo, приведенная в разделе then.

Рассмотрим еще один пример:

```
$ cat test2
#!/bin/bash
# проверка неправильно заданной команды
if asdfg
then
    echo "it did not work"
fi
echo "we are outside of the if statement"
$
```

```
$ ./test2
./test2: line 3: asdfg: command not found
we are outside of the if statement
$
```

В этом примере была преднамеренно использована такая команда, выполнение которой в строке инструкции `if` должно окончиться неудачей. Поскольку это неправильная команда, в ней вырабатывается статус выхода, отличный от нуля, и командный интерпретатор `bash` пропускает инструкцию `echo` в разделе `then`. Следует также отметить, что в выводе сценария все равно появляется сообщение об ошибке, сформированное при выполнении команды в инструкции `if`. Но иногда появление такого сообщения в выводе нежелательно. В главе 14 описано, как можно это предотвратить.

В разделе `then` не обязательно должна находиться только одна команда. В этот раздел можно включить целый ряд команд, как и в остальной части сценария командного интерпретатора. В командном интерпретаторе `bash` такая последовательность команд рассматривается как *блок*, и все команды выполняются, если команда в строке инструкции `if` возвращает нулевое значение статуса выхода, или все пропускаются при получении из команды ненулевого статуса выхода:

```
$ cat test3
#!/bin/bash
# проверка с выполнением нескольких команд в разделе then
testuser=rich
if grep $testuser /etc/passwd
then
    echo The bash files for user $testuser are:
    ls -a /home/$testuser/.b*
fi
```

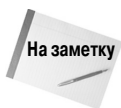
В строке инструкции `if` используется команда `grep` для определения с помощью файла `/etc/passwd` того, существует ли данное конкретное имя пользователя в системе в настоящее время. Если пользователь с указанным регистрационным именем имеется, сценарий отображает некоторый текст, а затем выводит с помощью `bash` список файлов в каталоге `HOME` пользователя:

```
$ ./test3
rich:x:500:500:Rich Blum:/home/rich:/bin/bash
The files for user rich are:
/home/rich/.bash_history /home/rich/.bash_profile
/home/rich/.bash_logout /home/rich/.bashrc
$
```

Но если в качестве значения переменной `testuser` будет задано имя пользователя, не существующего в системе, то ничего не произойдет:

```
$ ./test3
$
```

Но на этом возможности инструкции `if-then` не исчерпываются. Было бы удобно иметь возможность отобразить небольшое сообщение, указывающее, что пользователь с таким именем не был найден в системе. И действительно, можно воспользоваться еще одним средством инструкции `if-then`.



Например, в некоторых сценариях встречается такая альтернативная форма инструкции `if-then`:

```
if command; then
    commands
fi
```

Поместив точку с запятой после выполняемой команды, можно поместить инструкцию `then` на той же строке, что в большей степени напоминает организацию работы с инструкцией `if-then` в некоторых других языках программирования.

Инструкция `if-then-else`

В инструкции `if-then` предусмотрена возможность лишь определить, успешно ли была выполнена некоторая команда. Если же команда возвращает ненулевой код статуса выхода, то командный интерпретатор `bash` просто переходит к следующей команде в сценарии. В такой ситуации иногда было бы удобно иметь возможность выполнить альтернативный набор команд. Именно для этого предназначена инструкция `if-then-else`.

Инструкция `if-then-else` предоставляет возможность задать еще одну группу команд в инструкции:

```
if command
then
    commands
else
    commands
fi
```

Если команда в строке инструкции `if` возвращает нулевой код статуса выхода, выполняются команды, перечисленные в разделе `then`, как и в обычной инструкции `if-then`. Если же команда в строке инструкции `if` возвращает ненулевой код статуса выхода, командный интерпретатор `bash` выполняет команды в разделе `else`.

Теперь можно внести изменения в проверочный сценарий, который выглядит следующим образом:

```
$ cat test4
#!/bin/bash
# проверка с использованием раздела else
testuser=badtest
if grep $testuser /etc/passwd
then
    echo The files for user $testuser are:
    ls -a /home/$testuser/.b*
else
    echo "The user name $testuser does not exist on this system"
fi
$
$ ./test4
The user name badtest does not exist on this system
$
```

Такая конструкция является более удобной в использовании. Раздел `else`, как и раздел `then`, может включать несколько команд. Конец раздела `else` обозначается инструкцией `fi`.

Уровень вложенности инструкций if

Иногда в коде сценария необходимо проверить одно за другим несколько условий. Вместо того чтобы записывать отдельные инструкции `if-then`, можно воспользоваться альтернативным вариантом раздела `else`, который обозначается как `elif`.

Раздел `elif` продолжает раздел `else`, задавая еще одну инструкцию `if-then`:

```
if command1
then
    commands
elif command2
then
    more commands
fi
```

В строке инструкции `elif` содержится другая команда, подлежащая выполнению, но имеющая то же назначение, что и в строке исходной инструкции `if`. Если код статуса выхода команды `elif` равен нулю, командный интерпретатор `bash` выполняет команды во втором разделе инструкции `then`.

Инструкции `elif` можно продолжать связывать в цепочки, что иногда приводит к созданию огромных конгломератов инструкций `if-then-elif`:

```
if command1
then
    command set 1
elif command2
then
    command set 2
elif command3
then
    command set 3
elif command4
then
    command set 4
fi
```

Каждый блок команд выполняется в зависимости от того, возвращает ли команда нулевой или ненулевой код статуса выхода. Следует помнить, что командный интерпретатор `bash` выполняет инструкции `if` последовательно и только возврат первой командой нулевого статуса выхода приводит к выполнению раздела `then`. Ниже, в разделе “Команда `case`”, будет показано, как использовать команду `case`, вместо того чтобы нагромождать многочисленные инструкции `if-then`.

Команда test

До сих пор в данной главе рассматривалось применение в строке инструкции `if` лишь обычных команд командного интерпретатора. Может возникнуть вопрос, обладает ли инструкция `if-then` командного интерпретатора `bash` способностью оценивать какие-то другие условия, кроме кода статуса выхода команды.

Ответ является отрицательным — не может. Однако в составе командного интерпретатора `bash` имеется удобная программа, позволяющая проверять другие условия и предназначенная для использования в инструкции `if-then`.

Один из способов проверки различных условий в инструкции `if-then` как раз и состоит в использовании команды `test`. Если проверка условия, заданного в команде `test`, приводит к получению истинного значения, то команда `test` выполняет возврат с нулевым кодом статуса выхода. Таким образом, выполнение инструкции `if-then` в большей степени напоминает действие инструкции `if-then` в других языках программирования. Если проверка условия приводит к получению ложного значения, команда `test` возвращает 1, поэтому команды в разделе `then` инструкции `if-then` пропускаются.

Команда `test` имеет довольно простой формат:

```
test condition
```

Условие *condition* — это ряд параметров и значений, обрабатываемых командой `test`. При использовании в инструкции `if-then` команда `test` выглядит следующим образом:

```
if test condition
then
    commands
fi
```

В командном интерпретаторе `bash` предусмотрен альтернативный способ включения команды `test` в инструкцию `if-then`:

```
if [ condition ]
then
    commands
fi
```

Квадратные скобки определяют условие, используемое в команде `test`. При использовании этой команды необходимо соблюдать осторожность; должен быть *обязательно* введен пробел после первой квадратной скобки и перед последней квадратной скобкой, поскольку в противном случае будет получено сообщение об ошибке.

Команда `test` может обрабатывать условия трех типов:

- сравнение чисел;
- сравнение строк;
- сравнение файлов.

В следующем разделе описано, как использовать каждый из этих типов проверок в инструкциях `if-then`.

Сравнение чисел

Наиболее широко применяемый метод работы с командой `test` состоит в использовании этой команды для сравнения двух числовых значений. В табл. 11.1 приведен список параметров условий, которые могут служить для проверки двух значений.

Условия проверки числовых значений могут использоваться для работы и с числами, и с переменными. Ниже приведен пример применения такой проверки.

Таблица 11.1. Операции сравнения чисел в команде test

Сравнение	Описание
n1 -eq n2	Проверка того, что n1 равно n2
n1 -ge n2	Проверка того, что n1 больше или равно n2
n1 -gt n2	Проверка того, что n1 больше n2
n1 -le n2	Проверка того, что n1 меньше или равно n2
n1 -lt n2	Проверка того, что n1 меньше n2
n1 -ne n2	Проверка того, что n1 не равно n2

```
$ cat test5
#!/bin/bash
# использование операторов проверки числовых значений
val1=10
val2=11
```

```
if [ $val1 -gt 5 ]
then
    echo "The test value $val1 is greater than 5"
fi
```

```
if [ $val1 -eq $val2 ]
then
    echo "The values are equal"
else
    echo "The values are different"
fi
```

Первое проверяемое условие:

```
if [ $val1 -gt 5 ]
```

позволяет определить, не превышает ли значение переменной val1 величину 5. Второе проверяемое условие:

```
if [ $val1 -eq $val2 ]
```

позволяет установить, равно ли значение переменной val1 значению переменной val2. Вызовем сценарий на выполнение и ознакомимся с результатами:

```
$ ./test5
The test value 10 is greater than 5
The values are different
$
```

Проверка обоих числовых значений привела к получению ожидаемых результатов.

Тем не менее при проверке числовых значений необходимо учитывать определенные ограничения. Рассмотрим следующий сценарий:

```
$ cat test6
#!/bin/bash
# проверка чисел с плавающей точкой
val1=`echo "scale=4; 10 / 3 " | bc`
```

```

echo "The test value is $vall"
if [ $vall -gt 3 ]
then
    echo "The result is larger than 3"
fi
$
$ ./test6
The test value is 3.3333
./test6: line 5: [: 3.3333: integer expression expected
$

```

В этом примере калькулятор `bash` используется для выработки значения с плавающей запятой, сохраняемого в переменной `vall`. Затем в нем применяется команда `test` для вычисления требуемого значения. Очевидно, что в данном случае что-то происходит вопреки ожиданиям.

В главе 10 было показано, как обеспечить в командном интерпретаторе `bash` обработку значений с плавающей запятой, но при выполнении этого сценария обнаруживается еще одна проблема. В команде `test` окончилась неудачей попытка обработать значение с плавающей запятой, которое хранилось в переменной `vall`.

Напомним, что единственным типом числовых данных, которые может поддерживать командный интерпретатор `bash`, являются целые числа. При использовании калькулятора `bash` просто применяется обходной маневр с учетом этой особенности командного интерпретатора, который заключается в том, что значение с плавающей точкой сохраняется в переменной в виде строкового значения. Такая организация функционирования является вполне приемлемой, если требуется лишь отобразить результат с использованием инструкции `echo`, но не позволяет обеспечить применение функций для работы с числами, наподобие тех, что применяются при проверке условия, представленного числовыми данными. Из этого следует вывод, что возможность непосредственно использовать значения с плавающей запятой в команде `test` отсутствует.

Сравнение строк

Команда `test` позволяет также выполнять сравнения строковых значений. Но, как будет показано ниже, иногда для сравнения строк приходится использовать сложные конструкции. Операторы сравнения, которые могут использоваться для обработки двух строковых значений, приведены в табл. 11.2.

Таблица 11.2. Операторы сравнения строк в команде `test`

Сравнение	Описание
<code>str1 = str2</code>	Проверка того, что строка <code>str1</code> является такой же, как и строка <code>str2</code>
<code>str1 != str2</code>	Проверка того, что строка <code>str1</code> не совпадает со строкой <code>str2</code>
<code>str1 < str2</code>	Проверка того, что строка <code>str1</code> меньше строки <code>str2</code>
<code>str1 > str2</code>	Проверка того, что строка <code>str1</code> больше строки <code>str2</code>
<code>-n str1</code>	Проверка того, что строка <code>str1</code> имеет длину больше нуля
<code>-z str1</code>	Проверка того, что строка <code>str1</code> имеет длину, равную нулю

Различные применимые операции сравнения строк рассматриваются в следующих разделах.

Сравнение строк на равенство

Применительно к строкам проверка на равенство и неравенство представляет собой вполне очевидный процесс. Определить, являются ли два строковых значения одинаковыми или неодинаковыми, не составляет особого труда:

```
$cat test7
#!/bin/bash
# проверка строк на равенство
testuser=rich

if [ $USER = $testuser ]
then
    echo "Welcome $testuser"
fi
$
$ ./test7
Welcome rich
$
```

Кроме того, сравнение строк на неравенство также позволяет определить, имеют ли две строки одно и то же значение или нет:

```
$ cat test8
#!/bin/bash
# проверка строк на равенство
testuser=baduser

if [ $USER != $testuser ]
then
    echo "This is not $testuser"
else
    echo "Welcome $testuser"
fi
$
$ ./test8
This is not baduser
$
```

При сравнении строк с помощью команды `test` учитываются все знаки препинания и различия в применении прописных и строчных букв в ходе того, как происходит проверка строк на равенство.

Упорядочение строк

Задача усложняется при осуществлении попытки определить, должна ли одна строка рассматриваться как предшествующая другой строке в последовательности сортировки по возрастанию. Программисты на языке командного интерпретатора часто сталкиваются со следующими двумя коварными проблемами при попытке использовать операции “больше” или “меньше” применительно к строкам в команде `test`.

- Во-первых, символы “больше” и “меньше” должны быть экранированы, поскольку в противном случае в командном интерпретаторе эти символы будут рассматриваться как символы перенаправления, а строковые значения — как имена файлов.

- Во-вторых, строки, упорядочиваемые с помощью операторов “больше” и “меньше”, располагаются иначе, чем отсортированные с помощью команды `sort`.

Первая проблема может приводить к возникновению существенного недостатка, который часто с большим трудом поддается диагностированию при программировании сценариев. Ниже приведен типичный пример того, с чем иногда сталкиваются неопытные разработчики сценариев на языке командного интерпретатора:

```
$ cat badtest
#!/bin/bash
# неправильное использование операторов сравнения строк

val1=baseball
val2=hockey

if [ $val1 > $val2 ]
then
    echo "$val1 is greater than $val2"
else
    echo "$val1 is less than $val2"
fi
$
$ ./badtest
baseball is greater than hockey
$ ls -l hockey
-rw-r--r--  1 rich      rich              0 Sep 30 19:08 hockey
$
```

Символ “больше” был задан в сценарии в непосредственном виде, и одно лишь это привело к получению неправильных результатов, хотя и не были сформированы какие-либо ошибки. При обработке этого сценария символ “больше” интерпретировался как перенаправление вывода. Поэтому был создан файл с именем `hockey`. Операция перенаправления была выполнена успешно, поэтому команда `test` возвратила нулевой код статуса выхода, в связи с чем инструкция `if` была обработана так, как если бы проверка действительно завершилась успешно!

Чтобы устранить эту ошибку, требуется должным образом экранировать символ “больше”:

```
$ cat test9
#!/bin/bash
# неправильное использование операторов сравнения строк

val1=baseball
val2=hockey

if [ $val1 \> $val2 ]
then
    echo "$val1 is greater than $val2"
else
    echo "$val1 is less than $val2"
fi
$
$ ./test9
baseball is less than hockey
$
```

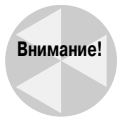
Теперь ответ в большей степени соответствует тому, что следовало ожидать после сравнения строк.

Вторая проблема немного сложнее поддается распознаванию, и с ней, возможно, даже не придется столкнуться, если не приходится учитывать различия между прописными и строчными буквами. В команде `sort` сравнение прописных и строчных букв происходит иначе, чем в команде `test`. Рассмотрим эту особенность в сценарии:

```
$ cat test9b
#!/bin/bash
# проверка последовательности сортировки строк
val1=Testing
val2=testing

if [ $val1 \> $val2 ]
then
    echo "$val1 is greater than $val2"
else
    echo "$val1 is less than $val2"
fi
$
$ ./test9b
Testing is less than testing
$ sort testfile
testing
Testing
$
```

В команде `test` строки с прописными буквами рассматривались как предшествующие строкам со строчными буквами. Если же такие строки будут помещены в файл и к ним применена команда `sort`, то строки со строчными буквами появятся раньше строк с прописными буквами. Это связано с тем, что в той и другой команде используются разные способы упорядочения строк. В команде `test` за основу взято расположение символов по стандарту ASCII и рассматривается числовое значение каждого символа ASCII для определения порядка сортировки. А в команде `sort` используется порядок сортировки, определенный для параметров языка региональной установки системы. Для английского языка настройки региональной установки указывают, что в последовательности сортировки строчные буквы должны находиться перед прописными буквами.



Следует также отметить, что в команде `test` используются стандартные математические символы сравнения при сравнении строк и текстовые коды при сравнении чисел. Это — еще один нюанс, который сбивает с толку многих программистов. Если для сравнения числовых значений используются математические символы сравнения, то командный интерпретатор рассматривает эти числовые значения как строковые, а это не всегда приводит к получению правильных результатов.

Размер строки

Сравнения с помощью параметров `-n` и `-z` являются применимыми, если требуется определить, содержит ли переменная данные или нет:

```
$ cat test10
#!/bin/bash
# проверка строк с учетом длины
```

```

val1=testing
val2=''

if [ -n $val1 ]
then
    echo "The string '$val1' is not empty"
else
    echo "The string '$val1' is empty"
fi

if [ -z $val2 ]
then
    echo "The string '$val2' is empty"
else
    echo "The string '$val2' is not empty"
fi

if [ -z $val3 ]
then
    echo "The string '$val3' is empty"
else
    echo "The string '$val3' is not empty"
fi
$
$ ./test10
The string 'testing' is not empty
The string '' is empty
The string '' is empty
$

```

В рассматриваемом примере создаются две строковые переменные. Переменная `val1` содержит непустую строку, а переменная `val2` создана как пустая строка. Операции сравнения, которые приведены в следующем сценарии:

```
if [ -n $val1 ]
```

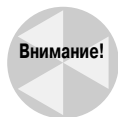
позволяют определить, имеет ли переменная `val1` ненулевую длину, и в случае положительного ответа обработать раздел `then`:

```
if [ -z $var2 ]
```

определить, имеет ли переменная `val2` нулевую длину, и в случае положительного ответа также обработать раздел `then`:

```
if [ -z $val3 ]
```

чтобы определить, имеет ли переменная `val3` нулевую длину. Последняя переменная еще не была определена в сценарии командного интерпретатора, поэтому в полученном результате указано, что заданная в ней строка имеет нулевую длину, несмотря на то, что сама переменная упоминается в сценарии впервые.



Не учитывая наличие пустых и неинициализированных переменных, можно столкнуться с катастрофическими ошибками при проведении проверок в сценариях командного интерпретатора. Если нет полной уверенности в том, каковым является значение пере-

менной и задано ли оно вообще, всегда лучше проверить, содержит ли переменная значение, с помощью параметра `-n` или `-z` и лишь затем использовать ее в числовом сравнении или в сравнении строк.

Сравнение файлов

Рассматриваемая в этом разделе категория сравнений при проверке данных, по всей видимости, относится к наиболее мощным и широко применяемым операциям сравнения в написании сценариев командного интерпретатора. Команда `test` позволяет проверять статус файлов и каталогов в файловой системе Linux. Применяемые при этом операторы сравнения перечислены в табл. 11.3.

Таблица 11.3. Сравнение файлов в команде <code>test</code>	
Сравнение	Описание
<code>-d file</code>	Проверка того, что <code>file</code> существует и является каталогом
<code>-e file</code>	Проверка того, что <code>file</code> существует
<code>-f file</code>	Проверка того, что <code>file</code> существует и является файлом
<code>-r file</code>	Проверка того, что <code>file</code> существует и предназначен для чтения
<code>-s file</code>	Проверка того, что <code>file</code> существует и не пуст
<code>-w file</code>	Проверка того, что <code>file</code> существует и предназначен для записи
<code>-x file</code>	Проверка того, что <code>file</code> существует и предназначен для выполнения
<code>-O file</code>	Проверка того, что <code>file</code> существует и принадлежит текущему пользователю
<code>-G file</code>	Проверка того, что <code>file</code> существует и его заданная по умолчанию группа является такой же, как и у текущего пользователя
<code>file1 -nt file2</code>	Проверка того, что <code>file1</code> — более новый, чем <code>file2</code>
<code>file1 -ot file2</code>	Проверка того, что <code>file1</code> — более старый, чем <code>file2</code>

Эти условия предоставляют возможность проверять в сценариях командного интерпретатора файлы в файловой системе и часто используются в сценариях, основанных на доступе к файлам. Поскольку данные проверки находят столь широкое применение, рассмотрим каждую из них отдельно.

Проверка каталогов

Проверка с помощью параметра `-d` позволяет определить, существует ли указанное имя файла в системе как каталог. Необходимость такой проверки обычно возникает, если должна быть предпринята попытка записать файл в каталог, а также в случае необходимости перейти в каталог:

```
$ cat test11
#!/bin/bash
# проверка перед переходом
if [ -d $HOME ]
then
    echo "Your HOME directory exists"
    cd $HOME
    ls -a
else
```

```

    echo "There is a problem with your HOME directory"
fi
$
$ ./test11
"Your HOME directory exists"
.      Documents      .gvfs      .pulse-cookie
..     Downloads      .ICEauthority .recently-used.xbel
.aptdaemon .esd_auth      .local
.sudo_as_admin_successful
.bash_history      examples      .desktop      .mozilla      Templates
.bash_logout      .fontconfig  Music      test11
.bashrc      .gconf      .nautilus  Videos
.cache      .gconfd      .openoffice.org
.xsession-errors
.config      .gksu.lock      Pictures
.xsession-errors.old
.dbus      .gnome2      .profile
Desktop      .gnome2_private Public
.dmrc      .gtk-bookmarks .pulse
$

```

В рассматриваемом образце кода условие проверки `-d` используется для определения того, существует ли каталог `$HOME` для пользователя. Если результат проверки оказывается положительным, то в ходе дальнейшей работы происходит переход с помощью команды `cd` в каталог `$HOME` и вывод листинга каталога.

Проверка того, существует ли объект

Проверка с помощью параметра `-e` позволяет определить, существует ли объект файла или каталога, перед осуществлением попытки использовать этот объект в своем сценарии:

```

$ cat test12
#!/bin/bash
# проверка того, существует ли каталог
if [ -e $HOME ]
then
    echo "OK on the directory, now to check the file"
    # проверка того, существует ли файл
    if [ -e $HOME/testing ]
    then
        # если файл существует, добавить к нему данные
        echo "Appending date to existing file"
        date >> $HOME/testing
    else
        # если файл не существует, создать новый файл
        echo "Creating new file"
        date > $HOME/testing
    fi
else
    echo "Sorry, you do not have a HOME directory"
fi
$

```

```
$ ./test12
OK on the directory, now to check the file
Creating new file
$ ./test12
OK on the directory, now to check the file
Appending date to existing file
$
```

В первой проверке используется параметр `-e` для определения того, имеет ли пользователь каталог `$HOME`. В случае положительного ответа в следующей проверке с помощью параметра `-e` определяется наличие файла `testing` в каталоге `$HOME`. Если этот файл не существует, то в сценарии командного интерпретатора используется одинарный символ перенаправления “больше” для создания нового файла, содержащего вывод команды `date`. После второго вызова этого сценария командного интерпретатора используется двойной символ “больше”, поэтому вывод команды `date` просто присоединяется к существующему файлу.

Проверка на наличие файла

Проверка с помощью параметра `-e` применяется для работы и с файлами, и с каталогами. Чтобы убедиться в том, что указанный объект является файлом, необходимо использовать сравнение с помощью параметра `-f`:

```
$ cat test13
#!/bin/bash
# проверка того, является ли объект файлом
if [ -e $HOME ]
then
    echo "The object exists, is it a file?"
    if [ -f $HOME ]
    then
        echo "Yes, it is a file!"
    else
        echo "No, it is not a file!"
        if [ -f $HOME/.bash_history ]
        then
            echo "But this is a file!"
        fi
    fi
else
    echo "Sorry, the object does not exist"
fi
$
$ ./test13
The object exists, is it a file?
No, it is not a file!
But this is a file!
$
```

Это — небольшой сценарий, но в нем выполняются очень важные проверки! Прежде всего в нем используется проверка с помощью параметра `-e` для определения того, существует ли каталог `$HOME`. В случае положительного ответа выполняется проверка с помощью параметра `-f`, что позволяет узнать, является ли данный объект файлом. Если это не файл (разумеется,

так и должно быть), используется проверка с помощью параметра `-f` для определения того, является ли файлом следующий объект, что и есть на самом деле.

Проверка того, предназначен ли файл для чтения

Прежде чем осуществлять попытку чтения данных из файла, обычно следует вначале проверить, является ли этот файл доступным для чтения. Для этого служит проверка с помощью параметра `-r`:

```
$ cat test14
#!/bin/bash
# проверка возможности чтения файла
pwfile=/etc/shadow

# вначале проверим, существует ли файл и действительно ли
# является файлом
if [ -f $pwfile ]
then
    # теперь проверим, доступен ли он для чтения
    if [ -r $pwfile ]
    then
        tail $pwfile
    else
        echo "Sorry, I am unable to read the $pwfile file"
    fi
else
    echo "Sorry, the file $file does not exist"
fi
$
$ ./test14
Sorry, I am unable to read the /etc/shadow file
$
```

Файл `/etc/shadow` содержит зашифрованные пароли для пользователей системы, поэтому он не предназначен для чтения обычными пользователями в системе. Проверка с помощью параметра `-r` привела к получению результата, согласно которому пользователь этого сценария не имеет права на чтение файла, поэтому выполнение команды `test` окончилось неудачей и командный интерпретатор `bash` выполнил раздел `else` инструкции `if-then`.

Проверка на наличие пустых файлов

Для определения того, является ли файл пустым, необходимо выполнять проверку с помощью параметра `-s`, которая позволяет убедиться в том, что файл пуст. Это особенно важно, если файл намечен для удаления. Следует внимательно относиться к результатам этой проверки, поскольку успешное завершение проверки с помощью параметра `-s` указывает на то, что в файле имеются данные:

```
$ cat test15
#!/bin/bash
# проверка того, является ли файл пустым
file=t15test
touch $file

if [ -s $file ]
```

```

then
    echo "The $file file exists and has data in it"
else
    echo "The $file exists and is empty"
fi
date > $file
if [ -s $file ]
then
    echo "The $file file has data in it"
else
    echo "The $file is still empty"
fi
$
$ ./test15
The t15test exists and is empty
The t15test file has data in it
$

```

Команда `touch` создает файл, но не помещает в него никаких данных. После вызова на выполнение команды `date` и перенаправления вывода в файл проверка с помощью параметра `-s` показывает, что в файле имеются данные.

Проверка того, предназначен ли файл для записи

Проверка с помощью параметра `-w` позволяет определить, имеет ли пользователь разрешение на запись в файл:

```

$ cat test16
#!/bin/bash
# проверка того, допускает ли файл запись

logfile=$HOME/t16test
touch $logfile
chmod u-w $logfile
now=`date +%Y%m%d-%H%M`

if [ -w $logfile ]
then
    echo "The program ran at: $now" > $logfile
    echo "The first attempt succeeded"
else
    echo "The first attempt failed"
fi

chmod u+w $logfile
if [ -w $logfile ]
then
    echo "The program ran at: $now" > $logfile
    echo "The second attempt succeeded"
else
    echo "The second attempt failed"
fi
$

```

```
$ ./test16
The first attempt failed
The second attempt succeeded
$ cat $HOME/t16test
The program ran at: 20110124-1602
$
```

Очевидно, что рассматриваемый сценарий командного интерпретатора обеспечивает выполнение многих важных действий! Вначале в этом сценарии проверяется наличие файла журнала в каталоге `$HOME`, имя этого файла сохраняется в переменной `logfile`, создается файл, а затем удаляется разрешение на запись для пользователя с помощью команды `chmod`. Затем в сценарии создается переменная `now` и сохраняется отметка времени с применением команды `date`. После завершения всех этих действий в сценарии выполняется проверка того, имеет ли пользователь разрешение на запись в новый файл журнала (который был только что создан). Очевидно, что пользователь не имеет разрешения на запись, поэтому должно появиться сообщение о неудачном завершении этой попытки.

После этого в сценарии снова используется команда `chmod`, позволяющая на сей раз предоставить пользователю разрешение на запись, и осуществляется еще одна попытка выполнить запись в файл. В данном случае операция записи выполняется успешно.

Проверка того, предназначен ли файл для выполнения

Проверка с помощью параметра `-x` представляет собой удобный способ определения того, имеет ли пользователь право на выполнение для конкретного файла. При использовании большинства команд такая необходимость обычно не возникает, но если из сценария командного интерпретатора происходит вызов большого количества других сценариев, это может пригодиться:

```
$ cat test17
#!/bin/bash
# проверка возможности выполнения файла
if [ -x test16 ]
then
    echo "You can run the script: "
    ./test16
else
    echo "Sorry, you are unable to execute the script"
fi
$
$ ./test17
You can run the script:
The first attempt failed
The second attempt succeeded
$
$ chmod u-x test16
$
$ ./test17
Sorry, you are unable to execute the script
$
```

В данном примере сценария командного интерпретатора используется проверка с помощью параметра `-x` для определения того, имеет ли пользователь разрешение на выполнение сценария `test16`. В случае положительного ответа этот сценарий выполняется (следует учитывать, что даже в сценарии командного интерпретатора необходимо правильно указывать путь, если должен быть выполнен сценарий, не находящийся в каталоге пользователя `$PATH`). После того как сценарий `test16` будет успешно выполнен в первый раз, изменим относящиеся к нему разрешения и сделаем еще одну попытку. На этот раз проверка с помощью параметра `-x` оканчивается неудачей, поскольку пользователь больше не имеет права на выполнение сценария `test16`.

Проверка права владения

Проверка с помощью параметра `-O` позволяет пользователю легко проверить, является ли он владельцем рассматриваемого файла:

```
$ cat test18
#!/bin/bash
# проверка принадлежности файла

if [ -O /etc/passwd ]
then
    echo "You are the owner of the /etc/passwd file"
else
    echo "Sorry, you are not the owner of the /etc/passwd file"
fi
$
$ ./test18
Sorry, you are not the owner of the /etc/passwd file
$
$ su
Password:
$
# ./test18
You are the owner of the /etc/passwd file
#
```

В сценарии используется проверка с помощью параметра `-O` для определения того, действительно ли пользователь, выполняющий сценарий, является владельцем файла `/etc/passwd`. В первый раз этот сценарий выполняется в обычной учетной записи пользователя, поэтому проверка оканчивается неудачей. Во второй раз применяется команда `su` для перехода в учетную запись пользователя `root`, после чего проверка выполняется успешно.

Проверка принадлежности к группе по умолчанию

Проверка с помощью параметра `-G` служит для определения заданной по умолчанию группы файла и выполняется успешно, если эта группа согласуется с заданной по умолчанию группой пользователя. При использовании параметра `-G` может возникнуть определенная путаница, поскольку этот параметр позволяет проверить только принадлежность к заданной по умолчанию группе, а не проверять все группы, к которым принадлежит пользователь. Пример применения параметра `-G` приведен ниже.

```

$ cat test19
#!/bin/bash
# проверка с определением группы файла

if [ -G $HOME/testing ]
then
    echo "You are in the same group as the file"
else
    echo "The file is not owned by your group"
fi
$
$ ls -l $HOME/testing
-rw-rw-r-- 1 rich rich 58 2011-01-30 15:51 /home/rich/testing
$
$ ./test19
You are in the same group as the file
$
$ chgrp sharing $HOME/testing
$
$ ./test19
The file is not owned by your group
$

```

При первом запуске сценария на выполнение файл `$HOME/testing` находится в группе `rich` и проверка с помощью параметра `-G` выполняется успешно. Затем вместо этой группы задается группа `sharing`, в которой пользователь также является членом. Но проверка с помощью параметра `-G` оканчивается неудачей, поскольку в ней предусмотрено сравнение только с учетом заданных по умолчанию групп, а не всех возможных вариантов принадлежности к группам.

Проверка даты файла

Последний ряд операторов проверки предназначен для сравнения значений времени создания двух файлов. Необходимость в этом возникает при написании сценариев установки программного обеспечения. Иногда приходится следить за тем, чтобы не был установлен файл, созданный ранее по сравнению с файлом, уже установленным в системе.

Проверка с помощью параметра `-nt` позволяет определить, является ли файл более новым по сравнению с другим файлом. Если файл является более новым, то время его создания меньше отстоит от текущего времени. Проверка с помощью параметра `-ot` служит для определения того, является ли файл более старым, чем другой файл. Если файл является более старым, то время его создания больше отстоит от текущего времени:

```

$ cat test20
#!/bin/bash
# проверка дат файла

if [ ./test19 -nt ./test18 ]
then
    echo "The test19 file is newer than test18"
else
    echo "The test18 file is newer than test19"
fi

```



```

if [ ./test17 -ot ./test19 ]
then
    echo "The test17 file is older than the test19 file"
fi
$
$ ./test20
The test19 file is newer than test18
The test17 file is older than the test19 file
$
$ ls -l test17 test18 test19
-rwxrw-r-- 1 rich rich 167 2011-01-30 16:31 test17
-rwxrw-r-- 1 rich rich 185 2011-01-30 17:46 test18
-rwxrw-r-- 1 rich rich 167 2011-01-30 17:50 test19
$

```

Пути к файлам, используемые в операциях сравнения, определяются относительно каталога, из которого происходит запуск сценария на выполнение. Это может стать причиной ошибок, если не исключена возможность перемещения проверяемых файлов в другие каталоги. Еще одна проблема состоит в том, что ни в одном из этих операторов сравнения не проводится предварительная проверка того, существует ли файл. Рассмотрим следующий вариант проверки:

```

$ cat test21
#!/bin/bash
# проверка дат файла

if [ ./badfile1 -nt ./badfile2 ]
then
    echo "The badfile1 file is newer than badfile2"
else
    echo "The badfile2 file is newer than badfile1"
fi
$
$ ./test21
The badfile2 file is newer than badfile1
$

```

Этот небольшой пример демонстрирует, что если файлы не существуют, то сравнение с помощью параметра `-nt` просто возвращает условие, указывающее на неудачное завершение. Следует обязательно обеспечивать проверку существования файлов, прежде чем предпринимать попытку их использовать в сравнении с помощью параметра `-ot` или `-nt`.

Проверка с помощью составных условий

Инструкция `if-then` позволяет использовать булеву логику для комбинирования проверки. Предусмотрена возможность задавать два следующих логических оператора:

- `[condition1] && [condition2]`
- `[condition1] || [condition2]`

В первой логической операции для объединения двух условий используется логический оператор AND. Операторы раздела `then` будут выполнены, только если истинными окажутся оба условия.

Во второй логической операции два условия объединяются с помощью логического оператора OR. Если проверка любого условия приведет к получению истинного значения, то произойдет выполнение операторов раздела then.

```
$ cat test22
#!/bin/bash
# проверка с помощью составных операторов сравнения

if [ -d $HOME ] && [ -w $HOME/testing ]
then
    echo "The file exists and you can write to it"
else
    echo "I cannot write to the file"
fi
$
$ ./test22
I cannot write to the file
$ touch $HOME/testing
$
$ ./test22
The file exists and you can write to it
$
```

При использовании логического оператора AND должны быть выполнены оба сравнения. Первая проверка предназначена для определения того, существует ли каталог \$HOME для пользователя. Вторая проверка позволяет выяснить, существует ли файл с именем testing в каталоге \$HOME пользователя и имеет ли пользователь разрешения на запись в этот файл. Если какая-либо из этих проверок оканчивается неудачей, то происходит неудачное завершение во всей инструкции if и командный интерпретатор переходит к выполнению раздела else. А после успешного завершения обеих проверок инструкция if выполняется успешно и командный интерпретатор выполняет инструкции в разделе then.

Дополнительные средства инструкции if-then

Относительно недавно в командный интерпретатор bash были внесены дополнения, поддерживающие следующие усовершенствованные функции, которые могут применяться в инструкциях if-then:

- применение двойных круглых скобок для оформления математических выражений;
- применение двойных квадратных скобок для работы с усовершенствованными функциями обработки строк.

В следующих разделах каждая из этих функций рассматривается более подробно.

Применение двойных круглых скобок

Команда с *двойными круглыми скобками* позволяет включать в операции сравнения дополнительные математические выражения. Команда test обеспечивает возможность применять

в операциях сравнения простые арифметические операции. Команда с двойными круглыми скобками позволяет воспользоваться более широким перечнем знаков математических операций, с которыми часто работают программисты, применяющие другие языки программирования. Команда с двойными круглыми скобками имеет следующий формат:

```
(( expression ))
```

В качестве элемента `expression` может быть представлено любое математическое выражение присваивания или сравнения. В табл. 11.4 приведен список дополнительных операторов, предназначенных для использования в команде с двойными круглыми скобками, которые могут применяться наряду со стандартными математическими операциями, предусмотренными в команде `test`.

Таблица 11.4. Операторы, применимые в команде с двойными круглыми скобками			
Оператор	Описание	Оператор	Описание
<code>val++</code>	Постинкрементное присваивание	<code><<</code>	Побитовый сдвиг влево
<code>val--</code>	Постдекрементное присваивание	<code>>></code>	Побитовый сдвиг вправо
<code>++val</code>	Предынкrementное присваивание	<code>&</code>	Побитовое логическое AND
<code>--val</code>	Предекрементное присваивание	<code> </code>	Побитовое логическое OR
<code>!</code>	Логическое отрицание	<code>&&</code>	Логический AND
<code>~</code>	Побитовое отрицание	<code> </code>	Логический OR
<code>**</code>	Возведение в степень		

Предусмотрена возможность не только использовать команду с двойными круглыми скобками в инструкции `if`, но и определять с ее помощью обычные команды в сценарии для присваивания значений:

```
$ cat test23
#!/bin/bash
# использование двойных скобок

val1=10

if (( $val1 ** 2 > 90 ))
then
    (( val2 = $val1 ** 2 ))
    echo "The square of $val1 is $val2"
fi
$
$ ./test23
The square of 10 is 100
$
```

Следует отметить, что в выражениях в двойных круглых скобках больше не требуется экранировать символ “больше”. Это — еще одно улучшение в работе, которое влечет за собой применение команды с двойными круглыми скобками.

Использование двойных квадратных скобок

Команда с *двойными квадратными скобками* предоставляет дополнительные возможности сравнения строк. Команда с двойными квадратными скобками имеет следующий формат:

```
[[ expression ]]
```

В выражении `expression`, заключенном в двойные квадратные скобки, используются те же стандартные операции сравнения строк, что и в команде `test`. Но кроме этого предоставляются дополнительные возможности, не предусмотренные в команде `test`, — *сопоставление с шаблоном*.

Средства сопоставления с шаблонами позволяют определять регулярные выражения (которые подробно обсуждаются в главе 19) и проводить их сопоставление со строковыми значениями:

```
$ cat test24
#!/bin/bash
# использование сопоставления с шаблоном

if [[ $USER == r* ]]
then
    echo "Hello $USER"
else
    echo "Sorry, I do not know you"
fi
$
$ ./test24
Hello rich
$
```

В рассматриваемой команде с двойными квадратными скобками проводится сопоставление значения переменной среды `$USER` для определения того, начинается ли это значение с буквы `r`. В случае получения положительного результата проверки командный интерпретатор выполняет команды из раздела `then`.

Команда `case`

При написании сценариев часто обнаруживается, что необходимо вычислить значение переменной, а затем провести поиск конкретного значения среди целого ряда возможных значений. В следующем сценарии для этого пришлось написать такую длинную инструкцию `if-then-else`, как показано ниже.

```
$ cat test25
#!/bin/bash
# поиск возможного значения

if [ $USER = "rich" ]
then
    echo "Welcome $USER"
    echo "Please enjoy your visit"
elif [ $USER = barbara ]
then
    echo "Welcome $USER"
    echo "Please enjoy your visit"
elif [ $USER = testing ]
then
    echo "Special testing account"
elif [ $USER = jessica ]
```

```

then
    echo "Do not forget to logout when you're done"
else
    echo "Sorry, you are not allowed here"
fi
$
$ ./test25
Welcome rich
Please enjoy your visit
$

```

С помощью инструкций `elif` снова и снова происходит проверка условий `if-then`, пока не будет найдено требуемое значение для одной сравниваемой переменной.

Тем не менее можно избавиться от необходимости применения всех этих инструкций `elif`, которые требуют неоднократно проверять одно и то же значение переменной. Для этого достаточно воспользоваться командой `case`. В команде `case` проверка значения одной переменной на соответствие нескольким разным значениям организована с применением формата списка:

```

case variable in
pattern1 | pattern2) commands1;;
pattern3) commands2;;
*) default commands;;
esac

```

Команда `case` сравнивает заданную переменную *variable* с несколькими разными шаблонами. При обнаружении согласования переменной с каким-либо из шаблонов *pattern* командный интерпретатор выполняет команды *commands*, указанные для этого шаблона. В одной строке можно задавать несколько шаблонов, используя операцию со знаком операции “вертикальная черта” для отделения одного шаблона от другого. Символ звездочки служит универсальной ловушкой для тех значений, которые не были сопоставлены ни с одним из перечисленных шаблонов. В примере, приведенном ниже, показано, как преобразовать рассматриваемую программу `if-then-else` с использованием команды `case`.

```

$ cat test26
#!/bin/bash
# использование команды case

case $USER in
rich | barbara)
    echo "Welcome, $USER"
    echo "Please enjoy your visit";;
testing)
    echo "Special testing account";;
jessica)
    echo "Do not forget to log off when you're done";;
*)
    echo "Sorry, you are not allowed here";;
esac
$
$ ./test26
Welcome, rich
Please enjoy your visit
$

```

Очевидно, что команда `case` предоставляет более удобный способ задания различных вариантов проверки для каждого возможного значения переменной.

Резюме

Структурированные команды позволяют изменять обычный ход выполнения сценария командного интерпретатора. Наиболее широко применяемые структурированные команды основаны на использовании инструкции `if-then`. Эта инструкция позволяет выполнить команду, а затем перейти к выполнению других команд с учетом того, была ли первая команда выполнена успешно.

Инструкцию `if-then` можно также дополнить, чтобы включить в нее ряд команд, которые командный интерпретатор `bash` должен выполнить в случае неудачного завершения указанной команды. Инструкция `if-then-else` позволяет указать, какие команды должны быть выполнены, только если команда, выполняемая в первую очередь, возвращает ненулевой код статуса выхода.

Предусмотрена также возможность соединять несколько инструкций `if-then-else` в цепочки с использованием инструкции `elif`. Применение инструкции `elif` эквивалентно применению инструкции `else if` и обеспечивает проведение дополнительной проверки, если выполнение предыдущей команды окончилась неудачей.

В большинстве сценариев вместо выполнения команды и выбора продолжения с учетом ее результатов достаточно лишь проверить определенное условие, такое как числовое значение, содержимое строки или статус файла либо каталога. Один из наиболее простых способов проверки всех необходимых условий заключается в использовании команды `test`. Если проверка условия приводит к получению истинного значения, то команда `test` вырабатывает нулевой код статуса выхода для передачи в инструкцию `if-then`. Если проверка условия приводит к получению ложного значения, то команда `test` вырабатывает ненулевой код статуса выхода и передает его в инструкцию `if-then`.

Квадратные скобки представляют собой специальную команду `bash`, которая является синонимом для команды `test`. Предусмотрена возможность заключить проверяемое условие в квадратные скобки в инструкции `if-then` для проверки условий, касающихся значений чисел и строк, а также данных о файлах.

Команда с двойными круглыми скобками позволяет выполнять более сложные математические вычисления с использованием дополнительных операторов, а команда с двойными квадратными скобками дает возможность проводить проверки на основе улучшенных способов сопоставления строк с шаблонами.

Наконец, в настоящей главе рассматривалась команда `case`, которая может стать основой более удобного способа выполнения многочисленных проверок значения одной переменной по списку значений, заменяя несколько команд `if-then-else`.

В следующей главе описание структурированных команд будет продолжено и речь пойдет об использовании команд командного интерпретатора, предназначенных для организации циклов. Команды `for` и `while` позволяют формировать циклы, с помощью которых итерации в ряде команд происходят требуемое количество раз.

Продолжение описания структурирован- ных команд

В предыдущей главе было показано, как управлять потоком инструкций в программе сценария командного интерпретатора, проверяя вывод команд и значения переменных. В настоящей главе продолжается рассмотрение структурированных команд, которые управляют ходом выполнения сценариев командного интерпретатора. В ней будет показано, как организовать выполнение повторяющихся процессов, и приведены команды, позволяющие обрабатывать в цикле ряд команд до достижения указанного условия. В данной главе обсуждаются команды организации циклов `for`, `while` и `until` командного интерпретатора `bash` и демонстрируется их использование.

Команда `for`

В практике программирования широко применяется способ организации работы путем выполнения итераций с охватом ряда команд. Часто возникает необходимость повторять ряд команд до тех пор, пока не будет выполнено какое-то конкретное условие, например, будут обработаны все файлы в каталоге, данные по всем пользователям в системе или все строки в текстовом файле.

Командный интерпретатор `bash` предоставляет возможность использовать команду `for` для создания цикла, в котором производится итерация по ряду значений. В каждой

ГЛАВА

12

В этой главе...

Команда `for`

Команда `for` в стиле языка C

Команда `while`

Команда `until`

Уровень вложенности циклов

Организация циклов на основе данных файла

Управление циклом

Обработка вывода в цикле

Резюме

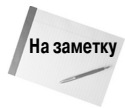
итерации выполняется определенный набор команд с использованием одного из значений в ряде. Ниже приведен основной формат команды `for` командного интерпретатора `bash`.

```
for var in list
do
  commands
done
```

Ряд значений, используемый в итерациях, задается с помощью параметра `list`. Предусмотрено несколько различных способов задания значений в этом списке.

В каждой итерации переменная `var` содержит текущее значение в списке `list`. При первой итерации используется первый элемент в списке, при второй итерации — второй элемент и так далее, до исчерпания всех элементов в списке.

В качестве команд `commands`, которые вводятся между инструкциями `do` и `done`, можно задавать одну или несколько стандартных команд командного интерпретатора `bash`. При выполнении команд может использоваться переменная `$var`, которая содержит текущее значение элемента списка для данной итерации.



При желании можно ввести инструкцию `do` в той же строке, что и инструкцию `for`, но при этом данную инструкцию необходимо отделить от элементов списка точкой с запятой: `for var in list; do`.

Как уже было сказано, предусмотрено несколько способов задания значений в списке. Применимые способы решения этой задачи рассматриваются в следующих разделах.

Чтение значений в списке

Наиболее простой способ использования команды `for` состоит в осуществлении с ее помощью итераций по списку значений, определенному в самой команде `for`:

```
$ cat test1
#!/bin/bash
# команда for в основной форме

for test in Alabama Alaska Arizona Arkansas California Colorado
do
  echo The next state is $test
done
$ ./test1
The next state is Alabama
The next state is Alaska
The next state is Arizona
The next state is Arkansas
The next state is California
The next state is Colorado
$
```

При осуществлении в команде `for` итераций по заданному списку значений происходит присваивание переменной `$test` следующего значения в списке. Переменная `$test` может использоваться в инструкциях команды `for` точно так же, как и любая другая переменная сценария. После последней итерации переменная `$test` остается действительной на протяжении всей оставшейся части сценария командного интерпретатора. Эта переменная сохраняет зна-

чение, присвоенное при последней итерации (если ее значение не изменяется в другом месте сценария):

```
$ cat test1b
#!/bin/bash
# проверка переменной for после прохода по циклу

for test in Alabama Alaska Arizona Arkansas California Colorado
do
    echo "The next state is $test"
done
echo "The last state we visited was $test"
test=Connecticut
echo "Wait, now we're visiting $test"
$ ./test1b
The next state is Alabama
The next state is Alaska
The next state is Arizona
The next state is Arkansas
The next state is California
The next state is Colorado
The last state we visited was Colorado
Wait, now we're visiting Connecticut
$
```

Очевидно, что переменная `$test` сохраняет свое значение, а также позволяет изменять значение и использовать его вне цикла команды `for`, как и любая другая переменная.

Чтение сложных значений в списке

На первый взгляд организация цикла `for` кажется простой, но это не всегда так. Иногда приходится сталкиваться с такими данными, применение которых вызывает проблемы. Ниже приведен классический пример того, что может стать источником проблем для программистов при использовании языка сценариев командного интерпретатора.

```
$ cat badtest1
#!/bin/bash
# еще один пример того, как не следует использовать команду for

for test in I don't know if this'll work
do
    echo "word:$test"
done
$ ./badtest1
word:I
word:dont know if thisll
word:work
$
```

Итак, получено нечто непредвиденное. Командный интерпретатор обнаружил одинарные кавычки среди значений в списке и попытался использовать их для определения отдельного значения данных, в результате чего в процессе обработки сценария возникла настоящая путаница.

Предусмотрены два способа решения этой проблемы:

- применение экранирующего символа (обратной косой черты) для экранирования одинарной кавычки;
- задание двойных кавычек для определения значений, в которых используются одинарные кавычки.

Ни в одном из этих решений нет ничего сложного, но и то и другое позволяет полностью решить данную проблему:

```
$ cat test2
#!/bin/bash
# еще один пример того, как не следует использовать команду for

for test in I don't know if "this'll" work
do
    echo "word:$test"
done
$ ./test2
word:I
word:don't
word:know
word:if
word:this'll
word:work
$
```

В первом значении, вызывающем нарушение в работе, был добавлен символ обратной косой черты для экранирования одинарной кавычки в значении `don't`. При исправлении второго неправильно заданного значения строка `this'll` была заключена в двойные кавычки. Оба эти метода показали себя вполне приемлемыми как способы задания правильного значения.

Еще с одной проблемой можно столкнуться при обработке строки, состоящей из нескольких слов. Напомним, что формат цикла `for` предполагает, что для разделения отдельных значений применяется пробел. Если же будут заданы значения данных, содержащие пробелы, то возникнет еще одна проблема:

```
$ cat badtest2
#!/bin/bash
# еще один пример того, как не следует использовать команду for

for test in Nevada New Hampshire New Mexico New York North Carolina
do
    echo "Now going to $test"
done
$ ./badtest1
Now going to Nevada
Now going to New
Now going to Hampshire
Now going to New
Now going to Mexico
Now going to New
Now going to York
Now going to North
```

```
Now going to Carolina
$
```

Но в данном случае получено не совсем то, что требуется. В команде `for` каждое значение в списке должно быть отделено от другого пробелом. Если же пробелы встречаются в отдельных значениях данных, то эти значения должны быть заключены в двойные кавычки:

```
$ cat test3
#!/bin/bash
# пример того, как правильно определять значения

for test in Nevada "New Hampshire" "New Mexico" "New York"
do
    echo "Now going to $test"
done
$ ./test3
Now going to Nevada
Now going to New Hampshire
Now going to New Mexico
Now going to New York
$
```

Теперь в команде `for` успешно различаются отдельные значения. Заслуживает также внимание то, что даже если значение заключено в двойные кавычки, командный интерпретатор не включает эти кавычки в состав значения.

Чтение списка из переменной

При программировании на языке сценариев командного интерпретатора часто приходится сталкиваться с такой ситуацией, когда накапливается список значений и сохраняется в переменной, после чего должны быть выполнены итерации по элементам списка. Это также можно сделать с помощью команды `for`:

```
$ cat test4
#!/bin/bash
# использование переменной для хранения списка

list="Alabama Alaska Arizona Arkansas Colorado"
list=$list" Connecticut"

for state in $list
do
    echo "Have you ever visited $state?"
done
$ ./test4
Have you ever visited Alabama?
Have you ever visited Alaska?
Have you ever visited Arizona?
Have you ever visited Arkansas?
Have you ever visited Colorado?
Have you ever visited Connecticut?
$
```

Переменная `$list` содержит стандартный список текстовых значений, которые должны использоваться для осуществления итераций. Заслуживает внимания то, что в этом коде используется также еще один оператор присваивания для добавления (или конкатенации) элемента к существующему списку, содержащемуся в переменной `$list`. Это — общепринятый метод добавления текста к концу существующей текстовой строки, сохраняющейся в переменной.

Чтение значений из команды

Еще один способ формирования значений для использования в списке состоит в применении вывода команды. При этом используются символы обратной одинарной кавычки для выполнения любой команды, вырабатывающей вывод, а затем вывод команды применяется в команде `for`:

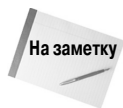
```
$ cat test5
#!/bin/bash
# чтение значений из файла

file="states"

for state in `cat $file`
do
    echo "Visit beautiful $state"
done
$ cat states
Alabama
Alaska
Arizona
Arkansas
Colorado
Connecticut
Delaware
Florida
Georgia
$ ./test5
Visit beautiful Alabama
Visit beautiful Alaska
Visit beautiful Arizona
Visit beautiful Arkansas
Visit beautiful Colorado
Visit beautiful Connecticut
Visit beautiful Delaware
Visit beautiful Florida
Visit beautiful Georgia
$
```

В настоящем примере команда `cat`, заданная в обратных одинарных кавычках, служит для отображения данных о состоянии файлов. Заслуживает внимания то, что файл с данными о состоянии имеет такой формат: каждое состояние приведено в отдельной строке, а не задано в общей строке с разделением пробелами. Команда `for` по-прежнему выполняет итерации по выводу команды `cat`, обрабатывая каждый раз одну строку, исходя из предположения, что каждое состояние представлено в отдельной строке. Однако это не решает проблему наличия

пробелов в данных. Если в списке находятся данные о состоянии, содержащие пробелы, то в команде `for` все равно каждый элемент данных, отделенных от других пробелами, рассматривается как отдельное значение. Для применения такой организации работы есть весомые основания, которые будут рассматриваться в следующем разделе.



В примере кода `test5` переменной присваивается имя файла без указания пути, т.е. задается только само имя. Это означает, что файл должен находиться в том же каталоге, что и сценарий. Если дело обстоит иначе, то должно использоваться полное имя пути (абсолютное или относительное) для ссылки на местоположение файла.

Изменение значения разделителя полей

Причина возникновения описанной выше проблемы состоит в том, что не откорректировано значение специальной переменной среды `IFS`, которую принято называть *внутренним разделителем полей*. Переменная среды `IFS` определяет список символов, используемых командным интерпретатором `bash` в качестве разделителей полей. По умолчанию в командном интерпретаторе `bash` как разделители полей применяются следующие символы:

- пробел;
- знак табуляции;
- знак перехода на новую строку.

Если командный интерпретатор `bash` обнаруживает любой из этих символов в данных, то принимает предположение, что после этого символа начинается новое поле данных в списке. В связи с такой организацией работы с данными, которые могут содержать пробелы (такими как имена файлов), могут возникать непредвиденные ситуации, как было показано в предыдущем примере сценария.

Для решения этой проблемы можно на время изменять значение переменной среды `IFS` в конкретном сценарии командного интерпретатора, чтобы исключить некоторые символы, распознаваемые командным интерпретатором `bash` как разделители полей. Но сам способ осуществления этого действия является довольно странным. Например, чтобы изменить значение `IFS` в целях обеспечения распознавания с его помощью только символа обозначения конца строки, необходимо сделать следующее:

```
IFS=$'\n'
```

Добавление этой инструкции к конкретному сценарию равносильно передаче командному интерпретатору `bash` указания, что пробелы и знаки табуляции в значениях данных следует пропускать. Применение данного способа к предыдущему сценарию приводит к получению следующих результатов:

```
$ cat test5b
#!/bin/bash
# чтение значений из файла

file="states"

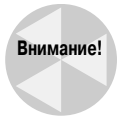
IFS=$'\n'
for state in `cat $file`
do
    echo "Visit beautiful $state"
done
$ ./test5b
```

```

Visit beautiful Alabama
Visit beautiful Alaska
Visit beautiful Arizona
Visit beautiful Arkansas
Visit beautiful Colorado
Visit beautiful Connecticut
Visit beautiful Delaware
Visit beautiful Florida
Visit beautiful Georgia
Visit beautiful New York
Visit beautiful New Hampshire
Visit beautiful North Carolina
$

```

Теперь в сценарии командного интерпретатора обработка значений в списке, содержащих пробелы, была выполнена успешно.



Однако при работе программиста над длинным сценарием может сложиться такая ситуация, что значение IFS будет изменено в одном месте, после чего программист об этом забудет и в другом месте в сценарии станет руководствоваться предположением, что IFS имеет значение по умолчанию. Поэтому необходимо придерживаться такого безопасного подхода: сохранять исходное значение IFS перед его изменением, а затем восстанавливать прежнее значение после завершения работы, в которой требовалось измененное значение.

Такой способ организации работы может быть представлен в коде примерно так:

```

IFS.OLD=$IFS
IFS=$'\n'
<use the new IFS value in code>
IFS=$IFS.OLD

```

Это позволяет гарантировать восстановление значения по умолчанию переменной IFS для применения в последующих операциях в сценарии.

Предусмотрены также другие превосходные способы применения переменной среды IFS. Предположим, должны быть выполнены итерации по значениям в файле, которые разделены двоеточиями (как в файле `/etc/passwd`). Для осуществления этого достаточно присвоить переменной IFS в качестве значения символ двоеточия:

```
IFS=:
```

Если в переменной IFS необходимо задать несколько символов, достаточно привести все эти символы в виде одной строки в операторе присваивания значения этой переменной:

```
IFS=$'\n':; "
```

Этот оператор присваивания указывает, что в качестве разделителей полей должны использоваться символ обозначения новой строки, двоеточие, точка с запятой и двойные кавычки. Количество возможных способов разбиения данных с помощью символов, заданных в переменной IFS, является буквально неограниченным.

Чтение каталога с использованием символов-заместителей

Наконец, можно применить команду `for` для обеспечения автоматической итерации по файлам в каталоге. Для этого необходимо использовать символы-заместители в именах фай-

лов или путей. Это вынуждает командный интерпретатор использовать операцию *подстановки имен файлов*. Подстановка имен файлов — это процесс создания имен файлов или путей, которые сопоставляются с указанным символом-заместителем.

Такое средство может стать основой великолепного способа обработки файлов в каталоге, если не известны все имена файлов:

```
$ cat test6
#!/bin/bash
# итерации по всем файлам в каталоге

for file in /home/rich/test/*
do

    if [ -d "$file" ]
    then
        echo "$file is a directory"
    elif [ -f "$file" ]
    then
        echo "$file is a file"
    fi
done
$ ./test6
/home/rich/test/dirl is a directory
/home/rich/test/myprog.c is a file
/home/rich/test/myprog is a file
/home/rich/test/myscript is a file
/home/rich/test/newdir is a directory
/home/rich/test/newfile is a file
/home/rich/test/newfile2 is a file
/home/rich/test/testdir is a directory
/home/rich/test/testing is a file
/home/rich/test/testprog is a file
/home/rich/test/testprog.c is a file
$
```

Команда `for` выполняет итерации по результатам формирования листинга `/home/rich/test/*`. В коде происходит проверка каждой записи с использованием команды `test` (для этого служит метод с квадратными скобками) в целях определения того, относится ли запись к каталогу, с применением параметра `-d`, или к файлу, с применением параметра `-f` (см. главу 11).

В этом примере заслуживает внимания то, что в проверках в инструкции `if` применена необычная конструкция:

```
if [ -d "$file" ]
```

В Linux имена каталогов и файлов, содержащие пробелы, являются вполне допустимыми. Чтобы учесть наличие пробелов, необходимо заключить переменную `$file` в двойные кавычки. Если это не будет сделано, то появится ошибка при обнаружении имени каталога или файла, которое содержит пробелы:

```
./test6: line 6: [: too many arguments
./test6: line 9: [: too many arguments
```

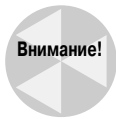
Командный интерпретатор `bash` в команде `test` интерпретирует части имени, разделенные пробелами, как отдельные слова, что приводит к возникновению ошибки.

Можно также объединить метод с поиском в каталоге и метод работы со списком в одной и той же инструкции `for`, для чего достаточно ввести ряд обозначений каталогов символами-заместителями в команде `for`:

```
$ cat test7
#!/bin/bash
# итерация по нескольким каталогам

for file in /home/rich/.b* /home/rich/badtest
do
    if [ -d "$file" ]
    then
        echo "$file is a directory"
    elif [ -f "$file" ]
    then
        echo "$file is a file"
    else
        echo "$file doesn't exist"
    fi
done
$ ./test7
/home/rich/.backup.timestamp is a file
/home/rich/.bash_history is a file
/home/rich/.bash_logout is a file
/home/rich/.bash_profile is a file
/home/rich/.bashrc is a file
/home/rich/badtest doesn't exist
$
```

В инструкции `for` вначале используется подстановка имен файлов для итерации по списку файлов, соответствующему символу-заместителю, после чего осуществляется итерация с переходом к следующему файлу в списке. В список можно объединить любое количество записей с символами-заместителями для проведения по ним итераций.



Заслуживает внимания то, что данные списка могут содержать практически любые значения. Даже если файл или каталог не существует, в инструкции `for` предпринимается попытка обработать запись, приведенную в списке. Это может стать источником проблем при работе с файлами и каталогами. В самой инструкции `for` не предусмотрен какой-либо способ определения того, не осуществляется ли попытка выполнить итерации по несуществующему каталогу. Поэтому рекомендуется всегда проверять наличие каждого файла или каталога перед попыткой обработать его.

Команда `for` в стиле языка C

У разработчиков, привыкших работать на языке программирования C, может вызвать удивление то, какой способ использования команды `for` предусмотрен в командном интерпретаторе `bash`. На языке C в цикле `for` обычно определяется переменная, которая затем изменяется автоматически после каждой итерации. Как правило, программисты используют эту переменную в качестве счетчика и увеличивают или уменьшают значение счетчика на единицу в каждой итерации. Команда `for` командного интерпретатора `bash` также предоставляет возможность организовать работу подобным образом. В настоящем разделе приведено описание того, как можно использовать команду `for` в стиле языка C в сценариях командного интерпретатора `bash`.

Команда `for` в стиле языка C

В языке C в инструкции `for` предусмотрены: определенный способ указания переменной, способ задания условия, которое должно оставаться истинным, чтобы продолжались итерации, и способ изменения значения переменной при каждой итерации. Если указанное условие принимает ложное значение, цикл `for` останавливается. Выражение условия определяется с использованием стандартных математических символов. В качестве примера рассмотрим следующий код на языке C:

```
for (i = 0; i < 10; i++)
{
    printf("The next number is %d\n", i);
}
```

При выполнении этого кода осуществляется простой итеративный цикл, в котором переменная `i` используется как счетчик. В первом разделе инструкции этой переменной присваивается значение по умолчанию. Во втором разделе определяется условие, при выполнении которого итерации в цикле продолжают. Если заданное условие принимает ложное значение, итерации в цикле `for` прекращаются. В третьем, последнем разделе инструкции определяется итеративный процесс. После каждой итерации выполняется выражение, определенное в последнем разделе. В настоящем примере переменная `i` после каждой итерации увеличивается на единицу.

В командном интерпретаторе `bash` также поддерживается версия цикла `for`, которая выглядит аналогично циклу `for` в языке C, хотя и имеет некоторые тонкие отличия, в том числе такие важные, которые могут стать причиной путаницы в работе программистов на языке сценариев командного интерпретатора. Ниже приведен основной формат цикла `for` командного интерпретатора `bash` в стиле языка C.

```
for ((
variable
assignment ;
condition ;
iteration
process ))
```

Формат цикла `for` в стиле языка C может стать причиной путаницы для программистов на языке сценариев командного интерпретатора `bash`, поскольку в нем используются ссылки на переменные в стиле C вместо ссылок на переменные в стиле командного интерпретатора. Ниже показано, как выглядит команда `for` в стиле C.

```
for (( a = 1; a < 10; a++ ))
```

Отметим несколько следующих особенностей этой команды, которые характеризуют ее отличие от стандартной инструкции `for` командного интерпретатора `bash`.

- Инструкция присваивания значения переменной может содержать пробелы.
- Переменной в условии не предшествует знак доллара.
- Выражение, с помощью которого организуется итеративный процесс, не соответствует формату команды `expr`.

Этот формат был создан разработчиками командного интерпретатора так, чтобы он в наибольшей степени соответствовал команде `for` в стиле языка C. Разумеется, это позволяет упростить работу программистов, которые владеют языком C, а что касается программистов на

языке сценариев командного интерпретатора, то даже наиболее опытные из них могут почувствовать себя растерянными. Соблюдайте осторожность при использовании цикла `for` в стиле `C` в конкретных сценариях.

Ниже приведен пример применения команды `for` в стиле `C` в программе командного интерпретатора `bash`.

```
$ cat test8
#!/bin/bash
# проверка в цикле for в стиле C

for (( i=1; i <= 10; i++ ))
do
    echo "The next number is $i"
done
$ ./test8
The next number is 1
The next number is 2
The next number is 3
The next number is 4
The next number is 5
The next number is 6
The next number is 7
The next number is 8
The next number is 9
The next number is 10
$
```

В цикле `for` осуществляется итерация по ряду команд с использованием переменной, определенной в цикле `for` (в данном случае переменная имеет имя `i`). В каждой итерации переменная `$i` содержит значение, присвоенное в цикле `for`. А после каждой итерации над переменной выполняется действие, предусмотренное в цикле; в данном примере происходит увеличение значения переменной на единицу.

Использование нескольких переменных

Команда `for` в стиле `C` позволяет также применять несколько переменных в каждой итерации. В цикле каждая переменная обрабатывается отдельно, что позволяет определить различные итеративные процессы для каждой переменной. Но, несмотря на то, что можно использовать несколько переменных, в цикле `for` допускается возможность определить только одно условие:

```
$ cat test9
#!/bin/bash
# несколько переменных

for (( a=1, b=10; a <= 10; a++, b-- ))
do
    echo "$a - $b"
done
$ ./test9
1 - 10
2 - 9
```

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

Переменные `a` и `b` инициализируются с присваиванием различных значений, и для них определяются разные процессы итерации. С каждым проходом по циклу значение переменной `a` увеличивается, а значение переменной `b` уменьшается.

Команда `while`

Команда `while` представляет собой своего рода сочетание инструкции `if-then` и цикла `for`. Команда `while` позволяет определить проверяемую команду, а затем обрабатывать в цикле ряд команд до тех пор, пока заданная команда `test` возвращает нулевой статус выхода. Проверка возвращаемого значения команды `test` происходит в начале каждой итерации. Как только команда `test` возвратит ненулевой статус выхода, выполнение ряда команд в команде `while` прекращается.

Основной формат `while`

Команда `while` имеет следующий формат:

```
while
test
command
do

other
commands
done
```

Команда проверки `test`, определенная в команде `while`, имеет такой же формат, как и в инструкциях `if-then` (см. главу 11). Как и в инструкции `if-then`, можно использовать любую обычную команду командного интерпретатора `bash` или включить команду `test` для проверки условий, таких как значения переменных.

Ключевой особенностью команды `while` является то, что в ней должен измениться статус выхода указанной команды проверки в результате выполнения команд, заданных в цикле. Если не будет изменяться статус выхода, цикл `while` может стать бесконечным.

Чаще всего команда проверки применяется в форме, которая предусматривает использование квадратных скобок для проверки значения одной из переменных командного интерпретатора, которая применяется в командах цикла:

```
$ cat test10
#!/bin/bash
# проверка в команде while

var1=10
```

```

while [ $var1 -gt 0 ]
do
    echo $var1
    var1=$(( $var1 - 1 ))
done
$ ./test10
10
9
8
7
6
5
4
3
2
1
$

```

Команда `while` определяет условие, которое проверяется при каждой итерации:

```
while [ $var1 -gt 0 ]
```

Пока проверяемое условие остается истинным, команда `while` продолжает обрабатывать в цикле определенные команды. По результатам этих команд должно изменяться значение переменной, используемой в проверяемом условии, поскольку иначе цикл может стать бесконечным. В рассматриваемом примере используется арифметическая операция командного интерпретатора для уменьшения значения переменной на единицу:

```
var1=$(( $var1 - 1 ))
```

Цикл `while` останавливается, если проверяемое условие перестает быть истинным.

Использование нескольких команд `test`

Допускается использовать даже такой нестандартный вариант команды `while`, который позволяет определять несколько команд `test` в строке инструкции `while`. При этом для определения условия останова цикла служит статус выхода только последней команды `test`. Такая конструкция должна использоваться с осторожностью, поскольку она способствует возникновению некоторых непредвиденных побочных эффектов. Рассмотрим один из примеров такой ситуации:

```

$ cat test11
#!/bin/bash
# проверка в цикле while с помощью нескольких команд

var1=10

while echo $var1
    [ $var1 -ge 0 ]
do
    echo "This is inside the loop"
    var1=$(( $var1 - 1 ))
done
$ ./test11

```

```

10
This is inside the loop
9
This is inside the loop
8
This is inside the loop
7
This is inside the loop
6
This is inside the loop
5
This is inside the loop
4
This is inside the loop
3
This is inside the loop
2
This is inside the loop
1
This is inside the loop
0
This is inside the loop
-1
$

```

То, что происходит в данном примере, заслуживает серьезного внимания. В инструкции `while` определены две команды проверки:

```

while echo $var1
[ $var1 -ge 0 ]

```

В первой проверке просто отображается текущее значение переменной `var1`. Во второй проверке используются квадратные скобки для определения значения переменной `var1`. В самом цикле инструкция `echo` выводит простое текстовое сообщение, которое указывает, что выполнен проход по циклу. Выполняя этот пример, необходимо отметить, как завершается этот вывод:

```

This is inside the loop
-1
$

```

В цикле `while` инструкция `echo` была выполнена при значении переменной `var1`, равном нулю, после чего произошло уменьшение значения переменной `var1`. Затем были выполнены команды проверки для перехода к следующей итерации. Команда проверки `echo` при ее выполнении показала значение переменной `var1`, которое теперь стало меньше нуля. Но цикл `while` завершился только после того, как командный интерпретатор выполнил команду проверки.

Этот пример показывает, что в инструкции `while` с несколькими командами проверки в каждой итерации выполняются все команды проверки, включая последнюю итерацию, в которой выполнение последней команды проверки оканчивается неудачей. Это означает, что контролируются не все результаты команд проверки, поэтому необходимо соблюдать осторожность. Следует также обратить внимание на то, как должны быть заданы многочисленные команды проверки. Важно отметить, что каждая команда проверки задана в отдельной строке!

Команда `until`

Команда `until` действует прямо противоположным способом по сравнению с командой `while`. Команда `until` требует, чтобы была задана команда проверки, которая при обычных обстоятельствах вырабатывает ненулевой статус выхода. До тех пор пока статус выхода команды проверки остается ненулевым, командный интерпретатор `bash` выполняет команды, которые входят в состав цикла. Как только команда проверки возвращает нулевой статус выхода, цикл останавливается.

Как и следовало ожидать, команда `until` имеет следующий формат:

```
until test
commands
do

other commands
done
```

Аналогично команде `while`, в инструкции команды `until` можно задавать несколько команд проверки *test commands*. Только статус выхода последней команды определяет, будет ли командный интерпретатор `bash` выполнять другие команды *other commands*, которые составляют цикл.

Пример использования команды `until` приведен ниже.

```
$ cat test12
#!/bin/bash
# #

var1=100

until [ $var1 -eq 0 ]
do
    echo $var1
    var1=$(( $var1 - 25 ])
done
$ ./test12
100
75
50
25
$
```

В рассматриваемом примере определяется переменная `var1`, которая позволяет узнать, когда должен остановиться цикл `until`. Как только значение этой переменной становится равным нулю, команда `until` останавливает цикл. При использовании команды `until` с несколькими командами проверки следует руководствоваться теми же предостережения, которые касаются команды `while`:

```
$ cat test13
#!/bin/bash
# #

var1=100

until echo $var1
```

```

        [ $var1 -eq 0 ]
do
    echo Inside the loop: $var1
    var1=$(( $var1 - 25 ))
done
$ ./test13
100
Inside the loop: 100
75
Inside the loop: 75
50
Inside the loop: 50
25
Inside the loop: 25
0
$

```

Командный интерпретатор выполняет указанные команды проверки и останавливается только тогда, когда возвращается истинное значение из последней команды.

Уровень вложенности циклов

В любой инструкции цикла могут быть заданы в составе цикла команды любого типа, включая другие команды организации цикла. В последнем случае создается так называемый *вложенный цикл*. При создании вложенных циклов необходимо соблюдать осторожность, поскольку в таком случае в ходе итераций происходят другие итерации, поэтому количество проходов по циклу многократно возрастает. Если этой проблеме не уделяется достаточное внимание, то в ходе выполнения сценариев могут происходить сбои.

Ниже приведен простой пример применения цикла `for`, вложенного в другой цикл `for`.

```

$ cat test14
#!/bin/bash
# вложение циклов for

for (( a = 1; a <= 3; a++ ))
do
    echo "Starting loop $a:"
    for (( b = 1; b <= 3; b++ ))
    do
        echo "    Inside loop: $b"
    done
done
$ ./test14
Starting loop 1:
    Inside loop: 1
    Inside loop: 2
    Inside loop: 3
Starting loop 2:
    Inside loop: 1
    Inside loop: 2
    Inside loop: 3

```

```
Starting loop 3:
  Inside loop: 1
  Inside loop: 2
  Inside loop: 3
$
```

Во вложенном цикле (называемом также *внутренним циклом*) происходит итерация по всем его значениям в каждой отдельной итерации внешнего цикла. Следует отметить, что в обоих этих циклах нет никаких различий между командами `do` и `done`. При выполнении первой команды `done` командный интерпретатор `bash` учитывает, что эта команда относится к внутреннему циклу, а не к внешнему.

То же правило соблюдается при формировании вложенных циклов с применением разных команд организации цикла, например, когда цикл `for` вкладывается в цикл `while`:

```
$ cat test15
#!/bin/bash
# размещение цикла for в цикле while

var1=5

while [ $var1 -ge 0 ]
do
  echo "Outer loop: $var1"
  for (( var2 = 1; $var2 < 3; var2++ ))
  do
    var3=$(( $var1 * $var2 ))
    echo "  Inner loop: $var1 * $var2 = $var3"
  done
  var1=$(( $var1 - 1 ))
done
$ ./test15
Outer loop: 5
  Inner loop: 5 * 1 = 5
  Inner loop: 5 * 2 = 10
Outer loop: 4
  Inner loop: 4 * 1 = 4
  Inner loop: 4 * 2 = 8
Outer loop: 3
  Inner loop: 3 * 1 = 3
  Inner loop: 3 * 2 = 6
Outer loop: 2
  Inner loop: 2 * 1 = 2
  Inner loop: 2 * 2 = 4
Outer loop: 1
  Inner loop: 1 * 1 = 1
  Inner loop: 1 * 2 = 2
Outer loop: 0
  Inner loop: 0 * 1 = 0
  Inner loop: 0 * 2 = 0
$
```

И в этом случае командный интерпретатор успешно отличает команды `do` и `done`, относящиеся к внутреннему циклу `for`, от тех же команд во внешнем цикле `while`.

Но настоящую проверку умственных способностей приходится проходить тем, кто пытается использовать сочетание циклов `until` и `while`:

```
$ cat test16
#!/bin/bash
# применение циклов until и while

var1=3

until [ $var1 -eq 0 ]
do
    echo "Outer loop: $var1"
    var2=1
    while [ $var2 -lt 5 ]
    do
        var3=`echo "scale=4; $var1 / $var2" | bc`
        echo "    Inner loop: $var1 / $var2 = $var3"
        var2=$(( $var2 + 1 ))
    done
    var1=$(( $var1 - 1 ))
done
$ ./test16
Outer loop: 3
    Inner loop: 3 / 1 = 3.0000
    Inner loop: 3 / 2 = 1.5000
    Inner loop: 3 / 3 = 1.0000
    Inner loop: 3 / 4 = .7500
Outer loop: 2
    Inner loop: 2 / 1 = 2.0000
    Inner loop: 2 / 2 = 1.0000
    Inner loop: 2 / 3 = .6666
    Inner loop: 2 / 4 = .5000
Outer loop: 1
    Inner loop: 1 / 1 = 1.0000
    Inner loop: 1 / 2 = .5000
    Inner loop: 1 / 3 = .3333
    Inner loop: 1 / 4 = .2500
$
```

Внешний цикл `until` начинается со значения 3 и продолжается до тех пор, пока это значение не достигнет 0. Внутренний цикл `while` начинается со значения 1 и продолжается, пока значение переменной остается меньше 5. В каждом цикле следует изменять значение, используемое в проверяемом условии, поскольку в противном случае цикл будет продолжаться до бесконечности.

Организация циклов на основе данных файла

В процессе работы часто возникает необходимость осуществлять итерации по элементам, которые хранятся в файле. Для этого необходимо применять сочетание двух рассматривавшихся ранее методик:

- использование вложенных циклов;
- изменение переменной среды IFS.

Изменяя значение переменной среды IFS, можно обеспечить обработку в команде `for` каждой строки в файле как отдельного элемента, даже если данные содержат пробелы. После извлечения отдельной строки из файла можно применить еще один цикл для извлечения данных, содержащихся в этой строке.

Классическим примером применения такого способа обработки данных является извлечение сведений из файла `/etc/passwd`. Для этого необходимо организовать построчную обработку файла `/etc/passwd` в цикле и после перехода к очередной строке изменять значение переменной IFS на двоеточие, что позволяет выделять отдельные компоненты в каждой строке.

Ниже приведен пример осуществления описанного способа обработки.

```
#!/bin/bash
# изменение значения IFS

IFS.OLD=$IFS
IFS=$'\n'
for entry in `cat /etc/passwd`
do
    echo "Values in $entry -"
    IFS=:
    for value in $entry
    do
        echo "    $value"
    done
done
$
```

В этом сценарии в ходе синтаксического анализа данных используются два различных значения IFS. Первое значение IFS позволяет выделять отдельные строки в файле `/etc/passwd`. А во внутреннем цикле `for` значение IFS изменяется на двоеточие, что позволяет проводить синтаксический разбор с выделением отдельных значений в строках файла `/etc/passwd`.

После запуска этого сценария на выполнение сформируется примерно такой вывод:

```
Values in rich:x:501:501:Rich Blum:/home/rich:/bin/bash -
rich
x
501
501
Rich Blum
/home/rich
/bin/bash
Values in katie:x:502:502:Katie Blum:/home/katie:/bin/bash -
katie
x
506
509
Katie Blum
/home/katie
/bin/bash
```

Во внутреннем цикле выделяется каждое отдельное значение из записи `/etc/passwd`. Этот способ также превосходно подходит для обработки данных в формате с разделителями-запятыми, который часто применяется для импорта данных из электронных таблиц.

Управление циклом

Впервые приступая к изучению тематики, связанной с циклами, многие считают, что после запуска цикла приходится терпеливо дожидаться, пока цикл полностью не завершит все свои итерации, независимо от сложившихся обстоятельств. Но дела обстоят иначе. Предусмотрено несколько команд, в том числе указанных ниже, которые помогают управлять тем, что происходит в цикле.

- Команда `break`.
- Команда `continue`.

Каждая из этих команд имеет разное назначение, когда дело касается управления работой цикла. В следующих разделах описано, как можно использовать эти команды, чтобы лучше организовать выполнение инструкций в составе цикла.

Команда `break`

Команда `break` предоставляет простой способ выхода из цикла в ходе его работы. Команду `break` можно использовать для выхода из цикла любого типа, включая циклы `until` и `while`.

Ниже описаны некоторые ситуации, в которых может применяться команда `break`. В настоящем разделе каждый из способов применения этой команды описан более подробно.

Выход из одинарного цикла

При выполнении команды `break` командный интерпретатор предпринимает попытку выйти из цикла, который обрабатывается в настоящее время:

```
$ cat test17
#!/bin/bash
# выход из цикла for

for var1 in 1 2 3 4 5 6 7 8 9 10
do
    if [ $var1 -eq 5 ]
    then
        break
    fi
    echo "Iteration number: $var1"
done
echo "The for loop is completed"
$ ./test17
Iteration number: 1
Iteration number: 2
Iteration number: 3
Iteration number: 4
The for loop is completed
$
```

При обычных обстоятельствах в цикле `for` должны быть проведены итерации по всем значениям, указанным в списке. Однако, если условие `if-then` становится истинным, командный интерпретатор выполняет команду `break`, которая останавливает цикл `for`.

Этот прием также применим для циклов `until` и `while`:

```
$ cat test18
#!/bin/bash
# выход из цикла while

var1=1

while [ $var1 -lt 10 ]
do
    if [ $var1 -eq 5 ]
    then
        break
    fi
    echo "Iteration: $var1"
    var1=$(( $var1 + 1 ))
done
echo "The while loop is completed"
$ ./test18
Iteration: 1
Iteration: 2
Iteration: 3
Iteration: 4
The while loop is completed
$
```

Цикл `while` завершается после выполнения условия `if-then` путем вызова команды `break`.

Выход из внутреннего цикла

Если работа программы организована по принципу применения нескольких циклов, то команда `break` автоматически завершает самый внутренний выполняемый цикл:

```
$ cat test19
#!/bin/bash
# выход из внутреннего цикла

for (( a = 1; a < 4; a++ ))
do
    echo "Outer loop: $a"
    for (( b = 1; b < 100; b++ ))
    do
        if [ $b -eq 5 ]
        then
            break
        fi
        echo "    Inner loop: $b"
    done
done
$ ./test19
Outer loop: 1
```

```

    Inner loop: 1
    Inner loop: 2
    Inner loop: 3
    Inner loop: 4
Outer loop: 2
    Inner loop: 1
    Inner loop: 2
    Inner loop: 3
    Inner loop: 4
Outer loop: 3
    Inner loop: 1
    Inner loop: 2
    Inner loop: 3
    Inner loop: 4
$

```

Инструкция `for` во внутреннем цикле указывает, что итерации должны продолжаться до тех пор, пока значение переменной `b` не станет равным 100. Но в этом внутреннем цикле имеется инструкция `if-then`, в соответствии с которой должна быть выполнена команда `break` после достижения переменной `b` значения 5. Заслуживает внимания то, что, несмотря на завершение внутреннего цикла с помощью команды `break`, внешний цикл продолжает работать в соответствии с тем, что указано.

Выход из внешнего цикла

Иногда при выполнении внутреннего цикла возникает необходимость остановить внешний цикл. Для этого в команде `break` предусмотрен отдельный параметр командной строки:

```
break n
```

где `n` указывает уровень цикла, из которого должен быть выполнен выход. По умолчанию `n` равно 1, а это означает, что выход должен быть выполнен из текущего цикла. Если будет задано значение `n`, равное 2, то команда `break` остановит выполнение внешнего цикла следующего уровня:

```

$ cat test20
#!/bin/bash
# выход из внешнего цикла

for (( a = 1; a < 4; a++ ))
do
    echo "Outer loop: $a"
    for (( b = 1; b < 100; b++ ))
    do
        if [ $b -gt 4 ]
        then
            break 2
        fi
        echo "    Inner loop: $b"
    done
done
$ ./test20
Outer loop: 1

```

```
Inner loop: 1
Inner loop: 2
Inner loop: 3
Inner loop: 4
$
```

Теперь после выполнения команды `break` командным интерпретатором происходит останов внешнего цикла.

Команда `continue`

Команда `continue` предоставляет способ преждевременного прекращения обработки части команд в цикле без полного завершения цикла. Эта команда позволяет задавать в цикле условия, при которых командный интерпретатор не будет выполнять некоторые команды. Ниже приведен простой пример использования команды `continue` в цикле `for`.

```
$ cat test21
#!/bin/bash
# применение команды continue

for (( var1 = 1; var1 < 15; var1++ ))
do
    if [ $var1 -gt 5 ] && [ $var1 -lt 10 ]
    then
        continue
    fi
    echo "Iteration number: $var1"
done
$ ./test21
Iteration number: 1
Iteration number: 2
Iteration number: 3
Iteration number: 4
Iteration number: 5
Iteration number: 10
Iteration number: 11
Iteration number: 12
Iteration number: 13
Iteration number: 14
$
```

Если выполняются условия инструкции `if-then` (значение становится больше 5 и меньше 10), командный интерпретатор выполняет команду `continue`, пропускает оставшуюся часть команд в цикле, но не прекращает работу цикла. После того как условие `if-then` перестает быть истинным, возобновляется нормальное функционирование цикла.

Команду `continue` можно использовать также в циклах `while` и `until`, но при этом необходимо соблюдать предельную осмотрительность при разработке сценария. Напомним, что при выполнении командным интерпретатором команды `continue` пропускаются все команды, оставшиеся в цикле. Если одна из этих команд применяется для увеличения значения переменной, проверяемой в условии, то возникают существенные нарушения в работе:

```
$ cat badtest3
#!/bin/bash
```


где *n* определяет уровень цикла, на котором должно происходить продолжение работы цикла без выполнения оставшихся операторов. Ниже приведен пример пропуска команд во внешнем цикле `for`.

```
$ cat test22
#!/bin/bash
# продолжение внешнего цикла

for (( a = 1; a <= 5; a++ ))
do
    echo "Iteration $a:"
    for (( b = 1; b < 3; b++ ))
    do
        if [ $a -gt 2 ] && [ $a -lt 4 ]
        then
            continue 2
        fi
        var3=$(( $a * $b ))
        echo "    The result of $a * $b is $var3"
    done
done
$ ./test22
Iteration 1:
    The result of 1 * 1 is 1
    The result of 1 * 2 is 2
Iteration 2:
    The result of 2 * 1 is 2
    The result of 2 * 2 is 4
Iteration 3:
Iteration 4:
    The result of 4 * 1 is 4
    The result of 4 * 2 is 8
Iteration 5:
    The result of 5 * 1 is 5
    The result of 5 * 2 is 10
$
```

В следующей инструкции `if-then`:

```
if [ $a -gt 2 ] && [ $a -lt 4 ]
then
    continue 2
fi
```

команда `continue` используется для прекращения обработки команд в текущем цикле и продолжения внешнего цикла. Обратите внимание на то, что согласно выводу сценария в итерации со значением 3 не обрабатываются какие-либо инструкции внутреннего цикла, поскольку команда `continue` остановила эту обработку, но продолжается обработка внешнего цикла.

Обработка вывода в цикле

Наконец, в сценарии командного интерпретатора можно либо передать вывод из цикла по каналу, либо перенаправить его. Для этого необходимо добавить команды обработки к концу команды `done`:

```
for file in /home/rich*
do
    if [ -d "$file" ]
    then
        echo "$file is a directory"
    elif
        echo "$file is a file"
    fi
done > output.txt
```

Вместо отображения результатов на мониторе командный интерпретатор перенаправит результаты команды `for` в файл `output.txt`.

Рассмотрим следующий пример перенаправления вывода команды `for` в файл:

```
$ cat test23
#!/bin/bash
# перенаправление вывода цикла for в файл

for (( a = 1; a < 10; a++ ))
do
    echo "The number is $a"
done > test23.txt
echo "The command is finished."
$ ./test23
The command is finished.
$ cat test23.txt
The number is 1
The number is 2
The number is 3
The number is 4
The number is 5
The number is 6
The number is 7
The number is 8
The number is 9
$
```

Командный интерпретатор создает файл `test23.txt` и перенаправляет вывод команды `for` только в файл. Командный интерпретатор отображает инструкцию `echo`, стоящую после команды `for`, как обычно.

Тот же принцип может применяться для отправки вывода цикла по каналу в другую команду:

```
$ cat test24
#!/bin/bash
# перенаправление из цикла по каналу в другую команду

for state in "North Dakota" Connecticut Illinois Alabama Tennessee
```

```
do
    echo "$state is the next place to go"
done | sort
echo "This completes our travels"
$ ./test24
Alabama is the next place to go
Connecticut is the next place to go
Illinois is the next place to go
North Dakota is the next place to go
Tennessee is the next place to go
This completes our travels
$
```

Значения состояния не перечисляются в выводе команды `for` в каком-либо определенном порядке. Вывод команды `for` передается по каналу в команду `sort`, которая упорядочивает последовательность вывода команды `for`. Результаты выполнения этого сценария становятся подтверждением того, что вывод был действительно отсортирован должным образом в сценарии.

Резюме

Задача организации циклов представляет собой одну из неотъемлемых составляющих программирования. В командном интерпретаторе `bash` предусмотрены три различные команды работы с циклами, которые могут использоваться в сценариях. Команда `for` позволяет выполнять итерации по списку значений, будь то значения, заданные в командной строке, содержащиеся в переменной или полученные путем подстановки имен файлов (для определения имен файлов и каталогов с помощью символа-заместителя).

Команда `while` предоставляет возможность организовать циклы с учетом результатов выполнения команды, для чего используется либо обычная команда, либо команда `test`, позволяющая проверять условия, которые заданы с помощью переменных. До тех пор пока в команде (или в условии) вырабатывается нулевой статус выхода, цикл `while` продолжает производить итерации по указанному набору команд.

Команда `until` также предоставляет метод осуществления итераций по ряду команд, но эти итерации продолжаются, если команда (или условие) вырабатывает ненулевой статус выхода. Это средство позволяет задать условие, которое должно выполняться, пока не произойдет прекращение итераций.

В сценариях командного интерпретатора циклы могут применяться в различных сочетаниях, что приводит к созданию многоуровневых циклов. В командном интерпретаторе `bash` предусмотрены команды `continue` и `break`, которые позволяют изменять обычный поток выполнения цикла с учетом изменения в цикле значений `tex` или иных переменных.

Кроме того, командный интерпретатор `bash` позволяет использовать стандартные средства перенаправления и передачи по каналу вывода команд для изменения обычного назначения вывода цикла. Предусмотрена возможность использовать перенаправление для отправки вывода цикла в файл или передачу данных по каналу для отправки вывода цикла в команду. Благодаря этому существенно увеличивается разнообразие средств, позволяющих контролировать выполнение сценариев командного интерпретатора.

В следующей главе рассматривается вопрос о том, как организовать взаимодействие с пользователем сценария командного интерпретатора. Зачастую невозможно обеспечить полностью автономную работу сценариев командного интерпретатора. В таких случаях для сценария требуются внешние данные того или иного типа, которые должны быть заданы во время его выполнения. В следующей главе обсуждаются различные методы, позволяющие предоставлять в реальном времени данные для обработки в сценариях командного интерпретатора.

ГЛАВА

13

В этой главе...

Параметры командной строки

Специальные переменные параметров

Применение сдвига

Работа с опциями

Стандартизация параметров

Получение ввода данных от пользователя

Резюме

Обработка ввода данных пользователем

В предыдущих главах были приведены сведения о том, как писать сценарии, которые взаимодействуют с данными, переменными и файлами в системе Linux. Но иногда возникает необходимость в написании сценария, который должен взаимодействовать с лицом, выполняющим сценарий. Командный интерпретатор `bash` предоставляет несколько различных методов получения данных от пользователей, включая параметры командной строки (значения данных, добавленные после команды), опции командной строки (односимвольные значения, которые изменяют поведение команды), а также возможность считывать ввод непосредственно с клавиатуры. В настоящей главе описано, как включить эти разные методы в сценарии командного интерпретатора `bash` для получения данных от лица, выполняющего конкретный сценарий.

Параметры командной строки

Наиболее простой способ передачи данных в сценарий командного интерпретатора состоит в использовании *параметров командной строки*. Параметры командной строки позволяют дополнительно задавать значения данных в командной строке при выполнении сценария:

```
$ ./addem 10 30
```

В этом примере в сценарий `addem` передаются два параметра командной строки (10 и 30). Обработка параметров командной строки в сценарии проводится с использованием специальных переменных. В следующих разделах будет описано, как использовать параметры командной строки в сценариях командного интерпретатора `bash`.

Чтение параметров

Командный интерпретатор `bash` присваивает специальные переменные, называемые *позиционными параметрами*, всем параметрам, введенным в командной строке. В число этих параметров включено даже имя программы, выполняемой командным интерпретатором. Позиционные переменные параметров представляют собой стандартные числовые обозначения: `$0` указывает имя программы, `$1` соответствует первому параметру, `$2` — второму параметру и так далее, до обозначения `$9`, соответствующего девятому параметру.

Ниже приведен простой пример использования одного параметра командной строки в сценарии командного интерпретатора.

```
$ cat test1
#!/bin/bash
# использование одного параметра командной строки

factorial=1
for (( number = 1; number <= $1 ; number++ ))
do
    factorial=$(( $factorial * $number ))
done
echo The factorial of $1 is $factorial
$
$ ./test1 5
The factorial of 5 is 120
$
```

Переменную `$1` можно использовать точно так же, как и любую другую переменную в сценарии командного интерпретатора. Сценарий командного интерпретатора автоматически присваивает переменной значение из параметра командной строки; пользователь не должен для этого выполнять какие-либо действия.

Если потребуется ввести дополнительные параметры командной строки, то их следует разделить в командной строке пробелами:

```
$ cat test2
#!/bin/bash
# проверка с двумя параметрами командной строки

total=$(( $1 * $2 ))
echo The first parameter is $1.
echo The second parameter is $2.
echo The total value is $total.
$
$ ./test2 2 5
The first parameter is 2.
The second parameter is 5.
The total value is 10.
$
```

Командный интерпретатор присваивает каждый параметр соответствующей переменной.

В этом примере оба применявшихся параметра командной строки представляли собой числовые значения. В командной строке можно также использовать текстовые строки:

```
$ cat test3
#!/bin/bash
# проверка строковых параметров

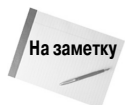
echo Hello $1, glad to meet you.
$
$ ./test3 Rich
Hello Rich, glad to meet you.
$
```

Командный интерпретатор передает строковое значение, введенное в командной строке, в сценарий. Но при попытке передать параметр в виде текстовой строки, содержащей пробелы, возникают проблемы:

```
$ ./test3 Rich Blum
Hello Rich, glad to meet you.
$
```

Напомним, что отдельные параметры рассматриваются как разделенные пробелами, поэтому в командном интерпретаторе этот пробел был принят за признак разделителя двух значений. Для включения пробелов в значение параметра необходимо использовать кавычки (одинарные или двойные):

```
$ ./test3 'Rich Blum'
Hello Rich Blum, glad to meet you.
$
$ ./test3 "Rich Blum"
Hello Rich Blum, glad to meet you.
$
```



Кавычки, используемые при передаче текстовых строк в качестве параметров, не рассматриваются как часть данных. Они просто обозначают начало и конец данных.

Если в сценарии потребуется больше девяти параметров командной строки, то можно продолжить их ввод, но в таком случае имена переменных немного изменятся. После девятой переменной необходимо задавать фигурные скобки вокруг номера переменной, как в примере `${10}`. Ниже показано, как этот способ обозначения переменных применяется в сценарии.

```
$ cat test4
#!/bin/bash
# обработка большого количества параметров

total=$(( ${10} * ${11} ])
echo The tenth parameter is ${10}
echo The eleventh parameter is ${11}
echo The total is $total
$
$ ./test4 1 2 3 4 5 6 7 8 9 10 11 12
The tenth parameter is 10
```

```
The eleventh parameter is 11
The total is 110
$
```

Данный метод позволяет вводить такое количество параметров командной строки для сценариев, которое может потребоваться на практике.

Чтение имени программы

Для определения имени программы, запущенной командным интерпретатором из командной строки, можно использовать параметр `$0`. Необходимость в этом возникает, например, при написании программы, которая может выполнять несколько функций. Однако при этом возникает небольшая проблема, которую приходится учитывать на практике. Рассмотрим, что происходит в следующем простом примере:

```
$ cat test5
#!/bin/bash
# проверка с применением параметра $0

echo The command entered is: $0
$
$ ./test5
The command entered is: ./test5
$
$ /home/rich/test5
The command entered is: /home/rich/test5
$
```

Если строкой, фактически переданной в переменную `$0`, является весь путь к сценарию, то для указания на программу используется полное обозначение пути, а не только имя самой программы.

Если должен быть написан сценарий, который выполняет различные функции с учетом того, под каким именем этот сценарий вызван из командной строки, то необходимо дополнительно проделать небольшую работу. Требуется удалить сведения о пути, который использовался для вызова сценария из командной строки.

К счастью, предусмотрена небольшая и удобная команда, которая предназначена именно для этого. Команда `basename` возвращает только имя программы без обозначения пути. Вне-сем изменения в пример сценария и рассмотрим, как он будет работать после этого:

```
$ cat test5b
#!/bin/bash
# применение команды basename в сочетании с параметром $0

name=`basename $0`
echo The command entered is: $name
$
$ ./test5b
The command entered is: test5b
$
$ /home/rich/test5b
The command entered is: test5b
$
```

Теперь полученный результат намного лучше! В дальнейшем описанную методику читатель сможет использовать для написания сценариев, которые выполняют различные функции в зависимости от того, под каким именем вызван этот сценарий. Ниже приведен простой пример, который показывает применение этой методики на практике.

```
$ cat test6
#!/bin/bash
# проверка многофункционального сценария

name='basename $0'

if [ $name = "addem" ]
then
    total=$(( $1 + $2 ))
elif [ $name = "multem" ]
then
    total=$(( $1 * $2 ))
fi
echo The calculated value is $total
$
$ chmod u+x test6
$ cp test6 addem
$ ln -s test6 multem
$ ls -l
-rwxr--r--  1 rich      rich    211 Oct 15 18:00 addem
lrwxrwxrwx  1 rich      rich      5 Oct 15 18:01 multem -> test6
-rwxr--r--  1 rich      rich    211 Oct 15 18:00 test6
$
$ ./addem 2 5
The calculated value is 7
$
$ ./multem 2 5
The calculated value is 10
$
```

В этом примере создаются два отдельных файла с разными именами на основе кода сценария test6; в одном из этих вариантов только копируется файл, а во втором используется ссылка для создания нового файла. В обоих случаях сценарий определяет базовое имя сценария и выполняет соответствующую функцию с учетом этого значения.

Проверка параметров

При использовании параметров командной строки в сценариях командного интерпретатора необходимо соблюдать осторожность. Если сценарий, который определен с параметрами, запускается без параметров, то могут происходить нарушения в работе:

```
$ ./addem 2
./addem: line 8: 2 + : syntax error: operand expected (error
token is " ")
The calculated value is
$
```


В частности, если в сценарии предполагается, что переменная параметра содержит данные, а этих данных фактически нет, то, вероятнее всего, будет получено сообщение об ошибке от конкретного сценария. Ниже приведен пример чреватого ошибками подхода к написанию сценариев. Рекомендуется всегда проверять параметры, чтобы убедиться в том, что они содержат данные, и только после этого их использовать.

```
$ cat test7
#!/bin/bash
# проверка параметров перед их использованием

if [ -n "$1" ]
then
    echo Hello $1, glad to meet you.
else
    echo "Sorry, you did not identify yourself. "
fi
$
$ ./test7 Rich
Hello Rich, glad to meet you.
$
$ ./test7
Sorry, you did not identify yourself.
$
```

В данном примере параметр `-n` применялся в команде `test` для проверки того, имелись ли данные в параметре командной строки. В следующем разделе будет показано, что есть еще один способ проверки параметров командной строки.

Специальные переменные параметров

В командном интерпретаторе `bash` предусмотрено несколько специальных переменных, которые позволяют отслеживать параметры командной строки. В настоящем разделе описано, что представляют собой эти переменные и как они используются.

Подсчет параметров

Как было показано в предыдущем разделе, чаще всего бывает оправдана рекомендация проверять параметры командной строки перед их использованием в сценарии. Но если в сценарии используется большое количество параметров командной строки, то выполнение этой рекомендации может стать трудоемким.

Иногда вместо проверки каждого параметра можно просто подсчитать, какое количество параметров было введено в командной строке. В этих целях в командном интерпретаторе `bash` предусмотрена специальная переменная.

Специальная переменная `$#` содержит значение количества параметров командной строки, заданных при вызове сценария на выполнение. Эту специальную переменную можно использовать в любом месте сценария, точно так же, как и обычную переменную:

```
$ cat test8
#!/bin/bash
# получение числа параметров
```

```

echo There were $# parameters supplied.
$
$ ./test8
There were 0 parameters supplied.
$
$ ./test8 1 2 3 4 5
There were 5 parameters supplied.
$
$ ./test8 1 2 3 4 5 6 7 8 9 10
There were 10 parameters supplied.
$
$ ./test8 "Rich Blum"
There were 1 parameters supplied.
$

```

Теперь воспользуемся возможностью проверить количество представленных параметров, прежде чем предпринимать попытку их использовать:

```

$ cat test9
#!/bin/bash
# проверка параметров

if [ $# -ne 2 ]
then
    echo Usage: test9 a b
else
    total=$(( $1 + $2 ))
    echo The total is $total
fi
$
$ ./test9
Usage: test9 a b
$
$ ./test9 10
Usage: test9 a b
$
$ ./test9 10 15
The total is 25
$
$ ./test9 10 15 20
Usage: test9 a b
$

```

В инструкции `if-then` применяется команда `test` для выполнения числовой проверки количества параметров, заданных в командной строке. Если в командной строке нет требуемого количества параметров, то можно вывести сообщение об ошибке, в котором показано, как правильно использовать сценарий.

Кроме того, эта переменная предоставляет превосходный способ захвата последнего параметра из командной строки, который не требует специально определять, сколько параметров было задано. Но, чтобы воспользоваться такой возможностью, необходимо пойти на небольшую хитрость.

Пытаясь сходу решить эту задачу, можно подумать, что для получения значения последней переменной параметра командной строки достаточно задать переменную `${$#}`, поскольку переменная `$#` содержит данные о количестве параметров. Предпримем попытку воспользоваться этим подходом и рассмотрим, что происходит:

```
$ cat badtest1
#!/bin/bash
# проверка одного способа получения последнего параметра

echo The last parameter was ${$#}
$
$ ./badtest1 10
The last parameter was 15354
$
```

Но что же случилось? Очевидно, произошло нечто непредвиденное. Как оказалось, знак доллара нельзя использовать в фигурных скобках. Вместо этого необходимо заменить знак доллара восклицательным знаком. Как ни странно, теперь все работает:

```
$ cat test10
#!/bin/bash
# получение последнего параметра

params=$#
echo The last parameter is $params
echo The last parameter is ${!#}
$
$ ./test10 1 2 3 4 5
The last parameter is 5
The last parameter is 5
$
$ ./test10
The last parameter is 0
The last parameter is ./test10
$
```

Просто идеальный результат. В проведенной проверке также присваивалось значение переменной `$#` переменной `params`, а затем эта переменная применялась в рамках специального формата переменной параметра командной строки. Обе версии являются работоспособными. Важно также отметить, что при отсутствии параметров в командной строке переменная `$#` получает нулевое значение, которое присваивается переменной `params`, поэтому переменная `${!#}` возвращает имя сценария, заданное в командной строке.

Захват всех данных

В некоторых ситуациях может потребоваться просто захватить все параметры, заданные в командной строке, и провести итерации по всем этим параметрам. В таком случае вместо применения трудоемкого способа, который предусматривает использование переменной `$#` для определения количества параметров в командной строке с последующей обработкой в цикле всех этих параметров, можно использовать несколько других специальных переменных.

Переменные `$*` и `$@` предоставляют удобный доступ ко всем параметрам. Обе эти переменные включают все параметры командной строки в виде одной переменной.

Переменная `$*` принимает все параметры, заданные в командной строке, как одно слово. В этом слове содержится каждое из значений в той последовательности, в которой они приведены в командной строке. По существу, переменная `$*` позволяет рассматривать параметры не как несколько объектов, а как один параметр.

С другой стороны, переменная `$@` воспринимает все параметры, заданные в командной строке, как отдельные слова в одной и той же строке. Она позволяет проводить итерации по полученному значению, выделяя каждый из заданных параметров. Для этого чаще всего используется команда `for`.

Однако можно легко запутаться, пытаясь разобраться в том, как работают эти две переменные. Рассмотрим, каким образом можно провести между ними различие:

```
$ cat test11
#!/bin/bash
# проверка переменных $* и @$

echo "Using the \$* method: $*"
echo "Using the @$ method: @$"
$
$ ./test11 rich barbara katie jessica
Using the $* method: rich barbara katie jessica
Using the @$ method: rich barbara katie jessica
$
```

Отметим, что на первый взгляд при использовании обеих переменных вырабатывается одинаковый вывод, который показывает сразу все предоставленные параметры командной строки.

А следующий пример показывает, в чем возникают различия:

```
$ cat test12
#!/bin/bash
# проверка переменных $* и @$

count=1
for param in "$*"
do
    echo "\$* Parameter #$count = $param"
    count=$(( $count + 1 ))
done

count=1
for param in "$@"
do
    echo "\@$ Parameter #$count = $param"
    count=$(( $count + 1 ))
done
$
$ ./test12 rich barbara katie jessica
$* Parameter #1 = rich barbara katie jessica
@$ Parameter #1 = rich
@$ Parameter #2 = barbara
@$ Parameter #3 = katie
@$ Parameter #4 = jessica
$
```

Теперь стало известно, с чего нужно начинать. Необходимо воспользоваться командой `for` для итерации по значениям специальных переменных, чтобы узнать, в чем состоят различия в представлениях параметров командной строки каждой из этих переменных. В переменной `$*` все параметры рассматриваются как один параметр, а в переменной `$@` каждый параметр представлен отдельно. Последний вариант предоставляет превосходный способ проведения итераций по параметрам командной строки.

Применение сдвига

Еще одним инструментом, который предоставляет командный интерпретатор `bash` в распоряжение пользователя, является команда `shift`. Назначение команды `shift`, поддерживаемой командным интерпретатором `bash`, состоит в том, что она упрощает управление параметрами командной строки. Команда `shift` в соответствии со своим смысловым значением (сдвиг) буквально смещает параметры командной строки, изменяя их относительные положения.

После каждого выполнения команды `shift` по умолчанию происходит перемещение каждой переменной параметра вниз на одну позицию. Таким образом, значение переменной `$3` перемещается в переменную `$2`, значение переменной `$2` — в переменную `$1`, а значение переменной `$1` отбрасывается (обратите внимание на то, что значение переменной `$0`, которая указывает имя программы, остается неизменным).

Применение указанной команды представляет собой еще один удобный способ итерации по параметрам командной строки, особенно если невозможно узнать заранее количество этих параметров. Достаточно просто провести обработку первого параметра, применить к параметрам команду сдвига, а затем снова обработать первый параметр.

Ниже приведен краткий пример осуществления описанного принципа работы.

```
$ cat test13
#!/bin/bash
# демонстрация применения команды shift

count=1
while [ -n "$1" ]
do
    echo "Parameter #$count = $1"
    count=$(( $count + 1 ))
    shift
done
$
$ ./test13 rich barbara katie jessica
Parameter #1 = rich
Parameter #2 = barbara
Parameter #3 = katie
Parameter #4 = jessica
$
```

В этом сценарии выполняется цикл `while`, в ходе которого проверяется длина значения первого параметра. После того как длина значения первого параметра становится равной нулю, цикл заканчивается.

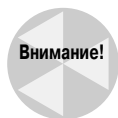
После проверки первого параметра применяется команда `shift` для сдвига всех параметров на одну позицию.

Еще один вариант состоит в том, чтобы проводить смещение на несколько позиций, задавая в команде `shift` соответствующий параметр. Достаточно просто указать количество позиций, на которое должны быть сдвинуты параметры:

```
$ cat test14
#!/bin/bash
# демонстрация сдвига на несколько позиций

echo "The original parameters: $*"
shift 2
echo "Here's the new first parameter: $1"
$
$ ./test14 1 2 3 4 5
The original parameters: 1 2 3 4 5
Here's the new first parameter: 3
$
```

Задавая значения в команде `shift`, можно легко обеспечить пропуск ненужных параметров.



В ходе работы с командой `shift` необходимо соблюдать осторожность. После сдвига параметра за пределы списка параметров его значение теряется и не может быть восстановлено.

Работа с опциями

Данные, задаваемые в командной строке команды `bash`, состоят из параметров и опций. Примеры применения тех и других можно найти во многих командах, которые были приведены в настоящей книге. *Опции* — это однобуквенные обозначения, которым предшествуют тире, предназначенные для изменения поведения команды. В этом разделе показаны три различных метода работы с опциями в сценариях командного интерпретатора.

Поиск опций

На первый взгляд кажется, что в работе с опциями командной строки не должны возникать какие-либо сложности. Опции должны быть заданы в командной строке непосредственно после имени сценария, как и параметры командной строки. И действительно, по желанию можно организовать обработку опций командной строки таким же способом, с помощью которого обрабатываются параметры командной строки.

Обработка простых опций

В приведенном выше сценарии `test13` было показано, как использовать команду `shift` для обработки параметров командной строки, заданных при вызове сценария. Тот же метод можно использовать для обработки опций командной строки.

Для этого после извлечения каждого отдельного параметра достаточно применить инструкцию `case` для определения того, не отформатирован ли параметр как опция:

```
$ cat test15
#!/bin/bash
# извлечение опций командной строки в качестве параметров
```

```

while [ -n "$1" ]
do
    case "$1" in
        -a) echo "Found the -a option" ;;
        -b) echo "Found the -b option" ;;
        -c) echo "Found the -c option" ;;
        *) echo "$1 is not an option" ;;
    esac
    shift
done
$
$ ./test15 -a -b -c -d
Found the -a option
Found the -b option
Found the -c option
-d is not an option
$

```

В данном случае в инструкции `case` проверяется каждый параметр на наличие допустимых опций. Если таковая обнаруживается, то в инструкции `case` выполняются соответствующие команды.

Этот метод функционирует независимо от того, в каком порядке опции представлены в командной строке:

```

$ ./test15 -d -c -a
-d is not an option
Found the -c option
Found the -a option
$

```

В инструкции `case` обрабатывается каждая опция, найденная среди параметров командной строки. Если в командной строке могут появляться другие параметры, то следует включить команды для их обработки в ту часть инструкции `case`, которая представляет собой универсальную ловушку.

Разделение опций и параметров

Часто возникают такие ситуации, в которых приходится задавать для сценария командного интерпретатора и опции, и параметры. Стандартный способ решения этой задачи в Linux состоит в отделении одних от других с помощью специального символического кода, который служит для командного интерпретатора указанием на то, что в вызове сценария опции закончились и начинаются обычные параметры.

В Linux в качестве этого специального символа применяется двойное тире (`--`). В командном интерпретаторе двойное тире используется как признак, обозначающий конец списка опций. После обнаружения двойного тире в сценарии можно без опасений приступить к обработке оставшихся параметров командной строки в качестве обычных параметров, а не опций.

Чтобы обеспечить проверку на наличие двойного тире, достаточно добавить еще одну запись в инструкцию `case`:

```

$ cat test16
#!/bin/bash
# извлечение опций и параметров

```

```

while [ -n "$1" ]
do
    case "$1" in
        -a) echo "Found the -a option" ;;
        -b) echo "Found the -b option" ;;
        -c) echo "Found the -c option" ;;
        --) shift
            break ;;
        *) echo "$1 is not an option" ;;
    esac
    shift
done

count=1
for param in $@
do
    echo "Parameter #$count: $param"
    count=$((count + 1))
done
$

```

В этом сценарии для разрыва цикла `while` после обнаружения двойного тире используется команда `break`. При такой организации работы прерывание цикла всегда происходит преждевременно, поэтому необходимо обеспечить применение еще одной команды `shift` для получения двойного тире из переменных параметра.

Для первой проверки попытаемся вызвать сценарий на выполнение с использованием обычного набора опций и параметров:

```

$ ./test16 -c -a -b test1 test2 test3
Found the -c option
Found the -a option
Found the -b option
test1 is not an option
test2 is not an option
test3 is not an option
$

```

Полученные результаты показывают, что сценарий выполнялся на основании предположения, что все параметры командной строки ко времени их обработки представляли собой опции. Затем попытаемся сделать то же самое, но на этот раз воспользуемся двойным тире для отделения опций от параметров в командной строке:

```

$ ./test16 -c -a -b -- test1 test2 test3
Found the -c option
Found the -a option
Found the -b option
Parameter #1: test1
Parameter #2: test2
Parameter #3: test3
$

```


После того как в сценарии достигается двойное тире, в нем прекращается обработка опций и принимается предположение, что все оставшиеся параметры представляют собой параметры командной строки.

Обработка опций со значениями

Для некоторых опций требуется дополнительное значение параметра. В подобных ситуациях командная строка должна выглядеть примерно так:

```
$ ./testing -a test1 -b -c -d test2
```

В сценарий необходимо заложить способность обнаруживать, что та или иная опция командной строки требует дополнительного параметра, и проводить обработку в этих ситуациях должным образом. Ниже приведен пример того, как можно решить эту задачу:

```
$ cat test17
#!/bin/bash
# извлечение из командной строки опций и значений

while [ -n "$1" ]
do
    case "$1" in
        -a) echo "Found the -a option";;
        -b) param="$2"
            echo "Found the -b option, with parameter value $param"
            shift 2;;
        -c) echo "Found the -c option";;
        --) shift
            break;;
        *) echo "$1 is not an option";;
    esac
    shift
done

count=1
for param in "$@"
do
    echo "Parameter #$count: $param"
    count=$((count + 1))
done
$
$ ./test17 -a -b test1 -d
Found the -a option
Found the -b option, with parameter value test1
-d is not an option
$
```

В этом примере инструкция `case` определяет три параметра, которые в ней обрабатываются. Кроме того, в опции `-b` требуется дополнительное значение параметра. Поскольку обрабатываемым параметром является параметр `$1`, известно, что дополнительное значение параметра находится в позиции `$2` (поскольку все параметры смещаются после их обработки). Таким образом, в этом случае достаточно извлечь значение параметра из переменной `$2`. Безусловно, при обработке этой опции использовались две позиции задания параметров, поэтому и в команде `shift` должен быть выполнен сдвиг на две позиции.

По аналогии с тем, что происходит при использовании параметров в основной форме, этот процесс действует независимо от того, в какой последовательности размещаются опции (достаточно лишь не забывать включать соответствующий параметр опции с каждой опцией):

```
$ ./test17 -b test1 -a -d
Found the -b option, with parameter value test1
Found the -a option
-d is not an option
$
```

Итак, к этому моменту описаны основные особенности обработки опций командной строки в сценариях командного интерпретатора, но необходимо ознакомиться и с некоторыми ограничениями. Например, описанным способом невозможно воспользоваться, если окажется, что несколько опций объединены в одном параметре:

```
$ ./test17 -ac
-ac is not an option
$
```

В Linux широко распространена такая практика, что опции не задаются каждая отдельно, а объединяются, поэтому, чтобы обеспечить удобство работы пользователей со своими сценариями, каждый разработчик также должен соблюдать это условие в выпускаемых им программных продуктах. К счастью, предусмотрен еще один метод обработки опций и параметров, который можно взять на вооружение.

Использование команды `getopt`

Команда `getopt` представляет собой превосходный инструмент, которым можно воспользоваться при обработке опций и параметров командной строки. Эта команда реорганизует параметры командной строки так, чтобы их синтаксический анализ в сценарии стал проще.

Формат команды

Команда `getopt` может принимать список опций и параметров командной строки в любой форме и автоматически преобразовывать их в надлежащий формат. В команде `getopt` используется следующий формат команды:

```
getopt options optstring parameters
```

Ключом к правильной организации процесса является параметр *optstring*. Этот параметр определяет допустимые символы опций, которые могут быть заданы в командной строке. Кроме того, данный параметр определяет, для каких символов опций требуются значения параметров.

Прежде всего необходимо перечислить все буквы опций командной строки, которые предназначены для использования в сценарии, с помощью параметра *optstring*. Затем после каждой буквы опции, которая требует значения параметра, размещается двоеточие. Команда `getopt` проводит синтаксический анализ предоставленных параметров на основе заданного значения параметра *optstring*.

Ниже приведен простой пример применения команды `getopt`.

```
$ getopt ab:cd -a -b test1 -cd test2 test3
-a -b test1 -c -d -- test2 test3
$
```

В параметре *optstring* определены четыре допустимых символа опций, a, b, c и d. Кроме того, в этом параметре определено, что для опции, обозначенной буквой b, требуется значение параметра. Команда *getopt* в ходе выполнения проверяет представленный список параметров и проводит его синтаксический анализ с учетом заданного значения *optstring*. Заслуживает внимания то, что команда *getopt* автоматически разделяет объединенный параметр *-cd* на два отдельных параметра и вставляет двойное тире, чтобы отделить дополнительные параметры в строке.

При обнаружении опции, не указанной в параметре *optstring*, команда *getopt* по умолчанию выдает сообщение об ошибке:

```
$ getopt ab:cd -a -b test1 -cde test2 test3
getopt: invalid option -- e
-a -b test1 -c -d -- test2 test3
$
```

Если предпочтительным является просто пропускать сообщения об ошибке, то можно воспользоваться опцией *-q* при вызове команды обработки опций:

```
$ getopt -q ab:cd -a -b test1 -cde test2 test3
-a -b 'test1' -c -d -- 'test2' 'test3'
$
```

Обратите внимание на то, что параметры команды *getopt* должны быть заданы перед параметром *optstring*. Теперь мы можем приступить к использованию этой команды в сценариях для обработки опций командной строки.

Использование команды *getopt* в сценариях

Команду *getopt* можно использовать в сценариях для форматирования всех опций и параметров командной строки, заданных на входе в сценарий. Однако при ее использовании необходимо учитывать некоторые нюансы.

В частности, должно быть учтено, что существующие опции и параметры командной строки заменяются отформатированной версией, вырабатываемой командой *getopt*. Для этого следует использовать команду *set*.

Команда *set* уже рассматривалась в главе 5. В частности, отметим, что команда *set* применяется для работы с различными переменными в командном интерпретаторе. В главе 5 было показано, как использовать команду *set* для вывода на экран всех значений системных переменных среды.

Одним из параметров команды *set* является двойное тире, которое служит для этой команды указанием, что переменные параметров командной строки должны быть заменены значениями из командной строки команды *set*.

После этого необходимо применить такой прием: передать исходные параметры командной строки сценария в команду *getopt*, а затем отправить вывод команды *getopt* в команду *set* для замены исходных параметров командной строки теми параметрами, которые были отформатированы должным образом с помощью команды *getopt*. Соответствующая последовательность действий выглядит примерно так:

```
set -- 'getopts -q ab:cd "$@"'
```

Таким образом, исходные значения переменных параметров командной строки заменяются выводом команды *getopt*, которая форматирует параметры командной строки в интересах пользователя.

Эта методика позволяет писать сценарии, в которых применяются результаты автоматической обработки параметров командной строки:

```
$ cat test18
#!/bin/bash
# извлечение опций и значений из командной строки с помощью
# команды getopt

set -- 'getopt -q ab:c "$@"'
while [ -n "$1" ]
do
    case "$1" in
        -a) echo "Found the -a option" ;;
        -b) param="$2"
            echo "Found the -b option, with parameter value $param"
            shift ;;
        -c) echo "Found the -c option" ;;
        --) shift
            break;;
        *) echo "$1 is not an option";;
    esac
    shift
done

count=1
for param in "$@"
do
    echo "Parameter #$count: $param"
    count=$((count + 1))
done
$
```

Можно заметить, что этот сценарий по существу является тем же, что и сценарий test17. Единственное изменение заключается в том, что дополнительно введена команда `getopt`, которая помогает правильно отформатировать параметры командной строки.

После этого вызов сценария на выполнение с объединенными опциями приводит к получению намного лучших результатов:

```
$ ./test18 -ac
Found the -a option
Found the -c option
$
```

Кроме того, разумеется, все первоначально предусмотренные средства продолжают функционировать столь же успешно:

```
$ ./test18 -a -b test1 -cd test2 test3 test4
Found the -a option
Found the -b option, with parameter value 'test1'
Found the -c option
Parameter #1: 'test2'
Parameter #2: 'test3'
Parameter #3: 'test4'
$
```

Теперь дела внешне обстоят много лучше. Однако при использовании команды `getopt` необходимо учитывать возможность возникновения еще одной небольшой ошибки. Рассмотрим следующий пример:

```
$ ./test18 -a -b test1 -cd "test2 test3" test4
Found the -a option
Found the -b option, with parameter value 'test1'
Found the -c option
Parameter #1: 'test2'
Parameter #2: 'test3'
Parameter #3: 'test4'
$
```

Команда `getopt` плохо справляется с обработкой значений параметров, содержащих пробелы. В этой команде пробел интерпретировался как разделитель параметров, а не отслеживались двойные кавычки с последующим объединением двух значений в один параметр. К счастью, предусмотрено еще одно решение, которое позволяет найти выход из этой ситуации.

Дополнительные возможности команды `getopts`

Команда `getopts` (отметим, что она обозначена словом во множественном числе) встроена в командный интерпретатор `bash`. Эта команда во многом напоминает аналогичную ей команду `getopt`, но имеет некоторые расширенные возможности.

В отличие от `getopt`, которая формирует один выходной результирующий набор для всех обработанных опций и параметров, обнаруженных в командной строке, команда `getopts` обрабатывает существующие переменные параметров командного интерпретатора последовательно.

Обработка присутствующих в командной строке параметров этой командой проводится последовательно, по одному, при каждом ее вызове. После того как параметры, подлежащие обработке, заканчиваются, команда возвращает статус выхода больше нуля. Благодаря этому данная команда великолепно подходит для использования в циклах, в которых осуществляется последовательная интерпретация всех параметров в командной строке.

Команда `getopts` имеет следующий формат:

```
getopts optstring variable
```

Значение *optstring* аналогично значению, применяемому в команде `getopt`. В параметре *optstring* перечисляются допустимые символы опций наряду с двоеточием, если для какой-то опции с буквенным обозначением необходимо задать параметр. Для подавления сообщений об ошибках в начале параметра *optstring* необходимо ввести двоеточие. Команда `getopts` помещает текущий параметр в *переменную*, определенную в командной строке.

В команде `getopts` используются две переменные среды. Переменная среды `OPTARG` содержит значение, которое будет использоваться, если для какой-то опции требуется значение параметра. Переменная среды `OPTIND` содержит значение текущего местоположения в списке параметров, на котором остановилась обработка в команде `getopts`. Это позволяет продолжить обработку других параметров командной строки после завершения обработки опций.

Рассмотрим простой пример, в котором используется команда `getopts`:

```
$ cat test19
#!/bin/bash
# простая демонстрация применения команды getopts
```

```

while getopts :ab:c opt
do
    case "$opt" in
        a) echo "Found the -a option" ;;
        b) echo "Found the -b option, with value $OPTARG";;
        c) echo "Found the -c option" ;;
        *) echo "Unknown option: $opt";;
    esac
done
$
$ ./test19 -ab test1 -c
Found the -a option
Found the -b option, with value test1
Found the -c option
$

```

В инструкции `while` определена команда `getopts`, указывающая, какие опции командной строки подлежат обработке, наряду с именем переменной, в которой должны сохраняться эти опции при каждой итерации.

В этом примере можно обнаружить некоторые отличия, касающиеся инструкции `case`. Дело в том, что при интерпретации опций командной строки команда `getopts` дополнительно удаляет в них ведущие тире, поэтому данные символы не требуется указывать в определениях `case`.

Команда `getopts` обладает некоторыми привлекательными особенностями. Первой из них является то, что эта команда позволяет без затруднений включать пробелы в значения параметров:

```

$ ./test19 -b "test1 test2" -a
Found the -b option, with value test1 test2
Found the -a option
$

```

Еще одним удобным свойством команды является возможность задавать букву опции и значение параметра вместе, не разделяя их пробелом:

```

$ ./test19 -abtest1
Found the -a option
Found the -b option, with value test1
$

```

Команда `getopts` правильно интерпретировала значение `test1` из опции `-b`. Представляет также интерес то, что команда `getopts` улавливает все нераспознанные опции, обнаруженные ею в командной строке, в отдельное выходное значение, обозначенное вопросительным знаком:

```

$ ./test19 -d
Unknown option: ?
$
$ ./test19 -acde
Found the -a option
Found the -c option
Unknown option: ?
Unknown option: ?
$

```

Все буквы опций, не определенные в значении *optstring*, возвращаются в код в виде вопросительного знака.

Команда `getopts` способна определить момент, в который должна быть прекращена обработка ею опций и дальнейшая обработка передана в другие команды сценария. После интерпретации командой `getopts` каждой опции происходит увеличение переменной среды `OPTIND` на единицу. Вслед за достижением конца обработки командой `getopts` достаточно просто задать текущее значение `OPTIND` в команде `shift` и перейти к параметрам:

```
$ cat test20
#!/bin/bash
# обработка опций и параметров с помощью команды getopts

while getopts :ab:cd opt
do
    case "$opt" in
        a) echo "Found the -a option" ;;
        b) echo "Found the -b option, with value $OPTARG";;
        c) echo "Found the -c option";;
        d) echo "Found the -d option";;
        *) echo "Unknown option: $opt";;
    esac
done
shift $[ $OPTIND - 1 ]

count=1
for param in "$@"
do
    echo "Parameter $count: $param"
    count=$(( $count + 1 ))
done
$
$ ./test20 -a -b test1 -d test2 test3 test4
Found the -a option
Found the -b option, with value test1
Found the -d option
Parameter 1: test2
Parameter 2: test3
Parameter 3: test4
$
```

Итак, в нашем распоряжении имеется полнофункциональное средство обработки опций и параметров командной строки, которое можно использовать во всех сценариях командного интерпретатора.

Стандартизация параметров

Очевидно, что полный контроль над происходящим в сценарии командного интерпретатора находится в руках того, кто пишет этот сценарий. Программист имеет полное право самостоятельно определять, какие буквенные опции должны использоваться и по какому принципу выбирать сами буквенные обозначения.

Однако в мире Linux было достигнуто своего рода соглашение о стандартизации в части применения некоторых буквенных опций. Придерживаясь аналогичных соглашений, касаю-

щихся опций, в своих сценариях для командного интерпретатора, можно добиться повышения удобства применения этих сценариев широким кругом пользователей.

В табл. 13.1 приведены некоторые из наиболее широко применяемых толкований опций командной строки, используемых в Linux.

Таблица 13.1. Широко применяемые опции командной строки Linux

Параметр	Описание
-a	Показать все объекты
-c	Подсчитать количество
-d	Задать каталог
-e	Развернуть объект
-f	Указать файл, предназначенный для чтения данных из него
-h	Отобразить справочное сообщение, относящееся к данной команде
-i	Игнорировать регистр буквенных символов
-l	Сформировать вывод в версии с длинным форматом
-n	Использовать неинтерактивный (пакетный) режим
-o	Указать выходной файл, в который должен быть перенаправлен весь вывод
-q	Выполнять работу в режиме без вывода сообщений
-r	Обрабатывать каталоги и файлы рекурсивно
-s	Выполнять работу в режиме без вывода сообщений
-v	Формировать подробный вывод
-x	Исключить некоторый объект
-y	Подразумевать положительные ответы на все вопросы

Читатель, по-видимому, может заметить, что большинство из этих значений опций ему уже знакомы, поскольку они часто встречались при описании различных команд `bash` в данной книге. Если для всех подобных опций применяются одинаковые буквенные обозначения, то пользователю становится проще осваивать работу со сценариями, поскольку ему не приходится каждый раз обращаться к описанию в справочном руководстве.

Получение ввода данных от пользователя

Безусловно, применение опций и параметров командной строки представляет собой превосходный способ получения данных от пользователей сценария, но иногда в сценарии требуется организовать более тесное взаимодействие с тем, кто с ним работает. В частности, в определенные моменты возникает необходимость в ходе выполнения сценария задать вопрос и ждать получения ответа от лица, вызвавшего сценарий на выполнение. В командном интерпретаторе `bash` как раз для этой цели предусмотрена команда `read`.

Основные способы чтения данных

Команда `read` принимает входные данные из файла стандартного ввода (с клавиатуры) или из файла с другим дескриптором (подробнее об этом — в главе 14). После получения входных данных команда `read` помещает их в стандартную переменную. Ниже показан наиболее простой вариант применения команды `read`.


```
$ cat test21
#!/bin/bash
# проверка команды read

echo -n "Enter your name: "
read name
echo "Hello $name, welcome to my program. "
$
$ ./test21
Enter your name: Rich Blum
Hello Rich Blum, welcome to my program.
$
```

Очевидно, что здесь нет ничего сложного. Заслуживает внимания то, что в команде `echo`, с помощью которой формируется приглашение к вводу информации, присутствует опция `-n`. Эта опция подавляет вывод символа обозначения конца строки после отображения текста самой строки, что позволяет пользователю сценария вводить данные сразу после показанной строки приглашения, а не в следующей строке. В результате работа со сценарием становится в большей степени похожей на работу с формами.

В действительности команда `read` поддерживает даже опцию `-p`, которая позволяет задавать приглашение непосредственно в командной строке `read`:

```
$ cat test22
#!/bin/bash
# проверка опции read -p
read -p "Please enter your age: " age
days=$(( $age * 365 ))
echo "That makes you over $days days old! "
$
$ ./test22
Please enter your age:10
That makes you over 3650 days old!
$
```

Заслуживает внимания то, что в первом примере, предназначенном для ввода имени, команда `read` присваивает оба значения, имя и фамилию, одной и той же переменной. Команда `read` может присваивать все данные, введенные в приглашении, одной переменной или, по желанию, нескольким переменным. В последнем случае каждое введенное значение данных присваивается очередной переменной в списке. Если список переменных содержит меньше переменных по сравнению с количеством элементов данных, то все оставшиеся данные присваиваются последней переменной:

```
$ cat test23
#!/bin/bash
# ввод нескольких переменных

read -p "Enter your name: " first last
echo "Checking data for $last, $first..."
$
$ ./test23
Enter your name: Rich Blum
Checking data for Blum, Rich...
$
```

В командной строке вызова команды `read` можно также вообще не задавать переменные. Если применяется такой вариант вызова команды `read`, то эта команда помещает все полученные ею данные в специальную переменную среды `REPLY`:

```
$ cat test24
#!/bin/bash
# проверка переменной среды REPLY

read -p "Enter a number: "
factorial=1
for (( count=1; count <= $REPLY; count++ ))
do
    factorial=$(( $factorial * $count ))
done
echo "The factorial of $REPLY is $factorial"
$
$ ./test24
Enter a number: 5
The factorial of 5 is 120
$
```

В таком случае переменная среды `REPLY` включает все данные, заданные в качестве входных, и может использоваться в сценарии командного интерпретатора наряду с любой другой переменной.

Выход по тайм-ауту

При использовании команды `read` приходится учитывать определенную опасность. Вполне возможно, что при выполнении сценария может произойти задержка на неопределенное время, связанная с ожиданием ввода данных пользователем сценария. Если сценарий должен продолжать свою работу независимо от того, вводит ли пользователь какие-либо данные, то можно применить опцию `-t` для задания тайм-аута. Опция `-t` определяет продолжительность времени в секундах, в течение которого команда `read` должна ожидать ввода. По истечении тайм-аута команда `read` выполняет возврат с ненулевым статусом выхода:

```
$ cat test25
#!/bin/bash
# ввод данных с учетом времени

if read -t 5 -p "Please enter your name: " name
then
    echo "Hello $name, welcome to my script"
else
    echo
    echo "Sorry, too slow! "
fi
$
$ ./test25
Please enter your name: Rich
Hello Rich, welcome to my script
$
$ ./test25
```

```
Please enter your name:
Sorry, too slow!
$
```

Пользуясь тем, что команда `read` способна завершать свою работу с ненулевым статусом выхода по истечении тайм-аута, можно легко ввести в действие стандартные структурированные инструкции, такие как инструкция `if-then` или цикл `while`, для контроля над тем, что происходит. В данном примере по истечении тайм-аута выполнение инструкции `if` оканчивается неудачей и командный интерпретатор выполняет команды из раздела `else`.

Вместо прекращения ожидания ввода по тайм-ауту можно также применить команду `read` для подсчета входных символов. В таком случае после ввода заранее заданного количества символов эта команда автоматически завершает работу, присваивая введенные данные переменной:

```
$ cat test26
#!/bin/bash
# получение только одного символа ввода

read -n1 -p "Do you want to continue [Y/N]? " answer
case $answer in
Y | y) echo
        echo "fine, continue on...";;
N | n) echo
        echo OK, goodbye
        exit;;
esac
echo "This is the end of the script"
$
$ ./test26
Do you want to continue [Y/N]? Y
fine, continue on...
This is the end of the script
$
$ ./test26
Do you want to continue [Y/N]? n
OK, goodbye
$
```

В данном примере используется опция `-n` со значением `1`, которая служит для команды `read` указанием принять только один символ и завершить свою работу. Сразу после нажатия пользователем в качестве ответа одного символа команда `read` принимает этот ввод и передает его в переменную. При этом отпадает необходимость нажимать клавишу <Ввод>.

Чтение данных без повтора на экране

Иногда возникает необходимость в получении входных данных от пользователя сценария, но при условии, чтобы эти данные не отображались на мониторе. Классическим примером может служить ввод паролей, но необходимость скрывать данные от постороннего взгляда возникает также при вводе данных многих других типов.

Для предотвращения отображения данных, введенных с помощью команды `read`, на мониторе применяется опция `-s` (в действительности данные на мониторе отображаются, но ко-

манда `read` при этом задает цвет текста, совпадающий с цветом фона). Ниже приведен пример использования опции `-s` в сценарии.

```
$ cat test27
#!/bin/bash
# сокрытие ввода данных на мониторе

read -s -p "Enter your password: " pass
echo
echo "Is your password really $pass? "
$
$ ./test27
Enter your password:
Is your password really T3st1ng?
$
```

Данные, введенные в приглашении, не отображаются на мониторе, но присваиваются переменной для использования в сценарии.

Чтение из файла

Наконец, команду `read` можно также использовать для чтения данных, которые хранятся в файле в системе Linux. Каждый вызов команды `read` приводит к чтению одной строки текста из файла. После исчерпания всех строк в файле команда `read` завершает свою работу с ненулевым статусом выхода.

При такой организации работы сложнее всего обеспечить передачу данных из файла в команду `read`. Для этого чаще всего применяется метод, основанный на том, что результат применения команды `cat` к файлу передается по каналу непосредственно в команду `while`, которая содержит команду `read`. Ниже приведен пример того, как это сделать.

```
$ cat test28
#!/bin/bash
# чтение данных из файла

count=1
cat test | while read line
do
    echo "Line $count: $line"
    count=$((count + 1))
done
echo "Finished processing the file"
$
$ cat test
The quick brown dog jumps over the lazy fox.
This is a test, this is only a test.
O Romeo, Romeo! Wherefore art thou Romeo?
$
$ ./test28
Line 1: The quick brown dog jumps over the lazy fox.
Line 2: This is a test, this is only a test.
Line 3: O Romeo, Romeo! Wherefore art thou Romeo?
Finished processing the file
$
```

В цикле команды `while` строки файла, полученные с помощью команды `read`, продолжают обрабатываться до тех пор, пока команда `read` не завершает свою работу с ненулевым статусом выхода.

Резюме

В настоящей главе показаны три различных метода получения данных из пользовательского сценария. Параметры командной строки позволяют пользователям вводить данные непосредственно в командной строке при вызове сценария на выполнение. В сценарии используются позиционные параметры для получения параметров командной строки и присваивания их переменным.

Команда `shift` позволяет манипулировать параметрами командной строки, осуществляя сдвиг в списке позиционных параметров. Эта команда позволяет легко осуществлять итерацию по параметрам, не определяя заранее количество доступных параметров.

Предусмотрены три специальные переменные, которые можно использовать при работе с параметрами командной строки. Командный интерпретатор задает значение переменной `$#`, равное количеству параметров, введенных в командной строке. Переменная `$*` включает все параметры, представленные в виде одной строки, а переменная `$@` содержит все параметры как отдельные слова. Эти переменные могут применяться при обработке длинных списков параметров.

Кроме параметров, пользователи сценариев могут также задавать опции командной строки для передачи информации в сценарии. Опции командной строки — это однобуквенные обозначения, которым предшествует тире. Различным опциям могут присваиваться разные значения в целях изменения поведения конкретных сценариев. Командный интерпретатор `bash` предоставляет три способа обработки опций командной строки.

Первый способ состоит в том, что опции обрабатываются полностью — аналогично параметрам командной строки. Может быть организована итерация по заданным опциям с использованием позиционных переменных параметров и последовательная обработка каждой опции, представленной в командной строке.

Второй способ обработки опций командной строки основан на применении команды `getopt`. Эта команда приводит опции и параметры командной строки к стандартному формату, которым можно руководствоваться при обработке командной строки в сценарии. Команда `getopt` позволяет указывать, какие буквы должны распознаваться как опции и для каких опций требуются дополнительные значения параметров. Команда `getopt` обрабатывает стандартные параметры командной строки и выводит опции и параметры в надлежащем порядке.

Третий и последний способ обработки опций командной строки состоит в применении команды `getopts` (еще раз подчеркнем, что для обозначения этой команды применяется слово в множественном числе). Команда `getopts` позволяет выполнять более сложную обработку параметров командной строки. Эта команда способна распознавать многозначные параметры, а также выявлять опции, которые не были определены в сценарии.

Для того чтобы сценарий стал интерактивным и позволял получать данные от пользователей в ходе своей работы, можно воспользоваться командой `read`. Команда `read` позволяет выводить в сценариях запросы к вводу информации для пользователей и ожидать получения ответов. Команда `read` помещает все данные, введенные пользователем сценария, в одну или несколько переменных, значения которых можно использовать в сценарии.

Предусмотрена возможность задавать в команде `read` несколько опций, которые позволяют настраивать ввод данных в сценарии, например, использовать скрытый ввод данных, организовывать завершение работы по тайм-ауту и запрашивать ввод конкретного количества символов.

В следующей главе будут представлены дополнительные сведения о выводе данных в сценариях командного интерпретатора `bash`. Выше в данной книге речь шла в основном о том, как отобразить данные на мониторе и перенаправить их в файл. Далее мы рассмотрим несколько дополнительных возможностей, имеющихся в распоряжении разработчика, которые позволяют не только направлять данные в конкретные местоположения, но и указывать, какие типы данных должны быть направлены в то или иное место. Это позволит вам еще больше повысить профессиональный уровень своих сценариев командного интерпретатора!

:

Представление данных

Выше в настоящей книге рассматривались сценарии, в которых предусматривалось отображение информации путем повтора данных на мониторе или перенаправления их в файл. В главе 10 было показано, как перенаправить вывод команды в файл. В настоящей главе приведены дополнительные сведения по этой теме и показано, как перенаправлять вывод сценария в различные местонахождения в системе Linux.

Основные сведения о вводе и выводе

В предыдущих главах были описаны следующие два метода отображения вывода, полученного из сценариев:

- представление вывода на экране монитора;
- перенаправление вывода в файл.

Оба эти метода вывода данных организованы по такому принципу, что от начала и до окончания вывода может применяться только один из них. Тем не менее иногда возникает необходимость отображать часть данных на мониторе, а другую часть выводить в файл. Чтобы можно было успешно действовать в подобных ситуациях, необходимо знать, как осуществляется ввод и вывод в Linux, поскольку без этого невозможно направить вывод сценария в нужное место.

В следующих разделах описано, как применять в своих интересах стандартную систему ввода и вывода Linux для успешной передачи вывода сценариев в конкретные местоположения.

ГЛАВА

14

В этой главе...

Основные сведения о вводе и выводе

Перенаправление вывода в сценариях

Перенаправление ввода в сценариях

Создание собственного перенаправления

Получение перечня открытых дескрипторов файлов

Подавление вывода команды

Использование временных файлов

Ведение журналов сообщений

Резюме

Стандартные дескрипторы файлов

В системе Linux каждый объект рассматривается как файл. Это правило относится также к процессу ввода и вывода. В Linux каждый файловый объект обозначается *дескриптором файла*. Дескриптор файла — это неотрицательное целое число, которое однозначно определяет файлы, открытые в сеансе. В каждом процессе разрешается одновременно иметь открытыми до девяти дескрипторов файлов. В командном интерпретаторе `bash` первые три дескриптора файла (0, 1 и 2) зарезервированы для специального назначения. Описание этих дескрипторов приведено в табл. 14.1.

Таблица 14.1. Стандартные дескрипторы файлов в Linux

Дескриптор файла	Сокращение	Описание
0	STDIN	Стандартный ввод
1	STDOUT	Стандартный вывод
2	STDERR	Стандартный вывод сообщений об ошибках

Эти три специальных дескриптора файлов предназначены для обработки ввода и вывода в сценариях. В командном интерпретаторе эти дескрипторы используются для перенаправления предусмотренного в нем по умолчанию ввода и вывода в соответствующее местоположение (в качестве которого по умолчанию обычно применяются клавиатура и монитор). В следующих разделах каждый из этих стандартных дескрипторов файлов рассматривается более подробно.

STDIN

Дескриптор файла `STDIN` указывает на стандартный ввод в командный интерпретатор. Если речь идет о терминальном интерфейсе, то стандартным вводом является клавиатура. Командный интерпретатор получает ввод от клавиатуры через дескриптор файла `STDIN` и обрабатывает символы один за другим по мере их ввода.

Если используется символ перенаправления ввода (`<`), в системе Linux происходит замена дескриптора стандартного входного файла тем файлом, который указан в операторе перенаправления. С помощью перенаправления чтение файла и получение данных происходит так же, как если бы они были введены на клавиатуре.

Ввод из дескриптора файла `STDIN` принимают многие команды командного интерпретатора `bash`, особенно если в командной строке не указаны какие-либо иные файлы. Ниже приведен пример использования команды `cat` с данными, введенными с помощью дескриптора `STDIN`.

```
$ cat
this is a test
this is a test
this is a second test.
this is a second test.
```

Если бы эта команда `cat` была введена в командной строке отдельно, то получение ввода этой командой происходило бы из дескриптора `STDIN`. При этом после ввода каждой строки команда `cat` выводила бы эту строку на дисплей.

Однако можно также использовать символ перенаправления `STDIN` в качестве указания на то, что команда `cat` должна принимать ввод из другого файла, отличного от `STDIN`:

```
$ cat < testfile
This is the first line.
This is the second line.
```



```
This is the third line.  
$
```

Теперь в команде `cat` в качестве ввода используются строки, содержащиеся в файле `testfile`. Этот метод можно использовать для ввода данных в любую команду командного интерпретатора, которая принимает данные из `STDIN`.

STDOUT

Дескриптор файла `STDOUT` указывает на стандартный вывод из командного интерпретатора. Если речь идет о терминальном интерфейсе, то стандартным выводом является дисплей терминала. Весь вывод из командного интерпретатора (включая программы и сценарии, выполняемые в командном интерпретаторе) перенаправляется в стандартное устройство вывода, функцию которого выполняет монитор.

В большинстве команд командного интерпретатора `bash` по умолчанию предусмотрено перенаправление вывода в дескриптор файла `STDOUT`. Как было показано в главе 10, можно изменить местоположение вывода с использованием перенаправления:

```
$ ls -l > test2  
$ cat test2  
total 20  
-rw-rw-r-- 1 rich rich 53 2010-10-16 11:30 test  
-rw-rw-r-- 1 rich rich  0 2010-10-16 11:32 test2  
-rw-rw-r-- 1 rich rich 73 2010-10-16 11:23 testfile  
$
```

Символ перенаправления вывода позволяет отправлять весь вывод, который обычно поступает на монитор, в файл перенаправления, указанный в команде для командного интерпретатора.

Предусмотрена также возможность присоединять данные к файлу. Для этого используется символ `>>`:

```
$ who >> test2  
$ cat test2  
total 20  
-rw-rw-r-- 1 rich rich 53 2010-10-16 11:30 test  
-rw-rw-r-- 1 rich rich  0 2010-10-16 11:32 test2  
-rw-rw-r-- 1 rich rich 73 2010-10-16 11:23 testfile  
rich      pts/0          2010-10-17 15:34 (192.168.1.2)  
$
```

Вывод, сформированный командой `who`, присоединяется к данным, которые уже находятся в файле `test2`.

Однако при использовании перенаправления вывода в другое место назначения взамен стандартного устройства вывода непосредственно в сценарии может возникнуть проблема. Ниже приведен пример того, что может произойти в сценарии:

```
$ ls -al badfile > test3  
ls: cannot access badfile: No such file or directory  
$ cat test3  
$
```

Если команда вырабатывает сообщение об ошибке, то в отличие от обычного вывода командный интерпретатор не перенаправляет сообщение об ошибке в выходной файл, который указан как используемый для перенаправления. Командным интерпретатором был создан файл

перенаправления вывода, но сообщение об ошибке появилось на экране монитора. Заслуживает внимания то, что при попытке отобразить содержимое файла `test3` ошибка не возникает. Файл `test3` был создан вполне успешно, просто он остался пустым.

Дело в том, что в командном интерпретаторе обработка сообщений об ошибках производится отдельно от обычного вывода. При создании сценария для командного интерпретатора, который должен действовать в фоновом режиме, часто приходится предусматривать отправку выходных сообщений в файл журнала. Если используется этот метод и обнаруживается сообщение об ошибках, то эти сообщения не будут отправлены в файл журнала. Необходимо предпринять что-то другое.

STDERR

В командном интерпретаторе для обработки сообщений об ошибках предназначен специальный дескриптор файла `STDERR`, который указывает стандартное устройство для вывода ошибок, применяемое командным интерпретатором. Данный дескриптор представляет собой местоположение, в которое командный интерпретатор отправляет сообщения об ошибках, сформированные либо самим командным интерпретатором, либо программами и сценариями, выполняемыми в командном интерпретаторе.

По умолчанию дескриптор файла `STDERR` указывает на то же место, что и дескриптор файла `STDOUT` (несмотря на то, что стандартным устройствам обычного вывода и вывода сообщений об ошибках присвоены разные значения дескрипторов файлов). Это означает, что по умолчанию все сообщения об ошибках поступают на дисплей монитора.

Но, как показано в рассматриваемом примере, перенаправление `STDOUT` не приводит автоматически к перенаправлению `STDERR`. Во время работы со сценариями часто возникает необходимость обеспечить перенаправление для обоих дескрипторов вывода, особенно если требуется, чтобы сообщения об ошибках регистрировались в файле журнала.

Перенаправление вывода сообщений об ошибках

Выше в данной главе уже было показано, как перенаправить данные `STDOUT` с использованием символа перенаправления. Перенаправление данных `STDERR` не во многом отличается от этого; достаточно лишь определить дескриптор файла `STDERR` при использовании символа перенаправления. Для этого предусмотрено несколько способов.

Перенаправление только сообщений об ошибках

Как было показано в табл. 14.1, для дескриптора файла `STDERR` предусмотрено числовое значение 2. Может быть выбран вариант с перенаправлением только сообщений об ошибках, для чего это значение дескриптора файла должно быть задано непосредственно перед символом перенаправления. Если за числовым значением дескриптора не будет сразу же следовать символ перенаправления, то операция перенаправления станет недействительной:

```
$ ls -al badfile 2> test4
$ cat test4
ls: cannot access badfile: No such file or directory
$
```

Теперь после запуска команды на выполнение сообщение об ошибке больше не появится на мониторе. Вместо этого все сообщения об ошибках, сформированные в команде, будут сохранены в выходном файле. Если используется данный метод, то командный интерпретатор

перенаправляет только сообщения об ошибках, но не обычные данные. Ниже приведен еще один пример смешивания сообщений STDOUT и STDERR в одном и том же выводе:

```
$ ls -al test badtest test2 2> test5
-rw-rw-r-- 1 rich rich 158 2010-10-16 11:32 test2
$ cat test5
ls: cannot access test: No such file or directory
ls: cannot access badtest: No such file or directory
$
```

Обычный вывод STDOUT из команды `ls` по-прежнему поступает в заданный по умолчанию дескриптор файла STDOUT и отображается на мониторе. В этой команде предусмотрено перенаправление вывода файла с дескриптором 2 (STDERR) в выходной файл, поэтому командный интерпретатор передает все сформированные сообщения об ошибках непосредственно в указанный файл перенаправления.

Перенаправление сообщений об ошибках и данных

Если возникает необходимость перенаправлять и сообщения об ошибках, и обычный вывод, то следует использовать два символа перенаправления. Перед каждым из этих символов должен быть задан соответствующий дескриптор файла для данных, которые необходимо перенаправить, после чего указан соответствующий выходной файл, в котором в конечном итоге будут содержаться данные:

```
$ ls -al test test2 test3 badtest 2> test6 1> test7
$ cat test6
ls: cannot access test: No such file or directory
ls: cannot access badtest: No such file or directory
$ cat test7
-rw-rw-r-- 1 rich rich 158 2010-10-16 11:32 test2
-rw-rw-r-- 1 rich rich  0 2010-10-16 11:33 test3
$
```

Командный интерпретатор перенаправляет обычный вывод команды `ls`, который должен поступать в STDOUT, не в это стандартное устройство, а в файл `test7` с использованием символа `1>`. С другой стороны, все сообщения об ошибках, которые должны были бы передаваться в стандартное устройство STDERR, перенаправляются в файл `test6` с использованием символа `2>`.

Этот метод можно применять для разделения обычного вывода сценария и всех сообщений об ошибках, которые возникают в сценарии. Это позволяет упростить выявление ошибок, поскольку больше не приходится просматривать, допустим, тысячи строк обычных выходных данных.

Еще один вариант состоит в том, что по желанию можно перенаправить вывод и STDERR, и STDOUT в один и тот же выходной файл. В командном интерпретаторе `bash` исключительно для этой цели предусмотрен специальный символ перенаправления `&>`:

```
$ ls -al test test2 test3 badtest &> test7
$ cat test7
ls: cannot access test: No such file or directory
ls: cannot access badtest: No such file or directory
-rw-rw-r-- 1 rich rich 158 2010-10-16 11:32 test2
-rw-rw-r-- 1 rich rich  0 2010-10-16 11:33 test3
$
```

Если используется символ `&>`, то весь вывод, сформированный командой, включая обычные данные и сообщения об ошибках, отправляется в одно и то же место. Заслуживает внимания то, что одно из сообщений об ошибках располагается не в том порядке, как следовало ожидать. Речь идет о сообщении об ошибке, относящемся к файлу `badtest` (последнему файлу, который подлежит к включению в список), которое появляется на втором месте в выходном файле. Дело в том, что командный интерпретатор `bash` автоматически присваивает сообщениям об ошибках более высокий приоритет по сравнению с данными, поступающими в стандартное устройство вывода. Это позволяет просматривать сообщения об ошибках, собранные вместе, а не разбросанные по всему выходному файлу.

Перенаправление вывода в сценариях

Дескрипторы файлов `STDOUT` и `STDERR` могут использоваться в сценариях в целях размещения вывода в нескольких местоположениях, для чего достаточно лишь применить перенаправление к соответствующим дескрипторам файлов. Предусмотрены два метода перенаправления вывода в сценарии:

- временное перенаправление для каждой строки;
- постоянное перенаправление для всех команд в сценарии.

В следующих разделах описано, как работает каждый из этих методов.

Временные перенаправления

Если возникает необходимость в преднамеренном формировании сообщений об ошибках в сценарии, то можно вместе с этим перенаправлять каждую отдельную строку вывода в `STDERR`. Для этого достаточно лишь воспользоваться символом перенаправления вывода, чтобы перенаправить вывод в дескриптор файла `STDERR`. При перенаправлении в дескриптор файла необходимо задать перед числовым обозначением дескриптора файла знак амперсанда (`&`):

```
echo "This is an error message" >&2
```

Эта команда указывает, что содержащийся в ней текст должен быть выведен в стандартное устройство вывода, применяемое для дескриптора файла `STDERR`, заданного в сценарии, а не в обычное устройство вывода `STDOUT`. Ниже приведен пример сценария, в котором используется такое средство.

```
$ cat test8
#!/bin/bash
# проверка сообщений STDERR

echo "This is an error" >&2
echo "This is normal output"
$
```

При выполнении этого сценария обычным образом нельзя будет обнаружить какие-либо отличия:

```
$ ./test8
This is an error
This is normal output
$
```

Напомним, что по умолчанию Linux направляет вывод STDERR в STDOUT. Но если при выполнении сценария будет перенаправлен вывод STDERR, то произойдет также перенаправление всех данных, которые направлены в сценарии в STDERR:

```
$ ./test8 2> test9
This is normal output
$ cat test9
This is an error
$
```

Это как раз то, что требуется! Текст, отображаемый с использованием STDOUT, появляется на мониторе, в то время как текст в инструкции echo, отправленный в STDERR, перенаправляется в выходной файл.

Этот метод очень хорошо подходит для выработки в сценариях собственных сообщений об ошибках. Другие программисты, применяющие такие сценарии, смогут легко перенаправить сообщения об ошибках с использованием дескриптора файла STDERR, как и было показано.

Постоянные перенаправления

Если в сценарии приходится перенаправлять большой объем данных, то может оказаться слишком трудоемким перенаправление вывода каждой отдельной инструкции echo. Вместо этого можно указать командному интерпретатору, что в ходе всего выполнения данного сценария должно осуществляться перенаправление вывода конкретного дескриптора файла, с помощью команды exec:

```
$ cat test10
#!/bin/bash
# перенаправление всего вывода в файл
exec 1>testout

echo "This is a test of redirecting all output"
echo "from a script to another file."
echo "without having to redirect every individual line"
$ ./test10
$ cat testout
This is a test of redirecting all output
from a script to another file.
without having to redirect every individual line
$
```

При вызове этой команды exec происходит запуск нового командного интерпретатора и перенаправление вывода дескриптора файла STDOUT в файл. Весь вывод в этом сценарии, который должен быть направлен в STDOUT, вместо этого перенаправляется в файл.

Предусмотрена также возможность перенаправить вывод в STDOUT не с самого начала, а в ходе выполнения сценария:

```
$ cat test11
#!/bin/bash
# перенаправление вывода в различные места

exec 2>testerror

echo "This is the start of the script"
echo "now redirecting all output to another location"
```

```

exec 1>testout

echo "This output should go to the testout file"
echo "but this should go to the testerror file" >&2
$
$ ./test11
This is the start of the script
now redirecting all output to another location
$ cat testout
This output should go to the testout file
$ cat testerror
but this should go to the testerror file
$

```

В данном сценарии применяется команда `exec` для перенаправления всего вывода, предназначенного для `STDERR`, в файл `testerror`. Затем в сценарии используется инструкция `echo` для вывода нескольких строк в стандартное устройство `STDOUT`. После этого применяется команда `exec` для перенаправления вывода `STDOUT` в файл `testout`. Заслуживает внимания то, что даже после перенаправления `STDOUT` остается возможным задавать `STDERR` в качестве стандартного устройства вывода для инструкции `echo`, что приводит в данном случае к перенаправлению в файл `testerror`.

Это средство может потребоваться, если возникает необходимость перенаправить лишь часть вывода сценария в другое место, например, в журнал ошибок. Тем не менее при использовании указанного средства приходится сталкиваться еще с одной проблемой.

Как только вывод `STDOUT` или `STDERR` будет перенаправлен, возникают затруднения при перенаправлении его снова в исходное местоположение. Поэтому в случае необходимости прямого и обратного переключения перенаправления приходится пользоваться еще одним приемом, который описан ниже. В разделе “Создание собственного перенаправления”, ниже в этой главе, приведено описание данного приема и показано, как его использовать в сценариях командного интерпретатора.

Перенаправление ввода в сценариях

Тот же способ, который предназначен для перенаправления вывода `STDOUT` и `STDERR` в сценариях, можно применить для перенаправления ввода `STDIN` с клавиатуры. В системе Linux в качестве средства перенаправления `STDIN` в файл может применяться команда `exec`:

```
exec 0< testfile
```

Эта команда служит для командного интерпретатора указанием, что ввод должен быть получен из файла `testfile`, а не из стандартного устройства `STDIN`. Это перенаправление применяется в любое время, когда в сценарии должны быть получены входные данные. Ниже приведен пример практического применения указанного средства.

```

$ cat test12
#!/bin/bash
# перенаправление вывода файла

exec 0< testfile
count=1

while read line

```

```
do
    echo "Line #\$count: \$line"
    count=$(( $count + 1 ))
done
$ ./test12
Line #1: This is the first line.
Line #2: This is the second line.
Line #3: This is the third line.
$
```

В главе 13 было показано, как использовать команду `read` для чтения данных, введенных с клавиатуры пользователем. Перенаправляя ввод `STDIN` в файл при каждой попытке чтения в команде `read` из стандартного устройства `STDIN`, можно обеспечить получение данных из файла вместо клавиатуры.

Это — превосходный метод чтения данных из файлов для последующей обработки в сценариях. В частности, системным администраторам Linux часто приходится обеспечивать чтение данных из файлов журналов для последующей обработки. Описанный способ позволяет выполнять указанную задачу проще всего.

Создание собственного перенаправления

При перенаправлении ввода и вывода в сценарии можно не ограничиваться тремя заданными по умолчанию дескрипторами файлов. Выше в этой главе уже было сказано, что командный интерпретатор позволяет иметь открытыми до девяти дескрипторов файлов. Остальные шесть дескрипторов файлов, кроме стандартных, получают числовые обозначения от трех до восьми и становятся доступными для применения при перенаправлении либо ввода, либо вывода. Кроме того, для любого из этих дескрипторов файлов можно назначать файлы, после чего также использовать их в своих сценариях. В настоящем разделе описано, как использовать остальные дескрипторы файлов в сценариях.

Создание дескрипторов выходных файлов

Предусмотрена возможность назначить дескриптор файла для вывода с помощью команды `exec`. Как и в отношении стандартных дескрипторов файлов, после присваивания альтернативного дескриптора файла для определенного местоположения файла создается перенаправление, которое остается постоянным, пока не будет переназначено. Ниже приведен простой пример использования альтернативного дескриптора файла в сценарии.

```
$ cat test13
#!/bin/bash
# использование альтернативного дескриптора файла

exec 3>test13out

echo "This should display on the monitor"
echo "and this should be stored in the file" >&3
echo "Then this should be back on the monitor"
$ ./test13
```

```
This should display on the monitor
Then this should be back on the monitor
$ cat test13out
and this should be stored in the file
$
```

В этом сценарии используется команда `exec` для перенаправления дескриптора файла 3 в альтернативное местоположение файла. При выполнении в этом сценарии инструкций `echo` результаты отображаются на стандартном устройстве `STDOUT`, как и следовало ожидать. Однако вывод инструкций `echo`, перенаправленный в дескриптор файла 3, поступает в альтернативный файл. Это позволяет по-прежнему отправлять обычный вывод на монитор, а специальную информацию передавать в те или иные файлы, например, в файлы журнала.

Можно также воспользоваться командой `exec` для присоединения данных к существующему файлу вместо создания нового файла:

```
exec 3>>test13out
```

Теперь вывод будет присоединяться к файлу `test13out`, а не создаваться новый файл.

Перенаправление дескрипторов файлов

Теперь можно приступить к описанию приема, позволяющего вернуть в прежнее место перенаправленный ранее дескриптор файла. При этом можно присваивать альтернативный дескриптор файла стандартному дескриптору файла, и наоборот. Это означает, что можно перенаправить исходное местоположение вывода `STDOUT` в альтернативный дескриптор файла, а затем снова перенаправить этот дескриптор файла в `STDOUT`. На первый взгляд такой прием работы кажется довольно сложным, но на практике осуществляется вполне легко. Приведем простой пример, после изучения которого все станет ясно:

```
$ cat test14
#!/bin/bash
# резервирование дескриптора STDOUT, затем снова возврат к нему

exec 3>&1
exec 1>test14out

echo "This should store in the output file"
echo "along with this line."

exec 1>&3

echo "Now things should be back to normal"
$
$ ./test14
Now things should be back to normal
$ cat test14out
This should store in the output file
along with this line.
$
```

При первом знакомстве приведенный пример выглядит весьма запутанным, поэтому разберем его последовательно. Прежде всего, в этом сценарии происходит перенаправление дескриптора файла 3 в текущее местоположение вывода дескриптора файла 1, каковым является `STDOUT`. Это означает, что весь вывод, отправленный в дескриптор файла 3, будет поступать на монитор.

Вторая команда `exec` перенаправляет вывод `STDOUT` в файл. После этого командный интерпретатор перенаправляет любой вывод, отправленный в `STDOUT`, непосредственно в выходной файл. Однако дескриптор файла 3 все еще указывает на исходное местоположение вывода `STDOUT`, каковым является монитор. Если на данном этапе выходные данные будут передаваться в дескриптор файла 3, то по-прежнему станут появляться на мониторе, несмотря на то, что произошло перенаправление `STDOUT`.

После отправки определенного объема вывода в стандартное устройство `STDOUT`, которое указывает на файл, на следующем этапе в этом сценарии происходит перенаправление `STDOUT` в текущее местоположение дескриптора файла 3, каковым по-прежнему является монитор. Это означает, что теперь стандартное устройство `STDOUT` указывает на свое исходное местоположение, т.е. на монитор.

Безусловно, при использовании данного метода может возникать путаница, но он широко применяется для временного перенаправления вывода в файлах сценариев, а затем повторного восстановления обычных настроек вывода.

Создание дескрипторов входных файлов

Предусмотрена возможность перенаправлять дескрипторы входных файлов по такому же принципу, как и дескрипторы выходных файлов. Для этого необходимо сохранить местоположение ввода дескриптора файла `STDIN` в другом дескрипторе файла перед перенаправлением его в другой файл, причем после завершения чтения файла можно восстановить исходное местоположение ввода `STDIN`:

```
$ cat test15
#!/bin/bash
# перенаправление дескрипторов входных файлов

exec 6<&0

exec 0< testfile

count=1
while read line
do
    echo "Line #$count: $line"
    count=$(( $count + 1 ))
done
exec 0<&6
read -p "Are you done now? " answer
case $answer in
Y|y) echo "Goodbye";;
N|n) echo "Sorry, this is the end.";;
esac
$ ./test15
Line #1: This is the first line.
Line #2: This is the second line.
Line #3: This is the third line.
Are you done now? y
Goodbye
$
```

В этом примере для хранения местоположения ввода `STDIN` используется дескриптор файла 6. Затем в этом сценарии происходит перенаправление ввода `STDIN` в файл. Весь ввод для команды `read` поступает из перенаправленного стандартного устройства `STDIN`, каковым теперь является входной файл.

После чтения всех строк в сценарии происходит возврат стандартного устройства `STDIN` в исходное местоположение, для чего применяется его перенаправление в дескриптор файла 6. Кроме того, в сценарии выполняется проверка, позволяющая убедиться в том, что `STDIN` снова имеет обычное местоположение, с помощью еще одной команды `read`, которая на сей раз ожидает ввода с клавиатуры.

Создание дескриптора файла для чтения-записи

При первом знакомстве с этим средством бывает трудно понять, для чего оно предназначено, однако предусмотрена также возможность открывать отдельный дескриптор файла и для ввода, и для вывода. После этого можно использовать один и тот же дескриптор файла для чтения данных из файла и записи данных в тот же файл.

Однако при использовании этого метода необходимо соблюдать исключительную осторожность. В ходе выполнения операций чтения данных из файла и записи в файл командный интерпретатор сопровождает внутренний указатель, значение которого служит для обозначения конкретного местоположения в файле. При любых операциях чтения или записи в качестве точки отсчета берется то местоположение в файле, в котором перед этим остался указатель. Если не применяются некоторые меры предосторожности, то могут возникать непредвиденные результаты. Рассмотрим следующий пример:

```
$ cat test16
#!/bin/bash
# проверка дескрипторов входных/выходных файлов

exec 3<> testfile
read line <&3
echo "Read: $line"
echo "This is a test line" >&3
$ cat testfile
This is the first line.
This is the second line.
This is the third line.
$ ./test16
Read: This is the first line.
$ cat testfile
This is the first line.
This is a test line
ine.
This is the third line.
$
```

В этом примере используется команда `exec` для назначения дескриптора файла 3 для ввода и вывода, что сводится к передаче данных в файл и из файла `testfile`. Затем в примере используется команда `read` для чтения первой строки в файле с помощью назначенного дескриптора файла, после чего считанная строка отображается на стандартном устройстве `STDOUT`. После этого применяется инструкция `echo` для записи строки данных в файл, открытый с тем же дескриптором файла.

После вызова этого сценария на выполнение создается впечатление, что дела обстоят вполне благополучно. Вывод показывает, что в сценарии считывается первая строка из файла `testfile`. Однако при отображении содержимого файла `testfile` после выполнения сценария обнаруживается, что данными, записанными в файл, были перезаписаны существующие данные.

При выводе данных в файл в этом сценарии запись начинается с того места, где находится указатель в файле. Команда `read` считывает первую строку данных, после чего указатель в файле остается на первом символе во второй строке данных. При выводе данных в файл с помощью инструкции `echo` запись начинается с текущего местоположения указателя в файле, что приводит к перезаписи всех находящихся там данных.

Заккрытие дескрипторов файлов

После создания новых дескрипторов входных или выходных файлов командный интерпретатор автоматически закрывает их по завершении работы сценария. Тем не менее в некоторых ситуациях возникает необходимость закрывать дескрипторы файлов вручную до окончания сценария.

Чтобы закрыть дескриптор файла, его необходимо перенаправить в специальный символ `&-`. Ниже показано, как такая операция может выглядеть в сценарии.

```
exec 3>&-
```

В этой инструкции закрывается дескриптор файла 3, что исключает возможность его дальнейшего использования в сценарии. Ниже приведен пример того, что происходит при попытке обратиться к закрытому дескриптору файла.

```
$ cat badtest
#!/bin/bash
# проверка дескрипторов закрытых файлов

exec 3> test17file

echo "This is a test line of data" >&3

exec 3>&-

echo "This won't work" >&3
$ ./badtest
./badtest: 3: Bad file descriptor
$
```

После закрытия дескриптора файла исключается возможность писать с его применением какие-либо данные в сценарии, поскольку при такой попытке командный интерпретатор выдает сообщение об ошибке.

При закрытии дескрипторов файлов необходимо учитывать еще один нюанс. Если тот же выходной файл будет еще раз открыт в сценарии, то командный интерпретатор заменит существующий файл новым файлом. Это означает, что вывод в повторно открытый файл любых данных приводит к перезаписи существующего файла. Рассмотрим следующий пример проявления этой проблемы:

```
$ cat test17
#!/bin/bash
# проверка дескрипторов закрытых файлов

exec 3> test17file
```

```

echo "This is a test line of data" >&3
exec 3>&-

cat test17file

exec 3> test17file
echo "This'll be bad" >&3
$ ./test17
This is a test line of data
$ cat test17file
This'll be bad
$

```

После отправки строки данных в файл `test17file` и закрытия дескриптора файла в этом сценарии используется команда `cat` для отображения содержимого файла. До сих пор дела обстоят нормально. Затем в сценарии происходит повторное открытие выходного файла и отправка в него другой строки данных. Теперь после отображения содержимого выходного файла обнаруживается лишь вторая строка данных. Командный интерпретатор перезаписал первоначально записанный выходной файл.

Получение перечня открытых дескрипторов файлов

Как уже было сказано выше, программисту доступны только девять дескрипторов файлов, поэтому может показаться, что их использование достаточно легко контролировать. Однако иногда, пытаясь проследить за тем, куда перенаправлены те или иные дескрипторы файлов, вполне можно потерять контроль над происходящим. Чтобы при работе со сценариями не приходилось лишний раз разбираться в проблеме перенаправления файлов, в командном интерпретаторе `bash` предусмотрена команда `lsuf`.

Команда `lsuf` перечисляет все дескрипторы открытых файлов во всей системе Linux. Не все специалисты одобряют применение данного средства, поскольку оно может предоставить лишнюю информацию о системе Linux лицам, не являющимися системными администраторами. По этой причине во многих системах Linux данная команда определена как скрытая, чтобы простые пользователи не могли случайно на нее натолкнуться, изучая систему.

А в системе Fedora Linux автора команда `lsuf` находится в каталоге `/usr/sbin`. Для запуска этой команды на выполнение с помощью учетной записи обычного пользователя необходимо задать полное имя пути к команде:

```
$ /usr/sbin/lsuf
```

Команда `lsuf` вырабатывает на удивление большой объем вывода. С ее помощью отображается информация о каждом файле, открытом в настоящее время в системе Linux. К этому относятся файлы, открытые во всех процессах, которые выполняются в фоновом режиме, а также файлы, открытые во всех учетных записях пользователей, зарегистрированных в системе.

Для фильтрации вывода команды `lsuf` предусмотрено большое количество опций и параметров командной строки. Чаще всего применяются опция `-p`, которая позволяет указать идентификатор процесса (PID), и опция `-d`, с помощью которой можно указать номера дескрипторов файлов, подлежащих отображению.

Чтобы было проще определить текущее значение PID процесса, можно воспользоваться специальной переменной среды \$\$, с помощью которой командный интерпретатор устанавливает текущий PID. Для выполнения логической операции AND над результатами применения двух других опций предназначена опция -a, которая вырабатывает, например, следующий вывод:

```
$ /usr/sbin/lsof -a -p $$ -d 0,1,2
COMMAND  PID USER  FD   TYPE DEVICE SIZE NODE NAME
bash     3344 rich   0u   CHR  136,0        2 /dev/pts/0
bash     3344 rich   1u   CHR  136,0        2 /dev/pts/0
bash     3344 rich   2u   CHR  136,0        2 /dev/pts/0
$
```

В этом примере показаны заданные по умолчанию дескрипторы файлов (0, 1 и 2) для текущего процесса (командного интерпретатора bash). Заданный по умолчанию вывод lsof содержит несколько столбцов информации, которые описаны в табл. 14.2.

Таблица 14.2. Вывод команды lsof по умолчанию

Столбец	Описание
COMMAND	Первые девять символов имени команды в текущем процессе
PID	Идентификатор рассматриваемого процесса
USER	Регистрационное имя пользователя, который является владельцем процесса
FD	Числовое значение дескриптора файла и тип доступа (r — чтение, w — запись, u — чтение и запись)
TYPE	Тип файла (CHR — символьный файл, BLK — блочный файл, DIR — каталог, REG — обычный файл)
DEVICE	Номера устройства (старший и младший)
SIZE	Размер файла, если эти данные доступны
NODE	Номер узла локального файла
NAME	Имя файла

Файлы, связанные со стандартными устройствами STDIN, STDOUT и STDERR, по своему типу являются символьными. Кроме того, все эти дескрипторы файлов, STDIN, STDOUT и STDERR, указывают на терминал, поэтому имя выходного файла — это имя терминального устройства. Все три стандартных файла доступны и для чтения, и для записи (хотя в действительности невозможно понять, что означает запись в STDIN и чтение из STDOUT).

А теперь рассмотрим результаты вызова команды lsof на выполнение из сценария, в котором предусмотрено открытие целого ряда альтернативных дескрипторов файлов:

```
$ cat testl8
#!/bin/bash
# проверка команды lsof с дескрипторами файлов

exec 3> testl8file1
exec 6> testl8file2
exec 7< testfile

/usr/sbin/lsof -a -p $$ -d0,1,2,3,6,7
$ ./testl8
COMMAND  PID  USER  FD   TYPE DEVICE SIZE  NODE NAME
testl8    3594 rich   0u   CHR  136,0        2 /dev/pts/0
testl8    3594 rich   1u   CHR  136,0        2 /dev/pts/0
testl8    3594 rich   2u   CHR  136,0        2 /dev/pts/0
```

```

18 3594 rich      3w  REG  253,0    0 360712 /home/rich/test18file1
18 3594 rich      6w  REG  253,0    0 360715 /home/rich/test18file2
18 3594 rich      7r  REG  253,0   73 360717 /home/rich/testfile
$

```

В этом сценарии создаются три альтернативных дескриптора файлов: два — для вывода (3 и 6) и один — для ввода (7). После выполнения в сценарии команды `ls -l` формируются результаты, в которых содержатся сведения о дескрипторах новых файлов. В настоящей книге первая часть вывода усечена, чтобы не происходила путаница при поиске в результатах имен файлов. Имена файлов отображаются с указанием полных имен путей к файлам, применяемым при работе с дескрипторами файлов. Каждый файл показан как имеющий тип REG, а это означает, что данные файлы представляют собой обычные файлы в файловой системе.

Подавление вывода команды

Иногда возникают такие ситуации, в которых необходимо предотвратить отображение какого-либо вывода из сценария. В частности, такое требование возникает при выполнении сценария как фонового процесса (см. главу 15). Если же в сценарии, выполняемом в фоновом режиме, вырабатываются какие-либо сообщения об ошибках, то командный интерпретатор передает эти сообщения по электронной почте владельцу процесса. Задача анализа всей этой почты может стать трудоемкой, особенно при эксплуатации сценариев, формирующих сообщения о мелких и незначительных ошибках.

Для решения этой проблемы можно перенаправить вывод стандартного устройства `STDERR` в специальный файл, называемый *файлом null*. Файл `null` в основном соответствует своему имени, которое указывает, что этот файл ничего не содержит. Любые данные, отправляемые командным интерпретатором в файл `null`, не сохраняются, поэтому бесследно исчезают.

Стандартным местоположением файла `null` в системах Linux является `/dev/null`. Любые данные, перенаправленные в это местоположение, отбрасываются и больше нигде не появляются:

```

$ ls -al > /dev/null
$ cat /dev/null
$

```

На этом основан общепринятый способ подавления любых сообщений об ошибках без их фактического сохранения:

```

$ ls -al badfile test16 2> /dev/null
-rwxr--r--  1 rich      rich          135 Oct 29 19:57 test16*
$

```

Файл `/dev/null` можно также использовать для перенаправления ввода как входной файл. Еще раз отметим, что файл `/dev/null` ничего не содержит, поэтому часто используется программистами для быстрого удаления данных из существующего файла без его удаления и повторного создания:

```

$ cat testfile
This is the first line.
This is the second line.
This is the third line.
$ cat /dev/null > testfile
$ cat testfile
$

```

Файл `testfile` продолжает существовать в системе, но теперь он пуст. В этом состоит еще один удобный метод, с помощью которого происходит очистка файлов журнала, которые должны всегда присутствовать в системе, чтобы приложения могли работать.

Использование временных файлов

В системе Linux предусмотрено специальное местоположение в каталоге, зарезервированное для временных файлов. В Linux для файлов, которые не должны храниться неопределенно долго, предназначен каталог `/tmp`. Кроме того, в большинстве дистрибутивов Linux настройка системы выполняется так, чтобы удаление всех файлов в каталоге `/tmp` происходило автоматически во время загрузки.

Любой учетной записи пользователя в системе предоставляется право доступа на чтение и запись файлов в каталоге `/tmp`. Это средство предоставляет простой способ создания временных файлов, об очистке которых не нужно беспокоиться.

Предусмотрена даже специальная команда, с помощью которой могут создаваться временные файлы. Это — команда `mktemp`, позволяющая легко создать временный файл с уникальным именем в папке `/tmp`. Командный интерпретатор создает необходимый файл, но не использует при этом предусмотренное по умолчанию значение пользовательской маски (см. главу 6). Вместо этого командный интерпретатор назначает разрешения на чтение и запись для владельца файла, указывая в качестве владельца того пользователя, которому потребовалось создать этот файл. После создания файла пользователю сценария предоставляется полный доступ для чтения и записи в файле в данном сценарии, однако больше никто не будет иметь возможность получения доступа к этому файлу (разумеется, речь не идет о пользователе `root`).

Создание локального временного файла

По умолчанию команда `mktemp` создает файл в локальном каталоге. Чтобы создать временный файл в локальном каталоге с помощью команды `mktemp`, достаточно лишь задать шаблон имени файла. Такой шаблон состоит из любого текстового имени файла, к которому в конце необходимо добавить шесть символов `x`:

```
$ mktemp testing.XXXXXX
$ ls -al testing*
-rw----- 1 rich      rich      0 Oct 17 21:30 testing.UfIi13
$
```

Команда `mktemp` заменяет шесть символов `x` шестизначным кодом, позволяющим гарантировать уникальность имени файла в каталоге. Предусмотрена возможность создавать произвольное количество временных файлов, оставаясь полностью уверенным в том, что каждый из этих файлов будет иметь уникальное имя:

```
$ mktemp testing.XXXXXX
testing.1DRLuV
$ mktemp testing.XXXXXX
testing.lVBtkW
$ mktemp testing.XXXXXX
testing.PgqNKG
$ ls -l testing*
-rw----- 1 rich      rich      0 Oct 17 21:57 testing.1DRLuV
-rw----- 1 rich      rich      0 Oct 17 21:57 testing.PgqNKG
```

```
-rw----- 1 rich      rich      0 Oct 17 21:30 testing.UfIi13
-rw----- 1 rich      rich      0 Oct 17 21:57 testing.lVBtkW
$
```

Вполне очевидно, что вывод команды `mktemp` представляет собой имя файла, созданного этой командой. При использовании команды `mktemp` в сценарии необходимо сохранить это имя файла в переменной, чтобы иметь возможность в дальнейшем обращаться к файлу в сценарии:

```
$ cat test19
#!/bin/bash
# создание и использование временного файла

tempfile='mktemp test19.XXXXXX'

exec 3>$tempfile

echo "This script writes to temp file $tempfile"

echo "This is the first line" >&3
echo "This is the second line." >&3
echo "This is the last line." >&3
exec 3>&-

echo "Done creating temp file. The contents are:"
cat $tempfile
rm -f $tempfile 2> /dev/null
$ ./test19
This script writes to temp file test19.vCHoya
Done creating temp file. The contents are:
This is the first line
This is the second line.
This is the last line.
$ ls -al test19*
-rwxr--r-- 1 rich      rich      356 Oct 29 22:03 test19*
$
```

В рассматриваемом сценарии используется команда `mktemp` для создания временного файла, а имя файла присваивается переменной `$tempfile`. Затем временный файл используется в качестве файла перенаправления вывода для дескриптора файла 3. После отображения имени временного файла с помощью стандартного устройства `STDOUT` происходит запись во временный файл нескольких строк, затем дескриптор файла закрывается. Наконец, в сценарии отображается содержимое временного файла и происходит удаление этого файла с помощью команды `rm`.

Создание временного файла в каталоге `/tmp`

Опция `-t` вынуждает команду `mktemp` создавать файлы в каталоге системы для временных файлов. Если используется это средство, команда `mktemp` возвращает полное имя пути, которое использовалось для создания временного файла, а не только имя файла:

```
$ mktemp -t test.XXXXXX
/tmp/test.xG3374
$ ls -al /tmp/test*
-rw----- 1 rich rich 0 2010-10-29 18:41 /tmp/test.xG3374
$
```


В связи с тем, что в данном случае команда `mktemp` возвращает полное имя пути, появляется возможность ссылаться на временный файл из любого каталога в системе Linux, независимо от того, где находится сам каталог для временных файлов:

```
$ cat test20
#!/bin/bash
# создание временного файла в каталоге /tmp

tempfile='mktemp -t tmp.XXXXXX'

echo "This is a test file." > $tempfile
echo "This is the second line of the test." >> $tempfile

echo "The temp file is located at: $tempfile"
cat $tempfile
rm -f $tempfile
$ ./test20
The temp file is located at: /tmp/tmp.Ma3390
This is a test file.
This is the second line of the test.
$
```

После создания в команде `mktemp` временного файла происходит возврат полного имени пути с помощью переменной среды. Затем это значение можно использовать в любой команде для ссылки на временный файл.

Создание временного каталога

Опция `-d` служит для команды `mktemp` указанием, что должен быть создан не временный файл, а временный каталог. Затем этот каталог может применяться для любых целей, в частности, для создания дополнительных временных файлов:

```
$ cat test21
#!/bin/bash
# использование временного каталога

tempdir='mktemp -d dir.XXXXXX'
cd $tempdir
tempfile1='mktemp temp.XXXXXX'
tempfile2='mktemp temp.XXXXXX'
exec 7> $tempfile1
exec 8> $tempfile2

echo "Sending data to directory $tempdir"
echo "This is a test line of data for $tempfile1" >&7
echo "This is a test line of data for $tempfile2" >&8
$ ./test21
Sending data to directory dir.ouT8S8
$ ls -al
total 72
drwxr-xr-x  3 rich    rich      4096 Oct 17 22:20 ./
drwxr-xr-x  9 rich    rich      4096 Oct 17 09:44 ../
drwx----- 2 rich    rich      4096 Oct 17 22:20 dir.ouT8S8/
-rwxr--r--  1 rich    rich      338 Oct 17 22:20 test21*
```

```

$ cd dir.ouT8S8
[dir.ouT8S8]$ ls -al
total 16
drwx-----  2 rich      rich      4096 Oct 17 22:20 ./
drwxr-xr-x   3 rich      rich      4096 Oct 17 22:20 ../
-rw-----  1 rich      rich        44 Oct 17 22:20 temp.N5F306
-rw-----  1 rich      rich        44 Oct 17 22:20 temp.SQslb7
[dir.ouT8S8]$ cat temp.N5F306
This is a test line of data for temp.N5F306
[dir.ouT8S8]$ cat temp.SQslb7
This is a test line of data for temp.SQslb7
[dir.ouT8S8]$

```

В этом сценарии предусмотрено создание подкаталога в текущем каталоге, а затем переход в этот каталог с помощью команды `cd` перед созданием двух временных файлов. После этого этим двум временным файлам присваиваются дескрипторы файлов, которые используются для сохранения вывода из сценария.

Ведение журналов сообщений

Иногда возникает необходимость отправлять вывод и на монитор, и в файл, для ведения журнала. При этом, чтобы не приходилось перенаправлять вывод дважды, можно воспользоваться специальной командой `tee`.

Команда `tee` по своему действию напоминает Т-образный тройник на трубопроводе, отсюда ее название. С помощью этой команды данные из `STDIN` передаются одновременно по двум назначениям. Одним из назначений является `STDOUT`. Другим назначением служит имя файла, указанное в командной строке `tee`:

```
tee filename
```

В связи с тем, что команда `tee` перенаправляет данные из `STDIN`, ее можно использовать вместе с командой канала для перенаправления вывода из любой команды:

```

$ date | tee testfile
Sun Oct 17 18:56:21 EDT 2010
$ cat testfile
Sun Oct 17 18:56:21 EDT 2010
$

```

Полученный вывод появляется на стандартном устройстве `STDOUT` и записывается в указанный файл. Но при этом необходимо соблюдать осторожность, поскольку по умолчанию команда `tee` при каждом вызове ее на выполнение перезаписывает выходной файл:

```

$ who | tee testfile
rich      pts/0          2010-10-17 18:41 (192.168.1.2)
$ cat testfile
rich      pts/0          2010-10-17 18:41 (192.168.1.2)
$

```

Если вместо этого требуется присоединять данные к файлу, то следует использовать опцию `-a`:

```

$ date | tee -a testfile
Sun Oct 17 18:58:05 EDT 2010

```

```
$ cat testfile
rich      pts/0      2010-10-17 18:41 (192.168.1.2)
Sun Oct 17 18:58:05 EDT 2010
$
```

Используя данный метод, можно одновременно и сохранять данные в файле, и отображать их на мониторе для пользователей сценария:

```
$ cat test22
#!/bin/bash
# использование команды tee для ведения журнала

tempfile=test22file

echo "This is the start of the test" | tee $tempfile
echo "This is the second line of the test" | tee -a $tempfile
echo "This is the end of the test" | tee -a $tempfile
$ ./test22
This is the start of the test
This is the second line of the test
This is the end of the test
$ cat test22file
This is the start of the test
This is the second line of the test
This is the end of the test
$
```

Благодаря этому появляется возможность, например, не только вывести данные для ознакомления с ними пользователей, но и сохранить постоянную копию данных.

Резюме

Успешное создание сценариев невозможно без понимания того, как происходит обработка ввода и вывода в командном интерпретаторе `bash`. Предусмотрена возможность управлять и тем, как в сценарии происходит получение данных, и тем, как эти данные отображаются, что позволяет настраивать сценарий для любой среды. В частности, можно перенаправить ввод сценария из стандартного устройства ввода (`STDIN`) в любой файл в системе. Кроме того, в любой файл в системе можно перенаправить вывод сценария из стандартного устройства вывода (`STDOUT`).

Кроме результатов, передаваемых в `STDOUT`, можно перенаправлять любые сообщения об ошибках, формируемые в сценарии, для чего служит перенаправление вывода `STDERR`. Эта задача выполняется путем перенаправления дескриптора файла, связанного с выводом `STDERR`, который имеет дескриптор файла 2. Кроме того, можно перенаправить вывод `STDERR` в тот же файл, что и вывод `STDOUT`, или полностью в отдельный файл. Благодаря этому появляется возможность отделить обычные результаты выполнения сценария от всех сообщений об ошибках, формируемых в сценарии.

Командный интерпретатор `bash` позволяет создавать собственные дескрипторы файлов для использования в сценариях. Предусмотрена возможность создавать дескрипторы файлов с числовыми обозначениями от 3 до 9 и присваивать их по желанию любому выходному файлу. После создания дескриптора файла в него можно перенаправить вывод любой команды с применением стандартных символов перенаправления.

Командный интерпретатор `bash` позволяет также перенаправить ввод в дескриптор файла, предоставляя тем самым удобный способ чтения данных, содержащихся в файле, непосредственно в сценарии. Для получения с помощью командного интерпретатора сведений об активных дескрипторах файлов можно использовать команду `ls -of`.

В системах Linux предусмотрен специальный файл с именем `/dev/null`, позволяющий перенаправить в него вывод, который подлежит уничтожению. Система Linux отбрасывает все, что было перенаправлено в файл `/dev/null`. Файл `null` можно также использовать для уничтожения содержимого любого файла, для чего достаточно перенаправить это содержимое в файл `/dev/null`.

Команда `mktemp` представляет собой одно из удобных средств командного интерпретатора `bash`, которое позволяет легко создавать временные файлы и каталоги. Достаточно просто задать для команды `mktemp` шаблон имени файла, после чего создается файл с уникальным именем при каждом вызове этой команды на основе заданного формата шаблона. Предусмотрена также возможность создавать временные файлы и подкаталоги в каталоге `/tmp` системы Linux, используя этот каталог в качестве специального местоположения, содержимое которого уничтожается после очередной начальной загрузки системы.

Если возникает необходимость отправлять выходные данные одновременно и в стандартное устройство вывода, и в файл журнала, то можно воспользоваться удобной командой `tee`. Это позволяет отображать результаты выполнения сценария на мониторе и вместе с тем сохранять их в файле журнала.

В главе 15 приведены сведения о том, как вызывать сценарии на выполнение и управлять их работой. В системе Linux предусмотрено несколько различных методов вызова сценариев на выполнение, кроме тех, что предусматривают вызов непосредственно из приглашения интерфейса командной строки. В этой главе будет показано, как планировать сценарии на выполнение в конкретное время, а также будут приведены сведения о том, как приостанавливать работу выполняемых сценариев.

Управление сценариями

По мере того как сценарии, создаваемые программистом, становятся все более сложными, у него все чаще возникает необходимость разбираться в нюансах выполнения и управления сценариями в системе Linux. В предыдущих главах рассматривался лишь один способ выполнения сценариев — запуск их непосредственно в интерфейсе командной строки в оперативном режиме. Но это не единственный путь выполнения сценариев в Linux. Предусмотрено также много дополнительных возможностей эксплуатации сценариев командного интерпретатора. В настоящей главе рассматриваются различные способы запуска сценариев. Кроме того, иногда в сценарии возникает бесконечный цикл и приходится искать способы, позволяющие прекратить его работу, не выключая компьютер или не останавливая систему Linux. В данной главе рассматриваются различные способы, которые позволяют указывать, как и когда те или иные сценарии командного интерпретатора должны выполняться в системе.

Обработка сигналов

В системе Linux для обеспечения взаимодействия процессов, функционирующих в системе, применяются сигналы. В главе 4 были описаны различные сигналы Linux и показано, как они используются в системе Linux для останова, запуска и уничтожения процессов. Сигналы можно также использовать для управления функционированием сценария командного интерпретатора. Для этого сценарий должен быть запрограммирован на выполнение определенных команд при получении конкретных сигналов от системы Linux.

ГЛАВА

15

В этой главе...

Обработка сигналов

Выполнение сценариев в фоновом режиме

Выполнение сценариев без привязки к консоли

Управление заданиями

Определение приоритета процесса

Выполнение в заданное время

Резюме

Дополнительные сведения о сигналах Linux

Предусмотрено более 30 сигналов Linux, которые могут быть сформированы системой и приложениями. В табл. 15.1 перечислены наиболее широко применяемые сигналы системы Linux, с которыми обычно приходится сталкиваться при программировании для Linux.

Таблица 15.1. Сигналы Linux

Сигнал	Значение	Описание
1	SIGHUP	Выход из процесса
2	SIGINT	Прерывание процесса
3	SIGQUIT	Остановка процесса
9	SIGKILL	Безусловное завершение процесса
15	SIGTERM	Завершение процесса, если это возможно
17	SIGSTOP	Безусловная остановка, но не завершение процесса
18	SIGTSTP	Остановка или приостановка процесса, но не завершение
19	SIGCONT	Продолжение остановленного процесса

По умолчанию командный интерпретатор `bash` пропускает все полученные им сигналы `SIGQUIT` (3) и `SIGTERM` (15) (это предусмотрено с целью предотвращения возможности непредумышленного завершения работы интерактивного командного интерпретатора). Но командный интерпретатор `bash` обрабатывает все полученные им сигналы `SIGHUP` (1) и `SIGINT` (2).

После получения сигнала `SIGHUP` происходит выход из командного интерпретатора `bash`. Но прежде чем завершить свою работу, командный интерпретатор передает сигнал `SIGHUP` всем запущенным им процессам (таким как выполняемые под его управлением сценарии командного интерпретатора). После получения сигнала `SIGINT` командным интерпретатором происходит лишь прерывание его работы. Ядро Linux прекращает выделять процессорное время для работы командного интерпретатора. Если это происходит, командный интерпретатор передает сигнал `SIGINT` всем запущенным им процессам в качестве уведомления о возникшей ситуации.

Командный интерпретатор передает полученные им сигналы в выполняемые программы сценариев командного интерпретатора для обработки. Но по умолчанию поведение сценариев командного интерпретатора состоит в том, что сигналы, которые могут оказать неблагоприятный эффект на функционирование сценария, должны игнорироваться. Для предотвращения такой ситуации можно запрограммировать сценарий так, чтобы он распознавал сигналы и выполнял необходимые команды по подготовке сценария к ситуациям, складывающимся вследствие появления сигнала.

Выработка сигналов

Командный интерпретатор `bash` позволяет формировать два основных сигнала Linux с использованием комбинаций клавиш. Это становится необходимым, если приходится останавливать или приостанавливать программу, вышедшую из-под контроля.

Прерывание процесса

Комбинация клавиш `<Ctrl+C>` позволяет сформировать сигнал `SIGINT` и отправить его всем процессам, которые выполняются в настоящее время в командном интерпретаторе. Это

можно проверить, вызвав на выполнение команду, которая обычно занимает много времени до завершения, и нажав комбинацию клавиш <Ctrl+C>:

```
$ sleep 100
^C
$
```

Комбинация клавиш <Ctrl+C> просто останавливает текущий процесс, выполняющийся в командном интерпретаторе. Команда `sleep` приостанавливает функционирование на указанное количество секунд. Как правило, приглашение командной строки не появляется вновь до тех пор, пока не истечет значение таймера. Нажатием комбинации клавиш <Ctrl+C> до истечения значения таймера можно вызвать преждевременное завершение работы команды `sleep`.

Приостановка процесса

Вместо завершения процесса можно приостановить его работу, не ожидая, пока процесс полностью ее выполнит. Иногда это может оказаться опасным (например, если в сценарии произошла блокировка важного системного файла), но чаще всего приостановка процесса позволяет просто ознакомиться с тем, что происходит в сценарии, без фактического завершения процесса.

Комбинация клавиш <Ctrl+Z> позволяет сформировать сигнал `SIGTSTP`, который останавливает все процессы, выполняемые в командном интерпретаторе. Следует отличать остановку процесса от его завершения, поскольку после остановки процесса программа продолжает оставаться в памяти и сохраняет способность продолжить работу с того места, где она была прервана. В приведенном ниже разделе “Управления заданиями” будет показано, как снова запустить остановленный процесс.

После выполнения действий, связанных с обработкой комбинации клавиш <Ctrl+Z>, командный интерпретатор сообщает, что процесс был остановлен:

```
$ sleep 100
^Z
[1]+  Stopped                  sleep 100
$
```

Число в квадратных скобках представляет собой *номер задания*, присвоенный командным интерпретатором. Каждый процесс, выполняемый в командном интерпретаторе, принято называть *заданием*. Командный интерпретатор присваивает каждому заданию уникальный номер задания. Первому запущенному процессу присваивается номер задания 1, второму — номер задания 2 и т.д.

Если программист останавливает задание с номером, присвоенным тому сеансу командного интерпретатора, в котором он работает, `bash` выводит предупреждение, что происходит попытка выйти из командного интерпретатора:

```
$ exit
logout
There are stopped jobs.
$
```

Для просмотра перечня остановленных заданий можно использовать команду `ps`:

```
$ ps au
USER PID  %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
rich 20560  0.0  1.2  2688  1624 pts/0    S   05:15   0:00 -bash
rich 20605  0.2  0.4   1564   552 pts/0    T   05:22   0:00 sleep 100
rich 20606  0.0  0.5   2584   740 pts/0    R   05:22   0:00 ps au
$
```

В столбце STAT команда `ps` отображает статус остановленного задания как T. Это указывает, что происходит отслеживание работы команды или процесс ее выполнения остановлен.

Если действительно требуется выйти из командного интерпретатора, в котором все еще остается активным остановленное задание, достаточно лишь еще раз ввести команду `exit`. Произойдет выход из командного интерпретатора, а остановленные задания будут завершены. Еще один вариант состоит в том, что, зная теперь идентификатор процесса остановленного задания, можно воспользоваться командой `kill` для отправки сигнала SIGKILL в целях завершения задания:

```
$ kill -9 20605
$
[1]+  Killed                  sleep 100
$
```

Уничтожение задания происходит так, что вначале не выдается какой-либо ответ. Но в следующий раз после выполнения любого действия, вслед за которым выводится приглашение командного интерпретатора, происходит вывод сообщения, указывающего, что задание было уничтожено. Кроме того, командный интерпретатор каждый раз при формировании своего приглашения отображает статус всех заданий, состояние которых в командном интерпретаторе изменилось. После уничтожения задания также отображается сообщение, указывающее, что задание было уничтожено в ходе его выполнения, после осуществления программистом любого действия, которое вынуждает командный интерпретатор сформировать свое приглашение.

Перехват сигналов

В качестве альтернативы тому, чтобы позволить сценарию пропускать сигналы, можно предусмотреть в сценарии перехват сигналов при их появлении и выполнение других команд. Команда `trap` дает возможность указать, какие сигналы Linux должен отслеживать сценарий командного интерпретатора и перехватывать после формирования их командным интерпретатором. Если сценарий получает один из сигналов, перечисленных в команде `trap`, то предотвращает обработку сигнала командным интерпретатором и вместо этого обрабатывает его локально.

Команда `trap` имеет следующий формат:

```
trap commands signals
```

Этот формат достаточно прост. В командной строке с ключевым словом `trap` достаточно лишь перечислить команды *commands*, выполнение которых программист желает передать командному интерпретатору, наряду с разделенным запятыми списком сигналов *signals*, которые он желает перехватывать. Сигналы можно указывать по числовым значениям или по именам сигналов, принятым в системе Linux.

Ниже приведен простой пример использования команды `trap` для пропуска сигналов SIGTERM и SIGINT.

```
$
$ cat test1
#!/bin/bash
# проверка перехвата сигналов
#
trap "echo ' Sorry! I have trapped Ctrl-C'" SIGINT SIGTERM
echo This is a test program
count=1
```



```
while [ $count -le 10 ]
do
    echo "Loop #$count"
    sleep 5
    count=$(( $count + 1 ))
done
echo This is the end of the test program
$
```

Команда `trap`, применяемая в этом примере, отображает простое текстовое сообщение после каждого обнаружения сигнала `SIGTERM` или `SIGINT`. Поскольку эти сигналы перехватываются, то сценарий становится неуязвимым для пользователя, пытающегося остановить программу с помощью комбинации клавиш `<Ctrl+C>` в ходе работы командного интерпретатора `bash`:

```
$
$ ./test1
This is a test program
Loop #1
Loop #2
Loop #3
^C Sorry! I have trapped Ctrl-C
Loop #4
Loop #5
Loop #6
Loop #7
^C Sorry! I have trapped Ctrl-C
Loop #8
Loop #9
Loop #10
This is the end of the test program
$
```

После каждого использования комбинации клавиш `<Ctrl+C>` в сценарии выполняется инструкция `echo`, указанная в команде `trap`, поэтому не происходит, как обычно, пропуск сигнала и предоставление командному интерпретатору возможности остановить сценарий.

Перехват команды выхода из сценария

Сигналы можно перехватывать не только при выполнении сценария командного интерпретатора, но и при выходе из сценария. Ниже показан удобный способ вызова команд на выполнение в тот момент, когда командный интерпретатор завершает задание.

Для перехвата условия выхода из сценария командного интерпретатора достаточно лишь добавить сигнал `EXIT` к команде `trap`:

```
$ cat test2
#!/bin/bash
# перехват выхода из сценария

trap "echo byebye" EXIT

count=1
while [ $count -le 5 ]
do
```

```

    echo "Loop #${count}"
    sleep 3
    count=$(( $count + 1 ))
done
$
$ ./test2
Loop #1
Loop #2
Loop #3
Loop #4
Loop #5
byebye
$

```

Команда `trap` формирует ловушку, которая активизируется в тот момент, как сценарий подходит к обычному пункту своего завершения, после чего командный интерпретатор выполняет команду, указанную в командной строке `trap`. Ловушка, связанная с сигналом `EXIT`, срабатывает также при преждевременном выходе из сценария:

```

$ ./test2
Loop #1
Loop #2
^Cbyebye
$

```

Если с помощью комбинации клавиш `<Ctrl+C>` происходит отправка сигнала `SIGINT`, сценарий завершает свою работу (поскольку этот сигнал не перечислен в списке команды `trap`), но перед выходом из сценария командный интерпретатор выполняет команду `trap`.

Удаление ловушки

Ловушку, установленную командой `trap`, можно удалить, для чего следует задать в качестве команды тире и привести список сигналов, при обработке которых необходимо вернуться к обычному поведению:

```

$ cat test3
#!/bin/bash
# удаление установленной ловушки

trap "echo byebye" EXIT

count=1
while [ $count -le 5 ]
do
    echo "Loop #${count}"
    sleep 3
    count=$(( $count + 1 ))
done
trap - EXIT
echo "I just removed the trap"
$
$ ./test3
Loop #1

```

```
Loop #2
Loop #3
Loop #4
Loop #5
I just removed the trap
$
```

После того как эта ловушка для сигнала будет удалена, в сценарии сигналы станут пропускаться. Но если получение сигнала произошло до удаления ловушки, то в сценарии его обработка происходит так, как указано в команде `trap`:

```
$ ./test3
Loop #1
Loop #2
^Cbyebye
$
```

В этом примере использовалась комбинация клавиш `<Ctrl+C>` для преждевременного завершения сценария. Но сценарий был завершен до удаления ловушки, поэтому в нем была выполнена команда, указанная в ловушке.

Выполнение сценариев в фоновом режиме

Иногда не совсем удобно вызывать на выполнение сценарий командного интерпретатора непосредственно из интерфейса командной строки. Обработка некоторых сценариев может требовать много времени, и нет смысла оставаться привязанным к интерфейсу командной строки, ожидая, когда эта работа закончится. В то время как происходит выполнение сценария, больше ничего нельзя сделать в терминальном сеансе. К счастью, предусмотрено простое решение этой проблемы.

После вызова на выполнение команды `ps` можно видеть, что в системе Linux одновременно работает значительное количество различных процессов. Безусловно, не все эти процессы связаны с данным конкретным терминальным сеансом. Значительная часть из них функционирует в так называемом *фоновом режиме*. Фоновый режим характеризуется тем, что выполняемый в нем процесс не связан со стандартными потоками терминального сеанса — `STDIN`, `STDOUT` и `STDERR` (см. главу 14).

Возможностью запуска в фоновом режиме можно также пользоваться и в собственных сценариях командного интерпретатора, разрешая им действовать незаметно для постороннего взгляда и не блокировать терминальный сеанс. В следующих разделах показано, как обеспечить выполнение сценариев в фоновом режиме в системе Linux.

Выполнение в фоновом режиме

Задача обеспечения выполнения сценария командного интерпретатора в фоновом режиме является довольно несложной. Для указания в интерфейсе командной строки на то, что сценарий командного интерпретатора должен быть выполнен в фоновом режиме, достаточно лишь поместить символ амперсанда после команды:

```

$ ./test1 &
[1] 1976
$ This is a test program
Loop #1
Loop #2
ls /home/user/Desktop
gnome-screenshot.desktop  konsole.desktop      virtualbox.desktop
gnome-terminal.desktop    ksnapshot.desktop
$ Loop #3
Loop #4
...

$

```

Обнаружив символ амперсанда после команды, командный интерпретатор `bash` прекращает сам выполнять эту команду и запускает для нее отдельный фоновый процесс в системе. В данном случае прежде всего отображается следующая строка:

```
[1] 1976
```

Число в квадратных скобках представляет собой номер задания, присвоенный фоновому процессу командным интерпретатором. Следующее число — это идентификатор процесса (Process ID — PID), который присвоен процессу системой Linux. Каждый процесс, исполняемый в системе Linux, должен иметь уникальный идентификатор процесса.

Сразу после отображения системой этих элементов появляется новое приглашение командной строки. Пользователь возвращается к интерфейсу командного интерпретатора, а вызванная им на выполнение команда функционирует под контролем системы в фоновом режиме.

Начиная с этого момента, можно вводить в приглашении командного интерпретатора новые команды (как показано в данном примере). Но до тех пор пока продолжается функционирование фонового процесса, в нем все еще используется окно терминала для вывода сообщений, относящихся к `STDOUT` и `STDERR`. В данном примере можно видеть, что результаты выполнения сценария `test1` появляются в выводе, чередуясь с результатами всех прочих команд, выполняемых в данном сеансе работы с командным интерпретатором.

После завершения фонового процесса также отображается сообщение на терминале:

```
[1]+  Done                  ./test1
```

В нем показаны номер задания и статус задания (`Done`), наряду с командой, которая использовалась для запуска задания.

Выполнение нескольких низкоприоритетных заданий

В приглашении командной строки можно запустить одновременно любое количество низкоприоритетных заданий:

```

$
$ ./test1 &
[3] 2174
$ This is Test Program 1

$ ./test2 &
[4] 2176

```

```
$ I am Test Program 2

$ ./test3 &
[5] 2178
$ Well this is Test Program

$ ./test4 &
[6] 2180
$ This is Test Program 4

$
```

При запуске каждого нового задания система Linux присваивает ему новый номер задания и идентификатор процесса. Для получения сведений обо всех выполняемых сценариях можно воспользоваться командой `ps`:

```
$ ps au
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START TIME COMMAND
...
user    1826  0.0  0.3  6704  3408 pts/0    Ss   14:07 0:00 bash
user    2174  0.0  0.1  4860  1076 pts/0    S   15:23 0:00 /bin/bash ./test1
user    2175  0.0  0.0  3884    504 pts/0    S   15:23 0:00 sleep 300
user    2176  0.0  0.1  4860  1068 pts/0    S   15:23 0:00 /bin/bash ./test2
user    2177  0.0  0.0  3884    508 pts/0    S   15:23 0:00 sleep 300
user    2178  0.0  0.1  4860  1068 pts/0    S   15:23 0:00 /bin/bash ./test3
user    2179  0.0  0.0  3884    504 pts/0    S   15:23 0:00 sleep 300
user    2180  0.0  0.1  4860  1068 pts/0    S   15:23 0:00 /bin/bash ./test4
user    2181  0.0  0.0  3884    504 pts/0    S   15:23 0:00 sleep 300
user    2182  0.0  0.1  4592  1100 pts/0    R+  15:24 0:00 ps au
$
```

Каждый запущенный в данном сеансе фоновый процесс появится в выходном листинге работающих процессов команды `ps`. Если же все эти процессы будут отображать свои результаты в том сеансе, в котором они были запущены, то может возникнуть значительная путаница. К счастью, существует простой способ решения этой проблемы, который будет рассматриваться в следующем разделе.

Выход из терминального сеанса

При использовании в фоновом режиме процессов, запущенных в терминальном сеансе, необходимо соблюдать осторожность. Необходимо отметить, что, как показано в выводе команды `ps`, каждый из фоновых процессов привязан к терминалу своего терминального сеанса (в данном случае — `pts/0`). По завершении терминального сеанса завершаются и фоновые процессы.

Некоторые эмуляторы терминала выводят предупреждение, что продолжают работу какие-то фоновые процессы, связанные с терминалом, а в других это не предусмотрено. Если желательно, чтобы сценарий продолжал работу в фоновом режиме и после закрытия сеанса работы с консолью, необходимо принять дополнительно еще некоторые меры. Этот процесс рассматривается в следующем разделе.

Выполнение сценариев без привязки к консоли

Иногда возникает необходимость запустить сценарий командного интерпретатора из терминального сеанса, а затем дать ему возможность выполняться в фоновом режиме независимо от того, продолжается ли функционирование терминального сеанса. Это можно сделать с помощью команды `nohup`.

Вызов команды `nohup` приводит к выполнению еще одной команды, блокирующей любые сигналы `SIGHUP`, передаваемые процессу. Благодаря этому не происходит выход из процесса даже после завершения терминального сеанса, в котором он запущен.

Для вызова на выполнение команды `nohup` применяется следующий формат:

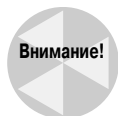
```
$ nohup ./test1 &
[1] 19831
$ nohup: ignoring input and appending output to 'nohup.out'
$
```

Как и при запуске обычного фонового процесса, командный интерпретатор присваивает команде номер задания, а система Linux присваивает номер идентификатора процесса. Различие состоит в том, что после применения команды `nohup` сценарий пропускает любые сигналы `SIGHUP`, отправленные в терминальном сеансе даже во время его закрытия.

Это означает, что команда `nohup` ликвидирует связь процесса с терминалом, поэтому процесс утрачивает ссылки на потоки вывода `STDOUT` и `STDERR`. Чтобы компенсировать потерю этих потоков и сохранить весь вывод, сформированный командой, команда `nohup` автоматически перенаправляет сообщения `STDOUT` и `STDERR` в файл `nohup.out`.

В файле `nohup.out` содержится весь вывод, который при обычных условиях был бы отправлен в окно терминала. После завершения выполнения процесса можно ознакомиться с файлом `nohup.out`, содержащем выходные результаты:

```
$ cat nohup.out
This is a test program
Loop #1
Loop #2
Loop #3
Loop #4
Loop #5
Loop #6
Loop #7
Loop #8
Loop #9
Loop #10
This is the end of the test program
$
```



После запуска еще одной команды с помощью команды `nohup` вывод следующей команды присоединяется к существующему файлу `nohup.out`. Необходимо соблюдать осторожность, запуская несколько команд из одного и того же каталога, поскольку весь их вывод будет отправлен в один и тот же файл `nohup.out`, а это может вызвать путаницу.

Вывод представлен в файле `nohup.out` точно в таком же виде, как если бы процесс выполнялся в командной строке!

Управление заданиями

Выше в данной главе было показано, как использовать определенную комбинацию клавиш для остановки задания, выполняемого в командном интерпретаторе. Система Linux после остановки задания позволяет уничтожить его или возобновить выполнение. Для уничтожения процесса можно воспользоваться командой `kill`. А для возобновления работы остановленного процесса необходимо отправить ему сигнал `SIGCONT`.

Осуществление функций запуска, остановки, уничтожения и возобновления заданий принято называть *управлением заданиями*. Имея в своем распоряжении средства управления заданиями, вы получаете полный контроль над организацией выполнения процессов в конкретной среде командного интерпретатора.

В данном разделе команды, которые используются для просмотра и управления заданиями, выполняемыми в командном интерпретаторе.

Просмотр заданий

С клавиатуры для управления заданиями можно ввести команду `jobs`. Команда `jobs` позволяет просматривать текущие задания, обрабатываемые командным интерпретатором:

```
$ cat test4
#!/bin/bash
# проверка управления заданиями

echo "This is a test program $$"
count=1
while [ $count -le 10 ]
do
    echo "Loop #$count"
    sleep 10
    count=$(( $count + 1 ))
done
echo "This is the end of the test program"
$
$ ./test4
This is a test program 29011
Loop #1
^Z
[1]+  Stopped                  ./test4
$
$ ./test4 > test4out &
[2] 28861
$
$ jobs
[1]+  Stopped                  ./test4
[2]-  Running                  ./test4 >test4out &
$
```

В этом сценарии для отображения идентификатора процесса, который система Linux присваивает сценарию, используется переменная \$\$; затем программа входит в цикл, в котором устанавливается ожидание на время в 10 секунд при каждой итерации. В данном примере первый сценарий был запущен из интерфейса командной строки, а затем остановлен с использованием комбинации клавиш <Ctrl+Z>. После этого было запущено другое задание в качестве фонового процесса, для чего использовался символ амперсанда. Для упрощения изучения этого примера вывод сценария был перенаправлен в файл, чтобы он не появлялся на экране.

После запуска этих двух заданий была вызвана команда jobs для просмотра заданий, назначенных на выполнение в командном интерпретаторе. Команда jobs показывает и остановленные, и действующие задания с указанием номеров заданий и команд, используемых в заданиях.

Команда jobs позволяет задавать несколько различных параметров командной строки, как показано в табл. 15.2.

Таблица 15.2. Параметры команды jobs	
Параметр	Описание
-l	Вывести список идентификаторов процессов наряду с номерами заданий
-n	Перечислять только задания, статус которых изменился со времени вывода последнего уведомления от командного интерпретатора
-p	Включить в список только идентификаторы процессов заданий
-r	Включить в список только выполняемые задания
-s	Перечислить только остановленные задания

Заслуживает внимания то, что в выводе команды jobs присутствуют знаки “плюс” и “минус”. Задание со знаком “плюс” рассматривается как применяемое по умолчанию. Под этим подразумевается задание, на которое ссылаются любые команды управления заданиями в случае отсутствия явного указания номера задания в командной строке. Задание со знаком “минус” представляет собой задание, которое станет применяемым по умолчанию после завершения обработки текущего задания, которое применяется по умолчанию. В любой момент времени только одно задание может быть обозначено знаком “плюс” и еще одно задание — знаком “минус”, независимо от того, сколько заданий выполняется в командном интерпретаторе.

Приведенный ниже пример показывает, как следующее по порядку задание получает статус применяемого по умолчанию после удаления текущего применяемого по умолчанию задания.

```
$ ./test4
This is a test program 29075
Loop #1
^Z
[1]+  Stopped                  ./test4
$
$ ./test4
This is a test program 29090
Loop #1
^Z
[2]+  Stopped                  ./test4
$
$ ./test4
This is a test program 29105
Loop #1
^Z
```



```

[3]+  Stopped                  ./test4
$
$ jobs -l
[1]- 29075 Stopped             ./test4
[2]- 29090 Stopped             ./test4
[3]+ 29105 Stopped             ./test4
$
$ kill -9 29105
$
$ jobs -l
[1]- 29075 Stopped             ./test4
[2]+ 29090 Stopped             ./test4
$

```

В этом примере были запущены, а затем остановлены три отдельных процесса. В листинге команды `jobs` показаны эти три процесса и их статус. Обратите внимание на то, что заданным по умолчанию процессом (тот, который отмечен знаком “плюс”) становится последний запущенный процесс.

После этого была выдана команда `kill` для отправки сигнала `SIGKILL` процессу, применяемому по умолчанию. В следующем листинге `jobs` показано, что применяемым по умолчанию заданием стало то задание, которое раньше было отмечено знаком “минус”.

Возобновление выполнения остановленных заданий

Пользуясь средствами управления заданиями командного интерпретатора `bash`, можно перезапускать любые остановленные задания, независимо от того, выполняются ли они в фоновом режиме или на переднем плане. Возобновленный процесс переднего плана берет на себя контроль над терминалом, в котором вы работаете, поэтому соблюдайте осторожность, прибегая к использованию этого средства.

Чтобы перезапустить задание в фоновом режиме, можно вызвать на выполнение команду `bg` наряду с номером задания:

```

$ bg 2
[2]+ ./test4 &
Loop #2
$ Loop #3
Loop #4

$ jobs
[1]+  Stopped                  ./test4
[2]-  Running                  ./test4 &
$ Loop #6
Loop #7
Loop #8
Loop #9
Loop #10
This is the end of the test program
[2]-  Done                     ./test4
$

```

После перезапуска задания в фоновом режиме снова появляется приглашение интерфейса командной строки, позволяющее вводить следующие команды. Вывод команды `jobs` теперь показывает, что задание действительно выполняется (об этом можно судить и по тому, что вывод этого задания теперь появляется на мониторе).

Чтобы перезапустить задание в режиме переднего плана, используйте команду `fg` наряду с номером задания:

```
$ jobs
[1]+  Stopped                  ./test4
$ fg 1
./test4
Loop #2
Loop #3
```

Запуск задания на выполнение в режиме переднего плана приводит к тому, что приглашение интерфейса командной строки исчезает и не появляется до завершения задания.

Определение приоритета процесса

В многозадачной операционной системе, такой как Linux, за выделение процессорного времени каждому процессу, работающему в системе, отвечает ядро. В любой момент времени процессор фактически выполняет команды только одного процесса, поэтому ядро последовательно отводит определенный квант процессорного времени каждому процессу.

По умолчанию все процессы, запущенные в приглашении командного интерпретатора, имеют в системе Linux одинаковый *приоритет планирования*. Приоритет планирования — это количество процессорного времени, назначаемое ядром конкретному процессу в отличие от других процессов.

Приоритет планирования представляет собой целочисленное значение от -20 (самый высокий приоритет) до $+19$ (самый низкий приоритет). По умолчанию командный интерпретатор `bash` запускает все процессы с приоритетом 0 .



Вызывает затруднения, когда приходится постоянно учитывать, что -20 , самое низкое значение, соответствует наивысшему приоритету, а 19 , самое высокое значение, обозначает самый низкий приоритет. Поэтому достаточно запомнить летучее выражение: "Nice guys finish last" ("Хорошие парни [у девушек] — на последних местах"). Чем "более хорошим", или высоким, является значение, от которого зависит приоритет процесса, тем ниже шанс процесса на получение процессорного времени.

Это означает, что с помощью установки приоритета можно изменить ситуацию, в которой простой сценарий, который может быть выполнен достаточно быстро, получает такой же квант процессорного времени, что и сложный математический алгоритм, на реализацию которого могут потребоваться целые часы.

Иногда необходимо изменить приоритет конкретной команды, например, понизить его, чтобы эта команда не потребляла в единицу времени так много обрабатываемой мощности процессора, или повысить приоритет в целях предоставления команде относительно больше процессорного времени. Это можно сделать с помощью команды `nice`.

Команда nice

Команда `nice` позволяет задать приоритет планирования команды при ее запуске. Чтобы обеспечить выполнение команды с меньшим приоритетом, достаточно задать опцию командной строки `-n` для `nice` с указанием нового уровня приоритета:

```
$ nice -n 10 ./test4 > test4out &
[1] 29476
$ ps al
  F   UID    PID  PPID  PRI   NI  WCHAN  STAT  TTY      TIME COMMAND
100   501  29459  29458   12    0  wait4   S    pts/0    0:00 -bash
000   501  29476  29459   15   10  wait4   SN    pts/0    0:00 /bin/bash
000   501  29490  29476   15   10  nanosl   SN    pts/0    0:00 sleep 10
000   501  29491  29459   14    0  -       R    pts/0    0:00 ps al
$
```

Команда `nice` вынуждает сценарий выполняться с более низким приоритетом. Но при попытке повысить приоритет одной из выполняемых команд возникает такой неожиданный результат:

```
$ nice -n -10 ./test4 > test4out &
[1] 29501
$ nice: cannot set priority: Permission denied

[1]+  Exit 1                  nice -n -10 ./test4 >test4out
$
```

Команда `nice` не позволяет обычным пользователям системы повышать приоритет команд, вызванных ими на выполнение. Это — мера предосторожности, препятствующая тому, чтобы пользователи запускали все свои команды с высоким приоритетом.

Команда `renice`

Иногда возникает необходимость изменить приоритет команды, которая уже выполняется в системе. Для этого предназначена команда `renice`. Она позволяет указать идентификатор функционирующего процесса, для которого должен быть изменен приоритет:

```
$ ./test4 > test4out &
[1] 29504
$ ps al
  F   UID    PID  PPID  PRI   NI  WCHAN  STAT  TTY      TIME COMMAND
100   501  29459  29458   12    0  wait4   S    pts/0    0:00 -bash
000   501  29504  29459    9    0  wait4   S    pts/0    0:00 /bin/bash .
000   501  29518  29504    9    0  nanosl   S    pts/0    0:00 sleep 10
000   501  29519  29459   14    0  -       R    pts/0    0:00 ps al
$ renice 10 -p 29504
29504: old priority 0, new priority 10
$ ps al
  F   UID    PID  PPID  PRI   NI  WCHAN  STAT  TTY      TIME COMMAND
100   501  29459  29458   16    0  wait4   S    pts/0    0:00 -bash
000   501  29504  29459   14   10  wait4   SN    pts/0    0:00 /bin/bash .
000   501  29535  29504    9    0  nanosl   S    pts/0    0:00 sleep 10
000   501  29537  29459   14    0  -       R    pts/0    0:00 ps al
$
```

Команда `renice` автоматически обновляет приоритет планирования выполняющегося процесса. Как и команда `nice`, команда `renice` налагает определенные ограничения, которые перечислены ниже.

- Пользователь может применять команду `renice` по отношению только к тем процессам, владельцем которых он является.
- С помощью команды `renice` можно задавать лишь более низкий приоритет для собственных процессов.
- Пользователь `root` может применять команду `renice` для присваивания любого нового приоритета любому процессу.

Чтобы иметь возможность полностью управлять приоритетами функционирующих процессов, необходимо либо зарегистрироваться в учетной записи `root`, либо воспользоваться командой `sudo`.

Выполнение в заданное время

Практика показывает, что пользователи, которые освоили работу со сценариями, со временем неизбежно сталкиваются с ситуацией, когда требуется выполнить сценарий в заранее заданное время, причем чаще всего в отсутствие самого пользователя. В системе Linux предусмотрены следующие способы запуска сценариев на выполнение в установленное время: команда `at` и таблица `cron`. Каждый из этих способов основан на разных принципах планирования запуска, определяющих, когда и как часто должен выполняться сценарий. Указанные способы рассматриваются в следующих разделах.

Планирование заданий с использованием команды `at`

Команда `at` позволяет указать время, в которое система Linux должна будет вызвать сценарий на выполнение. Команда `at` передает задание в очередь, сопровождая его указаниями в отношении того, когда командный интерпретатор должен выполнить задание. Демон команды `at`, `atd`, функционирует в фоновом режиме и проверяет очередь на наличие заданий, которые должны быть выполнены. В большинстве дистрибутивов Linux этот демон запускается автоматически во время начальной загрузки.

Демон `atd` осуществляет свои проверки в специальном каталоге в системе (обычно `/var/spool/at`) для выявления заданий, направленных в очередь с помощью команды `at`. По умолчанию демон `atd` проверяет этот каталог через каждые 60 секунд. При обнаружении очередного задания демон `atd` определяет, в какое время должно быть выполнено задание. Если заданное время совпадает с текущим временем, демон `atd` вызывает задание на выполнение.

В следующих разделах описано, как использовать команду `at` для передачи задания на выполнение и как управлять заданиями.

Формат команды `at`

Основной формат команды `at` является довольно простым:

```
at [-f filename] time
```

По умолчанию команда `at` передает полученные ею входные данные из потока `STDIN` в очередь. С помощью опции `-f` можно указать имя файла `filename`, в котором находятся команды, предназначенные для чтения и выполнения (файла сценария).

Параметр `time` с указанием времени позволяет определить, когда в системе Linux должно быть выполнено задание. Применимые форматы указания времени открывают большой простор для творчества. Команда `at` распознает много разных форматов времени, указанных ниже.

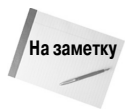
- Стандартное обозначение часов и минут, такое как 10:15.
- Указатель времени до полудня или после полудня, А.М./Р.М., такой как 10:15P.M.
- Часть времени дня, обозначенная определенным именем, например `now`, `noon`, `midnight` или `teatime` (4P.M.).

При указании в команде `at` прошедшего времени команда выполняет это задание в заданное время на следующий день.

Можно указывать не только время выполнения задания, но и конкретный день, для чего применяется несколько различных форматов определения даты.

- Стандартный формат даты, такой как ММДДГГ, ММ/ДД/ГГ или ДД.ММ.ГГ.
- Текстовое обозначение даты, такое как `Jul 4` или `Dec 25`, с указанием или без указания года.
- Предусмотрена также возможность указывать относительные значения времени:
 - `now + 25 minutes`;
 - `10:15P.M. tomorrow`;
 - `10:15 + 7 days`.

При использовании команды `at` указанное в ней задание передается в *очередь заданий*. Из очереди заданий те задания, которые были переданы с помощью команды `at`, поступают на обработку. Предусмотрено 26 различных очередей заданий, рассчитанных на разные уровни приоритета. Очереди заданий обозначаются строчными буквами от `a` до `z`.



Всего лишь несколько лет тому назад в Linux была предусмотрена команда `batch`, которая позволяла применять еще один метод запуска сценариев в последующее время. Уникальной особенностью команды `batch` было то, что с ее помощью можно было запланировать трудоемкий сценарий на выполнение в то время, когда загрузка системы является наиболее низкой. Однако в настоящее время команда `batch` представляет собой просто сценарий, `/usr/bin/batch`, который вызывает команду `at` и передает задание в очередь `b`.

Чем дальше от начала алфавита находится буквенное обозначение очереди заданий, тем ниже приоритет (выше значение `nice`), с которым будет выполняться задание. По умолчанию задания `at` передаются в очередь заданий `at`. Если необходимо выполнить задание с более высоким приоритетом, можно указать другую очередь с использованием параметра обозначения очереди `-q`.

Получение вывода задания

При прогоне задания в системе Linux не предусмотрено связывание с заданием какого-либо терминала. Вместо этого в системе Linux применяется адрес электронной почты пользователя, который передал задание на выполнение, в качестве устройств `STDOUT` и `STDERR`. Любой вывод, предназначенный для `STDOUT` или `STDERR`, отправляется по почте пользователю с помощью системы электронной почты.

Ниже приведен простой пример использования команды `at` для планирования задания на выполнение.

```

$
$ cat test5
#!/bin/bash
#
# проверка команды at
#

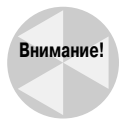
echo This script ran at 'date'
echo This is the end of the script >&2
$
$ date
Mon Oct 18 14:38:17 EDT 2010
$
$ at -f test5 14:39
warning: commands will be executed using /bin/sh
job 57 at Mon Oct 18 14:39:00 2010
$
$ mail
"/var/mail/user": 1 message 1 new
>N 1 user Mon Oct 18 14:39 15/538
↳Output from your job
& 1
Date: Mon, 18 Oct 2010 14:39:00 -0400
Subject: Output from your job 57
To: user@user-desktop
From: user <user@user-desktop>

This script ran at Mon Oct 18 14:39:00 EDT 2010
This is the end of the script

& exit
$

```

Команда `at` формирует предупреждающее сообщение, указывая, какой командный интерпретатор использует система для выполнения сценария (в данном случае `/bin/sh`), а также указывает номер, присвоенный заданию, и время, на которое запланировано выполнение задания.



В большинстве дистрибутивов Linux заданным по умолчанию командным интерпретатором, который применяется в качестве `/bin/sh`, является командный интерпретатор `bash`. Но в дистрибутиве `Ubuntu` в качестве заданного по умолчанию командного интерпретатора используется командный интерпретатор `dash`. Дополнительная информация о командном интерпретаторе `dash` приведена в главе 22.

После завершения задания на мониторе не появляются какие-либо сведения об этом событии, но система формирует сообщение электронной почты, в котором отображается вывод, сформированный сценарием. Если сценарий не вырабатывает никаких выходных данных, то по умолчанию сообщение электронной почты не формируется. Это правило можно изменить с использованием опции `-m` в команде `at`. Эта опция указывает, что должно быть сформировано сообщение электронной почты с обозначением заверщенного задания, даже если сценарий не вырабатывает никакого вывода.

Формирование листинга отложенных заданий

Для просмотра заданий, которые рассматриваются в системе как отложенные на более позднее время, применяется команда `atq`:

```
$
$ at -f test5 15:05
warning: commands will be executed using /bin/sh
job 58 at Mon Oct 18 15:05:00 2010
$
$ at -f test5 15:10
warning: commands will be executed using /bin/sh
job 59 at Mon Oct 18 15:10:00 2010
$
$ at -f test5 15:15
warning: commands will be executed using /bin/sh
job 60 at Mon Oct 18 15:15:00 2010
$
$ at -f test5 15:20
warning: commands will be executed using /bin/sh
job 61 at Mon Oct 18 15:20:00 2010
$$ atq
61    Mon Oct 18 15:20:00 2010 a user
58    Mon Oct 18 15:05:00 2010 a user
59    Mon Oct 18 15:10:00 2010 a user
60    Mon Oct 18 15:15:00 2010 a user
$
```

С помощью этой команды создается листинг заданий, который показывает номер задания, дату и время намеченного выполнения задания системой, а также очередь заданий, в которой хранится задание.

Удаление заданий

Имея в своем распоряжении информацию о том, какие задания, намеченные на выполнение, хранятся в очередях заданий, можно воспользоваться командой `atrm` для удаления отложенного задания, если оно больше не требуется:

```
$
$ atq
59    Mon Oct 18 15:10:00 2010 a user
60    Mon Oct 18 15:15:00 2010 a user
$
$ atrm 59
$
$ atq
60    Mon Oct 18 15:15:00 2010 a user
$
```

Достаточно указать номер задания, которое должно быть удалено. Пользователь может удалять только те задания, которые сам передал на выполнение. Не разрешается удалять задания, переданные другими пользователями.

Планирование регулярного выполнения сценариев

Возможность запланировать сценарий на выполнение в заранее установленное время с помощью команды `at` является очень удобной, но иногда возникает такая необходимость, чтобы определенный сценарий выполнялся в одно и то же время один раз в сутки, один раз в неделю или один раз в месяц. Для этого можно воспользоваться еще одним средством системы Linux, которое позволяет избавиться от необходимости снова и снова передавать одни и те же задания на выполнение с помощью команды `at`.

В системе Linux предусмотрена программа `cron`, которая позволяет планировать задания, предназначенные для выполнения на регулярной основе. Программа `cron` выполняется в фоновом режиме и проверяет специальные таблицы, называемые *таблицами cron*, на наличие заданий, которые намечены для выполнения.

Таблица cron

В таблице `cron` используется специальный формат, позволяющий указывать, когда должно быть выполнено задание. Формат таблицы `cron` является следующим:

```
min hour dayofmonth month dayofweek command
```

Таблица `cron` позволяет задавать в ней записи в качестве конкретных значений и диапазонов значений (таких как 1–5), а также символов-шаблонов (звездочек). Например, если некоторая команда должна выполняться каждый день в 10:15, то можно воспользоваться следующей записью в таблице `cron`:

```
15 10 * * * command
```

Символы-шаблоны, используемые в полях `dayofmonth`, `month` и `dayofweek`, указывают, что программа `cron` должна выполнять команду в каждый день каждого месяца в 10:15. А для указания, например, что команда должна выполняться в 4:15 пополудни в каждый понедельник, можно использовать следующую запись:

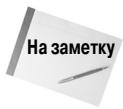
```
15 16 * * 1 command
```

Элемент `dayofweek` может быть задан в виде трехсимвольного текстового значения (`mon`, `tue`, `wed`, `thu`, `fri`, `sat`, `sun`) или числового значения из ряда значений, такого, что 0 соответствует воскресенью, а 6 — субботе.

Ниже приведен еще один пример, который обеспечивает выполнение команды в 12:00 пополудни в первые сутки каждого месяца, для чего используется следующий формат:

```
00 12 1 * * command
```

Элемент `dayofmonth` указывает значение числа месяца (1–31).



Проницательного читателя может заинтересовать вопрос: как обеспечить выполнение той или иной команды в последний день каждого месяца, ведь параметр `dayofmonth` принимает разные значения в различные месяцы? Над решением этой проблемы бились многие программисты Linux и Unix, и в результате удалось найти много разных способов выхода из ситуации. Общепринятый метод состоит во введении инструкции `if-then`, в которой используется команда `date` для проверки того, приходится ли завтрашняя дата на 1-е число:

```
00 12 * * * if ['date +%d -d tomorrow' = 01 ] ; then ; command
```

В этом фрагменте кода предусмотрено выполнение каждый день в 12:00 команд для определения того, является ли текущий день последним днем месяца, и в случае положительного результата вызывается команда `cron`.

В списке команд должны быть указаны полные имена путей к командам или к сценариям командного интерпретатора, подлежащим выполнению. Предусмотрена возможность задавать любые параметры командной строки или символы перенаправления, как и в обычной командной строке:

```
15 10 * * * /home/rich/test4 > test4out
```

Программа `cron` выполняет сценарий с применением учетной записи пользователя, который передал задание на выполнение. Таким образом, пользователь должен иметь надлежащие разрешения на доступ к команде и к выходным файлам, указанным в вызове команды.

Построение таблицы `cron`

Каждый пользователь системы (включая пользователя `root`) может иметь собственную таблицу `cron`, предназначенную для выполнения запланированных заданий. В Linux для обработки таблицы `cron` предусмотрена команда `crontab`. Для вывода существующей таблицы `cron` на внешнее устройство используется параметр `-l`:

```
$ crontab -l
no crontab for rich
$
```

По умолчанию файл таблицы `cron` не создается автоматически для каждого пользователя. Для добавления записей в таблицу `cron` применяется параметр `-e`. Команда `crontab` после вызова с этим параметром запускает текстовый редактор (см. главу 9) применительно к существующей таблице `cron` (или к пустому файлу, если таблица `cron` еще не была создана).

Каталоги `cron`

Если должен быть задан сценарий, для которого не требуется слишком точно указывать время выполнения, то проще использовать один из заранее предусмотренных каталогов сценариев `cron`. Применяются четыре основных каталога для определения времени выполнения в часах, днях, месяцах и неделях: `hourly`, `daily`, `monthly` и `weekly`.

```
$
$ ls /etc/cron.*ly
/etc/cron.daily:
0anacron      aptitude      exim4-base    man-db        sysstat
apt           bsdmainutils  logrotate     popularity-contest
apport        dpkg          mlocate       standard
/etc/cron.hourly:

/etc/cron.monthly:
0anacron

/etc/cron.weekly:
0anacron apt-xapian-index man-db
$
```

Таким образом, если какой-то сценарий должен выполняться один раз в день, достаточно скопировать этот сценарий в каталог `daily`, и `cron` будет вызывать его на выполнение каждый день.

Программа `anacron`

Единственным затруднением при использовании программы `cron` является то, что работа этой программы основана на предположении, будто конкретная система Linux функционирует

24 часа в сутки, 7 дней в неделю. Такое условие не всегда соблюдается, особенно если речь не идет о применении системы Linux в качестве серверной среды.

Если система Linux не функционирует в то время, когда задание намечено на выполнение в таблице `cron`, то запуск задания так и не произойдет. В программе `cron` не предусмотрено выполнение пропущенных заданий задним числом после следующего запуска системы. В целях решения этой проблемы во многие дистрибутивы Linux включена также программа `anacron`.

Если программа `anacron` определяет, что пропущен запуск запланированного задания, то выполняет это задание как можно скорее. Это означает, что если система Linux несколько дней не эксплуатировалась, то после запуска системы указанная программа начинает наверстывать то, что не было сделано, автоматически подгоняя всю работу, которая была пропущена со времени последнего выключения системы.

Это средство часто используется для выполнения сценариев, с помощью которых осуществляется периодическое сопровождение журналов. В противном случае, если система всегда выключена к тому времени, как должен быть применен один из таких сценариев, усечение файлов журналов так и не происходит, в результате чего эти файлы возрастают до нежелательных размеров. Применение программы `anacron` позволяет гарантировать, что сокращение размеров файлов журналов будет происходить по крайней мере после каждого запуска системы.

Программа `anacron` действует только по отношению к программам, находящимся в каталогах `cron`, таких как `/etc/cron.monthly`. В этой программе используются отметки времени для определения того, произошло ли выполнение тех или иных заданий в надлежащем запланированном интервале времени. Файл с отметками времени предусмотрен для каждого каталога `cron` и находится в пути `/var/spool/anacron`:

```
$
$ sudo cat /var/spool/anacron.monthly
20101123
$
```

Программа `anacron` имеет собственную таблицу (обычно находящуюся в пути `/etc/anacrontab`), с помощью которой осуществляется проверка каталогов с заданиями:

```
$
$ cat /etc/anacrontab
# /etc/anacrontab: файл конфигурации для anacron

# Дополнительные сведения см. в anacron(8) и anacrontab(5).

SHELL=/bin/sh
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin

# Применяется для замены записей cron
1 5 cron.daily nice run-parts --report /etc/cron.daily
7 10 cron.weekly nice run-parts --report /etc/cron.weekly
@monthly 15 cron.monthly nice run-parts --report /etc/cron.monthly
$
```

Основной формат таблицы `anacron` немного отличается от формата таблицы `cron`:

period delay identifier command

Запись *period* определяет в сутках, как часто должны выполняться задания. Программа `anacron` использует эту запись для сверки с файлом, содержащим отметки времени заданий. Запись *delay* указывает, через сколько минут после запуска системы программа `anacron`

должна приступать к выполнению пропущенных сценариев. Запись *command* содержит программу *run-parts* и имя каталога сценария *cron*. Программа *run-parts* отвечает за выполнение всех сценариев в переданном ей каталоге.

Необходимо учитывать, что программа *anacron* не выполняет сценарии, находящиеся в каталоге */etc/cron.hourly*. Это связано с тем, что программа *anacron* не предназначена для работы со сценариями, которые должны выполняться чаще, чем раз в сутки.

Запись *identifier* представляет собой уникальную символьную строку, не содержащую пробелы, такую как *cron-weekly*. Эта строка используется для однозначной идентификации задания в записях журналов и сообщениях электронной почты об ошибках.

Запуск с момента загрузки

Последний метод запуска сценариев командного интерпретатора состоит в обеспечении того, чтобы тот или иной сценарий выполнялся автоматически сразу после начальной загрузки системы Linux или после каждого запуска пользователем нового сеанса командного интерпретатора *bash*. Вариант с запуском сценариев во время начальной загрузки обычно остается зарезервированным для специальных сценариев, выполняющих системные функции, такие как настройка сетевого интерфейса или запуск серверного процесса. Однако в работе системного администратора Linux может возникнуть необходимость выполнять определенную функцию при каждой начальной загрузке системы Linux, например, задавать для использования нестандартный файл журнала или запускать какое-то конкретное приложение.

Может также оказаться удобной возможность вызывать на выполнение некоторый сценарий при каждом запуске пользователем нового сеанса работы с командным интерпретатором *bash* (и даже учитывать время, в которое конкретный пользователь запускает командный интерпретатор *bash*). Кроме того, иногда возникает необходимость задать применяемые средства командного интерпретатора для определенного сеанса работы с командным интерпретатором или даже просто подготовить к использованию какой-то конкретный файл.

В этом разделе описано, как настроить систему Linux на выполнение сценариев либо во время начальной загрузки, либо при каждом запуске нового сеанса работы с командным интерпретатором *bash*.

Запуск сценариев во время начальной загрузки

Прежде чем приступать к подготовке сценариев командного интерпретатора для запуска во время начальной загрузки, необходимо ознакомиться с тем, как осуществляется процесс загрузки Linux. В системе Linux для запуска сценариев во время начальной загрузки применяется определенная последовательность, и знания об этом процессе могут помочь при настройке сценариев на выполнение в полном соответствии с намеченным замыслом.

Процесс начальной загрузки

После включения системы Linux в оперативную память компьютера загружается и вызывается на выполнение ядро Linux. Прежде всего в ядре осуществляется запуск процесса *init* типа UNIX System V или процесс *init* типа Upstart, в зависимости от применяемого дистрибутива и его версии. Затем этот процесс берет на себя ответственность за запуск всех прочих процессов в системе Linux.

Процесс init типа System V

В ходе процесса начальной загрузки в процессе init типа System V осуществляется чтение файла `/etc/inittab`. В файле `inittab` перечислены *режимы работы* системы. В разных режимах работы Linux происходит запуск различных программ и сценариев. В табл. 15.3 перечислены режимы работы Linux для дистрибутивов на основе Red Hat.

Таблица 15.3. Режимы работы Linux для дистрибутивов на основе Red Hat

Режим работы	Описание
0	Останов
1	Однопользовательский режим
2	Многопользовательский режим, обычно без поддержки сетевого взаимодействия
3	Полноценный многопользовательский режим с организацией сетей
4	Определяемый пользователем
5	Многопользовательский режим с организацией сетей и графическим сеансом X Window
6	Перезагрузка

В дистрибутивах Linux принято использовать режимы работы, перечисленные в табл. 15.3, для обеспечения более детализированного контроля над тем, какие службы должны быть запущены в системе. Но в дистрибутивах на основе Debian, таких как Ubuntu и Linux Mint, не проводятся различия между режимами работы от 2 до 5, как показано в табл. 15.4.

Таблица 15.4. Режимы работы Linux для дистрибутивов на основе Debian

Режим работы	Описание
0	Останов
1	Однопользовательский режим
2–5	Многопользовательский режим с организацией сетей и графическим сеансом X Window
6	Перезагрузка

В дистрибутиве Ubuntu даже не предусмотрен файл `/etc/inittab`, предназначенный для задания режимов работы. По умолчанию система Ubuntu переходит в режим работы 2. Если потребуется изменить этот заданный по умолчанию режим работы, то необходимо будет создать файл `/etc/inittab` для дистрибутива Ubuntu.

Каждый режим работы определяет, какие сценарии должны быть запущены или остановлены процессом init типа System V. Под *сценариями запуска* подразумеваются сценарии командного интерпретатора, которые запускают приложения, предоставляя необходимые переменные среды для выполнения этих приложений.

До сих пор мы рассматривали довольно простую тему, а с этого момента при дальнейшем описании начальной загрузки на основе процесса init типа System V появляются определенные сложности, в основном связанные с тем, что в различных дистрибутивах Linux предусмотрено размещение сценариев запуска в разных местах. В некоторых дистрибутивах сценарии запуска размещаются в каталоге `/etc/rc#.d`, где # — цифровое обозначение режима работы. В других дистрибутивах используется каталог `/etc/init.d`. Есть и такие дистрибутивы, в которых для размещения сценариев запуска служит каталог `/etc/init.d/rc.d`. Обычно специалисту достаточно рассмотреть структуру каталогов `/etc`, чтобы определить, какой формат используется в данном конкретном дистрибутиве.

Процесс init типа Upstart

Процесс init типа Upstart представляет собой результат применения более нового стандарта управления сервисными процессами, к реализации которого продвигаются многие дистрибутивы Linux. Процесс Upstart сосредоточивается не на режимах работы системы, а на событиях, таких как начальная загрузка системы. В терминологии Upstart загрузка системы именуется *событием запуска*.

В процессе Upstart используются файлы, находящиеся либо в каталоге `/etc/event.d`, либо в каталоге `/etc/init`, для запуска других процессов, в зависимости от дистрибутива и его версии. Для обеспечения обратной совместимости во многих реализациях Upstart все еще вызываются старые сценарии init типа System V из каталогов `/etc/init.d` и (или) `/etc/rc#.d`.

Стандарт применения процесса init типа Upstart все еще продолжает активно развиваться усилиями ведущих разработчиков Linux и в наши дни. Если же создавать сценарии для запуска при возникновении события начальной загрузки системы намеревается пользователь, не принадлежащий к категории разработчиков, то для него более разумнее использовать старый метод, основанный на процессе init типа System V. Этот метод описан в следующем разделе.

Определение сценариев

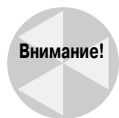
Как правило, лучше самому не заниматься корректировкой отдельных файлов сценариев запуска в дистрибутиве Linux. В дистрибутивах часто предусмотрены инструменты автоматического построения этих сценариев во время добавления серверных приложений, поэтому при попытке вручную вносить изменения в сценарии могут возникнуть проблемы.

Вместо этого можно воспользоваться тем, что в большинстве дистрибутивов Linux предусмотрен локальный файл запуска, предназначенный именно для того, чтобы системный администратор мог вводить сценарии для запуска во время начальной загрузки. Безусловно, имя и местоположение этого файла определены по-разному в различных дистрибутивах Linux. В табл. 15.5 указаны местоположения файлов запуска в пяти популярных дистрибутивах Linux.

Таблица 15.5. Местоположения локальных файлов запуска в дистрибутивах Linux

Дистрибутив	Местоположение файла
debian	<code>/etc/init.d/rc.local</code>
Fedora	<code>/etc/rc.d/rc.local</code>
Mandriva	<code>/etc/rc.local</code>
openSuse	<code>/etc/init.d/boot.local</code>
Ubuntu	<code>/etc/rc.local</code>

В локальном файле запуска можно либо задавать конкретные команды и инструкции, либо вводить имена сценариев, которые должны быть запущены во время начальной загрузки. Следует помнить, что при использовании сценариев необходимо указывать полное имя пути для сценария, чтобы система могла найти его во время начальной загрузки.



Кроме того, в различных дистрибутивах Linux локальные сценарии запуска выполняются на разных этапах процесса загрузки. Иногда сценарий выполняется до осуществления определенных действий, таких как запуск поддержки сети. Ознакомьтесь с документацией по конкретному дистрибутиву Linux, чтобы определить, когда происходит запуск локального сценария начальной загрузки в этом дистрибутиве.

Запуск при открытии нового сеанса работы с командным интерпретатором

В исходном каталоге каждого пользователя содержатся два файла, применяемые командным интерпретатором `bash` для автоматического запуска сценариев и задания переменных среды:

- файл `.bash_profile`;
- файл `.bashrc`.

Командный интерпретатор `bash` выполняет файл `.bash_profile` при запуске нового сеанса работы с командным интерпретатором, когда пользователь вновь входит в систему. В этот файл можно поместить любые сценарии, которые должны быть выполнены во время регистрации пользователя.

Командный интерпретатор `bash` вызывает на выполнение файл `.bashrc` каждый раз с началом нового сеанса работы с командным интерпретатором, включая открытие сеанса при регистрации пользователя. В этом можно убедиться путем добавления простой инструкции `echo` к файлу `.bashrc` в исходном каталоге, а затем запуска командного интерпретатора:

```
$ bash
This is a new shell!!
$
```

Если некоторый сценарий должен выполняться для каждого пользователя в системе, то можно воспользоваться тем, что в большинстве дистрибутивов Linux предусмотрен файл `/etc/bashrc` (обратите внимание на то, что точка перед именем файла `bashrc` отсутствует). Командный интерпретатор `bash` выполняет инструкции в этом файле каждый раз, когда любой пользователь в системе запускает новый экземпляр командного интерпретатора `bash`.

Резюме

Система Linux позволяет управлять сценариями командного интерпретатора с использованием сигналов. Командный интерпретатор `bash` принимает сигналы и передает их любому процессу, выполняемому под управлением процесса командного интерпретатора. Сигналы Linux позволяют легко уничтожить процесс, вышедший из-под контроля, или на время приостановить процесс, выполнение которого занимает много времени.

В сценариях можно использовать инструкцию `trap` для перехвата сигналов и выполнения команд. Это средство предоставляет простой способ управления тем, может ли один пользователь прервать выполнение сценария, запущенного другим пользователем.

По умолчанию при запуске сценария в терминальном сеансе командного интерпретатора работа интерактивного командного интерпретатора приостанавливается до завершения сценария. Предусмотрена возможность обеспечить выполнение сценария или команды в фоновом режиме путем добавления знака амперсанда (`&`) после имени команды. Если сценарий или команда вызывается на выполнение в фоновом режиме, то в интерактивном командном интерпретаторе после этого происходит возврат к интерфейсу командной строки, что позволяет пользователю продолжить ввод команд. Все фоновые процессы, вызванные на выполнение таким образом, остаются привязанными к тому же терминальному сеансу. По завершении терминального сеанса завершаются и фоновые процессы.

Для предотвращения преждевременного завершения фоновых процессов можно воспользоваться командой `nohup`. Эта команда перехватывает все сигналы, предназначенные для ко-

манды, которая в противном случае прекратила бы свою работу, например, как это происходит при выходе из терминального сеанса. Таким образом обеспечивается дальнейшее выполнение сценариев в фоновом режиме даже после завершения терминального сеанса, в котором они запущены.

После перевода процесса в фоновый режим все еще остается возможность управлять тем, что происходит в сценарии. Команда `jobs` позволяет просматривать процессы, запущенные в сеансе работы с командным интерпретатором. После того как становится известным идентификатор задания, присвоенный фоновому процессу, можно использовать команду `kill` для отправки процессу сигналов Linux или команду `fg` для перевода процесса на передний план в сеансе командного интерпретатора. Предусмотрена возможность приостановить функционирующий процесс переднего плана с помощью комбинации клавиш `<Ctrl+Z>`, а затем снова перевести его в фоновый режим, применяя команду `bg`.

Команды `nice` и `renice` позволяют изменить уровень приоритета процесса. Чем ниже приоритет процесса, тем меньше процессорного времени выделяется для этого процесса. Необходимость в снижении приоритета может возникнуть при эксплуатации продолжительных процессов, которые могут потребовать большого количества процессорного времени.

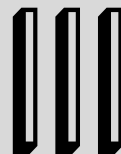
Предусмотрена возможность не только управлять процессами в ходе их выполнения, но и регламентировать время запуска процесса в системе. Вместо вызова сценария на выполнение непосредственно в приглашении интерфейса командной строки можно запланировать процесс на выполнение в другое время. Для решения этой задачи предусмотрено несколько разных способов. Одноразовый вызов сценария в заранее заданное время позволяет осуществить команда `at`. Программа `cron` предоставляет интерфейс, с помощью которого можно обеспечить регулярное выполнение сценариев через запланированные интервалы.

Наконец, в системе Linux предусмотрены файлы сценариев, которые могут использоваться для планирования запуска других сценариев либо во время начальной загрузки системы, либо после каждого открытия пользователем нового сеанса работы с командным интерпретатором `bash`. В состав дистрибутивов входит много файлов, с помощью которых можно задавать перечни сценариев, запускаемых при каждой начальной загрузке системы. Это позволяет системным администраторам выполнять специальные сценарии для сопровождения системы во время начальной загрузки. Аналогичным образом в исходном каталоге каждого пользователя находятся файлы `.bash_profile` и `.bashrc`, предоставляющие возможность размещения сценариев и команд, запуск которых должен происходить при открытии каждого нового сеанса работы с командным интерпретатором. Файл `.bash_profile` обеспечивает выполнение сценариев при каждой регистрации пользователя в системе, а с помощью файла `.bashrc` осуществляется запуск сценариев вместе с запуском каждого нового экземпляра командного интерпретатора.

В следующей главе описано, как ведется разработка функций сценариев. Функции сценариев позволяют один раз записывать блоки кода, а затем использовать их в нескольких местах во всем сценарии.

Усовершенствованные сценарии командного интерпретатора

ЧАСТЬ



В этой части...

Глава 16

Создание функций

Глава 17

Написание сценариев для графических рабочих столов

Глава 18

Общие сведения о редакторах sed и gawk

Глава 19

Регулярные выражения

Глава 20

Дополнительные сведения о редакторе sed

Глава 21

Дополнительные сведения о редакторе gawk

Глава 22

Работа с другими командными интерпретаторами

ГЛАВА

16

В этой главе...

Основные сведения
о применении функций
в сценариях

Возврат значения

Использование переменных
в функциях

Переменные типа массива
и функции

Рекурсивный вызов функций

Создание библиотеки

Использование функций
в командной строке

Резюме

Создание функций

При написании сценариев командного интерпретатора часто возникает необходимость использовать один и тот же код в нескольких местах. Если это лишь небольшой фрагмент кода, то не приходится испытывать значительных затруднений. Но если в сценарии командного интерпретатора необходимо снова и снова вводить большие куски одинакового кода, то существенно возрастают затраты времени на написание программы. В командном интерпретаторе `bash` предусмотрен способ действия в подобных ситуациях, который основан на применении функций, определяемых пользователем. Повторяющиеся фрагменты кода сценария командного интерпретатора можно оформлять в виде функции, а затем использовать уже саму функцию в сценарии столько раз, сколько потребуется. В настоящей главе кратко рассматривается процесс создания собственных функций в сценариях командного интерпретатора и показано, как использовать готовые функции в других приложениях сценариев командного интерпретатора.

Основные сведения о применении функций в сценариях

По мере того как разрабатываемые сценарии командного интерпретатора становятся все более сложными, приходится сталкиваться с тем, что части кода, которые выполняют определенные задачи, должны снова и снова повторяться в одном и том же сценарии. Иногда это достаточно простые фрагменты кода, допустим, с помощью которых отображается текстовое сообщение и запрашивается ответ от пользователя сценария. Но встречаются и такие ситуации, когда в определенных частях кода выполняется

сложное вычисление, которое должно производиться несколько раз в сценарии в составе более общего процесса.

В каждой из таких ситуаций приходится заниматься утомительной работой, снова и снова вводя одни и те же блоки кода в текст сценария. Поэтому в свое время разработчики пришли к мысли, что было бы удобно, если бы можно было записать блок программы один раз, а затем обращаться к этому блоку кода в любом месте сценария, не повторяя его буквально.

В командном интерпретаторе `bash` предусмотрены средства, позволяющие сделать именно это. Одним из таких средств являются *функции* — блоки кода сценария, которым присваивается имя, после чего появляется возможность вызывать эти блоки в сценарии по имени там, где это потребуется. В любое время при возникновении необходимости воспользоваться заранее подготовленным блоком кода в сценарии достаточно ввести имя функции, присвоенное блоку кода (эта операция называется *вызовом функции*). В настоящем разделе описано, как создаются и используются функции в сценариях командного интерпретатора.

Создание функции

Для создания функции в сценариях командного интерпретатора `bash` можно воспользоваться одним из двух форматов. В первом формате используется ключевое слово `function` наряду с именем функции, которым обозначается блок кода:

```
function name {  
    commands  
}
```

Атрибут `name` определяет уникальное имя, присвоенное функции. Каждая функция, определяемая в сценарии, должна получить уникальное имя.

В этом определении `commands` — это одна или несколько команд командного интерпретатора `bash`, из которых состоит данная функция. После вызова функции командный интерпретатор `bash` выполняет каждую из этих команд в том порядке, в котором они присутствуют в определении функции, точно так же, как и при выполнении обычного кода сценария.

Второй формат определения функции в сценарии командного интерпретатора `bash` в большей степени напоминает формат, который применяется для определения функций в других языках программирования:

```
name () {  
    commands  
}
```

Пустые круглые скобки после имени функции указывают на то, что далее следует определение функции. На этот формат распространяются такие же правила именования функций, как и на исходный формат функций сценариев командного интерпретатора.

Использование функций

Чтобы воспользоваться функцией в сценарии, необходимо указать в строке кода имя функции, по такому же принципу, как происходит вызов любой другой команды командного интерпретатора:

```
$ cat test1  
#!/bin/bash  
# использование функции в сценарии
```

```

function func1 {
    echo "This is an example of a function"
}

count=1
while [ $count -le 5 ]
do
    func1
    count=$(( $count + 1 ))
done

echo "This is the end of the loop"
func1
echo "Now this is the end of the script"
$
$ ./test1
This is an example of a function
This is an example of a function
This is an example of a function
This is an example of a function
This is an example of a function
This is the end of the loop
This is an example of a function
Now this is the end of the script
$

```

В данном случае при каждой ссылке на имя функции `func1` командный интерпретатор `bash` обращается к определению `func1 function` и выполняет все команды, предусмотренные в этом определении.

Определение функции не обязательно должно быть приведено в сценарии командного интерпретатора в первую очередь, однако необходимо соблюдать осторожность. При попытке использовать функцию до ее определения появляется сообщение об ошибке:

```

$ cat test2
#!/bin/bash
# использование функции, которая определена не в начале сценария

count=1
echo "This line comes before the function definition"

function func1 {
    echo "This is an example of a function"
}

while [ $count -le 5 ]
do
    func1
    count=$(( $count + 1 ))
done
echo "This is the end of the loop"
func2
echo "Now this is the end of the script"

function func2 {

```

```

    echo "This is an example of a function"
}
$
$ ./test2
This line comes before the function definition
This is an example of a function
This is an example of a function
This is an example of a function
This is an example of a function
This is an example of a function
This is the end of the loop
./test2: func2: command not found
Now this is the end of the script
$

```

Первая функция, `func1`, была определена после нескольких инструкций в сценарии, что вполне допустимо. Когда в сценарии происходит обращение к функции `func1`, командному интерпретатору уже известно, где найти определение этой функции.

Но в этом сценарии предпринимается также попытка использовать функцию `func2` до ее определения. Тем не менее функция `func2` еще не была определена к тому времени, как в сценарии было достигнуто место ее использования, поэтому появилось сообщение об ошибке.

Необходимо также соблюдать определенные правила, касающиеся имен функций. Напомним, что каждое имя функции должно быть уникальным, поскольку в противном случае возникает проблема. Если в сценарии встречается еще одно определение одной и той же функции, то происходит так называемое *переопределение*, и новая версия функции перекрывает ее исходную версию, не вызывая никаких сообщений об ошибке:

```

$ cat test3
#!/bin/bash
# проверка использования повторяющегося имени функции

function func1 {
echo "This is the first definition of the function name"
}

func1

function func1 {
    echo "This is a repeat of the same function name"
}

func1
echo "This is the end of the script"
$
$ ./test3
This is the first definition of the function name
This is a repeat of the same function name
This is the end of the script
$

```

Исходное определение функции `func1` действовало вполне удовлетворительно, но после обработки второго определения функции `func1` во всем последующем коде сценария вызов функции приводит к применению второго определения.

Возврат значения

В командном интерпретаторе `bash` функции рассматриваются как своего рода мини-сценарии, со своим статусом выхода (см. главу 10). Предусмотрены три способа, с помощью которых можно формировать статус выхода для пользовательских функций.

Статус выхода, заданный по умолчанию

По умолчанию статус выхода функции определяется как статус выхода, возвращенный последней командой в функции. После завершения выполнения функции можно воспользоваться стандартной переменной `$?` для определения статуса выхода функции:

```
$ cat test4
#!/bin/bash
# проверка статуса выхода функции

func1() {
    echo "trying to display a non-existent file"
    ls -l badfile
}

echo "testing the function: "
func1
echo "The exit status is: $?"
$
$ ./test4
testing the function:
trying to display a non-existent file
ls: badfile: No such file or directory
The exit status is: 1
$
```

В данном случае статус выхода функции равен 1, поскольку выполнение последней команды в функции окончилось неудачей. Но это не позволяет узнать, окончилось ли выполнение всех прочих команд в функции успешно или нет. Рассмотрим следующий пример:

```
$ cat test4b
#!/bin/bash
# проверка статуса выхода функции

func1() {
    ls -l badfile
    echo "This was a test of a bad command"
}

echo "testing the function:"
func1
echo "The exit status is: $?"
$
$ ./test4b
testing the function:
ls: badfile: No such file or directory
```

```
This was a test of a bad command
The exit status is: 0
$
```

В этот раз последней инструкцией в функции был вызов команды `echo`, который завершился успешно, поэтому функция имеет статус выхода 0, несмотря на то, что вызов одной из команд в функции окончился неудачей. Таким образом, подход, предусматривающий использование заданного по умолчанию статуса выхода функции, не всегда оправдан. К счастью, предусмотрены другие способы формирования статуса выхода функции.

Использование команды `return`

В командном интерпретаторе `bash` для выхода из функции с конкретным статусом выхода может применяться команда `return`. Команда `return` позволяет задать одно целочисленное значение для определения статуса выхода функции, что может служить простым способом задания статуса выхода функции программным путем:

```
$ cat test5
#!/bin/bash
# использование команды return в функции

function dbl {
    read -p "Enter a value: " value
    echo "doubling the value"
    return ${value * 2}
}

dbl
echo "The new value is $?"
$
```

Функция `dbl` удваивает содержащееся в переменной `$value` значение, полученное в результате ввода данных пользователем. Затем в этой функции происходит возврат сформированного результата с помощью команды `return`, а возвращенное значение отображается в сценарии с использованием переменной `$?`.

Однако при использовании данного способа возврата значения из функции следует соблюдать осторожность. Чтобы избежать проблем, необходимо руководствоваться двумя рекомендациями:

- обязательно выполнять выборку возвращаемого значения сразу после завершения функции;
- не забывать о том, что статус выхода должен находиться в пределах от 0 до 255.

Если перед получением значения функции с помощью переменной `$?` будет выполнена любая другая команда, то возвращаемое значение функции окажется потерянным. Следует помнить, что переменная `$?` возвращает статус выхода последней выполненной команды.

Дополнительные ограничения на использование этого способа возврата значения налагают еще одна проблема. Статус выхода должен быть меньше 256, поэтому результат выполнения функции должен представлять собой целочисленное значение, не превышающее 256. Попытка вернуть любое более высокое значение приводит к возникновению ошибки:

```
$ ./test5
Enter a value: 200
```

```
doubling the value
The new value is 1
$
```

Таким образом, данный способ возврата значения не может применяться, если необходимо выполнить возврат из функции, допустим, более крупного целочисленного значения или строкового значения. Вместо этого необходимо использовать другой способ, который рассматривается в следующем разделе.

Использование вывода из функции

По аналогии с тем, что можно перехватывать вывод команды с помощью переменной командного интерпретатора, можно также перехватывать с помощью переменной командного интерпретатора вывод функции. Этот способ может использоваться для получения вывода любого типа из функции для присваивания его переменной:

```
result='dbl'
```

Эта команда присваивает вывод функции `dbl` переменной командного интерпретатора `$result`. Ниже приведен пример использования данного способа в сценарии.

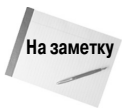
```
$ cat test5b
#!/bin/bash
# использование команды echo для возврата значения

function dbl {
    read -p "Enter a value: " value
    echo ${value * 2}
}

result='dbl'
echo "The new value is $result"
$
$ ./test5b
Enter a value: 200
The new value is 400
$
$ ./test5b
Enter a value: 1000
The new value is 2000
$
```

В новой версии функции теперь применяется инструкция `echo` для отображения результата вычисления. В сценарии просто происходит перехват вывода функции `dbl`, а не просмотр статуса выхода для получения ответа.

В этом примере демонстрируется еще один тонкий прием. Обратите внимание на то, что функция `dbl` фактически выводит два сообщения. Команда `read` выводит краткое сообщение, запрашивая у пользователя удваиваемое значение. Но алгоритмы работы средств поддержки сценариев командного интерпретатора `bash` оказываются достаточно интеллектуальными, чтобы это сообщение не рассматривалось как часть вывода `STDOUT`, поэтому оно пропускается. Если бы для формирования этого сообщения с запросом пользователю применялась инструкция `echo`, то сообщение было бы перехвачено и сохранено в переменной командного интерпретатора наряду с выходным значением.



Использование этого способа позволяет также производить возврат значений с плавающей запятой и строковых значений. Благодаря этому данный способ становится чрезвычайно гибким способом возврата значений из функций.

Использование переменных в функциях

Заслуживает внимания то, что в примере `test5` предыдущего раздела в функции применялась переменная `$value` для хранения обрабатываемого значения. При использовании переменных в функциях необходимо соблюдать некоторые предосторожности, касающиеся того, как должны определяться и обрабатываться эти переменные. Несоблюдение этих предосторожностей является одной из распространенных причин возникновения нарушений в работе сценариев командного интерпретатора. В настоящем разделе рассматриваются несколько новых способов обработки переменных, которые могут применяться не только в функциях сценариев командного интерпретатора, но и в самих сценариях.

Передача параметров в функцию

Как уже было сказано в разделе “Возврат значения”, в командном интерпретаторе `bash` функция рассматривается как своего рода мини-сценарий. Это, в частности, означает, что в функцию можно передавать параметры, как и в обычный сценарий (см. главу 13).

В функциях можно использовать стандартные переменные среды параметров для представления любых параметров, передаваемых в функцию в командной строке. Например, имя функции определено в переменной `$0`, а все параметры в командной строке функции определяются с использованием переменных `$1`, `$2` и т.д. Кроме того, для определения количества параметров, передаваемых в функцию, можно использовать специальную переменную `$#`.

Задавая функцию в сценарии, необходимо приводить параметры в той же командной строке, как и функцию, примерно так:

```
func1 $value1 10
```

Затем в функции можно осуществить выборку значений параметров с использованием переменных среды параметров. Ниже приведен пример применения данного способа для передачи значений в функцию.

```
$ cat test6
#!/bin/bash
# передача параметров в функцию

function addem {
    if [ $# -eq 0 ] || [ $# -gt 2 ]
    then
        echo -1
    elif [ $# -eq 1 ]
    then
        echo $[ $1 + $1 ]
    else
        echo $[ $1 + $2 ]
    fi
}
```

```
echo -n "Adding 10 and 15: "
```

```

value='addem 10 15'
echo $value
echo -n "Let's try adding just one number: "
value='addem 10'
echo $value
echo -n "Now trying adding no numbers: "
value='addem'
echo $value
echo -n "Finally, try adding three numbers: "
value='addem 10 15 20'
echo $value
$
$ ./test6
Adding 10 and 15: 25
Let's try adding just one number: 20
Now trying adding no numbers: -1
Finally, try adding three numbers: -1
$

```

В сценарии `test6` функция `addem` вначале проверяет количество параметров, переданных из сценария. Если параметры не заданы или количество параметров больше двух, функция `addem` возвращает значение `-1`. Если задан только один параметр, функция `addem` складывает значение этого параметра с самим собой для выработки результата. Если заданы два параметра, функция `addem` для формирования результата складывает полученные значения.

Таким образом, в этой функции для получения собственных значений параметров используются специальные переменные среды параметров, поэтому в функции нельзя получить непосредственный доступ к значениям параметров сценария из командной строки вызова сценария. Попытка выполнения сценария в следующем примере окончится неудачей:

```

$ cat badtest1
#!/bin/bash
# попытка обратиться к параметрам сценария из функции

function badfunc1 {
    echo ${ $1 * $2 }
}

if [ $# -eq 2 ]
then
    value='badfunc1'
    echo "The result is $value"
else
    echo "Usage: badtest1 a b"
fi
$
$ ./badtest1
Usage: badtest1 a b
$ ./badtest1 10 15
./badtest1: * : syntax error: operand expected (error token is "**")
The result is
$

```

Несмотря на то что в функции используются переменные `$1` и `$2`, это не те же переменные `$1` и `$2`, которые доступны в основной части сценария. Если же действительно потребуется использовать эти значения в функции, то необходимо передать их вручную при вызове функции:

```
$ cat test7
#!/bin/bash
# попытка обратиться к параметрам сценария из функции

function func7 {
    echo $[ $1 * $2 ]
}

if [ $# -eq 2 ]
then
    value=`func7 $1 $2`
    echo "The result is $value"
else
    echo "Usage: badtest1 a b"
fi
$
$ ./test7
Usage: badtest1 a b
$ ./test7 10 15
The result is 150
$
```

Передача переменных `$1` и `$2` в функцию приводит к тому, что эти переменные становятся доступными для использования в функции, как и любой другой параметр.

Обработка переменных в функции

Программисты на языке сценариев командного интерпретатора сталкиваются еще с одним понятием, неправильное толкование которого может вызывать проблему. Речь идет об *области определения* переменной. Областью определения называют тот участок кода, в котором переменная является видимой. Переменные, определенные в функциях, могут иметь другую область определения по сравнению с обычными переменными. Таким образом, они могут быть скрыты от остальной части сценария.

В функциях используются переменные двух типов:

- глобальные переменные;
- локальные переменные.

В следующих разделах речь пойдет о том, как использовать оба типа переменных в конкретных функциях.

Глобальные переменные

Глобальные переменные — это переменные, которые остаются действительными в любом месте сценария командного интерпретатора. Если глобальная переменная определена в основной части сценария, то ее значение можно получить и в функции. Аналогичным образом к значению глобальной переменной, определенной в функции, можно обратиться в основной части сценария.

По умолчанию все переменные, определенные в сценарии, рассматриваются как глобальные переменные. К переменным, определенным вне функции, можно вполне успешно обращаться непосредственно в функции:

```
$ cat test8
#!/bin/bash
# использование глобальной переменной для передачи значения

function dbl {
    value=$(( $value * 2 ))
}

read -p "Enter a value: " value
dbl
echo "The new value is: $value"
$
$ ./test8
Enter a value: 450
The new value is: 900
$
```

Переменная `$value` определена вне функции, и значение ей присвоено также за пределами функции. При вызове функции `dbl` и эта переменная, и ее значение по-прежнему остаются действительными в функции. А если этой переменной будет присвоено новое значение в функции, то такое новое значение все еще останется действительным при обработке в сценарии ссылки на переменную.

Но организация работы с переменными по такому принципу чревата ошибками, особенно если одна и та же функция должна применяться в различных сценариях командного интерпретатора. Для этого требуется точно знать, какие переменные используются в функции, включая любые переменные, применяемые для вычисления значений, не предназначенных для возврата в сценарий. Ниже приведен пример того, как может возникнуть нарушение в работе.

```
$ cat badtest2
#!/bin/bash
# пример неправильного использования переменных

function func1 {
    temp=$(( $value + 5 ))
    result=$(( $temp * 2 ))
}

temp=4
value=6

func1
echo "The result is $result"
if [ $temp -gt $value ]
then
    echo "temp is larger"
else
    echo "temp is smaller"
fi
$
```

```
$ ./badtest2
The result is 22
temp is larger
$
```

В данном случае переменная *\$temp* применяется и в функции, и в самом сценарии, поэтому происходит изменение ее значения в сценарии, не предусмотренное в функции, что приводит к получению непредвиденного результата. Для решения этой проблемы в функциях может применяться простой способ, как показано в следующем разделе.

Локальные переменные

Вместо использования в функции глобальных переменных можно объявить все переменные, применяемые исключительно в самой функции, как локальные переменные. Для этого достаточно задать ключевое слово *local* перед объявлением переменной:

```
local temp
```

Ключевое слово *local* может быть задано также в операторе присваивания при назначении значения переменной:

```
local temp=$(( $value + 5 )
```

Ключевое слово *local* гарантирует то, что область действия переменной будет ограничиваться только пределами функции. Если в сценарии вне функции появится переменная с тем же именем, то в командном интерпретаторе будет учитываться, что это — две отдельные переменные с разными значениями. Теперь можно легко обеспечить использование переменных функции отдельно от переменных сценария и предоставить общий доступ только к тем переменным, которые должны быть доступными и в основном коде сценария, и в функции:

```
$ cat test9
#!/bin/bash
# пример применения ключевого слова local

function func1 {
    local temp=$(( $value + 5 )
    result=$(( $temp * 2 )
}

temp=4
value=6

func1
echo "The result is $result"
if [ $temp -gt $value ]
then
    echo "temp is larger"
else
    echo "temp is smaller"
fi
$
$ ./test9
The result is 22
temp is smaller
$
```

В данном случае изменение значения переменной *\$temp* в пределах функции *func1* не приводит к изменению значения, присвоенного переменной *\$temp* в основном сценарии.

Переменные типа массива и функции

В главе 5 рассматривался усовершенствованный способ работы с переменными, который позволяет применять одну переменную для хранения нескольких значений с помощью массивов. Задача использования значений переменной с типом массива в функциях немного сложнее, и при этом необходимо руководствоваться некоторыми особыми требованиями. В этом разделе рассматривается методика, позволяющая успешно решать данную задачу.

Передача массивов в функции

Способ передачи переменной с типом массива в функцию сценария является довольно сложным. При попытке передать переменную с типом массива в виде отдельного параметра ничего не получится:

```
$ cat badtest3
#!/bin/bash
# попытка передать переменную с типом массива

function testit {
    echo "The parameters are: $@"
    thisarray=$1
    echo "The received array is ${thisarray[*]}"
}

myarray=(1 2 3 4 5)
echo "The original array is: ${myarray[*]}"
testit $myarray
$
$ ./badtest3
The original array is: 1 2 3 4 5
The parameters are: 1
./badtest3: thisarray[*]: bad array subscript
The received array is
$
```

Если предпринимается попытка использовать переменную с типом массива как параметр функции, то в функцию считывается только первое значение из переменной с типом массива.

Для устранения этой проблемы необходимо разобрать переменную с типом массива на отдельные значения, а затем задать эти значения как параметры функции. В самой же функции можно повторно собрать все параметры в виде новой переменной с типом массива. Ниже приведен пример выполнения указанных действий.

```
$ cat test10
#!/bin/bash
# проверка передачи переменной с типом массива в функцию

function testit {
    local newarray
    newarray=('echo "$@"')
}
```

```

    echo "The new array value is: ${newarray[*]}"
}

myarray=(1 2 3 4 5)
echo "The original array is ${myarray[*]}"
testit ${myarray[*]}
$
$ ./test10
The original array is 1 2 3 4 5
The new array value is: 1 2 3 4 5
$

```

В этом сценарии используется переменная *\$myarray* для сохранения всех отдельных значений элементов массива, которые должны быть помещены в командной строке при вызове функции. После этого в функции вновь формируется переменная с типом массива из параметров командной строки. Затем в этой функции полученный массив может использоваться точно так же, как и любой другой массив:

```

$ cat test11
#!/bin/bash
# сложение значений в массиве

function addarray {
    local sum=0
    local newarray
    newarray=('echo "$@"')
    for value in ${newarray[*]}
    do
        sum=$(( $sum + $value ))
    done
    echo $sum
}

myarray=(1 2 3 4 5)
echo "The original array is: ${myarray[*]}"
arg1='echo ${myarray[*]}'
result='addarray $arg1'
echo "The result is $result"
$
$ ./test11
The original array is: 1 2 3 4 5
The result is 15
$

```

В функции *addarray* осуществляется итерация по значениям элементов массива для получения их общей суммы. В переменную с типом массива *myarray* можно поместить любое количество значений, и функция *addarray* произведет их сложение.

Возврат массивов из функций

При передаче переменной с типом массива из функции обратно в сценарий командного интерпретатора может использоваться аналогичный способ. В функции используется инструкция *echo* для вывода отдельных значений элементов массива в надлежащем порядке, а затем

в сценарии должна быть произведена повторная сборка этих элементов в качестве новой переменной с типом массива:

```
$ cat test12
#!/bin/bash
# возврат значения массива

function arraydbl {
    local origarray
    local newarray
    local elements
    local i
    origarray=('echo "$@"')
    newarray=('echo "$@"')
    elements=$(( $# - 1 ))
    for (( i = 0; i <= $elements; i++ ))
    {
        newarray[$i]=$[ ${origarray[$i]} * 2 ]
    }
    echo ${newarray[*]}
}

myarray=(1 2 3 4 5)
echo "The original array is: ${myarray[*]}"
arg1='echo ${myarray[*]}'
result=('arraydbl $arg1')
echo "The new array is: ${result[*]}"
$
$ ./test12
The original array is: 1 2 3 4 5
The new array is: 2 4 6 8 10
```

В сценарии значение элемента массива передается в функцию `arraydbl` с помощью переменной `$arg1`. В функции `arraydbl` происходит повторная сборка массива для получения новой переменной с типом массива, а также формируется копия для применения в качестве выходной переменной с типом массива. Затем в функции осуществляется итерация по отдельным значениям переменной с типом массива, каждое значение удваивается и помещается в копию переменной с типом массива.

После этого в функции `arraydbl` используется инструкция `echo` для вывода отдельных значений переменной с типом массива. В сценарии вывод функции `arraydbl` применяется для повторной сборки новой переменной с типом массива на основе полученных значений.

Рекурсивный вызов функций

Одной из особенностей, которой локальные переменные наделяют функцию, является *автономность*. Автономная функция не использует никаких ресурсов за пределами функции, кроме тех переменных, которые были переданы ей сценарием в командной строке.

Такая особенность позволяет вызывать функцию *рекурсивно*. Под этим подразумевается вызов функции в самой функции для получения ответа. Обычно в рекурсивной функции предусмотрено базовое значение, к которому в конечном итоге должна сходиться рекурсия. Рекурсия используется во многих усовершенствованных математических алгоритмах для поэтап-

ного уменьшения степени сложности уравнения до тех пор, пока не будет достигнута степень сложности, определяемая базовым значением.

Классическим примером рекурсивного алгоритма является вычисление факториала. *Факториал* числа — это произведение членов натурального ряда чисел, оканчивающегося данным числом. Таким образом, чтобы найти факториал числа 5, необходимо выполнить следующее вычисление:

$$5! = 1 * 2 * 3 * 4 * 5 = 120$$

С использованием рекурсии приведенное здесь выражение можно сокращенно представить в следующей форме:

$$x! = x * (x-1)!$$

или, в описательном виде: “факториал x равен произведению x на факториал $x-1$ ”. Такое вычисление может быть реализовано с помощью следующего простого рекурсивного сценария:

```
function factorial {
    if [ $1 -eq 1 ]
    then
        echo 1
    else
        local temp=$(( $1 - 1 ))
        local result=`factorial $temp`
        echo $[ $result * $1 ]
    fi
}
```

Функция `factorial` вызывает саму себя для вычисления значения факториала:

```
$ cat test13
#!/bin/bash
# использование рекурсии

function factorial {
    if [ $1 -eq 1 ]
    then
        echo 1
    else
        local temp=$(( $1 - 1 ))
        local result=`factorial $temp`
        echo $[ $result * $1 ]
    fi
}

read -p "Enter value: " value
result=`factorial $value`
echo "The factorial of $value is: $result"
$
$ ./test13
Enter value: 5
The factorial of 5 is: 120
$
```

Очевидно, что функция `factorial` является несложной в использовании. После создания подобной функции иногда обнаруживается необходимость ее применения в других сценариях. В следующем разделе будет показано, каким образом можно эффективно решить эту задачу.

Создание библиотеки

Как уже было сказано, функции позволяют избавиться от необходимости многократно вводить одинаковый код в одном сценарии, но иногда повторы кода встречаются в разных сценариях. Очевидно, что в таком случае нет смысла определять в каждом сценарии функцию, включающую повторяющийся код, лишь для того, чтобы использовать ее по одному разу в каждом сценарии.

Для этой проблемы предусмотрено эффективное решение. Командный интерпретатор `bash` позволяет создать *файл библиотеки* для собственных функций, а затем вместо повторного ввода определений функций ссылаться на этот файл библиотеки во всех сценариях, где потребуются представленные в нем функции.

Первый шаг в этом процессе состоит в создании общего файла библиотеки, содержащего функции, которые могут потребоваться в сценариях. Ниже приведен пример простого файла библиотеки с именем `myfuncs`, в котором определены три несложные функции:

```
$ cat myfuncs
# функции для применения в сценариях

function addem {
    echo $[ $1 + $2 ]
}

function multem {
    echo $[ $1 * $2 ]
}

function divem {
    if [ $2 -ne 0 ]
    then
        echo $[ $1 / $2 ]
    else
        echo -1
    fi
}
$
```

Следующий шаг состоит во включении ссылки на файл библиотеки `myfuncs` в файлы сценариев, в которых должны использоваться какие-либо из этих функций. С этого момента задача усложняется.

Проблема состоит в том, что необходимо учитывать область определения функций командного интерпретатора. Как и в отношении переменных среды, функции командного интерпретатора применимы только для сеанса командного интерпретатора, в котором они определены. А после вызова на выполнение сценария командного интерпретатора `myfuncs` в приглашении интерфейса командной строки командного интерпретатора происходит то, что командный интерпретатор создает новый экземпляр командного интерпретатора и выполняет сценарий в этом новом экземпляре. Это приведет к определению трех функций для данного командного интерпретатора, но после вызова другого сценария, в котором должны использоваться эти функции, они окажутся недоступными.

То же происходит и при работе со сценариями. Если будет предпринята попытка просто вызвать на выполнение файл библиотеки как обычный файл сценария, функции из библиотеки не появятся в том сценарии, где они требуются:

```
$ cat badtest4
#!/bin/bash
# using a library file the wrong way
./myfuncs

result='addem 10 15'
echo "The result is $result"
$
$ ./badtest4
./badtest3: addem: command not found
The result is
$
```

Ключом к использованию библиотек функций является команда `source`, которая выполняет команды в контексте текущего командного интерпретатора вместо создания нового командного интерпретатора для их выполнения. Команда `source` применяется для выполнения сценария с файлом библиотеки непосредственно в сценарии командного интерпретатора. В результате функции библиотеки становятся доступными для сценария.

Команда `source` имеет псевдоним, применяемый для ее сокращенного обозначения, который принято называть *оператором точки*. Чтобы использовать в сценарии командного интерпретатора в качестве источника (`source`) текст из файла библиотеки `myfuncs`, достаточно ввести в сценарий следующую строку:

```
. ./myfuncs
```

В данном примере предполагается, что файл библиотеки `myfuncs` находится в том же каталоге, что и сценарий командного интерпретатора. В противном случае потребуется задать соответствующий путь доступа к этому файлу. Ниже приведен пример создания сценария, в котором используется файл библиотеки `myfuncs`.

```
$ cat test14
#!/bin/bash
# использование функций, определенных в файле библиотеки
. ./myfuncs

value1=10
value2=5
result1='addem $value1 $value2'
result2='multem $value1 $value2'
result3='divem $value1 $value2'
echo "The result of adding them is: $result1"
echo "The result of multiplying them is: $result2"
echo "The result of dividing them is: $result3"
$
$ ./test14
The result of adding them is: 15
The result of multiplying them is: 50
The result of dividing them is: 2
$
```

В этом сценарии вызов функций, определенных в файле библиотеки `myfuncs`, происходит вполне успешно.

Использование функций в командной строке

Функции сценария можно применять для создания довольно сложных операций. Иногда возникает необходимость в использовании таких функций непосредственно в приглашении интерфейса командной строки.

По аналогии с тем, что функции сценария можно использовать в качестве команды в сценарии командного интерпретатора, предусмотрена также возможность вызывать функции сценария как команды в интерфейсе командной строки. Это — удобное средство, поскольку после определения функции в командном интерпретаторе появляется возможность вызывать эту функцию из любого каталога в системе, не задумываясь над тем, позволяет ли переменная среды PATH получить доступ к сценарию. Задача заключается лишь в том, чтобы командный интерпретатор распознал такую функцию как применимую в нем. Для решения этой задачи предусмотрен целый ряд способов.

Создание функций в командной строке

Командный интерпретатор интерпретирует команды по мере их ввода, поэтому существует возможность определить функцию непосредственно в командной строке. Это можно сделать с помощью одного из двух способов.

Первый способ предусматривает определение всей функции в одной строке:

```
$ function divem { echo $[ $1 / $2 ]; }
$ divem 100 5
20
$
```

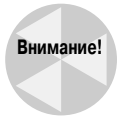
Определяя функцию в командной строке, следует помнить, что после каждой команды должна стоять точка с запятой, чтобы командный интерпретатор мог распознать, где кончается одна команда и начинается другая:

```
$ function doubleit { read -p "Enter value: " value; echo $[
  $ value * 2 ]; }
$
$ doubleit
Enter value: 20
40
$
```

Второй способ состоит в использовании нескольких строк для определения функции. При подготовке такого определения в командном интерпретаторе bash применяется вторичное приглашение, которое указывает на готовность к приему следующей команды. При использовании этого способа нет необходимости задавать точку с запятой в конце каждой команды; достаточно нажимать клавишу <ВВОД>:

```
$ function multem {
> echo $[ $1 * $2 ]
> }
$ multem 2 5
10
$
```

Ввод фигурной скобки в конце определения функции рассматривается командным интерпретатором как завершение задания функции.



При создании функций в командной строке необходимо соблюдать чрезвычайную осторожность. Если функция, предназначенная для использования, получает то же имя, что и встроенная команда или другая существующая команда, то определение функции перекрывает исходную команду.

Определение функций в файле `.bashrc`

Очевидным недостатком определения функций командного интерпретатора непосредственно в командной строке является то, что после выхода из командного интерпретатора введенные в нем функции становятся больше не доступными. Эта проблема вызывает самые значительные затруднения при использовании сложных функций.

Гораздо более удобный способ состоит в определении функции в том месте, где возможна ее повторная загрузка командным интерпретатором при каждом запуске нового экземпляра командного интерпретатора.

Наиболее подходящим для этого местом является файл `.bashrc`. Командный интерпретатор `bash` ищет этот файл в исходном каталоге пользователя при каждом запуске, будь то запуск в интерактивном режиме или в результате вызова нового командного интерпретатора из существующего командного интерпретатора.

Непосредственное определение функций

Для пользователя предусмотрена возможность определять функции непосредственно в файле `.bashrc` в его исходном каталоге. В большинстве дистрибутивов Linux в файле `.bashrc` уже заданы некоторые функции, которые действительно необходимы, поэтому их не следует удалять. Достаточно лишь ввести дополнительные функции в конце существующего файла. Ниже приведен пример того, как это можно сделать.

```
$ cat .bashrc
# .bashrc

# Включение глобальных определений
if [ -r /etc/bashrc ]; then
    . /etc/bashrc
fi

function addem {
    echo $[ $1 + $2 ]
}
$
```

Заданные определения функций вступают в силу только после следующего запуска нового командного интерпретатора `bash`. После того как это произошло, функцию можно использовать в любом месте системы.

Использование в качестве исходного текста файлов с определениями функций

По аналогии со сценариями командного интерпретатора, команду `source` (или ее псевдоним — оператор точки) можно использовать для добавления функций из существующего файла библиотеки в сценарий `.bashrc`:

```
$ cat .bashrc
# .bashrc

# Включение глобальных определений
if [ -r /etc/bashrc ]; then
    . /etc/bashrc
fi

. /home/rich/libraries/myfuncs
$
```

При этом необходимо предусмотреть указание надлежащего имени пути для ссылки на файл библиотеки, который должен быть найден командным интерпретатором `bash`. При следующем запуске командного интерпретатора все функции из библиотеки становятся доступными в интерфейсе командной строки:

```
$ addem 10 5
15
$ multem 10 5
50
$ divem 10 5
2
$
```

Дополнительное удобство состоит в том, что командный интерпретатор передает также все определения функций в дочерние процессы командного интерпретатора, поэтому единожды заданные функции автоматически становятся доступными для любых сценариев командного интерпретатора, вызываемых на выполнение в сеансе командного интерпретатора. В этом можно убедиться, написав сценарий, в котором используются необходимые функции, не задавая при этом определения функций и не используя ссылки на эти определения:

```
$ cat test15
#!/bin/bash
# использование функции, определенной в файле .bashrc

value1=10
value2=5
result1=`addem $value1 $value2`
result2=`multem $value1 $value2`
result3=`divem $value1 $value2`
echo "The result of adding them is: $result1"
echo "The result of multiplying them is: $result2"
echo "The result of dividing them is: $result3"
$
$ ./test15
The result of adding them is: 15
The result of multiplying them is: 50
The result of dividing them is: 2
$
```

Очевидно, что в этом сценарии командного интерпретатора вызываемые функции действовали вполне успешно и при этом не потребовалось ссылаться на файл библиотеки.

Резюме

Функции сценария командного интерпретатора позволяют определять в одном месте фрагменты кода сценария, которые должны неоднократно повторяться по тексту сценария. Вместо того чтобы неоднократно вводить один и тот же блок кода, можно создать функцию, содержащую этот блок кода, а затем просто указывать имя функции в сценарии. Командный интерпретатор `bash` переходит к блоку кода, определенному в функции, каждый раз, когда в сценарии встречается имя функции.

Предусмотрена возможность создавать даже такие функции сценария, которые возвращают значения. Благодаря этому появляется возможность определять функции, которые взаимодействуют со сценарием и возвращают числовые или символьные данные. Функции сценария могут возвращать числовые данные с использованием статуса выхода последней команды в функции или с помощью команды `return`. Команда `return` позволяет задавать программным путем конкретное значение статуса выхода функции с учетом результата выполнения функции.

В функциях можно также возвращать значения с помощью стандартной инструкции `echo`. Предусмотрена возможность с использованием символа обратной одинарной кавычки перехватывать выходные данные, как и при работе с любой другой командой командного интерпретатора. Это позволяет возвращать из функции данные любого типа, включая строки и числа с плавающей запятой.

В функциях можно использовать переменные командного интерпретатора, присваивая значения переменным и получая значения из существующих переменных. Благодаря этому появляется возможность передавать в функции сценариев и из функций данные любого типа, осуществляя тем самым обмен данными с основной частью программы сценария. Функции позволяют также определять локальные переменные, которые доступны только в блоке кода самой функции. Локальные переменные дают возможность создавать автономные функции, в ходе работы которых исключено взаимодействие с какими-либо переменными или процессами в основном сценарии командного интерпретатора.

Функции могут также вызывать другие функции, включая самих себя. Вызов в функции самой этой функции называется *рекурсией*. В рекурсивных функциях часто предусматривается применение базового значения, которое является конечным значением в функции. Рекурсивная функция продолжает вызывать саму себя, уменьшая каждый раз значение некоторого параметра, до тех пор, пока не достигается базовое значение.

Если пользователю в своих сценариях командного интерпретатора часто приходится изменять одни и те же определенные функции, то можно создать файлы библиотек для функций сценариев. Для включения файлов библиотек в любой файл сценария командного интерпретатора можно применять команду `source` или ее псевдоним, оператор точки. Эту операцию принято называть *заданием ссылки* на файл библиотеки. При этом командный интерпретатор не выполняет файл библиотеки, а обеспечивает доступ к функциям в том сценарии командного интерпретатора, который выполняется в настоящее время. Тот же способ организации работы можно использовать для создания функций, которые становятся применимыми в обычной командной строке командного интерпретатора. Предусмотрена возможность определять функции непосредственно в командной строке или добавлять определения функций в файл `.bashrc`, после чего эти функции становятся доступными в каждом вновь открытом сеансе командного интерпретатора. Это — удобный способ создания вспомогательных программ, которые могут использоваться независимо от того, обеспечен ли к ним доступ с помощью переменной среды `PATH`.

В следующей главе рассматривается применение текстовой графики в сценариях. В наши дни, когда находят все более широкое распространение современные графические интерфейсы, обычный текстовый интерфейс иногда рассматривается как неприемлемый. Командный интерпретатор `bash` предоставляет некоторые удобные способы включения простых графических средств в сценарии, что способствует существенному улучшению работы.

Написание сценариев для графических рабочих столов

ГЛАВА

17

В этой главе...

Создание текстовых меню

Организация работы по такому же принципу, как в Windows

Применение графического режима

Резюме

Когда-то сценарии командного интерпретатора представляли собой один из наиболее распространенных способов организации работы в системе, но со временем они стали восприниматься как унылые и скучные. Но так не должно быть, если сценарий предназначен для выполнения в графической среде. Предусмотрен широкий спектр возможностей обеспечения взаимодействия с пользователем сценария, не основанных исключительно на инструкциях `read` и `echo`. В настоящей главе рассматриваются несколько способов, которые можно использовать для придания привлекательности интерактивным сценариям, чтобы они не выглядели столь старомодными.

Создание текстовых меню

Наиболее распространенный способ создания интерактивного сценария командного интерпретатора состоит в использовании *меню*. Меню открывает перед пользователем список возможных опций, изучая который пользователи могут точно определить, что позволяет сделать сценарий.

После вызова на выполнение сценариев с помощью меню обычно происходит очистка области отображения, а затем разворачивается перечень доступных опций.

Пользователь может выбирать опции, нажимая клавиши с буквами или цифрами, назначенными каждой опции. Пример компоновки типичного меню показан на рис. 17.1.

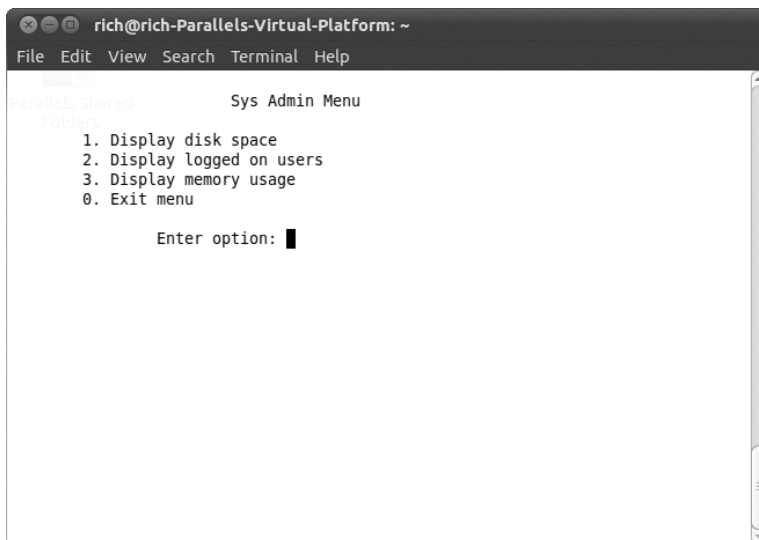


Рис. 17.1. Отображение меню с помощью сценария командного интерпретатора

В основе организации работы меню сценария командного интерпретатора лежит команда `case` (см. главу 11), которая обеспечивает выполнение конкретных команд в зависимости от того, какой символ выбран пользователем в меню.

В следующих разделах описаны шаги, которые должны быть выполнены для создания сценария командного интерпретатора на основе меню.

Создание компоновки меню

Безусловно, первый шаг в создании меню состоит в том, чтобы определить, какие элементы должны отображаться в меню, и расположить их на экране намеченным способом.

Перед развертыванием меню обычно следует очищать экран монитора. Это позволяет отобразить меню "с чистого листа", на экране, на котором нет постороннего текста.

Для удаления всего текста, имеющегося на экране, служит команда `clear`, в которой используются данные `terminfo` текущего терминального сеанса (см. главу 2). После выполнения команды `clear` можно приступить к использованию команды `echo` для отображения элементов меню.

По умолчанию команда `echo` позволяет отображать только такие текстовые символы, которые могут быть выведены на печать. Но при создании элементов меню часто возникает необходимость использовать непечатаемые элементы, такие как знаки табуляции и символы обозначения конца строки. Чтобы иметь возможность включать эти символы в команду `echo`, следует использовать опцию `-e`. Таким образом, команда

```
echo -e "1.\tDisplay disk space"
```

приводит к получению следующей выходной строки:

```
1.      Display disk space
```

Итак, указанная опция значительно облегчает формирование компоновки элементов меню. Меню, которое выглядит вполне достойно, можно создать с помощью лишь нескольких команд `echo`:

```
clear
echo
echo -e "\t\t\tSys Admin Menu\n"
echo -e "\t1. Display disk space"
echo -e "\t2. Display logged on users"
echo -e "\t3. Display memory usage"
echo -e "\t0. Exit menu\n\n"
echo -en "\t\tEnter option: "
```

Опция `-en` в последней инструкции формирования меню сценария указывает, что после вывода текстовой строки не нужно добавлять символ обозначения конца строки. Это позволяет придать меню более профессиональный внешний вид, поскольку курсор останется в конце строки, указывая, что должен последовать ввод символа пользователем.

Последней частью процесса создания меню является обеспечение получения ввода от пользователя. Для этого служит команда `read` (см. главу 13). Предполагается, что для выбора элемента меню должен быть введен только один символ, поэтому целесообразно задать в команде `read` опцию `-n`, которая указывает, что после нажатия единственного символа ввод прекращается. Это дает возможность пользователю вводить обозначение элемента меню, не нажимая клавишу <ВВОД>:

```
read -n 1 option
```

После этого необходимо создать функции меню.

Создание функций меню

Опции меню сценария командного интерпретатора проще создавать как группу отдельных функций. Это позволяет организовать работу с меню, используя простую, краткую команду `case`, проверка правильности которой не создает сложности.

Для этого необходимо подготовить отдельные функции командного интерпретатора для каждой опции меню. Первый шаг в создании сценария командного интерпретатора для поддержки меню состоит в определении того, какие функции должен выполнять сценарий. После этого необходимо отдельно оформить каждую функцию в коде сценария.

Обычно принято создавать *функции-заглушки* вместо тех функций, которые еще не реализованы. Функция-заглушка представляет собой функцию, которая не содержит никаких команд или включает лишь инструкцию `echo`, указывающую, что в конечном итоге должна выполнять данная опция:

```
function diskpace {
    clear
    echo "This is where the diskpace commands will go"
}
```

Это позволяет обеспечить бесперебойную работу меню и продолжать заниматься разработкой нереализованных функций. Иными словами, функции-заглушки дают возможность передать сценарий формирования меню в эксплуатацию, даже если он еще не совсем закончен. Следует отметить, что код определения функции начинается с команды `clear`. Это позволяет начать выполнение функции на чистом экране монитора, на котором уже не показано меню.

Разработка сценария командного интерпретатора на основе меню может быть дополнительно упрощена путем создания самой компоновки меню с помощью функции:

```
function menu {
    clear
    echo
    echo -e "\t\t\tSys Admin Menu\n"
    echo -e "\t1. Display disk space"
    echo -e "\t2. Display logged on users"
    echo -e "\t3. Display memory usage"
    echo -e "\t0. Exit program\n\n"
    echo -en "\t\tEnter option: "
    read -n 1 option
}
```

Это позволяет упростить развертывание меню на экране и в первый раз, и после выполнения любой команды, поскольку достаточно лишь вызвать функцию `menu`.

Добавление средств организации работы меню

После создания компоновки меню и функций, относящихся к отдельным опциям, остается лишь ввести в действие средства организации работы меню, чтобы привязать функции к компоновке. Как уже было сказано, для этого применяется команда `case`.

Команда `case` должна вызывать соответствующую функцию после выбора в меню символа, которым обозначается эта функция. Рекомендуется всегда использовать для перехвата всех неверно заданных опций меню предусмотренный по умолчанию в команде `case` специальный символ (звездочку).

Пример применения команды `case` в типичном меню показан в следующем коде:

```
menu
case $option in
0)
    break ;;
1)
    diskspace ;;
2)
    whoseon ;;
3)
    memusage ;;
*)
    clear
    echo "Sorry, wrong selection";;
esac
```

В этом коде прежде всего используется функция `menu` для очистки экрана монитора и отображения меню. Команда `read` в функции `menu` приостанавливает выполнение сценария до тех пор, пока пользователь не нажмет какой-то символ на клавиатуре. Сразу после этого вступает в действие команда `case`, которая вызывает соответствующую функцию с учетом полученного символа. После выполнения функции осуществляется выход из команды `case`.

Соединение описанных компонентов в одном сценарии

В предыдущих разделах было описано, из каких частей состоит сценарий командного интерпретатора на основе меню, а в данном разделе показано, как соединить эти части, а также описано их взаимодействие. Ниже приведен законченный пример сценария с меню.

```
$ cat menu1
#!/bin/bash
# простое меню сценария

function diskspace {
    clear
    df -k
}

function whoseon {
    clear
    who
}

function memusage {
    clear
    cat /proc/meminfo
}

function menu {
    clear
    echo
    echo -e "\t\t\tSys Admin Menu\n"
    echo -e "\t1. Display disk space"
    echo -e "\t2. Display logged on users"
    echo -e "\t3. Display memory usage"
    echo -e "\t0. Exit program\n\n"
    echo -en "\t\tEnter option: "
    read -n 1 option
}

while [ 1 ]
do
    menu
    case $option in
        0)
            break ;;
        1)
            diskspace ;;
        2)
            whoseon ;;
        3)
            memusage ;;
        *)
            clear
    esac
done
```

```

        echo "Sorry, wrong selection";;
    esac
    echo -en "\n\n\t\t\tHit any key to continue"
    read -n 1 line
done
clear
$

```

В этом меню применяются три функции для получения административной информации об использовании системы Linux, для чего служат обычные команды. В сценарии используется цикл `while`, в котором снова и снова вызывается меню, до тех пор, пока пользователь не выберет опцию 0. Выбор данной опции приводит к вызову команды `break` для выхода из цикла `while`.

Аналогичный шаблон может использоваться для создания любого интерфейса меню с помощью сценария командного интерпретатора. А с помощью меню часто можно значительно упростить работу пользователей.

Использование команды `select`

Читатель мог заметить, что трудности при создании текстового меню наполовину определяются необходимостью просто создать компоновку меню и получить ответ, введенный пользователем. В командном интерпретаторе `bash` предусмотрена небольшая, но очень удобная вспомогательная программа, которая позволяет организовать всю эту работу автоматически.

Команда `select` дает возможность создавать меню с применением единственной командной строки, а затем получать введенный ответ и автоматически его обрабатывать. Команда `select` имеет следующий формат:

```

select variable in list
do
    commands
done

```

Параметр `list` представляет собой разделенный запятыми список текстовых элементов `list`, из которых должно состоять меню. Команда `select` отображает каждый элемент в списке как нумерованную опцию, а затем выводит специальное приглашение к выбору элемента `variable`, определяемое переменной среды `PS3`.

Ниже приведен простой пример команды `select` в действии.

```

$ cat smenu1
#!/bin/bash
# использование функции select в меню

function diskpace {
    clear
    df -k
}

function whoseon {
    clear
    who
}

function memusage {

```

```

clear
cat /proc/meminfo
}

PS3="Enter option: "
select option in "Display disk space" "Display logged on users"
"Display memory usage" "Exit program"
do
    case $option in
        "Exit program")
            break ;;
        "Display disk space")
            diskusage ;;
        "Display logged on users")
            whoseon ;;
        "Display memory usage")
            memusage ;;
        *)
            clear
            echo "Sorry, wrong selection";;
    esac
done
clear
$

```

После вызова на выполнение этой программы автоматически формируется следующее меню:

```

$ ./smenu1
1) Display disk space      3) Display memory usage
2) Display logged on users 4) Exit program
Enter option:

```

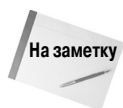
При использовании команды `select` следует помнить, что результирующее значение, сохраняемое в переменной, представляет собой всю текстовую строку содержимого элемента меню, а не только число, связанное с этим элементом. Поэтому в инструкциях `case` необходимо предусмотреть сравнение строковых значений.

Организация работы по такому же принципу, как в Windows

Переход к использованию текстовых меню в интерактивных сценариях представляет собой шаг в правильном направлении, но условия взаимодействия с пользователем все еще требуют значительного улучшения, особенно если эти условия приходится сравнивать с теми, что предоставляет мир графики Windows. К счастью, к нам на выручку пришли некоторые очень находчивые люди, которые посвящают свою деятельность созданию программ с открытым исходным кодом.

К числу таких программ относится пакет `dialog` — изящный небольшой инструмент, первоначально созданный Савио Ламом (Savio Lam) и в настоящее время поддерживаемый Томасом Э. Дики (Thomas E. Dickey). Этот пакет позволяет создавать стандартные диалоговые

окна Windows в текстовой среде с использованием экранирующих управляющих кодов ANSI. Такие диалоговые окна можно легко включить в сценарии командного интерпретатора для организации взаимодействия с пользователями сценария. В данном разделе описан пакет `dialog` и показано, как использовать его в сценариях командного интерпретатора.



Пакет `dialog` не во всех дистрибутивах Linux устанавливается по умолчанию. Но даже если этот пакет не установлен по умолчанию, то его почти всегда можно найти в репозитории программного обеспечения для конкретного дистрибутива, поскольку он является весьма популярным. Просмотрите документацию к используемому дистрибутиву Linux, чтобы узнать, как загрузить пакет `dialog`. Следующая команда командной строки позволяет установить указанный пакет для дистрибутива Ubuntu Linux:

```
sudo apt-get install dialog
```

Эта команда устанавливает сам пакет `dialog` и все необходимые библиотеки для конкретной системы.

Пакет `dialog`

В команде `dialog` используются параметры командной строки для определения того, какой *графический элемент* Windows должен быть сформирован. Термин *графический элемент* (widget) применяется в пакете `dialog` для обозначения графического объекта, примерно соответствующего одному из объектов Windows. Пакет `dialog` в настоящее время поддерживает типы графических элементов, которые приведены в табл. 17.1.

Таблица 17.1. Графические элементы пакета `dialog`

Графический элемент	Описание
<code>calendar</code>	Предоставляет календарь, в котором может быть выбрана дата
<code>checklist</code>	Отображает несколько элементов и позволяет указывать, какие из этих элементов должны быть включены или выключены
<code>form</code>	Позволяет построить форму с надписями и текстовыми полями, предназначенными для заполнения
<code>fselect</code>	Предоставляет окно выбора файла, с помощью которого можно перейти к месту нахождения файла и выбрать его
<code>gauge</code>	Отображает измерительную шкалу, показывающую процентную долю выполнения
<code>infobox</code>	Отображает сообщение, на которое не требуется ответ
<code>inputbox</code>	Отображает отдельное текстовое поле формы для ввода текста
<code>inputmenu</code>	Предоставляет редактируемое меню
<code>menu</code>	Отображает список с вариантами, предназначенными для выбора
<code>msgbox</code>	Отображает сообщение и требует, чтобы пользователь щелкнул на кнопке OK
<code>pause</code>	Отображает измерительную шкалу и показывает статус прохождения указанного периода паузы
<code>passwordbox</code>	Отображает отдельное текстовое поле, в котором введенный текст становится скрытым
<code>passwordform</code>	Отображает форму с надписями и полями скрытого текста
<code>radiolist</code>	Предоставляет группу элементов меню, среди которых может быть выбран только один элемент
<code>tailbox</code>	Отображает текст из файла в окне прокрутки с использованием команды <code>tail</code>
<code>tailboxbg</code>	То же, что и <code>tailbox</code> , но работает в фоновом режиме

Графический элемент	Описание
textbox	Отображает содержимое файла в окне прокрутки
timebox	Предоставляет окно для выбора часов, минут и секунд
yesno	Выводит простое сообщение с кнопками Да и Нет

Как показано в табл. 17.1, выбор различных графических элементов является довольно широким. Благодаря этому появляется возможность создавать сценарии, имеющие более профессиональный вид, не прилагая значительных усилий.

Чтобы указать конкретный графический элемент в командной строке, необходимо использовать формат с двойным тире:

```
dialog --widget parameters
```

где *widget* — имя графического элемента из числа тех, что показаны в табл. 17.1, а *parameters* — размер окна графического элемента и текст, которым должен сопровождаться графический элемент.

Каждый графический элемент `dialog` предоставляет вывод в двух формах:

- с использованием потока `STDERR`;
- с помощью статуса кода завершения.

Статус кода завершения команды `dialog` определяет, какая кнопка выбрана пользователем. Если выбрана кнопка **ОК** или **Да**, команда `dialog` возвращает статус выхода 0. Если выбрана кнопка **Отмена** или **Нет**, в команде `dialog` происходит возврат статуса выхода 1. Предусмотрена возможность использовать стандартную переменную `?` для определения того, какая кнопка была выбрана в графическом элементе `dialog`.

Если графический элемент возвращает какие-либо данные, например, относящиеся к выбору в меню, команда `dialog` отправляет данные в поток `STDERR`. Для перенаправления вывода `STDERR` в другой файл или дескриптор файла можно использовать стандартный способ, предусмотренный в командном интерпретаторе `bash`:

```
dialog --inputbox "Enter your age:" 10 20 2>age.txt
```

Эта команда перенаправляет текст, введенный в текстовом поле, в файл `age.txt`.

В следующих разделах рассматриваются некоторые примеры применения более сложных графических элементов пакета `dialog` в сценариях командного интерпретатора.

Графический элемент `msgbox`

Графический элемент `msgbox` представляет собой наиболее широко применяемый тип диалогового окна. Этот элемент отображает в окне простое текстовое сообщение и ожидает щелчка пользователем на кнопке **ОК**, после чего исчезает. Для использования графического элемента `msgbox` должен быть задан следующий формат:

```
dialog --msgbox text height width
```

В качестве параметра *text* может быть задана любая текстовая строка, которая должна быть выведена в окне. Команда `dialog` обеспечивает автоматический перенос текста на следующую строку, чтобы не происходил выход за пределы размеров созданного окна, для обозначения которых служат параметры *height* и *width*. Если появится необходимость поместить название в верхней части окна, то можно также воспользоваться параметром `--title`, с по-

мощью которого задается текст названия. Ниже приведен пример использования графического элемента `msgbox`.

```
$ dialog --title Testing --msgbox "This is a test" 10 20
```

После ввода этой команды в используемом сеансе эмулятора терминала на экране появится окно сообщения, показанное на рис. 17.2.

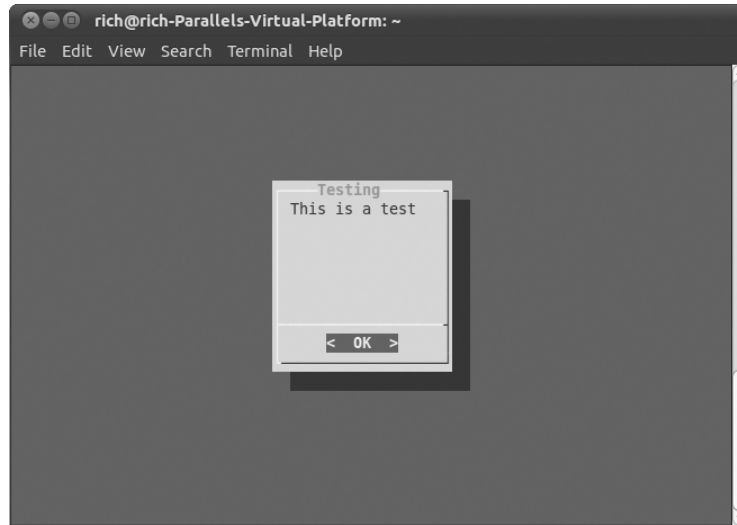


Рис. 17.2. Использование графического элемента `msgbox` в команде `dialog`

Если применяемый эмулятор терминала поддерживает мышь, то появляется дополнительная возможность щелкать на кнопке **ОК** для закрытия диалогового окна. Для моделирования щелчка можно также использовать команды клавиатуры; в данном случае достаточно нажать клавишу `<ВВОД>`.

Графический элемент `yesno`

Графический элемент `yesno` представляет собой более сложный вариант графического элемента `msgbox`, который позволяет пользователю ответить на вопрос, отображенный в окне, положительно или отрицательно. Этот элемент создает в нижней части окна две кнопки, на одной из которых имеется надпись "Да", а на другой — "Нет". Пользователь может переходить от одной кнопки к другой с помощью мыши, клавиши табуляции или клавиш со стрелками. Чтобы выбрать кнопку, пользователь может нажать клавишу пробела или `<ВВОД>`.

Ниже приведен пример использования графического элемента `yesno`.

```
$ dialog --title "Please answer" --yesno "Is this thing on?" 10 20
$ echo $?
1
$
```

В этом коде формируется графический элемент, который показан на рис. 17.3.

Статус выхода команды `dialog` устанавливается в зависимости от того, какую кнопку выбрал пользователь. Если выбрана кнопка **No** (Нет), статусом выхода является 1, а если выбрана кнопка **Yes** (Да), статус выхода принимает значение 0.

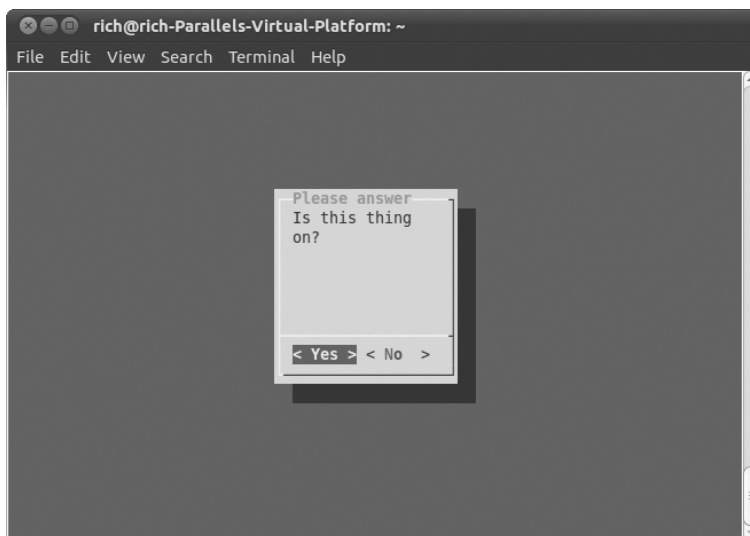


Рис. 17.3. Использование графического элемента *yesno* в команде *dialog*

Графический элемент *inputbox*

Графический элемент *inputbox* создает простую область с текстовым полем, в котором пользователь может ввести текстовую строку. Команда *dialog* отправляет значение введенной текстовой строки в поток *STDERR*. Чтобы получить ответ, это значение необходимо перенаправить. На рис. 17.4 показано, как выглядит графический элемент *inputbox*.

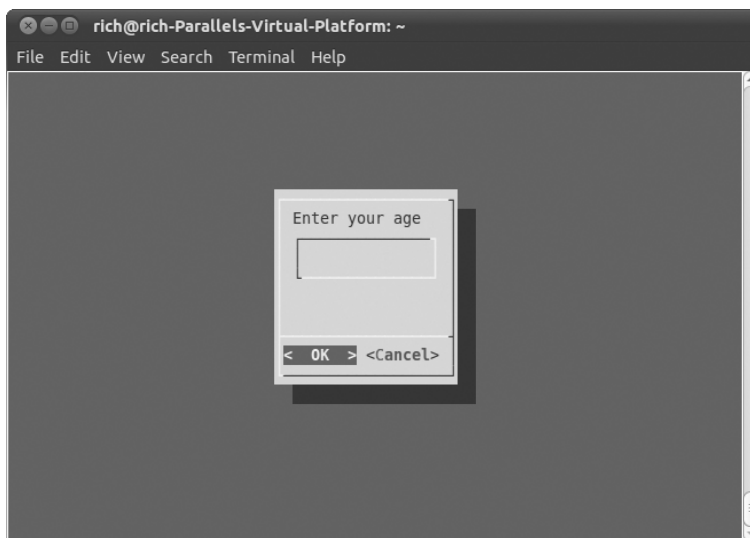


Рис. 17.4. Графический элемент *inputbox*

Как показано на рис. 17.4, графический элемент `inputbox` создает две кнопки — **ОК** и **Cancel Отмена**. Если выбрана кнопка **Отмена**, то статусом выхода команды является 1; в противном случае статус выхода будет равен 0:

```
$ dialog --inputbox "Enter your age:" 10 20 2>age.txt
$ echo $?
0
$ cat age.txt
12$
```

Следует отметить, что если для отображения содержимого текстового файла используется команда `cat`, то после вывода на экран соответствующего текстового значения не формируется символ обозначения конца строки. Это позволяет легко перенаправить содержимое файла в переменную в сценарии командного интерпретатора, чтобы извлечь строку, введенную пользователем.

Графический элемент `textbox`

Графический элемент `textbox` предоставляет превосходный способ отображения в окне большого объема информации. С его помощью создается прокручиваемое окно, которое содержит текст из файла, указанного в параметрах:

```
$ dialog --textbox /etc/passwd 15 45
```

Например, на рис. 17.5 показано, что в прокручиваемое текстовое окно выведено содержимое файла `/etc/passwd`.

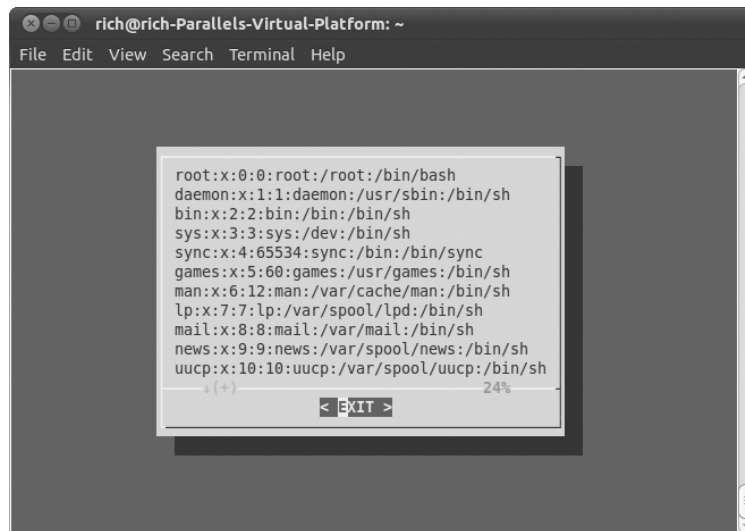


Рис. 17.5. Графический элемент `textbox`

Для прокрутки в текстовом файле влево и вправо, а также вверх и вниз можно использовать клавиши со стрелками. В нижней строке окна показано, какая процентная доля файла находится выше того места, в котором в данный момент происходит просмотр. Графический элемент `textbox` содержит единственную кнопку, **EXIT Выход**, на которой необходимо щелкнуть, чтобы закрыть окно.

Графический элемент menu

Графический элемент menu позволяет создать еще один вариант окна с текстовым меню, в дополнение к тому, которое было создано ранее в этой главе. Для использования данного графического элемента достаточно лишь указать дескриптор выбора и предоставить текст для каждого элемента:

```
$ dialog --menu "Sys Admin Menu" 20 30 10 1 "Display disk space"  
2 "Display users" 3 "Display memory usage" 4 "Exit" 2> test.txt
```

Первый параметр определяет название меню. Следующие два параметра задают высоту и ширину окна меню, а еще один параметр определяет количество элементов меню, отображаемых в окне одновременно. Если фактическое количество элементов меню превышает это значение, то предоставляется возможность выполнять прокрутку в окне меню с использованием клавиш со стрелками.

Вслед за этими параметрами необходимо добавить пары, определяющие элементы меню. Первым элементом пары является дескриптор, предназначенный для выбора элемента меню. Каждый дескриптор, определяющий элемент меню, должен быть уникальным, а его выбор должен быть сделан возможным путем нажатия соответствующей клавиши. Вторым элементом пары является текст, используемый для обозначения элемента меню. На рис. 17.6 показано меню, формируемое с помощью команды, приведенной в качестве примера.

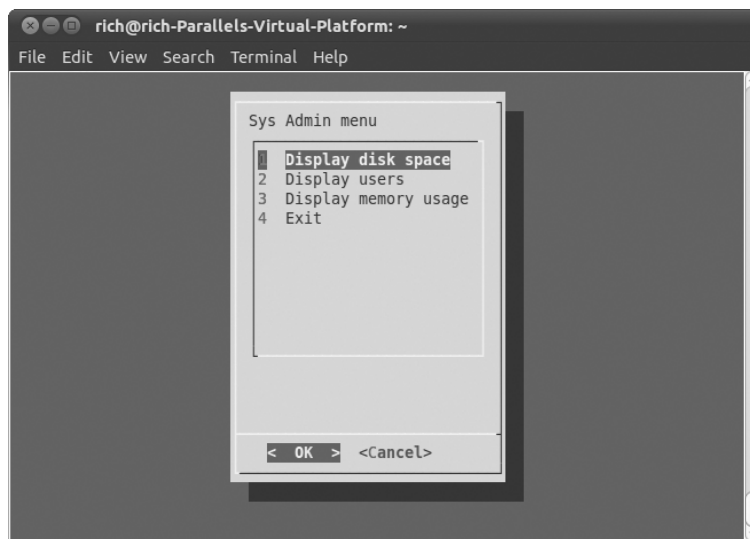


Рис. 17.6. Графический элемент menu с элементами меню

После выбора пользователем элемента меню путем нажатия клавиши, соответствующей дескриптору, происходит выделение этого элемента меню подсветкой, но не его выбор. Выбор не выполняется до щелчка на кнопке **OK** с использованием либо мыши, либо клавиши <ВВОД>. Команда dialog передает выбранный текст элемента меню в поток STDERR, который можно перенаправить по месту назначения.

Графический элемент fselect

Команда `dialog` позволяет воспользоваться еще несколькими привлекательными встроенными графическими элементами. При работе с именами файлов чрезвычайно удобным является графический элемент `fselect`. Графический элемент `fselect` позволяет избавиться от необходимости вводить имена используемых файлов, поскольку с его помощью можно перейти непосредственно к тому каталогу, где находится файл, и произвести его выбор (рис. 17.7).

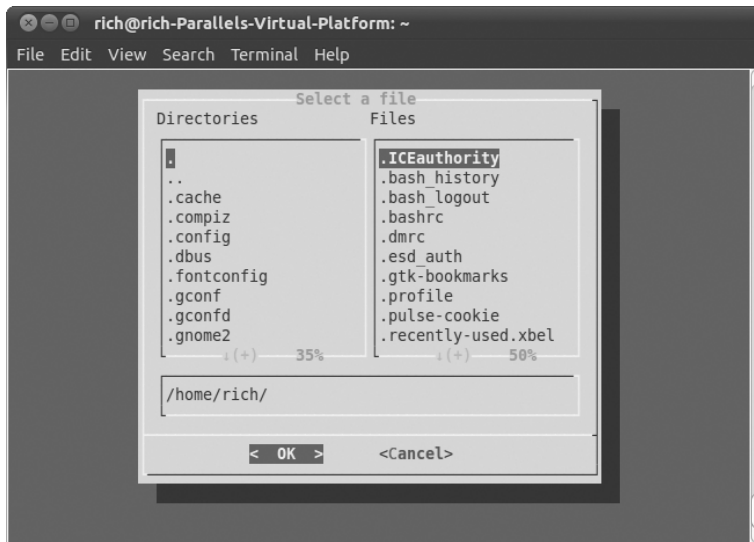


Рис. 17.7. Графический элемент `fselect`

Графический элемент `fselect` имеет примерно такой формат вызова:

```
$ dialog --title "Select a file" --fselect $HOME/ 10 50 2>file.txt
```

Первый параметр после опции `fselect` задает местоположение каталога, с которого начинается переход по каталогам в окне. Окно графического элемента `fselect` содержит листинг каталогов, находящийся слева, листинг файлов, который представлен справа, и простое текстовое поле с именем выбранного в настоящее время файла или каталога. Предусмотрена возможность вручную ввести имя файла в этом текстовом поле или использовать для выбора файла листинги каталогов и файлов.

Параметры команды `dialog`

Кроме определения стандартных графических элементов, с помощью команды `dialog` можно также настраивать много разных параметров. В частности, выше в данной главе уже был показан параметр `--title` в действии. Этот параметр позволяет задать текстовое обозначение для графического элемента, которое появляется в верхней части окна.

Предусмотрено также большое количество других опций, которые позволяют осуществлять полную настройку и внешнего вида, и поведения окон. В табл. 17.2 перечислены опции, предназначенные для работы с командой `dialog`.

Таблица 17.2. Опции команды dialog

Параметр	Описание
<code>--add-widget</code>	Переходить к следующему диалоговому окну, если не нажата кнопка Esc или Cancel (Отмена)
<code>--aspect ratio</code>	Задать соотношение размеров окна — ширины и высоты
<code>--backtitle title</code>	Указать название, которое должно отображаться на общем фоне в верхней части экрана
<code>--begin x y</code>	Указать начальное местоположение верхнего левого угла окна
<code>--cancel-label label</code>	Задать альтернативную надпись для кнопки EXIT (Отмена)
<code>--clear</code>	Очистить дисплей, оставив фоновое изображение в цвете, применяемом по умолчанию для диалоговых окон
<code>--colors</code>	Предусмотреть возможность использования цветовых кодов ANSI в тексте графического элемента dialog
<code>--cr-wrap</code>	Разрешить применение символов обозначения конца строки в тексте графического элемента dialog и принудительно выполнять перенос по строкам
<code>--create-rc file</code>	Вывести образец файла конфигурации в указанный файл
<code>--defaultno</code>	Задать Нет в качестве выбора по умолчанию в диалоговом окне с кнопками Yes (Да) и No (Нет)
<code>--default-item string</code>	Определить заданный по умолчанию элемент в диалоговом окне со списком выбора, формой или меню
<code>--exit-label label</code>	Задать альтернативную надпись для кнопки Exit (Выход)
<code>--extra-button</code>	Отобразить дополнительную кнопку между кнопками OK и Cancel (Отмена)
<code>--extra-label label</code>	Задать альтернативную надпись для дополнительной кнопки, вместо Extra
<code>--help</code>	Отобразить справочное сообщение команды dialog
<code>--help-button</code>	Отобразить кнопку Help (Справка) после кнопок OK и EXIT (Отмена)
<code>--help-label label</code>	Задать альтернативную надпись для кнопки Help (Справка)
<code>--help-status</code>	Вывести информацию о списке выбора, списке переключателей или форме после справочной информации, выбранной с помощью кнопки Help (Справка)
<code>--ignore</code>	Игнорировать опции, не распознанные командой dialog
<code>--input-fd fd</code>	Задать альтернативный дескриптор файла, отличный от STDIN
<code>--insecure</code>	Внести изменения в графический элемент password , чтобы в нем при вводе отображались звездочки
<code>--item-help</code>	Добавить столбец справки по дескриптору в нижней части экрана для каждого дескриптора в списке выбора, списке переключателей или меню
<code>--keep-window</code>	Не удалять с экрана выведенные ранее графические элементы
<code>--max-input size</code>	Указать максимальный размер строки ввода. По умолчанию применяется значение 2048
<code>--nocancel</code>	Подавить вывод кнопки EXIT (Отмена)
<code>--no-collapse</code>	Не выводить пробелы вместо знаков табуляции в тексте графического элемента dialog
<code>--no-kill</code>	Перевести диалоговое окно tailboxbg в фоновый режим и отключить сигнал SIGUP для процесса
<code>--no-label label</code>	Задать альтернативную надпись для кнопки No (Нет)

Параметр	Описание
<code>--no-shadow</code>	Не отображать тень вокруг диалоговых окон
<code>--ok-label label</code>	Задать альтернативную надпись для кнопки ОК
<code>--output-fd fd</code>	Задать альтернативный дескриптор файла вывода, отличный от <code>STDERR</code>
<code>--print-maxsize</code>	Вывести на устройство вывода значение максимального размера диалоговых окон
<code>--print-size</code>	Вывести на устройство вывода размер каждого диалогового окна
<code>--print-version</code>	Вывести на устройство вывода версию команды <code>dialog</code>
<code>--separate-output</code>	Каждый раз выводить результат использования графического элемента <code>checklist</code> в виде одной строки без кавычек
<code>--separator string</code>	Задать строку, которая разделяет результаты вывода для каждого графического элемента
<code>--separate-widget string</code>	Задать строку, которая разделяет результаты вывода для каждого графического элемента
<code>--shadow</code>	Показывать тень справа и снизу от каждого окна
<code>--single-quoted</code>	По мере необходимости в выводе данных списка выбора использовать одинарные кавычки
<code>--sleep sec</code>	Устанавливать паузу на заданное количество секунд после обработки диалогового окна
<code>--stderr</code>	Отправлять выходные данные в поток <code>STDERR</code> (это - поведение по умолчанию)
<code>--stdout</code>	Отправлять выходные данные в поток <code>STDOUT</code>
<code>--tab-correct</code>	Заменять знаки табуляции пробелами
<code>--tab-len n</code>	Задать количество пробелов, заменяющих знаки табуляции (по умолчанию применяется значение 8)
<code>--timeout sec</code>	Задать количество секунд, по истечении которых осуществляется выход с ошибкой при отсутствии ввода от пользователя
<code>--title title</code>	Указать название диалогового окна
<code>--trim</code>	Удалять ведущие пробелы и символы обозначения конца строки из текста графического элемента <code>dialog</code>
<code>--visit-items</code>	Изменять количество знаков табуляции в диалоговом окне с учетом наличия списка элементов
<code>--yes-label label</code>	Задать альтернативную надпись для кнопки Yes (Да)

Опция `--backtitle` предоставляет удобный способ создания общего названия для меню, применяемого во всем сценарии. Если название задано таким образом для каждого диалогового окна, то остается неизменным во всем приложении, поэтому результаты выполнения сценария выглядят более профессионально.

Как показывает табл. 17.2, предусмотрена возможность заменять надписи на всех кнопках в диалоговом окне. Благодаря этому появляется возможность создавать окна, соответствующие практически любым условиям применения.

Использование команды `dialog` в сценарии

Команда `dialog` является очень удобной для использования в сценариях. При этом необходимо лишь учитывать два требования:

- проверять статус выхода команды `dialog`, если применяется кнопки **Отмена** или **Нет**;
- перенаправлять поток `STDERR` для получения выходного значения.

Соблюдая эти два правила, можно легко создать интерактивный сценарий профессионального уровня с минимальными затратами времени. Ниже приведен пример использования графических элементов `dialog` для получения иным способом такого же меню системного администратора, которое было создано ранее в данной главе.

```
$ cat menu3
#!/bin/bash
# использование команды dialog для создания меню

temp='mktemp -t test.XXXXXX'
temp2='mktemp -t test2.XXXXXX'

function diskpace {
    df -k > $temp
    dialog --textbox $temp 20 60
}

function whoseon {
    who > $temp
    dialog --textbox $temp 20 50
}

function memusage {
    cat /proc/meminfo > $temp
    dialog --textbox $temp 20 50
}

while [ 1 ]
do
    dialog --menu "Sys Admin Menu" 20 30 10 1 "Display disk space" 2
    "Display users" 3 "Display memory usage" 0 "Exit" 2> $temp2
    if [ $? -eq 1 ]
    then
        break
    fi

    selection='cat $temp2'

    case $selection in
    1)
        diskpace ;;
    2)
        whoseon ;;
    3)
        memusage ;;
    0)
        break ;;
    *)
        dialog --msgbox "Sorry, invalid selection" 10 30
    esac
done
```

```
rm -f $temp 2> /dev/null
rm -f $temp2 2> /dev/null
$
```

В сценарии используется цикл `while` со значением константы `true` для создания бесконечного цикла, в котором отображается диалоговое окно меню. Это означает, что после выполнения каждой функции в сценарии снова воспроизводится меню.

Диалоговое окно `menu` включает кнопку **Отмена**, поэтому в сценарии проверяется статус выхода команды `dialog` на тот случай, если пользователь щелкнул на кнопке **Отмена** для завершения работы. Поскольку команды развертывания меню находятся в цикле `while`, для выхода из цикла и перехода к выполнению инструкций, которые следуют за инструкцией `while`, достаточно вызвать команду `break`.

В сценарии используется команда `mktemp` для создания двух временных файлов, предназначенных для хранения данных, которые применяются в команде `dialog`. Первый файл, `$temp`, предназначен для сохранения вывода команд `df` и `meminfo` в целях последующего отображения этого вывода в диалоговом окне `textbox` (рис. 17.8). Второй временный файл, `$temp2`, служит для сохранения значения выбора, сделанного в диалоговом окне главного меню.

Наконец-то наш сценарий начинает напоминать настоящее приложение, которое не стыдно показать другим людям!

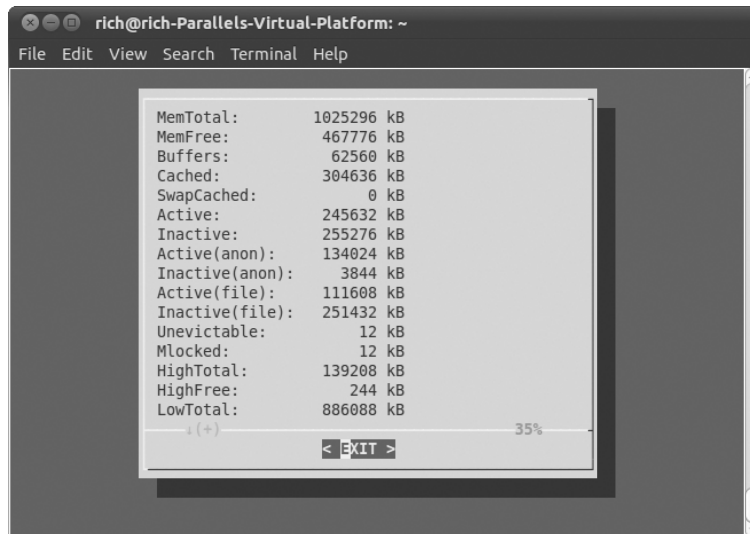


Рис. 17.8. Вывод команды `meminfo` отображается с использованием опции диалогового окна `textbox`

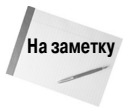
Применение графического режима

Дополнительную привлекательность интерактивным сценариям может придать более широкое применение графических средств, для чего необходимо пройти на шаг дальше. В вариантах среды рабочего стола KDE и GNOME (см. главу 1) замысел команды `dialog` был в еще большей степени усовершенствован и включены команды, с помощью которых формируются графические элементы X Window для соответствующей среды.

В настоящем разделе описаны пакеты `kdiallog` и `zenity`, с помощью которых создаются графические элементы окна для рабочих столов KDE и GNOME соответственно.

Среда KDE

Графическая среда KDE включает по умолчанию пакет `kdiallog`. В пакете `kdiallog` используется команда `kdiallog` для формирования в приложениях для рабочего стола KDE стандартных окон, аналогичных графическим элементам в стиле команды `dialog`. Но эти окна имеют такую особенность, что не выглядят как инородное тело, а безукоризненно сочетаются с остальными окнами приложения KDE! Это позволяет создавать пользовательские интерфейсы, не уступающие по качеству интерфейсам приложений Windows, непосредственно в сценариях командного интерпретатора!



Однако следует помнить, что применение в дистрибутиве Linux рабочего стола KDE не означает, что пакет `kdiallog` обязательно устанавливается по умолчанию. Может потребоваться установить этот пакет вручную, получив его из репозитория конкретного дистрибутива.

Графические элементы `kdiallog`

По аналогии с командой `dialog`, команда `kdiallog` предусматривает задание опций командной строки для указания на то, какой графический элемент окна должен использоваться. Команда `kdiallog` имеет следующий формат:

```
kdiallog display-options window-options arguments
```

Опции *window-options* позволяют определять предназначенный для использования графический элемент окна. Опции, предусмотренные в этой команде, приведены в табл. 17.3.

Таблица 17.3. Опции окна `kdiallog`

Параметр	Описание
<code>--checklist title [tag item status]</code>	Компонент меню в виде списка выбора, в котором параметр <i>status</i> указывает, отмечен элемент списка или нет
<code>--error text</code>	Поле сообщения об ошибке
<code>--inputbox text [init]</code>	Текстовое поле ввода. Предусмотрена возможность задать текстовую строку, отображаемую в поле <i>text</i> по умолчанию, с помощью значения <i>init</i>
<code>--menu title [tag item]</code>	Заголовок окна меню в виде списка выбора и список элементов, обозначенных дескрипторами
<code>--msgbox text</code>	Простое окно сообщения с указанным текстом
<code>--password text</code>	Текстовое поле ввода пароля, которое скрывает ввод данных пользователем
<code>--radiolist title [tag item status]</code>	Меню в виде списка переключателей, где параметр <i>status</i> указывает, выбран элемент или нет
<code>--separate-output</code>	Команда возвращает элементы меню со списками выбора и списками переключателей в отдельных строках
<code>--sorry text</code>	Окно сообщения "sorry" (сожалею)
<code>--textbox file [width] [height]</code>	Текстовое поле, отображающее содержимое файла <i>file</i> , для которого дополнительно можно указать ширину и высоту, <i>width</i> и <i>height</i>

Параметр	Описание
<code>--title title</code>	Задаёт название для области TitleBar диалогового окна
<code>--warningyesno text</code>	Окно с предупреждающим сообщением, в котором предусмотрены кнопки Yes (Да) и (Нет)
<code>--warningcontinuecancel text</code>	Окно с предупреждающим сообщением, в котором предусмотрены кнопки Continue (Продолжить) и Cancel (Отмена)
<code>--warningyesnocancel text</code>	Окно с предупреждающим сообщением, в котором предусмотрены кнопки Yes (Да), No (Нет), Cancel (Отмена)
<code>--yesno text</code>	Поле с вопросом, в котором предусмотрены кнопки Yes (Да) и No (Нет)
<code>--yesnocancel text</code>	Поле с вопросом, в котором предусмотрены кнопки Yes (Да), No (Нет), Cancel (Отмена)

Как показывает табл. 17.3, представлены все стандартные типы диалоговых окон. Но если используется графический элемент окна `kdiallog`, то отображается на рабочем столе KDE как отдельное окно, а не как окно в сеансе эмулятора терминала!

Графические элементы `checklist` и `radiolist` позволяют определять отдельные элементы в списках выбора и списках переключателей и указывать, являются ли они выбранными по умолчанию:

```
$kdiallog --checklist "Items I need" 1 "Toothbrush" on 2 "Toothpaste"
off 3 "Hair brush" on 4 "Deodorant" off 5 "Slippers" off
```



Полученное в итоге окно со списком выбора показано на рис. 17.9.

Элементы, обозначенные как отмеченные ("on"), выделяются в списке выбора подсветкой. Чтобы выбрать или отменить выбор элемента в списке выбора, достаточно щелкнуть на нем. После щелчка на кнопке ОК команда `kdiallog` отправляет значения дескрипторов в поток STDOUT:

```
"1" "3" "5"
$
```

Рис. 17.9. Диалоговое окно `kdiallog` со списком выбора

После нажатия клавиши <ВВОД> отображается поле `kdiallog` с выбранными элементами. После щелчка на кнопке ОК или Отмена команда `kdiallog` передает содержимое каждого дескриптора в виде строкового значения в поток STDOUT (это — значения "1", "3" и "5", которые показаны в выводе). В сценарии необходимо предусмотреть синтаксический анализ результирующих значений и согласование их с первоначальными значениями.

Использование команды `kdiallog`

Графические элементы окна `kdiallog` можно использовать в сценариях командного интерпретатора по аналогии с тем, как используются графические элементы `dialog`. Существенным различием является то, что графические элементы окна `kdiallog` передают выходные значения в поток STDOUT, а не в поток STDERR.

Ниже приведен сценарий, который позволяет преобразовать приведенное выше меню системного администратора в приложение KDE.

```
$ cat menu4
#!/bin/bash
# использование команды kdialog для создания меню

temp='mktemp -t temp.XXXXXX'
temp2='mktemp -t temp2.XXXXXX'

function diskpace {
    df -k > $temp
    kdialog --textbox $temp 1000 10
}

function whoseon {
    who > $temp
    kdialog --textbox $temp 500 10
}

function memusage {
    cat /proc/meminfo > $temp
    kdialog --textbox $temp 300 500
}

while [ 1 ]
do
kdialog --menu "Sys Admin Menu" "1" "Display diskpace" "2" "Display
users" "3" "Display memory usage" "0" "Exit" > $temp2
if [ $? -eq 1 ]
then
    break
fi

selection='cat $temp2'

case $selection in
1)
    diskpace ;;
2)
    whoseon ;;
3)
    memusage ;;
0)
    break ;;
*)
    kdialog --msgbox "Sorry, invalid selection"
esac
done
$
```

Очевидно, что этот сценарий не потребовал значительных изменений после перехода от использования команды `dialog` к команде `kdialog`. Сформированное в конечном итоге главное меню показано на рис. 17.10.



Очевидно, что теперь простые сценарии командного интерпретатора позволяют получить такие же результаты, как настоящее приложение KDE! Теперь не существует пределов для творческих возможностей, открывающихся при использовании интерактивных сценариев!

Рис. 17.10. Результаты применения сценария меню системного администратора, в котором используется команда `kdialog`

Среда GNOME

Графическая среда GNOME поддерживает следующие два популярных пакета, позволяющих формировать стандартные окна:

- `gdialog`;
- `zenity`.

По сравнению с `gdialog` пакет `zenity` является намного более распространенным и предусмотрен в большинстве дистрибутивов Linux с рабочим столом GNOME (в частности, он устанавливается по умолчанию в дистрибутивах Ubuntu и Fedora). В настоящем разделе описаны средства `zenity` и показано, как их использовать в сценариях командного интерпретатора.

Графические элементы zenity

Как и следовало ожидать, команда `zenity` позволяет создавать различные графические элементы окон с использованием опций командной строки. В табл. 17.4 перечислены различные графические элементы, которые могут быть сформированы с помощью команды `zenity`.

Таблица 17.4. Графические элементы окон, формируемые с помощью команды `zenity`

Параметр	Описание
<code>--calendar</code>	Отобразить полный календарь на месяц
<code>--entry</code>	Показать диалоговое окно ввода текста
<code>--error</code>	Показать диалоговое окно сообщения об ошибке
<code>--file-selection</code>	Отобразить диалоговое окно с полным именем пути и именем файла
<code>--info</code>	Отобразить информационное диалоговое окно
<code>--list</code>	Вывести диалоговое окно со списком выбора или списком переключателей
<code>--notification</code>	Отобразить значок уведомления
<code>--progress</code>	Вывести диалоговое окно с индикатором хода работы
<code>--question</code>	Вывести диалоговое окно с вопросом и кнопками Yes (Да) и No (Нет)
<code>--scale</code>	Отобразить диалоговое окно определения масштаба
<code>--text-info</code>	Отобразить текстовое поле, содержащее текст
<code>--warning</code>	Отобразить диалоговое окно с предупреждением

Программы для командной строки с командой `zenity` действуют немного иначе по сравнению с программами на основе `kdialog` и `dialog`. В частности, многие типы графических элементов определяются с использованием дополнительных опций в командной строке, а не задаются путем включения их параметров опций.

Некоторые диалоговые окна, создаваемые с помощью команды `zenity`, являются действительно привлекательными и совершенными. Например, опция `calendar` позволяет сформировать календарь на полный месяц, как показано на рис. 17.11.

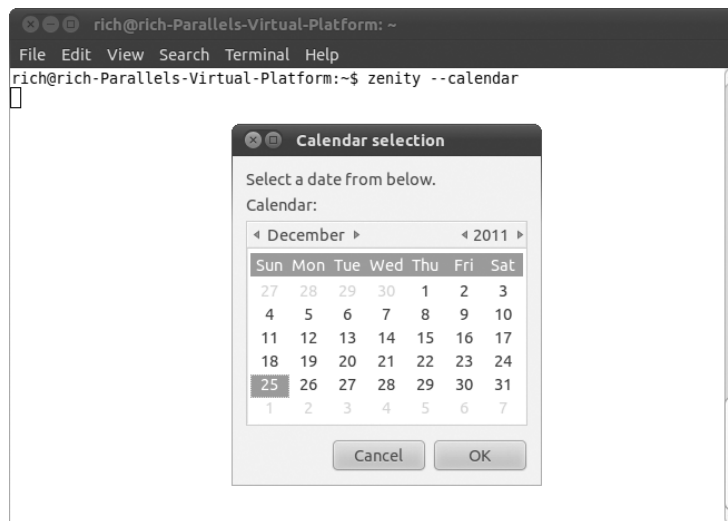


Рис. 17.11. Диалоговое окно `zenity` с календарем

После выбора в календаре конкретной даты команда `zenity` возвращает полученное значение в поток `STDOUT` точно так же, как команда `kdialog`:

```
$ zenity --calendar
12/25/2011
$
```

Еще одним довольно привлекательным окном, формируемым с помощью `zenity`, является окно, для создания которого служит опция выбора файла (рис. 17.12).

Это диалоговое окно можно использовать для перехода к местоположению любого каталога в системе (при наличии прав доступа для просмотра каталога) и выбора файла. После выбора файла команда `zenity` возвращает полное имя файла и имя пути:

```
$ zenity --file-selection
/home/ubuntu/menu5
$
```

Таким образом, в распоряжении разработчика имеются возможности, открывающие безграничные перспективы создания сценариев командного интерпретатора!

Использование команды `zenity` в сценариях

Как и можно было предположить, команда `zenity` очень хорошо проявляет себя в сценариях командного интерпретатора. Но, к сожалению, разработчики пакета `zenity` отказались от со-

блюдения соглашений по выбору опций, используемых в пакетах `dialog` и `kdiallog`, поэтому при преобразовании любых существующих интерактивных сценариев в сценарии для команды `zenity` могут возникать сложности.

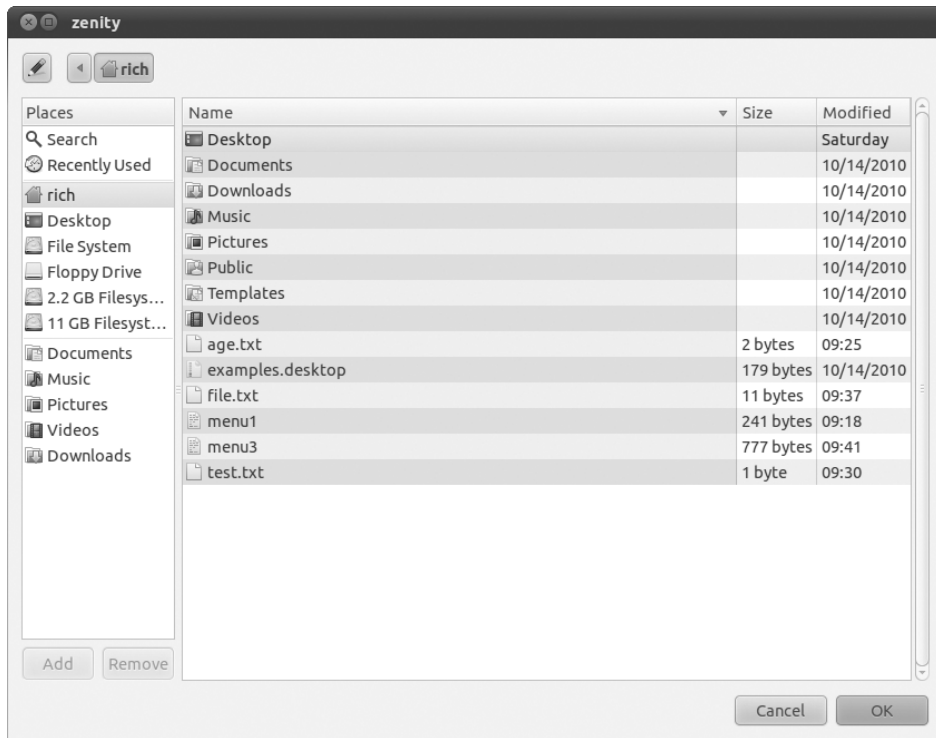


Рис. 17.12. Диалоговое окно выбора файла команды `zenity`

В частности, авторы настоящей книги, занимаясь преобразованием меню системного администратора с переходом от `kdiallog` к `zenity`, обнаружили, что с определениями графических элементов приходится выполнять довольно много манипуляций:

```
$cat menu5
#!/bin/bash
# использование команды zenity для создания меню

temp=`mktemp -t temp.XXXXXX`
temp2=`mktemp -t temp2.XXXXXX`

function diskpace {
    df -k > $temp
    zenity --text-info --title "Disk space" --filename=$temp
    --width 750 --height 10
}

function whoseon {
    who > $temp
    zenity --text-info --title "Logged in users" --filename=$temp
```



```

--width 500 --height 10
}

function memusage {
    cat /proc/meminfo > $temp
    zenity --text-info --title "Memory usage" --filename=$temp
--width 300 --height 500
}

while [ 1 ]
do
    zenity --list --radiolist --title "Sys Admin Menu" --column "Select"
--column "Menu Item" FALSE "Display disk space" FALSE "Display users"
FALSE "Display memory usage" FALSE "Exit" > $temp2
    if [ $? -eq 1 ]
    then
        break
    fi

    selection=`cat $temp2`
    case $selection in
        "Display disk space")
            disk space ;;
        "Display users")
            whoseon ;;
        "Display memory usage")
            memusage ;;
        Exit)
            break ;;
        *)
            zenity --info "Sorry, invalid selection"
    esac
done
$

```

Команда `zenity` не поддерживает диалоговое окно меню, поэтому для главного меню пришлось использовать окно меню, действующее по принципу списка переключателей, как показано на рис. 17.13.

В списке переключателей применяются два столбца, для каждого из которых предусмотрен заголовок столбца. Первый столбец содержит переключатели, предназначенные для выбора. Во втором столбце приведен текст элементов. Список переключателей отличается также тем, что в нем не используются дескрипторы для элементов. После выбора элемента в поток `STDOUT` передается полный текст элемента. В связи с этим при использовании команды `case` приходится выполнять некоторую дополнительную работу. В частности, в опциях `case` должна предусматриваться проверка полного текста каждого элемента. Если в тексте имеются пробелы, то необходимо заключить текст в кавычки.

С помощью пакета `zenity` можно добиться того, что предложения для рабочего стола GNOME, сформированные с помощью интерактивных сценариев командного интерпретатора, будут выглядеть как приложения Windows.

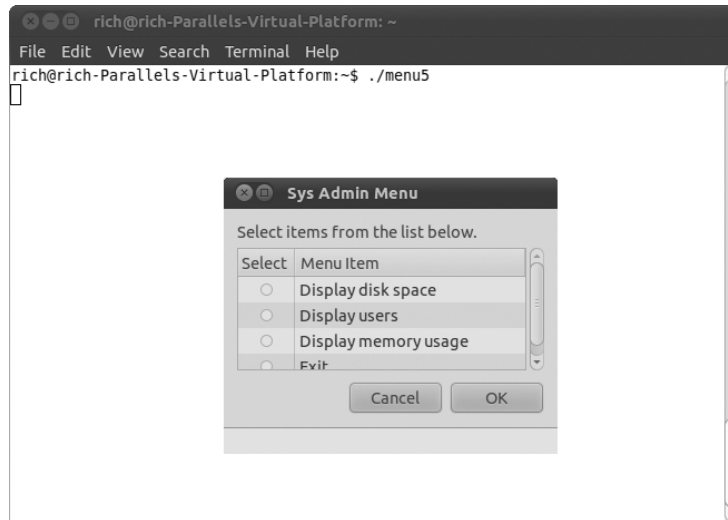


Рис. 17.13. Меню системного администратора, сформированное с использованием команды *zenity*

Резюме

Сложилось мнение, что приложения, создаваемые с помощью интерактивных сценариев командного интерпретатора, неизбежно остаются примитивными и неинтересными. Это мнение можно опровергнуть, используя некоторые разнообразные средства и инструменты, предусмотренные в большинстве систем Linux. Прежде всего, можно создавать системы меню для интерактивных сценариев с помощью команды *case* и функций сценариев командного интерпретатора.

Команда *case* позволяет формировать меню с использованием стандартной команды *echo* и читать ответ от пользователя с помощью команды *read*. Затем команда *case* выбирает соответствующую функцию сценария командного интерпретатора с учетом введенного значения.

Программа *dialog* предоставляет несколько предварительно сформированных текстовых графических элементов для создания графических объектов, подобных тем, что применяются в системе Windows, в алфавитно-цифровом эмуляторе терминала. Предусмотрена возможность создавать диалоговые окна для отображения текста, вводить текст, выбирать файлы и указывать даты с использованием диалоговой программы. Это позволяет еще больше повысить привлекательность сценария командного интерпретатора.

Если же сценарии командного интерпретатора применяются в графической среде X Window, то появляется возможность использовать в интерактивных сценариях еще больше средств. Для рабочего стола KDE предусмотрена программа *kdiallog*, предоставляющая простые команды создания графических элементов окон для всех основных функций окон. Для рабочего стола GNOME предусмотрены программы *gdiallog* и *zenity*. Каждая из этих программ предоставляет возможность воспользоваться графическими элементами окон, которые безукоризненно встраиваются в среду рабочего стола GNOME, поэтому создает приложение, не уступающее по привлекательности типичному приложению Windows.

В следующей главе мы перейдем к рассмотрению темы редактирования и манипулирования файлами с текстовыми данными. Одной из важнейших областей применения сценариев командного интерпретатора часто становится синтаксический анализ и отображение данных, содержащихся в текстовых файлах, таких как журналы и файлы с сообщениями об ошибках. В среде Linux для работы с текстовыми данными в сценариях командного интерпретатора предусмотрены два чрезвычайно полезных инструмента, `sed` и `gawk`. Эти инструменты и основные сведения об их использовании рассматриваются в следующей главе.

ГЛАВА

18

В этой главе...

Работа с текстом

Основные сведения
о редакторе sed

Резюме

Общие сведения о редакторах sed и gawk

Безусловно, одной из функций, для выполнения которых используются сценарии командного интерпретатора, является работа с текстовыми файлами. Сценарии командного интерпретатора могут помочь не только при исследовании файлов журналов, чтении файлов конфигурации и обработке элементов данных, но и при автоматизации трудоемких задач манипулирования данными любого типа, содержащимися в текстовых файлах. Тем не менее попытки манипулировать содержимым текстовых файлов с использованием только команд сценария командного интерпретатора могут потребовать слишком много трудозатрат. Тем пользователям, которым приходится выполнять любые виды манипулирования данными в сценариях командного интерпретатора, необходимо ознакомиться с инструментами `sed` и `gawk`, предусмотренными в Linux. Эти инструменты позволяют существенно упростить любые задачи обработки данных, которые только потребуются.

Работа с текстом

В главе 9 было показано, как редактировать текстовые файлы с использованием различных программ редакторов, предусмотренных в среде Linux. Эти редакторы позволяют легко манипулировать текстом, содержащемся в текстовом файле, с использованием простых команд или щелчков кнопкой мыши.

Но иногда возникает необходимость в проведении манипуляций с текстом в текстовом файле динамически, без

привлечения полнофункционального интерактивного текстового редактора. В этих ситуациях было бы удобно иметь простой редактор с интерфейсом командной строки, позволяющий легко форматировать, вставлять, изменять или удалять текстовые элементы автоматически.

В системе Linux предусмотрены два инструмента, которые находят очень широкое распространение и предназначены для решения именно этих задач. В настоящем разделе рассматриваются эти два наиболее широко применяемых редактора с интерфейсом командной строки, которые распространены в мире Linux, *sed* и *gawk*.

Редактор *sed*

Редактор *sed* относится к категории *поточковых редакторов*, в отличие от обычного интерактивного текстового редактора. В интерактивном текстовом редакторе, таком как *vim*, применяются клавиатурные команды, которые позволяют вставлять, удалять или заменять текст в данных в интерактивном режиме. С другой стороны, поточковый редактор вносит изменения в поток данных на основе ряда правил, установленных заранее, до начала обработки данных редактором.

Редактор *sed* позволяет манипулировать данными в потоке данных с учетом команд, заданных в командной строке или сохраненных в текстовом файле с командами. Этот редактор считывает с устройства ввода одновременно по одной строке данных, согласовывает содержимое строки с данными, представленными в командах редактора, вносит в данные изменения в режиме потоковой обработки согласно указаниям команд, а затем выводит вновь сформированные данные в поток *STDOUT*. После того как поточковый редактор сопоставляет все команды со строкой данных, происходит переход к следующей строке данных и процесс повторяется. По завершении обработки потоковым редактором всех строк данных в потоке происходит выход из программы редактора.

Поскольку команды применяются к данным последовательно, строка за строкой, редактор *sed* должен осуществлять только один проход через поток данных для внесения всех изменений. Благодаря этому редактор *sed* выполняет свою работу намного быстрее по сравнению с интерактивным редактором и позволяет быстро, динамически вносить изменения в данные в файле данных.

Команда *sed* имеет следующий формат:

```
sed options script file
```

Опции *options*, для которых предусмотрены параметры, позволяют настраивать поведение команды *sed*. К ним относятся опции, показанные в табл. 18.1.

Таблица 18.1. Опции команды *sed*

Параметр	Описание
<code>-e script</code>	Добавить команды, указанные в сценарии <i>script</i> , к командам, выполняемым при обработке входных данных
<code>-f file</code>	Добавить команды, приведенные в файле <i>file</i> , к командам, выполняемым при обработке входных данных
<code>-n</code>	Не вырабатывать выходные данные после выполнения каждой команды, а ожидать команду <i>print</i>

Параметр *script* указывает отдельную команду, применяемую к потоковым данным. Если требуется больше одной команды, то следует воспользоваться либо опцией `-e` для задания команд в командной строке, либо опцией `-f` для определения их в отдельном файле. Для манипулирования данными предусмотрены многочисленные команды. Некоторые из основных

команд, используемых в редакторе `sed`, рассматриваются ниже в этой главе, а другие, более сложные команды будут описаны в главе 20.

Определение одной команды редактора в командной строке

По умолчанию редактор `sed` применяет заданную при его вызове команду к входному потоку `STDIN`. Это позволяет передавать данные по каналу для обработки непосредственно в редактор `sed`. Ниже приведен краткий пример, который показывает, как выполняется подобная операция.

```
$ echo "This is a test" | sed 's/test/big test/'
This is a big test
$
```

В этом примере используется команда `s` редактора `sed`. Команда `s` обеспечивает подстановку второй текстовой строки вместо первой текстовой строки, заданной с помощью шаблона между символами косой черты. В этом примере происходит подстановка слов `big test` вместо слова `test`.

После запуска этого примера на выполнение результаты должны отобразиться почти мгновенно. В этом быстродействии состоит одно из преимуществ редактора `sed`. Он позволяет произвести несколько операций внесения изменений в данные примерно за такое же время, в которое некоторые из интерактивных редакторов могут лишь осуществить свой запуск.

Безусловно, в этом простом примере изменена только одна строка данных. Но столь же быстро результаты формируются и при изменении крупных файлов с данными:

```
$ cat data1
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
$
$ sed 's/dog/cat/' data1
The quick brown fox jumps over the lazy cat.
The quick brown fox jumps over the lazy cat.
The quick brown fox jumps over the lazy cat.
The quick brown fox jumps over the lazy cat.
$
```

Команда `sed` выполняет свою работу и возвращает данные почти мгновенно. Результаты отображаются после обработки каждой строки данных. Таким образом, просмотр полученных данных может начаться еще до того, как редактор `sed` завершит обработку всего файла.

Важно учитывать, что редактор `sed` не изменяет данные в самом текстовом файле. Он лишь передает измененный текст в поток `STDOUT`. Просмотр исходного текстового файла позволяет убедиться в том, что в нем по-прежнему содержатся те же данные:

```
$ cat data1
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
$
```

Использование нескольких команд редактора в командной строке

Чтобы вызвать на выполнение больше чем одну команду из командной строки `sed`, достаточно воспользоваться опцией `-e`:

```
$ sed -e 's/brown/green/; s/dog/cat/' data1
The quick green fox jumps over the lazy cat.
The quick green fox jumps over the lazy cat.
The quick green fox jumps over the lazy cat.
The quick green fox jumps over the lazy cat.
$
```

Обе команды применяются к каждой строке данных в файле. Команды должны быть разделены точками с запятой, а между концом команды и точкой с запятой не должно быть никаких пробелов.

Вместо использования точек с запятой для разделения команды можно вызвать вторичное приглашение к вводу информации в командном интерпретаторе `bash`. Для этого достаточно ввести первую, открывающую одинарную кавычку, чтобы открыть сценарий, после чего командный интерпретатор `bash` продолжит запрашивать ввод следующих команд, пока не будет введена закрывающая кавычка:

```
$ sed -e '
> s/brown/green/
> s/fox/elephant/
> s/dog/cat/' data1
The quick green elephant jumps over the lazy cat.
The quick green elephant jumps over the lazy cat.
The quick green elephant jumps over the lazy cat.
The quick green elephant jumps over the lazy cat.
$
```

Нельзя забывать, что команда должна быть завершена на той же строке, в которой имеется закрывающая одинарная кавычка. После того как командный интерпретатор `bash` обнаруживает закрывающую кавычку, начинается выполнение самой команды. После запуска команда `sed` применяет каждую заданную команду к каждой строке данных в текстовом файле.

Чтение команд редактора из файла

Наконец, если требуется обеспечить применение к данным большого количества команд `sed`, проще сохранить их в отдельном файле. Для указания файла в команде `sed` применяется опция `-f`:

```
$ cat script1
s/brown/green/
s/fox/elephant/
s/dog/cat/
$
$ sed -f script1 data1
The quick green elephant jumps over the lazy cat.
The quick green elephant jumps over the lazy cat.
The quick green elephant jumps over the lazy cat.
The quick green elephant jumps over the lazy cat.
$
```

В таком случае не требуется помещать точку с запятой после каждой команды. Редактор `sed` исходит из того, что в каждой строке файла с командами содержится отдельная команда. Как и при вводе команд в командной строке, редактор `sed` считывает команды из указанного файла и применяет их к каждой строке в файле данных.

Некоторые другие команды редактора `sed`, которые могут потребоваться при манипулировании данными, приведены в разделе “Основные сведения о редакторе `sed`”. Но вначале кратко рассмотрим еще один редактор данных Linux.

Программа `gawk`

Несмотря на то что редактор `sed` представляет собой весьма удобный инструмент для оперативного внесения изменений в текстовые файлы, он имеет свои ограничения. Часто для манипулирования данными в файле требуется более развитый инструмент, такой, который способен предоставить среду, подобную среде обычного языка программирования, позволяющую модифицировать и реорганизовывать данные в файле. Именно в таких обстоятельствах может применяться редактор `gawk`.

Программа `gawk` представляет собой версию GNU исходной программы `awk`, предусмотренной в Unix. Программа `gawk` позволяет поднять на новый уровень сложность выполняемых задач по сравнению с редактором `sed`, поскольку в ней вместо простых команд редактора применяются целые сценарии на отдельном языке программирования. Язык программирования `gawk` позволяет выполнять следующее:

- определять переменные для хранения данных;
- использовать арифметические и строковые операции для обработки данных;
- задавать структурированные программные конструкции наподобие инструкций `if-then` и циклов для обеспечения обработки данных с помощью средств проверки условий;
- создавать отформатированные отчеты, извлекая элементы данных из файла данных, а затем комбинируя эти данные по-другому, с изменением последовательности расположения или формата.

Возможности формирования отчетов программы `gawk` часто используются для извлечения элементов данных из больших текстовых файлов и представления данных в виде отчета, удобного для чтения. Идеальным примером решения этой задачи может служить форматирование файлов журналов. Задача поиска причин нарушения в работе путем изучения сообщений об ошибках в файле журнала может оказаться сложной. Программа `gawk` позволяет выбрать из файла журнала с помощью фильтра только те элементы данных, которые представляют интерес, а затем отформатировать их таким образом, чтобы чтение важных данных стало проще.

Формат команды `gawk`

Программа `gawk` имеет следующий основной формат:

```
gawk options program file
```

В табл. 18.2 приведены опции *options*, поддерживаемые программой `gawk`.

Опции командной строки предоставляют простой способ настройки средств, применяемых в программе `gawk`. Эти средства будут описаны более подробно в разделе, в котором рассматривается программа `gawk`.

Основное преимущество программы `gawk` заключается в том, что она позволяет использовать программные сценарии. Предусмотрена возможность написания сценариев для чтения данных в текстовой строке, а затем отображения данных и манипулирования ими в целях создания выходного отчета любого типа.

Таблица 18.2. Опции gawk

Параметр	Описание
-F <i>fs</i>	Задать для файла разделитель, который служит для разграничения полей данных в строке
-f <i>file</i>	Указать имя файла, из которого должна быть считана программа
-v <i>var=value</i>	Определить переменную <i>var</i> и значение по умолчанию <i>value</i> для использования в программе gawk
-mf <i>N</i>	Задать максимальное количество полей, подлежащих обработке в файле данных
-mr <i>N</i>	Задать максимальный размер записи в файле данных
-W <i>keyword</i>	Указать режим совместимости или уровень предупреждения для программы gawk

Чтение программного сценария из командной строки

Сценарий программы gawk заключен в открывающие и закрывающие фигурные скобки. Между этими двумя фигурными скобками должны быть помещены команды сценария. Поскольку предполагается, что в командной строке gawk весь сценарий представлен в виде одной текстовой строки, необходимо также заключить сценарий в одинарные кавычки. Ниже приведен пример простого сценария программы gawk, заданного в командной строке.

```
$ gawk '{print "Hello John!"}'
```

В этом программном сценарии определена единственная команда — `print`. Команда `print` выполняет действие, соответствующее ее названию: выводит текст в поток `STDOUT`. При попытке непосредственно выполнить эту команду можно испытать определенное разочарование, поскольку сразу же после ее ввода ничего не происходит. Дело в том, что в этой командной строке не задано имя файла, поэтому программа gawk приступает к приему данных из потока `STDIN`. После запуска программы на выполнение просто происходит переход в режим ожидания поступления текста через `STDIN`.

После ввода строки текста и нажатия клавиши <ВВОД> программа gawk обрабатывает текст с помощью программного сценария:

```
$ gawk '{print "Hello John!"}'
This is a test
Hello John!
hello
Hello John!
This is another test
Hello John!

$
```

Полностью аналогично редактору `sed`, программа gawk выполняет программный сценарий применительно к каждой строке текста, имеющейся в потоке данных. Данный программный сценарий настроен на отображение постоянно заданной текстовой строки, поэтому независимо от текста, введенного в поток данных, отображаются одни и те же текстовые выходные данные.

Для завершения работы программы gawk необходимо подать сигнал об окончании потока данных. В командном интерпретаторе `bash` предусмотрена комбинация клавиш для формирования символа конца файла (End-of-File — EOF). Символ, представляющий собой признак конца файла EOF, в командном интерпретаторе `bash` формируется с помощью комбинации клавиш <Ctrl+D>. После ввода этой комбинации клавиш работа программы gawk оканчивается и происходит возврат к приглашению интерфейса командной строки.

Использование переменных с обозначением полей данных

Одна из наиболее важных возможностей программы `gawk` состоит в том, что она позволяет манипулировать данными в текстовом файле. Для этого в программе предусмотрено автоматическое назначение переменной каждому элементу данных в строке. По умолчанию `gawk` назначает следующие переменные различным полям данных, обнаруженным в строке текста:

- `$0` представляет всю строку текста;
- `$1` представляет первое поле данных в строке текста;
- `$2` представляет второе поле данных в строке текста;
- `$n` представляет *n*-е поле данных в строке текста.

Для разграничения полей данных в текстовой строке применяется *символ разделения полей*. После считывания программой `gawk` строки текста происходит разбиение полученного текста на поля данных с использованием заданного символа разделения полей. По умолчанию в программе `gawk` в качестве символов разделения полей рассматриваются все пробельные символы (такие как знаки табуляции и пробелы).

Ниже приведен пример программы `gawk`, которая считывает текстовый файл и отображает только первое значение поля данных:

```
$ cat data3
One line of test text.
Two lines of test text.
Three lines of test text.
$
$ gawk '{print $1}' data3
One
Two
Three
$
```

В этой программе используется переменная поля `$1` для отображения только первого поля данных из каждой строки текста.

Если считывается файл, в котором используется другой символ разделения полей, таковой можно указать с помощью опции `-F`:

```
$ gawk -F: '{print $1}' /etc/passwd
root
daemon
bin
sys
sync
games
man
lp
mail
...
```

Эта короткая программа отображает первое поле данных в файле паролей системы. В файле `/etc/passwd` для разделения полей данных служит двоеточие, поэтому, чтобы отделить друг от друга различные элементы данных, необходимо указать двоеточие в качестве символа разделения полей в опциях `gawk`.

Использование нескольких команд в программном сценарии

Язык программирования был бы слишком маловыразительным, если с его помощью можно было задавать в программе только одну команду. Поэтому и в языке программирования `gawk` предусмотрена возможность комбинировать команды, создавая полноценную программу. Чтобы воспользоваться несколькими командами в программном сценарии, заданном в командной строке, достаточно разграничить команды точками с запятой:

```
$ echo "My name is Rich" | gawk '{ $4="Christine"; print $0 }'
My name is Christine
$
```

Первая команда присваивает значение переменной поля `$4`. Затем вторая команда выводит содержимое всего этого поля данных. Заслуживает внимания то, что в выводе программы `gawk` четвертое поле данных заменено в исходном тексте новым значением.

Для ввода команд программного сценария по одной строке одновременно можно также использовать вторичное приглашение командного интерпретатора:

```
$ gawk '{
> $4="testing"
> print $0 }'
This is not a good test.
This is not testing good test.
$
```

После задания открывающей одинарной кавычки командный интерпретатор `bash` предоставляет вторичное приглашение к вводу информации, запрашивая продолжение ввода данных. Предусмотрена возможность добавлять команды по одной в каждой строке, пока не будет введена закрывающая одинарная кавычка. Чтобы выйти из программы, достаточно нажать комбинацию клавиш `<Ctrl+D>` для указания на то, что на этом данные заканчиваются.

Чтение программы из файла

Как и в случае редактора `sed`, редактор `gawk` позволяет сохранять программы в файле и вызывать их из командной строки:

```
$ cat script2
{ print $1 "'s home directory is " $6 }
$
$ gawk -F: -f script2 /etc/passwd
root's home directory is /root
daemon's home directory is /usr/sbin
...
Samantha's home directory is /home/Samantha
Timothy's home directory is /home/Timothy
Christine's home directory is /home/Christine
$
```

В программном сценарии `script2` снова используется команда `print` для вывода содержимого поля с данными об исходном каталоге в файле `/etc/passwd` (переменная поля `$6`) и поле с данными об идентификаторе пользователя (переменная поля `$1`).

Предусмотрена возможность задавать в программном файле несколько команд. Для этого достаточно поместить каждую команду на отдельной строке. Точки с запятой использовать не требуется:

```
$
$ cat script3
{
text = "'s home directory is "
print $1 text $6
}
$
$ gawk -F: -f script3 /etc/passwd
root's home directory is /root
daemon's home directory is /usr/sbin
...
Samantha's home directory is /home/Samantha
Timothy's home directory is /home/Timothy
Christine's home directory is /home/Christine
$
```

В программном сценарии `script3` определена переменная, предназначенная для хранения текстовой строки, используемой в команде `print`. Заслуживает внимания то, что, в отличие от сценария командного интерпретатора, в программе `gawk` для ссылки на значение переменной не используется знак доллара.

Выполнение сценариев перед обработкой данных

Программа `gawk` позволяет также указывать, когда должен быть выполнен программный сценарий. По умолчанию программа `gawk` считывает строку текста из входного потока, а затем выполняет программный сценарий применительно к данным в строке текста. Иногда может потребоваться выполнить сценарий перед обработкой данных, например, чтобы создать раздел с заголовком для отчета. Для этой цели используется ключевое слово `BEGIN`. Команда `BEGIN` вынуждает программу `gawk` выполнить фрагмент программного сценария, заданный после ключевого слова `BEGIN`, и лишь затем наступает этап чтения данных программой `gawk`:

```
$ gawk 'BEGIN {print "Hello World!"}'
Hello World!
$
```

На этот раз команда `print` отображает текст перед чтением любых данных. Однако непосредственно после отображения текста сразу же происходит выход из программы без ожидания каких-либо данных.

Причина этого состоит в том, что ключевое слово `BEGIN` требует лишь выполнения заданного фрагмента сценария до начала обработки данных. Если требуется произвести обработку данных с помощью обычного программного сценария, то необходимо определить для этого соответствующую программу с помощью еще одного раздела сценария:

```
$
$ cat data4
Line 1
Line 2
Line 3
$
$ gawk 'BEGIN { print "The data4 File Contents:" } { print $0 }' data4
The data4 File Contents:
Line 1
Line 2
```

```
Line 3
$
```

Теперь после выполнения сценария `BEGIN` программой `gawk` происходит переход к использованию второго сценария для обработки всех необходимых данных из файла. Применяя подобные программы, необходимо соблюдать осторожность, поскольку все разделы сценария по-прежнему должны быть заданы в виде одной текстовой строки в командной строке вызова программы `gawk`. Соответствующим образом должны быть также размещены одинарные кавычки.

Выполнение сценариев после обработки данных

Как и ключевое слово `BEGIN`, ключевое слово `END` позволяет задать фрагмент программного сценария, выполняемого программой `gawk` после чтения данных:

```
$
$ gawk 'BEGIN { print "The data4 File Contents:" } { print $0 }
      END { print "End of File" }' data4
The data4 File Contents:
Line 1
Line 2
Line 3
End of File
$
```

Вслед за тем как программа `gawk` завершает вывод обработанного содержимого файла, начинается выполнение команд во фрагменте сценария `END`. В этом состоит замечательная возможность добавления данных нижнего колонтитула к отчетам после завершения обработки всех текущих данных.

Указанные элементы сценария могут быть собраны воедино как отдельный, удобный в использовании файл программного сценария, предназначенного для создания полного отчета из простого файла данных:

```
$ cat script4
BEGIN {
  print "The latest list of users and shells"
  print "  Userid      Shell"
  print "-----      -"
  FS=":"
}

{
  print $1 "      " $7
}

END {
  print "This concludes the listing"
}
$
```

В этом сценарии используется раздел `BEGIN` для создания раздела заголовка отчета. В нем также определена специальная переменная `FS`. В этом состоит еще один способ определения символа разделения полей. Благодаря этому исключается зависимость от того, будет ли поль-

зователь сценария учитывать требование по определению символа разделения полей в опциях командной строки.

Ниже приведен немного сокращенный пример вывода, полученный при выполнении этого сценария программы `gawk`:

```
$ gawk -f script4 /etc/passwd
The latest list of users and shells
  Userid      Shell
  -----
root         /bin/bash
daemon       /bin/sh
bin          /bin/sh
...
Samantha     /bin/bash
Timothy      /bin/sh
Christine    /bin/sh
This concludes the listing
$
```

Как и следовало ожидать, раздел `BEGIN` создает текст заголовка, программный сценарий обрабатывает информацию из указанного файла данных (файла `/etc/passwd`), а раздел `END` выводит текст нижнего колонтитула.

Изучая этот сценарий, читатель может хоть в какой-то степени ощутить, какие возможности открываются при использовании даже самых простых сценариев `gawk`. В главе 21 рассматриваются некоторые другие основные программные конструкции, которые могут применяться в сценариях `gawk`, а также описаны более сложные концепции программирования, применимые в программных сценариях `gawk`, с помощью которых можно создавать превосходно отформатированные отчеты исходя из самых сложных по структуре файлов данных.

Основные сведения о редакторе `sed`

Ключом к успешному использованию редактора `sed` является то, что при работе с ним необходимо правильно задавать команды и форматы, предназначенные для настройки процесса редактирования текста, количество которых весьма велико. В настоящем разделе приведено описание некоторых из основных команд и средств, которые можно включить в сценарий для начала использования редактора `sed`.

Более подробное описание опций подстановки

Выше в данной главе уже было показано, как использовать команду `s` (сокращение от `substitute`) для подстановки нового текста вместо существующего в строке. Но в команде `substitute` можно задавать несколько дополнительных опций, позволяющих упростить обработку текста.

Флаги подстановки

Применяя команду `substitute` для подстановки нового текста вместо существующего текста, сопоставляемого с шаблонами в текстовой строке, необходимо соблюдать осторожность. Рассмотрим, что происходит в следующем примере:

```
$ cat data5
This is a test of the test script.
This is the second test of the test script.
$
$ sed 's/test/trial/' data5
This is a trial of the test script.
This is the second trial of the test script.
$
```

Команда `substitute` превосходно справляется с заменой текста в нескольких строках, но по умолчанию она заменяет только первое вхождение в каждой строке. Чтобы обеспечить обработку с помощью команды `substitute` нескольких вхождений текста, необходимо задать *флаг подстановки*. Флаг подстановки задается после строк команды подстановки:

s/pattern/replacement/flags

Предусмотрены флаги подстановки четырех указанных ниже типов.

- Число. Указывает, к какому от начала строки вхождению шаблона должна быть применена операция подстановки нового текста вместо старого.
- *g*. Указывает, что новый текст необходимо подставить вместо всех вхождений существующего текста.
- *p*. Указывает, что необходимо вывести содержимое исходной строки.
- *w file*. Предусматривает запись результатов подстановки в файл.

При подстановке первого типа можно указать, вместо какого вхождения шаблона сопоставления редактор `sed` должен подставить новый текст:

```
$ sed 's/test/trial/2' data5
This is a test of the trial script.
This is the second test of the trial script.
$
```

Если в качестве флага подстановки будет задано число 2, редактор `sed` заменит текст, определяемый шаблоном, новым текстом, только во втором вхождении шаблона в каждой строке. Флаг подстановки *g* позволяет выполнить замену для всех вхождений шаблона в тексте:

```
$ sed 's/test/trial/g' data5
This is a trial of the trial script.
This is the second trial of the trial script.
$
```

Флаг подстановки *p* выводит строку, содержащую шаблон сопоставления, который задан в команде `substitute`. Этот флаг чаще всего применяется в сочетании с опцией `-n` редактора `sed`:

```
$ cat data6
This is a test line.
This is a different line.
$
$ sed -n 's/test/trial/p' data5
This is a trial line.
$
```

Опция `-n` подавляет вывод из редактора `sed`. Тем не менее флаг подстановки *p* выводит все строки, которые были изменены. Применение сочетания этих двух флагов приводит

к формированию таких результатов, в которых показаны только строки, измененные командой `substitute`.

Флаг подстановки `w` предусматривает формирование такого же вывода, но полученные данные сохраняются в указанном файле:

```
$ sed 's/test/trial/w test' data6
This is a trial line.
This is a different line.
$
$ cat test
This is a trial line.
$
```

Обычные выходные данные редактора `sed` поступают в поток `STDOUT`, но только строки, которые включают шаблон сопоставления, сохраняются в указанном выходном файле.

Замена символов

Иногда приходится сталкиваться с тем, что в текстовых строках имеются символы, которые невозможно задать в непосредственном виде в шаблоне подстановки. В мире Linux одним из характерных примеров подобных символов является косая черта.

Каждый из таких символов необходимо сопроводить экранирующим символом, в результате, например, для замены имен путей в файлах приходится применять чрезмерно сложные конструкции. В частности, предположим, что в файле `/etc/passwd` вместо командного интерпретатора `bash` необходимо указать командный интерпретатор `C`. Для этого можно выполнить следующую команду:

```
$ sed 's/\/bin\/bash/\/bin\/csh/' /etc/passwd
```

Отметим, что косая черта используется в качестве разграничителя строк, но этот символ невозможно задать непосредственно в тексте шаблона, поэтому каждому символу косой черты должен предшествовать символ обратной косой черты, который выполняет роль экранирующего символа. Это часто приводит к путанице и ошибкам.

Чтобы можно было проще решить эту проблему, в редакторе `sed` предусмотрена возможность выбрать в команде `substitute` другой символ в качестве разграничителя полей строки:

```
$ sed 's!/bin/bash!/bin/csh!' /etc/passwd
```

В данном примере разграничителем полей строки служит восклицательный знак, поэтому задача чтения и анализа имен путей значительно упрощается.

Использование адресов

По умолчанию команды, заданные для выполнения в редакторе `sed`, применяются ко всем строкам текстовых данных. Но иногда требуется применить команду только к конкретной строке или группе строк, для чего предназначена *построчная адресация*.

В редакторе `sed` предусмотрены две формы построчной адресации:

- задание диапазона номеров строк;
- применение текстового шаблона, позволяющего отфильтровывать нужные строки.

В обеих формах для указания адреса *address* используется одинаковый формат:

```
[address] command
```


Предусмотрена также возможность собрать в группу несколько команд для применения к строкам с конкретным адресом:

```
address {  
    command1  
    command2  
    command3  
}
```

Редактор `sed` проводит обработку с помощью заданных команд только тех строк, которые согласуются с указанным адресом.

В настоящем разделе демонстрируется использование обоих этих методов адресования в сценариях редактора `sed`.

Числовая построчная адресация

Если используется числовая построчная адресация, то для ссылки на строки служат данные об их положении в текстовом потоке. Редактор `sed` присваивает первой строке в текстовом потоке номер один, затем последовательно наращивает номера строк после обработки каждого символа перевода на новую строку.

Адрес, указанный в команде, может представлять собой номер единственной строки или диапазон номеров строк, который задан как номер начальной строки, запятая и номер конечной строки. Ниже приведен пример определения номера строки, к которой должна применяться команда `sed`.

```
$ sed '2s/dog/cat/' data1  
The quick brown fox jumps over the lazy dog  
The quick brown fox jumps over the lazy cat  
The quick brown fox jumps over the lazy dog  
The quick brown fox jumps over the lazy dog  
$
```

Изменения текста внесены редактором `sed` только по указанному адресу, в двух строках, начиная со строки два. Ниже приведен еще один пример, но на этот раз с использованием диапазона адресов строк.

```
$ sed '2,3s/dog/cat/' data1  
The quick brown fox jumps over the lazy dog  
The quick brown fox jumps over the lazy cat  
The quick brown fox jumps over the lazy cat  
The quick brown fox jumps over the lazy dog  
$
```

Если команду необходимо применить к группе строк, начинающейся с некоторой позиции в тексте и продолжающейся до конца текста, можно задать специальный адрес, знак доллара:

```
$ sed '2,$s/dog/cat/' data1  
The quick brown fox jumps over the lazy dog  
The quick brown fox jumps over the lazy cat  
The quick brown fox jumps over the lazy cat  
The quick brown fox jumps over the lazy cat  
$
```

Дело в том, что не всегда известно заранее, каково количество строк данных в тексте, поэтому знак доллара является очень удобным.

Использование фильтров в виде текстовых шаблонов

Предусмотрен еще один метод сокращенного обозначения строк, к которым применяется команда, но он немного сложнее. Редактор `sed` позволяет задавать текстовый шаблон, используемый для фильтрации строк, которые предназначены для обработки с помощью команды. Для этого применяется следующий формат:

```
/pattern/command
```

Параметр *pattern* (шаблон) должен быть задан между символами косой черты. Редактор `sed` будет применять команду только к строкам, в которых содержится указанный текстовый шаблон.

Например, если требуется сменить применяемый по умолчанию командный интерпретатор только для пользователя `Samantha`, можно использовать команду `sed`:

```
$
$ grep Samantha /etc/passwd
Samantha:x:1001:1002:Samantha,4,,:/home/Samantha:/bin/bash
$
$ sed '/Samantha/s/bash/csh/' /etc/passwd
root:x:0:0:root:/root:/bin/bash
...
Samantha:x:1001:1002:Samantha,4,,:/home/Samantha:/bin/csh
Timothy:x:1002:1005::/home/Timothy:/bin/sh
Christine:x:1003:1006::/home/Christine:/bin/sh
$
```

Очевидно, что команда была применена лишь к строке, которая содержит сопоставляемый текстовый шаблон. Безусловно, применение шаблона в виде строки с неизменным значением в некоторых случаях является вполне удовлетворительным, как в приведенном примере с идентификатором пользователя, `userid`, но возможности такой конструкции несколько ограничены. Редактор `sed` позволяет использовать в текстовых шаблонах средство, называемое *регулярными выражениями*, благодаря чему появляется возможность создавать достаточно сложные шаблоны.

С помощью регулярных выражений можно создавать усовершенствованные текстовые шаблоны, представляющие собой фактически формулы сопоставления с данными любых типов. В подобных формулах применяются в сочетании ряды подстановочных знаков, специальных символов и фиксированных текстовых символов для формирования четкого шаблона, который может быть правильно подобран с учетом практически любой ситуации обработки текста. При работе с регулярными выражениями приходится сталкиваться с одной из наиболее сложных частей программирования сценариев командного интерпретатора, и в главе 19 приведено их более подробное описание.

Группирование команд

Если требуется выполнить несколько команд, приведенных в одной строке, то следует сгруппировать эти команды с использованием фигурных скобок. Редактор `sed` будет обрабатывать каждую команду, перечисленную в строках адреса:

```
$ sed '2{
> s/fox/elephant/
> s/dog/cat/
> }' data1
```

```
The quick brown fox jumps over the lazy dog.
The quick brown elephant jumps over the lazy cat.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
$
```

Обе команды выполняются применительно к данному адресу. Кроме того, безусловно, можно также указывать диапазон адресов перед сгруппированными командами:

```
$ sed '3,$ {
> s/brown/green/
> s/lazy/active/
> }' data1
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick green fox jumps over the active dog.
The quick green fox jumps over the active dog.
$
```

Редактор `sed` применяет все эти команды ко всем строкам в диапазоне адресов.

Удаление строк

Команды, предусмотренные в редакторе `sed`, не ограничиваются лишь командой подстановки одних текстовых строк вместо других. На тот случай, что придется удалить конкретные строки текста в потоке текста, имеется команда `delete`.

Действие, выполняемое командой `delete` (сокращенно `d`), в основном соответствует ее названию. Эта команда удаляет все текстовые строки, сопоставляемые с заданной схемой адресации. При использовании команды `delete` необходимо соблюдать осторожность, поскольку, если не будет задана схема адресации, произойдет удаление всех строк из потока:

```
$ cat data1
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
$ sed 'd' data1
$
```

Очевидно, что команда `delete` становится наиболее удобной при использовании в сочетании с указанным адресом. С ее помощью можно удалять конкретные строки текста из потока данных либо по номеру строки:

```
$ sed '3d' data7
This is line number 1.
This is line number 2.
This is line number 4.
$
```

либо с указанием конкретного диапазона строк:

```
$ sed '2,3d' data7
This is line number 1.
This is line number 4.
$
```

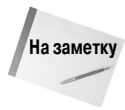
либо с использованием специального символа конца файла:

```
$ sed '3,$d' data7
This is line number 1.
This is line number 2.
$
```

В команде `delete` может применяться и средство сопоставления с шаблонами редактора `sed`:

```
$ sed '/number 1/d' data7
This is line number 2.
This is line number 3.
This is line number 4.
$
```

Редактор `sed` удаляет строку, содержащую текст, который сопоставляется с указанным шаблоном.



Напомним, что редактор `sed` не затрагивает исходный файл. Все удаленные строки исчезают только в выходных данных редактора `sed`. Исходный файл по-прежнему содержит строки, которые пользователь может считать уже "удаленными".

Предусмотрена также возможность удаления диапазона строк с использованием двух текстовых шаблонов, но при этом необходимо соблюдать осторожность. Первый заданный шаблон "включает" удаление строк, а второй "выключает" удаление строк. Редактор `sed` удаляет все строки, находящиеся между двумя указанными строками (в том числе и сами указанные строки):

```
$ sed '/1/,/3/d' data6
This is line number 4.
$
```

Кроме того, необходимо соблюдать осторожность и в связи с тем, что это средство удаления "включается" при каждом обнаружении редактором `sed` начального шаблона в потоке данных. Это может привести к получению непредвиденных результатов:

```
$ cat data8
This is line number 1.
This is line number 2.
This is line number 3.
This is line number 4.
This is line number 1 again.
This is text you want to keep.
This is the last line in the file.
$
$ sed '/1/,/3/d' data7
This is line number 4.
$
```

Обнаружение второго вхождения строки, в которой содержится цифра 1, привело к повторному вызову команды `delete` и удалению остальной части строк в потоке данных, поскольку конечный шаблон не был обнаружен. Безусловно, еще одна очевидная проблема возникает, если задан такой конечный шаблон, который так и не обнаруживается в тексте:

```
$ sed '/1/,/5/d' data8
$
```

В данном случае средства удаления были “включены” при первом обнаружении начального сопоставления с шаблоном, а конечное сопоставление с шаблоном так и не было найдено, поэтому произошло удаление всего остального потока данных.

Вставка и добавление текста

Как и следовало ожидать, редактор `sed`, подобно любому другому редактору, позволяет вставлять текстовые строки в поток данных и присоединять к концу потока данных дополнительные строки. Изучая различия между этими двумя действиями, необходимо учитывать следующее, чтобы избежать путаницы:

- команда `insert` (сокращенно `i`) добавляет символ перевода на новую строку перед указанной строкой;
- команда `append` (сокращенно `a`) добавляет символ перевода на новую строку после указанной строки.

Причиной возможной путаницы при работе с этими двумя командами являются их форматы. Задавая команду вставки или добавления, нельзя приводить саму команду и ее данные в одной командной строке. Строку, подлежащую вставке или добавлению, необходимо задавать отдельно, на отдельной строке. Для этого применяется следующий формат:

```
sed '[address]command\
new line'
```

Текст, приведенный в строке `new line`, появляется в выходных данных редактора `sed` в том месте, которое указано в команде. Напомним, что при использовании команды `insert` текст появляется перед текстовым содержимым потока данных:

```
$ echo "Test Line 2" | sed 'i\Test Line 1'
Test Line 1
Test Line 2
$
```

А при использовании команды `append` добавляемая строка появляется после указанного текста в потоке данных:

```
$ echo "Test Line 2" | sed 'a\Test Line 1'
Test Line 2
Test Line 1
$
```

Если работа с редактором `sed` выполняется в приглашении интерфейса командной строки, то появляется вторичное приглашение к вводу информации, в котором должна быть введена вставляемая (или добавляемая) строка. После ввода новой строки в этом приглашении необходимо также ввести последнюю часть команды редактора `sed`. Вслед за тем, как вводится закрывающая одинарная кавычка, командный интерпретатор `bash` приступает к обработке команды:

```
$ echo "Test Line 2" | sed 'i\
> Test Line 1'
Test Line 1
Test Line 2
$
```

Такой формат команды позволяет легко добавить текст до или после всего заданного текста в потоке данных, но иногда возникает необходимость добавления текста в поток данных.

Чтобы вставить или добавить данные между строками в потоке данных, необходимо использовать адресование, чтобы указать редактору `sed`, где должны появиться данные. При использовании описанных выше команд каждый раз можно указывать адрес только одной строки. Для сопоставления с конкретной строкой можно применять либо числовой номер строки, либо текстовый шаблон, но возможность использования диапазона адресов не предусмотрена. И это ограничение вполне оправдано, поскольку нельзя представить себе, как можно произвести вставку или добавление, ориентируясь на диапазон строк, а не на отдельную строку.

Ниже приведен пример вставки новой строки перед строкой 3 в потоке данных.

```
$ sed '3i\  
> This is an inserted line.' data7  
This is line number 1.  
This is line number 2.  
This is an inserted line.  
This is line number 3.  
This is line number 4.  
$
```

А в следующем примере новая строка добавляется после строки 3 в потоке данных:

```
$ sed '3a\  
>This is an appended line.' data7  
This is line number 1.  
This is line number 2.  
This is line number 3.  
This is an appended line.  
This is line number 4.  
$
```

При этом осуществляется такой же процесс, как и во время работы команды `insert`; различие заключается лишь в том, что вставка новой текстовой строки происходит не до, а после строки с указанным номером. Если поток данных включает много строк и требуется присоединить новую строку текста к концу потока данных, то достаточно задать знак доллара, `$`, который представляет последнюю строку данных:

```
$ sed '$a\  
> This is a new line of text.' data7  
This is line number 1.  
This is line number 2.  
This is line number 3.  
This is line number 4.  
This is a new line of text.  
$
```

Тот же принцип применяется, если возникает необходимость добавить новую строку до начала потока данных. Для этого достаточно предусмотреть вставку новой строки перед строкой с номером один.

Чтобы вставить или добавить несколько строк текста, необходимо вводить обратную косую черту после каждой строки вновь задаваемого текста, пока не будет достигнута последняя текстовая строка в том месте, где требуется вставить или добавить текст:

```
$ sed 'li\
> This is one line of new text.\
> This is another line of new text.' data7
This is one line of new text.
This is another line of new text.
This is line number 1.
This is line number 2.
This is line number 3.
This is line number 4.
$
```

Обе заданные строки добавляются к потоку данных.

Внесение изменений в строки

Команда `change` позволяет изменить содержимое всей строки текста в потоке данных. Эта команда имеет формат, аналогичный формату команд `insert` и `append`, поскольку в ней вновь вводимая строка также должна быть задана отдельно от остальной части команды `sed`:

```
$ sed '3c\
> This is a changed line of text.' data7
This is line number 1.
This is line number 2.
This is a changed line of text.
This is line number 4.
$
```

В этом примере редактор `sed` изменяет текст в строке с номером 3. Для адресования строки можно также использовать текстовый шаблон:

```
$ sed '/number 3/c\
> This is a changed line of text.' data7
This is line number 1.
This is line number 2.
This is a changed line of text.
This is line number 4.
$
```

Команда `change` с этим текстовым шаблоном вносит изменения во все строки текста в потоке данных, сопоставление с которыми оканчивается успешно.

```
$ sed '/number 1/c\
> This is a changed line of text.' data8
This is a changed line of text.
This is line number 2.
This is line number 3.
This is line number 4.
This is a changed line of text.
This is yet another line.
This is the last line in the file.
$
```

В команде `change` допускается применение диапазона адресов, но полученные при этом результаты могут оказаться не такими, как ожидается:

```
$ sed '2,3c\  
> This is a new line of text.' data7  
This is line number 1.  
This is a new line of text.  
This is line number 4.  
$
```

Вместо изменения обеих строк с заданным текстом редактор `sed` применил одну и ту же заданную строку текста для замены и той и другой строки.

Команда `transform`

Команда `transform` (сокращенно `y`) — это единственная команда редактора `sed`, действие которой распространяется на отдельные символы. В команде `transform` используется следующий формат:

```
[address]y/inchars/outchars/
```

Команда `transform` выполняет взаимно однозначное отображение значений символов *inchars* в значения символов *outchars*. Первый символ из *inchars* преобразуется в первый символ из *outchars*. Второй символ из *inchars* преобразуется во второй символ из *outchars*. Такое отображение указанных символов продолжается по всей длине заданных строк *inchars* и *outchars*. Если строки *inchars* и *outchars* не имеют одинаковую длину, то редактор `sed` формирует сообщение об ошибке.

Ниже приведен простой пример использования команды `transform`.

```
$ sed 'y/123/789/' data8  
This is line number 7.  
This is line number 8.  
This is line number 9.  
This is line number 4.  
This is line number 7 again.  
This is yet another line.  
This is the last line in the file.  
$
```

Как показывают результаты выполнения этой команды, каждый экземпляр из числа символов, указанных в шаблоне *inchars*, был заменен символом из той же позиции в шаблоне *outchars*.

Команда `transform` — это глобальная команда; иными словами, с ее помощью автоматически выполняется преобразование всех символов, обнаруженных в текстовой строке, не учитывая того, в какой позиции находится само вхождение того или иного символа:

```
$ echo "This 1 is a test of 1 try." | sed 'y/123/456/'  
This 4 is a test of 4 try.  
$
```

Редактор `sed` преобразовал в текстовой строке оба экземпляра символа `1`, сопоставленного с шаблоном. Возможность ограничивать преобразование лишь конкретными вхождениями символов отсутствует.

Дополнительные сведения о формировании выходных данных

В разделе “Более подробное описание опций подстановки” было показано, как использовать с командой подстановки флаг `p` для отображения строк, измененных редактором `sed`. Для вывода информации, поступающей из потока данных, могут также использоваться следующие команды:

- команда `p` (строчная буква `P`), предназначенная для вывода текстовых строк;
- команда `=` (знак равенства), с помощью которой можно показать номера строк;
- команда `l` (строчная буква `L`), предназначенная для вывода строк с произвольным содержанием.

В следующих разделах описано, как каждая из этих трех команд применяется для вывода данных в редакторе `sed`.

Вывод строк

Подобно тому, как действует флаг `p` в команде `substitution`, команда `p` отправляет строку в выходной поток редактора `sed`. Отдельно взятая, эта команда не выполняет каких-либо сложных действий:

```
$ echo "this is a test" | sed 'p'
this is a test
this is a test
$
```

Все результаты ее применения сводятся к тому, что в выходной поток поступают текстовые данные, в отношении которых и без того известно, что они должны присутствовать в выводе. Гораздо чаще команда `print` (сокращенно `p`) применяется для вывода строк, которые содержат текст, сопоставленный с текстовым шаблоном:

```
$ sed -n '/number 3/p' data7
This is line number 3.
$
```

Используя в командной строке опцию `-n`, можно подавить вывод всех прочих строк и включить в состав результатов только такие строки, которые содержат текст, сопоставленный с текстовым шаблоном.

Команду в таком формате можно также использовать как удобный способ вывода в поток данных определенного подмножества строк:

```
$ sed -n '2,3p' data7
This is line number 2.
This is line number 3.
$
```

Команду `print` можно также использовать, если необходимо видеть содержимое строки до внесения в нее изменения, например, с помощью команды `substitution` или `change`. Сценарий, в котором отображается строка до ее изменения, можно составить следующим образом:

```
$ sed -n '/3/{
p
s/line/test/p
}
```

```
} 'data7
This is line number 3.
This is test number 3.
$
```

В этой команде редактора `sed` происходит поиск строки, содержащей цифру 3, а затем выполняются две команды. Прежде всего, в сценарии используется команда `p` для вывода исходной версии строки; затем в нем для замены текста выполняется команда `s` наряду с флагом `p` для вывода результирующего текста. В выводе команды отображаются и исходный текст строки, и новый текст строки.

Вывод номеров строк

Команда `=` (знак равенства) выводит номер текущей строки в потоке данных. Номера строк увеличиваются после обнаружения каждого очередного символа обозначения конца строки в потоке данных. Каждый раз, когда в потоке данных обнаруживается символ обозначения конца строки, редактор `sed` обрабатывает этот символ как признак завершения строки текста:

```
$ sed '=' data1
1
The quick brown fox jumps over the lazy dog.
2
The quick brown fox jumps over the lazy dog.
3
The quick brown fox jumps over the lazy dog.
4
The quick brown fox jumps over the lazy dog.
$
```

Редактор `sed` выводит номер строки перед самым текстовым содержимым строки. Команда `=` может применяться при поиске конкретного текстового шаблона в потоке данных:

```
$ sed -n '/number 4/{
=
p
}' data7
4
This is line number 4.
$
```

Путем задания опции `-n` можно обеспечить, чтобы редактор `sed` отображал только те строки, которые содержат текст, сопоставленный с шаблоном, а также указывал их номера строк.

Вывод строк с произвольным содержимым

Команда `list` (сокращенно `l`) позволяет выводить в поток данных и текстовые символы, и непечатаемые символы ASCII. Все непечатаемые символы отображаются с использованием либо их восьмеричных значений, которым предшествует обратная косая черта, либо стандартных обозначений широко применяемых непечатаемых символов в стиле языка C, таких как обозначение `\t` для знаков табуляции:

```
$ cat data9
This   line   contains      tabs.
$
$ sed -n 'l' data9
```

```
This\tline\tcontains\ttabs.$  
$
```

Если знаки табуляции служат для форматирования текста, то в соответствующих им позициях также приводится обозначение `\t`. Знак доллара в конце строки указывает символ обозначения конца строки. Если в потоке данных содержатся экранирующие символы, то при использовании команды `list` отображение этих символов также происходит с применением восьмеричного кода:

```
$ cat data10  
This line contains an escape character  
$  
$ sed -n 'l' data10  
This line contains an escape character \033[44m$  
$
```

В текстовом файле `data10` содержатся экранирующие управляющие коды (см. главу 17), предназначенные для изменения цвета дисплея. Если этот текстовый файл выводится с использованием команды `cat`, то такие экранирующие управляющие коды не отображаются, но происходит их обработка и, соответственно, изменяется цвет на мониторе.

А с помощью команды `list` можно ознакомиться с тем, какие экранирующие управляющие коды фактически заданы в файле. Например, `\033` — это восьмеричное значение кода ASCII для клавиши `<Escape>`.

Использование файлов при работе с редактором `sed`

Команда `substitution` поддерживает флаги, позволяющие работать с файлами. В этих целях могут применяться и другие команды редактора `sed`, которые не связаны с необходимостью предусматривать замену текста.

Запись в файл

Для записи строк в файл предназначена команда `w`, которая имеет следующий формат:

```
[address]w filename
```

Значение имени файла `filename` может быть задано с указанием относительного или полного имени пути, но в любом случае пользователь редактора `sed` должен иметь разрешение на запись в файл. Для обозначения адреса может применяться метод адресования любого типа, используемый в редакторе `sed`, будь то метод, основанный на указании единственного номера строки, текстового шаблона, диапазона номеров строк или текстовых шаблонов.

Ниже приведен пример, в котором происходит вывод в текстовый файл только первых двух строк из потока данных.

```
$ sed '1,2w test' data7  
This is line number 1.  
This is line number 2.  
This is line number 3.  
This is line number 4.  
$  
$ cat test  
This is line number 1.
```

```
This is line number 2.  
$
```

Разумеется, это не исключает возможность использовать опцию `-n` команды `sed`, чтобы предотвратить вывод строк в поток `STDOUT`.

Это — превосходный инструмент, который может применяться, если необходимо создать файл данных из основного файла на основе общих текстовых значений, наподобие тех, что содержатся в списке рассылки:

```
$ cat data11  
Blum, Katie      Chicago, IL  
Mullen, Riley    West Lafayette, IN  
Snell, Haley     Ft. Wayne, IN  
Woenker, Matthew Springfield, IL  
Wisecarver, Emma Grant Park, IL  
$  
$ sed -n '/IN/w INcustomers' data11  
$  
$ cat INcustomers  
Mullen, Riley    West Lafayette, IN  
Snell, Haley     Ft. Wayne, IN  
$
```

Редактор `sed` записывает в целевой файл только строки данных, которые содержат текстовый шаблон.

Чтение данных из файла

Выше уже было описано, как вставлять данные и добавлять текст к потоку данных из командной строки `sed`. Команда `read` (сокращенно `r`) позволяет вставлять данные, содержащиеся в отдельном файле.

Команда `read` имеет следующий формат:

```
[address]r filename
```

Параметр *filename* указывает абсолютное или относительное имя пути для файла, который содержит данные. При использовании команды `read` не предусмотрена возможность задавать диапазон адресов. Разрешается задавать только единственный номер строки или адрес в виде текстового шаблона. Редактор `sed` вставляет текст из файла после строки, указанной этим адресом.

```
$ cat data12  
This is an added line.  
This is the second added line.  
$  
$ sed '3r data12' data7  
This is line number 1.  
This is line number 2.  
This is line number 3.  
This is an added line.  
This is the second added line.  
This is line number 4.  
$
```

При этом редактор `sed` вставляет в поток данных все текстовые строки из файла данных. Такое же действие осуществляется при использовании адреса, заданного текстовым шаблоном:

```
$ sed '/number 2/r data12' data7
This is line number 1.
This is line number 2.
This is an added line.
This is the second added line.
This is line number 3.
This is line number 4.
$
```

Если текст должен быть добавлен к концу потока данных, достаточно лишь воспользоваться в качестве символического обозначения адреса знаком доллара:

```
$ sed '$r data12' data7
This is line number 1.
This is line number 2.
This is line number 3.
This is line number 4.
This is an added line.
This is the second added line.
$
```

Превосходным приложением команды `read` является ее использование в сочетании с командой `delete` для замены местозаполнителя в файле данными из другого файла. Например, предположим, что в текстовом файле хранится образец письма, который выглядит примерно таким образом:

```
$ cat letter
Would the following people:
LIST
please report to the office.
$
```

В этом образце письма используется общий местозаполнитель `LIST` вместо списка адресатов. Чтобы вставить действительный список адресатов после местозаполнителя, остается лишь воспользоваться командой `read`. Но при этом в выходных данных по-прежнему остается заполняющий текст. Чтобы удалить его, достаточно применить команду `delete`. Результат выглядит следующим образом:

```
$ sed '/LIST/{
> r data11
> d
> }' letter
Would the following people:
Blum, Katie      Chicago, IL
Mullen, Riley    West Lafayette, IN
Snell, Haley     Ft. Wayne, IN
Woenker, Matthew Springfield, IL
Wisecarver, Emma Grant Park, IL
please report to the office.
$
```

Итак, заполняющий текст заменен списком адресатов из файла данных.

Резюме

Очевидно, что с помощью лишь одних сценариев командного интерпретатора можно выполнять большой объем работы, но если приходится применять только команды командного интерпретатора, то задача манипулирования данными становится затруднительной. В системе Linux предусмотрены две удобные программы, с помощью которых можно значительно упростить обработку текстовых данных. Одной из них является редактор `sed` — потоковый редактор, который быстро, динамически обрабатывает данные в процессе их считывания. Перед вызовом редактора `sed` необходимо подготовить список команд редактирования, применяемых им к данным.

Вторая программа — это программа `gawk`, разработанная организацией GNU, которая повторяет и расширяет функциональные возможности программы `awk` системы Unix. Программа `gawk` поддерживает встроенный язык программирования, который может использоваться для написания сценариев манипулирования данными и обработки данных. Программу `gawk` можно использовать для извлечения элементов данных из крупных файлов данных и последующего вывода результатов почти в любом желаемом формате. В результате существенно упрощается обработка файлов большого объема, например, журналов, а также появляется возможность легко создавать определяемые пользователем отчеты по данным из файлов данных.

При использовании обеих программ, `sed` и `gawk`, крайне важно знать, как действуют регулярные выражения. Регулярные выражения являются ключом к созданию специализированных фильтров, которые служат для извлечения данных и манипулирования ими в текстовых файлах. В следующей главе раскрывается мир регулярных выражений, изучение которого часто связано со значительными затруднениями, а также показано, как формировать регулярные выражения для манипулирования данными любых типов.

Регулярные выражения

Ключом к успешному применению редактора `sed` и программы `gawk` в сценариях командного интерпретатора является полное овладение принципами работы с регулярными выражениями. Использование регулярных выражений — не всегда простая задача, поэтому для обеспечения фильтрации данных из большой совокупности данных могут потребоваться значительные трудозатраты. В настоящей главе описано, как создавать регулярные выражения для редактора `sed` и программы `gawk`, чтобы иметь возможность отфильтровывать только нужные данные.

Общее определение понятия регулярного выражения

Первый шаг к пониманию того, что такое регулярные выражения, состоит в точном определении понятия регулярного выражения. В данном разделе приведено описание того, чем является регулярное выражение, и показано, как используются регулярные выражения в системе Linux.

Определение

Регулярное выражение — это заданный особым образом шаблон, который используется в программе Linux для фильтрации текста. В программе Linux (такой как редактор `sed` или программа `gawk`) шаблон регулярного выражения сопоставляется с данными по мере поступления данных в программу. Если данные успешно сопоставляются

ГЛАВА

19

В этой главе...

Общее определение понятия регулярного выражения

Определение шаблонов BRE

Расширенные регулярные выражения

Регулярные выражения в действии

Резюме

Согласование данных с шаблоном регулярного выражения

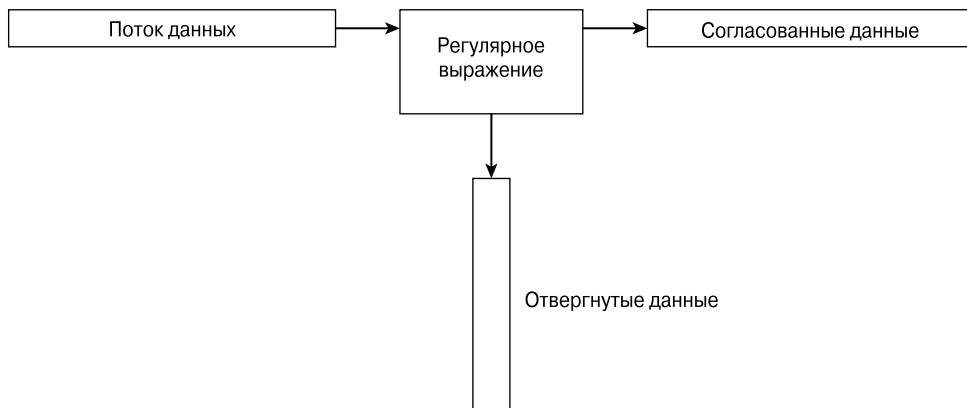


Рис. 19.1. Сопоставление данных с шаблоном регулярного выражения

с шаблоном, то принимаются для обработки. При отсутствии сопоставления данных с шаблоном эти данные отвергаются. Иллюстрация к этому описанию приведена на рис. 19.1.

В шаблоне регулярного выражения используются подстановочные знаки для представления одного или нескольких символов в потоке данных. При работе в системе Linux возможность задавать подстановочные знаки для представления данных, о которых нет полных сведений, используется очень часто. В настоящей книге уже был приведен пример применения подстановочных знаков с командой `ls` системы Linux для формирования листинга файлов и каталогов (см. главу 3).

Подстановочный знак звездочки позволяет включить в список только те файлы, которые соответствуют определенным критериям. Например:

```

$ ls -al da*
-rw-r--r--  1 rich    rich      45 Nov 26 12:42 data
-rw-r--r--  1 rich    rich      25 Dec  4 12:40 data.tst
-rw-r--r--  1 rich    rich     180 Nov 26 12:42 data1
-rw-r--r--  1 rich    rich      45 Nov 26 12:44 data2
-rw-r--r--  1 rich    rich      73 Nov 27 12:31 data3
-rw-r--r--  1 rich    rich      79 Nov 28 14:01 data4
-rw-r--r--  1 rich    rich     187 Dec  4 09:45 datatest
$
  
```

Параметр `da*` служит для команды `ls` указанием, что в выходных данных должны быть перечислены только те файлы, имена которых начинаются с подстроки `da`. При этом в имени файла после `da` может присутствовать любое количество символов (включая нулевое количество). В ходе выполнения команды `ls` осуществляется считывание информации об именах всех файлов в каталоге, но отображаются только те имена, которые сопоставляются с выражением, содержащим подстановочный знак.

Аналогичным образом действуют подстановочные знаки в шаблонах регулярных выражений. Шаблон регулярного выражения содержит текст и (или) специальные символы, определяющие образец для редактора `sed` и программы `gawk`, которому необходимо следовать при сопоставлении данных с шаблоном. Предусмотрен целый ряд специальных символов, которые могут использоваться в регулярном выражении в целях определения конкретного шаблона для фильтрации данных.

Типы регулярных выражений

Наибольшая сложность при использовании регулярных выражений заключается в том, что отсутствует единый подход к их формированию. В среде Linux в различных приложениях используются регулярные выражения разных типов. К ним относятся такие разнообразные приложения, как языки программирования (Java, Perl и Python), программы Linux (наподобие редактора sed, программы gawk и программы grep), а также целые программные системы (например, серверы баз данных MySQL и PostgreSQL).

Регулярное выражение вводится в действие с использованием *обработчика регулярных выражений*. Обработчик регулярных выражений — это основополагающее программное обеспечение, которое интерпретирует шаблоны регулярных выражений и использует эти шаблоны для сопоставления с текстом.

В мире Linux широко применяются следующие два обработчика регулярных выражений:

- обработчик базовых регулярных выражений (Basic Regular Expression — BRE) по стандарту POSIX;
- обработчик расширенных регулярных выражений (Extended Regular Expression — ERE) по стандарту POSIX.

Большинство программ Linux как минимум соответствует спецификации обработчика BRE POSIX, обеспечивая распознавание всех символов шаблона, определенных в этой спецификации. Но, к сожалению, некоторые программы (в частности, редактор sed) поддерживают только подмножество спецификации обработчика BRE. Это обусловлено необходимостью соблюдать ограничения по быстродействию, поскольку редактор sed предназначен для максимально быстрой обработки текста в потоке данных.

Обработчик POSIX ERE часто применяется в языках программирования, в которых для фильтрации текста предусмотрено использование регулярных выражений. Обработчик POSIX ERE поддерживает дополнительные символы шаблона наряду с теми специальными символами, которые применяются в обычных шаблонах, например, для согласования с цифрами, отдельными словами и алфавитно-цифровыми символами. В программе gawk для обработки шаблонов регулярных выражений применяется обработчик ERE.

Для решения одной и той же задачи фильтрации можно применить очень много разных способов реализации регулярных выражений, поэтому нелегко составить общее краткое описание для всех возможных регулярных выражений. В следующих разделах рассматриваются наиболее широко применяемые регулярные выражения и показано, как их использовать в редакторе sed и программе gawk.

Определение шаблонов BRE

Наиболее простыми являются такие шаблоны BRE, которые служат для сопоставления с текстовыми символами в потоке данных. В настоящем разделе показано, как задать текст в шаблоне регулярного выражения и чего следует ожидать в качестве результатов.

Обычный текст

В главе 18 было показано, как использовать обычные текстовые строки в редакторе sed и программе gawk для фильтрации данных. Ниже приведен пример, позволяющий освежить эти сведения в памяти.

```
$ echo "This is a test" | sed -n '/test/p'
This is a test
$ echo "This is a test" | sed -n '/trial/p'
$
$ echo "This is a test" | gawk '/test/{print $0}'
This is a test
$ echo "This is a test" | gawk '/trial/{print $0}'
$
```

Первый шаблон определяет отдельное слово, `test`. В сценариях редактора `sed` и программы `gawk` используются разные версии команды `print`, которая предназначена для вывода всех строк, сопоставленных с шаблоном регулярного выражения. В данном случае в текстовой строке, которая выводится с помощью инструкции `echo`, присутствует слово `"test"`, поэтому текст в потоке данных успешно сопоставляется с данным конкретным шаблоном регулярного выражения, и редактор `sed` отображает строку.

Второй шаблон также определяет только отдельное слово, на сей раз слово `"trial"`. Но текстовая строка в инструкции `echo` не содержит это слово, поэтому не происходит успешное сопоставление с шаблоном регулярного выражения, и вывод строки не происходит ни в редакторе `sed`, ни в программе `gawk`.

Читатель мог уже заметить, что для регулярного выражения такого типа не имеет значения, в каком месте в потоке данных обнаруживается шаблон. Не имеет также значения то, сколько раз встречается шаблон. При условии, что обработчику регулярных выражений удастся сопоставить шаблон с любой подстрокой текстовой строки, происходит передача строки в программу Linux, использующую данный обработчик.

Ключом к успешному использованию регулярных выражений является обеспечение сопоставления шаблонов регулярных выражений с текстом в потоке данных. Важно помнить, что успешное функционирование обработчика регулярных выражений при сопоставлении с шаблоном полностью зависит от того, правильно ли составлено регулярное выражение. Первое правило, о котором всегда следует помнить, состоит в том, что в шаблонах регулярных выражений учитывается регистр. Это означает, что сопоставление происходит успешно только с теми шаблонами, в которых регистр символов задан должным образом:

```
$ echo "This is a test" | sed -n '/this/p'
$
$ echo "This is a test" | sed -n '/This/p'
This is a test
$
```

Первая попытка сопоставления окончилась неудачей, поскольку слово `"this"` со всеми строчными буквами в данной текстовой строке отсутствует, а вторая попытка, в которой меняется шаблон с прописной буквой в нужном месте, оказалась вполне успешной.

Задавая шаблоны регулярных выражений, не обязательно использовать в тексте лишь целые слова. Если определенный в шаблоне текст обнаружится в любом месте в потоке данных, то произойдет сопоставление с регулярным выражением следующим образом:

```
$ echo "The books are expensive" | sed -n '/book/p'
The books are expensive
$
```

Несмотря на то что в тексте в потоке данных содержится подстрока `books`, учитывается то, что эта подстрока успешно сопоставляется с включенным в регулярное выражение текстом `book`, поэтому сопоставление шаблона регулярного выражения с данными рассматривается

как успешное. Безусловно, при попытке поменять местами эти две подстроки в потоке данных и в регулярном выражении сопоставление окончится неудачей:

```
$ echo "The book is expensive" | sed -n '/books/p'
$
```

Полный текст регулярного выражения не появляется в потоке данных, поэтому не удастся выполнить сопоставление и редактор `sed` не отображает текст.

Кроме того, при формировании регулярных выражений можно не ограничиваться применением отдельных слов текста. Предусмотрена возможность включать также в текстовые строки пробелы и цифры:

```
$ echo "This is line number 1" | sed -n '/ber 1/p'
This is line number 1
$
```

Пробелы рассматриваются в регулярном выражении наравне с любыми другими символами:

```
$ echo "This is line number1" | sed -n '/ber 1/p'
$
```

Но если в регулярном выражении определен пробел, то он должен появиться и в потоке данных. Могут быть созданы даже такие шаблоны регулярных выражений, которые сопоставляются с несколькими смежными пробелами:

```
$ cat data1
This is a normal line of text.
This is  a line with too many spaces.
$ sed -n '/ /p' data1
This is  a line with too many spaces.
$
```

В данном случае с шаблоном регулярного выражения сопоставляется строка с двумя пробелами между словами. Это — превосходный способ обнаружения нарушений форматирования в текстовых файлах, которые связаны с неправильным применением пробелов!

Специальные символы

При использовании текстовых строк в шаблонах регулярных выражений необходимо учитывать некоторые нюансы. Задавая текстовые символы в регулярном выражении, следует учитывать несколько исключений. Дело в том, что некоторые символы в шаблонах регулярных выражений трактуются особым образом. При попытке непосредственно использовать эти символы в текстовом шаблоне результат окажется далеким от ожидаемого.

Специальные символы, распознаваемые в регулярных выражениях, приведены ниже.

```
. * [ ] ^ $ { } \ + ? | ( )
```

В следующих разделах будет описано, как именно используются эти специальные символы в регулярных выражениях. А на данный момент достаточно запомнить, что эти символы нельзя использовать в непосредственном виде в текстовом шаблоне.

Если же один из специальных символов потребуется применить в качестве текстового символа, то его необходимо вначале *экранировать*. Экранирование специальных символов представляет собой добавление перед ними специального символа, который служит для обработчика регулярных выражений указанием, что следующий символ должен интерпретироваться

как обычный текстовый символ. Специальным символом, предназначенным для этой цели, является символ обратной косой черты (`\`).

Например, если в тексте требуется найти знак доллара, то в шаблоне, предназначенном для поиска, достаточно задать перед знаком доллара символ обратной косой черты:

```
$ cat data2
The cost is $4.00
$ sed -n '/\$/p' data2
The cost is $4.00
$
```

Сам символ обратной косой черты является специальным символом, поэтому, если возникает необходимость использовать его в шаблоне регулярного выражения, экранирование должно быть применено и к нему самому, что приводит к получению двойной обратной косой черты:

```
$ echo "\ is a special character" | sed -n '/\\/p'
\ is a special character
$
```

Наконец, отметим, что при попытке непосредственного использования косой черты в шаблоне регулярного выражения в редакторе `sed` или программе `gawk` также возникает ошибка, несмотря на то, что косая черта не относится к категории специальных символов для регулярного выражения:

```
$ echo "3 / 2" | sed -n '///p'
sed: -e expression #1, char 2: No previous regular expression
$
```

Чтобы включить символ косой черты в регулярное выражение, его также необходимо экранировать:

```
$ echo "3 / 2" | sed -n '/\\/p'
3 / 2
$
```

Теперь редактор `sed` получает возможным образом интерпретировать шаблон регулярного выражения и сопоставление оканчивается успешно.

Символы обозначения точек привязки

Как показано в разделе “Обычный текст”, по умолчанию при использовании шаблона регулярного выражения сопоставление происходит успешно, независимо от того, в каком месте потока данных обнаруживается шаблон. Но иногда возникает необходимость учесть расположение шаблона по отношению к началу или к концу строк в потоке данных, для чего используются два специальных символа.

Поиск с начала строки

Для определения шаблона, который сопоставляется с началом строки текста в потоке данных, служит знак вставки (`^`). Если шаблон обнаруживается в любом другом месте, кроме начала строки текста, то сопоставление с этим шаблоном регулярного выражения оканчивается неудачей.

Чтобы воспользоваться знаком вставки, необходимо поместить его перед шаблоном, заданным в регулярном выражении:

```
$ echo "The book store" | sed -n '/^book/p'
$
$ echo "Books are great" | sed -n '/^Book/p'
Books are great
$
```

Символ точки привязки в виде знака вставки применяется для проверки того, представлен ли данный шаблон в начале каждой новой строки данных, с учетом того, что переход на новую строку происходит вслед за обнаружением символа обозначения конца предыдущей строки:

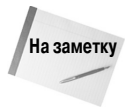
```
$ cat data3
This is a test line.
this is another test line.
A line that tests this feature.
Yet more testing of this
$ sed -n '/^this/p' data3
this is another test line.
$
```

При условии, что шаблон обнаруживается в начале новой строки, происходит фиксация его наличия с помощью точки привязки, заданной знаком вставки.

Если знак вставки представлен в любом другом месте, отличном от начала шаблона, то он рассматривается как обычный, а не специальный символ:

```
$ echo "This ^ is a test" | sed -n '/s ^/p'
This ^ is a test
$
```

В данном случае знак вставки приведен в шаблоне регулярного выражения не на первом месте, поэтому в редакторе sed этот символ используется для сопоставления как обычный символ.



Если требуется задать шаблон регулярного выражения, в котором знак вставки обозначает сам себя, то не требуется его экранировать с помощью обратной косой черты. Но если знак вставки, за которым следует дополнительный текст в шаблоне, должен быть показан на первом месте, то необходимо ввести экранирующий символ перед знаком вставки.

Поиск конца строки

Задачей, противоположной поиску шаблона в начале строки, является его поиск в конце строки. Конечная точка привязки определяется с помощью специального символа — знака доллара (\$). Этот специальный символ должен быть добавлен после текстового шаблона для указания на то, что этим текстовым шаблоном должна заканчиваться строка данных:

```
$ echo "This is a good book" | sed -n '/book$/p'
This is a good book
$ echo "This book is good" | sed -n '/book$/p'
$
```

С использованием конечного текстового шаблона связаны определенные сложности, заключающиеся в том, что при его определении необходимо тщательно проверять правильность задания искомой строки:

```
$ echo "There are a lot of good books" | sed -n '/book$/p'
$
```

В данном случае слово "book" задано в конце строки во множественном числе, а это означает, что больше не произойдет сопоставление с шаблоном регулярного выражения, несмотря на то, что само слово "book" присутствует в потоке данных. Текстовый шаблон обязательно должен стоять на последнем месте в строке, чтобы сопоставление с шаблоном было выполнено успешно.

Совместное применение разных точек привязки

В процессе обработки данных обнаруживается ряд обычных ситуаций, в которых может оказаться удобным объединение и начальной, и конечной точек привязки в одной и той же строке. Первая из этих ситуаций такова. Предположим, необходимо найти строку данных, в которой не содержится ничего, кроме какого-то конкретного текстового шаблона:

```
$ cat data4
this is a test of using both anchors
I said this is a test
this is a test
I'm sure this is a test.
$ sed -n '/^this is a test$/p' data4
this is a test
$
```

Редактор sed пропускает строки, которые включают другой текст в дополнение к указанному тексту.

Вторая ситуация на первый взгляд может показаться немного надуманной, но решение по выходу из нее является чрезвычайно полезным. Применяя в сочетании обе точки привязки в шаблоне, не содержащем текста, можно отфильтровать пустые строки из потока данных. Рассмотрим следующий пример:

```
$ cat data5
This is one test line.

This is another test line.
$ sed '/^$/d' data5
This is one test line.
This is another test line.
$
```

Шаблон регулярного выражения, который определен в данном случае, обеспечивает поиск строк, в которых нет ничего между началом и концом. Вообще говоря, пустыми являются те строки, в которых нет никакого текста между двумя символами обозначения конца строки, поэтому происходит их сопоставление с данным шаблоном регулярного выражения. В редакторе sed используется команда delete (сокращенно d) для удаления строк, сопоставленных с шаблоном регулярного выражения, таким образом, происходит удаление всех пустых строк из текста. Это — эффективный способ очистки документов от пустых строк.

Символ точки

Специальный символ точки предназначен для сопоставления с любым отдельным символом, кроме символа обозначения конца строки. Однако, чтобы произошло сопоставление с символом точки, в соответствующем месте текста должен присутствовать хотя бы какой-то символ; если нет такого символа, который мог бы быть сопоставлен с точкой, то применение шаблона оканчивается неудачей.

Рассмотрим несколько примеров использования символа точки в шаблоне регулярного выражения:

```
$ cat data6
This is a test of a line.
The cat is sleeping.
That is a very nice hat.
This test is at line four.
at ten o'clock we'll go home.
$ sed -n '/.at/p' data6
The cat is sleeping.
That is a very nice hat.
This test is at line four.
$
```

Читатель должен сам попытаться определить, почему обработка первой строки окончилась неудачей, а вторая и третья строки прошли сопоставление успешно. Что касается четвертой строки, то ситуация немного сложнее. Заметим, что подстрока `at` должна быть сопоставлена успешно, но перед ней отсутствует какой-либо символ, который можно было бы сопоставить с символом точки. Да, но есть нечто другое! В регулярных выражениях пробелы рассматриваются как символы, поэтому благодаря наличию пробела перед подстрокой `at` сопоставление с шаблоном происходит успешно. Это утверждение можно проверить по пятой строке, в которой подстрока `at` находится в самом начале, поэтому сопоставление с шаблоном оканчивается неудачей.

Классы символов

Специальный символ точки превосходно подходит для сопоставления символьной позиции с любым символом, но иногда приходится ограничивать состав символов, с которыми должно быть выполнено сопоставление. Для этого в регулярных выражениях применяются так называемые *классы символов*.

Предусмотрены широкие возможности определения классов символов, предназначенных для сопоставления с определенной позицией в текстовом шаблоне. Если один из символов, принадлежащих к классу символов, обнаруживается в потоке данных, то происходит его успешное сопоставление с шаблоном.

Для определения класса символов используются квадратные скобки, в которых должны быть заданы все символы, предназначенные для включения в класс. Затем весь класс символов в шаблоне может быть задан полностью аналогично тому, как применяется любой другой подстановочный знак. На первых порах бывает сложно привыкнуть к использованию классов символов, но после их освоения можно добиться весьма полезных результатов.

Ниже приведен пример создания класса символов.

```
$ sed -n '/[ch]at/p' data6
The cat is sleeping.
That is a very nice hat.
$
```

При использовании того же файла данных, что и в примере со специальным символом точки, был получен совсем другой результат. На сей раз мы сумели отфильтровать строку, которая содержала только слово `at`. С этим шаблоном были сопоставлены только строки, в которых содержится слово `cat` или `hat`. Следует также отметить, что не произошло и сопоставление

строки, которая начинается с подстроки `at`. Это связано с тем, что в строке должен присутствовать символ из класса символов, который сопоставляется с соответствующей позицией.

Классы символов становятся удобными и в тех случаях, если заранее не известно, в каком регистре будет представлен некоторый буквенный символ:

```
$ echo "Yes" | sed -n '/[Yy]es/p'
Yes
$ echo "yes" | sed -n '/[Yy]es/p'
yes
$
```

Предусмотрена возможность использовать в одном выражении несколько классов символов:

```
$ echo "Yes" | sed -n '/[Yy][Ee][Ss]/p'
Yes
$ echo "yEs" | sed -n '/[Yy][Ee][Ss]/p'
yEs
$ echo "yeS" | sed -n '/[Yy][Ee][Ss]/p'
yeS
$
```

В этом регулярном выражении применяются три класса символов, соответствующие прописным и строчным буквам данного слова во всех трех символьных позициях.

Классы символов не обязательно должны содержать только буквенные символы; в них могут быть также представлены цифры:

```
$ cat data7
This line doesn't contain a number.
This line has 1 number on it.
This line a number 2 on it.
This line has a number 4 on it.
$ sed -n '/[0123]/p' data7
This line has 1 number on it.
This line a number 2 on it.
$
```

Данный шаблон регулярного выражения сопоставляется с любыми строками, которые содержат цифры 0, 1, 2 или 3. Строки с любыми другими цифрами, а также строки вообще без цифр пропускаются.

Классы символов можно объединять для проверки того, отформатированы ли должным образом такие строки с цифрами, как номера телефонов и почтовые индексы. Но при попытке обеспечить сопоставление с конкретным форматом необходимо соблюдать осторожность. Ниже приведен пример того, как может возникнуть ошибка при сопоставлении с почтовым индексом.

```
$ cat data8
60633
46201
223001
4353
22203
$ sed -n '
>/[0123456789][0123456789][0123456789][0123456789][0123456789]/p
>' data8
60633
```



```
46201
223001
22203
$
```

По-видимому, этот результат далек от ожидаемого. Регулярное выражение успешно выполняет задачу отфильтровывания чисел, количество цифр в которых слишком мало по сравнению с форматом почтового индекса, поскольку при обработке коротких чисел не происходит сопоставление с последними классами символов. Однако регулярное выражение не позволяет исключить шестизначное число, несмотря на то, что в нем определено только пять классов символов.

Напомним, что для успешного сопоставления с шаблоном регулярного выражения достаточно того, чтобы этот шаблон был обнаружен в любом месте строки текста в потоке данных. Иначе говоря, приведенный выше шаблон не исключает возможность наличия дополнительных символов, кроме тех, что заданы в шаблоне сопоставления. Чтобы обеспечить успешное сопоставление только с теми числами, которые состоят исключительно из пяти цифр, необходимо выделить такие числа тем или иным образом из потока данных, например, с помощью пробелов или, как в данном случае, разместить в отдельной строке и применить привязки к началу и концу строки:

```
$ sed -n '
> /^[0123456789][0123456789][0123456789][0123456789][0123456789]$/p
> ' data8
60633
46201
22203
$
```

Теперь полученный результат много лучше! Ниже в главе будет показано, как еще больше упростить это решение.

Классы символов весьма широко используются для выполнения еще одной задачи — интерпретации слов, которые могут быть введены с орфографическими ошибками, наподобие данных, введенных пользователем в форме. Можно легко создать регулярные выражения, позволяющие успешно справляться с такими ситуациями, когда в данных могут иметь место распространенные орфографические ошибки:

```
$ cat data9
I need to have some maintenance done on my car.
I'll pay that in a seperate invoice.
After I pay for the maintenance my car will be as good as new.
$ sed -n '
/maint[ea]n[ae]nce/p
/sep[ea]r[ea]te/p
' data9
I need to have some maintenance done on my car.
I'll pay that in a seperate invoice.
After I pay for the maintenance my car will be as good as new.
$
```

В двух командах `print` редактора `sed` в данном примере используются классы символов регулярного выражения, чтобы можно было распознать в тексте слова `maintenance` и `separate` с обычными орфографическими ошибками. Тот же шаблон регулярного выражения успешно сопоставляется и с правильно написанным словом `"maintenance"`.

Обращение классов символов

В шаблонах регулярного выражения можно также обеспечить сопоставление с символами, не входящими в классы символов. В таком случае вместо поиска символа, содержащегося в классе, происходит поиск символа, который не относится к классу. Для этого достаточно поместить знак вставки перед определением диапазона в классе символов:

```
$ sed -n '/[^ch]at/p' data6
This test is at line two.
$
```

Такая операция называется обращением класса символов. В данном случае шаблон регулярного выражения, в котором также задана текстовая строка, сопоставляется с любым символом, отличным от `c` и от `h`. К категории символов, отличных от указанных, относится также пробел, поэтому и этот символ обеспечивает успешное сопоставление с шаблоном. Но даже после обращения класс символов должен сопоставляться с конкретным символом, поэтому проверка строки, начинающейся с подстроки `at`, также оканчивается неудачей.

Использование диапазонов

Читатель мог заметить, что приведенный выше пример проверки кода почтового индекса — довольно громоздкий, поскольку в нем для каждой позиции перечисляются все возможные цифры в каждом классе символов. К счастью, предусмотрена возможность воспользоваться определенным сокращением и обойтись без таких перечислений.

В классе символов можно задавать диапазоны символов с помощью дефиса. Достаточно указать первый символ в диапазоне (дефис), а затем последний символ. Регулярное выражение включит все символы, находящиеся в пределах указанного диапазона, с учетом кодировки символов, применяемой в системе Linux (см. главу 2).

Теперь можем упростить пример с почтовым индексом, указывая диапазоны цифр:

```
$ sed -n '/^[0-9][0-9][0-9][0-9][0-9]$/p' data8
60633
46201
45902
$
```

Очевидно, что при этом объем кода значительно сокращается! Каждый из этих классов символов сопоставляется с любой цифрой от 0 до 9. Сопоставление с шаблоном окончится неудачей, если в любом месте в данных будет присутствовать буква:

```
$ echo "a8392" | sed -n '/^[0-9][0-9][0-9][0-9][0-9]$/p'
$
$ echo "1839a" | sed -n '/^[0-9][0-9][0-9][0-9][0-9]$/p'
$
$ echo "18a92" | sed -n '/^[0-9][0-9][0-9][0-9][0-9]$/p'
$
```

Тот же метод применим и по отношению к буквенным символам:

```
$ sed -n '/[c-h]at/p' data6
The cat is sleeping.
That is a very nice hat.
$
```

Новый шаблон `[c-h]at` сопоставляется со словами, первой буквой которых может быть любая буква от `c` до `h`. В данном случае не удастся сопоставить с шаблоном только строку, в которой подстрока `at` находится в самом начале.

В одном классе символов можно также задать несколько несмежных диапазонов:

```
$ sed -n '/[a-ch-m]at/p' data6
The cat is sleeping.
That is a very nice hat.
$
```

Данный класс символов допускает появление любой буквы из диапазонов от `a` до `c` и от `h` до `m` перед подстрокой `at`. Этот диапазон отвергает любые символы от `d` до `g`:

```
$ echo "I'm getting too fat." | sed -n '/[a-ch-m]at/p'
$
```

В данном случае окончилась неудачей проверка с помощью шаблона строки со словом `fat`, поскольку первая буква слова не находится в указанном диапазоне.

Специальные классы символов

Спецификация BRE не только позволяет определять собственные классы символов, но и содержит специальные классы символов, которые могут использоваться для сопоставления с конкретными типами символов. В табл. 19.1 описаны специальные символы BRE, которые могут использоваться в регулярных выражениях.

Таблица 19.1. Специальные классы символов BRE

Класс	Описание
<code>[:alpha:]</code>	Обеспечивает сопоставление с любым буквенным символом, прописным или строчным
<code>[:alnum:]</code>	Сопоставляется с любым алфавитно-цифровым символом: 0–9, A–Z или a–z
<code>[:blank:]</code>	Сопоставляется с пробелом или знаком табуляции
<code>[:digit:]</code>	Сопоставляется с цифрой от 0 до 9
<code>[:lower:]</code>	Сопоставляется с любым строчным алфавитным символом от <code>a</code> до <code>z</code>
<code>[:print:]</code>	Сопоставляется с любым печатаемым символом
<code>[:punct:]</code>	Сопоставляется со знаком пунктуации
<code>[:space:]</code>	Сопоставляется с любым пробельным символом: пробела, табуляции, перевода на новую строку (NL), прогона формы (FF), вертикальной табуляции (VT), возврата каретки (CR)
<code>[:upper:]</code>	Сопоставляется с любым прописным алфавитным символом от <code>A</code> до <code>Z</code>

Специальные классы символов можно использоваться в шаблонах регулярных выражений так же, как обычные классы символов:

```
$ echo "abc" | sed -n '/[:digit:]/p'
$
$ echo "abc" | sed -n '/[:alpha:]/p'
abc
$ echo "abc123" | sed -n '/[:digit:]/p'
abc123
$ echo "This is, a test" | sed -n '/[:punct:]/p'
This is, a test
```

```
$ echo "This is a test" | sed -n '/[[:punct:]]/p'
$
```

Специальные классы символов позволяют проще определять диапазоны. Вместо того чтобы использовать диапазон [0–9], можно просто указать [[:digit:]].

Звездочка

Чтобы указать, что некоторый символ должен появиться в тексте нуль или большее количество раз для сопоставления с шаблоном, можно поместить после символа звездочку:

```
$ echo "ik" | sed -n '/ie*k/p'
ik
$ echo "iek" | sed -n '/ie*k/p'
iek
$ echo "ieek" | sed -n '/ie*k/p'
ieek
$ echo "ieeek" | sed -n '/ie*k/p'
ieeek
$ echo "ieeeeek" | sed -n '/ie*k/p'
ieeeeek
$
```

Этот символ шаблона широко используется для обработки слов, в которых встречаются распространенные орфографические ошибки или которые имеют разные варианты написания в различных диалектах языка. Например, если необходимо написать сценарий, предназначенный для использования и в американском, и в британском диалекте английского языка, то можно применить следующую конструкцию:

```
$ echo "I'm getting a color TV" | sed -n '/colou*r/p'
I'm getting a color TV
$ echo "I'm getting a colour TV" | sed -n '/colou*r/p'
I'm getting a colour TV
$
```

Подстрока `u*` в шаблоне указывает, что буква `u` может либо присутствовать, либо отсутствовать в тексте, сопоставляемом с шаблоном. Аналогичным образом, если известно, что некоторое слово часто бывает написано с орфографическими ошибками, то к этому можно заранее подготовиться с помощью шаблона со звездочками:

```
$ echo "I ate a potatoe with my lunch." | sed -n '/potatoe*/p'
I ate a potatoe with my lunch.
$ echo "I ate a potato with my lunch." | sed -n '/potatoe*/p'
I ate a potato with my lunch.
$
```

Размещение звездочки вслед за возможной лишней буквой позволит успешно распознавать слова с орфографической ошибкой.

Еще одна удобная возможность состоит в совместном применении специального символа точки со специальным символом звездочки. С помощью такой комбинации можно составить шаблон, сопоставимый с любым количеством любых символов. Необходимость в этом часто возникает, если в потоке данных две текстовые строки могут стоять рядом или быть отделены друг от друга:

```
$ echo "this is a regular pattern expression" | sed -n '  
> /regular.*expression/p'  
this is a regular pattern expression  
$
```

С использованием этого шаблона можно легко обеспечить поиск нескольких слов, которые могут появляться в любом месте строки текста в потоке данных.

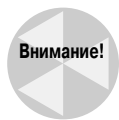
Звездочку можно применять не только к отдельному символу, но и к классу символов. Это позволяет задавать группы или диапазоны символов, которые могут неоднократно появляться в тексте:

```
$ echo "bt" | sed -n '/b[ae]*t/p'  
bt  
$ echo "bat" | sed -n '/b[ae]*t/p'  
bat  
$ echo "bet" | sed -n '/b[ae]*t/p'  
bet  
$ echo "btt" | sed -n '/b[ae]*t/p'  
btt  
$  
$ echo "baat" | sed -n '/b[ae]*t/p'  
baat  
$ echo "baaeet" | sed -n '/b[ae]*t/p'  
baaeet  
$ echo "baeeaeet" | sed -n '/b[ae]*t/p'  
baeeaeet  
$ echo "baakeet" | sed -n '/b[ae]*t/p'  
$
```

Сопоставление с данным шаблоном осуществляется успешно при условии, что символы `a` и `e` появляются в любых сочетаниях между символами `b` и `t` (включая тот случай, когда они вообще не появляются). При обнаружении любого другого символа, не относящегося к этому заданному классу символов, сопоставление с шаблоном оканчивается неудачей.

Расширенные регулярные выражения

Шаблоны POSIX ERE включают несколько дополнительных символов, которые используются в определенных приложениях и программах Linux. Программа `gawk` распознает шаблоны ERE, но редактор `sed` этого не обеспечивает.



Важно помнить, что в редакторе `sed` и программе `gawk` применяются разные обработчики регулярных выражений. В программе `gawk` допускается использование большинства символов шаблонов расширенных регулярных выражений, поэтому эта программа предоставляет определенные дополнительные возможности фильтрации, которые не предусмотрены в редакторе `sed`. Но это влечет за собой то, что программа `gawk` часто функционирует медленнее при обработке потоков данных.

В настоящем разделе описаны наиболее широко используемые символы шаблонов ERE, которые могут быть включены в конкретные сценарии программы `gawk`.

Вопросительный знак

Вопросительный знак по своему назначению аналогичен звездочке, но имеет небольшое отличие. Вопросительный знак указывает, что предыдущий символ может повторяться нуль или один раз, но не больше. С помощью вопросительного знака нельзя обеспечить сопоставление с большим количеством вхождений символа:

```
$ echo "bt" | gawk '/be?t/{print $0}'
bt
$ echo "bet" | gawk '/be?t/{print $0}'
bet
$ echo "beet" | gawk '/be?t/{print $0}'
$
$ echo "beeet" | gawk '/be?t/{print $0}'
$
```

Если символ `e` отсутствует в тексте или появляется в нем только один раз, сопоставление с шаблоном оканчивается успешно.

Так же, как и звездочку, вопросительный знак можно задавать и применительно к классу символов:

```
$ echo "bt" | gawk '/b[ae]?t/{print $0}'
bt
$ echo "bat" | gawk '/b[ae]?t/{print $0}'
bat
$ echo "bot" | gawk '/b[ae]?t/{print $0}'
$
$ echo "bet" | gawk '/b[ae]?t/{print $0}'
bet
$ echo "baet" | gawk '/b[ae]?t/{print $0}'
$
$ echo "beat" | gawk '/b[ae]?t/{print $0}'
$
$ echo "beet" | gawk '/b[ae]?t/{print $0}'
$
```

При обнаружении символа из класса символов в количестве нуль или один сопоставление с шаблоном оканчивается успешно. Но если появляются подряд два символа из класса символов или один символ обнаруживается дважды, то сопоставление с шаблоном оканчивается неудачей.

Знак “плюс”

Знак “плюс” — еще один символ шаблона, аналогичный звездочке, но также действует иначе по сравнению с вопросительным знаком (хотя и не так, как звездочка). Знак “плюс” указывает, что предшествующий ему символ может появляться один или несколько раз, но должен присутствовать в тексте не меньше, чем в одном экземпляре. Сопоставление с этим шаблоном не происходит, если указанный символ отсутствует в тексте:

```
$ echo "beeet" | gawk '/be+t/{print $0}'
beeet
$ echo "beet" | gawk '/be+t/{print $0}'
beet
```

```
$ echo "bet" | gawk '/be+t/{print $0}'
bet
$ echo "bt" | gawk '/be+t/{print $0}'
$
```

Если символ `e` не обнаруживается в тексте, то сопоставление с шаблоном оканчивается неудачей. Знак “плюс”, как и другие знаки повтора, может применяться с классами символов, по такому же принципу, как звездочка и вопросительный знак:

```
$ echo "bt" | gawk '/b[ae]+t/{print $0}'
$
$ echo "bat" | gawk '/b[ae]+t/{print $0}'
bat
$ echo "bet" | gawk '/b[ae]+t/{print $0}'
bet
$ echo "beat" | gawk '/b[ae]+t/{print $0}'
beat
$ echo "beet" | gawk '/b[ae]+t/{print $0}'
beet
$ echo "beeat" | gawk '/b[ae]+t/{print $0}'
beeat
$
```

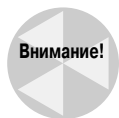
На сей раз текст сопоставляется с указанным шаблоном, если обнаруживается любой символ, определенный в классе символов.

Использование фигурных скобок

В спецификации ERE предусмотрено использование фигурных скобок, которые позволяют определять пределы повторения предшествующих символов в регулярном выражении. Такие пределы повторения часто именуются *интервалами*. Интервал может быть представлен в одном из двух указанных ниже форматов.

- m . Регулярное выражение появляется точно m раз.
- m, n . Регулярное выражение появляется по меньшей мере m раз, но не больше чем n раз.

Это средство позволяет выполнять точную настройку того, сколько раз допускается появление символа (или класса символов) в шаблоне.



По умолчанию программа `gawk` не распознает интервалы в регулярных выражениях. Чтобы обеспечить распознавание программой `gawk` интервалов регулярных выражений, необходимо задать при ее вызове опцию командной строки `--re-interval`.

Ниже приведен пример использования простого интервала с единичным значением.

```
$ echo "bt" | gawk --re-interval '/be{1}t/{print $0}'
$
$ echo "bet" | gawk --re-interval '/be{1}t/{print $0}'
bet
$ echo "beet" | gawk --re-interval '/be{1}t/{print $0}'
$
```

Задавая интервал, равный единице, можно точно указать, в каком количестве символ может присутствовать в строке, чтобы обеспечить ее сопоставление с шаблоном. Если символ появляется больше одного раза, то сопоставление с шаблоном оканчивается неудачей.

Кроме того, часто возникают ситуации, в которых требуется определить и нижний, и верхний пределы:

```
$ echo "bt" | gawk --re-interval '/be{1,2}t/{print $0}'
$
$ echo "bet" | gawk --re-interval '/be{1,2}t/{print $0}'
bet
$ echo "beet" | gawk --re-interval '/be{1,2}t/{print $0}'
beet
$ echo "beeet" | gawk --re-interval '/be{1,2}t/{print $0}'
$
```

В данном примере символ `e` может появляться один или два раза, чтобы сопоставление с шаблоном оказалось успешным; в противном случае сопоставление оканчивается неудачей.

Шаблоны сопоставления с интервалами могут также применяться к классам символов:

```
$ echo "bt" | gawk --re-interval '/b[ae]{1,2}t/{print $0}'
$
$ echo "bat" | gawk --re-interval '/b[ae]{1,2}t/{print $0}'
bat
$ echo "bet" | gawk --re-interval '/b[ae]{1,2}t/{print $0}'
bet
$ echo "beat" | gawk --re-interval '/b[ae]{1,2}t/{print $0}'
beat
$ echo "beet" | gawk --re-interval '/b[ae]{1,2}t/{print $0}'
beet
$ echo "beeat" | gawk --re-interval '/b[ae]{1,2}t/{print $0}'
$
$ echo "baeet" | gawk --re-interval '/b[ae]{1,2}t/{print $0}'
$
$ echo "baeaet" | gawk --re-interval '/b[ae]{1,2}t/{print $0}'
$
```

Сопоставление этого шаблона регулярного выражения происходит успешно, если в текстовой строке имеются точно одно или два вхождения буквы `a` или `e`, но оканчивается неудачей, если присутствует любая другая комбинация этих букв.

Символ канала

Символ канала позволяет задавать два или несколько шаблонов, применяемых обработчиком регулярных выражений в соответствии с логической операцией `OR` при исследовании потока данных. Если с текстом в потоке данных сопоставляется любой из шаблонов, соединенных символом канала, то текст успешно проходит проверку. Если же не удастся согласовать ни один из шаблонов, то проверка текста в потоке данных по регулярному выражению оканчивается неудачей.

Регулярное выражение с символом канала имеет следующий формат:

```
expr1|expr2|...
```


Ниже приведен пример применения такого регулярного выражения.

```
$ echo "The cat is asleep" | gawk '/cat|dog/{print $0}'
The cat is asleep
$ echo "The dog is asleep" | gawk '/cat|dog/{print $0}'
The dog is asleep
$ echo "The sheep is asleep" | gawk '/cat|dog/{print $0}'
$
```

В этом примере осуществляется проверка наличия подстроки `cat` или `dog` в потоке данных. При использовании символа канала в регулярных выражениях нельзя вводить какие-либо пробелы, поскольку при обнаружении пробелов происходит их добавление к шаблону регулярного выражения.

Регулярные выражения по обе стороны от символа канала могут использоваться в любом шаблоне регулярного выражения, включая классы символов, для определения текста:

```
$ echo "He has a hat." | gawk '/[ch]at|dog/{print $0}'
He has a hat.
$
```

В этом примере обеспечивается сопоставление с данными в потоке текстовых данных любой подстроки — `cat`, `hat` или `dog`.

Выражения группирования

Шаблоны регулярных выражений можно также группировать с использованием круглых скобок. После группирования шаблона регулярного выражения вся полученная группа рассматривается как один обычный символ. К группе можно применять специальные символы по такому же принципу, как и к отдельным символам. Например:

```
$ echo "Sat" | gawk '/Sat(urday)?/{print $0}'
Sat
$ echo "Saturday" | gawk '/Sat(urday)?/{print $0}'
Saturday
$
```

Формирование группы из окончания `"urday"` и применения вопросительного знака позволяет обеспечить сопоставление с шаблоном полного названия дня недели `Saturday` и сокращенного названия `Sat`.

Широко принято использовать группирование наряду с символом канала для создания групп возможных сопоставлений с шаблоном:

```
$ echo "cat" | gawk '/(c|b)a(b|t)/{print $0}'
cat
$ echo "cab" | gawk '/(c|b)a(b|t)/{print $0}'
cab
$ echo "bat" | gawk '/(c|b)a(b|t)/{print $0}'
bat
$ echo "bab" | gawk '/(c|b)a(b|t)/{print $0}'
bab
$ echo "tab" | gawk '/(c|b)a(b|t)/{print $0}'
$
$ echo "tac" | gawk '/(c|b)a(b|t)/{print $0}'
$
```

Шаблон (c|b)a(b|t) сопоставляется с любым вариантом выбора символов из первой группы наряду с любым вариантом выбора из второй группы.

Регулярные выражения в действии

Выше описаны правила использования шаблонов регулярных выражений и приведено несколько простых примеров, а теперь мы можем применить полученные знания на деле. В следующих разделах рассматриваются некоторые более практически значимые примеры использования регулярных выражений в сценариях командного интерпретатора.

Подсчет количества файлов в каталоге

Для начала рассмотрим сценарий командного интерпретатора, с помощью которого можно подсчитать количество исполняемых файлов, присутствующих в каталогах, которые определены в переменной среды PATH. Для этого необходимо провести синтаксический анализ переменной PATH для получения отдельных имен каталогов. В главе 5 было показано, как отобразить содержимое переменной среды PATH:

```
$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/usr/games:/usr/java/
j2sdk1.4.1_01/bin
$
```

В разных системах Linux переменная среды PATH имеет различные значения, в зависимости от того, какие приложения развернуты в системе. Ключом к решению задачи распознавания отдельных каталогов в переменной PATH является то, что каталоги разделены двоеточиями. Чтобы получить перечень каталогов, которые могут использоваться в сценарии, достаточно заменить каждое двоеточие пробелом, поскольку в самих именах каталогов пробелы отсутствуют. Читатель уже знает, что в редакторе sed эту задачу можно решить с использованием простого регулярного выражения:

```
$ echo $PATH | sed 's:/ /g'
/usr/local/bin /bin /usr/bin /usr/X11R6/bin /usr/games /usr/java/
j2sdk1.4.1_01/bin
$
```

А после того как каталоги будут разделены, можно применить содержащую их строку стандартной инструкции for (см. главу 12) для формирования цикла с итерацией по каждому каталогу:

```
mypath='echo $PATH | sed 's:/ /g'`
for directory in $mypath
do
...
done
```

После получения сведений о каждом каталоге можно воспользоваться командой ls для получения листингов всех файлов во всех каталогах и ввести в действие еще одну инструкцию for, чтобы провести итерацию по каждому файлу, наращивая при этом счетчик.

Окончательная версия сценария выглядит следующим образом:

```
$ cat countfiles
#!/bin/bash
# подсчет количества файлов в пути PATH
```

```

mypath='echo $PATH | sed 's:/ /g'`
count=0
for directory in $mypath
do
    check='ls $directory`
    for item in $check
    do
        count=$(( $count + 1 ))
    done
    echo "$directory - $count"
    count=0
done
$ ./countfiles
/usr/local/bin - 79
/bin - 86
/usr/bin - 1502
/usr/X11R6/bin - 175
/usr/games - 2
/usr/java/j2sdk1.4.1_01/bin - 27
$

```

Очевидно, что теперь становится ясно, какими возможностями обладают регулярные выражения!

Проверка правильности номера телефона

В предыдущем примере было показано, как обеспечить совместное применение простого регулярного выражения и редактора `sed` для замены символов в потоке данных в целях последующей обработки данных. Регулярные выражения часто используются также для проверки данных и предоставления гарантий того, что данные заданы в надлежащем формате, приемлемом для некоторого сценария.

Одним из широко используемых приложений проверки данных является определение правильности формата номеров телефонов. Задание номеров телефонов предусмотрено во многих формах ввода данных, но пользователи часто ошибаются, когда первый раз пытаются ввести правильно отформатированный номер телефона. В Соединенных Штатах предусмотрено несколько общепринятых способов представления номера телефона:

```

(123) 456-7890
(123) 456.7890
123-456-7890
123.456.7890

```

Это означает, что пользователи могут вводить номера телефонов в форме одним из четырех способов. Регулярное выражение должно быть достаточно надежным, чтобы справляться с любой из этих ситуаций.

При формировании регулярного выражения лучше всего начинать слева, а затем постепенно наращивать шаблон для обеспечения сопоставления со всеми возможными символами, которые могут встретиться в дальнейшем. В данном примере начнем с того, что в номере телефона может присутствовать или отсутствовать левая круглая скобка. Для сопоставления с этим символом можно применить следующий шаблон:

```
^\(?
```

Знак вставки используется для указания на то, что данные с номером телефона начинаются с новой строки. Но левая круглая скобка — специальный символ, поэтому его необходимо экранировать, чтобы иметь возможность использовать как обычный символ. Вопросительный знак указывает на то, что левая круглая скобка может присутствовать или отсутствовать в данных, но при этом сопоставление должно происходить успешно.

За этим следует трехзначный код города. В Соединенных Штатах коды городов начинаются с цифры 2 (ни один код города не начинается с цифр 0 или 1), т.е. первой цифрой кода может быть любая цифра от 2 до 9. Для сопоставления с кодом города необходимо использовать следующий шаблон:

```
[2-9] [0-9] {2}
```

Этот шаблон требует, чтобы первым символом кода была цифра от 2 до 9 включительно, а за ней могут следовать любые две цифры. После кода города может быть задана или не задана закрывающая, правая круглая скобка:

```
\) ?
```

За кодом города может присутствовать или отсутствовать пробел либо находиться дефис или точка. Выражения, задающие эти условия, можно сгруппировать с использованием групп символов, соединенных символами канала:

```
(| |-\|.)
```

Самый первый символ канала появляется сразу после левой круглой скобки и применяется для сопоставления при том условии, что пробел отсутствует. Для точки необходимо использовать экранирующий символ; в противном случае она будет интерпретироваться как сопоставляемая с любым символом.

Далее следует трехзначный дополнительный номер телефона. В этом случае не требуется учитывать какие-то особые условия:

```
[0-9] {3}
```

После дополнительного номера телефона необходимо обеспечить сопоставление с пробелом, дефисом или точкой (на сей раз можно не беспокоиться в связи необходимостью сопоставления при отсутствии пробела, поскольку должен быть задан по крайней мере пробел между дополнительным номером телефона и остальной частью номера):

```
( | |-\|.)
```

Затем, чтобы завершить формирование регулярного выражения, необходимо обеспечить сопоставление четырехзначного локального дополнительного номера телефона в конце строки:

```
[0-9] {4}$
```

Соединив указанные части в общий шаблон, получим следующее:

```
^(? [2-9] [0-9] {2}\) ? (| |-\|. ) [0-9] {3} ( | |-\|. ) [0-9] {4}$
```

Этот шаблон регулярного выражения можно использовать в программе `gawk` для исключения неправильно заданных номеров телефонов. Осталось только ознакомиться с тем, как создать простой сценарий в программе `gawk` с применением регулярного выражения и обработать список телефонов с помощью сценария. Напомним, что при использовании интервалов регулярного выражения в программе `gawk` необходимо задавать опцию командной строки `--re-interval`, так как в противном случае будут получены не правильные результаты.

Указанный сценарий приведен ниже.

```
$ cat isphone
#!/bin/bash
```

```
# сценарий отфильтровывания неправильных номеров телефонов
gawk --re-interval '/^\([? [2-9] [0-9] {2}\)\? (| |-|\.)
[0-9] {3} ( |-|\.) [0-9] {4} /{print $0}'
$
```

Данный листинг не позволяет убедиться в этой особенности, но команда `gawk` приведена на одной строке в сценарии командного интерпретатора. После этого появляется возможность перенаправлять номера телефонов в сценарий для обработки:

```
$ echo "317-555-1234" | ./isphone
317-555-1234
$ echo "000-555-1234" | ./isphone
$
```

Еще один вариант состоит в том, что можно перенаправить целый файл со списком номеров телефонов, чтобы отфильтровать неправильные номера:

```
$ cat phonelist
000-000-0000
123-456-7890
212-555-1234
(317) 555-1234
(202) 555-9876
33523
1234567890
234.123.4567
$ cat phonelist | ./isphone
212-555-1234
(317) 555-1234
(202) 555-9876
234.123.4567
$
```

В выводе будут присутствовать только допустимые номера телефонов, которые успешно сопоставлены с шаблоном регулярного выражения.

Синтаксический анализ адреса электронной почты

В наши дни информационный обмен по электронной почте с применением адресов электронной почты стал одной из наиболее важных форм связи. Но разработчики сценариев, в которых должна осуществляться проверка адресов электронной почты, сталкиваются со значительными сложностями, поскольку предусмотрено много разных способов формирования такого адреса. Адрес электронной почты имеет следующую основную форму:

`username@hostname`

В значении `username` может использоваться любой алфавитно-цифровой символ наряду с некоторыми специальными символами:

- точка;
- дефис;
- знак “плюс”;
- знак подчеркивания.

Идентификатор пользователя электронной почты, в котором эти символы появляются в любой комбинации, остается допустимым. Часть *hostname* адреса электронной почты состоит из одного или нескольких имен доменов и имени сервера. Имя сервера и имена доменов также должны следовать строгим правилам именования, поскольку в них разрешается использовать только алфавитно-цифровые символы, а также некоторые специальные символы:

- точка;
- знак подчеркивания.

Все элементы адреса, представляющие собой имя сервера и имена доменов, должны быть разделены точками, причем имя сервера присутствует в самом начале, за ним следуют имена поддоменов, а в конечном итоге появляется имя домена верхнего уровня без заключительной точки.

В свое время количество имен доменов верхнего уровня было довольно ограниченным, и разработчики шаблонов регулярных выражений стремились задать все эти имена в шаблонах для проверки допустимости. К сожалению, по мере развития Интернета произошло также увеличение количества применимых доменов верхнего уровня. Поэтому указанный выше метод проверки больше не является осуществимым.

Начнем формирование шаблона регулярного выражения с левой части. Известно, что ряд символов является допустимым в имени пользователя. Таким образом, проверка имени пользователя должна быть довольно несложной:

```
^[a-zA-Z0-9_\-\.\\+])@
```

В этом группировании показаны допустимые символы в имени пользователя, а знак “плюс” указывает, что должен быть представлен по крайней мере один символ. Очевидно, что следующим символом должен быть символ @, поэтому создание данной части шаблона не вызывает затруднений.

В шаблоне имени хоста используется такой же способ сопоставления с именем сервера и именами поддоменов:

```
([a-zA-Z0-9_\-\.\\+])
```

Данный шаблон сопоставляется с текстом:

```
server
server.subdomain
server.subdomain.subdomain
```

Что касается доменов верхнего уровня, то их имена подчиняются специальным правилам. В доменах верхнего уровня могут быть заданы только алфавитные символы, а их длина должна быть не меньше двух символов (которые используются в кодах страны) и не больше пяти. Ниже приведен шаблон регулярного выражения для домена верхнего уровня.

```
\\.([a-zA-Z]{2,5})$
```

Соединение всех частей шаблона приводит к получению следующего результата:

```
^[a-zA-Z0-9_\-\.\\+])@([a-zA-Z0-9_\-\.\\+])\\.([a-zA-Z]{2,5})$
```

Этот шаблон позволяет исключить неправильно отформатированные адреса электронной почты из списка адресов. Теперь мы можем приступить к созданию сценария для реализации регулярного выражения:

```
$ echo "rich@here.now" | ./isemail
rich@here.now
$ echo "rich@here.now." | ./isemail
```

```
$  
$ echo "rich@here.n" | ./isemail  
$  
$ echo "rich@here-now" | ./isemail  
$  
$ echo "rich.blum@here.now" | ./isemail  
rich.blum@here.now  
$ echo "rich_blum@here.now" | ./isemail  
rich_blum@here.now  
$ echo "rich/blum@here.now" | ./isemail  
$  
$ echo "rich#blum@here.now" | ./isemail  
$  
$ echo "rich*blum@here.now" | ./isemail  
$
```

Резюме

Чтобы успешно решать задачу манипулирования данными в файлах данных с помощью сценариев командного интерпретатора, необходимо освоить работу с регулярными выражениями. Регулярные выражения вводятся в действие в программах Linux, языках программирования и приложениях с помощью обработчиков регулярных выражений. В мире Linux предусмотрено несколько разных обработчиков регулярных выражений. Двумя наиболее широко применяемыми из них являются обработчик базовых регулярных выражений (Basic Regular Expression — BRE) POSIX и обработчик расширенных регулярных выражений (Extended Regular Expression — ERE) POSIX. Редактор `sed` соответствует главным образом требованиям обработчика BRE, а в программе `gawk` используется большинство средств, предусмотренных в обработчике ERE.

Регулярное выражение определяет образец шаблона, применяемого для фильтрации текста в потоке данных. Шаблон состоит из комбинации стандартных текстовых символов и специальных символов. Специальные символы используются обработчиком регулярных выражений для сопоставления с последовательностями из одного или нескольких символов по аналогии с тем, как действуют подстановочные знаки в других приложениях.

Применяя в сочетании обычные и специальные символы, можно определять шаблоны для сопоставления с данными почти любого типа. Затем можно воспользоваться редактором `sed` или программой `gawk`, чтобы отфильтровать конкретные данные из более крупного потока данных или проверить допустимость данных, полученных из приложений ввода данных.

В следующей главе более подробно рассматривается тема использования редактора `sed` для выполнения более сложных манипуляций с текстом. Редактор `sed` поддерживает широкий набор дополнительных функций, благодаря чему он становится применимым для обработки больших потоков данных и отфильтровывания из них только требуемых данных.

ГЛАВА 20

В этой главе...

Многострочные команды

Пространство хранения

Обращение команды

Изменение потока управления

Замена шаблона

Использование редактора sed в сценариях

Создание программ sed

Резюме

Дополнительные сведения о редакторе sed

В главе 18 было показано, как использовать основные возможности редактора sed для управления текстом в потоках данных. Основные команды редактора sed способны обеспечивать выполнение большинства повседневных требований по редактированию текста. А в настоящей главе рассматриваются более сложные функции, которые могут осуществляться с помощью редактора sed. К ним относятся такие средства, которые не требуются слишком часто, но если возникает необходимость в их использовании, то важно знать о существовании этих средств и о том, как с ними работать.

Многострочные команды

Знакомясь с работой основных команд редактора sed, читатель мог заметить одно ограничение. Все ранее описанные команды редактора sed выполняют функции применительно к одной строке данных. По мере того как редактор sed считывает поток данных, происходит разбиение данных на строки по признаку наличия символов обозначения конца строки. Редактор sed обрабатывает одновременно одну строку данных, применяет к строке текста команды, которые определены в сценарии, а затем переходит к следующей строке текста и повторяет процесс.

Но иногда возникает необходимость выполнять те или иные действия по отношению к данным, которые занимают больше одной строки. Особенно часто это происходит при осуществлении попыток поиска или замены целой фразы.

Например, если происходит поиск в данных фразы `Linux System Administrators Group`, то весьма велика вероятность того, что эта фраза может быть разбита на две строки, причем разбиение может произойти между любыми парами слов во фразе. При осуществлении обработки данного текста с помощью обычных команд редактора `sed` невозможно определить, каким образом была разбита фраза.

К счастью, разработчики редактора `sed` предвидели возникновение подобных ситуаций и предложили решение. В редакторе `sed` предусмотрены три специальные команды, которые могут использоваться для обработки многострочного текста, описанные ниже.

- `N`. Добавление следующей строки из потока данных для создания многострочной группы в целях последующей обработки.
- `D`. Удаление отдельной строки в многострочной группе.
- `P`. Вывод отдельной строки в многострочной группе.

В следующих разделах эти многострочные команды рассматриваются более подробно и показано, как их можно использовать в конкретных сценариях.

Команда `next`

Прежде чем перейти к описанию многострочной команды `next`, необходимо в первую очередь рассмотреть, как работает однострочная версия команды `next`. После того как станет известно, для чего предназначена эта команда, будет проще понять, как действует многострочная версия команды `next`.

Однострочная команда `next`

Команда `n`, задаваемая в нижнем регистре, указывает редактору `sed`, что необходимо перейти к следующей строке текста в потоке данных, не возвращаясь к началу списка команд. Напомним, что обычно редактор `sed` применяет к строке все определенные для него команды, прежде чем перейти к следующей строке текста в потоке данных. Однострочная команда `next` служит для изменения этого потока управления.

На первых порах бывает трудно понять, что при этом происходит, а иногда осуществляемые действия становятся почти непостижимыми. В рассматриваемом примере осуществляется обработка файла данных, состоящего из пяти строк, причем две строки — пустые. Задача состоит в том, чтобы удалить пустую строку после строки заголовка, но оставить пустую строку перед последней строкой нетронутой. Если предусмотреть в сценарии `sed` просто удаление пустых строк, то произойдет удаление обеих пустых строк:

```
$ cat data1
This is the header line.

This is a data line.

This is the last line.
$
$ sed '/^$/d' data1
This is the header line.
This is a data line.
This is the last line.
$
```

Причем строка, которую требуется удалить по условию задачи, пуста, поэтому отсутствует какой-либо текст, который можно было бы задать в критерии поиска для однозначного определения строки. Решение состоит в использовании команды `n`. В следующем примере сценария происходит поиск уникальной строки, которая содержит слово `header`. После обнаружения в сценарии этой строки команда `n` переводит редактор `sed` на следующую строку текста, которая является пустой.

```
$ sed '/header/{n ; d}' data1
This is the header line.
This is a data line.

This is the last line.
$
```

С этого момента редактор `sed` продолжает обработку списка команд, в котором предусмотрена команда `d`, предназначенная для удаления пустой строки. После достижения редактором `sed` конца командного сценария происходит считывание следующей строки текста из потока данных и запуск команд на обработку с начала командного сценария. Редактор `sed` не обнаруживает еще одну строку со словом `header`, поэтому больше не происходит удаление каких-либо строк.

Объединение строк текста

Теперь, после ознакомления с описанием однострочной команды `next`, можно перейти к рассмотрению многострочной версии. Однострочная команда `next` фактически обеспечивает перемещение следующей строки текста из потока данных в пространство обработки (называемое *пространством шаблонов*) редактора `sed`. Многострочная версия команды `next` (для обозначения которой служит прописная буква `N`) добавляет следующую строку текста к тому тексту, который уже находится в пространстве шаблонов.

Результатом данного действия становится то, что две строки текста из потока данных объединяются в одном и том же пространстве шаблонов. Эти строки текста по-прежнему разделены символом обозначения конца строки, но теперь редактор `sed` может рассматривать обе строки текста как одну строку.

Ниже приведен пример того, как работает команда `N`.

```
$ cat data2
This is the header line.
This is the first data line.
This is the second data line.
This is the last line.
$
$ sed '/first/{ N ; s/\n/ / }' data2
This is the header line.
This is the first data line. This is the second data line.
This is the last line.
$
```

В данном сценарии редактора `sed` происходит поиск строки текста, в которой содержится слово `"first"`. После обнаружения этой строки используется команда `N` для объединения с ней следующей строки. Затем применяется команда `substitution` (сокращенно `s`) для замены символа обозначения конца строки пробелом. Результатом становится то, что в выводе редактора `sed` эти две строки из текстового файла преобразуются в одну строку.

Такой образец сценария может найти практическое применение, если ведется поиск текстовой фразы, которая может быть разбита между двумя строками в файле данных. Соответствующий пример приведен ниже.

```
$ cat data3
The first meeting of the Linux System
Administrator's group will be held on Tuesday.
All System Administrators should attend this meeting.
Thank you for your attendance.
$
$ sed 's/System Administrator/Desktop User/' data3
The first meeting of the Linux System
Administrator's group will be held on Tuesday.
All Desktop Users should attend this meeting.
Thank you for your attendance.
$
```

Команда substitution применяется для поиска в текстовом файле конкретной фразы System Administrator, состоящей из двух слов. Если вся фраза обнаруживается в одной строке, то дела обстоят проще всего; достаточно воспользоваться командой substitution для замены текста. Но в такой ситуации, в которой эта фраза разбита между двумя строками, команда substitution не распознает шаблон сопоставления.

Для решения этой проблемы можно применить команду N:

```
$ sed 'N ; s/System.Administrator/Desktop User/' data3
The first meeting of the Linux Desktop User's group will be
held on Tuesday.
All Desktop Users should attend this meeting.
Thank you for your attendance.
$
```

Применение команды N для объединения следующей строки с той строкой, в которой обнаружено первое слово фразы, позволяет выполнить задание и в той ситуации, когда фраза разбита на две строки.

Следует отметить, что в применяемой команде substitution задан шаблон с подстановочным знаком (.) между словами "System" и "Administration", чтобы сопоставление можно было успешно выполнить и при наличии пробела, и при наличии символа обозначения конца строки. Но после сопоставления с символом обозначения конца строки этот символ удаляется из строки, что приводит к объединению двух строк в одну строку. Возможно, такой побочный эффект является нежелательным.

Чтобы решить эту проблему, можно воспользоваться двумя командами substitution в сценарии редактора sed; одна команда должна выполняться при сопоставлении с многострочным вхождением и еще одна — при обнаружении однострочного вхождения:

```
$ sed '
> N
> s/System\nAdministrator/Desktop\nUser/
> s/System Administrator/Desktop User/
> ' data3
The first meeting of the Linux Desktop
User's group will be held on Tuesday.
All Desktop Users should attend this meeting.
```

```
Thank you for your attendance.  
$
```

В первой команде substitution специально заданы поиск символа обозначения конца строки между двумя искомыми словами и включение этого символа в строку замены. Это позволяет добавить символ обозначения конца строки в новом тексте в том же месте.

Тем не менее этот сценарий имеет еще один трудноуловимый недостаток. В нем всегда происходит считывание следующей строки текста в пространство шаблонов перед выполнением команды редактора sed. А после достижения последней строки текста больше не обнаруживается следующая строка текста, которую можно было бы прочитать, поэтому команда N вынуждает редактор sed остановиться. Если сопоставляемый текст будет находиться в последней строке в потоке данных, то команды данного сценария не смогут перехватить сопоставляемые данные:

```
$ cat data4  
The first meeting of the Linux System  
Administrator's group will be held on Tuesday.  
All System Administrators should attend this meeting.  
$  
$ sed '  
> N  
> s/System\nAdministrator/Desktop\nUser/  
> s/System Administrator/Desktop User/  
> ' data4  
The first meeting of the Linux Desktop  
User's group will be held on Tuesday.  
All System Administrators should attend this meeting.  
$
```

В данном случае текст System Administrator появляется в последней строке в потоке данных, поэтому команда N его пропускает, в связи с тем, что больше нет другой строки, которую можно было бы считать в пространство шаблонов для объединения. Эту проблему можно легко решить. Для этого достаточно поместить однострочные команды перед командой N и предусмотреть появление после команды N только многострочных команд следующим образом:

```
$ sed '  
> s/System Administrator/Desktop User/  
> N  
> s/System\nAdministrator/Desktop\nUser/  
> ' data4  
The first meeting of the Linux Desktop  
User's group will be held on Tuesday.  
All Desktop Users should attend this meeting.  
$
```

Теперь команда substitution, предназначенная для поиска фразы в одной строке, успешно справляется с последней строкой в потоке данных, а многострочная команда substitution применяется к вхождениям, обнаруживаемым в середине потока данных.

Многострочная команда delete

В главе 18 было приведено вводное описание однострочной команды delete (сокращенно d). В редакторе sed эта команда используется для удаления текущей строки в пространстве

шаблонов. Однако в процессе работы с командой `N` необходимо соблюдать осторожность, используя однострочную команду `delete`:

```
$ sed 'N ; /System\nAdministrator/d' data4
All System Administrators should attend this meeting.
$
```

При использовании этой команды `delete` осуществляется поиск слов `System` и `Administrator` в отдельных строках, а затем удаление обеих строк в пространстве шаблонов. Это может соответствовать или не соответствовать замыслу разработчика сценария.

В редакторе `sed` предусмотрена многострочная команда `delete` (обозначаемая как `D`), которая удаляет только первую строку в пространстве шаблонов. С помощью этой команды удаляются все символы указанной строки, включая символ обозначения конца строки:

```
$ sed 'N ; /System\nAdministrator/d' data4
Administrator's group will be held on Tuesday.
All System Administrators should attend this meeting.
$
```

Вторая строка текста, добавленная в пространство шаблонов командой `N`, остается нетронутой. Необходимость в использовании такой программной конструкции может возникнуть, если требуется удалить строку текста, появляющуюся перед строкой, в которой обнаруживается искомая строка данных.

Ниже приведен пример удаления пустой строки, которая появляется перед первой строкой в потоке данных.

```
$ cat data5

This is the header line.
This is a data line.

This is the last line.
$
$ sed '/^$/N ; /header/D' data5This is the header line.
This is a data line.

This is the last line.
$
```

В этом сценарии редактора `sed` происходит поиск пустой строки, а затем используется команда `N` для добавления следующей строки текста в пространство шаблонов. Если новое содержимое пространства шаблонов содержит слово `header`, команда `D` удаляет первую строку в пространстве шаблонов. Без использования такой комбинации команд `N` и `D` было бы невозможно удалить первую пустую строку, не удаляя все прочие пустые строки.

Многострочная команда `print`

К этому моменту читателю должно быть ясно, в чем состоят различия между однострочными и многострочными версиями команд. Функционирование многострочной команды `print` (обозначается как `P`) происходит по такому же принципу. Эта команда осуществляет вывод только первой строки из многострочного пространства шаблонов. В состав вывода входят все символы в пространстве шаблонов вплоть до символа обозначения конца строки. Многострочная команда вывода применяется во многом аналогично однострочной команде `p`, для отображения текста в той ситуации, когда для подавления вывода из сценария используется опция `-n`.

```
$ sed -n 'N ; /System\nAdministrator/P' data3
The first meeting of the Linux System
$
```

Если происходит многострочное сопоставление, то команда `P` выводит только первую строку из пространства шаблонов. Но все возможности многострочной команды `P` обнаруживаются только при ее использовании в сочетании с многострочными командами `N` и `D`.

Команда `D` обладает уникальной особенностью — она вынуждает редактор `sed` возвратиться к началу сценария и повторно выполнить команды применительно к тому же пространству шаблонов, что и перед этим (она не считывает символ перевода на новую строку текста из потока данных). Путем включения команды `N` в командный сценарий можно по существу обеспечить одноразовое прохождение через пространство шаблонов, сопоставляя одновременно несколько строк.

После этого с помощью команды `P` можно вывести первую строку, а затем воспользоваться командой `D`, а это равносильно удалению первой строки и переходу по циклу в начало сценария. После того как происходит возврат в начало сценария, команда `N` считывает следующую строку текста и опять запускает процесс; это происходит снова и снова. Такой цикл продолжается до достижения конца потока данных.

Пространство хранения

Пространство шаблонов — это активная область буферов, в которой хранится текст, просматриваемый редактором `sed` в ходе обработки команд. Но это не единственное пространство, предусмотренное в редакторе `sed` для хранения текста.

В редакторе `sed` может использоваться еще одна область буферов, называемая *пространством хранения*. Пространство хранения можно применять для сохранения на время одних строк текста в ходе работы над другими строками в пространстве шаблонов. Для работы с пространством хранения предусмотрены пять команд, которые приведены в табл. 20.1.

Таблица 20.1. Команды для работы с пространством хранения редактора `sed`

Команда	Описание
<code>h</code>	Скопировать пространство шаблонов в пространство хранения
<code>H</code>	Присоединить пространство шаблонов к пространству хранения
<code>g</code>	Скопировать пространство хранения в пространство шаблонов
<code>G</code>	Присоединить пространство хранения к пространству шаблонов
<code>x</code>	Осуществить обмен содержимым между пространством шаблонов и пространством хранения

Эти команды позволяют выполнять различные операции копирования текста из пространства шаблонов в пространство хранения. Это позволяет освободить пространство шаблонов, чтобы загрузить еще одну строку для обработки.

Обычно после использования команды `h` или `H` для перемещения строки в пространство хранения в конечном итоге приходится применять команду `g`, `G` или `x` для обратного перемещения хранимой строки в пространство шаблонов (в противном случае не нужно было бы, прежде всего, заботиться о ее сохранении).

В связи с наличием двух областей буферов попытка определить, какая строка текста находится в той или иной области буферов, иногда может стать затруднительной. Ниже приведен

небольшой пример, который показывает, как использовать команды `h` и `g` для прямого и обратного перемещения данных между пространствами буферов редактора `sed`.

```
$ cat data2
This is the header line.
This is the first data line.
This is the second data line.
This is the last line.
$
$ sed -n '/first/{
> h
> p
> n
> p
> g
> p
> }' data2
This is the first data line.
This is the second data line.
This is the first data line.
$
```

Рассмотрим этот пример кода шаг за шагом.

1. В сценарии `sed` используется регулярное выражение в адресе для фильтрации строки, содержащей слово `first`.
2. При обнаружении строки со словом `first` команда `h` помещает ее в пространство хранения.
3. Команда `p` выводит содержимое пространства шаблонов, в котором все еще находится первая строка данных.
4. Команда `n` осуществляет выборку следующей строки в потоке данных (строки `This is the second data line`) и помещает ее в пространство шаблонов.
5. Команда `p` выводит содержимое пространства шаблонов, в котором теперь находится вторая строка данных.
6. Команда `g` помещает содержимое пространства хранения (строку `This is the first data line`) обратно в пространство шаблонов, заменяя текущий текст.
7. Команда `p` выводит текущее содержимое пространства шаблонов, в которое теперь возвращена первая строка данных.

Таким образом, происходит обмен местами текстовых строк с использованием пространства хранения, что позволяет принудительно обеспечить появление в выводе первой строки данных после второй строки данных. Если просто опустить первую команду `p`, то можно вывести эти две строки в обратном порядке:

```
$ sed -n '/first/{
> h
> n
> p
> g
> p
> }' data2
```

```
This is the second data line.  
This is the first data line.  
$
```

Итак, вырисовываются контуры создания действительно полезных приложений. Например, эту методику можно использовать для создания сценария `sed`, который производит обращение (размещение всех строк в обратном порядке) всего файла с текстовыми строками! Но для этого прежде всего необходимо ознакомиться со средствами обращения команд редактора `sed`, описанию которых полностью посвящен следующий раздел.

Обращение команды

В главе 18 было показано, что редактор `sed` применяет команды либо к каждой текстовой строке в потоке данных, либо к строкам, определенно указанным с помощью отдельного адреса или диапазона адресов. Можно также выполнить настройку команды таким образом, чтобы она производила обратное действие — не применялась к конкретному адресу или диапазону адресов в потоке данных.

Для такого обращения команды применяется команда восклицательного знака (!). Обращение команды означает, что в ситуациях, в которых команда обычно активизируется, этого не происходит. Ниже приведен пример, демонстрирующий данное средство.

```
$ sed -n '/header/!p' data2  
This is the first data line.  
This is the second data line.  
This is the last line.  
$
```

В обычных обстоятельствах команда `p` вывела бы только ту строку из файла `data2`, которая содержит слово `header`. Но после добавления восклицательного знака выводятся все строки в файле, кроме строки, содержащей текст, на который ссылается адрес.

Применение восклицательного знака становится необходимым в целом ряде приложений. Напомним, что выше в этой главе, в разделе “Команды `next`”, была показана ситуация, в которой команду редактора `sed` в сценарии нельзя было применить к последней строке текста в потоке данных, поскольку за ней не следовала еще одна строка. Чтобы исправить этот недостаток сценария, можно воспользоваться восклицательным знаком:

```
$ sed 'N; s/System.Administrator/Desktop User/' data4  
The first meeting of the Linux Desktop User's group will be held  
⚡ on Tuesday  
All System Administrators should attend this meeting.  
$  
$ sed '$!N; s/System.Administrator/Desktop User/' data4  
The first meeting of the Linux Desktop User's group will be held  
⚡ on Tuesday  
All Desktop Users should attend this meeting.  
$
```

В этом примере показано использование восклицательного знака в сочетании с командой `N`, а также со специальным адресом в виде знака доллара. Знак доллара представляет последнюю строку текста в потоке данных, поэтому после достижения редактором `sed` последней строки команда `N` больше не выполняется. Но для всех прочих строк данная команда выполняется.

С использованием этого подхода можно изменить последовательность текстовых строк в потоке данных на противоположную. Чтобы расположить строки в порядке, обратном тому, в котором они появляются в потоке текстовых строк (отобразить в начале последнюю строку, а в конце — первую), необходимо выполнить некоторые сложные манипуляции с пространством хранения.

Шаблон, необходимый для работы в этой ситуации, выглядит примерно так:

1. Поместить строку в пространство шаблонов.
2. Переместить строку из пространства шаблонов в пространство хранения.
3. Поместить следующую строку текста в пространство шаблонов.
4. Присоединить пространство хранения к пространству шаблонов.
5. Поместить все содержимое пространства шаблонов в пространство хранения.
- 6–9. Повторять шаги 3–5 до тех пор, пока все строки не будут представлены в обратном порядке в пространстве хранения.
10. Осуществить выборку строк и вывести эти строки.

На рис. 20.1 описанный порядок действий показан схематически более подробно.

Обращение последовательности строк в текстовом файле с использованием пространства хранения

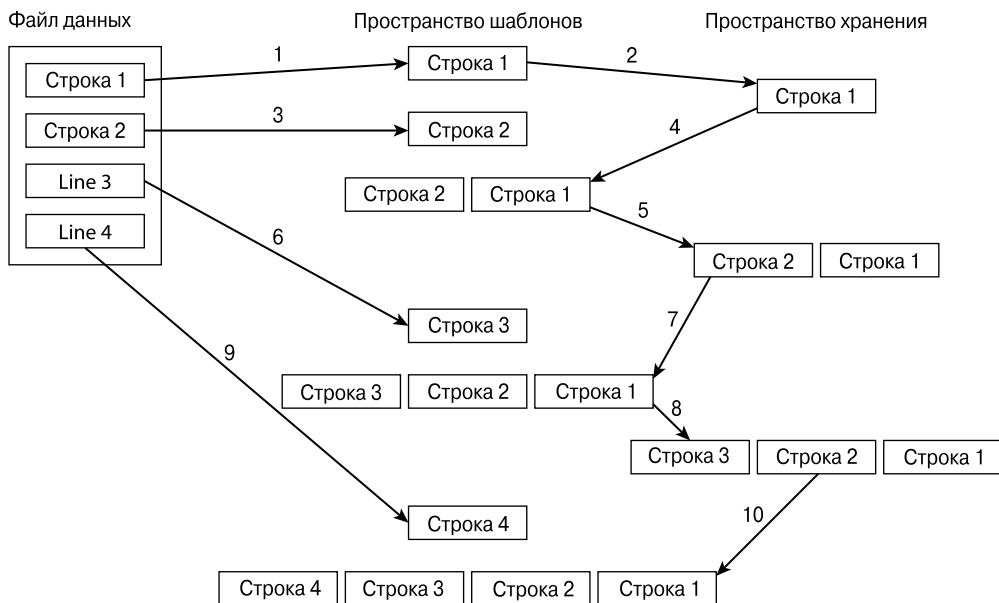


Рис. 20.1. Обращение порядка следования строк в текстовом файле с помощью пространства хранения

При использовании данного метода следует избегать вывода строк в ходе их обработки. Это означает, что для редактора `sed` должна использоваться опция командной строки `-n`. Затем необходимо определить, как обеспечить присоединение текста из пространства хранения к тексту в пространстве шаблонов. Для этого используется команда `G`. Единственная проблема состоит в том, что нужно предотвратить присоединение содержимого пространства хранения

к первой строке обработанного текста. Эту проблему можно легко решить с использованием команды восклицательного знака:

```
1!G
```

Следующий шаг состоит в перемещении нового пространства шаблонов (текстовой строки с добавленными обращенными строками) в пространство хранения. Это достаточно просто; следует лишь воспользоваться командой `h`.

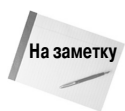
После того как весь поток данных будет представлен в пространстве шаблонов в обратном порядке, остается только вывести полученные результаты. Узнавать о том, что весь поток данных уже находится в пространстве шаблонов, можно по тому признаку, достигнута ли последняя строка в потоке данных. Чтобы вывести результаты, остается воспользоваться следующей командой:

```
$p
```

Выше описаны фрагменты, необходимые для создания сценария обращения последовательности строк для редактора `sed`. Теперь проведем испытание этого сценария в тестовом прогоне:

```
$ cat data2
This is the header line.
This is the first data line.
This is the second data line.
This is the last line.
$
$ sed -n '{1!G ; h ; $p }' data2
This is the last line.
This is the second data line.
This is the first data line.
This is the header line.
$
```

Очевидно, что этот сценарий редактора `sed` действует в соответствии с ожидаемым. В выводе этого сценария происходит обращение порядка исходных строк в текстовом файле. Данный сценарий наглядно демонстрирует широкие возможности использования пространства хранения в сценариях `sed`. Пространство хранения предоставляет удобный способ управления расположением строк в выводе сценария.



Любознательный читатель может узнать, что есть команда Linux, специально предназначенная для выполнения функции обращения текстового файла. Таковой является команда `tac`, которая отображает строки текстового файла в обратном порядке. Читатель мог заметить, насколько остроумно выбрано имя для этой команды, поскольку она выполняет функцию, обратную по отношению к команде `cat`.

Изменение потока управления

В обычных обстоятельствах редактор `sed` последовательно обрабатывает команды сценария от начала и до конца (исключением является команда `D`, которая вынуждает редактор `sed` возвратиться в верхнюю часть сценария, не читая в тексте символ перевода на новую строку). Но, кроме этого, в редакторе `sed` предусмотрен способ изменения потока команд командного сценария, в результате чего достигается эффект, аналогичный применению среды структурного программирования.

Выполнение перехода

В предыдущем разделе было показано, что команда восклицательного знака позволяет достичь эффекта, обратного по отношению к обычному результату применения команды к строке текста. В редакторе `sed` предусмотрен способ обращения целого раздела команд с учетом отдельного адреса, шаблона адреса или диапазона адресов. Это позволяет выполнять группу команд по отношению лишь к конкретному подмножеству строк в потоке строк данных.

Команда перехода `branch` имеет следующий формат:

```
[address]b [label]
```

Параметр адреса `address` определяет, какая строка (или строки) данных вызывает команду `branch`. Параметр метки `label` определяет местоположение, в которое должен быть выполнен переход. Если параметр `label` отсутствует, то команда `branch` выполняет переход к концу сценария.

```
$ sed '{2,3b ; s/This is/Is this;/s/line./test?/}' data2
Is this the header test?
This is the first data line.
This is the second data line.
Is this the last test?
$
```

Команда `branch` обеспечивает пропуск двух команд `substitution` применительно ко второй и третьим строкам в потоке данных.

Вместо перехода в конец сценария можно предусмотреть метку для команды `branch`, к которой должен быть выполнен переход. Метки начинаются с двоеточия и могут иметь длину до семи символов:

```
:label2
```

Для задания метки достаточно добавить ее после команды `b`. Использование меток позволяет пропускать команды, которые сопоставляются с адресом `branch`, но по-прежнему обрабатывать другие команды в сценарии:

```
$ sed '{/first/b jump1 ; s/This is the/No jump on/
> :jump1
> s/This is the/Jump here on/}' data2
No jump on header line
Jump here on first data line
No jump on second data line
No jump on last line
$
```

Команда `branch` указывает, что в программе должен быть выполнен переход к строке сценария с меткой `jump1`, если в строке обнаруживается сопоставляемый текст "first". Если сопоставление с шаблоном команды `branch` не происходит, то редактор `sed` продолжает обрабатывать команды в сценарии, включая команду после метки `branch`. (Таким образом, все три команды `substitution` выполняются по отношению к строкам, не сопоставляемым с шаблоном `branch`.)

Если же строка сопоставляется с шаблоном `branch`, то редактор `sed` переходит к строке с меткой `branch`. Таким образом, выполняется только последняя команда `substitution`.

В этом примере показано выполнение перехода к метке, расположенной еще дальше в сценарии `sed`. Можно также применить команду `branch` для перехода к метке, которая появляется выше в сценарии, создавая тем самым эффект, аналогичный организации циклов:

```
$ echo "This, is, a, test, to, remove, commas. " | sed -n '{
> :start
> s/,//lp
> b start
> }'
This is, a, test, to, remove, commas.
This is a, test, to, remove, commas.
This is a test, to, remove, commas.
This is a test to, remove, commas.
This is a test to remove, commas.
This is a test to remove commas.
```

В каждой итерации в этом сценарии происходит удаление первого вхождения запятой из текстовой строки и вывод строки. Но этот сценарий имеет один недостаток: он никогда не заканчивается. В этой ситуации создается бесконечный цикл поиска запятых, который повторяется до тех пор, пока сценарий не будет оставлен вручную путем отправки сигнала с помощью комбинации клавиш `<Ctrl+C>`.

Для предотвращения возникновения подобной проблемы необходимо указывать для команды `branch` шаблон адреса, с помощью которого должен осуществляться поиск. Если такой шаблон отсутствует, то выполнение перехода должно прекратиться:

```
$ echo "This, is, a, test, to, remove, commas. " | sed -n '{
> :start
> s/,//lp
> /,/b start
> }'
This is, a, test, to, remove, commas.
This is a, test, to, remove, commas.
This is a test, to, remove, commas.
This is a test to, remove, commas.
This is a test to remove, commas.
This is a test to remove commas.
$
```

Теперь команда `branch` выполняет переход, только если в строке есть запятая. После удаления последней запятой выполнение команды `branch` прекращается, что позволяет завершить работу сценария должным образом.

Проверка

Аналогично команде `branch`, для изменения потока управления в сценарии редактора `sed` может также использоваться команда `test` (сокращенно `t`). Но вместо перехода к метке с учетом адреса команда `test` переходит к метке с учетом результата команды `substitution`.

Если команда `substitution` успешно выполняет сопоставление и подстановку шаблона, то команда `test` осуществляет переход к указанной метке. Если же в команде `substitution` указанный шаблон не сопоставляется, то не происходит переход с помощью команды `test`.

В команде `test` используется такой же формат, как и в команде `branch`:

```
[address]t [label]
```

Аналогично команде `branch`, если метка не указана, редактор `sed` осуществляет переход к концу сценария в случае успешного выполнения проверки командой `test`.

Команда `test` позволяет проще всего реализовать базовую инструкцию `if-then`, действие которой основано на проверке текста в потоке данных. Например, если нужно предотвратить выполнение подстановки после того, как уже была сделана другая подстановка, на помощь придет команда `test`:

```
$ sed '{
> s/first/matched/
> t
> s/This is the/No match on/
> }' data2
No match on header line
This is the matched data line
No match on second data line
No match on last line
$
```

Первая команда substitution применяется для поиска текста шаблона `first`. Если происходит сопоставление с шаблоном в строке, то выполняется подстановка текста, и команда `test` обеспечивает пропуск следующей команды substitution. Если же первая команда substitution не сопоставляется с шаблоном, то обрабатывается вторая команда substitution.

С использованием команды `test` можно упростить организацию цикла, который мы пытались создать перед этим с помощью команды `branch`:

```
$ echo "This, is, a, test, to, remove, commas. " | sed -n '{
> :start
> s/,//lp
> t start
> }'
This is, a, test, to, remove, commas.
This is a, test, to, remove, commas.
This is a test, to, remove, commas.
This is a test to, remove, commas.
This is a test to remove, commas.
This is a test to remove commas.
$
```

Если больше не обнаруживаются подстановки, подлежащие выполнению, то команда `test` не осуществляет переход, поэтому продолжается работа с остальной частью сценария.

Замена шаблона

Выше было показано, как использовать шаблоны в командах `sed` для замены текста в потоке данных. Но при использовании подстановочных знаков нелегко определить, какой текст сопоставляется с шаблоном.

Например, предположим, что требуется поместить двойные кавычки вокруг слова, которое сопоставлено с шаблоном в строке. Это достаточно просто, если в шаблоне задано только одно слово, подлежащее сопоставлению:

```
$ echo "The cat sleeps in his hat. " | sed 's/cat/"cat"/'  
The "cat" sleeps in his hat.  
$
```

Но могут быть и такие ситуации, в которых, допустим, в шаблоне используется подстановочный знак (.) для сопоставления больше чем с одним словом.

```
$ echo "The cat sleeps in his hat. " | sed 's/.at/"at"/g'  
The ".at" sleeps in his ".at".  
$
```

В этой строке подстановки использовался подстановочный знак точки для сопоставления с вхождением любой буквы, за которым следует подстрока "at". К сожалению, заменяющая строка выглядит иначе, чем сопоставленное слово с буквально присутствующим в нем подстановочным знаком.

Амперсанд

В редакторе sed предусмотрено решение этой проблемы. Для представления шаблона сопоставления в команде substitution может применяться символ амперсанда (&). При каждом сопоставлении текста с определенным шаблоном можно использовать символ амперсанда для восстановления содержимого данного текста в шаблоне замены. Это позволяет управлять дальнейшими действиями с учетом того, какое слово сопоставлено с определенным шаблоном:

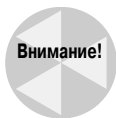
```
$ echo "The cat sleeps in his hat. " | sed 's/.at/"&"/g'  
The "cat" sleeps in his "hat".  
$
```

Если с шаблоном сопоставлено слово cat, то в заменяющем слове появляется "cat". Если же произошло сопоставление со словом hat, то в заменяющем слове появляется "hat".

Замена отдельных слов

Применение символа амперсанда приводит к получению всей строки, которая сопоставлена с шаблоном, заданным в команде substitution. Но иногда возникает необходимость произвести выборку лишь подстроки этой строки. Предусмотрена и такая возможность, но при этом приходится выполнять более сложные действия.

В редакторе sed для определения составляющих подстрок в пределах шаблона подстановки используются круглые скобки. Если с помощью круглых скобок определены составляющие подстроки, то на каждую из них можно сослаться в шаблоне замены с помощью специального символа — символа замены. Символ замены состоит из обратной косой черты и числа. Число указывает порядковый номер составляющей подстроки. Редактор sed присваивает первой составляющей подстроке символ замены \1, второй — символ замены \2 и т.д.



При использовании круглых скобок в команде substitution необходимо задавать экранирующие символы для указания на то, что эти круглые скобки являются символами группирования, а не обычными круглыми скобками. Это действие противоположно тому, которое осуществляется при экранировании других специальных символов.

Рассмотрим пример применения этого средства в сценарии редактора sed:

```
$ echo "The System Administrator manual" | sed '  
> s/(System\) Administrator/\1 User/'
```

The System User manual
\$

В этой команде substitution используется пара круглых скобок вокруг слова "System" для его обозначения как составляющей подстроки. Затем в шаблоне замены используется символ замены \1 для восстановления значения первой заданной составляющей подстроки. В данном случае мы фактически не вышли за рамки обычных подстановок, но когда происходит применение шаблонов с подстановочными знаками, раскрываются реальные возможности этого метода.

Если, скажем, требуется заменить фразу только отдельным словом, то достаточно задать необходимую подстроку для этой фразы, но иногда для определения подстроки приходится использовать подстановочные знаки; в этом случае составляющие подстроки действительно могут прийти на помощь:

```
$ echo "That furry cat is pretty" | sed 's/furry \(.at\)\/\1/'
That cat is pretty
$ echo "That furry hat is pretty" | sed 's/furry \(.at\)\/\1/'
That hat is pretty
$
```

В данной ситуации нельзя использовать символ амперсанда, поскольку он предназначен для замены всего шаблона сопоставления. Для решения задачи можно использовать составляющие подстроки, которые позволяют выбрать только определенную часть шаблона для применения в качестве шаблона замены.

Это средство может стать особенно полезным, если возникает необходимость вставить текст между двумя или несколькими составляющими подстроками. Ниже приведен сценарий, в котором составляющие подстроки применяются для вставки запятых в числа, состоящие из большого количества цифр:

```
$ echo "1234567" | sed '{
> :start
> s/\(.*[0-9]\)\([0-9]\{3\}\)/\1,\2/
> t start
> }'
1,234,567
$
```

В этом сценарии шаблон сопоставления подразделяется на две составляющие подстроки:

```
.*[0-9]
[0-9]{3}
```

Данный шаблон обеспечивает поиск двух подстрок. Первая определяет любое количество символов, заканчивающихся цифрой. Вторая подстрока задает ряд из трех цифр (о том, как использовать фигурные скобки в регулярном выражении, см. в главе 19). При обнаружении этого шаблона в тексте замещающий текст применяется для ввода запятой между двумя составляющими подстроками, каждая из которых определяется своим символом замены. В этом сценарии используется команда test для итерационной обработки числа, которая происходит до тех пор, пока не будут расставлены все запятые.

Использование редактора sed в сценариях

Выше были описаны разнообразные направления использования редактора sed, а теперь мы можем собрать эти сведения воедино и применить их в сценариях командного интерпретатора. В данном разделе демонстрируются определенные средства, о которых следует знать при использовании редактор sed в сценариях командного интерпретатора bash.

Использование оболочек

Читатель мог заметить, что попытка задать действующий сценарий редактора sed для практического использования может оказаться трудоемкой, особенно если сценарий должен состоять из большого количества команд. Предусмотрена возможность не вводить заново весь сценарий каждый раз, когда возникает необходимость в его использовании, а поместить соответствующие команды редактор sed в *оболочку* сценария командного интерпретатора. Оболочка действует как посредник между сценарием редактора sed и командной строкой.

При этом после перехода в сценарии командного интерпретатора можно использовать обычные переменные и параметры командного интерпретатора в сценарии редактора sed. Ниже приведен пример применения переменной параметра командной строки в качестве входных данных для сценария sed.

```
$ cat reverse
#!/bin/bash
# оболочка командного интерпретатора для сценария редактора sed,
# обеспечивающего обращение строк

sed -n '{
1!G
h
$P
}' $1
$
```

В этом сценарии командного интерпретатора, называемом *reverse*, используется сценарий редактора sed для обращения текстовых строк в потоке данных. В нем используется параметр командного интерпретатора \$1 для получения первого параметра из командной строки, который должен задавать имя файла, подлежащего обращению:

```
$ ./reverse data2
This is the last line.
This is the second data line.
This is the first data line.
This is the header line.
$
```

Таким образом, появляется возможность легко применять сценарии редактора sed для обработки любых файлов, не задавая каждый раз весь текст сценария в командной строке.

Перенаправление вывода sed

По умолчанию редактор sed выводит результаты выполнения сценария в поток STDOUT. В сценариях командного интерпретатора можно использовать все стандартные методы перенаправления вывода редактора sed.

Например, можно использовать обратные одинарные кавычки для перенаправления вывода команды редактора sed в переменную для дальнейшего применения в сценарии. Ниже приведен пример использования сценария sed для добавления запятых к результату числового расчета.

```
$ cat fact
#!/bin/bash
# вставка запятых в числовой результат вычисления факториала

factorial=1
counter=1
number=$1

while [ $counter -le $number ]
do
    factorial=$(( $factorial * $counter )
    counter=$(( $counter + 1 )
done

result=$(echo $factorial | sed '{
:start
s/(.*[0-9])\([0-9]\{3\})/1,2/
t start
}'
)

echo "The result is $result"
$
$ ./fact 20
The result is 2,432,902,008,176,640,000
$
```

После вызова на выполнение обычного сценария вычисления факториала полученный в этом сценарии результат используется в качестве входных данных для сценария редактора sed, предназначенного для добавления запятых. Затем это значение используется в инструкции echo для вывода результата.

Создание программ sed

Даже краткие примеры, приведенные выше, показывают, что с помощью редактора sed можно получить весьма привлекательные результаты в области форматирования данных. В настоящем разделе приведено несколько удобных и широко известных сценариев редактора sed, предназначенных для выполнения распространенных функций обработки данных.

Строки с двойными интервалами

Для начала рассмотрим простой сценарий sed, предназначенный для вставки пустой строки между строками в текстовом файле:

```
$ sed 'G' data2
This is the header line.

This is the first data line.

This is the second data line.

This is the last line.
$
```

Очевидно, что это весьма несложно! Весь секрет применяемого при этом приема состоит в использовании значения пространства хранения по умолчанию. Напомним, что команда `G` просто присоединяет содержимое пространства хранения к текущему содержимому пространства шаблонов. Сразу после запуска редактора `sed` пространство хранения содержит пустую строку. Добавление ее к существующей строке приводит к созданию пустой строки после существующей строки.

Можно заметить, что при работе этого сценария происходит также добавление пустой строки к последней строке в потоке данных, а это приводит к появлению пустой строки в конце файла. Если необходимо обойтись без этого, то можно воспользоваться символом обращения и символом обозначения последней строки, чтобы сценарий не добавлял пустую строку к последней строке в потоке данных:

```
$ sed '$!G' data2
This is the header line.

This is the first data line.

This is the second data line.

This is the last line.
$
```

Теперь полученный результат выглядит немного лучше. При условии, что строка не является последней, команда `G` присоединяет к ней содержимое пространства хранения. А после перехода редактора `sed` к последней строке происходит пропуск команды `G`.

Файлы с двойными интервалами, которые могут уже содержать пустые строки

Продолжим тему обработки файлов с двойными интервалами и рассмотрим вариант, в котором текстовый файл уже содержит несколько пустых строк, а теперь необходимо задать двойной интервал между всеми строками. В случае непосредственного использования предыдущего сценария в некоторых частях файла появляется слишком много пустых строк, поскольку удваивается также каждая существующая пустая строка:

```
$ cat data6
This is line one.
This is line two.

This is line three.
This is line four.
$
$ sed '$!G' data6
```

```
This is line one.  
This is line two.  
  
This is line three.  
  
This is line four.  
$
```

Теперь имеются три пустые строки там, где вначале находилась исходная пустая строка. Одним из решений этой проблемы может стать, прежде всего, удаление всех пустых строк из потока данных, а затем применение команды G для вставки новых пустых строк после каждой из существующих строк. Для удаления всех имеющихся пустых строк достаточно лишь воспользоваться командой d с шаблоном, который сопоставляется с пустой строкой:

```
/^$/d
```

В этом шаблоне используются дескриптор начала строки (знак вставки) и дескриптор конца строки (знак доллара). Добавление этого шаблона к сценарию приводит к получению требуемых результатов:

```
$ sed '/^$/d;$!G' data6  
This is line one.  
  
This is line two.  
  
This is line three.  
  
This is line four.  
$
```

Это как раз то, что требуется!

Нумерация строк в файле

В главе 18 было показано, как использовать знак равенства для отображения номеров строк в потоке данных:

```
$ sed '=' data2  
1  
This is the header line.  
2  
This is the first data line.  
3  
This is the second data line.  
4  
This is the last line.  
$
```

Полученный результат не совсем удобен для чтения, поскольку номера строк располагаются на одну строку выше каждой из нумеруемых строк в потоке данных. Лучшее решение состоит в том, чтобы номера строк размещались на одной строке с текстом.

Мы уже знакомы с тем, как объединить строки с помощью команды N, поэтому не должны возникать слишком большие затруднения при использовании этой информации для создания необходимого сценария редактора sed. Однако при реализации данного замысла возникает

сложность, заключающаяся в том, что нельзя объединить две команды, которые обеспечивают нумерацию и объединение строк, в одном и том же сценарии.

Поэтому после получения вывода команды знака равенства можно перенаправить этот вывод по каналу в другой сценарий редактора `sed`, в котором используется команда `N` для объединения строки с номером и строки с текстом. Необходимо также воспользоваться командой `substitution` для замены символа обозначения конца строки пробелом или знаком табуляции. Ниже показано, как может выглядеть окончательное решение.

```
$ sed '=' data2 | sed 'N; s/\n/ /'
1 This is the header line.
2 This is the first data line.
3 This is the second data line.
4 This is the last line.
$
```

Теперь полученный результат намного лучше! По существу, создана небольшая, но полезная программа, которую удобно иметь под рукой, выполняя работу по программированию, допустим, если потребуется найти строку кода по номеру, указанному в сообщении об ошибке.

Вывод последних строк

До сих пор в настоящей книге речь шла о том, как использовать команду `p` для вывода всех строк в потоке данных или только тех строк, которые сопоставляются с конкретным шаблоном. Но иногда возникает необходимость обрабатывать только последние несколько строк длинного листинга, такого как файл журнала.

Вывести лишь последнюю строку в потоке данных несложно, поскольку на появление последней строки указывает знак доллара:

```
$ sed -n '$p' data2
This is the last line.
$
```

Но можно ли использовать символ знака доллара для отображения заданного количества строк в конце потока данных? Ответ заключается в том, что необходимо создать *скользящее окно*.

Скользящее окно широко применяется для исследования блоков текстовых строк в пространстве шаблонов, созданных путем объединения строк с помощью команды `N`. Команда `N` присоединяет следующую строку текста к тому тексту, который уже находится в пространстве шаблонов. После накопления в пространстве шаблонов блока, скажем, из 10 текстовых строк, можно проверить, достигнут ли конец потока данных, с помощью знака доллара. Если это не последние строки, можно продолжить вводить дополнительные строки в пространство шаблонов, удаляя наряду с этим исходные строки (напомним, что команда `D` удаляет первую строку в пространстве шаблонов).

Организовав цикл, содержащий команды `N` и `D`, можно продолжить добавление новых строк к блоку строк в пространстве шаблонов и удаление старых строк. Для такого цикла идеально подходит команда `branch`. Чтобы завершить работу цикла, достаточно лишь определить наличие в блоке последней строки файла и вызвать команду `q` для выхода из цикла.

Ниже показан общий вид окончательного варианта сценария редактора `sed` такого типа.

```
$ sed '{
> :start
> $q
> N
```

```

> 11,$D
> b start
> }' /etc/passwd
user:x:1000:1000:user,,,:/home/user:/bin/bash
polkituser:x:113:121:PolicyKit,,,:/var/run/PolicyKit:/bin/false
sshd:x:114:65534::/var/run/sshd:/usr/sbin/nologin
Samantha:x:1001:1002:Samantha,4,,:/home/Samantha:/bin/bash
Debian-exim:x:115:124::/var/spool/exim4:/bin/false
usbmux:x:116:46:usbmux daemon,,,:/home/usbmux:/bin/false
rtkit:x:117:125:RealtimeKit,,,:/proc:/bin/false
Timothy:x:1002:1005::/home/Timothy:/bin/sh
Christine:x:1003:1006::/home/Christine:/bin/sh
kdm:x:118:65534::/home/kdm:/bin/false
$

```

В сценарии прежде всего проверяется, не является ли текущая строка последней строкой в потоке данных. В случае положительного ответа вызывается команда `quit`, которая останавливает цикл. Команда `N` присоединяет следующую строку к текущей строке в пространстве шаблонов. Команда `11,$D` удаляет первую строку в пространстве шаблонов, если текущая строка следует за строкой 10. В результате этого по существу создается подвижное окно в пространстве шаблонов.

Удаление строк

Еще одной программой для редактора `sed`, которая может оказаться полезной, является программа удаления нежелательных пустых строк в потоке данных. Задача удаления всех пустых строк из потока данных решается просто, но если потребуется удалять пустые строки выборочно, то необходимо будет проявить некоторую изобретательность. В настоящем разделе показано несколько несложных сценариев редактора `sed`, которыми можно воспользоваться, если потребуется удалить нежелательные пустые строки из данных.

Удаление подряд идущих пустых строк

Ситуация, в которой в файлах данных обнаруживаются дополнительные пустые строки, может оказаться трудно разрешимой. Часто приходится сталкиваться с файлами данных, которые должны содержать пустые строки, но иногда обнаруживается, что в одних местах требуемые пустые строки данных пропущены, а в других находится слишком много пустых строк (как в приведенном выше примере вывода через два интервала).

Наиболее простой способ удаления подряд идущих пустых строк состоит в том, чтобы проводить проверку потока данных с использованием диапазона адресов. В главе 18 показано, как использовать диапазоны адресов, в том числе, как включать шаблоны в диапазон адресов. Редактор `sed` выполняет команду для всех строк, сопоставляемых в пределах указанного диапазона адресов.

Ключом к решению задачи удаления подряд идущих пустых строк становится создание диапазона адресов, который включает непустую строку и пустую строку. При обнаружении редактором `sed` такого диапазона пустая строка не должна удаляться. Если же встречаются несколько строк, не сопоставляемых с этим диапазоном (в которых находятся две или большее количество подряд идущих пустых строк), пустые строки должны быть удалены.

Ниже приведен сценарий, позволяющий воплотить эту идею.

```
./.,/^$/!d
```

Диапазон охватывает строки от `./` до `^$/`. Начальный адрес в этом диапазоне сопоставляется с любой строкой, которая содержит по крайней мере один символ. Конечный адрес в диапазоне сопоставляется с пустой строкой. Строки в пределах такого диапазона не удаляются.

Ниже показано, как действует этот сценарий.

```
$ cat data6
This is the first line.

This is the second line.

This is the third line.


This is the fourth line.
$
$ sed '/./,/^$/!d' data6
This is the first line.

This is the second line.

This is the third line.

This is the fourth line.
$
```

Независимо от того, сколько пустых строк появляется между строками данных в файле, в выводе между строками находится лишь по одной пустой строке.

Удаление пустых строк в начале файла

Нарушением формата файла может стать также наличие пустых строк в начале файла данных. Часто происходит так, что при попытке импортировать данные из текстового файла в базу данных на основе пустых строк создаются нулевые записи, поэтому работа алгоритмов вычисления на основе данных в базе данных нарушается.

Задача удаления пустых строк в начале потока данных является несложной. Сценарий, который выполняет эту функцию, приведен ниже.

```
./,,$!d
```

В данном сценарии используется диапазон адресов для определения того, какие строки удалены. Диапазон начинается со строки, которая содержит любой символ, и продолжается до конца потока данных. Любая строка в пределах этого диапазона не удаляется из вывода. Это означает, что удаляются все строки, находящиеся перед первой строкой, которая содержит символ.

Рассмотрим этот простой сценарий в действии:

```
$ cat data7

This is the first line.

This is the second line.
$
$ sed '/./,$!d' data7
This is the first line.

This is the second line.
$
```

В файле `test` перед строками данных содержатся две пустые строки. Сценарий успешно удаляет обе ведущие пустые строки, оставляя пустые строки в остальном массиве данных нетронутыми.

Удаление заключительных пустых строк

К сожалению, задача удаления заключительных пустых строк не столь проста, как удаление ведущих пустых строк. Аналогично тому, как обеспечивается вывод конца потока данных, для удаления пустых строк в конце потока данных требуются определенная изобретательность и применение циклов.

Прежде чем начать обсуждение этой темы, рассмотрим, как может выглядеть сценарий, применимый в качестве решения:

```
sed '{
:start
/^\\n*$/{{$d; N; b start }}
}'
```

На первых порах трудно представить себе, как действует этот сценарий. Обратите внимание на то, что в обычных фигурных скобках, которые обозначают сценарий, находятся и другие фигурные скобки. В данном случае вложенные фигурные скобки служат для группирования части команд, которые входят в состав общего командного сценария. Эта группа команд применяется к указанному шаблону адреса. Шаблон адреса сопоставляется с любой строкой, которая содержит только символ обозначения конца строки. При обнаружении такой строки для ее удаления вызывается команда `delete`, если это — последняя строка. Если обнаруженная строка не является последней, команда `N` присоединяет к ней следующую строку, а команда `branch` обеспечивает переход в начало цикла.

Ниже показано, как действует этот сценарий.

```
$ cat data8
This is the first line.
This is the second line.

$
$ sed '{
:start
/^\\n *$/{{$d ; N; b start }}
}' data8
This is the first line.
This is the second line.
$
```

Очевидно, что сценарий успешно удалил пустые строки в конце текстового файла.

Удаление дескрипторов HTML

В наше время часто приходится загружать текст с веб-сайта для его сохранения или использования в качестве данных в приложении. Но иногда в тексте, загруженном с веб-сайта, встречаются дескрипторы HTML, применяемые для форматирования данных. Если же требуются только данные, то возникает необходимость в удалении этих дескрипторов.

Стандартная веб-страница HTML содержит несколько разных типов дескрипторов HTML, представляющих различные средства форматирования, которые применяются для отображения информации страницы должным образом. Ниже приведен пример того, как выглядит файл HTML.

```
$ cat data9
<html>
<head>
<title>This is the page title</title>
</head>
<body>
<p>
This is the <b>first</b> line in the Web page. This should provide
some <i>useful</i> information for us to use in our shell script.
</body>
</html>
$
```

Дескрипторы HTML обозначаются знаками “меньше” и “больше”. Большинство дескрипторов HTML являются парными. Один дескриптор иницирует процесс форматирования (например, дескриптор `` служит для выделения полужирным шрифтом), а другой останавливает процесс форматирования (например, дескриптор `` предназначен для отмены выделения полужирным шрифтом).

Однако при удалении дескрипторов HTML необходимо соблюдать осторожность, поскольку в противном случае может возникнуть проблема. На первый взгляд можно подумать, что для удаления дескрипторов HTML достаточно обеспечить поиск текстовой строки, начинающейся со знака “меньше” (`<`), заканчивающейся знаком “больше” (`>`) и содержащей данные между этими знаками:

```
s/<.*>/g
```

К сожалению, применение этой команды приводит к возникновению некоторых непредвиденных последствий:

```
$ sed 's/<.*>/g' data9
```

```
This is the  line in the Web page. This should provide
some  information for us to use in our shell script.
```

```
$
```

Обратите внимание на то, что пропущен сам текст заголовка, а также текст, выделенный полужирным шрифтом и курсивом. Редактор `sed` интерпретирует этот сценарий буквально, как требующий удаления всего текста между знаками “меньше” и “больше”, включая все прочие знаки “меньше” и “больше”! Каждый раз при обнаружении текста, включенного в дескрипторы HTML (такие как `first`), сценарий `sed` удалял весь текст.

Решение этой проблемы состоит в обеспечении того, чтобы редактор `sed` пропускал все внутренние знаки “больше” между исходными дескрипторами. Для этого можно создать класс символов, который представляет собой обращение знака “больше”. Сценарий, исправленный с учетом этого изменения, может выглядеть так:

```
s/<[^>]*>/g
```

Теперь сценарий работает должным образом и извлекает только те данные из кода HTML веб-страницы, которые представляют интерес:


```
$ sed 's/<[^>]*>//g' data9
```

```
This is the page title
```

```
This is the first line in the Web page. This should provide  
some useful information for us to use in our shell script.
```

```
$
```

Полученный результат стал более удобным для использования. Чтобы обеспечить дополнительную очистку данных, можно добавить команду `delete` для удаления пустых строк, которые создают лишь помехи в работе:

```
$ sed 's/<[^>]*>//g; /^\$/d' data9
```

```
This is the page title
```

```
This is the first line in the Web page. This should provide  
some useful information for us to use in our shell script.
```

```
$
```

Теперь полученные данные представлены намного более компактно; в них нет ничего лишнего.

Резюме

В редакторе `sed` предусмотрены некоторые усовершенствованные средства, которые позволяют работать с текстовыми шаблонами, охватывающими несколько строк. В настоящей главе показано, как использовать команду `next` для выборки следующей строки в потоке данных и размещения ее в пространстве шаблонов. После переноса данных в пространство шаблонов появляется возможность применять к данным сложные команды *substitution*, например, для замены фраз, разбитых на несколько строк текста.

Многострочная команда `delete` позволяет удалить первую строку, если пространство шаблонов содержит две или несколько строк. В этом состоит удобный способ обработки в цикле сразу нескольких строк из потока данных. Аналогичным образом многострочная команда `print` позволяет вывести только первую строку, если в пространстве шаблонов содержатся две или большее количество строк текста. Применение многострочных команд в сочетании позволяет выполнять итерации в потоке данных и создавать системы многострочной подстановки текста.

Далее в настоящей главе описано пространство хранения, которое позволяет “отложить в сторону” строку текста, продолжая обработку других строк текста. К содержимому пространства хранения можно снова обратиться в любое время, заменяя им текст в пространстве шаблонов или присоединяя это содержимое пространства хранения к тексту в пространстве шаблонов. Применение пространства хранения позволяет выполнять сортировку данных в потоке данных или изменять на противоположный порядок текстовых строк, появляющихся в данных.

В этой главе приведено также обсуждение команд управления ходом выполнения редактора `sed`. Команда `branch` предоставляет возможность изменения обычного хода выполнения команд редактора `sed` в сценарии, создания циклов или пропуска команд при определенных условиях. Команда `test` выполняет в командных сценариях редактора `sed` задачу, аналогичную инструкции `if-then`. Команда `test` выполняет переход, только если замена текста в строке с помощью предшествующей ей команды *substitution* прошла успешно.

В завершении настоящей главы приведено обсуждение того, как использовать сценарии `sed` в сценариях командного интерпретатора. Если сценарий `sed` состоит из большого количества команд, то обычно принято помещать его в оболочку командного интерпретатора. Предусмотрена возможность использовать в сценарии `sed` переменные параметров командной строки для передачи значений из командной строки командного интерпретатора. Таким образом, реализуется простой способ вызова сценариев редактора `sed` непосредственно из командной строки или даже из других сценариев командного интерпретатора.

В следующей главе более подробно рассматривается программа `gawk`, которая поддерживает много средств, характерных для высокоуровневых языков программирования. С помощью одной лишь программы `gawk` можно создавать некоторые довольно сложные программы манипулирования данными и формирования отчетов. В следующей главе будут описаны различные средства программирования и показано, как их использовать для формирования собственных отчетов с привлекательным оформлением из простых данных.

Дополнительные сведения о редакторе gawk

В главе 18 приведены вводные сведения о программе gawk и показаны основы ее использования для создания отформатированных отчетов по данным из файлов с данными, требующими дополнительной обработки. А в настоящей главе более подробно рассматривается настройка gawk в целях формирования отчетов. Программа gawk поддерживает полнофункциональный язык программирования и предоставляет средства, позволяющие разрабатывать сложные программы управления данными. Разработчики, владеющие другим языком программирования, которые впервые вступают в мир сценариев командного интерпретатора, почувствуют себя в знакомой обстановке, приобщившись к работе с программой gawk. В этой главе будет показано, как использовать язык программирования gawk для написания программ, позволяющих решать почти любые задачи форматирования данных, с которыми придется сталкиваться.

Использование переменных

Одним из важных средств любого языка программирования является способность сохранять и восстанавливать значения с использованием переменных. Язык программирования gawk поддерживает переменные двух типов:

- встроенные переменные;
- переменные, определяемые пользователем.

ГЛАВА

21

В этой главе...

Использование переменных

Работа с массивами

Использование шаблонов

Структурированные команды

Форматированный вывод

Встроенные функции

Определяемые пользователем функции

Резюме

Предусмотрен целый ряд встроенных переменных, предназначенных для использования в программе `gawk`. Встроенные переменные содержат информацию, применяемую при обработке полей данных и записей в файле данных. В программах `gawk` могут также создаваться пользовательские переменные. В следующих разделах описано, как использовать переменные в программах `gawk`.

Встроенные переменные

В программе `gawk` встроенные переменные используются для ссылки на конкретные средства, применяемые при работе в программе с данными. В настоящем разделе описаны встроенные переменные, доступные для использования в программах `gawk`, и показано, как с ними обращаться.

Переменные, применяемые в качестве разделителей полей и записей

В главе 18 были показаны встроенные переменные одного из типов, предусмотренных в `gawk`, — *переменные полей данных*. Переменные полей данных позволяют ссылаться на отдельные поля данных в записи данных с использованием знака доллара и числового обозначения положения поля данных в записи. Таким образом, для ссылки на первое поле данных в записи используется переменная `$1`. Чтобы сослаться на второе поле данных, можно применить переменную `$2` и т.д.

Границы между полями данных устанавливаются по символу разделителя полей. По умолчанию в качестве символа разделителя полей применяется любой *пробельный символ*, такой как пробел или знак табуляции. В главе 18 было показано, как изменить символ разделителя полей либо в командной строке с помощью параметра командной строки `-F`, либо в программе `gawk` с использованием специальной встроенной переменной `FS`.

Встроенная переменная `FS` относится к группе встроенных переменных, предназначенных для управления в программе `gawk` обработкой полей и записей во входных и выходных данных. В табл. 21.1 приведен перечень встроенных переменных, содержащихся в этой группе.

Таблица 21.1. Переменные полей и записей данных программы `gawk`

Переменная	Описание
<code>FIELDWIDTHS</code>	Разделенный пробелами список чисел, определяющих точную ширину (в пробелах) каждого поля данных
<code>FS</code>	Символ разделителя полей ввода
<code>RS</code>	Символ разделителя записей ввода
<code>OFS</code>	Символ разделителя полей вывода
<code>ORS</code>	Символ разделителя записей вывода

Переменные `FS` и `OFS` определяют, как в программе `gawk` происходит обработка полей данных в потоке данных. Выше в данной книге уже было показано, как использовать переменную `FS` для указания на то, какой символ служит для разделения полей данных в записи. Переменная `OFS` выполняет ту же функцию, но при выводе с использованием команды `print`.

По умолчанию в программе `gawk` в качестве значения переменной `OFS` задан пробел, поэтому при использовании, например, такой команды:

```
print $1, $2, $3
```

формируется следующий вывод:

```
field1 field2 field3
```

В этом можно убедиться на следующем примере:

```
$ cat data1
data11,data12,data13,data14,data15
data21,data22,data23,data24,data25
data31,data32,data33,data34,data35
$ gawk 'BEGIN{FS=","} {print $1,$2,$3}' data1
data11 data12 data13
data21 data22 data23
data31 data32 data33
$
```

Команда `print` автоматически помещает значение переменной `OFS` между каждым полем данных в выводе. Задавая переменную `OFS`, можно указать любую строку, которая должна служить для разделения полей данных в выводе:

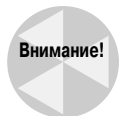
```
$ gawk 'BEGIN{FS=","; OFS="-"} {print $1,$2,$3}' data1
data11-data12-data13
data21-data22-data23
data31-data32-data33
$ gawk 'BEGIN{FS=","; OFS="--"} {print $1,$2,$3}' data1
data11--data12--data13
data21--data22--data23
data31--data32--data33
$ gawk 'BEGIN{FS=","; OFS="<-->"} {print $1,$2,$3}' data1
data11<-->data12<-->data13
data21<-->data22<-->data23
data31<-->data32<-->data33
$
```

Переменная `FIELDWIDTHS` позволяет считывать записи без использования символа разделителя полей. В некоторых приложениях не используется символ разделителя полей, а вместо этого данные располагаются в пределах записи по полям с фиксированной длиной, которые в совокупности записей образуют столбцы. В этих случаях должна быть задана переменная `FIELDWIDTHS`, которая отражает компоновку данных в записях.

Если переменная `FIELDWIDTHS` задана, программа `gawk` игнорирует значение `FS` и извлекает из записей значения данных с учетом указанных размеров полей. Ниже приведен пример использования полей с фиксированной длиной вместо полей, разграниченных символом разделителя.

```
$ cat data1b
1005.3247596.37
115-2.349194.00
05810.1298100.1
$ gawk 'BEGIN{FIELDWIDTHS="3 5 2 5"}{print $1,$2,$3,$4}' data1b
100 5.324 75 96.37
115 -2.34 91 94.00
058 10.12 98 100.1
$
```

Переменная `FIELDWIDTHS` определяет четыре поля данных, а программа `gawk` интерпретирует каждую запись данных соответствующим образом. Разбиение строки чисел в каждой записи осуществляется с учетом заданных значений длины полей.



Важно помнить, что после задания переменной `FIELDWIDTHS` указанные в ней значения должны оставаться постоянными. Этот способ обработки входных данных не может применяться в сочетании с вводом полей данных, имеющих переменную длину.

Переменные `RS` и `ORS` определяют, как программа `gawk` обрабатывает записи в потоке данных. По умолчанию в программе `gawk` в качестве значений переменных `RS` и `ORS` задан символ обозначения конца строки. Это заданное по умолчанию значение переменной `RS` указывает, что за каждым символом перевода на новую строку текста в потоке входных данных следует новая запись.

Иногда приходится сталкиваться с ситуациями, в которых поля данных разбиты на несколько строк в потоке данных. Классическим примером этого могут служить данные, содержащие поля с адресом и номером телефона, причем каждое из этих полей задано в отдельной строке:

```
Riley Mullen
123 Main Street
Chicago, IL 60601
(312) 555-1234
```

При попытке считать эти данные с использованием заданных по умолчанию значений переменных `FS` и `RS` программа `gawk` прочитает каждую строку как отдельную запись, интерпретируя каждый пробел в записи как разделитель полей. Это вряд ли допустимо в данном случае.

Для решения этой проблемы необходимо задать в качестве значения переменной `FS` символ обозначения конца строки. Это будет служить указанием, что каждая строка в потоке данных представляет собой отдельное поле и все данные в строке принадлежат к этому полю данных. Но в таком случае становится невозможно определить, где начинается новая запись.

Чтобы решить эту проблему, можно задать в качестве значения переменной `RS` пустую строку, а затем оставлять пустую строку между записями данных в потоке данных. Программа `gawk` будет интерпретировать каждую пустую строку как разделитель записей.

Ниже приведен пример использования такого способа организации входных данных.

```
$ cat data2
Riley Mullen
123 Main Street
Chicago, IL 60601
(312) 555-1234

Frank Williams
456 Oak Street
Indianapolis, IN 46201
(317) 555-9876

Haley Snell
4231 Elm Street
Detroit, MI 48201
(313) 555-4938
$ gawk 'BEGIN{FS="\n"; RS=""} {print $1,$4}' data2
Riley Mullen (312) 555-1234
Frank Williams (317) 555-9876
```

Haley Snell (313)555-4938
\$

Превосходный результат. Программа `gawk` интерпретировала каждую строку в файле как поле данных и пустые строки как разделители записей.

Переменные данных

Кроме переменных, которые задают значения разделителей полей и записей, в программе `gawk` предусмотрены некоторые другие встроенные переменные, позволяющие определить, что происходит с данными, и извлечь информацию из среды командного интерпретатора. В табл. 21.2 приведены дополнительные сведения о таких встроенных переменных `gawk`.

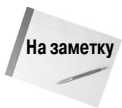
Таблица 21.2. Дополнительные сведения о встроенных переменных `gawk`

Переменная	Описание
ARGC	Количество представленных параметров командной строки
ARGIND	Индекс в ARGV текущего обрабатываемого файла
ARGV	Массив параметров командной строки
CONVFMT	Формат преобразования для чисел (см. инструкцию <code>printf</code>). Значением по умолчанию является <code>%.6g</code>
ENVIRON	Ассоциативный массив текущих переменных среды командного интерпретатора и их значений
ERRNO	Зафиксированная в системе ошибка, если при чтении или закрытии входных файлов произошла ошибка
FILENAME	Имя файла данных, используемого для ввода данных в программу <code>gawk</code>
FNR	Текущий номер записи в файле данных
IGNORECASE	Переменная, которая, будучи установленной в ненулевое значение, указывает, что должен игнорироваться регистр символов в строках, используемых в команде <code>gawk</code>
NF	Общее количество полей данных в файле данных
NR	Количество обработанных входных записей
OFMT	Формат вывода для отображения чисел. Значением по умолчанию является <code>%.6g</code>
RLENGTH	Длина подстроки, сопоставленной в функции <code>match</code>
RSTART	Индекс начала подстроки, сопоставленной в функции <code>match</code>

Читатель может заметить, что некоторые из этих переменных уже встречались при изучении программирования сценариев командного интерпретатора. Переменные `ARGC` и `ARGV` позволяют определять количество параметров командной строки и осуществлять выборку их значений из командного интерпретатора. Но иногда при решении этой задачи возникают сложности, поскольку в программе `gawk` сам программный сценарий не рассматривается как относящийся к параметрам командной строки:

```
$ gawk 'BEGIN{print ARGC,ARGV[1]}' data1
2 data1
$
```

Переменная `ARGC` указывает, что в командной строке имеются два параметра. В число этих параметров входят команда `gawk` и параметр `data1` (напомним, что сам программный сценарий не рассматривается как параметр). Массив `ARGV` начинается с индекса 0, который представляет команду. Значением элемента массива с индексом 1 является первый параметр командной строки после команды `gawk`.



Следует учитывать, что в отличие от переменных командного интерпретатора при ссылке на переменную `gawk` в сценарии не нужно добавлять знак доллара перед именем переменной.

При первом знакомстве с переменной `ENVIRON` приходится встречаться с чем-то непривычным. Дело в том, что в этой переменной для представления переменных среды командного интерпретатора используется *ассоциативный массив*. В ассоциативном массиве в качестве значений индексов массива вместо числовых значений применяются текстовые значения.

Текстовым значением индекса этого массива является имя переменной среды командного интерпретатора. Значение массива представляет собой значение переменной среды командного интерпретатора. Ниже приведен пример выполнения указанных действий.

```
$ gawk '  
> BEGIN{  
> print ENVIRON["HOME"]  
> print ENVIRON["PATH"]  
> }'  
/home/rich  
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin  
$
```

Переменная `ENVIRON["HOME"]` осуществляет выборку значения переменной среды `HOME` из командного интерпретатора. Аналогичным образом переменная `ENVIRON["PATH"]` позволяет получить значение переменной среды `PATH`. Такой же прием можно использовать для выборки любого значения переменной среды из командного интерпретатора в целях дальнейшего применения в программах `gawk`.

Переменные `FNR`, `NF` и `NR` позволяют отслеживать поля данных и записи, обрабатываемые в программе `gawk`. Иногда в программе приходится учитывать такие ситуации, когда точно не известно, сколько полей данных находится в записи. Переменная `NF` дает возможность указать на последнее поле данных в записи, не имея сведений о его действительном положении:

```
$ gawk 'BEGIN{FS=":"; OFS=":"} {print $1,$NF}' /etc/passwd  
rich:/bin/bash  
testy:/bin/csh  
mark:/bin/bash  
dan:/bin/bash  
mike:/bin/bash  
test:/bin/bash  
$
```

Переменная `NF` содержит числовое значение последнего поля данных в файле данных. Затем это значение можно применить в качестве переменной поля данных, поместив перед ним знак доллара.

Переменные `FNR` и `NR` аналогичны друг другу, но имеют небольшие различия. Переменная `FNR` содержит количество записей, обработанных в текущем файле данных. Переменная `NR` содержит общее количество обработанных записей. Рассмотрим ряд примеров, позволяющих понять это различие:

```
$ gawk 'BEGIN{FS=","}{print $1,"FNR="FNR}' data1 data1  
data11 FNR=1  
data21 FNR=2  
data31 FNR=3
```



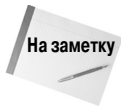
```
data11 FNR=1
data21 FNR=2
data31 FNR=3
$
```

В данном примере в командной строке программы `gawk` определены два входных файла. (В этой командной строке один и тот же входной файл указан дважды.) В сценарии происходит вывод первого значения поля данных и текущего значения переменной `FNR`. Заслуживает внимания то, что значение `FNR` снова сбрасывается в 1 с началом обработки в программе `gawk` второго файла данных.

Теперь применим дополнительно переменную `NR` и рассмотрим, к чему это приведет:

```
$ gawk '
> BEGIN {FS=","}
> {print $1,"FNR="FNR,"NR="NR}
> END{print "There were",NR,"records processed"}' data1 data1
data11 FNR=1 NR=1
data21 FNR=2 NR=2
data31 FNR=3 NR=3
data11 FNR=1 NR=4
data21 FNR=2 NR=5
data31 FNR=3 NR=6
There were 6 records processed
$
```

Значение переменной `FNR` было переустановлено после того, как в программе `gawk` начал обрабатываться второй файл данных, а переменная `NR` сохранила свое значение после перехода к этому второму файлу данных. Из этого следует вывод, что если в качестве входного используется только один файл данных, то значения `FNR` и `NR` остаются одинаковыми. Если в качестве входных используется несколько файлов данных, то значение `FNR` сбрасывается для каждого файла данных, а значение `NR` наращивается с учетом всех файлов данных.



При работе с программой `gawk` часто приходится замечать, что сценарии `gawk` имеют больший объем по сравнению с другими сценариями командного интерпретатора. В примерах настоящей главы в целях упрощения сценарии `gawk` выполняются лишь непосредственно из командной строки с использованием средства многострочного ввода командного интерпретатора. Если `gawk` используется в сценарии командного интерпретатора, следует помещать различные команды `gawk` на отдельных строках. Это позволяет намного упростить чтение и понимание сценария, поскольку не приходится разбираться в его содержимом, полностью размещенном в одной строке в сценарии командного интерпретатора. Кроме того, следует помнить, что если обнаруживается, что один и тот же сценарий `gawk` используется в разных сценариях командного интерпретатора, то можно сохранить его в отдельном файле, а затем сослаться на этот сценарий с применением параметра `-f` (см. главу 18).

Пользовательские переменные

Как и любой другой полноценный язык программирования, `gawk` позволяет определять собственные переменные для использования в коде программы. Имя определяемой пользователем переменной `gawk` может состоять из любого количества букв, цифр и знаков подчеркивания, но не может начинаться с цифры. Важно также помнить, что в именах переменных `gawk` учитывается регистр.

Присваивание значений переменным в сценариях

Значения переменным в программах `gawk` присваиваются так же, как в сценариях командного интерпретатора, с использованием *оператора присваивания*:

```
$ gawk '  
> BEGIN{  
> testing="This is a test"  
> print testing  
> }'  
This is a test  
$
```

Выводом инструкции `print` является текущее значение переменной `testing`. Как и переменные сценария командного интерпретатора, переменные `gawk` могут содержать числовые или строковые значения:

```
$ gawk '  
> BEGIN{  
> testing="This is a test"  
> print testing  
> testing=45  
> print testing  
> }'  
This is a test  
45  
$
```

В данном примере значение переменной `testing` изменяется, и вместо строкового значения в ней сохраняется числовое значение.

Операторы присваивания могут также включать математические выражения, в которых выполняются операции с числовыми значениями:

```
$ gawk 'BEGIN{x=4; x= x * 2 + 3; print x}'  
11  
$
```

Как показывает этот пример, язык программирования `gawk` включает стандартные математические операции для работы с числовыми значениями. В число этих операций входят вычисление остатка от деления (знак операции — `%`) и возведение в степень (знак операции `^` или `**`).

Присваивание значений переменным в командной строке

Для присваивания значений переменным в целях применения в программе `gawk` может также использоваться командная строка `gawk`. Это позволяет задавать значения вне обычного кода, изменяя при этом значения динамически. Ниже приведен пример использования переменной командной строки в целях выбора для отображения конкретного поля данных в файле.

```
$ cat script1  
BEGIN{FS=","}  
{print $n}  
$ gawk -f script1 n=2 data1  
data12  
data22
```

```
data32
$ gawk -f script1 n=3 data1
data13
data23
data33
$
```

Это средство позволяет изменять поведение сценария без необходимости вносить изменения в сам код сценария. В первом примере отображается второе поле данных в файле, а во втором примере — третье поле данных, при этом изменяется лишь значение переменной `n` в командной строке.

Но при использовании параметров командной строки для определения значений переменных следует учитывать наличие определенной проблемы. Дело в том, что при таком задании переменной ее значение не доступно в разделе `BEGIN` кода:

```
$ cat script2
BEGIN{print "The starting value is",n; FS=","}
{print $n}
$ gawk -f script2 n=3 data1
The starting value is
data13
data23
data33
$
```

Решение этой проблемы состоит в использовании параметра командной строки `-v`. Этот параметр позволяет определять переменные, значение которых задается перед разделом кода `BEGIN`. Параметр командной строки `-v` должен быть задан перед кодом сценария в командной строке:

```
$ gawk -v n=3 -f script2 data1
The starting value is 3
data13
data23
data33
$
```

Теперь переменная `n` содержит значение, которое определено в командной строке перед разделом кода `BEGIN`.

Работа с массивами

Понятие массивов как переменных, позволяющих хранить сразу несколько значений, определено во многих языках программирования. В языке программирования `gawk` предусмотрено средство работы с массивами в виде *ассоциативных массивов*.

Ассоциативные массивы отличаются от массивов с числовыми индексами тем, что в них в качестве значения индекса может быть задана любая строка текста. Для ссылки на элементы данных, содержащиеся в массиве, не обязательно требуется использовать последовательно возрастающие числа. Вместо этого в качестве ключей ассоциативного массива применяются произвольные строки, которые служат ссылками на значения в массиве. Каждая строка индекса должна быть уникальной; индексы однозначно определяют элементы данных, которые сохра-

нены под этими индексами. Специалисты, знакомые с другими языками программирования, могут понять, что по своему принципу организации ассоциативные массивы аналогичны *хеши-массивам*, или *словарям*.

В следующих разделах кратко описано, как переменные ассоциативного массива могут использоваться в программах *gawk*.

Определение переменных с типом массива

Переменную с типом массива можно определить с помощью стандартного оператора присваивания. Формат присваивания значения переменной с типом массива является следующим:

```
var[index] = element
```

где *var* — имя переменной; *index* — значение индекса ассоциативного массива; *element* — значение элемента данных. Ниже приведены некоторые примеры использования переменных с типом массива в программе *gawk*.

```
capital["Illinois"] = "Springfield"
capital["Indiana"] = "Indianapolis"
capital["Ohio"] = "Columbus"
```

При ссылке на переменную с типом массива необходимо задавать значение индекса, чтобы получить соответствующее значение элемента данных:

```
$ gawk 'BEGIN{
> capital["Illinois"] = "Springfield"
> print capital["Illinois"]
> }'
Springfield
$
```

Применение ссылки на переменную с типом массива приводит к возврату значения элемента данных. Аналогичная операция является осуществимой и по отношению к числовым значениям элементов данных:

```
$ gawk 'BEGIN{
> var[1] = 34
> var[2] = 3
> total = var[1] + var[2]
> print total
> }'
37
$
```

Как показывает этот пример, переменные с типом массива могут использоваться в программе *gawk* точно так же, как и переменные любых других типов.

Обработка переменных с типом массива в цикле

При использовании переменных с типом ассоциативного массива возникает проблема, связанная с тем, что не всегда известны все значения индексов. В отличие от массивов с числовыми индексами, в которых для индексирования значений используются последовательно возрастающие числа, индексы ассоциативного массива могут иметь произвольные строковые или числовые значения.

Если возникает необходимость провести обработку в цикле ассоциативного массива в программе `gawk`, можно воспользоваться специальным форматом инструкции `for`:

```
for (var in array)
{
    statements
}
```

В инструкции `for` включенные в нее инструкции `statements` обрабатываются в цикле, и при этом каждый раз переменной `var` присваивается следующее значение индекса из ассоциативного массива `array`. Важно помнить, что значением этой переменной является значение индекса, а не значение элемента данных. Указанная переменная позволяет легко извлечь значение элемента данных с применением ее в качестве индекса массива:

```
$ gawk 'BEGIN{
> var["a"] = 1
> var["g"] = 2
> var["m"] = 3
> var["u"] = 4
> for (test in var)
> {
>     print "Index:",test," - Value:",var[test]
> }
> }'
Index: u - Value: 4
Index: m - Value: 3
Index: a - Value: 1
Index: g - Value: 2
$
```

Следует отметить, что не происходит возврат значений индексов в каком-то определенном порядке, но каждый из этих индексов ссылается на соответствующее значение элемента данных. Иногда эту особенность важно учитывать при написании программы, поскольку нельзя полагаться на то, что возвращаемые значения будут всегда представлены в одном и том же порядке; неизменным является лишь соответствие значений отдельных индексов и значений данных, на которые указывают эти индексы.

Удаление переменных с типом массива

Для удаления определенного индекса массива из ассоциативного массива необходимо применять специальную команду:

```
delete array[index]
```

Команда `delete` удаляет из массива `array` значение индекса ассоциативного массива `index` и связанное с ним значение элемента данных:

```
$ gawk 'BEGIN{
> var["a"] = 1
> var["g"] = 2
> for (test in var)
> {
>     print "Index:",test," - Value:",var[test]
> }
> }
```

```

> delete var["g"]
> print "---"
> for (test in var)
>   print "Index:",test," - Value:",var[test]
> }'
Index: a - Value: 1
Index: g - Value: 2
---
Index: a - Value: 1
$

```

Восстановление значения индекса после его удаления из ассоциативного массива невозможно.

Использование шаблонов

В программе gawk поддерживаются шаблоны сопоставления нескольких типов, с помощью которых можно отфильтровывать записи данных во многом по такому же принципу, как в редакторе sed. В главе 18 уже были показаны в действии два специальных шаблона. К специальным шаблонам относятся ключевые слова BEGIN и END, которые выполняют инструкции до и после чтения данных из потока данных. Аналогичным образом можно создавать другие шаблоны для выполнения определенных инструкций при обнаружении сопоставляемых с шаблонами данных в потоке данных.

В настоящем разделе показано, как использовать шаблоны сопоставления в сценариях gawk для задания ограничений на то, к каким записям применяется программный сценарий.

Регулярные выражения

В главе 19 было показано, как использовать регулярные выражения в качестве шаблонов сопоставления. Для задания фильтров, определяющих, к каким строкам данных в потоке данных должен применяться программный сценарий, может использоваться формат базовых регулярных выражений (Basic Regular Expression — BRE) или формат расширенных регулярных выражений (Extended Regular Expression — ERE).

Если используется регулярное выражение, то оно должно быть приведено перед левой фигурной скобкой управляемого им программного сценария:

```

$ gawk 'BEGIN{FS=","} /11/{print $1}' data1
data11
$

```

Регулярное выражение /11/ сопоставляется с записями, которые содержат подстроку 11 в любом месте в полях данных. Программа gawk сопоставляет заданное регулярное выражение со всеми полями данных в записи, включая символ разделителя полей:

```

$ gawk 'BEGIN{FS=","} /,d/{print $1}' data1
data11
data21
data31
$

```

В данном примере в регулярном выражении происходит сопоставление с запятой, используемой в качестве разделителя полей. Это не всегда является приемлемым. Проблемы могут также возникнуть при попытке сопоставления с данными, которые относятся к какому-то конкретному полю данных, но могут появиться и в другом поле данных. Если необходимо сопоставить регулярное выражение с конкретным экземпляром данных, то следует использовать оператор сопоставления.

Оператор сопоставления

Оператор сопоставления позволяет ограничить применение регулярного выражения лишь конкретным полем данных в записях. В качестве оператора сопоставления применяется знак тильды (~). При этом необходимо задать оператор сопоставления наряду с переменной поля данных и регулярным выражением, применяемым для сопоставления:

```
$1 ~ /^data/
```

Переменная \$1 представляет первое поле данных в записи. Это выражение служит для фильтрации записей, в которых первое поле данных начинается с текстовых данных. Ниже приведен пример применения оператора сопоставления в сценарии программы gawk.

```
$ gawk 'BEGIN{FS=","} $2 ~ /^data2/{print $0}' data1
data21,data22,data23,data24,data25
$
```

В операторе сопоставления второе поле данных сравнивается с регулярным выражением /^data2/, которое указывает, что строка начинается с текста data2.

Это — мощное инструментальное средство, которое широко используется в программных сценариях gawk для поиска конкретных элементов данных в файле данных:

```
$ gawk -F: '$1 ~ /rich/{print $1,$NF}' /etc/passwd
rich /bin/bash
$
```

В этом примере происходит поиск текста rich в первом поле данных. Если этот шаблон обнаруживается в записи, то выводятся значения первого и последнего полей данных в записи.

Предусмотрена также возможность обратить сопоставление с регулярным выражением с помощью символа !:

```
$1 !~ /expression/
```

Программный сценарий применяется к данным записи, если строка, соответствующая регулярному выражению, не обнаруживается в записи:

```
$ gawk -F: '$1 !~ /rich/{print $1,$NF}' /etc/passwd
root /bin/bash
daemon /bin/sh
bin /bin/sh
sys /bin/sh
--- вывод сокращен ---
$
```

В данном примере программный сценарий gawk применяется для вывода идентификатора пользователя и имени командного интерпретатора по результатам обработки всех записей в файле /etc/passwd, не сопоставляемых с идентификатором пользователя rich!

Математические выражения

В шаблоне сопоставления могут применяться не только регулярные, но и математические выражения. Данное средство становится применимым, если возникает необходимость обеспечить сопоставление с числовыми значениями в полях данных. Например, предположим, что необходимо вывести сведения обо всех пользователях системы, принадлежащих к группе пользователей root (к группе с номером 0). Для этого может использоваться следующий сценарий:

```
$ gawk -F: '$4 == 0{print $1}' /etc/passwd
root
sync
shutdown
halt
operator
$
```

В этом сценарии осуществляется проверка на наличие записей, в которых четвертое поле данных содержит значение 0. В рассматриваемой системе Linux имеются пять учетных записей пользователей, которые принадлежат к группе пользователей root.

Предусмотрена возможность использовать в сравнениях любые из следующих обычных математических выражений:

- `x == y`. Значение `x` равно `y`.
- `x <= y`. Значение `x` меньше или равно `y`.
- `x < y`. Значение `x` меньше `y`.
- `x >= y`. Значение `x` больше или равно `y`.
- `x > y`. Значение `x` больше `y`.

Можно также использовать выражения с текстовыми данными, но при этом необходимо соблюдать осторожность. В отличие от регулярных выражений, в таких выражениях должно обеспечиваться точное соответствие. Данные должны точно сопоставляться с шаблоном:

```
$ gawk -F, '$1 == "data"{print $1}' data1
$
$ gawk -F, '$1 == "data1"{print $1}' data1
data1
$
```

В первой проверке не происходит сопоставление никаких записей, поскольку значение первого поля данных не содержит каких-либо данных любой из записей. Вторая проверка приводит к сопоставлению с одной записью со значением `data1`.

Структурированные команды

Язык программирования `gawk` поддерживает обычный набор команд структурированного программирования. В настоящем разделе описана каждая из этих команд и показано, как их использовать в среде программирования `gawk`.

Оператор if

В языке программирования gawk поддерживается стандартный формат if-then-else инструкции if. Для инструкции if необходимо определить проверяемое условие *condition*, задавая его в круглых скобках. Если проверка условия приводит к получению значения TRUE, то выполняется инструкция *statement*, которая непосредственно следует за инструкцией if. Если значением условия является FALSE, то следующая инструкция пропускается. При этом может использоваться такой формат:

```
if (condition)
    statement1
```

Этот код может быть также размещен на одной строке, примерно так:

```
if (condition) statement1
```

Ниже приведен простой пример, демонстрирующий этот формат.

```
$ cat data4
10
5
13
50
34
$ gawk '{if ($1 > 20) print $1}' data4
50
34
$
```

Очевидно, что в этом нет ничего сложного. Если по условию инструкции if необходимо выполнить несколько инструкций, то их следует заключить в фигурные скобки:

```
$ gawk '{
> if ($1 > 20)
> {
>   x = $1 * 2
>   print x
> }
> }' data4
100
68
$
```

Необходимо учитывать, что фигурные скобки инструкции if не следует путать с фигурными скобками, используемыми для обозначения начала и конца программного сценария. Программа gawk может обнаружить пропущенные фигурные скобки и выдать сообщение об ошибке, если в сценарии расстановка скобок выполнена неправильно:

```
$ gawk '{
> if ($1 > 20)
> {
>   x = $1 * 2
>   print x
> }' data4
gawk: cmd. line:7: (END OF FILE)
```

```
gawk: cmd. line:7: parse error
$
```

Инструкция `if` языка `gawk` также поддерживает предложение `else`, что позволяет выполнить одну или несколько инструкций, если условие инструкции `if` примет ложное значение. Ниже приведен пример использования предложения `else`.

```
$ gawk '{
> if ($1 > 20)
> {
>   x = $1 * 2
>   print x
> } else
> {
>   x = $1 / 2
>   print x
> } }' data4
5
2.5
6.5
100
68
$
```

Предложение `else` может быть задано в одной строке, но тогда после раздела с инструкцией `if` необходимо вводить точку с запятой:

```
if (condition) statement1; else statement2
```

Ниже приведен такой же пример, в котором используется формат с одной строкой.

```
$ gawk '{if ($1 > 20) print $1 * 2; else print $1 / 2}' data4
5
2.5
6.5
100
68
$
```

Применение этого формата приводит к получению меньшего объема кода, однако изучение такого кода может стать более затруднительным.

Инструкция `while`

Инструкция `while` представляет собой основное средство организации циклов в программе `gawk`. Формат инструкции `while` приведен ниже.

```
while (condition)
{
    statements
}
```

Цикл `while` позволяет проводить итерацию по набору данных, проверяя условие *condition*, которое может привести к прекращению работы цикла. Такая организация работы становится удобной, если в каждой записи имеется несколько значений данных, которые необходимо использовать в вычислениях:

```

$ cat data5
130 120 135
160 113 140
145 170 215
$ gawk '{
> total = 0
> i = 1
> while (i < 4)
> {
>     total += $i
>     i++
> }
> avg = total / 3
> print "Average:",avg
> }' data5
Average: 128.333
Average: 137.667
Average: 176.667
$

```

С помощью инструкции `while` проводится итерация по полям данных в записи, каждое значение складывается со значением итоговой переменной, а затем происходит увеличение значения переменной счетчика, `i`. После того как значение счетчика становится равным 4, условие `while` принимает значение `FALSE` и работа цикла прекращается, вслед за чем происходит переход к следующей инструкции в сценарии. В этой инструкции вычисляется среднее значение; затем среднее значение выводится. Этот процесс повторяется для каждой записи в файле данных.

Язык программирования `gawk` поддерживает использование инструкций `break` и `continue` в циклах `while`, что позволяет в случае необходимости прекратить выполнение цикла:

```

$ gawk '{
> total = 0
> i = 1
> while (i < 4)
> {
>     total += $i
>     if (i == 2)
>         break
>     i++
> }
> avg = total / 2
> print "The average of the first two data elements is:",avg
> }' data5
The average of the first two data elements is: 125
The average of the first two data elements is: 136.5
The average of the first two data elements is: 157.5
$

```

В данном случае инструкция `break` используется для выхода из цикла `while`, если значение переменной `i` становится равным 2.

Инструкция do-while

Инструкция `do-while` аналогична инструкции `while`, но предусматривает выполнение содержащихся в ней инструкций перед проверкой условия. Формат инструкции `do-while` приведен ниже.

```
do
{
    statements
} while (condition)
```

Данный формат обеспечивает выполнение инструкций по меньшей мере один раз до проверки условия. Такая организация работы является удобной, если необходимо выполнить какие-либо инструкции и лишь затем проверить условие:

```
$ gawk '{
> total = 0
> i = 1
> do
> {
>     total += $i
>     i++
> } while (total < 150)
> print total }' data5
250
160
315
$
```

В этом сценарии считываются поля данных в каждой записи и их сумма вычисляется до тех пор, пока совокупное значение не достигнет 150. Даже если значение первого поля данных превышает 150 (как показано во второй записи), гарантируется, что в сценарии по крайней мере произойдет чтение первого поля данных и лишь затем будет выполнена проверка условия.

Инструкция for

Инструкция `for` применяется для организации циклов во многих языках программирования. Язык программирования `gawk` поддерживает циклы `for` в стиле C:

```
for( variable assignment; condition; iteration process)
```

Благодаря этому создание цикла становится проще в связи с тем, что с помощью одной инструкции решается сразу несколько задач:

```
$ gawk '{
> total = 0
> for (i = 1; i < 4; i++)
> {
>     total += $i
> }
> avg = total / 3
> print "Average:", avg
> }' data5
Average: 128.333
```

Average: 137.667
Average: 176.667
\$

Поскольку счетчик итераций определен в цикле `for`, не приходится заботиться о том, чтобы наращивать его самому, как при использовании инструкции `while`.

Форматированный вывод

Читатель мог заметить, что оператор `print` не позволяет получить достаточно большой контроль над тем, как происходит вывод данных в программе `gawk`. Возможности управления выводом почти полностью ограничиваются применением символа-разделителя выходных полей (OFS). Но при создании подробных отчетов часто возникает необходимость размещать данные в определенных форматах и задавать местоположение данных.

Решением в этой ситуации становится применение команды форматирования вывода, называемой `printf`. Для специалистов, знакомых с программированием на языке C, можно указать, что команда `printf` в программе `gawk` действует аналогичным образом, позволяя задавать подробные указания, касающиеся того, как должны быть отображены данные.

Формат команды `printf` приведен ниже.

```
printf "format string", var1, var2 . . .
```

Ключом к получению форматированного вывода является *строка формата*, *format string*. Строка формата точно задает, как должен выглядеть форматированный вывод, для чего применяются текстовые элементы и так называемые *спецификаторы формата*. Спецификатор формата — это специальный код, который указывает, к какому типу относится выводимое значение переменной и как вывести это значение. В программе `gawk` каждый спецификатор формата используется в качестве местозаполнителя для соответствующих переменных, перечисленных в команде. Первый спецификатор формата сопоставляется с первой переменной в списке, второй спецификатор — со второй переменной и т.д.

Для задания спецификаторов формата используется следующий формат:

```
%[modifier]control-letter
```

где *control-letter* — односимвольный код, который указывает, какое значение данных должно быть отображено, а значение *modifier* определяет необязательное средство форматирования.

В табл. 21.3 перечислены символы управления, которые могут использоваться в спецификаторе формата.

Таблица 21.3. Символы управления спецификатора формата

Символ управления	Описание
c	Отображает число в виде символа ASCII
d	Отображает целочисленное значение
i	Отображает целочисленное значение (то же, что и d)
e	Отображает число в экспоненциальном представлении чисел
f	Отображает значение с плавающей точкой
g	Отображает число в экспоненциальном представлении чисел или число с плавающей точкой, в зависимости от того, какое представление короче
o	Отображает восьмеричное значение

Символ управления	Описание
s	Отображает текстовую строку
x	Отображает шестнадцатеричное значение
X	Отображает шестнадцатеричное значение, но с использованием прописных букв от А до F

Таким образом, чтобы отобразить строковую переменную, следует использовать спецификатор формата %s. Если необходимо отобразить целочисленную переменную, можно воспользоваться спецификатором формата %d или %i (%d — спецификатор формата для десятичных чисел в стиле языка C). Чтобы отобразить большое значение с использованием экспоненциального представления чисел, следует применить спецификатор формата %e:

```
$ gawk 'BEGIN{
> x = 10 * 100
> printf "The answer is: %e\n", x
> }'
The answer is: 1.000000e+03
$
```

Кроме этих символов управления спецификатором формата, можно использовать три модификатора, которые позволяют добиться еще большего контроля над выводом.

- **width.** Числовое значение, которое указывает минимальную ширину выходного поля. Если выводимое значение короче, команда `printf` вводит дополнительные пробелы, используя для текста выравнивание вправо. Если выводимое значение имеет представление с длиной больше указанной, то значение `width` переопределяется.
- **prec.** Числовое значение, которое указывает количество цифр справа от десятичной точки в числах с плавающей точкой или максимальное количество символов, отображаемых в текстовой строке.
- **- (знак “минус”).** Указывает, что вместо выравнивания вправо должно применяться выравнивание влево при помещении данных в пространство форматирования.

Применение инструкции `printf` позволяет получить полный контроль над тем, какой формат будут иметь выводимые данные. В качестве примера отметим, что в разделе “Встроенные переменные” для отображения полей данных из указанных записей использовалась команда `print`:

```
$ gawk 'BEGIN{FS="\n"; RS="" } {print $1,$4}' data2
Riley Mullen (312)555-1234
Frank Williams (317)555-9876
Haley Snell (313)555-4938
$
```

Для форматирования вывода, чтобы он выглядел лучше, может использоваться команда `printf`. Прежде всего просто заменим команду `print` командой `printf`, чтобы определить, в чем различие между этими командами:

```
$ gawk 'BEGIN{FS="\n"; RS="" } {printf "%s %s\n", $1, $4}' data 2
Riley Mullen (312)555-1234
Frank Williams (317)555-9876
Haley Snell (313)555-4938
$
```

В данном случае формируется такой же вывод, как и с помощью команды `print`. В команде `printf` используется спецификатор формата `%s` в качестве местозаполнителя для этих двух строковых значений.

Следует отметить, что нам пришлось вручную добавить символ обозначения конца строки в конце команды `printf`, чтобы обеспечить переход на новую строку после вывода текущей строки. Без этого в команде `printf` продолжала бы использоваться для вывода одна и та же строка во всех последующих операциях вывода.

Этот вариант также может применяться, если необходимо вывести несколько элементов данных в одной и той же строке, но с использованием отдельных команд `printf`:

```
$ gawk 'BEGIN{FS=","} {printf "%s ", $1} END{printf "\n"}' data1
data11 data21 data31
$
```

Все элементы данных, выведенные командой `printf`, появляются в одной строке. Чтобы можно было завершить строку, в разделе `END` выводится отдельный символ обозначения конца строки.

Теперь рассмотрим, как использовать модификатор для форматирования первого строкового значения:

```
$ gawk 'BEGIN{FS="\n"; RS=""} {printf "%16s %s\n", $1, $4}' data2
Riley Mullen   (312)555-1234
Frank Williams (317)555-9876
Haley Snell    (313)555-4938
$
```

Добавление модификатора со значением 16 приводит к тому, что для вывода первой строки используется поле с шириной 16 пробелов. По умолчанию в команде `printf` используется выравнивание вправо при размещении данных в пространстве, предусмотренном в формате. Чтобы обеспечить выравнивание значения влево, достаточно только добавить к модификатору знак “минус”:

```
$ gawk 'BEGIN{FS="\n"; RS=""} {printf "%-16s %s\n", $1, $4}' data2
Riley Mullen      (312)555-1234
Frank Williams    (317)555-9876
Haley Snell       (313)555-4938
$
```

Очевидно, что теперь вывод выглядит намного профессиональнее!

С помощью команды `printf` можно также легко обеспечить вывод значений с плавающей точкой. Определив формат для переменной, можно добиться формирования более единообразного вывода:

```
$ gawk '{
> total = 0
> for (i = 1; i < 4; i++)
> {
>   total += $i
> }
> avg = total / 3
> printf "Average: %5.1f\n", avg
> }' data5
Average: 128.3
Average: 137.7
```

Average: 176.7
\$

С использованием спецификатора формата `%5.1f` можно обеспечить применение команды `printf` для округления значения с плавающей точкой до одной десятичной позиции.

Встроенные функции

В языке программирования `gawk` предусмотрено довольно много встроенных функций, которые предназначены для выполнения математических вычислений, действий со строками и даже операций со значениями времени. Эти функции можно непосредственно использовать в программах `gawk` вместо того, чтобы выполнять предусмотренные ими действия с помощью кода сценариев. В настоящем разделе описаны различные встроенные функции, которые под-держиваются в программе `gawk`.

Математические функции

Специалисты, которые когда-либо занимались программированием на другом языке лю-бого типа, скорее всего, знакомы с тем, как использовать встроенные функции в коде для вы-полнения обычных математических вычислений. Язык программирования `gawk` не разочарует даже тех, кому приходится использовать довольно развитые математические средства.

В табл. 21.4 приведены встроенные математические функции, предусмотренные в языке `gawk`.

Таблица 21.4. Математические функции, применяемые в языке `gawk`

Функция	Описание
<code>atan2(x, y)</code>	Арктангенс отношения x/y , в котором величины x и y заданы в радианах
<code>cos(x)</code>	Косинус x , в котором величина x задана в радианах
<code>exp(x)</code>	Экспонента значения x
<code>int(x)</code>	Целочисленная часть значения x , усеченная в сторону 0
<code>log(x)</code>	Натуральный логарифм значения x
<code>rand()</code>	Случайное число с плавающей точкой со значением больше 0 и меньше 1
<code>sin(x)</code>	Синус x , в котором величина x задана в радианах
<code>sqrt(x)</code>	Квадратный корень величины x
<code>srand(x)</code>	Начальное значение для вычисления случайных чисел

Перечень математических функций, применимых в программе `gawk`, не слишком велик, но в языке `gawk` предусмотрены некоторые основные элементы, необходимые для осуществления типичных математических вычислений. Функция `int()` служит для получения целочисленной части значения, но округление при этом не происходит. По своему действию эта функция во многом напоминает функцию `floor`, применяемую в других языках программирования. Эта функция позволяет получить целое число между 0 и значением аргумента, ближайшее к этому значению.

Это означает, что применение функции `int()` к значению 5.6 приведет к получению числа 5, а вызов функции `int()` с аргументом -5.6 приведет к получению числа -5.

Функция `rand()` великолепно подходит для выработки случайных чисел, но необходимо проводить определенную подготовку для получения значений в требуемом диапазоне. Дело в том, что случайные числа, возвращаемые функцией `rand()`, находятся в диапазоне только от 0 до 1 (не включая 0 или 1). Для получения чисел с большими значениями необходимо масштабировать возвращаемые значения.

Общепринятый метод получения более крупных целочисленных случайных значений состоит в применении алгоритма, в котором используется функция `rand()` наряду с функцией `int()`:

```
x = int(10 * rand())
```

При этом происходит возврат случайных целочисленных значений от 0 до 9 включительно. Достаточно лишь подставить число 10 в уравнение с верхним значением предела для приложения, и генератор случайных чисел готов к использованию.

Но при использовании некоторых математических функций необходимо соблюдать осторожность, поскольку в языке программирования `gawk` диапазон числовых значений, с которыми можно работать, является ограниченным. Выход за пределы этого диапазона приводит к получению сообщения об ошибке:

```
$ gawk 'BEGIN{x=exp(100); print x}'
26881171418161356094253400435962903554686976
$ gawk 'BEGIN{x=exp(1000); print x}'
gawk: warning: exp argument 1000 is out of range
inf
$
```

В первом примере вычисляется результат возведения в степень числа 100, который действительно очень велик, но не выходит за пределы допустимого диапазона в системе. Во втором примере предпринимается попытка вычислить результат возведения в степень числа 1000, что приводит к выходу за пределы допустимого диапазона числовых значений в системе и формированию сообщения об ошибке.

Кроме стандартных математических функций, в языке `gawk` предусмотрено также несколько функций для манипулирования с двоичными представлениями данных.

- `and(v1, v2)`. Выполняет побитовую операцию AND применительно к значениям `v1` и `v2`.
- `compl(val)`. Выполняет операцию побитового дополнения значения `val`.
- `lshift(val, count)`. Смещает битовое представление значения `val` влево на количество битов, указанное параметром `count`.
- `or(v1, v2)`. Выполняет побитовую операцию OR применительно к значениям `v1` и `v2`.
- `rshift(val, count)`. Смещает битовое представление значения `val` вправо на количество битов, указанное параметром `count`.
- `xor(v1, v2)`. Выполняет побитовую операцию XOR применительно к значениям `v1` и `v2`.

Функции побитовой обработки предназначены для работы с двоичными представлениями данных.

Строковые функции

Как показано в табл. 21.5, в языке программирования `gawk` представлено также несколько функций, которые можно использовать для манипулирования строковыми значениями.

Таблица 21.5. Строковые функции языка gawk

Функция	Описание
<code>asort(s[,d])</code>	Обеспечивает сортировку массива <i>s</i> с учетом значений элементов данных. Значения индекса заменяются последовательно возрастающими числами, обозначающими новый порядок сортировки. Еще один вариант состоит в том, что новый отсортированный массив может быть сохранен в массиве <i>d</i> , если он задан
<code>asorti(s[,d])</code>	Обеспечивает сортировку массива <i>s</i> с учетом значений элементов индекса. Результирующий массив содержит значения индекса в качестве значений элементов данных, а последовательно возрастающие индексы указывают порядок сортировки. Еще один вариант состоит в том, что новый отсортированный массив может быть сохранен в массиве <i>d</i> , если он задан
<code>gensub(r, s, h[,t])</code>	Обеспечивает поиск сопоставлений с регулярным выражением <i>r</i> либо в переменной <code>\$0</code> , либо в целевой строке <i>t</i> , если она задана. Если <i>h</i> — строка, начинающаяся либо с <i>g</i> , либо с <i>G</i> , заменяет текст сопоставления <i>s</i> . Если <i>h</i> — число, оно используется как указание на то, какое вхождение <i>r</i> необходимо заменить
<code>gsub(r, s[,t])</code>	Обеспечивает поиск сопоставлений с регулярным выражением <i>r</i> либо в переменной <code>\$0</code> , либо в целевой строке <i>t</i> , если она задана. Если поиск окажется успешным, производит глобальную замену строки <i>s</i>
<code>index(s, t)</code>	Возвращает индекс строки <i>t</i> в строке <i>s</i> или 0, если поиск окончится неудачей
<code>length([s])</code>	Возвращает длину строки <i>s</i> , если же эта строка не указана, возвращает длину <code>\$0</code>
<code>match(s, r[,a])</code>	Возвращает индекс строки <i>s</i> , в которой обнаруживается шаблон регулярного выражения <i>r</i> . Если задан массив <i>a</i> , он содержит часть <i>s</i> , сопоставленную с регулярным выражением
<code>split(s, a[,r])</code>	Выполняет разбиение строки <i>s</i> на элементы массива <i>a</i> с использованием либо символа FS, либо регулярного выражения <i>r</i> , если оно задано. Возвращает количество полей
<code>sprintf(format, variables)</code>	Возвращает строку, аналогичную той, вывод которой осуществляется с помощью функции <code>printf</code> , для чего используются заданные параметры <i>format</i> и <i>variables</i>
<code>sub(r, s[,t])</code>	Обеспечивает поиск сопоставлений с регулярным выражением <i>r</i> либо в переменной <code>\$0</code> , либо в целевой строке <i>t</i> . Если поиск окажется успешным, производит подстановку строки <i>s</i> вместо первого вхождения
<code>substr(s, i[,n])</code>	Возвращает подстроку строки <i>s</i> с длиной <i>n</i> символов, начиная с индекса <i>i</i> . Если значение <i>n</i> не задано, используется остальная часть <i>s</i>
<code>tolower(s)</code>	Преобразовывает все символы в строке <i>s</i> в нижний регистр
<code>toupper(s)</code>	Преобразовывает все символы в строке <i>s</i> в верхний регистр

Очевидно, что часть этих строковых функций не требует особых пояснений:

```
$ gawk 'BEGIN{x = "testing"; print toupper(x); print length(x) }'
TESTING
7
$
```

Но при работе с некоторыми другими строковыми функциями могут возникать затруднения. Функции `asort` и `asorti` — это новые функции языка gawk, которые позволяют сортировать переменные типа массива с учетом либо значений элементов данных (`asort`), либо значений индекса (`asorti`). Ниже приведен пример использования функции `asort`.

```
$ gawk 'BEGIN{
> var["a"] = 1
```

```

> var["g"] = 2
> var["m"] = 3
> var["u"] = 4
> asort(var, test)
> for (i in test)
>   print "Index:",i," - value:",test[i]
> }'
Index: 4 - value: 4
Index: 1 - value: 1
Index: 2 - value: 2
Index: 3 - value: 3
$

```

Новый массив, `test`, содержит вновь отсортированные элементы данных первоначального массива, но значения индекса теперь преобразованы в числовые значения, указывающие надлежащий порядок сортировки.

Функция `split` предоставляет великолепный способ преобразования полей данных в элементы массива для дальнейшей обработки:

```

$ gawk 'BEGIN{ FS="," }{
> split($0, var)
> print var[1], var[5]
> }' data1
data11 data15
data21 data25
data31 data35
$

```

В новом массиве в качестве индексов используются последовательно возрастающие числа, начиная со значения индекса 1, содержащего первое поле данных.

Функции работы со временем

Язык программирования `gawk` содержит несколько функций, с помощью которых можно проводить различные манипулирования со значениями времени, как показано в табл. 21.6.

Таблица 21.6. Функции работы со временем языка `gawk`

Функция	Описание
<code>mktime (datespec)</code>	Преобразует дату, заданную в формате <code>YYYY MM DD HH MM SS [DST]</code> , в значение отметки времени
<code>strftime (format [, timestamp])</code>	Форматирует либо текущее время из отметки времени дня, либо время из представленной отметки времени <code>timestamp</code> для получения значений дня и времени, для чего используется формат <code>format</code> функции командного интерпретатора <code>date()</code>
<code>systemtime()</code>	Возвращает отметку времени для текущего времени суток

Функции времени часто используются при работе с файлами журнала, содержащими даты, которые применяются при сравнениях. Сравнение значений дат существенно упрощается после преобразования текстовых представлений дат в значения времени с начала эпохи (в количество секунд, истекшее после полуночи от 1 января 1970 года).

Ниже приведен пример использования функции времени в программе `gawk`.

```
$ gawk 'BEGIN{
> date = systime()
> day = strftime("%A, %B %d, %Y", date)
> print day
> }'
Friday, December 28, 2010
$
```

В этом примере применяется функция `systime` для получения текущей отметки времени от начала эпохи из системы, а затем вызывается функция `strftime` для преобразования этой отметки времени в удобный для чтения формат с учетом символов управления форматом даты для команды командного интерпретатора `date`.

Определяемые пользователем функции

При работе с программой `gawk` можно не ограничиваться лишь применением встроенных функций, предусмотренных в языке `gawk`. Пользователь может создавать собственные функции для применения в своих программах `gawk`. В настоящем разделе показано, как определять и использовать собственные функции в программах `gawk`.

Определение функции

Для определения функции необходимо использовать ключевое слово `function`:

```
function name([variables])
{
    statements
}
```

Имя функции должно однозначно определять назначение создаваемой функции. В функцию можно передать одну или несколько переменных *variables* из вызывающей программы `gawk`:

```
function printthird()
{
    print $3
}
```

Эта функция выводит третье поле данных в записи.

Для возврата значения из функции может также использоваться инструкция `return`:

```
return value
```

В этой инструкции в качестве значения *value* можно задать переменную или выражение, вычисление которого приводит к получению значения:

```
function myrand(limit)
{
    return int(limit * rand())
}
```

Значение, возвращенное из функции, можно присвоить переменной в программе `gawk`:

```
x = myrand(100)
```

Переменная будет содержать значение, возвращенное из функции.

Использование собственных функций

Определение функции должно быть приведено отдельно, перед всеми разделами программы (включая раздел BEGIN). На первых порах такая организация программы может показаться непривычной, но это позволяет отделить код функции от остальной части программы gawk:

```
$ gawk '  
> function myprint()  
> {  
>     printf "%-16s - %s\n", $1, $4  
> }  
> BEGIN{FS="\n"; RS=""}  
> {  
>     myprint()  
> }' data2  
Riley Mullen      - (312)555-1234  
Frank Williams    - (317)555-9876  
Haley Snell       - (313)555-4938  
$
```

В данном примере приведено определение функции `myprint()`, которая форматирует первое и четвертое поля данных в записи, подготавливая их к печати. Затем эта функция используется в программе gawk для отображения данных из файла данных.

После определения функции ее можно вызывать в коде раздела программы столько раз, сколько потребуется. Это позволяет сэкономить много усилий при разработке кода сложных алгоритмов.

Создание библиотеки функций

Очевидно, что необходимость снова и снова определять собственные функции gawk каждый раз, когда они потребуются, вряд ли можно назвать оправданной. Тем не менее в языке gawk предусмотрен способ объединения функций в единый файл библиотеки, который можно использовать во всех своих проектах программирования на языке gawk.

Прежде всего необходимо создать файл, который содержит все применяемые функции gawk:

```
$ cat funclib  
function myprint()  
{  
    printf "%-16s - %s\n", $1, $4  
}  
function myrand(limit)  
{  
    return int(limit * rand())  
}  
function printthird()  
{  
    print $3  
}  
$
```

Файл `funclib` содержит три определения функций. Чтобы воспользоваться этими определениями, необходимо задать параметр командной строки `-f`. К сожалению, не предусмотрена

возможность одновременно задавать параметр командной строки `-f` и вводить встроенный сценарий `gawk`, но можно использовать несколько параметров `-f` в одной и той же командной строке.

Таким образом, чтобы воспользоваться библиотекой, достаточно создать файл, содержащий программу `gawk`, и указать в командной строке и файл библиотеки, и файл программы:

```
$ cat script4
BEGIN{ FS="\n"; RS="" }
{
    myprint()
}
$ gawk -f funclib -f script4 data2
Riley Mullen      - (312)555-1234
Frank Williams    - (317)555-9876
Haley Snell       - (313)555-4938
$
```

Итак, после определения библиотеки достаточно лишь дополнительно указывать файл `funclib` в командной строке `gawk` в каждом случае, когда потребуется использовать одну из функций, определенных в библиотеке.

Резюме

В настоящей главе описаны более сложные средства языка программирования `gawk`. Ни в одном языке программирования нельзя обойтись без использования переменных, и `gawk` не является исключением. Язык программирования `gawk` включает некоторые встроенные переменные, которые можно использовать для ссылки на конкретные значения полей данных и получения информации о количестве полей данных и записей, обработанных в файле данных. Предусмотрена также возможность создавать собственные переменные для использования в своих сценариях.

Кроме того, в языке программирования `gawk` предусмотрены многие стандартные структурированные команды, на наличие которых можно рассчитывать при работе с любым языком программирования. На этом языке можно легко создавать весьма развитые программы с использованием логических конструкций `if-then`, а также циклов `while`, `for` и `do-while`. Каждая из этих команд позволяет изменить поток управления в сценарии на основе программы `gawk` для организации циклов по значениям полей данных и создания подробных отчетов по данным.

Команда `printf` — это великолепный инструмент, без которого невозможно обойтись во время настройки вывода отчета. Эта команда позволяет точно задавать формат отображения данных, выводимых из сценария на основе программы `gawk`. Пользователь может легко создавать отформатированные отчеты, размещая элементы данных по точно выверенным позициям.

Наконец, в данной главе рассматривались многочисленные встроенные функции, имеющиеся в языке программирования `gawk`, и показано, как создавать собственные функции. В языке `gawk` предусмотрены многие полезные функции, предназначенные для выполнения математических вычислений, таких как стандартные функции извлечения квадратных корней и вычисления логарифмов, а также тригонометрические функции. Имеется также несколько строковых функций, в том числе таких, которые позволяют легко извлекать подстроки из более крупных строк.

При разработке программы `gawk` можно не ограничиваться лишь встроенными функциями. Работая над приложением, в котором используется много специализированных алгоритмов, можно создать собственные функции для реализации этих алгоритмов, а затем использовать их в своем коде. Можно также задать файл библиотеки, содержащий все функции, используемые в программах `gawk`, что позволяет дополнительно экономить время и усилия во всей своей работе по программированию.

В следующей главе тематика немного изменится. В этой главе будут рассматриваться некоторые другие варианты среды командного интерпретатора, с которыми приходится сталкиваться, занимаясь разработкой сценариев для системы Linux. Безусловно, командный интерпретатор `bash` является наиболее широко применяемым в Linux, но он — не единственный. Разработчику не помешает немного узнать о некоторых других имеющихся командных интерпретаторах, а также о том, чем они отличаются от командного интерпретатора `bash`.

ГЛАВА

22

В этой главе...

Общие сведения о командном интерпретаторе `dash`

Средства командного интерпретатора `dash`

Сценарная поддержка в командном интерпретаторе `dash`

Командный интерпретатор `zsh`

Компоненты командного интерпретатора `zsh`

Сценарная поддержка с помощью командного интерпретатора `zsh`

Резюме

Работа с другими командными интерпретаторами

Безусловно, командный интерпретатор `bash` представляет собой наиболее широко используемый командный интерпретатор в дистрибутивах Linux, но он — не единственный. В предыдущих главах был описан стандартный командный интерпретатор `bash` системы Linux и показаны его возможности, а теперь настало время для изучения некоторых других командных интерпретаторов, применяемых в мире Linux. В настоящей главе описаны еще два командных интерпретатора, с которыми можно столкнуться, работая с различными дистрибутивами Linux, и показано, чем они отличаются от командного интерпретатора `bash`.

Общие сведения о командном интерпретаторе `dash`

Командный интерпретатор `dash` дистрибутива Debian имеет интересное прошлое. Он непосредственно происходит от командного интерпретатора `ash`, который является упрощенной копией оригинального командного интерпретатора Bourne, применяющегося в системах Unix (см. главу 1). В свое время Кеннет Алмквист (Kenneth Almquist) создал небольшую версию командного интерпретатора Bourne для систем Unix и назвал ее командным интерпретатором Almquist; в дальнейшем этот командный интерпретатор стал именоваться `ash`. Исходная версия командного

интерпретатора `ash` была весьма небольшой и быстродействующей, но в ней отсутствовали многие дополнительные возможности, такие как средства редактирования командной строки или ведения хронологии команд, поэтому при использовании этой версии в качестве интерактивного командного интерпретатора возникали сложности.

Командный интерпретатор `ash` был включен в состав операционной системы NetBSD Unix и до сих пор используется в ней в качестве командного интерпретатора по умолчанию. Разработчики NetBSD приспособили для своих целей командный интерпретатор `ash` путем добавления нескольких новых средств, поэтому он стал в большей степени напоминать по своим возможностям командный интерпретатор Bourne. Эти новые средства включают функции редактирования командной строки с использованием команд редакторов `emacs` и `vi`, а также команду `history`, позволяющую вести хронологию ранее выполненных команд. Аналогичная версия командного интерпретатора `ash` используется также в операционной системе FreeBSD в качестве командного интерпретатора, применяемого по умолчанию для входа в систему.

Для дистрибутива Debian системы Linux создана собственная версия командного интерпретатора `ash` (называемая Debian `ash`, или сокращенно *dash*) для включения ее в соответствующую версию Linux. Командный интерпретатор `dash` по большей части дублирует возможности командного интерпретатора `ash` для версии NetBSD, но предоставляет дополнительные возможности редактирования командной строки.

Очевидно, что во всех этих версиях командных интерпретаторов можно легко запутаться, но положение еще больше усложняется тем, что командный интерпретатор `dash` фактически не является заданным по умолчанию командным интерпретатором во многих дистрибутивах Linux на основе Debian. Учитывая то, что командный интерпретатор `bash` получил в различных версиях Linux очень широкое распространение, в большинстве дистрибутивов Linux на основе Debian в качестве командного интерпретатора, обычно применяемого для входа в систему, используется командный интерпретатор `bash`, а командный интерпретатор `dash` служит лишь в качестве командного интерпретатора для быстрого запуска сценариев инсталляции, применяемых для установки файлов дистрибутива.

Исключением из этого правила является широко распространенный дистрибутив Ubuntu. Описанная ситуация часто становится причиной нарушений в работе программистов, применяющих сценарии для командного интерпретатора, и вызывает множество проблем при выполнении сценариев командного интерпретатора в среде Linux. В дистрибутиве Ubuntu Linux командный интерпретатор `bash` используется как заданный по умолчанию интерактивный командный интерпретатор, а командный интерпретатор `dash` — как заданный по умолчанию командный интерпретатор `/bin/sh`. Такая, с позволения сказать, “особенность” вносит серьезную путаницу в работу программистов, которые применяют сценарии командного интерпретатора.

Как было показано в главе 10, каждый сценарий командного интерпретатора должен начинаться со строки, которая объявляет командный интерпретатор, используемый для сценария. В приведенных выше в данной книге сценариях командного интерпретатора `bash` применялось следующее объявление:

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

Эта строка служит для командного интерпретатора указанием, что нужно использовать программу командного интерпретатора, находящуюся в пути `/bin/bash`, для выполнения сценария. В мире Unix заданным по умолчанию командным интерпретатором всегда был `/bin/sh`. Многие программисты, применяющие сценарии командного интерпретатора, которые привыкли работать в среде Unix, просто копируют это объявление в свои сценарии командного интерпретатора для Linux:

```
#!/bin/sh
```

В большинстве дистрибутивов Linux файл `/bin/sh` представляет собой символическую ссылку (см. главу 3) на программу командного интерпретатора `/bin/bash`. Благодаря этому упрощается перенос сценариев командного интерпретатора, разработанных для командного интерпретатора Bourne системы Unix, в среду Linux, поскольку в сценарии не приходится вносить никаких изменений.

К сожалению, в дистрибутиве Ubuntu Linux `/bin/sh` представляет собой ссылку на программу командного интерпретатора `/bin/dash`. Но командный интерпретатор `dash` поддерживает лишь подмножество команд, доступных в исходном командном интерпретаторе Bourne, поэтому может случиться (и часто случается), что некоторые сценарии командного интерпретатора не функционируют должным образом.

В следующем разделе приведены основные сведения о командном интерпретаторе `dash` и показано, чем он отличается от командного интерпретатора `bash`. Об этом особенно важно знать, если возникает необходимость в написании сценариев командного интерпретатора `bash`, которые, возможно, будут выполняться в среде Ubuntu.

Средства командного интерпретатора `dash`

Несмотря на то что оба эти командные интерпретаторы, `bash` и `dash`, разработаны на основе командного интерпретатора Bourne, между ними имеются некоторые различия. В настоящем разделе описаны средства, предусмотренные в командном интерпретаторе `dash` системы Debian, что позволяет ознакомиться с тем, как работает командный интерпретатор `dash`, прежде чем перейти к изучению средств сценарной поддержки этого командного интерпретатора.

Параметры командной строки командного интерпретатора `dash`

Для управления функционированием командного интерпретатора `dash` используются параметры командной строки. Эти параметры командной строки кратко описаны в табл. 22.1.

Таблица 22.1. Параметры командной строки командного интерпретатора `dash`

Параметр	Описание
<code>-a</code>	Экспортировать все переменные, значения которым присвоены в командном интерпретаторе
<code>-c</code>	Прочитать команды из указанной строки с командами
<code>-e</code>	Если не применяется интерактивный режим, немедленно завершать работу при неудачном окончании выполнения любой непроверенной команды
<code>-f</code>	Отображать подстановочные знаки в имени пути
<code>-n</code>	Если не применяется интерактивный режим, прочесть команды, но не выполнять их
<code>-u</code>	Вывести в поток <code>STDERR</code> сообщение об ошибке при обнаружении попытки раскрыть переменную, которая не была задана
<code>-v</code>	Выводить входные данные в поток <code>STDERR</code> по мере их чтения
<code>-x</code>	Выводить каждую команду в поток <code>STDERR</code> по мере выполнения
<code>-I</code>	Если применяется интерактивный режим, игнорировать символы признака конца файла EOF во входных данных

Параметр	Описание
-i	Принудительно перевести командный интерпретатор в интерактивный режим
-m	Включить управление заданиями (включается по умолчанию в интерактивном режиме)
-s	Считывать команды из потока STDIN (применяется по умолчанию, если не заданы параметры с указанием файлов)
-E	Разрешить использование редактора командной строки emacs
-V	Разрешить использование редактора командной строки vi

В дистрибутиве Debian к исходному списку параметров командной строки командного интерпретатора ash добавлено лишь несколько дополнительных параметров командной строки. Параметры командной строки `-E` и `-V` обеспечивают использование специальных средств редактирования командной строки командного интерпретатора dash.

Параметр командной строки `-E` разрешает использовать команды редактора emacs для редактирования текста в командной строке (см. главу 9). Предусмотрена возможность использовать все команды emacs, предназначенные для манипулирования текстом в одной строке, с помощью комбинаций клавиш на основе клавиш `<Ctrl>` и `<Meta>`.

Параметр командной строки `-V` дает возможность использовать для редактирования текста в командной строке команды редактора vi (см. также главу 9). Это средство позволяет переключаться в командной строке между обычным режимом и режимом редактора vi с помощью клавиши `<Esc>`. При работе в режиме редактора vi можно использовать все обычные команды редактора vi (такие как `x`, для удаления символа, и `i`, для вставки текста). После завершения всех операций по редактированию командной строки необходимо снова нажать клавишу `<Esc>` для выхода из режима редактора vi.

Переменные среды командного интерпретатора dash

По умолчанию командный интерпретатор dash использует весьма значительное количество переменных среды для отслеживания значений различных параметров; кроме того, пользователь может создавать собственные переменные среды. В настоящем разделе описаны переменные среды и показано, как они обрабатываются в командном интерпретаторе dash.

Переменные среды, применяемые по умолчанию

Переменные среды, применяемые по умолчанию в командном интерпретаторе dash, перечислены в табл. 22.2 и показано, для чего они предназначены.

Таблица 22.2. Переменные среды командного интерпретатора dash

Переменная	Описание
CDPATH	Путь поиска файлов, используемый для команды <code>cd</code>
HISTSIZE	Количество строк, хранимых в файле хронологии команд
HOME	Заданный по умолчанию начальный каталог пользователя
IFS	Символы разделителей полей ввода. По умолчанию применяются значения <code>space</code> (пробел), <code>tab</code> (знак табуляции) и <code>newline</code> (символ обозначения конца строки)

Переменная	Описание
MAIL	Имя файла почтового ящика пользователя
MAILCHECK	Частота, с которой должна проводиться проверка файла почтового ящика на наличие новой почты
MAILPATH	Разделенный двоеточиями список из нескольких имен файлов почтовых ящиков. Если этот параметр задан, его значение перекрывает значение переменной среды MAIL
OLDPWD	Значение предыдущего рабочего каталога
PATH	Заданный по умолчанию путь поиска для исполняемых файлов
PPID	Идентификатор процесса родительского объекта текущего командного интерпретатора
PS1	Строка приглашения первичного интерфейса командной строки командного интерпретатора
PS2	Строка приглашения вторичного интерфейса командной строки командного интерпретатора
PS4	Символ, выводимый в начале каждой строки, если применяется трассировка выполнения
PWD	Значение текущего рабочего каталога
TERM	Терминал, заданный по умолчанию для командного интерпретатора

Заслуживает внимания то, что переменные среды командного интерпретатора `dash` весьма напоминают переменные среды, используемые в командном интерпретаторе `bash` (см. главу 5). Это не случайно. Напомним, что оба эти командные интерпретаторы, `dash` и `bash`, представляют собой расширения командного интерпретатора `Bourne`, поэтому и тот и другой включает многие из его средств. Но основным требованием при разработке командного интерпретатора `dash` явилось достижение максимального упрощения, поэтому он поддерживает гораздо меньшее количество переменных среды по сравнению с командным интерпретатором `bash`. Это необходимо учитывать при создании сценариев командного интерпретатора в среде `dash`.

В командном интерпретаторе `dash` для отображения переменных среды используется команда `set`:

```
$set
COLORTERM=''
DESKTOP_SESSION='default'
DISPLAY=':0.0'
DM_CONTROL='/var/run/xdmctl'
GS_LIB='/home/atest/.fonts'
HOME='/home/atest'
IFS='
'
KDEROOTHOME='/root/.kde'
KDE_FULL_SESSION='true'
KDE_MULTIHEAD='false'
KONSOLE_DCOP='DCOPRef(konsole-5293, konsole) '
KONSOLE_DCOP_SESSION='DCOPRef(konsole-5293, session-1) '
LANG='en_US'
LANGUAGE='en'
LC_ALL='en_US'
LOGNAME='atest'
OPTIND='1'
PATH='/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin'
PPID='5293'
```

```
PS1='$ '
PS2='> '
PS4='+ '
PWD='/home/atest'
SESSION_MANAGER='local/testbox:/tmp/.ICE-unix/5051'
SHELL='/bin/dash'
SHLVL='1'
TERM='xterm'
USER='atest'
XCURSOR_THEME='default'
_='ash'
$
```

Читатель должен помнить, что применяемая им по умолчанию среда командного интерпретатора `dash`, скорее всего, будет отличаться по значениям своих параметров, поскольку в различных дистрибутивах Linux переменным среды при входе в систему присваиваются разные значения по умолчанию.

Позиционные параметры

Кроме заданных по умолчанию переменных среды, командный интерпретатор `dash` предусматривает также присваивание значений специальным переменным с учетом того, какие параметры определены в командной строке. Ниже перечислены позиционные переменные параметров, которые могут использоваться в командном интерпретаторе `dash`.

- `$0`. Имя командного интерпретатора.
- `$n`. Позиционный параметр в позиции *n*.
- `$*`. Отдельное значение с содержимым всех параметров, разделенных первым символом из переменной среды с определением разделителя внутренних полей или пробелом, если разделитель внутренних полей не определен.
- `$@`. Разворачивается в несколько параметров, в состав которых входят все параметры командной строки.
- `$#`. Количество позиционных параметров.
- `$?`. Код завершения самой последней команды.
- `$-`. Текущие флаги опций.
- `$$`. Идентификатор процесса (Process ID — PID) текущего командного интерпретатора.
- `$!`. Идентификатор процесса самой последней команды, выполняемой в фоновом режиме.

Все позиционные параметры командного интерпретатора `dash` представляют собой аналоги таких же позиционных параметров, которые предусмотрены в командном интерпретаторе `bash`. Каждый из этих позиционных параметров можно использовать в сценариях командного интерпретатора `dash` по такому же принципу, как в командном интерпретаторе `bash`.

Переменные среды, определяемые пользователем

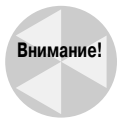
Командный интерпретатор `dash` также позволяет задавать собственные переменные среды. Как и при использовании `bash`, можно определять новые переменные среды в командной строке с помощью инструкции `assign`:

```
$ testing=10
$ echo $testing
10
$
```

По умолчанию переменные среды являются видимыми только в том сеансе командного интерпретатора, в котором они определены. Чтобы обеспечить использование переменных в дочернем командном интерпретаторе или процессе, необходимо задать команду `export`:

```
$ testing=10 ; export testing
$ dash
$ echo $testing
10
$
```

Если команда `export` не применяется, то определяемые пользователем переменные среды остаются видимыми только в текущем командном интерпретаторе или процессе.



Между переменными `dash` и `bash` есть одно огромное различие. Командный интерпретатор `dash` не поддерживает массивы переменных. Из-за этой с виду незначительной особенности перед программистами, разрабатывающими достаточно сложные сценарии командного интерпретатора, возникают определенные проблемы.

Встроенные команды `dash`

Командный интерпретатор `dash`, как и командный интерпретатор `bash`, распознает целый ряд встроенных команд. Эти команды можно вызывать на выполнение непосредственно в интерфейсе командной строки или включать их в сценарии командного интерпретатора. Встроенные команды командного интерпретатора `dash` перечислены в табл. 22.3.

Таблица 22.3. Встроенные команды командного интерпретатора `dash`

Команда	Описание
<code>alias</code>	Создать строку-псевдоним для представления текстовой строки
<code>bg</code>	Продолжить выполнение указанного задания в фоновом режиме
<code>cd</code>	Перейти в указанный каталог
<code>echo</code>	Вывести на внешнее устройство текстовую строку и переменные среды
<code>eval</code>	Соединить все параметры через пробелы
<code>exec</code>	Заменить процесс командного интерпретатора указанной командой
<code>exit</code>	Завершить процесс командного интерпретатора
<code>export</code>	Экспортировать указанную переменную среды для использования во всех дочерних командных интерпретаторах
<code>fc</code>	Вывести в виде списка, отредактировать или снова вызвать на выполнение команды, ранее введенные в командной строке
<code>fg</code>	Продолжить указанное задание в режиме переднего плана
<code>getopts</code>	Получить опции и параметры из списка параметров
<code>hash</code>	Сопровождать хеш-таблицу последних по времени команд и определять их местоположение
<code>pwd</code>	Вывести на внешнее устройство значение текущего рабочего каталога
<code>read</code>	Прочитать строку из потока <code>STDIN</code> и присвоить ее значение переменной

Команда	Описание
<code>readonly</code>	Прочитать строку из потока <code>STDIN</code> в переменную, значение которой не может быть изменено
<code>printf</code>	Вывести на внешнее устройство текст и переменные с использованием строки формата
<code>set</code>	Вывести в виде списка или задать флаги опций и переменные среды
<code>shift</code>	Сдвинуть позиционные параметры на указанное число позиций
<code>test</code>	Вычислить выражение и вернуть 0, если полученный результат является истинным, или 1, если он является ложным
<code>times</code>	Вывести на внешнее устройство накопленные значения затрат времени пользовательскими и системными процессами для командного интерпретатора и всех процессов командного интерпретатора
<code>trap</code>	Интерпретировать и выполнить действие при получении командным интерпретатором указанного сигнала
<code>type</code>	Интерпретировать заданное имя и отобразить результат его разрешения (псевдоним, встроенная команда, обычная команда, ключевое слово)
<code>ulimit</code>	Запросить значения лимитов или задать значения лимитов для процессов
<code>umask</code>	Задать значения применяемых по умолчанию прав доступа к файлам и каталогам
<code>unalias</code>	Удалить указанный псевдоним
<code>unset</code>	Удалить указанную переменную или флаг опции из состава экспортируемых переменных
<code>wait</code>	Ожидать завершения указанного задания и вернуть статус выхода

Читатель мог заметить, что все эти встроенные команды предусмотрены и для командного интерпретатора `bash`. Очевидно, что командный интерпретатор `dash` поддерживает во многом такие же встроенные команды, что и командный интерпретатор `bash`. Однако следует также отметить, что в нем не предусмотрены какие-либо команды для работы с файлом хронологии команд или со стеком каталогов. Командный интерпретатор `dash` не поддерживает эти средства.

Сценарная поддержка в командном интерпретаторе `dash`

К сожалению, командный интерпретатор `dash` распознает не все средства сценарной поддержки командного интерпретатора `bash`. Сценарии командного интерпретатора, написанные для среды `bash`, часто оканчиваются неудачей при запуске в командном интерпретаторе `dash`, что приводит к многочисленным нарушениям в работе программистов, использующих сценарии командного интерпретатора. В настоящем разделе описаны различия, о которых следует знать, чтобы обеспечить надлежащее выполнение сценариев командного интерпретатора в среде `dash`.

Создание сценариев командного интерпретатора `dash`

Читатель уже мог заметить, что процесс создания сценариев для командного интерпретатора `dash` практически не отличается от тех работ, которые осуществляются при разработке сценариев для командного интерпретатора `bash`. В сценариях следует всегда указывать, какой

командный интерпретатор должен использоваться, для обеспечения того, чтобы вызов сценария на выполнение происходил под управлением надлежащего командного интерпретатора.

Для такого указания используется первая строка сценария командного интерпретатора:

```
#!/bin/dash
```

В этой строке можно также задавать параметры командной строки командного интерпретатора, как было описано выше, в разделе “Параметры командной строки dash”.

Возможные причины возникновения нарушений в работе

К сожалению, командный интерпретатор dash поддерживает лишь подмножество средств командного интерпретатора Bourne, поэтому не все возможности сценариев командного интерпретатора bash реализуемы в командном интерпретаторе dash. Особенности, характерные только для bash, часто называют *средствами bash* (bashism). В настоящем разделе приведен краткий перечень средств командного интерпретатора bash, применимых в сценариях командного интерпретатора bash, которые не будут действовать после переноса сценария в среду командного интерпретатора dash.

Использование арифметических вычислений

В главе 10 были показаны три перечисленных ниже способа представления математических выражений в сценарии командного интерпретатора bash.

- Использование команды `expr`.
- `expr operation`
- Задание квадратных скобок.
- `$[operation]`
- Использование двойных круглых скобок.
- `$((operation))`

Командный интерпретатор dash поддерживает команду `expr` и метод с двойными круглыми скобками, но не поддерживает метод с квадратными скобками. Это может стать причиной проблем, если в сценарии приходится применять большое количество математических выражений, в которых используются квадратные скобки:

```
$ cat test5
#!/bin/dash
# проверка математических операций

value1=10
value2=15

value3=$[ $value1 * $value2 ]
echo "The answer is $value3"
$ ./test5
./test5: 7: value1: not found
The answer is
$
```


Надлежащим форматом для представления математических выражений в сценариях командного интерпретатора `dash` является формат, основанный на методе с двойными круглыми скобками:

```
$ cat test5b
#!/bin/dash
# проверка математических операций

value1=10
value2=15

value3=$(( $value1 * $value2 ))
echo "The answer is $value3"
$ ./test5b
The answer is 150
$
```

Теперь командный интерпретатор может выполнить вычисления должным образом.

Команда `test`

Кроме того, хотя в командном интерпретаторе `dash` поддерживается команда `test`, необходимо соблюдать осторожность при ее использовании. Версия команды `test` для командного интерпретатора `bash` немного отличается от версии для командного интерпретатора `dash`.

Команда `test` для командного интерпретатора `bash` позволяет использовать двойной знак равенства (`==`) для проверки того, равны ли две строки. Эта операция проверки введена дополнительно в интересах программистов, которые привыкли ее использовать в других языках программирования:

```
$ cat test6
#!/bin/bash
# проверка оператора сравнения ==

test1=abcdef
test2=abcdef

if [ $test1 == $test2 ]
then
    echo "They're the same!"
else
    echo "They're different"
fi
$ ./test6
They're the same!
$
```

Все достаточно просто. Но при попытке выполнить этот сценарий в среде командного интерпретатора `dash` будет получен отрицательный результат:

```
$ ./test6
[: ==: unexpected operator
They're different
$
```

В команде `test`, предусмотренной в командном интерпретаторе `dash`, не распознается символ `==` как знак операции сравнения текстовых строк. Вместо этого в этой команде распозна-

ется только символ =. Тем не менее после замены знака операции сравнения текстовых строк в виде двойного знака равенства одинарным знаком равенства сценарий становится вполне приемлемым и для командного интерпретатора dash, и для командного интерпретатора bash:

```
$ cat test7
#!/bin/dash
# проверка оператора сравнения =

test1=abcdef
test2=abcdef

if [ $test1 = $test2 ]
then
    echo "They're the same!"
else
    echo "They're different"
fi
$ ./test7
They're the same!
$
```

Из-за этого с виду незначительного средства bash были бесплодно потеряны в поиске ошибок многие рабочие часы программистов, разрабатывающих сценарии для командного интерпретатора!

Опции инструкции echo

Еще одним источником неприятностей для программистов, работающих с командным интерпретатором dash, является простая инструкция echo. Эта инструкция действует по-разному в командных интерпретаторах dash и bash.

Если в командном интерпретаторе bash необходимо представить в выводе специальный символ, то следует задать параметр командной строки -e:

```
echo -e "This line contains\t a special character"
```

В приведенной выше инструкции echo задан специальный символ \t для представления знака табуляции. Если параметр командной строки -e не задан, то версия инструкции echo для командного интерпретатора bash пропускает специальный символ. Ниже приведены результаты проверки, которые позволяют убедиться в этом на примере сценария для командного интерпретатора bash.

```
$ cat test8
#!/bin/bash
# проверка команды echo

echo "This is a normal test"
echo "This test uses a\tspecial character"
echo -e "This test uses a\tspecial character"
echo -n "Does this work: "
read test
echo "This is the end of the test"
$ ./test8
This is a normal test
This test uses a\tspecial character
This test uses a special character
```

```
Does this work: N
This is the end of the test
$
```

Выполнение инструкции `echo`, в которой не был задан параметр командной строки `-e`, привело к отображению символа `\t` в виде обычного текста, поэтому для получения знака табуляции необходимо задать параметр командной строки `-e`.

При работе с командным интерпретатором `dash` дело обстоит немного иначе. Инструкция `echo` в командном интерпретаторе `dash` автоматически распознает и отображает специальные символы. В связи с этим для данного командного интерпретатора не предусмотрен параметр командной строки `-e`. При попытке выполнить этот сценарий в среде командного интерпретатора `dash` формируется следующий вывод:

```
$ ./test8
This is a simple test
This line uses a      special character
-e This line uses a   special character
Does this work: N
This is the end of the test
$
```

Как показывает этот вывод, в версии команды `echo` для командного интерпретатора `dash` вполне успешно распознается специальный символ в строке без параметра командной строки `-e`, а если в командной строке содержится параметр `-e`, то возникает путаница и `-e` отображается как обычный текст.

К сожалению, простого решения этой проблемы не найдено. Если приходится писать сценарии, предназначенные для работы и в среде командного интерпретатора `bash`, и в среде командного интерпретатора `dash`, то наилучшее решение состоит в использовании для отображения текста команды `printf`. Эта команда работает одинаково в обоих вариантах среды командного интерпретатора и обеспечивает успешное отображение специальных символов.

Команда `function`

В главе 16 было показано, как определить собственные функции в сценариях командного интерпретатора. В командном интерпретаторе `bash` поддерживаются два метода определения функций. Первый метод предусматривает использование инструкции `function`:

```
function name {
    commands
}
```

Атрибут `name` определяет уникальное имя, присвоенное функции. Каждая функция, определяемая в сценарии, должна получить уникальное имя.

В этом определении `commands` — это одна или несколько команд командного интерпретатора `bash`, из которых состоит данная функция. После вызова функции командный интерпретатор `bash` выполняет каждую из этих команд в том порядке, в котором они присутствуют в определении функции, точно так же, как и при выполнении обычного кода сценария.

Второй формат определения функций в сценарии командного интерпретатора `bash` в большей степени напоминает формат, который применяется для определения функций в других языках программирования:

```
name() {
    commands
}
```

Пустые круглые скобки после имени функции указывают на то, что далее следует определение функции. На этот формат распространяются такие же правила именования функций, как и на исходный формат функций сценариев командного интерпретатора.

Командный интерпретатор `dash` не поддерживает первый метод определения функций (так как в нем не поддерживается инструкция `function`). Вместо этого в командном интерпретаторе `dash` для определения функции необходимо использовать формат, предусматривающий задание имени функции с круглыми скобками. При попытке вызвать на выполнение функцию, разработанную для командного интерпретатора `bash`, в командном интерпретаторе `dash` будет получено следующее сообщение об ошибке:

```
$ cat test9
#!/bin/dash
# проверка функций

function func1() {
    echo "This is an example of a function"
}
count=1
while [ $count -le 5 ]
do
    func1
    count=$(( $count + 1 ))
done
echo "This is the end of the loop"
func1
echo "This is the end of the script"
$ ./test9
./test9: 4: Syntax error: "(" unexpected
$
```

Вместо включения кода функции в определение функции командный интерпретатор `dash` просто выполняет код, заданный в определении функции, а затем указывает на ошибку в формате сценария командного интерпретатора.

Поэтому при написании сценариев командного интерпретатора, которые могут быть вызваны на выполнение в среде `dash`, следует всегда использовать второй метод определения функций:

```
$ cat test10
#!/bin/dash
# проверка функций

func1() {
    echo "This is an example of a function"
}

count=1
while [ $count -le 5 ]
do
    func1
    count=$(( $count + 1 ))
done
echo "This is the end of the loop"
func1
echo "This is the end of the script"
```

```
$ ./test10
This is an example of a function
This is an example of a function
This is an example of a function
This is an example of a function
This is an example of a function
This is the end of the loop
This is an example of a function
This is the end of the script
$
```

Теперь командный интерпретатор `dash` успешно распознал функцию, определенную в сценарии, и вызвал ее на выполнение в том месте сценария, где это потребовалось.

Командный интерпретатор `zsh`

Еще одним широко применяемым командным интерпретатором, с которым можно столкнуться в своей работе, является командный интерпретатор `Z` (сокращенно называемый `zsh`). Командный интерпретатор `zsh` — это командный интерпретатор Unix с открытым исходным кодом, разработанный Полом Фолстадом (Paul Falstad). При его разработке были изучены возможности всех существующих командных интерпретаторов и добавлены многие уникальные средства в целях создания полномасштабного развитого командного интерпретатора, предназначенного для использования квалифицированными программистами.

Ниже перечислены определенные особенности командного интерпретатора `zsh`, благодаря которым он приобретает уникальные возможности.

- Улучшенная обработка опций командного интерпретатора.
- Режимы совместимости командного интерпретатора.
- Загружаемые модули.

Среди всех этих средств наиболее перспективной особенностью проекта командного интерпретатора `zsh` являются загружаемые модули. Как было показано при описании командных интерпретаторов `bash` и `dash`, каждый командный интерпретатор поддерживает ряд встроенных команд, позволяющих выполнять многие функции без привлечения внешних служебных программ. Одним из преимуществ встроенных команд является высокая скорость выполнения. В командном интерпретаторе не приходится загружать служебную программу в память для последующего ее выполнения; код встроенных команд уже находится в памяти командного интерпретатора, готовый для выполнения.

Командный интерпретатор `zsh` предоставляет для использования основной набор встроенных команд, а также позволяет присоединять дополнительные *командные модули*. Каждый командный модуль обеспечивает возможность использовать ряд дополнительных встроенных команд определенного назначения, таких как команды поддержки сети и развитые математические функции. При этом можно добавлять только те модули, которые, по всей видимости, могут потребоваться в определенной ситуации.

Такая особенность предоставляет великолепный способ, с одной стороны, ограничения объема памяти, занимаемого командным интерпретатором `zsh` в тех ситуациях, в которых требуется командный интерпретатор с небольшими размерами и ограниченным набором команд, а с другой стороны, расширения количества доступных встроенных команд в тех ситуациях, когда требуется обеспечить более высокую скорость выполнения.

Компоненты командного интерпретатора zsh

В настоящем разделе приведены основные сведения о командном интерпретаторе zsh и описаны встроенные команды, которые непосредственно доступны (или могут быть добавлены путем установки модулей), а также параметры командной строки и переменные среды, используемые командным интерпретатором zsh.

Опции командного интерпретатора

В большинстве командных интерпретаторов используются параметры командной строки для определения поведения командного интерпретатора. В командном интерпретаторе zsh применяется значительное количество параметров командной строки для определения того, как должен действовать командный интерпретатор при выполнении тех или иных операций, но для настройки поведения этого командного интерпретатора главным образом используются *опции*. Опции этого командного интерпретатора можно задавать либо в командной строке, либо в самом командном интерпретаторе с помощью команды `set`.

Параметры командной строки, предусмотренные для командного интерпретатора zsh, приведены в табл. 22.4.

Таблица 22.4. Параметры командной строки командного интерпретатора zsh

Параметр	Описание
-c	Выполнить только указанную команду и выйти
-i	Запустить в качестве интерактивного командного интерпретатора, предоставляя приглашение к вводу информации интерфейса командной строки
-s	Вынудить командный интерпретатор считывать команды из устройства <code>STDIN</code>
-o	Задать опции командной строки

Безусловно, может показаться, что этот набор параметров командной строки весьма невелик, но в действительности параметр `-o` позволяет сделать очень многое. С его помощью можно задавать опции командного интерпретатора, которые определяют различные средства, применимые в самом командном интерпретаторе. В целом можно отметить, что командный интерпретатор zsh относится к числу командных интерпретаторов, предоставляющих наиболее широкие возможности настройки. В среде этого командного интерпретатора можно вносить изменения в работу многих его средств. Всевозможные применимые опции подразделяются на несколько общих категорий, описанных ниже.

- **Смена каталогов.** Опции, управляющие тем, как осуществляется смена каталогов с помощью команд `cd` и `dirs`.
- **Завершение команд.** Опции, которые управляют средствами завершения команд.
- **Развертывание и подстановка имен.** Опции, управляющие тем, как происходит раскрытие имен файлов в командах.
- **Журнал.** Опции, с помощью которых осуществляется управление вызовом команд из журнала выполненных команд.

- **Инициализация.** Опции, которые управляют обработкой переменных и файлов запуска командным интерпретатором в начале его работы.
- **Ввод-вывод.** Опции, управляющие обработкой команд.
- **Управление заданиями.** Такие опции, с помощью которых можно определить, как ведется обработка и запуск заданий командным интерпретатором.
- **Отображение приглашений.** Опции, определяющие то, как командный интерпретатор должен работать с приглашениями командной строки.
- **Сценарии и функции.** Опции, от которых зависит обработка командным интерпретатором сценариев командного интерпретатора и определение функций командного интерпретатора.
- **Эмуляция командного интерпретатора.** Опции, которые позволяют задавать поведение командного интерпретатора zsh как имитацию поведения командных интерпретаторов других типов.
- **Состояние командного интерпретатора.** Опции, определяющие тип командного интерпретатора, который должен быть запущен.
- **Редактор zle.** Опции, управляющие средствами построчного редактора zsh (zsh line editor — zle).
- **Опции для определения псевдонимов.** Специальные опции, которые могут использоваться в качестве псевдонимов для других имен опций.

Очевидно, что количество различных категорий опций командного интерпретатора весьма велико. Остается только представить себе, сколько всего опций фактически поддерживает командный интерпретатор zsh. В следующих разделах приведена подборка различных опций командного интерпретатора zsh, которые можно использовать при настройке среды командного интерпретатора zsh.

Состояние командного интерпретатора

Как описано ниже, предусмотрено шесть опций командного интерпретатора zsh, которые определяют тип командного интерпретатора, подлежащего запуску.

- **interactive (-i).** Предоставляет приглашение интерфейса командной строки для ввода встроенных команд и имен программ.
- **login (-l).** Применяемый по умолчанию тип запуска командного интерпретатора zsh, который обеспечивает обработку файлов запуска командного интерпретатора zsh и предоставляет приглашение интерфейса командной строки.
- **privileged (-p).** Применяется по умолчанию, если заданный идентификатор пользователя (User ID — UID) не совпадает с действительным идентификатором пользователя (в связи с переходом пользователя к использованию учетной записи root). Эта опция не предусматривает применение пользовательских файлов запуска.
- **restricted (-r).** Не позволяет пользователю выходить за пределы указанной структуры каталогов в командном интерпретаторе.
- **shin_stdin (-s).** Предусматривает считывание команд из устройства STDIN.
- **single_command (-t).** Обеспечивает выполнение одной команды, считанной из устройства STDIN, и выход.

Состояния командного интерпретатора определяют, должен ли он запускаться с приглашением интерфейса командной строки, а также то, к каким объектам пользователь имеет доступ в ходе работы в командном интерпретаторе.

Эмуляция командного интерпретатора

Опции эмуляции командного интерпретатора позволяют настраивать командный интерпретатор `zsh` таким образом, чтобы он действовал аналогично командному интерпретатору `C` (`C shell` — `csh`) или `Korn` (`Korn shell` — `ksh`). Это — еще два командных интерпретатора, широко применяемых программистами. Указанные опции перечислены ниже.

- **`bsd_echo`**. Обеспечить совместимость инструкции `echo` с командой `echo` командного интерпретатора `C`.
- **`csh_junkie_history`**. Использовать команду `history` без спецификатора, чтобы она всегда ссылалась на предыдущую команду.
- **`csh_junkie_loops`**. Предусмотреть возможность использовать в циклах `while` и `for` команды `end`, как в командном интерпретаторе `C`, вместо `do` и `done`.
- **`csh_junkie_quotes`**. Изменить правила использования одинарных и двойных кавычек так, чтобы они соответствовали правилам командного интерпретатора `C`.
- **`csh_nullcmd`**. Не использовать значения переменных `NULLCMD` и `READNULLCMD` во время выполнения перенаправлений при отсутствии команд.
- **`ksh_arrays`**. Использовать массивы в стиле командного интерпретатора `Korn`, в которых числовые индексы массивов начинаются с 0 и требуется задание фигурных скобок для ссылки на элементы массива.
- **`ksh_autoload`**. Эмулировать возможности функции автозагрузки командного интерпретатора `Korn`.
- **`ksh_option_print`**. Эмулировать применяемый в командном интерпретаторе `Korn` способ задания параметров печати.
- **`ksh_typeset`**. Изменить способ обработки параметров команды `typeset`.
- **`posix_builtins`**. Использовать команду `builtin` для выполнения встроенных команд.
- **`sh_file_expansion`**. Выполнять развертывание имен файлов перед развертыванием любых других параметров.
- **`sh_nullcmd`**. Не использовать переменные `NULLCMD` и `READNULLCMD` при выполнении перенаправления.
- **`sh_option_letters`**. Интерпретировать однобуквенные опции командной строки командного интерпретатора по аналогии с командным интерпретатором `Korn`.
- **`sh_word_split`**. Производить разбиение поля после развертывания параметров, не заключенных в кавычки.
- **`traps_async`**. Во время ожидания выхода из программы немедленно обрабатывать сигналы и реагировать на прерывания.

Очевидно, что количество опций настройки весьма велико, а это позволяет выборочно задавать подлежащие эмуляции средства командного интерпретатора `csh` или `ksh` в командном интерпретаторе `zsh`, не эмулируя полностью весь командный интерпретатор `csh` или `ksh`.

Инициализация

Предусмотрен целый ряд опций, предназначенных для работы со средствами запуска командного интерпретатора, которые перечислены ниже.

- **all_export.** Все параметры и переменные экспортируются в дочерние процессы командного интерпретатора автоматически.
- **global_export.** Параметры, экспортируемые в среду, не определяются как локальные для функции.
- **global_rcs.** Если эта опция не задана, командный интерпретатор `zsh` не выполняет глобальные файлы запуска, но выполняет локальные файлы запуска.
- **rcs.** Если эта опция не задана, командный интерпретатор `zsh` выполняет файл запуска `/etc/zshenv`, но какие-либо другие файлы запуска не выполняет.

Опции инициализации позволяют указать, какие файлы запуска командного интерпретатора `zsh` (если таковые имеются) должны быть выполнены в среде командного интерпретатора. Можно также задать эти значения в самих файлах запуска для определения ограничений на то, какие именно файлы должен выполнять командный интерпретатор.

Сценарии и функции

Опции работы со сценариями и функциями позволяют настраивать среду сценарной поддержки в командном интерпретаторе `zsh`. Это — удобный способ определения того, как должны выполняться функции в командном интерпретаторе.

- **c_bases.** Определяет отображение шестнадцатеричных чисел в формате языка C (`0xddd`) вместо формата командного интерпретатора (`16#ddd`).
- **err_exit.** Если команда завершается с ненулевым статусом выхода, выполнить команду с помощью прерывания `ZERR` и выйти.
- **err_return.** Если команда имеет ненулевой статус выхода, немедленно произвести возврат из включающей ее функции.
- **eval_lineno.** Если эта опция задана, номера строк выражений, вычисленные с использованием встроенной команды `eval`, отслеживаются отдельно от остальной части среды командного интерпретатора.
- **exec.** Определяет выполнение команд. Если эта опция не задана, команды считываются и выводятся сообщения об ошибках, но команды не выполняются.
- **function_argzero.** Задаёт в качестве параметра `$0` имя функции или сценария.
- **local_options.** Если эта опция задана, то после возврата из функции командного интерпретатора восстанавливаются все опции, заданные перед ее вызовом.
- **local_traps.** Если эта опция задана, то в случае задания в функции прерывания от сигнала восстанавливается предыдущее состояние прерывания при выходе из функции.
- **multios.** Предусматривает выполнение неявно заданных команд `tee` или `cat` при попытке осуществления нескольких перенаправлений.
- **octal_zeroes.** Обеспечивает интерпретацию любой целочисленной строки, начинающейся с нуля, как восьмеричного числа.
- **typeset_silent.** Если эта опция не задана, то вызов команды `typeset` с именем параметра приводит к отображению текущего значения параметра.

- **verbose.** Предусматривает отображение строк ввода командного интерпретатора в том виде, в каком они были считаны командным интерпретатором.
- **xtrace.** Обеспечивает отображение команд и их параметров по мере выполнения команд командным интерпретатором.

Командный интерпретатор `zsh` позволяет настраивать многие средства, применяемые при выходе из функций, которые определены в командном интерпретаторе.

Встроенные команды

Уникальной особенностью командного интерпретатора `zsh` является то, что он позволяет разворачивать встроенные команды, которые имеются в командном интерпретаторе. Благодаря этому в распоряжении пользователя оказывается широкий набор удобных программ, предназначенных для работы с многими приложениями.

В настоящем разделе описаны основные встроенные команды, а также различные модули, которые были доступны ко времени написания данной книги.

Основные встроенные команды

Ядро командного интерпретатора `zsh` поддерживает такие же основные встроенные команды, с которыми приходится сталкиваться при работе в других командных интерпретаторах. Основные доступные встроенные команды перечислены в табл. 22.5.

Таблица 22.5. Основные встроенные команды `zsh`

Команда	Описание
<code>alias</code>	Определяет альтернативное имя для команды и параметров
<code>autoload</code>	Обеспечивает предварительную загрузку функции командного интерпретатора в память для ускорения доступа
<code>bg</code>	Выполняет задание в фоновом режиме
<code>bindkey</code>	Связывает клавиатурные комбинации с командами
<code>builtin</code>	Выполняет указанную встроенную команду вместо исполняемого файла с тем же именем
<code>bye</code>	Действует аналогично команде <code>exit</code>
<code>cd</code>	Заменяет текущий рабочий каталог указанным каталогом
<code>chdir</code>	Заменяет текущий рабочий каталог указанным каталогом
<code>command</code>	Выполняет указанную команду <code>command</code> в качестве внешнего файла вместо функции или встроенной команды
<code>declare</code>	Задаёт тип данных переменной (аналогично команде <code>typeset</code>)
<code>dirs</code>	Отображает содержимое стека каталогов
<code>disable</code>	Отключает на время указанные элементы хеш-таблицы
<code>disown</code>	Удаляет указанное задание из таблицы заданий
<code>echo</code>	Отображает переменные и текст
<code>emulate</code>	Задаёт <code>zsh</code> как применяемый для эмуляции другого командного интерпретатора, такого как командный интерпретатор Bourne, Korn или C
<code>enable</code>	Включает указанные элементы хеш-таблицы
<code>eval</code>	Выполняет указанную команду с параметрами в текущем процессе командного интерпретатора

Команда	Описание
<code>exec</code>	Выполняет указанную команду с параметрами, замещая текущий процесс командного интерпретатора
<code>exit</code>	Обеспечивает выход из командного интерпретатора с указанным статусом выхода. Если таковой не указан, используется статус выхода последней команды
<code>export</code>	Позволяет использовать указанные имена и значения переменных среды в дочерних процессах командного интерпретатора
<code>false</code>	Возвращает статус выхода 1
<code>fc</code>	Выбирает заданный диапазон команд из списка в журнале команд
<code>fg</code>	Выполняет указанное задание в режиме переднего плана
<code>float</code>	Задаёт указанную переменную для использования в качестве переменной с плавающей точкой
<code>functions</code>	Задаёт указанное имя в качестве функции
<code>getln</code>	Считывает следующее значение из стека буферов и размещает его в указанной переменной
<code>getopts</code>	Обеспечивает выборку значения следующей допустимой опции в параметрах командной строки и размещение его в указанной переменной
<code>hash</code>	Позволяет непосредственно изменить содержимое хеш-таблицы команд
<code>history</code>	Выводит список команд, содержащихся в файле журнала команд
<code>integer</code>	Задаёт указанную переменную для использования в качестве целочисленного значения
<code>jobs</code>	Выводит информацию об указанном задании или обо всех заданиях, для которых назначен данный процесс командного интерпретатора
<code>kill</code>	Отправляет сигнал (по умолчанию — <code>SIGTERM</code>) указанному процессу или заданию
<code>let</code>	Вычисляет математическое выражение и присваивает результат переменной
<code>limit</code>	Задаёт или отображает предельные значения ресурсов
<code>local</code>	Задаёт средства работы с данными для указанной переменной
<code>log</code>	Отображает данные обо всех пользователях, зарегистрированных в настоящее время в системе, на работе которых отражается применение параметра <code>watch</code>
<code>logout</code>	То же, что и <code>exit</code> , но применяется, только если командный интерпретатор работает в режиме <code>login</code>
<code>popd</code>	Удаляет следующую запись из стека каталогов
<code>print</code>	Отображает переменные и текст
<code>printf</code>	Отображает переменные и текст с использованием строк формата в стиле языка C
<code>pushd</code>	Обеспечивает смену текущего рабочего каталога и ввод данных о предыдущем каталоге в стек каталогов
<code>pushln</code>	Размещает указанные параметры в стеке буферов редактирования
<code>pwd</code>	Отображает полное имя пути текущего рабочего каталога
<code>read</code>	Считывает строку и присваивает значения полей данных указанным переменным с использованием символов <code>IFS</code>
<code>readonly</code>	Присваивает переменной значение, которое не может быть изменено
<code>rehash</code>	Перестраивает хеш-таблицы команд
<code>set</code>	Задаёт опции или позиционные параметры для командного интерпретатора
<code>setopt</code>	Задаёт опции для командного интерпретатора

Команда	Описание
shift	Считывает и удаляет первый позиционный параметр, а затем сдвигает оставшиеся параметры вниз на одну позицию
source	Обеспечивает поиск указанного файла и копирует его содержимое в текущее местоположение
suspend	Приостанавливает выполнение командного интерпретатора до получения им сигнала SIGCONT
test	Возвращает статус выхода 0, если указанное условие имеет значение TRUE
times	Отображает совокупные значения времени пользователя и системы для командного интерпретатора и процессов, выполняемых в командном интерпретаторе
trap	Блокирует указанные сигналы, предотвращая их обработку командным интерпретатором, и выполняет указанные команды при получении сигналов
true	Возвращает нулевой статус выхода
ttystctl	Блокирует и разблокирует дисплей
type	Показывает, как заданная команда должна интерпретироваться командным интерпретатором
typeset	Задаёт или отображает атрибуты переменных
ulimit	Задаёт или отображает предельные значения ресурсов командного интерпретатора или процессы, выполняемые в командном интерпретаторе
umask	Задаёт или отображает разрешения, применяемые по умолчанию при создании файлов и каталогов
unalias	Удаляет указанный псевдоним команды
unfunction	Удаляет указанную функцию, определённую ранее
unhash	Удаляет указанную команду из хеш-таблицы
unlimit	Удаляет указанный предел ресурса
unset	Удаляет указанный атрибут переменной
unsetopt	Удаляет указанную опцию командного интерпретатора
wait	Ожидает завершения указанного задания или процесса
whence	Показывает, как заданная команда должна интерпретироваться командным интерпретатором
where	Отображает имя пути указанной команды, если она найдена командным интерпретатором
which	Отображает имя пути указанной команды с использованием вывода в стиле csh
zcompile	Компилирует указанную функцию или сценарий для более быстрой автозагрузки
zmodload	Выполняет операции на загружаемых модулях zsh

Очевидно, что перечень встроенных команд командного интерпретатора zsh весьма значителен! Но большинство этих команд должны быть знакомы читателю по их аналогам в командном интерпретаторе bash. Наиболее важным средством работы со встроенными командами командного интерпретатора zsh являются модули.

Модули надстройки

Список модулей, предоставляющих дополнительные встроенные команды для командного интерпретатора zsh, весьма велик и продолжает расти, поскольку изобретательные программисты не оставляют своих усилий по созданию новых модулей. В табл. 22.6 приведены модули, доступные ко времени написания данной книги.

Модули командного интерпретатора zsh охватывают широкий перечень направлений, начиная от предоставления простых средств редактирования командной строки и заканчивая

сложными функциями для работы в сети. В основе разработки командного интерпретатора `zsh` лежит замысел: создание базовой минимальной среды командного интерпретатора и предоставление возможности добавлять все компоненты, необходимые для выполнения конкретной работы по программированию.

Таблица 22.6. Модули `zsh`

Модуль	Описание
<code>zsh/cap</code>	Команды совместимости с POSIX
<code>zsh/clone</code>	Команды клонирования выполняемого командного интерпретатора на другом терминале
<code>zsh/compctl</code>	Команды управления завершением команд
<code>zsh/complete</code>	Команды, предназначенные для завершения ввода в командной строке
<code>zsh/compllist</code>	Команды, применяемые для дополнения листинга завершения командной строки
<code>zsh/computil</code>	Вспомогательные команды для завершения командной строки
<code>zsh/datetime</code>	Дополнительные команды и переменные для работы со значениями даты и времени
<code>zsh/deltochar</code>	Функции строкового редактора, которые повторяют некоторые возможности редактора <code>emacs</code>
<code>zsh/files</code>	Основные команды для работы с файлами
<code>zsh/mapfile</code>	Команды доступа к внешним файлам с помощью ассоциативных массивов
<code>zsh/mathfunc</code>	Дополнительные функции для научных вычислений
<code>zsh/parameter</code>	Команды доступа к хеш-таблицам команд с помощью ассоциативных массивов
<code>zsh/pcre</code>	Библиотека расширенных регулярных выражений
<code>zsh/sched</code>	Команды планирования для выполнения команд с учетом времени
<code>zsh/net/socket</code>	Команды поддержки сокетов домена Unix
<code>zsh/stat</code>	Команды доступа к системному вызову <code>stat</code> для получения статистических данных системы
<code>zsh/system</code>	Интерфейс к различным средствам системы низкого уровня
<code>zsh/net/tcp</code>	Команды доступа к сокетам TCP
<code>zsh/termcap</code>	Интерфейс к базе данных <code>termcap</code>
<code>zsh/terminfo</code>	Интерфейс к базе данных <code>terminfo</code>
<code>zsh/zftp</code>	Специализированные команды клиента FTP
<code>zsh/zle</code>	Команды строкового редактора <code>zshell</code>
<code>zsh/zleparameter</code>	Команды доступа, позволяющие изменять значения переменных с использованием редактора <code>zle</code>
<code>zsh/zprof</code>	Команды, обеспечивающие профилирование функций командного интерпретатора
<code>zsh/zpty</code>	Команды запуска команд в псевдотерминале
<code>zsh/zselect</code>	Команды блокирования и возврата по достижении готовности дескрипторов файлов
<code>zsh/zutil</code>	Различные программы командного интерпретатора

Просмотр, добавление и удаление модулей

Интерфейсом к модулям `zsh` является команда `zmodload`. Эта команда используется для просмотра, добавления и удаления модулей в сеансе работы с командным интерпретатором `zsh`.

При вызове команды `zmodload` без параметров командной строки отображаются модули, установленные в настоящее время в командном интерпретаторе `zsh`:

```
% zmodload
zsh/zutil
zsh/complete
zsh/main
zsh/terminfo
zsh/zle
zsh/parameter
%
```

В различных реализациях командного интерпретатора `zsh` по умолчанию включены разные модули. Для добавления нового модуля достаточно указать имя модуля в командной строке `zmodload`:

```
% zmodload zsh/zftp
%
```

После этого не отображаются какие-либо указания на то, что модуль загружен. Но можно еще раз вызвать на выполнение команду `zmodload`, и новый модуль должен появиться в списке установленных модулей.

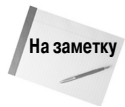
После загрузки модуля связанные с ним команды становятся доступными как встроенные команды:

```
% zftp open myhost.com rich testing1
Welcome to the myhost FTP server.
% zftp cd test
% zftp dir
01-21-11 11:21PM      120823 test1
01-21-11 11:23PM      118432 test2
% zftp get test1 > test1.txt
% zftp close
%
```

Команда `zftp` позволяет полностью провести сеанс работы с FTP-сервером непосредственно из командной строки командного интерпретатора `zsh`! Команды работы с FTP-сервером можно включать в сценарии командного интерпретатора `zsh` для выполнения операций передачи файлов исключительно с помощью сценариев.

Для удаления установленного модуля необходимо задать параметр `-u` наряду с именем модуля:

```
% zmodload -u zsh/zftp
% zftp
zsh: command not found: zftp
%
```



На заметку

Рекомендуется помещать команды `zmodload` в файл запуска `$HOME/.zshrc`, чтобы обеспечить автоматическую загрузку своих предпочтительных функций при запуске командного интерпретатора `zsh`.

Сценарная поддержка с помощью командного интерпретатора zsh

Основное назначение командного интерпретатора zsh состоит в предоставлении развитой среды программирования для программистов, использующих командный интерпретатор. Зная об этом, можно не удивляться тому, как много средств командного интерпретатора zsh предусмотрено для упрощения сценарной поддержки командным интерпретатором.

Математические выражения

Как и следовало ожидать, командный интерпретатор zsh позволяет легко выполнять математические вычисления. В прошлом этот путь был проложен создателями командного интерпретатора Korn, которые обеспечили поддержку математических вычислений, предоставив возможность работать с числами с плавающей точкой. А командный интерпретатор zsh предусматривает полную поддержку чисел с плавающей точкой во всех выполняемых с его помощью математических вычислениях!

Проведение вычислений

Командный интерпретатор zsh поддерживает два метода выполнения математических операций:

- команда `let`;
- двойные круглые скобки.

При использовании команды `let` необходимо заключить математическое выражение в двойные кавычки, чтобы применить в нем пробелы:

```
% let value1=" 4 * 5.1 / 3.2 "  
% echo $value1  
6.3749999999999991  
%
```

Необходимо отметить, что при использовании чисел с плавающей точкой приходится учитывать проблему точности представления результатов. Для решения этой проблемы рекомендуется всегда использовать для вывода команду `printf` и задавать количество знаков после точки в десятичном представлении, которое требуется для правильного отображения ответа:

```
% printf "%6.3f\n" $value1  
6.375  
%
```

Теперь полученный результат намного лучше!

Второй метод состоит в использовании двойных круглых скобок. Этот метод предусматривает применение двух способов определения математического выражения:

```
% value1=$(( 4 * 5.1 ))  
% (( value2 = 4 * 5.1 ))  
% printf "%6.3f\n" $value1 $value2  
20.400  
20.400  
%
```

Следует отметить, что в двойные круглые скобки можно поместить либо одно выражение (которому предшествует знак доллара), либо весь оператор присваивания. При обоих способах записи достигаются одинаковые результаты.

Если для заблаговременного объявления типов данных переменных не используется команда `typeset`, то командный интерпретатор `zsh` предпринимает попытки определить тип данных автоматически. Это может привести к непредсказуемым последствиям, если в математическом выражении должны применяться и целые числа, и числа с плавающей точкой. Рассмотрим следующий пример:

```
% value=10
% value2=$(( $value1 / 3 ))
% echo $value2
3
%
```

При этом может быть получен не такой ответ, который следовало ожидать после выполнения вычисления. Если при определении чисел не задаются дробные десятичные разряды, командный интерпретатор `zsh` рассматривает эти числа как целые и проводит с ними целочисленные вычисления. Для того чтобы результатом было число с плавающей точкой, необходимо задавать числа с дробными десятичными разрядами:

```
% value=10.
% value2=$(( $value1 / 3. ))
% echo $value2
3.3333333333333335
%
```

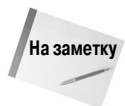
Теперь результат представлен в формате с плавающей точкой.

Математические функции

Работая с командным интерпретатором `zsh`, приходится учитывать, что должны быть загружены все необходимые встроенные математические функции, так как в противном случае не удастся провести задуманные вычисления. По умолчанию командный интерпретатор `zsh` не включает какие-либо специальные математические функции. Но после установки модуля `zsh/mathfunc` количество доступных математических функций увеличивается настолько, что позволяет выполнить практически любые требуемые расчеты:

```
% value1=$(( sqrt(9) ))
zsh: unknown function: sqrt
% zmodload zsh/mathfunc
% value1=$(( sqrt(9) ))
% echo $value1
3.
%
```

Очевидно, что это весьма несложно! После загрузки этого модуля в распоряжении программиста оказывается вся библиотека широко применяемых математических функций.



Командный интерпретатор `zsh` поддерживает большое количество математических функций. Полный перечень всех математических функций, предоставляемых модулем `zsh/mathfunc`, можно найти в справочном руководстве по команде `zshmodules`.

Структурированные команды

Командный интерпретатор `zsh` предоставляет обычный набор структурированных команд для сценариев командного интерпретатора:

- инструкции `if-then-else`;
- циклы `for` (в том числе в стиле `C`);
- циклы `while`;
- циклы `until`;
- инструкции `select`;
- инструкции `case`.

В командном интерпретаторе `zsh` используется такой же синтаксис для каждой из этих структурированных команд, как и в командном интерпретаторе `bash`. Командный интерпретатор `zsh` включает также отличную от обычно применяемых структурированную команду `repeat`. В команде `repeat` используется следующий формат:

```
repeat param
do
    commands
done
```

Параметр `param` должен представлять собой число или математическое выражение, результатом вычисления которого является число. После получения значения параметра `param` команда `repeat` выполняет заданные команды `commands` такое количество раз, которое определяется этим значением:

```
% cat test1
#!/bin/zsh
# использование команды repeat

value1=$(( 10 / 2 ))
repeat $value1
do
    echo "This is a test"
done
$ ./test1
This is a test
This is a test
This is a test
This is a test
This is a test
%
```

Данная команда позволяет повторно выполнять команды в разделах кода столько раз, сколько определяется вычисленным значением.

Функции

Командный интерпретатор `zsh` обеспечивает создание собственных функций с использованием либо команды `function`, либо определения имени функции с круглыми скобками:

```
% function functest1 {
> echo "This is the test1 function"
}
% functest2() {
> echo "This is the test2 function"
}
% functest1
This is the test1 function
% functest2
This is the test2 function
%
```

Как и при работе с функциями командного интерпретатора bash (см. главу 16), можно определять функции в сценарии командного интерпретатора, а затем использовать глобальные переменные или передавать в функции параметры. Ниже приведен пример использования глобальной переменной.

```
% cat test3
#!/bin/zsh
# проверка функций в zsh

dbl() {
    value=$(( $value * 2 ))
    return $value
}

value=10
dbl
echo The answer is $?
% ./test3
The answer is 20
%
```

Функции не обязательно нужно задавать в сценариях командного интерпретатора. Командный интерпретатор zsh позволяет определять функции в отдельных файлах, к которым он может получить доступ, пытаясь разрешить имя функции.

При поиске функций командный интерпретатор zsh использует путь, который определен переменной среды fpath. Предусмотрена возможность сохранять файлы с функциями в любом каталоге, который задан в этом пути. Ниже приведено значение fpath на типичной рабочей станции Linux.

```
% echo $fpath
/usr/local/share/zsh/site-functions
/usr/share/zsh/4.2.5/functions/Completion
/usr/share/zsh/4.2.5/functions/Completion/AIX
/usr/share/zsh/4.2.5/functions/Completion/BSD
/usr/share/zsh/4.2.5/functions/Completion/Base
/usr/share/zsh/4.2.5/functions/Completion/Cygwin
/usr/share/zsh/4.2.5/functions/Completion/Darwin
/usr/share/zsh/4.2.5/functions/Completion/Debian
/usr/share/zsh/4.2.5/functions/Completion/Linux
/usr/share/zsh/4.2.5/functions/Completion/Mandrake
/usr/share/zsh/4.2.5/functions/Completion/Redhat
```

```

/usr/share/zsh/4.2.5/functions/Completion/Unix
/usr/share/zsh/4.2.5/functions/Completion/X
/usr/share/zsh/4.2.5/functions/Completion/Zsh
/usr/share/zsh/4.2.5/functions/MIME
/usr/share/zsh/4.2.5/functions/Misc
/usr/share/zsh/4.2.5/functions/Prompts
/usr/share/zsh/4.2.5/functions/TCP
/usr/share/zsh/4.2.5/functions/Zftp
/usr/share/zsh/4.2.5/functions/Zle
%

```

Вполне очевидно, что командный интерпретатор `zsh` может проводить поиск для разрешения имен функций во многих местах. В этой системе, рассматриваемой в качестве примера, можно разместить собственные функции в каталоге `/usr/local/share/zsh/site-functions`, и командный интерпретатор `zsh` будет иметь возможность разрешать имена этих функций.

Но прежде чем командный интерпретатор `zsh` получит возможность разрешить имя функции, необходимо задать команду `autoload`. Эта команда загружает функцию в память, чтобы командный интерпретатор мог получить к ней доступ.

Ниже приведен пример использования отдельной функции.

```

% cat dbl
#!/bin/zsh
# функция для удваивания значения
dbl() {
    value=$(( $1 * 2 ))
    return $value
}
% cp dbl /usr/local/share/zsh/site-functions
%

```

В зависимости от того, как выполнена настройка системы Linux, может потребоваться перейти к учетной записи пользователя `root` (или применить команду `sudo`), чтобы скопировать файлы в каталоги библиотеки `zsh`. Итак, теперь функция создана в файле и сохранена в одном из каталогов, который задан в переменной `fpath`. Но при попытке использовать ее без загрузки в память будет формироваться сообщение об ошибке:

```

% dbl 5
zsh: command not found: dbl
% autoload dbl
% dbl 5
% echo $?
10
%

```

Это также относится к сценариям командного интерпретатора. Если возникает необходимость воспользоваться какой-либо функцией, следует прежде всего применить команду `autoload` для обеспечения доступа к этой функции:

```

% cat test4
#!/bin/zsh
# проверка внешней функции

autoload dbl

```

```
dbl $1
echo The answer is $?
% ./test4 5
The answer is 10
%
```

Еще одним интересным средством командного интерпретатора `zsh` является команда `zcompile`, которая обрабатывает файл с функциями и создает “откомпилированную” версию для командного интерпретатора. В действительности при этом происходит не такая компиляция, которая обычно применяется в других языках программирования. Речь идет о компиляции, которая приводит к преобразованию функции в двоичный формат, что обеспечивает ускорение загрузки функции командным интерпретатором `zsh`.

После выполнения команды `zcompile` создается версия файла с функциями, имеющая расширение `.zwc`. Команда `autoload` при ее использовании для поиска команды в пути `fpath` обнаруживает версии с расширением `.zwc` и загружает их вместо текстовых файлов с функциями.

Резюме

В настоящей главе описаны два широко применяемых альтернативных командных интерпретатора Linux, с которыми часто приходится сталкиваться в работе. Командный интерпретатор `dash` был разработан в составе дистрибутива `Debian Linux` и применяется главным образом в дистрибутиве `Ubuntu Linux`. Это — сокращенная версия командного интерпретатора `Bourne`, поэтому в ней не поддерживается столь широкое разнообразие средств, как в командном интерпретаторе `bash`. Об этом следует помнить при написании сценариев.

Командный интерпретатор `zsh` часто встречается в различных вариантах среды программирования, поскольку он предоставляет большое количество великолепных средств, востребованных программистами, которые разрабатывают сценарии для командного интерпретатора. В нем используются загружаемые модули для загрузки отдельных библиотек кода, благодаря чему применение самых развитых функций становится столь же простым, как вызов на выполнение обычных команд командной строки! Загружаемые модули предусмотрены для поддержки весьма разнообразных наборов функций, начиная от сложных математических вычислений и заканчивая сетевыми приложениями, такими как `FTP` и `HTTP`.

Следующая часть книги посвящена описанию некоторых конкретных приложений со сценарной поддержкой, с которыми приходится сталкиваться в среде Linux. В следующей главе будет показано, как включить в сценарии командного интерпретатора поддержку двух наиболее широко применяемых в мире Linux пакетов для работы с базами данных, `MySQL` и `PostgreSQL`, для обеспечения обработки данных.

Дальнейшее расширение средств работы со сценариями

ЧАСТЬ

IV

В этой части...

Глава 23

Работа с базами данных

Глава 24

Работа в Интернете

Глава 25

Использование
электронной почты

Глава 26

Написание программ на
основе сценариев

Глава 27

Более сложные
сценарии командного
интерпретатора

ГЛАВА

23

В этой главе...

База данных MySQL

База данных PostgreSQL

Работа с таблицами

Использование базы данных
в сценарии

Резюме

Работа с базами данных

Одна из проблем, возникающих при использовании сценариев командного интерпретатора, состоит в обеспечении работы с постоянно хранимыми данными. Например, в ходе выполнения сценария командного интерпретатора можно хранить всю необходимую информацию в переменных, но после выхода из сценария переменные просто уничтожаются. Но иногда требуется обеспечить сохранение данных в сценарии, чтобы иметь возможность использовать их в дальнейшем. В свое время для сохранения и получения данных в сценарии командного интерпретатора предусматривалась лишь возможность создания файла, чтения данных из файла, интерпретация данных, а затем сохранение данных снова в файле. Для поиска данных в файле необходимо было считать каждую строку в файле для выявления нужных данных. А в наши дни получили весьма широкое распространение базы данных, поэтому стало совсем несложно связать сценарий командного интерпретатора с интерфейсом базы данных с открытым исходным кодом, разработанной на профессиональном уровне. Двумя базами данных с открытым исходным кодом, наиболее широко применяемыми в мире Linux, являются MySQL и PostgreSQL. В настоящей главе показано, как обеспечить функционирование этих баз данных в конкретной системе Linux, а затем несколько слов посвящено описанию работы с базами данных из командной строки. После этого будет описано, как организовать взаимодействие с той и другой базой данных с помощью обычных сценариев командного интерпретатора bash.

База данных MySQL

Безусловно, базой данных, наиболее широко применяемой в среде Linux, является база данных MySQL. Наибольшим стимулом к ее распространению стало создание и развитие серверной среды Linux-Apache-MySQL-PHP (LAMP), которая используется на многих веб-серверах Интернета для размещения сетевых хранилищ данных, блогов и приложений.

В настоящем разделе описано, как установить базу данных MySQL в среде Linux и создать объекты базы данных, необходимые для использования в сценариях командного интерпретатора.

Установка MySQL

В свое время при установке MySQL в системе Linux возникали определенные сложности, но в наши дни в дистрибутивах Linux нередко предусмотрены автоматизированные программы установки программного обеспечения, в том числе MySQL (см. главу 8). Эти программы позволяют не только легко загружать и устанавливать новое программное обеспечение из сетевых репозитариев, но и автоматически проверять наличие обновлений для установленных пакетов программ, а также устанавливать их.

На рис. 23.1 демонстрируется использование средства Add Software (Добавление программного обеспечения) в дистрибутиве Ubuntu Linux.

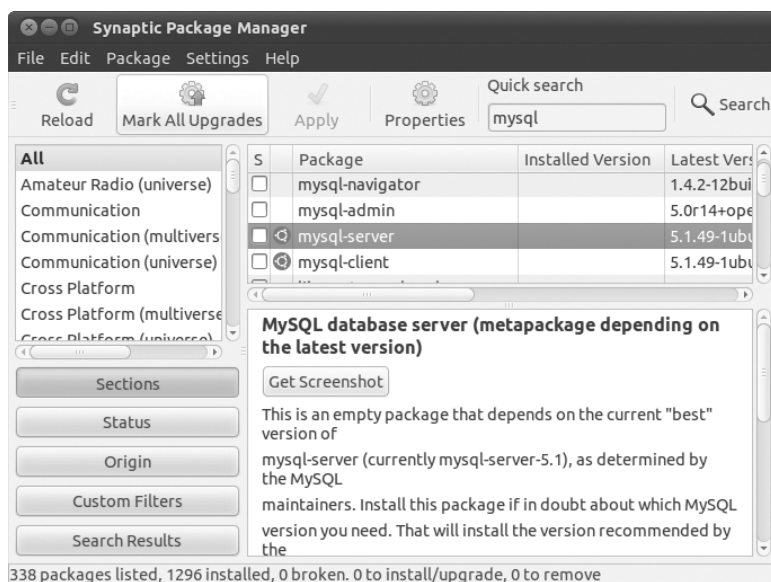


Рис. 23.1. Установка MySQL в системе Ubuntu Linux

Достаточно лишь выбрать опцию для приложения mysql-server, и программа диспетчера пакетов Synaptic Package Manager загрузит и установит полный комплект серверного (и клиентского) программного обеспечения MySQL. Трудно представить себе что-то более простое!

В дистрибутиве openSUSE Linux используется также усовершенствованная система управления программным обеспечением. На рис. 23.2 показано окно Software Management, в котором можно выбирать пакеты, расположенные по категориям программного обеспечения, или искать конкретные пакеты.

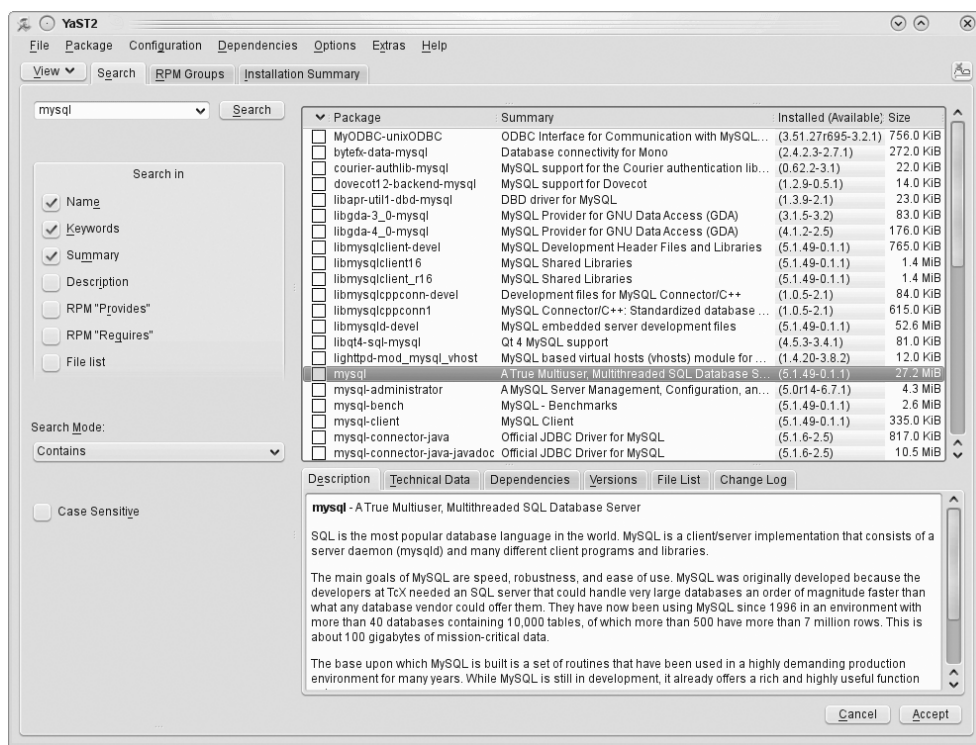
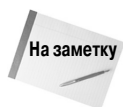


Рис. 23.2. Установка MySQL в системе openSUSE Linux

И в этом случае можно выбрать опцию загрузки программного обеспечения MySQL и установки необходимых пакетов программ.

Если используется дистрибутив Linux, который не поддерживает автоматическую загрузку программного обеспечения, или просто требуется установить последнюю и самую лучшую версию сервера MySQL, то можно загрузить инсталляционный файл непосредственно с веб-сайта MySQL (www.mysql.com) и установить его самостоятельно.

На веб-сайте MySQL имеются и предварительно созданные двоичные пакеты для конкретных дистрибутивов Linux (в которых применяется либо формат пакетов RPM для Red Hat, либо формат пакетов DEB для Debian) и пакеты с исходным кодом. Предварительно созданные двоичные пакеты проще для установки, поэтому, если они предусмотрены для вашей системы Linux, попытайтесь применить их в первую очередь. Возвратитесь к главе 8, если вам потребуется помощь при установке двоичного пакета или пакета с исходным кодом для определенного дистрибутива Linux.



Если вы предпочитаете вручную загрузить и установить MySQL (либо в виде предварительно построенного двоичного пакета, либо в виде загрузки исходного кода), то вам потребуется произвести некоторую дополнительную настройку для подготовки сервера MySQL к работе, тогда как установочный пакет для дистрибутива Linux обеспечивает осуществление установки с предварительной настройкой, не требуя или почти не требуя никаких усилий с вашей стороны!

Клиентский интерфейс MySQL

Порталом, применяемым для работы с базой данных MySQL, служит программа интерфейса командной строки `mysql`. В настоящем разделе описана клиентская программа `mysql` и показано, как ее использовать для взаимодействия с базой данных.

Подключение к серверу

Клиентская программа `mysql` позволяет подключиться к любому серверу базы данных MySQL, находящемуся в любом месте в сети, с использованием любой учетной записи пользователя и пароля. По умолчанию, если вызов программы `mysql` на выполнение произведен в командной строке без указания каких-либо параметров, эта программа предпринимает попытку подключиться к серверу MySQL, эксплуатируемому в той же системе Linux, с использованием того регистрационного имени, с которым пользователь вошел в систему Linux.

Но чаще всего необходимо предусмотреть другие условия подключения к базе данных. Предусмотрено большое количество параметров командной строки, которые позволяют не только управлять тем, к какому серверу MySQL должно быть выполнено подключение, но и указывать, как должен действовать интерфейс `mysql`. В табл. 23.1 приведены параметры командной строки, которые можно использовать с программой `mysql`.

Таблица 23.1. Параметры командной строки `mysql`

Параметр	Описание
-A	Запретить автоматическое повторное хеширование
-b	Запретить выдачу звукового сигнала после ошибки
-B	Не использовать файл журнала команд
-C	Обеспечивать сжатие всей информации, передаваемой между клиентом и сервером
-D	Указать используемую базу данных
-e	Выполнить указанную инструкцию и выйти
-E	Отображать выходные данные запроса по вертикали, по одному полю данных в строке
-f	Продолжать работу при возникновении ошибки SQL
-G	Разрешить использование именованных команд <code>mysql</code>
-h	Указать имя хоста сервера MySQL (по умолчанию применяется имя <code>localhost</code>)
-H	Отображать выходные данные запроса в коде HTML
-i	Пропускать пробелы после имен функций
-N	Не отображать имена столбцов в результатах
-o	Пропускать все инструкции, кроме тех, которые относятся к применяемой по умолчанию базе данных, указанной в командной строке
-P	Выводить приглашение для указания пароля учетной записи пользователя
-p	Задать номер порта TCP, используемого для сетевого подключения
-q	Не кешировать результаты каждого запроса
-r	Отображать значения столбцов без преобразования с применением управляющих кодов
-s	Использовать режим без вывода сообщений
-S	Определить сокет для соединения с хостом <code>localhost</code>
-t	Отображать вывод в форме таблицы

Параметр	Описание
-T	Показывать отладочную информацию, а также статистические данные об использовании памяти процессором при выходе из программы
-u	Задать учетную запись пользователя, в которой должна быть выполнена регистрация
-U	Разрешить использование только таких инструкций UPDATE и DELETE, в которых указаны значения ключей
-v	Использовать режим подробных сообщений
-w	Если соединение не может быть установлено, ожидать в течение определенного времени и повторить попытку
-X	Отображать выходные данные запроса в коде XHTML

Вполне очевидно, что количество параметров командной строки, определяющих то, как должна быть выполнена регистрация на сервере MySQL, весьма велико.

По умолчанию в клиентской программе `mysql` предпринимается попытка зарегистрироваться на сервере MySQL с использованием имени, с которым пользователь вошел в систему Linux. Если это имя не задано в конфигурации MySQL в качестве учетной записи пользователя, то необходимо задавать параметр `-u` для указания имени, с которым должна производиться регистрация на сервере MySQL:

```
$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 39
Server version: 5.1.49-1ubuntu8.1 (Ubuntu)

Copyright (c) 2000, 2010, Oracle and/or its affiliates. All rights
reserved.
This software comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to modify and redistribute it under the GPL v2
license

Type 'help;' or '\h' for help. Type '\c' to clear the current input
statement.

mysql>
```

Параметр `-p` служит для программы `mysql` указанием, что должно быть выведено приглашение к вводу пароля, который должен использоваться с данной учетной записью пользователя для регистрации. После регистрации на сервере можно приступать к вводу команд.

Команды `mysql`

В программе `mysql` используются команды двух типов:

- специальные команды `mysql`;
- стандартные инструкции SQL.

В программе `mysql` предусмотрен собственный набор команд, которые позволяют легко управлять средой и получать информацию о сервере MySQL. Эти команды приведены в табл. 23.2.

Таблица 23.2. Команды mysql

Команда	Сокращенное обозначение	Описание
?	\?	Вывести справку
clear	\c	Очистить командную строку
connect	\r	Подключиться к базе данных и серверу
delimiter	\d	Задать разграничитель инструкций SQL
edit	\e	Отредактировать команду с помощью редактора командной строки
ego	\G	Отправить команду на сервер MySQL и отобразить результаты, расположив их по вертикали
exit	\q	Выйти из программы mysql
go	\g	Отправить команду на сервер MySQL
help	\h	Вывести справку
nopager	\n	Отменить разбивку вывода на страницы и отправить выходные данные на устройство STDOUT
note	\t	Не отправлять вывод в выходной файл
pager	\P	Задать в качестве команды разбивки на страницы указанную программу (по умолчанию используется <code>more</code>)
print	\p	Вывести текущую команду
prompt	\R	Изменить приглашение командной строки mysql
quit	\q	Выйти из программы mysql (то же, что и <code>exit</code>)
rehash	\#	Перестроить хеш-таблицу завершения команд
source	\	Выполнить сценарий SQL в указанном файле
status	\s	Получить информацию о состоянии сервера MySQL
system	\!	Выполнить команду оболочки в системе
tee	\T	Присоединить весь вывод к указанному файлу
use	\u	Использовать другую базу данных
charset	\C	Перейти на другую кодировку
warnings	\W	Выводить предупреждения после каждой инструкции
nowarning	\w	Не показывать предупреждения после каждой инструкции

Предусмотрена возможность задавать команды в полной или сокращенной форме непосредственно в командной строке mysql:

```
mysql> \s
-----
mysql Ver 14.12 Distrib 5.0.45, for redhat-linux-gnu (i386) using
readline 5.0

Connection id:          10
Current database:
Current user:           root@localhost
SSL:                    Not in use
Current pager:          stdout
Using outfile:          ''
```

```

Using delimiter:      ;
Server version:      5.1.49-1ubuntu8.1 (Ubuntu)
Protocol version:    10
Connection:         Localhost via UNIX socket
Server characterset: latin1
Db characterset:    latin1
Client characterset: latin1
Conn. characterset: latin1
UNIX socket:        /var/lib/mysql/mysql.sock
Uptime:             4 hours 15 min 24 sec

Threads: 1 Questions: 53 Slow queries: 0 Opens: 23 Flush tables:
1 Open tables: 17 Queries per second avg: 0.003
-----

```

mysql>

Программа `mysql` реализует все стандартные команды SQL (Structured Query Language), поддерживаемые сервером MySQL. Дополнительные сведения об этом приведены ниже, в разделе “Создание объектов базы данных MySQL”.

Одной из редко применяемых команд SQL, реализованных в программе `mysql`, является команда `SHOW`. Используя ее, можно извлечь информацию о сервере MySQL, в частности, о созданных базах данных и таблицах:

```

mysql> SHOW DATABASES;
+-----+
| Database |
+-----+
| information_schema |
| mysql |
+-----+
2 rows in set (0.04 sec)

mysql> USE mysql;
Database changed
mysql> SHOW TABLES;
+-----+
| Tables_in_mysql |
+-----+
| columns_priv |
| db |
| func |
| help_category |
| help_keyword |
| help_relation |
| help_topic |
| host |
| proc |
| procs_priv |
| tables_priv |
| time_zone |
| time_zone_leap_second |
| time_zone_name |

```

```
| time_zone_transition |
| time_zone_transition_type |
| user |
+-----+
17 rows in set (0.00 sec)
mysql>
```

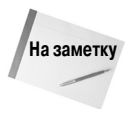
В этом примере команда `SHOW` языка SQL использовалась для отображения баз данных, заданных в настоящее время на сервере MySQL, а затем с помощью команды `USE` языка SQL было выполнено подключение к отдельной базе данных. С помощью сеанса `mysql` можно подключаться одновременно только к одной базе данных.

Заслуживает внимания то, что в приведенном выше примере после каждой команды представлена точка с запятой. Точка с запятой в программе `mysql` указывает конец команды. Если в каком-то месте точка с запятой не введена, то программа `mysql` отображает запрос для продолжения ввода данных:

```
mysql> SHOW
-> DATABASES;
+-----+
| Database |
+-----+
| information_schema |
| mysql |
+-----+
2 rows in set (0.00 sec)

mysql>
```

Такое средство может потребоваться при работе с командами, имеющими большую длину. Это означает, что можно ввести часть команды в одной строке, нажать клавишу <ВВОД>, а затем продолжить набирать команду в следующей строке. Подобный ввод может продолжаться с разбивкой команды на такое количество строк, какое потребуется, после чего нужно будет ввести точку с запятой, чтобы указать конец команды.



Во всей этой главе для представления команд SQL используются прописные буквы. Такой способ написания команд SQL стал общепринятым, но программа `mysql` позволяет задавать команды SQL и в верхнем, и в нижнем регистре.

Создание объектов базы данных MySQL

Прежде чем можно будет приступить к написанию сценариев командного интерпретатора для взаимодействия с базой данных, необходимо создать несколько применяемых для этого объектов базы данных. Необходимо, как минимум, иметь следующие объекты:

- базу данных с уникальным именем для хранения данных прикладной программы;
- уникальную учетную запись пользователя, предназначенную для получения доступа к базе данных из сценариев;
- одну или несколько таблиц данных для хранения данных.

Для создания всех этих объектов используется программа `mysql`, взаимодействующая непосредственно с сервером MySQL, а команды SQL используются для создания и изменения каждого из объектов.

С помощью программы `mysql` можно передавать на сервер MySQL команды SQL любого типа. В настоящем разделе кратко описаны различные инструкции SQL, необходимые для создания основных объектов базы данных, предназначенных для работы с ними в сценариях командного интерпретатора.

Создание базы данных

На сервере MySQL доступ к данным организован с применением *баз данных*. Каждая база данных обычно хранит данные отдельного приложения, что позволяет представлять эти данные независимо от других приложений, которые работают с тем же сервером базы данных. Если для каждого приложения для работы со сценариями командного интерпретатора создается отдельная база данных, то появляется возможность устранить путаницу и неопределенность в отношении того, к чему относятся данные.

Для создания новой базы данных необходимо ввести примерно такую инструкцию SQL:

```
CREATE DATABASE name;
```

Очевидно, что здесь нет ничего сложного. Безусловно, пользователь для создания новых баз данных на сервере MySQL обязан обладать надлежащими правами доступа. Проще всего можно решить задачу создания новой базы данных, зарегистрировавшись с помощью учетной записи пользователя `root`:

```
$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 39
Server version: 5.1.49-lubuntu8.1 (Ubuntu)

Copyright (c) 2000, 2010, Oracle and/or its affiliates. All rights
 reserved.
This software comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to modify and redistribute it under the GPL v2
 license

Type 'help;' or '\h' for help. Type '\c' to clear the current input
 statement.

mysql> CREATE DATABASE test;
Query OK, 1 row affected (0.02 sec)

mysql>
```

Затем, чтобы определить, успешно ли создана новая база данных, можно воспользоваться командой `SHOW`:

```
mysql> SHOW DATABASES;
+-----+
| Database                |
+-----+
| information_schema      |
| mysql                   |
| test                    |
+-----+
3 rows in set (0.01 sec)

mysql>
```

Полученные результаты указывают на то, что база данных была успешно создана. После этого должна появиться возможность подключения к новой базе данных:

```
mysql> USE test;
Database changed;
mysql> SHOW TABLES;
Empty set (0.00 sec)
mysql>
```

Команда `SHOW TABLES` позволяет определить, созданы ли в базе данных какие-либо таблицы. Результат “Empty set” указывает, что пока еще в базе данных нет таблиц, с которыми можно было бы работать. Но прежде чем приступать к созданию таблиц, необходимо решить еще одну задачу.

Создание учетной записи пользователя

Выше в главе было показано, как подключиться к серверу MySQL с использованием учетной записи администратора `root`. Эта учетная запись предоставляет полный контроль над всеми объектами сервера MySQL (во многом аналогично тому, как учетная запись `root` системы Linux позволяет полностью управлять этой системой).

Но использовать учетную запись `root` базы данных MySQL при эксплуатации обычных приложений чрезвычайно опасно. Если существует угроза нарушения безопасности и кто-то выяснит пароль для учетной записи пользователя `root`, то в работе системы могут произойти очень многие нарушения (в том числе в составе данных).

Для предотвращения этого рекомендуется создавать в системе MySQL отдельные учетные записи пользователей, предоставляющие права доступа только к тем базам данных, которые используются в приложениях соответствующих пользователей. Для этого предназначена инструкция `GRANT` языка SQL:

```
mysql> GRANT SELECT,INSERT,DELETE,UPDATE ON test.* TO test IDENTIFIED
by 'test';
Query OK, 0 rows affected (0.35 sec)

mysql>
```

Эта команда имеет действительно широкий набор опций. Ниже последовательно рассматривается каждая опция и показано, для чего она предназначена.

В первом разделе инструкции определяются права доступа, которые имеет учетная запись пользователя применительно к той или иной базе данных. Эта инструкция позволяет с помощью учетной записи пользователя выполнять запросы к данным базы данных (право доступа для выборки), вставлять новые записи данных, удалять существующие записи данных и обновлять существующие записи данных.

Запись `test.*` определяет базу данных и таблицы, к которым относятся эти права доступа. Для указания этих сведений применяется следующий формат:

```
database.table
```

Как показывает данный пример, при указании базы данных *database* и таблиц *table* решается использовать подстановочные символы. В этом случае указано, что перечисленные права доступа применяются ко всем таблицам, содержащимся в базе данных с именем `test`.

Наконец, указаны учетные записи пользователей, к которым относятся права доступа. Удобной особенностью команды `grant` является то, что если учетная запись пользователя не существует, то она создается. В части `identified by` разрешается задавать применяемый по умолчанию пароль для новой учетной записи пользователя.

Проверить результаты создания новой учетной записи пользователя можно непосредственно из программы `mysql`:

```
$ mysql test -u test -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 40
Server version: 5.1.49-lubuntu8.1 (Ubuntu)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>
```

Первый параметр задает применяемую по умолчанию базу данных (в данном случае `test`), и, как уже было сказано, параметр `-u` определяет учетную запись пользователя, с помощью которой проводится регистрация, а параметр `-p` указывает, что должно быть выведено приглашение для задания пароля. После ввода пароля, назначенного учетной записи пользователя `test`, происходит подключение к серверу.

Теперь, после того как в нашем распоряжении появились база данных и учетная запись пользователя, можно приступить к созданию некоторых таблиц для данных. Но вначале рассмотрим еще один сервер базы данных, который можно использовать.

База данных PostgreSQL

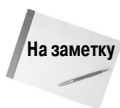
Разработка базы данных PostgreSQL была начата в рамках исследовательского проекта, предназначенного для демонстрации того, как дополнить сервер базы данных с обычными функциями, предусмотрев в нем более усовершенствованные способы работы с базами данных. Со временем эволюция базы данных PostgreSQL привела к созданию одного из наиболее развитых серверов базы данных с открытым исходным кодом в числе тех, что могут применяться в среде Linux.

В настоящем разделе описано, как установить и эксплуатировать сервер базы данных PostgreSQL, а затем показана подготовка к работе учетной записи пользователя и базы данных в целях их применения в сценариях командного интерпретатора.

Установка PostgreSQL

Как и MySQL, пакет сервера базы данных PostgreSQL можно установить либо с использованием автоматизированной системы установки программного обеспечения конкретного дистрибутива, либо вручную путем загрузки этого пакета с веб-сайта PostgreSQL (www.postgresql.org).

Для использования автоматизированной системы установки программного обеспечения в применяемом дистрибутиве Linux выполните такие же действия, которые вкратце были описаны выше, в разделе “Установка MySQL”. По аналогии с MySQL, область загрузки PostgreSQL на веб-сайте этой базы данных содержит и предварительно построенные пакеты для нескольких дистрибутивов Linux, и пакет исходного кода. Снова рекомендуем обратиться к главе 8 для определения того, используется ли в вашем дистрибутиве Linux система управления пакетами, поддерживающая двоичный пакет PostgreSQL. В случае отрицательного ответа потребуется загрузить пакет исходного кода для PostgreSQL и вручную откомпилировать его в конкретной системе Linux. В ходе осуществления такого проекта может потребоваться решить нетривиальные задачи.



Для компиляции пакета исходного кода необходимо иметь пакет разработки программного обеспечения C, загруженный в систему Linux. В наши дни такие действия нередко приходится выполнять при подготовке к работе серверных компьютеров, но если используется дистрибутив Linux для рабочего стола, то пакет исходного кода может оказаться еще не загруженным. Для ознакомления с тем, как должна осуществляться компиляция программного обеспечения и какие пакеты программ необходимы для компиляции проектов с исходным кодом на языке C, обратитесь к документации по применяемому дистрибутиву Linux.

После установки сервера базы данных PostgreSQL можно приступить к регистрации на сервере PostgreSQL, но при этом обнаруживаются некоторые различия по сравнению с сервером MySQL. Напомним, что сервер MySQL поддерживает собственную внутреннюю базу данных пользователей, которым может быть предоставлен доступ к объектам базы данных. Такая же возможность предусмотрена и в сервере PostgreSQL, но в большинстве реализаций PostgreSQL (включая применяемую по умолчанию установку с помощью исходного кода) для проверки пользователей PostgreSQL служат существующие учетные записи пользователей системы Linux.

Безусловно, иногда при этом возникает путаница, но в целом такая система регистрации пользователей предоставляет прозрачный и удобный способ управления учетными записями пользователей в базе данных PostgreSQL. Необходимо лишь проследить за тем, чтобы каждый пользователь PostgreSQL имел действительную учетную запись в системе Linux, и не брать на себя заботу о сопровождении целого отдельного набора учетных записей пользователей.

Еще одним значительным различием, которое обнаруживается при работе с базами данных PostgreSQL, является то, что учетная запись администратора в PostgreSQL носит имя `postgres`, а не `root`. В связи с наличием этой особенности перед установкой базы данных PostgreSQL в системе Linux необходимо заранее определить учетную запись пользователя Linux под именем `postgres`.

В следующем разделе показано, как использовать учетную запись `postgres` для получения доступа к серверу PostgreSQL.

Интерфейс команд PostgreSQL

Клиентская программа PostgreSQL для работы с командной строкой носит имя `psql`. Эта программа предоставляет полный доступ к объектам базы данных, конфигурация которых определена на сервере PostgreSQL. В настоящем разделе описана команда `psql` и показано, как с ее помощью можно взаимодействовать с сервером PostgreSQL.

Подключение к серверу

Клиентская программа `psql` предоставляет интерфейс командной строки для работы с сервером PostgreSQL. Как и следовало ожидать, для вызова этой программы предназначены параметры командной строки, управляющие тем, какие средства разрешены в клиентском интерфейсе. Для каждой опции предусмотрен длинный или короткий формат имени. В табл. 23.3 перечислены имеющиеся параметры командной строки.

Как было указано в предыдущем разделе, учетная запись администратора для PostgreSQL носит имя `postgres`. В связи с тем, что в базе данных PostgreSQL для проверки пользователей применяются учетные записи пользователей Linux, для получения доступа к серверу PostgreSQL в качестве пользователя `postgres` необходимо зарегистрироваться в системе Linux с помощью учетной записи `postgres`.

Таблица 23.3. Параметры командной строки psql

Имя в коротком формате	Имя в длинном формате	Описание
-a	--echo-all	Отобразить в выводе все обработанные строки SQL из файла сценария
-A	--no-align	Задать формат вывода в режиме без выравнивания. Данные не отображаются в виде отформатированной таблицы
-c	--command	Выполнить указанную инструкцию SQL и выйти
-d	--dbname	Указать базу данных, к которой необходимо подключиться
-e	--echo-queries	Производить эхо-повтор всех запросов на экране
-E	--echo-hidden	Производить эхо-повтор скрытых метакоманд psql на экране
-f	--file	Выполнить команды SQL из указанного файла и выйти
-F	--field-separator	Задать символ, используемый для разделения данных столбцов в режиме без выравнивания. Значением по умолчанию является запятая
-h	--host	Указать IP-адрес или имя хоста удаленного сервера PostgreSQL
-l	--list	Вывести список имеющихся баз данных на сервере и выйти
-o	--output	Перенаправить выходные данные запроса в указанный файл
-p	--port	Указать порт TCP сервера PostgreSQL, к которому должно быть выполнено подключение
-P	--pset	Задать указанное значение указанного параметра вывода таблицы
-q	--quiet	Определить режим без вывода сообщений, в котором не отображаются выходные сообщения
-R	--record-separator	Использовать указанный символ в качестве разделителя записей. Значением по умолчанию является символ обозначения конца строки
-s	--single-step	Выводить приглашение, в котором можно указать, продолжать ли работу или отменить задание, после каждого запроса SQL
-S	--single-line	Указать, что конец ввода запроса SQL определяется путем нажатия клавиши <ВВОД>, а не задания точки с запятой
-t	--tuples-only	Запретить отображение заголовков столбцов и нижних колонтитулов в выводе таблицы
-T	--table-attr	Использовать указанный тег таблицы HTML в режиме HTML
-U	--username	Использовать указанное имя пользователя для подключения к серверу PostgreSQL
-v	--variable	Задать указанное значение указанной переменной
-V	--version	Отобразить номер версии psql и выйти
-W	--password	Принудительно обеспечить вывод приглашения для задания пароля
-x	--expanded	Разрешить применение расширенного вывода таблицы для отображения дополнительных сведений, относящихся к записям
-X	--nopsqlrc	Не обрабатывать файл запуска psql
-?	--help	Отобразить справку по параметрам командной строки psql и выйти

Чтобы было проще решить эту проблему, можно использовать команду `sudo` для вызова на выполнение программы командной строки `psql` в качестве пользователя с учетной записью `postgres`.

```
$ sudo -u postgres psql
[sudo password for rich]:
psql (8.4.5)
Type "help" for help.
```

```
postgres=#
```

Применяемое по умолчанию приглашение `psql` указывает, к какой базе данных подключен пользователь. Знак диеза (`#`) в приглашении свидетельствует о том, что регистрация была выполнена в учетной записи пользователя, имеющей права администратора. Теперь мы можем приступить к вводу некоторых команд, предназначенных для взаимодействия с сервером PostgreSQL.

Команды `psql`

Как и программа `mysql`, программа `psql` предусматривает использование команд двух различных типов, описанных ниже.

- Стандартные инструкции SQL.
- Метакоманды PostgreSQL.

Метакоманды PostgreSQL позволяют легко получить информацию о среде базы данных, а также задать свойства сеанса `psql`. Для обозначения метакоманд служит обратная косая черта. В связи с большим количеством различных настроек и средств предусмотрено много различных метакоманд PostgreSQL, но пока мы можем заняться изучением лишь некоторых из них. Обычно применяемые команды перечислены ниже.

- `\l`. Предназначена для получения перечня имеющихся баз данных.
- `\c`. Обеспечивает подключение к базе данных.
- `\dt`. Предназначена для получения перечня таблиц в базе данных.
- `\du`. Предназначена для получения перечня пользователей PostgreSQL.
- `\z`. Служит для вывода перечня прав доступа к таблице.
- `\?`. Предназначена для получения перечня всех имеющихся метакоманд.
- `\h`. Позволяет получить перечень всех имеющихся команд SQL.
- `\q`. Предназначена для выхода из программы работы с базой данных.

Чтобы узнать о том, какие метакоманды являются доступными, достаточно ввести метакоманду `\?`. Появится список всех имеющихся метакоманд, а также их описание.

Для проверки того, как работают метакоманды, воспользуемся метакомандой `\l` для получения перечня имеющихся баз данных:

```
postgres=# \l
List of databases
  Name          | Owner      | Encoding | Collation | Ctype      | Access
-----+-----+-----+-----+-----+-----
 postgres      | postgres  | UTF8     | en_US.utf8 | en_US.utf8 |
 template0     | postgres  | UTF8     | en_US.utf8 | en_US.utf8 | =c/postgres
 template1     | postgres  | UTF8     | en_US.utf8 | en_US.utf8 | =c/postgres
(3 rows)
postgres=#
```

Полученный листинг показывает базы данных, имеющиеся на сервере, наряду с предусмотренными в них средствами (авторы исключили элементы столбца `Access`, чтобы вывод команды можно было удобно разместить на странице книги). В этом списке фигурируют применяемые по умолчанию базы данных, предоставляемые сервером PostgreSQL. База данных `postgres` служит для сопровождения всех системных данных, относящихся к серверу. Базы данных `template0` и `template1` содержат применяемые по умолчанию шаблоны баз данных, которые пользователь может копировать, создавая новую базу данных.

Теперь мы можем приступить к работе с собственными данными в PostgreSQL.

Создание объектов базы данных PostgreSQL

В настоящем разделе рассматривается процесс создания конкретной базы данных и учетной записи пользователя, предназначенной для получения доступа к этой базе данных. В нем будет показано, что определенная часть работы в PostgreSQL осуществляется полностью аналогично работе в базе данных MySQL, но есть и такие действия, которые характерны исключительно для PostgreSQL.

Создание объекта базы данных

Создание базы данных представляет собой одно из тех действий, которое является одинаковым и в PostgreSQL, и в MySQL. Еще раз напомним, что для создания новой базы данных необходимо зарегистрироваться в учетной записи администратора `postgres`:

```
$ sudo -u postgres psql
psql (8.4.5)
Type "help" for help.
```

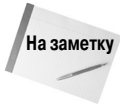
```
postgres=# CREATE DATABASE test;
CREATE DATABASE
postgres=#
```

После создания базы данных можно воспользоваться метакомандой `\l`, чтобы ознакомиться с тем, присутствуют ли сведения об этой базе данных в листинге, а затем ввести метакоманду `\c`, чтобы подключиться к этой базе данных:

```
postgres=# \l
List of databases
  Name      | Owner      | Encoding | Collation | Ctype      | Access
-----+-----+-----+-----+-----+-----
 postgres   | postgres   | UTF8     | en_US.utf8 | en_US.utf8 |
 template0  | postgres   | UTF8     | en_US.utf8 | en_US.utf8 | =c/postgres
 template1  | postgres   | UTF8     | en_US.utf8 | en_US.utf8 | =c/postgres
 test       | postgres   | UTF8     | en_US.utf8 | en_US.utf8 |
(4 rows)

postgres=# \c test
psql (8.4.5)
You are now connected to database "test".
test=#
```

После того как вы подключитесь к базе данных `test`, приглашение программы `psql` изменится, указывая новое имя базы данных. Это — очень удобное напоминание о том, какая среда базы данных используется, благодаря чему при создании объектов можно легко определить, где вы находитесь в системе.



В программе PostgreSQL предусмотрена возможность введения дополнительного уровня управления к базе данных, называемого *схемой*. База данных может содержать несколько схем, а в каждой схеме может содержаться несколько таблиц. Это позволяет подразделять базу данных с учетом конкретных приложений или пользователей.

По умолчанию каждая база данных содержит только одну схему, `public`. Если в намерения пользователя входит применение базы данных только для одного приложения, то он вполне может ограничиться работой лишь со схемой `public`. Но если возникнет необходимость усложнить структуру базы данных, то можно создать новые схемы. В этом примере для работы с таблицами применяется только схема `public`.

Создание учетных записей пользователей

После создания новой базы данных следующим шагом должно стать создание учетной записи пользователя, которая будет получать доступ к ней из сценария командного интерпретатора. Как уже было сказано, учетные записи пользователей в PostgreSQL значительно отличаются от применяемых в базе данных MySQL.

Учетные записи пользователей в PostgreSQL именуются *ролями регистрации*. Сервер PostgreSQL согласовывает роли регистрации с учетными записями пользователей системы Linux. В связи с этим предусмотрены два общепринятых подхода к созданию ролей регистрации, которые предназначены для выполнения сценариев командного интерпретатора, получающих доступ к базе данных PostgreSQL:

- создание специальной учетной записи Linux и соответствующей роли регистрации PostgreSQL, предназначенной для выполнения всех сценариев командного интерпретатора;
- создание учетных записей пользователей PostgreSQL для каждой учетной записи пользователя Linux, которая должна применяться в сценариях командного интерпретатора для получения доступа к базе данных.

Для данного примера выберем второй метод и создадим учетную запись PostgreSQL, согласованную с учетной записью в системе Linux. Это позволяет вызывать на выполнение сценарии командного интерпретатора, которые обеспечивают доступ к базе данных PostgreSQL непосредственно из учетной записи пользователя Linux.

Прежде всего необходимо создать роль регистрации:

```
test=# CREATE ROLE rich login;  
CREATE ROLE  
test=#
```

Очевидно, что это достаточно просто. Если не применяется параметр `login`, то роли не предоставляется разрешение регистрироваться на сервере PostgreSQL, но ей могут быть назначены права доступа. Роль такого типа именуется *ролью группы*. Роли группы великолепно подходят для тех условий, когда работа осуществляется в крупной производственной среде с большим количеством пользователей и таблиц. Вместо того чтобы брать на себя обязанности следить за тем, какие пользователи имеют те или иные права доступа к тем или иным таблицам, достаточно лишь создать роли группы для конкретных типов доступа к таблицам, а затем назначить роли регистрации соответствующим ролям группы.

Но для упрощения работы со сценариями командного интерпретатора вам, скорее всего, не следует заниматься созданием ролей группы, а лишь только назначить права доступа непосредственно ролям регистрации. Именно такой подход применяется в рассматриваемом примере.

Но в базе данных PostgreSQL права доступа трактуются немного иначе по сравнению с MySQL. PostgreSQL не позволяет предоставлять полные права доступа на все объекты базы данных, распространяющиеся вплоть до уровня таблиц. Вместо этого необходимо обеспечивать предоставление прав доступа для каждой отдельной создаваемой таблицы. Безусловно, это требует определенных дополнительных усилий, но способствует введению в действие достаточно строгих политик безопасности. Работу по назначению прав доступа необходимо отложить до тех пор, пока не будет создана таблица. В этом состоит следующий шаг рассматриваемого процесса.

Работа с таблицами

Итак, после запуска сервера MySQL или PostgreSQL, а также создания новой базы данных и учетной записи пользователя для получения к ней доступа мы можем приступить к работе с данными! К счастью, в обеих программах, `mysql` и `psql`, используются стандартные инструкции SQL для создания таблиц данных и управления ими. В настоящем разделе кратко описаны команды SQL, необходимые для создания таблиц, вставки и удаления данных, а также запрос существующих данных в обеих средах.

Создание таблицы

И серверы MySQL, и серверы PostgreSQL относятся к категории серверов *реляционных* баз данных. Особенностью реляционной базы данных является то, что данные организованы с применением *полей данных, строк и таблиц*. Поле данных — это отдельный фрагмент информации, такой как фамилия сотрудника или оклад. Строка — это совокупность взаимосвязанных полей данных, таких как идентификационный номер сотрудника, фамилия, имя, адрес и оклад. Каждая строка представляет собой отдельный набор полей данных.

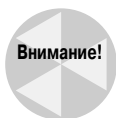
Таблица содержит все строки, в которых хранятся взаимосвязанные данные. Таким образом, может быть предусмотрена, допустим, таблица `Employees`, в которой хранятся все строки, относящиеся к каждому из сотрудников.

Для создания новой таблицы в базе данных необходимо воспользоваться командой `CREATE TABLE` языка SQL:

```
$ mysql test -u root -p
Enter password:
mysql> CREATE TABLE employees (
-> empid int not null,
-> lastname varchar(30),
-> firstname varchar(30),
-> salary float,
-> primary key (empid));
Query OK, 0 rows affected (0.14 sec)

mysql>
```

Прежде всего следует отметить, что для создания новой таблицы нам потребовалось зарегистрироваться на сервере MySQL с помощью учетной записи пользователя `root`, поскольку пользователь `test` не имеет прав доступа для создания новой таблицы. Кроме того, следует указать, что база данных `test` задана в командной строке программы `mysql`. Если бы это не было сделано, то пришлось бы воспользоваться командой `USE` языка SQL для подключения к базе данных `test`.



Крайне важно каждый раз проверять, действительно ли произошло подключение к требуемой базе данных, перед созданием новой таблицы. Кроме того, следует обязательно регистрироваться в базе данных, используя учетную запись администратора (`root` в MySQL и `postgres` в PostgreSQL), чтобы создать таблицу.

При определении каждого поля данных в таблице должен быть указан тип данных для этого поля. Базы данных MySQL и PostgreSQL поддерживают много типов данных. В табл. 23.4 показаны некоторые из наиболее широко применяемых типов данных, которые могут потребоваться в работе каждого программиста.

Таблица 23.4. Типы данных MySQL и PostgreSQL

Тип данных	Описание
char	Строковое значение фиксированной длины
varchar	Строковое значение переменной длины
int	Целочисленное значение
float	Значение с плавающей точкой
Boolean	Логическое значение <code>true</code> или <code>false</code>
Date	Значение даты в формате <code>YYY-MM-DD</code>
Time	Значение времени в формате <code>HH:mm:ss</code>
Timestamp	Значения даты и времени, заданные вместе
Text	Строковое значение большой длины
BLOB	Двоичное значение большой длины, такое как изображение или видеозапись

По отношению к полю данных `empid` задано также *ограничение данных*. Ограничение данных лимитирует то, данные какого типа могут быть введены для создания допустимой строки. Ограничение данных `not null` указывает, что значение `empid` должно быть приведено в каждой строке.

Наконец, ключевое слово `primary key` определяет поле данных, которое однозначно задает каждую отдельную строку. Это означает, что в каждой записи данных в таблице должно присутствовать уникальное значение `empid`.

После создания новой таблицы можно воспользоваться соответствующей командой, чтобы убедиться в том, что операция создания выполнена успешно. В программе `mysql` в этих целях применяется команда `show tables`:

```
mysql> show tables;
+-----+
| Tables_in_test |
+-----+
| employees      |
+-----+
1 row in set (0.00 sec)

mysql>
```

А в программе `psql` таковой является метакоманда `\dt`:

```
test=# \dt
          List of relations
Schema | Name      | Type  | Owner
-----+-----+-----+-----
```

```
public | employees | table | postgres
(1 row)
```

```
test=#
```

Напомним, что в разделе “Создание объектов базы данных PostgreSQL” было сказано, что в базе данных PostgreSQL должны быть назначены права доступа на уровне таблицы. Теперь, после создания таблицы, необходимо предоставить к ней доступ на уровне роли регистрации:

```
$ sudo -u postgres psql
psql (8.4.5)
Type "help" for help.
```

```
postgres=# \c test
psql (8.4.5)
You are now connected to database "test".
test=# GRANT SELECT, INSERT, DELETE, UPDATE ON public.employees
↳ TO rich;
GRANT
test=#
```

Формат задания таблицы должен включить имя схемы, которым по умолчанию является public. Кроме того, следует помнить, что эта команда должна выполняться в роли регистрации postgres, а подключение следует произвести к базе данных test.

После создания таблицы можно приступать к выполнению некоторых операций сохранения данных. О том, как это сделать, речь пойдет в следующем разделе.

Вставка и удаление данных

Не удивительно, что для вставки новых записей данных в таблицу используется команда INSERT языка SQL. В каждой команде INSERT необходимо задать значения полей данных, поскольку, если это не будет сделано, сервер MySQL или PostgreSQL не примет строку для обработки.

Команда INSERT языка SQL имеет следующий формат:

```
INSERT INTO table VALUES (...)
```

Значения *VALUES* должны быть заданы в виде разделенного запятыми списка значений данных, предусмотренных для каждого поля данных:

```
$ mysql test -u test -p
Enter password:
```

```
mysql> INSERT INTO employees VALUES (1, 'Blum', 'Rich', 25000.00);
Query OK, 1 row affected (0.35 sec)
```

В базе данных PostgreSQL эта команда выглядит так:

```
$ psql test
psql (8.4.5)
Type "help" for help.

test=> INSERT INTO employees VALUES (1, 'Blum', 'Rich', 25000.00);
INSERT 0 1
test=>
```


В примере для MySQL используется приглашение командной строки `-u` для регистрации в учетной записи пользователя `test`, которая была создана в MySQL. А в примере для PostgreSQL используется текущая учетная запись пользователя `Linux`, поэтому в этом примере просто ведется работа в заранее созданной учетной записи пользователя `rich`.

Команда `INSERT` вставляет заданные значения данных в поля данных таблицы. При попытке добавления еще одной строки, в которой значение поля данных `empid` дублируется, выработывается сообщение об ошибке:

```
mysql> INSERT INTO employees VALUES (1, 'Blum', 'Barbara', 45000.00);
ERROR 1062 (23000): Duplicate entry '1' for key 1
```

Но, если значение `empid` будет изменено так, что станет уникальным, операция вставки завершится успешно:

```
mysql> INSERT INTO employees VALUES (2, 'Blum', 'Barbara', 45000.00);
Query OK, 1 row affected (0.00 sec)
```

Теперь в таблице должны присутствовать две записи данных.

Для удаления данных из таблицы служит команда `DELETE` языка SQL. Однако при использовании этой команды необходимо соблюдать предельную осторожность.

Команда `DELETE` имеет следующий основной формат:

```
DELETE FROM table;
```

где параметр *table* указывает таблицу, из которой должны быть удалены строки. Тем не менее приведенная выше команда имеет один небольшой недостаток: очевидно, что при ее выполнении были удалены все строки в таблице.

Чтобы указать для удаления лишь единственную строку или группу строк, необходимо изменить конструкцию `WHERE`. Конструкция `WHERE` позволяет создать фильтр, который указывает записи, предназначенные для удаления. Конструкция `WHERE` применяется примерно так:

```
DELETE FROM employees WHERE empid = 2;
```

Тем самым операция удаления ограничивается только теми строками, которые имеют значение `empid`, равное 2. После выполнения этой команды программа `mysql` возвращает сообщение, указывающее, какое количество строк соответствовало фильтру:

```
mysql> DELETE FROM employees WHERE empid = 2;
Query OK, 1 row affected (0.29 sec)
```

Как и следовало ожидать, фильтру соответствовала и была удалена только одна строка.

Выполнение запросов к данным

После ввода всех необходимых данных в базу данных можно приступить к запуску операций получения отчетов для извлечения информации.

Средством выполнения всех без исключения запросов служит команда `SELECT` языка SQL. Она предоставляет весьма широкие возможности, но из-за такой разносторонности при ее использовании возникают сложности.

Инструкция `SELECT` имеет следующий основной формат:

```
SELECT datafields FROM table
```

Параметр *datafields* представляет собой разделенный запятыми список имен полей данных, значения которых должны быть возвращены запросом. Если требуется получить значения всех полей данных, то можно воспользоваться звездочкой в качестве подстановочного символа.

Кроме того, должна быть указана конкретная таблица *table*, в которой необходимо выполнить поиск данных с помощью запроса. Чтобы полученные результаты имели смысл, необходимо обеспечить согласование полей данных запроса с соответствующей таблицей.

По умолчанию команда `SELECT` возвращает данные из всех записей данных в указанной таблице:

```
mysql> SELECT * FROM employees;
+-----+-----+-----+-----+
| empid | lastname | firstname | salary |
+-----+-----+-----+-----+
|      1 | Blum     | Rich      | 25000  |
|      2 | Blum     | Barbara   | 45000  |
|      3 | Blum     | Katie Jane | 34500  |
|      4 | Blum     | Jessica   | 52340  |
+-----+-----+-----+-----+
4 rows in set (0.00 sec)

mysql>
```

Предусмотрена возможность использовать один или несколько модификаторов для определения того, как сервер базы данных должен возвращать данные, затребованные в запросе. Ниже приведен список обычно используемых модификаторов.

- **WHERE.** Отображает подмножество строк, соответствующих конкретному условию.
- **ORDER BY.** Выводит записи в указанном порядке.
- **LIMIT.** Отображает только подмножество строк.

Наиболее широко применяемым модификатором команды `SELECT` является конструкция `WHERE`, которая позволяет задавать условия для выборки данных по условию из результирующего набора. Ниже приведен пример использования конструкции `WHERE`.

```
mysql> SELECT * FROM employees WHERE salary > 40000;
+-----+-----+-----+-----+
| empid | lastname | firstname | salary |
+-----+-----+-----+-----+
|      2 | Blum     | Barbara   | 45000  |
|      4 | Blum     | Jessica   | 52340  |
+-----+-----+-----+-----+
2 rows in set (0.01 sec)

mysql>
```

Итак, теперь в распоряжении читателя имеются все возможности, связанные с внедрением средств доступа к базе данных в сценарии командного интерпретатора! Эти средства позволяют легко реализовать все потребности управления данными, для чего достаточно лишь задать несколько команд SQL, применяя программу `mysql` или `psql`. В следующем разделе описано, как включить эти средства в сценарии командного интерпретатора.

Использование базы данных в сценарии

Итак, после описания ввода в действие работоспособной базы данных настало время снова обратить внимание на использование средств сценарной поддержки. В этом разделе описаны

действия, необходимые для обеспечения работы с базами данных с использованием сценариев командного интерпретатора.

Подключение к базе данных

Очевидно, что для подключения к базе данных в сценарии командного интерпретатора необходимо каким-то образом воспользоваться программой `mysql` или `psql`. Этот процесс — не слишком сложный, однако при его осуществлении необходимо учесть несколько обязательных условий.

Поиск программы

Первое препятствие, которое приходится преодолевать, состоит в том, чтобы выяснить, в каком каталоге системы Linux находится клиентская программа командной строки `mysql` или `psql`. Одна из сложностей при использовании программного обеспечения Linux состоит в том, что в разных дистрибутивах Linux установка пакетов программ приводит к их размещению в различных местоположениях.

К счастью, в системе Linux предусмотрена команда `which`, позволяющая узнать, в каком каталоге командный интерпретатор может найти команду при попытке вызова ее на выполнение из командной строки:

```
$ which mysql
/usr/bin/mysql
$ which psql
/usr/bin/psql
$
```

Чтобы воспользоваться этой информацией, проще всего присвоить полученное значение переменной среды, а затем задать это значение в сценарии командного интерпретатора, когда потребуются сослаться на соответствующую программу:

```
MYSQL='which mysql'
PSQL='which psql'
```

Итак, переменная `$MYSQL` указывает на исполняемый файл программы `mysql`, а переменная `$PSQL` — на исполняемый файл программы `psql`.

Регистрация на сервере

После определения местонахождения клиентской программы можно воспользоваться этой программой в своем сценарии, чтобы получить доступ к серверу базы данных. Что касается сервера PostgreSQL, то должна быть выполнена следующая простая команда:

```
$ cat ptest1
#!/bin/bash
# проверка подключения к серверу PostgreSQL

PSQL='which psql'

$PSQL test
$ ./ptest1
psql (8.4.5)
Type "help" for help.

test=>
```

Сценарий вызывается на выполнение из учетной записи пользователя Linux, а учетная запись PostgreSQL имеет то же имя, поэтому в командной строке `psql` достаточно лишь указать имя базы данных, к которой должно быть выполнено подключение. В сценарии `pctest1` произошло подключение к базе данных `test`, после чего на экране осталось приглашение `psql` для работы с этой базой данных.

Что касается MySQL, то если была создана специальная учетная запись пользователя для применения в сценариях командного интерпретатора, в командной строке `mysql` необходимо указать эту учетную запись:

```
$ cat mtest1
#!/bin/bash
# проверка подключения к серверу MySQL

MYSQL='which mysql'

$MYSQL test -u test -p
$ ./mtest1
Enter password:
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 43
Server version: 5.1.49-lubuntu8.1 (Ubuntu)

Copyright (c) 2000, 2010, Oracle and/or its affiliates. All rights
 reserved.
This software comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to modify and redistribute it under the GPL v2
 license

Type 'help;' or '\h' for help. Type '\c' to clear the current input
 statement.

mysql>
```

Таким образом, мы уже кое-чего добились, но это не совсем подходит для создания не-интерактивных сценариев. Применение параметра командной строки `-p` приводит к тому, что программа `mysql` приостанавливает свою работу и запрашивает пароль. Чтобы решить эту проблему, можно задать пароль непосредственно в командной строке:

```
$MYSQL test -u test -ptest
```

Однако это — неудачное решение. Любой, кто получит доступ к вашему сценарию, сможет определить учетную запись пользователя и пароль для вашей базы данных.

Чтобы найти решение этой проблемы, можно воспользоваться специальным файлом конфигурации, применяемым в программе `mysql`. В программе `mysql` для чтения специальных команд запуска и параметров конфигурации применяется файл `$HOME/.my.cnf`. Одним из таких параметров конфигурации является применяемый по умолчанию пароль для сеансов `mysql`, запущенных в учетной записи пользователя.

Чтобы задать в этом файле пароль, применяемый по умолчанию, достаточно ввести следующую запись:

```
$ cat .my.cnf
[client]
```

```
password = test
$ chmod 400 .my.cnf
$
```

Команда `chmod` используется для задания прав доступа к файлу `.my.cnf` таким образом, чтобы его могли видеть только вы. Полученные результаты можно проверить с помощью командной строки:

```
$ mysql test -u test
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 44
Server version: 5.1.49-lubuntu8.1 (Ubuntu)

Copyright (c) 2000, 2010, Oracle and/or its affiliates. All rights
reserved.
This software comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to modify and redistribute it under the GPL v2
license

Type 'help;' or '\h' for help. Type '\c' to clear the current input
statement.

mysql>
```

Это как раз то, что требуется! После этого необходимость задавать пароль в командной строке в сценариях командного интерпретатора отпадает.

Передача команд на сервер

После установления соединения с сервером необходимо обеспечить передачу команд на сервер для организации взаимодействия с конкретной базой данных. Для этого предусмотрены два метода:

- отправка одной команды и выход;
- отправка нескольких команд.

Чтобы передать на сервер одну команду, необходимо включить ее в состав командной строки `psql` или `mysql`.

Что касается команды `mysql`, то для этого используется параметр `-e`:

```
$ cat mtest2
#!/bin/bash
# передача команды на сервер MySQL

MYSQL='which mysql'

$MYSQL test -u test -e 'select * from employees'
$ ./mtest2
+-----+-----+-----+-----+
| empid | lastname | firstname | salary |
+-----+-----+-----+-----+
|      1 | Blum     | Rich      | 25000  |
```

```

|      2 | Blum      | Barbara   | 45000   |
|      3 | Blum      | Katie Jane | 34500   |
|      4 | Blum      | Jessica   | 52340   |
+-----+-----+-----+-----+
$

```

Для команды `psql` это действие можно осуществить с помощью параметра `-c`:

```

$ cat ptest2
#!/bin/bash
# передача команды на сервер PostgreSQL

PSQL='which psql'

$PSQL test -c 'select * from employees'
$ ./ptest2
  empid | lastname | firstname | salary
+-----+-----+-----+-----+
      1 | Blum     | Rich      | 25000
      2 | Blum     | Barbara   | 45000
      3 | Blum     | Katie Jane | 34500
      4 | Blum     | Jessica   | 52340
(4 rows)
$

```

Сервер базы данных возвращает результаты выполнения команд SQL в сценарий командного интерпретатора, в котором эти результаты отображаются с помощью устройства `STDOUT`.

Если потребуется передать несколько команд SQL, то можно воспользоваться перенаправлением файла (см. главу 14). Чтобы перенаправить строки в сценарий командного интерпретатора, необходимо определить строку с обозначением *конца файла*. Строка конца файла указывает, где начинаются и где оканчиваются перенаправленные данные.

Ниже приведен пример определения строки конца файла, которая позволяет передать конкретные данные.

```

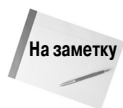
$ cat mtest3
#!/bin/bash
# передача нескольких команд в MySQL

MYSQL='which mysql'
$MYSQL test -u test <<EOF
show tables;
select * from employees where salary > 40000;
EOF
$ ./mtest3
Tables_in_test
employees
  empid  lastname  firstname  salary
-----
      2   Blum    Barbara    45000
      4   Blum    Jessica    52340
$

```

Командный интерпретатор перенаправляет все строки, находящиеся между разграничителями EOF, в команду `mysql`, которая обрабатывает эти строки так, как если бы они были введены непосредственно в приглашении командной строки. Применение этого способа по-

звolyет передавать на сервер MySQL любое необходимое количество команд. Однако следует отметить, что вывод одной команды не отделен от вывода другой. В следующем разделе, “Форматирование данных”, будет показано, как устранить эту проблему.



Следует также отметить, что в программе `mysql` был изменен применяемый по умолчанию стиль вывода после применения указанного способа ввода с помощью перенаправления. Вместо создания рамок с помощью символов ASCII вокруг данных программой `mysql` были возвращены просто бесформатные данные, поскольку в ней было обнаружено перенаправление входных данных. Такой формат вывода является более удобным, если из данных приходится извлекать отдельные элементы данных.

Тот же способ организации работы может применяться и в программе `psql`:

```
$ cat ptest3
#!/bin/bash
# передача нескольких команд в PostgreSQL

PSQL='which psql'

$PSQL test <<EOF
\dt
select * from employees where salary > 40000;
EOF
$ ./ptest3

      List of relations
 Schema |   Name   | Type | Owner
-----+-----+-----+-----
 public | employees | table | postgres
(1 row)

 empid | lastname | firstname | salary
-----+-----+-----+-----
      2 | Blum     | Barbara   | 45000
      4 | Blum     | Jessica   | 52340
(2 rows)
$
```

Программа `psql` отображает вывод каждой команды непосредственно на устройстве `STDOUT` в том порядке, в каком этот вывод был задан.

Разумеется, такой способ вывода не ограничивается лишь получением данных из таблиц. Предусмотрена возможность использовать в сценарии команды SQL любого типа, например, инструкцию `INSERT`:

```
$ cat mtest4
#!/bin/bash
# передача данных в таблицу базы данных MySQL

MYSQL='which mysql'

if [ $# -ne 4 ]
then
    echo "Usage: mtest4 empid lastname firstname salary"
else
    statement="INSERT INTO employees VALUES ($1, '$2', '$3', $4)"

```

```

$MYSQL test -u test << EOF
$statement
EOF
if [ $? -eq 0 ]
then
    echo Data successfully added
else
    echo Problem adding data
fi
fi
$ ./mtest4
Usage: mtest4 empid lastname firstname salary
$ ./mtest4 5 Blum Jasper 100000
Data added successfully
$
$ ./mtest4 5 Blum Jasper 100000
ERROR 1062 (23000) at line 1: Duplicate entry '5' for key 1
Problem adding data
$

```

Этот пример демонстрирует несколько особенностей применения данного способа. Если применяется строка обозначения конца файла, то в строке ввода должна присутствовать лишь одна эта строка в самом начале строки ввода. Если текст EOF будет введен с отступом для согласования с отступом в остальной части команды if-then, то оператор перенаправления перестанет действовать.

Следует также отметить, что в инструкции INSERT заданы одинарные кавычки вокруг текстовых значений, а вся инструкция INSERT заключена в двойные кавычки. Нельзя путать кавычки, используемые для строковых значений, и кавычки, предназначенные для определения текста переменной, в которой задан сам сценарий.

Кроме того, необходимо отметить, что для проверки статуса выхода программы mysql использовалась специальная переменная \$? . Это позволяет определить, была ли команда выполнена успешно.

Способ, предусматривающий просто передачу вывода из команд в устройство STDOUT, не совсем подходит для управления и манипулирования данными. В следующем разделе демонстрируются некоторые приемы, которые можно использовать для того, чтобы в сценариях было проще перехватывать данные, полученные из базы данных.

Форматирование данных

Стандартный вывод команд mysql и psql не совсем подходит для выборки данных. Если требуется действительно осуществить определенные операции с полученными данными, то возникает необходимость прибегнуть к использованию более сложных способов манипулирования данными. В настоящем разделе описаны некоторые приемы, которые можно использовать для более простого извлечения данных из отчетов, сформированных в базе данных.

Присваивание вывода переменной

Первый шаг в осуществлении попытки перехвата данных из базы данных состоит в перенаправлении вывода команды mysql или psql в переменную среды. Это позволяет использовать такую выходную информацию в других командах. Соответствующий пример приведен ниже.


```

$ cat mtest5
#!/bin/bash
# перенаправление вывода SQL в переменную

MYSQL='which mysql'

dbs='$MYSQL test -u test -Bse 'show databases''
for db in $dbs
do
    echo $db
done
$ ./mtest5
information_schema
test
$

```

В рассматриваемом примере применяются два дополнительных параметра в командной строке программы `mysql`. Параметр `-B` служит для программы `mysql` указанием, что работа должна осуществляться в пакетном режиме (batch), и при его использовании в сочетании с параметром `-s` (сокращение от silent) происходит подавление вывода заголовков столбцов и символов форматирования.

В этом примере вывод команды `mysql` перенаправляется в переменную, поэтому появляется возможность осуществить пошаговую обработку отдельных значений из каждой возвращенной строки.

Использование тегов форматирования

В предыдущем примере было показано, что дополнительное введение параметров `-B` и `-s` в командную строку программы `mysql` позволяет подавить вывод информации заголовков столбцов, поэтому происходит получение только данных, без элементов оформления. Предусмотрено также еще несколько других параметров, которые могут использоваться для упрощения работы с базой данных.

В качестве примера можно указать, что в наши дни базы данных широко применяются для формирования содержимого веб-страниц. Опция, позволяющая отображать вывод с использованием формата HTML, предусмотрена в обеих программах, `mysql` и `psql`. В обоих случаях для ввода этой опции в действие применяется параметр командной строки `-H`:

```

$ psql test -H -c 'select * from employees where empid = 1'
<table border="1">
<tr>
  <th align="center">empid</th>
  <th align="center">lastname</th>
  <th align="center">firstname</th>
  <th align="center">salary</th>
</tr>
<tr valign="top">
  <td align="right">1</td>
  <td align="left">Blum</td>
  <td align="left">Rich</td>
  <td align="right">25000</td>
</tr>
</table>

```

```
<p>(1 row)<br />
</p>
$
```

Программа `mysql` поддерживает также еще один широко применяемый формат, основанный на использовании языка XML (Extensible Markup Language). В этом языке, как и в языке HTML, применяются теги, но дополнительно предусмотрена возможность обозначать значения данных идентификаторами.

При работе с программой `mysql` для этого служит параметр командной строки `-X`:

```
$ mysql test -u test -X -e 'select * from employees where empid = 1'
<?xml version="1.0"?>

<resultset statement="select * from employees">
<row>
  <field name="empid">1</field>
  <field name="lastname">Blum</field>
  <field name="firstname">Rich</field>
  <field name="salary">25000</field>
</row>
</resultset>
$
```

Если используется формат XML, то появляется возможность легко идентифицировать не только отдельные строки данных, но и конкретные значения данных в каждой строке.

Резюме

В настоящей главе было показано, в чем может состоять возможность сохранять, изменять и получать данные в сценариях командного интерпретатора с применением баз данных. Непосредственно из сценария командного интерпретатора можно легко получить доступ и к серверу базы данных MySQL, и к серверу базы данных PostgreSQL.

После установки сервера MySQL или PostgreSQL можно воспользоваться клиентской программой для того или иного сервера, чтобы получить доступ к серверу из командной строки или из сценария командного интерпретатора. Клиентская программа `mysql` предоставляет интерфейс командной строки для работы с сервером MySQL. С ее помощью можно передавать на сервер команды SQL, а также собственные команды MySQL из сценария командного интерпретатора, а затем получать результаты.

Клиентская программа `psql` выполняет аналогичные действия для сервера PostgreSQL. Для указанных клиентских программ предусмотрено большое количество параметров командной строки, с помощью которых можно обеспечить форматирование данных строго определенным способом.

Обе клиентские программы позволяют передавать на сервер отдельные команды или использовать перенаправление ввода для отправки пакета команд. Эти программы обычно отправляют выходные данные с сервера на устройство `STDOUT`, но полученный вывод можно перенаправить в переменную и использовать эту информацию в сценарии командного интерпретатора.

В следующей главе рассматривается тематика World Wide Web. Обеспечение взаимодействия сценариев командного интерпретатора с веб-сайтами в Интернете — сложная задача, но после ее успешного решения перед вами раскроется весь мир, где вы сможете получить любые необходимые данные.

Работа в Интернете

Обычно принято считать, что программирование сценариев командного интерпретатора не имеет ничего общего с Интернетом. Мир, в котором работа ведется с помощью командной строки, часто кажется чуждым по отношению к фантастически привлекательному миру графики Интернета. Тем не менее предусмотрено несколько программ, которые могут легко использоваться в сценариях командного интерпретатора для получения доступа к содержимому, представленному в виде данных на сайтах в Интернете, а также на других сетевых устройствах. В настоящей главе рассматриваются три широко применяемых метода, которые позволяют обеспечить взаимодействие сценариев командного интерпретатора с источниками данных, находящимися в сети.

Программа Lynx

Программа Lynx, созданная в 1992 году студентами Канзасского университета в качестве текстового браузера, имеет почти такой же возраст, как и сам Интернет. Программа Lynx является текстовой, поэтому позволяет просматривать веб-сайты непосредственно в терминальном сеансе после замены в ней привлекательного графического оформления веб-страниц текстовыми тегами HTML. Это позволяет заниматься серфингом в Интернете с применением терминала Linux почти любого типа. Типичный пример экрана Lynx показан на рис. 24.1.

В программе Lynx для перехода по веб-страницам используются стандартные клавиши клавиатуры. Ссылки отображаются на веб-странице в виде выделенного текста. С помощью клавиши со стрелкой вправо можно перейти по ссылке на следующую веб-страницу.

ГЛАВА

24

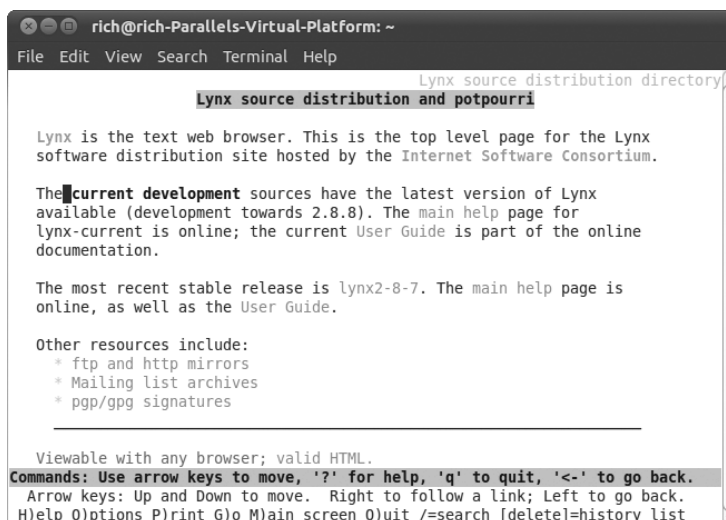
В этой главе...

Программа Lynx

Программа cURL

Работа в сети с помощью
командного интерпретатора
zsh

Резюме



```
rich@rich-Parallels-Virtual-Platform: ~
File Edit View Search Terminal Help

Lynx source distribution directory
Lynx source distribution and potpourri

Lynx is the text web browser. This is the top level page for the Lynx
software distribution site hosted by the Internet Software Consortium.

The current development sources have the latest version of Lynx
available (development towards 2.8.8). The main help page for
lynx-current is online; the current User Guide is part of the online
documentation.

The most recent stable release is lynx2-8-7. The main help page is
online, as well as the User Guide.

Other resources include:
* ftp and http mirrors
* Mailing list archives
* pgp/gpg signatures

Viewable with any browser; valid HTML.
Commands: Use arrow keys to move, '?' for help, 'q' to quit, '<-' to go back.
Arrow keys: Up and Down to move. Right to follow a link; Left to go back.
H)elp O)ptions P)rint G)o M)ain screen Q)uit /=search [delete]=history list
```

Рис. 24.1. Просмотр веб-страницы с помощью программы Lynx

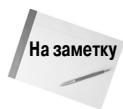
У читателя может возникнуть вопрос, как использовать графическую текстовую программу в сценариях командного интерпретатора. Дело в том, что в программе Lynx предусмотрено также средство, позволяющее выводить текстовое содержимое веб-страницы на устройство STDOUT. Такая возможность великолепно подходит для работы с данными, представленными на веб-странице. В настоящем разделе описано, как использовать программу Lynx в сценарии командного интерпретатора для извлечения данных с веб-сайтов Интернета.

Установка программы Lynx

Несмотря на то что программа Lynx имеет довольно почтенный возраст, ее разработка все еще активно продолжается. Ко времени написания этой книги новейшей версией Lynx была версия 2.8.8, выпущенная в июне 2010 года, и подготавливался новый выпуск. Программа Lynx широко применяется разработчиками сценариев командного интерпретатора, поэтому во многих дистрибутивах Linux предусмотрена ее установка по умолчанию.

Если вы используете установку, в которой не представлена программа Lynx, проверьте установочные пакеты применяемого вами дистрибутива. Наиболее вероятно, что эту программу удастся найти и легко установить.

Если же в применяемый дистрибутив не включен пакет Lynx или если оказалось, что уже имеется более новая версия, то можно загрузить исходный код с веб-сайта lynx.isc.org и откомпилировать его самостоятельно (при условии, что в системе Linux установлены библиотеки разработки C). Сведения о том, как компилировать и устанавливать пакеты дистрибутивов исходного кода, приведены в главе 8.



В программе Lynx используется текстовая графическая библиотека `curses` системы Linux. В большинстве дистрибутивов эта библиотека устанавливается по умолчанию. Если в применяемом дистрибутиве такая установка не предусмотрена, то просмотрите инструкции к данному конкретному дистрибутиву, касающиеся того, как установить библиотеку `curses`, и лишь после этого предпринимайте попытку откомпилировать Lynx.

В следующем разделе описано, как вызвать на выполнение команду `lynx` из командной строки.

Командная строка `lynx`

Интерфейс командной строки программы `lynx` предоставляет весьма разнообразные возможности, в связи с тем, что эта программа позволяет получать самую различную информацию с удаленного веб-сайта. При просмотре веб-страницы в браузере пользователь обычно видит лишь часть информации, переданной на локальный компьютер. Веб-страницы состоят из элементов данных трех типов:

- заголовки HTTP;
- cookie-файлы;
- содержимое HTML.

Заголовки HTTP предоставляют информацию о том, данные какого типа передаются в соединении, о сервере, передающем данные, а также о способе обеспечения безопасности, используемом в соединении. Если передаются специальные типы данных, такие как видео- или звукозаписи, сервер указывает это в заголовках HTTP. Программа `Lynx` позволяет просматривать все заголовки HTTP, отправленные в рамках сеанса работы с веб-страницей.

Пользователи, имеющие достаточный опыт путешествий в Интернете, наверняка знакомы с так называемыми *cookie-файлами*, применяемыми на веб-страницах. Cookie-файлы применяются веб-сайтами для сохранения на локальном компьютере данных о посещении веб-сайта для дальнейшего использования. Возможность сохранения cookie-файлов получает каждый отдельный сайт, но может иметь доступ только к тем данным, которые он сам оставил на компьютере. Команда `lynx` предоставляет возможность просматривать cookie-файлы, отправленные веб-серверами, а также отвергать или принимать конкретные cookie-файлы, отправленные с серверов.

Программа `Lynx` позволяет просматривать действительное содержимое веб-страницы в коде HTML в трех форматах:

- на текстовом графическом дисплее в терминальном сеансе с использованием графической библиотеки `curses`;
- в виде текстового файла, полученного путем непосредственного считывания данных с веб-страницы;
- в качестве текстового файла, который получен в результате непосредственного считывания исходного кода HTML с веб-страницы.

Для разработчиков сценариев командного интерпретатора возможность просмотра полученных в непосредственном виде данных или исходного кода HTML — просто золотая жила. После перехвата данных, полученных с веб-сайта, можно легко извлекать отдельные фрагменты информации.

Вполне очевидно, что применение программы `Lynx` позволяет получить исключительно широкие возможности. Но из этого следует, что работа с этой программой может быть сопряжена с определенными сложностями, особенно когда дело касается определения параметров командной строки. `Lynx` — это одна из наиболее сложных программ, с которыми приходится сталкиваться в мире Linux.

Команда `lynx` имеет следующий основной формат:

```
lynx options URL
```

где *URL* — адрес сайта HTTP или HTTPS, к которому необходимо подключиться, а *options* — одна или несколько опций, от которых зависит поведение Lynx при взаимодействии этой программы с удаленным веб-сайтом. Предусмотрены опции для взаимодействия с помощью Интернета почти любого типа, которое может потребоваться при использовании Lynx. В табл. 24.1 показаны все доступные параметры командной строки, которые можно использовать в сочетании с командой `lynx`.

Таблица 24.1. Параметры команды lynx

Параметр	Описание
-	Получать опции и параметры из устройства STDIN
-accept_all_cookies	Принимать cookie-файлы без вывода запросов на подтверждение, если задана обработка в режиме Set-Cookie. По умолчанию установлено значение <code>off</code>
-anonymous	Применять ограничения, относящиеся к анонимной учетной записи
-assume_charset=name	Определить применяемый по умолчанию набор символов для документов, в которых таковой не задан
-assume_local_charset=name	Определить применяемый по умолчанию набор символов для локальных файлов
-assume_unrec_charset=name	Определить набор символов, применяемый по умолчанию вместо нераспознанных наборов символов
-auth=id:pw	Задать информацию аутентификации для документов с установленным уровнем защиты
-base	Выводить комментарий к запросу URL и тег <code>BASE</code> перед выводом в формате <code>text/HTML</code> для потоков <code>-source</code>
-bibhost=URL	Задать URL локального сервера bibp (по умолчанию применяется значение <code>http://bibhost/</code>)
-book	Использовать страницу закладки в качестве начальной страницы. По умолчанию установлено значение <code>off</code>
-buried_news	Включить или отключить просмотр новостных статей, относящихся к скрытым ссылкам. По умолчанию устанавливается равным <code>on</code>
-cache=n	Определить количество документов, кешируемых в памяти
-case	Включить пользовательский поиск с учетом регистра. По умолчанию установлено значение <code>off</code>
-center	Включить или отключить выравнивание по центру в тегах <code><table></code> HTML. По умолчанию установлено значение <code>off</code>
-cfg=filename	Указать файл конфигурации, отличный от применяемого по умолчанию файла <code>lynx.cfg</code>
-child	Предусмотреть выход с помощью стрелки влево в исходном файле и отключить сохранение на диск
-cmd_log=filename	Регистрировать команды, задаваемые с помощью комбинаций клавиш, в указанном файле
-cmd_script=filename	Прочитать команды, задаваемые с помощью комбинаций клавиш, из указанного файла
-connect_timeout=n	Задать тайм-аут соединения (в секундах). Значение по умолчанию — 18 000 секунд
-cookie_file=filename	Указать файл, который используется для чтения cookie-файлов

Параметр	Описание
<code>-cookie_save_file=filename</code>	Указать файл, который используется для сохранения cookie-файлов
<code>-cookies</code>	Включить или отключить обработку заголовков Set-Cookie. По умолчанию устанавливается равным <code>on</code>
<code>-core</code>	Включить или отключить принудительный вывод дампов ядра после неисправимых ошибок. По умолчанию установлено значение <code>off</code>
<code>-crawl</code>	При использовании в сочетании с параметром <code>-traversal</code> выводить каждую страницу в файл с помощью параметра <code>-dump</code> и форматировать вывод, как обусловлено параметром <code>-traversal</code> , но осуществлять вывод на устройство STDOUT
<code>-curses_pads</code>	Использовать средство дополнения библиотеки <code>curses</code> для поддержки сдвига влево или вправо. По умолчанию устанавливается равным <code>on</code>
<code>-debug_partial</code>	Выводить последовательно формируемое изображение поэтапно, с учетом задержки на величину <code>MessageSecs</code> . По умолчанию установлено значение <code>off</code>
<code>-delay=n</code>	Задать задержку после сообщения <code>statusline</code> (в секундах). По умолчанию устанавливается значение <code>0.000</code>
<code>-display=display</code>	Задать переменную <code>display</code> для программ среды X Window
<code>-display_charset=name</code>	Определить набор символов для вывода на терминал
<code>-dont_wrap_pre</code>	Не переносить строки текста в разделах <code><pre></code> , если заданы параметры <code>-dump</code> и <code>-crawl</code> . Маркировать строки после разбивки в интерактивном сеансе. По умолчанию устанавливается равным <code>on</code>
<code>-dump</code>	Вывести содержимое, полученное из первого URL, на устройство STDOUT и выйти
<code>-editor=editor</code>	Включить режим редактирования с помощью указанного редактора
<code>-emacskeys</code>	Включить режим перемещения курсора по экрану с помощью команд, аналогичных применяемым в редакторе <code>emacs</code> . По умолчанию установлено значение <code>off</code>
<code>-enable_scrollback</code>	Включить или отключить режим совместимости с применением клавиш прокрутки. По умолчанию установлено значение <code>off</code>
<code>-error_file=filename</code>	Записать код статуса HTTP в указанный файл
<code>-exec</code>	Разрешить выполнение локальной программы
<code>-force_empty_hrefless_a</code>	Принудительно задать, что элементы <code><a></code> без атрибута <code>href</code> должны быть пустыми. По умолчанию установлено значение <code>off</code>
<code>-force_html</code>	Принудительно задать, что первый документ должен интерпретироваться как представленный в коде HTML. По умолчанию установлено значение <code>off</code>
<code>-force_secure</code>	Потребовать применения флага безопасности для cookie-файлов SSL. По умолчанию установлено значение <code>off</code>
<code>-forms_options</code>	Использовать меню опций на основе форм. По умолчанию устанавливается равным <code>on</code>
<code>-from</code>	Включить передачу заголовков <code>From</code> . По умолчанию устанавливается равным <code>on</code>
<code>-ftp</code>	Отключить доступ по FTP. По умолчанию установлено значение <code>off</code>

Параметр	Описание
<code>-get_data</code>	Прочитать данные для получения форм из устройства STDIN, заканчивающиеся знаками <code>---</code>
<code>-head</code>	Отправить запрос <code>HEAD</code> . По умолчанию установлено значение <code>off</code>
<code>-help</code>	Вывести сообщение с инструкцией по использованию
<code>-hiddenlinks=option</code>	Указать способ обработки скрытых ссылок. Параметр <i>option</i> может иметь значение <code>merge</code> , <code>listonly</code> или <code>ignore</code>
<code>-historical</code>	Использовать <code>></code> вместо <code>--></code> в качестве признака окончания комментариев. По умолчанию установлено значение <code>off</code>
<code>-homepage=URL</code>	Задать начальную страницу отдельно от исходной страницы
<code>-image_links</code>	Разрешить включение ссылок для всех изображений. По умолчанию установлено значение <code>off</code>
<code>-index=URL</code>	Задать применяемое по умолчанию имя файла индекса
<code>-ismap</code>	Разрешить использование ссылок <code>ISMAP</code> , если присутствуют клиентские ссылки <code>MAP</code> . По умолчанию установлено значение <code>off</code>
<code>-link=n</code>	Задать исходный отсчет для файлов <code>lnk#.dat</code> , формируемых при использовании параметра <code>-crawl</code> . По умолчанию применяется значение 0
<code>-localhost</code>	Запретить использование URL, которые указывают на удаленные узлы. По умолчанию установлено значение <code>off</code>
<code>-locexec</code>	Разрешить выполнение локальных программ только по отношению к локальным файлам. По умолчанию установлено значение <code>off</code>
<code>-mime_header</code>	Включить заголовки MIME и принудительно задать вывод исходного кода
<code>-minimal</code>	Использовать минимальную интерпретацию комментариев вместо интерпретации с проверкой допустимости. По умолчанию установлено значение <code>off</code>
<code>-nested_tables</code>	Использовать программные средства обработки вложенных таблиц. По умолчанию установлено значение <code>off</code>
<code>-newschunksize=n</code>	Задать количество статей во фрагментированных листингах новостей
<code>-newsmaxchunk=n</code>	Задать максимальное количество новостных статей в листингах перед разделением на фрагменты
<code>-nobold</code>	Отключить атрибут отображения полужирным шрифтом
<code>-nobrowse</code>	Отключить просмотр каталогов
<code>-nocc</code>	Запретить использование приглашений для задания значения поля <code>Cc:</code> , применяемого для автоматического копирования почтовых отправок. По умолчанию установлено значение <code>off</code>
<code>-nocolor</code>	Отключить поддержку цветов
<code>-noexec</code>	Запретить выполнение локальной программы. По умолчанию устанавливается равным <code>on</code>
<code>-nofilereferer</code>	Запретить передачу заголовков <code>Referer</code> , относящихся к URL файлов. По умолчанию устанавливается равным <code>on</code>
<code>-nolist</code>	Запретить применение средства формирования списка ссылок в выводимой информации. По умолчанию установлено значение <code>off</code>
<code>-nolog</code>	Запретить передачу по почте сообщений об ошибках владельцам документов. По умолчанию устанавливается равным <code>on</code>

Параметр	Описание
<code>-nonrestarting_sigwinch</code>	Указать, что обработчик изменения размеров окна не должен перезапускаться. По умолчанию установлено значение <code>off</code>
<code>-nopause</code>	Запретить принудительное применение пауз для сообщений <code>statusline</code>
<code>-noprint</code>	Отключить некоторые функции вывода, такие как <code>-restrictions=print</code> . По умолчанию установлено значение <code>off</code>
<code>-noredir</code>	Не выполнять команды перенаправления <code>Location:..</code> По умолчанию установлено значение <code>off</code>
<code>-noreferer</code>	Отключить передачу заголовков <code>Referer</code> . По умолчанию установлено значение <code>off</code>
<code>-noreverse</code>	Запретить использование атрибута негативного изображения
<code>-nostatus</code>	Запретить использование различных информационных сообщений. По умолчанию установлено значение <code>off</code>
<code>-nounderline</code>	Запретить использование атрибута подчеркивания в видеоизображении
<code>-number_fields</code>	Принудительно задать нумерацию ссылок, а также полей входных данных формы. По умолчанию установлено значение <code>off</code>
<code>-number_links</code>	Принудительно задать нумерацию ссылок. По умолчанию установлено значение <code>off</code>
<code>-partial</code>	Отображать страницы по частям в ходе загрузки. По умолчанию устанавливается равным <code>on</code>
<code>-partial_thres=n</code>	Задать количество строк, которые должны быть развернуты до того, как произойдет обновление изображения на дисплее при использовании поэтапного вывода. По умолчанию установлено значение <code>-1</code> , которое отключает это средство
<code>-pauth=id:pw</code>	Задать информацию аутентификации для защищенного прокси-сервера
<code>-popup</code>	Обрабатывать опции <code>SELECT</code> с одинарным выбором с помощью всплывающих окон, а не списков переключателей. По умолчанию установлено значение <code>off</code>
<code>-post_data</code>	Считывать с устройства <code>STDIN</code> данные для передачи форм, оканчивающиеся знаками <code>---</code>
<code>-preparsed</code>	Отображать прошедшие синтаксический анализ данные MIME типа <code>text/html</code> при использовании параметра <code>-source</code> , а также выводить их на экран в исходном виде, чтобы можно было понять, как действует программа Lynx при обработке недопустимого кода HTML. По умолчанию установлено значение <code>off</code>
<code>-prettysrc</code>	Использовать выделение синтаксических конструкций и обработку гиперссылок при выводе на экран исходного кода. По умолчанию установлено значение <code>off</code>
<code>-print</code>	Разрешить использование функций вывода. Это — параметр, противоположный по отношению к параметру <code>-noprint</code> . По умолчанию устанавливается равным <code>on</code>
<code>-pseudo_inlines</code>	Использовать псевдокоманды с модификаторами <code>ALT</code> во встроенном коде со строками без модификаторов <code>ALT</code> . По умолчанию устанавливается равным <code>on</code>

Параметр	Описание
-raw	Использовать применяемую по умолчанию настройку преобразований 8-битовых символов или режим CJK для исходной кодировки. По умолчанию установлено значение <code>off</code>
-realm	Ограничить доступ к URL в исходной области. По умолчанию установлено значение <code>off</code>
-reload	Сбросить на диск кеш прокси-сервера (это касается только первого документа). По умолчанию установлено значение <code>off</code>
-restrictions=options	Задать параметры ограничения. Опция <code>-restrictions</code> без параметров используется для просмотра списка
-resubmit_posts	Принудительно задать повторную передачу (без кеширования) форм с помощью метода <code>POST</code> , если документы, возвращаемые формами, были запрошены с помощью команды <code>PREV_DOC</code> или команды из списка ранее выполненных команд. По умолчанию установлено значение <code>off</code>
-rlogin	Отключить средство <code>rlogin</code> . По умолчанию установлено значение <code>off</code>
-selective	Затребовать файлы <code>.www_browsable</code> для определения каталогов, применимых для просмотра
-short_url	Разрешить проверку начала и конца длинного URL в строке статуса. По умолчанию установлено значение <code>off</code>
-show_cursor	При использовании значения <code>off</code> скрывать курсор в правом нижнем углу, в противном случае — отображать курсор. По умолчанию устанавливается равным <code>on</code>
-show_rate	Выводить на экран данные о скорости передачи. По умолчанию устанавливается равным <code>on</code>
-soft_dquotes	Использовать эмуляцию для устранения старой программной ошибки Netscape и Mosaic, из-за которой знак <code>></code> рассматривался как дополнительный признак завершения для двойных кавычек и тегов. По умолчанию установлено значение <code>off</code>
-source	Вывести содержимое первого URL в исходном виде на устройство <code>STDOUT</code> и выйти
-stack_dump	Отключить обработчик очистки <code>SIGINT</code> . По умолчанию установлено значение <code>off</code>
-startfile_ok	Разрешить применение исходной и начальной страницы в протоколе, отличном от <code>HTTP</code> , в сочетании с параметром <code>-validate</code> . По умолчанию установлено значение <code>off</code>
-stdin	Считывать файл запуска, <code>startfile</code> , из устройства <code>STDIN</code> . По умолчанию установлено значение <code>off</code>
-tagsoup	Использовать средство синтаксического анализа <code>TagSoup</code> , а не <code>SortaSGML</code> . По умолчанию установлено значение <code>off</code>
-telnet	Запретить использование сеансов <code>telnet</code> . По умолчанию установлено значение <code>off</code>
-term=term	Указать эмулируемый тип терминала
-tlog	Использовать журнал трассировки <code>Lynx</code> для текущего сеанса. По умолчанию устанавливается равным <code>on</code>

Параметр	Описание
-tna	Использовать режим Textfields Need Activation (Текстовые поля требуют активизации). По умолчанию установлено значение <code>off</code>
-trace	Использовать режим трассировки Lynx. По умолчанию установлено значение <code>off</code>
-trace_mask	Настроить режим трассировки Lynx. По умолчанию применяется значение 0
-traversal	Пройти по всем ссылкам HTTP, полученным из файла <code>startfile</code>
-trim_input_fields	Усекать входные поля <code>text/textarea</code> в формах. По умолчанию установлено значение <code>off</code>
-underline_links	Использовать атрибут подчеркивания или задания полужирного шрифта для ссылок. По умолчанию установлено значение <code>off</code>
-underscore	Использовать формат подчеркивания в выходных данных. По умолчанию установлено значение <code>off</code>
-use_mouse	Включить поддержку мыши. По умолчанию установлено значение <code>off</code>
-useragent=Name	Задать альтернативный заголовок User-Agent программы Lynx
-validate	Принимать только URL <code>http</code> (предназначенные для проверки допустимости), из чего следует больше ограничений, чем при использовании параметра <code>-anonymous</code> , но разрешать перенаправление для <code>http</code> и <code>https</code> . По умолчанию установлено значение <code>off</code>
-verbose	Использовать комментарии <code>[LINK]</code> , <code>[IMAGE]</code> и <code>[INLINE]</code> с именами файлов для соответствующих изображений. По умолчанию устанавливается равным <code>on</code>
-version	Вывести информацию о версии Lynx
-vikeys	Включить режим перемещения курсора по экрану с помощью команд, аналогичных применяемым в редакторе <code>vi</code> . По умолчанию установлено значение <code>off</code>
-width=n	Задать ширину экрана, применяемую при форматировании вывода. Значение по умолчанию составляет 80 столбцов
-with_backspaces	Задавать знаки возврата на один символ в выводе, если используется параметр <code>-dump</code> или <code>-crawl</code> . По умолчанию установлено значение <code>off</code>

Вполне очевидно, что параметры командной строки позволяют непосредственно управлять настройками HTTP и HTML почти любого типа. Например, если необходимо вывести данные в веб-форму с использованием метода `POST` протокола HTTP, то достаточно включить собственные данные в параметр `-post-data`. Если требуется сохранить `cookie`-файлы, полученные веб-сайтом, в каком-то особом месте, можно воспользоваться параметром `-cookie_save_file`.

Многие параметры командной строки определяют средства, от которых зависит функционирование программы Lynx при ее использовании в полноэкранном режиме, что позволяет настроить Lynx наилучшим образом с точки зрения удобства перехода по веб-страницам.

В обычно применяемой среде просмотра веб-сайтов параметры командной строки часто задаются не отдельно, а целыми группами. Поэтому в программе Lynx предусмотрена возможность не вводить эти параметры в командной строке каждый раз при вызове этой программы, а настраивать общий файл конфигурации, который определяет основные режимы поведения при использовании Lynx. Этот файл конфигурации рассматривается в следующем разделе.

Файл конфигурации Lynx

Команда `lynx` считывает многие настройки своих параметров из файла конфигурации. По умолчанию этот файл имеет имя `lynx.cfg` и находится в каталоге `/usr/local/lib`, но во многих дистрибутивах Linux вместо этого каталога применяется каталог `/etc` (`/etc/lynx.cfg`) (в дистрибутиве Ubuntu файл `lynx.cfg` находится в каталоге `/etc/lynx-curl`).

В файле конфигурации `lynx.cfg` взаимосвязанные параметры сгруппированы по разделам, благодаря чему поиск требуемых параметров становится проще. Файл конфигурации имеет следующий формат записей:

```
PARAMETER: value
```

где *PARAMETER* — полное имя параметра (которое часто, но не всегда, задается прописными буквами); *value* — значение, связанное с параметром.

Просматривая этот файл, можно найти много параметров, аналогичных параметрам командной строки, таких как параметр `ACCEPT_ALL_COOKIES`, определение которого эквивалентно заданию параметра командной строки `-accept_all_cookies`.

Предусмотрено также несколько параметров конфигурации, которые имеют аналогии среди параметров командной строки, но отличаются по имени. Значение параметра файла конфигурации `FORCE_SSL_COOKIES_SECURE` может быть переопределено с помощью параметра командной строки `-force_secure`.

Но предусмотрено также довольно значительное количество параметров конфигурации, не имеющих соответствия среди параметров командной строки. Значения таких параметров могут быть заданы только с применением файла конфигурации.

К числу наиболее широко применяемых параметров конфигурации, которые не могут быть заданы в командной строке, относятся параметры, определяющие работу *прокси-серверов*. В некоторых сетях (особенно во внутренних сетях учреждений и организаций) прокси-серверы используются в качестве посредника между браузером клиента и сервером веб-сайта, для доступа к которому применяется браузер. Вместо передачи HTTP-запросов непосредственно на удаленный веб-сервер браузеры клиентов должны отправлять свои запросы на прокси-сервер. Прокси-сервер, в свою очередь, отправляет запросы удаленному веб-серверу, получает результаты и перенаправляет их снова в браузер клиента.

На первый взгляд может показаться, что такая организация работы приводит к излишним затратам времени, но в действительности с ее помощью осуществляется крайне важная функция защиты клиентов от опасностей, которые кроются в Интернете. Прокси-сервер способен фильтровать неприемлемое содержимое и код, подготовленный со злыми намерениями, а также может обнаруживать узлы, используемые для реализации схем выуживания данных в Интернете с помощью фишинга (*фишинг* — это способ применения фиктивных серверов, имитирующих настоящие серверы, для перехвата конфиденциальных данных, передаваемых клиентами). Применение прокси-серверов может также способствовать сокращению использования пропускной способности каналов связи в Интернете, поскольку они кешируют обычно просматриваемые веб-страницы и возвращают их клиентам, что позволяет избавиться от необходимости снова и снова загружать одну и ту же исходную страницу.

Параметры конфигурации, используемыми для определения прокси-серверов, приведены ниже.

```
http_proxy:http://some.server.dom:port/  
https_proxy:http://some.server.dom:port/  
ftp_proxy:http://some.server.dom:port/  
gopher_proxy:http://some.server.dom:port/
```

```
news_proxy:http://some.server.dom:port/
newspost_proxy:http://some.server.dom:port/
newsreply_proxy:http://some.server.dom:port/
snews_proxy:http://some.server.dom:port/
snewspost_proxy:http://some.server.dom:port/
snewsreply_proxy:http://some.server.dom:port/
nntp_proxy:http://some.server.dom:port/
wais_proxy:http://some.server.dom:port/
finger_proxy:http://some.server.dom:port/
cso_proxy:http://some.server.dom:port/
no_proxy:host.domain.dom
```

Предусмотрена возможность определять разные прокси-серверы для каждого из сетевых протоколов, поддерживаемых программой Lynx. Можно также задать разделенный запятыми список веб-сайтов, к которым целесообразно предоставить непосредственный доступ без использования прокси-сервера. Для этого служит параметр `NO_PROXY`. В этом списке чаще всего присутствуют внутренние веб-сайты, которые не требуют фильтрации.

Переменные среды Lynx

Ознакомившись со всем разнообразием опций командной строки и параметров файла конфигурации, можно понять, насколько широкие возможности настройки предусмотрены в программе Lynx. Тем не менее подготовка этой программы к работе не ограничивается настройкой лишь указанных параметров. Для переопределения значений многих параметров файла конфигурации могут использоваться переменные среды. Иногда применяемая рабочая среда не позволяет получить доступ к файлу конфигурации `lynx.cfg`. В таком случае для присваивания других значений некоторым параметрам, заданным по умолчанию, могут использоваться локальные переменные среды. В табл. 24.2 перечислены наиболее широко применяемые переменные среды Lynx, с которыми часто приходится сталкиваться при использовании программы Lynx в ограниченной среде.

Таблица 24.2. Переменные среды Lynx

Переменная	Описание
<code>LYNX_CFG</code>	Указать местонахождение альтернативного файла конфигурации
<code>LYNX_LSS</code>	Указать местонахождение применяемой по умолчанию таблицы стилей для кодировки Lynx
<code>LYNX_SAVE_SPACE</code>	Указать местонахождение каталога на диске для сохранения файлов
<code>NNTPSERVER</code>	Указать сервер, используемый для получения и передачи новостей USENET
<code>PROTOCOL_PROXY</code>	Переопределить прокси-сервер для указанного протокола
<code>SSL_CERT_DIR</code>	Указать каталог, содержащий заверенные сертификаты для получения доступа к сайтам, заслуживающим доверия
<code>SSL_CERT_FILE</code>	Указать файл, содержащий заверенные сертификаты
<code>WWW_HOME</code>	Определить URL, используемый по умолчанию в программе Lynx при запуске

Эти переменные среды можно задавать перед вызовом на выполнение программы Lynx, как и любые другие переменные среды:

```
$ http_proxy=http://myproxy.com:8080
$ lynx
```

Чтобы задать параметры прокси-сервера, необходимо указать протокол, имя сервера и порт, используемый для обмена данными с прокси-сервером. Если возникает необходимость задать значение этой переменной, то обычно бывает целесообразно включить ее в общий файл запуска для командного интерпретатора (такой как файл `.bashrc` для командного интерпретатора `bash`), чтобы не задавать это значение каждый раз при вызове программы `Lynx`.

Перехват данных, поступающих из программы `Lynx`

Если вызов программы `Lynx` используется в сценарии командного интерпретатора, то чаще всего при этом предпринимается попытка получить определенный фрагмент (или несколько фрагментов) информации с веб-страницы. Для этого применяется способ, известный под названием *сбор данных с экрана*. При сборе данных с экрана предпринимается попытка найти программным путем данные, находящиеся в определенном месте на графическом экране, для их захвата и использования в сценарии командного интерпретатора.

Проще всего можно осуществить сбор данных с экрана с помощью программы `lynx` путем применения опции `-dump`. При использовании этой опции не предпринимается попытка отобразить веб-страницу на экране терминала. Вместо этого текстовые данные веб-страницы выводятся непосредственно на устройство `STDOUT`:

```
$ lynx -dump http://localhost/RecipeCenter/
The Recipe Center
    "Just like mom used to make"
Welcome
  [1]Home
  [2>Login to post
  [3]Register for free login

_____

  [4]Post a new recipe
```

Каждая ссылка обозначается номером тега, кроме того, `Lynx` вслед за данными веб-страницы отображает листинг всех ссылок на теги.

Зная о том, что в его распоряжении могут поступить все текстовые данные, содержащиеся на веб-странице, читатель, вероятнее всего, сумеет угадать, какими инструментами из набора доступных инструментов мы воспользуемся для извлечения данных. Так и есть. Мы прибегнем к использованию наших старых знакомых — программ `sed` и `gawk` (см. главу 18).

Прежде всего попытаемся определить, какие данные, представляющие интерес, можно собрать. В качестве примера рассмотрим веб-страницу `Yahoo!` с информацией о погоде, представляющую собой превосходный источник сведений о текущих погодных условиях во многих регионах мира. Для каждого региона используется отдельный URL для отображения сведений о погоде с привязкой к конкретному городу (о том, какой именно URL относится к тому или иному городу, можно узнать, перейдя на сайт `Yahoo!` в обычном браузере и указав название искомого города). Ниже приведена команда `lynx` для получения информации о погоде в Чикаго, штат Иллинойс.

```
lynx -dump http://weather.yahoo.com/united-states/illinois/chicago-2379574/
```

Выполнение этой команды приводит к выводу с веб-страницы огромного объема данных. Прежде всего необходимо найти именно ту информацию, которая требуется. Для этого следует перенаправить вывод команды `lynx` в файл, а затем отыскать в файле требуемые данные.

После выполнения этих действий с помощью приведенной выше команды можно обнаружить этот текст в выходном файле:

```
Current conditions as of 1:54 pm EDT
Mostly Cloudy
```

```
Feels Like:
    32 °F

Barometer:
    30.13 in and rising

Humidity:
    50%

Visibility:
    10 mi

Dewpoint:
    15 °F

Wind:
    W 10 mph
```

Приведенные данные содержат практически все сведения о текущей погоде, которые может потребоваться узнать. Однако полученный листинг имеет один небольшой недостаток. Можно видеть, что номера столбцов заданы в строке под заголовком. Попытка извлечь сведения лишь из отдельных нумерованных столбцов может оказаться сложной. В главе 18 уже было показано, как можно справиться с подобной проблемой.

Ключ к решению состоит в написании сценария `sed`, который обеспечивает вначале поиск в заголовке к данным. А после получения необходимых данных можно безошибочно переходить на ту строку, из которой должны быть извлечены данные. В этом примере упрощению нашей работы способствует удачная особенность, заключающаяся в том, что все требуемые данные находятся в строках, каждая из которых приведена отдельно. Это означает, что для решения рассматриваемой задачи достаточно только воспользоваться сценарием `sed`. Если бы в тех же строках находился также другой текст, то нам бы потребовалось прибегнуть к применению в качестве инструмента программы `gawk`, чтобы отфильтровать только необходимые данные.

Прежде всего необходимо создать сценарий `sed`, который производит поиск текста `Current conditions`, переходит к следующей строке для получения текста, описывающего текущие погодные условия, а затем выводит эти сведения. Ниже показано, как выглядят результаты нашей работы.

```
$ cat sedcond
/Current conditions/{
n
p
}
$
```

Адрес в команде указывает, что должен быть выполнен поиск строки с требуемым текстом. Если программа `sed` находит желаемую строку, то с помощью команды `n` осуществляется переход к следующей строке, после чего команда `p` выводит содержимое строки, представляющее собой текст с описанием текущих погодных условий для указанного города.

После этого нам потребуется сценарий `sed`, который может выполнить поиск текста `Feels Like:`, а затем перейти к следующей строке, чтобы извлечь и вывести данные о температуре:

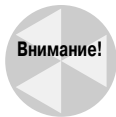
```
$ cat sedtemp
/Feels Like:/{
n
p
}
$
```

Просто идеальный результат. Итак, эти два сценария `sed` могут использоваться в сценарии командного интерпретатора, который вначале перехватывает вывод веб-страницы, полученный в программе `lynx`, и записывает его во временный файл, а затем применяет два сценария `sed` к данным веб-страницы для извлечения только требуемых данных. Ниже приведен пример того, как можно решить эту задачу.

```
$ cat weather
#!/bin/bash
# извлечение сведений о погоде в Чикаго, шт. Иллинойс

URL="http://weather.yahoo.com/united-states/illinois/chicago-2379574/"
LYNX='which lynx'
TMPFILE='mktemp tmpXXXXXX'
$LYNX -dump $URL > $TMPFILE
conditions='cat $TMPFILE | sed -n -f sedcond'
temp='cat $TMPFILE | sed -n -f sedtemp'
rm -f $TMPFILE
echo "Current conditions: $conditions"
echo The current temp outside is: $temp
$ ./weather
Current conditions: Mostly Cloudy
The current temp outside is: 32 °F
$
```

Этот сценарий поиска информации о погоде подключается к веб-странице Yahoo!, содержащей сведения о погоде в интересующем нас городе, сохраняет веб-страницу во временном файле, извлекает соответствующий текст, удаляет временный файл, а затем отображает данные о погоде. Преимущество этого подхода состоит в том, что после получения данных с веб-сайта можно выполнять с ними всевозможные операции обработки, например, создавать таблицы температур. Затем можно создать задание `cron` (см. главу 15), которое выполняется ежедневно и позволяет следить за температурой в каждые сутки.



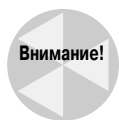
В Интернете постоянно происходят изменения. Поэтому не удивляйтесь, если, выполнив трудоемкую работу по определению точного местонахождения требуемых данных на веб-странице, вы через пару недель обнаружите, что в структуру интересующей вас веб-страницы внесены такие изменения, что нормальное функционирование всех сценариев стало невозможным. В действительности, возможно даже, что не будет правильно работать и тот пример, который приведен в данной книге, после выхода ее из печати. Но это не столь важно, если сам процесс извлечения данных с веб-страниц хорошо усвоен. Он остается применимым в любой ситуации, поэтому достаточно лишь вовремя вносить в свои сценарии необходимые исправления.

Программа cURL

Программа Lynx нашла широкое применение для решения указанной задачи, что послужило стимулом к созданию аналогичного программно продукта, который получил имя cURL. Программа cURL позволяет автоматически передавать файлы, указанные в командной строке, с помощью заданного URL. В настоящее время эта программа поддерживает протоколы FTP, FTPS, HTTP, HTTPS, SCP, SFTP, TFTP, telnet, DICT, LDAP, LDAPS и FILE, на применение каждого из которых указывает конкретный URL.

Программа cURL как таковая не используется в качестве браузера веб-страниц, но позволяет легко отправлять или получать данные автоматически с применением команд командной строки или сценариев командного интерпретатора, для чего достаточно задать одну простую команду. Благодаря наличию этой программы в наборе инструментов сценарной поддержки командного интерпретатора появляется еще один великолепный инструмент.

В настоящем разделе рассматривается процесс установки и использования программы cURL в сценариях командного интерпретатора.



Следует отметить, что имеется также язык программирования под названием curl, который принадлежит корпорации Sumisho Computer System Corporation и распространяется ею на коммерческой основе. Не следует путать программу cURL с языком программирования curl.

Установка программы cURL

Программа cURL находит все более широкое распространение, поэтому все чаще и чаще устанавливается в различных дистрибутивах Linux по умолчанию. Что же касается дистрибутива Ubuntu, то при его использовании программу cURL необходимо установить вручную из репозитория программного обеспечения. Для этого достаточно воспользоваться следующей командой:

```
$ sudo apt-get install curl
```

Если программа cURL не предусмотрена в применяемом вами дистрибутиве Linux или вы хотите получить новейшую версию, загрузите исходный код с веб-сайта curl.haxx.se и откомпилируйте его в своей системе Linux.

И в этом случае остаются в силе стандартные оговорки: для решения этой задачи необходимо, чтобы в системе Linux были установлены библиотеки разработки для языка C.

В следующем разделе речь пойдет о том, как использовать программу cURL в командной строке.

Получение веб-ресурсов с помощью программы cURL

По умолчанию программа cURL выводит весь объем кода HTML, относящегося к веб-странице, на устройство STDOUT:

```
$ curl http://www.google.com
<!doctype html><html><head><meta http-equiv="content-type" content=
"text/html; charset=ISO-8859-1"><title>Google</title>
[ листинг сокращен ]
```

Как и при работе с программой Lynx, можно воспользоваться стандартными средствами сценарной поддержки командного интерпретатора для извлечения отдельных элементов данных с загруженной веб-страницы.

Практика показывает, что программу cURL удобно использовать для пакетной загрузки файлов. Например, авторам данной книги часто приходится загружать ISO-файлы новейших дистрибутивов Linux. Но размеры этих ISO-файлов весьма велики, поэтому приходится так организовывать загрузку, чтобы выполнение этой операции не вынуждало нас бесполезно просиживать перед компьютером. Зная URL для ISO-файла, можно создать простой сценарий командного интерпретатора с использованием cURL, чтобы автоматизировать процесс загрузки:

```
$ cat downld
#!/bin/bash
# автоматическая загрузка новейшей версии файла cURL
curl -s -o /home/rich/curl-7.18.0.tar.gz
http://curl.haxx.se/download/curl-7.18.0.tar.gz
$
```

Опция командной строки `-s` переводит программу cURL в режим работы без вывода сообщений, при котором не происходит отправка каких-либо данных на устройство `STDOUT`. Опция командной строки `-o` применяется для перенаправления вывода в файл с указанным именем. Приведенный листинг не позволяет об этом судить, но фактически весь сценарий состоит из одной строки с вызовом команды `curl`. Это — простой сценарий, который приступает к работе и непосредственно загружает файл с веб-сайта с помощью cURL. После создания этого сценария пользователь получает возможность применить команду `at` или `cron` (см. главу 15), чтобы запланировать загрузку, допустим, на ночное время, когда он не работает за персональным компьютером или когда нагрузка сети снижается до минимума.

Работа в сети с помощью командного интерпретатора zsh

В главе 22 были описаны все средства, предусмотренные в командном интерпретаторе `zsh`. Как уже было сказано, командный интерпретатор `zsh` относится к поколению более новых командных интерпретаторов, предусмотренных для сред Linux и Unix. Одной из удобных возможностей командного интерпретатора `zsh` является применение внешних модулей. Разработчики `zsh` приняли решение не объединять все возможные средства в ядре командного интерпретатора `zsh`, а предусмотреть применение в нем специализированных модулей, позволяющих выбрать для загрузки необходимый набор команд. Одним из таких модулей является модуль `TCP`.

Благодаря модулю `TCP` командного интерпретатора `zsh` можно воспользоваться весьма привлекательными средствами работы с сетью с помощью лишь команд командной строки. Непосредственно с помощью командной строки можно создать полноценный сеанс обмена данными в сети `TCP` с другим сетевым устройством (или сценарием командного интерпретатора). В настоящем разделе рассматриваются средства модуля `TCP` командного интерпретатора `zsh`, а также показано простое приложение “клиент/сервер”, которое можно создать с использованием сценариев командного интерпретатора `zsh`.

Модуль TCP

В командном интерпретаторе `zsh` модули используются для добавления дополнительных средств к основному командному интерпретатору `zsh`. Каждый модуль содержит встроенные команды, которые относятся к определенной области. Модуль TCP предоставляет встроенные команды для большого набора средств работы в сети.

Для установки модуля TCP в командный интерпретатор `zsh` необходимо выполнить следующее:

```
% zmodload zsh/net/tcp
%
```

И этого достаточно для установки библиотек модуля в командный интерпретатор! Если в сценарии командного интерпретатора предусмотрено использование модуля TCP, то следует обязательно включить эту строку в сценарий. Модуль применяется только в текущем сеансе работы с командным интерпретатором.

После загрузки модуля TCP становится доступной команда `ztcp`, имеющая следующий формат:

```
ztcp [-acflLtv] [ -d fd] [args]
```

Применимые опции командной строки этой команды `args` перечислены ниже.

- `-a`. Подтвердить установление нового соединения.
- `-c`. Закрыть существующее соединение.
- `-d`. Использовать указанный дескриптор файла для соединения.
- `-f`. Принудительно закрыть соединение.
- `-l`. Открыть новый сокет для прослушивания.
- `-L`. Вывести список сокетов, подключенных в настоящее время.
- `-t`. Выйти, если отсутствуют запросы на установление соединения.
- `-v`. Отобразить подробную информацию о соединении.

В команде `ztcp` используется дескриптор файла для взаимодействия с открытым соединением TCP. По умолчанию в командном интерпретаторе `zsh` для ссылки на дескриптор файла используется переменная среды `$RESULT`. Все, что должен сделать разработчик, — это отправить данные в файл с дескриптором, указанным в переменной `$RESULT`, и модуль TCP перенаправит эти данные на удаленный узел. Аналогичным образом, если удаленный узел отправляет какие-либо данные, достаточно лишь считать их из файла с дескриптором, который указан в переменной `$RESULT`. Трудно представить более простой вариант организации работы в сети!

Подход на основе принципа “клиент/сервер”

Прежде чем приступить к изучению процесса создания программ “клиент/сервер” с использованием командного интерпретатора `zsh`, рассмотрим, как именно работают программы клиента и сервера. Очевидно, что каждая из этих программ выполняет разные функции при создании соединения и передаче данных.

Программа *сервера* прослушивает сеть на наличие запросов, поступающих от *клиентов*. Клиенты инициируют запросы к серверу для создания соединений. После приема сервером запроса на создание соединения открывается двухсторонний канал связи между клиентом и сервером для передачи и получения данных. Этот процесс показан на рис. 24.2.

Схема взаимодействия “клиент–сервер”



Рис. 24.2. Схема связи между клиентом и сервером

Как показано на рис. 24.2, сервер должен выполнить две функции, прежде чем появится возможность начать обмен данными с клиентом. Прежде всего должен быть определен конкретный порт TCP для прослушивания входящих запросов. Затем, после поступления запроса на создание соединения, сервер должен подтвердить установление нового соединения.

Функции клиента много проще. Все его действия заключаются в том, чтобы предпринять попытку подключиться к серверу через конкретный порт TCP, прослушиваемый сервером. Если сервер подтверждает установление нового соединения, становится доступной двухсторонняя связь и появляется возможность передавать данные.

После установления соединения между сервером и клиентом они обмениваются данными в соответствии с определенным *протоколом* (так называются правила обмена данными). В противном случае, если оба этих сетевых объекта будут пытаться выполнить одновременно одну и ту же операцию, возникнет взаимная блокировка и обмен данными станет невозможным. В качестве примера можно указать, что при попытке обеих сторон отправить сообщения в одно и то же время эти сообщения так и не будут получены.

Поэтому на сетевого программиста возложена задача определения правил протокола, которым должны следовать программы клиента и сервера.

Программирование “клиент/сервер” с применением командного интерпретатора `zsh`

Подготовим простое сетевое приложение, чтобы продемонстрировать создание среды “клиент/сервер” с помощью программы `ztcp`. Создаваемая программа сервера должна прослушивать запросы на установление соединения через порт 5150 протокола TCP. При поступлении запроса на соединение сервер принимает его, а затем отправляет клиенту приветственное сообщение.

Затем программа сервера переходит в состояние ожидания для получения сообщения от клиента. Если сообщение будет получено, сервер отображает его, а затем отправляет в неиз-

менном виде обратно клиенту. После отправки сообщения сервер переходит по циклу в режим прослушивания для приема следующего сообщения. Этот цикл продолжается до тех пор, пока сервер не получит сообщение, состоящее из текста `exit`. После этого сервер оканчивает сеанс.

Создаваемая клиентская программа должна отправлять запрос на создание соединения серверу через порт 5150 протокола TCP. После установления соединения клиент должен получить приветственное сообщение от сервера.

После получения сообщения клиент отображает его, а затем передает запрос пользователю на ввод данных, которые должны быть отправлены на сервер. После получения сообщения от пользователя клиентская программа отправляет его на сервер и переходит в состояние ожидания, чтобы получить то же сообщение назад. Если это сообщение возвращается с сервера, клиент отображает сообщение и переходит по циклу к этапу запроса еще одного сообщения от пользователя. Этот цикл продолжается до тех пор, пока пользователь не введет текст `exit`. После того как это происходит, клиент отправляет серверу текст с запросом на завершение работы, а затем оканчивает сеанс.

Программы сервера и клиента приведены в следующих разделах.

Программа сервера

Ниже приведен код программы сервера.

```
% cat server
#!/bin/zsh
# сценарий сервера TCP для zsh
zmodload zsh/net/tcp
ztcp -l 5150
fd=$REPLY

echo "Waiting for a client..."
ztcp -a $fd
clientfd=$REPLY
echo "client connected"

echo "Welcome to my server" >& $clientfd

while [ 1 ]
do
    read line <& $clientfd
    if [[ $line = "exit" ]]
    then
        break
    else
        echo Received: $line
        echo $line >& $clientfd
    fi
done
echo "Client disconnected session"
ztcp -c $fd
ztcp -c $clientfd
%
```

Эта программа сервера разработана на основе принципа “клиент/сервер”, который проиллюстрирован на рис. 24.2. В этой программе прежде всего используется параметр `-l` для указания порта (5150), через который должно выполняться прослушивание. Переменная `$RESULT`

содержит дескриптор файла, возвращаемый системой Linux для идентификации соединения. В программе сервера используется параметр `-a` для подтверждения установления нового соединения. В соответствии с этой командой должно происходить ожидание до поступления нового запроса на создание соединения (такая организация работы именуется *блокирующей*). В сценарии не будут выполняться какие-либо действия до приема запроса на соединение.

В каждом клиентском соединении используется дескриптор файла, отличный от дескриптора файла, который соответствует порту прослушивания. Это позволяет при желании поддерживать сразу несколько клиентских соединений (такая возможность в этом простом примере не используется).

После подтверждения установления нового соединения сервер отправляет приветственное сообщение клиенту с помощью файла с соответствующим дескриптором:

```
echo "Welcome to my server" >& $clientfd
```

Все действия, необходимые для передачи данных удаленному клиенту, выполняет модуль TCP командного интерпретатора `zsh`.

После этого программа сервера входит в бесконечный цикл. В этом цикле используется команда `read`, которая обеспечивает ожидание поступления ответных данных от клиента:

```
read line <& $clientfd
```

Данная команда также блокирует выполнение сценария до тех пор, пока не будут получены данные от клиента. При этом одной из неприятных ситуаций может оказаться разрыв соединения с клиентом, установленного в сети. Для предотвращения этой проблемы можно использовать опцию `-t` в команде чтения для указания значения тайм-аута (в секундах). Если сервер не получает данные от клиента до завершения тайм-аута, то продолжает выполнение сценария.

Если же сервер получает данные от клиента, то выводит их на устройство `STDOUT`, а затем отправляет назад клиенту. Если при сравнении данных с текстовой строкой `exit` обнаруживается равенство, сервер выходит из цикла, а в программе `ztcp` используется параметр `-c` для закрытия дескриптора файла клиента и дескриптора файла порта прослушивания. Если требуется, чтобы после закрытия дескриптора файла клиента сервер перешел к прослушиванию еще одного запроса на создание соединения, то можно организовать переход по циклу в режим ожидания приема нового соединения.

Программа клиента

Ниже приведен код программы клиента в виде сценария командного интерпретатора.

```
% cat client
#!/bin/zsh
# сценарий клиента TCP для zsh
zmodload zsh/net/tcp

ztcp localhost 5150
hostfd=$REPLY

read line <& $hostfd
echo $line

while [ 1 ]
do
    echo -n "Enter text: "
    read phrase
    echo Sending $phrase to remote host...
```

```

echo $phrase >& $hostfd
if [[ $phrase = "exit" ]]
then
    break
fi
read line <& $hostfd
echo "    Received: $line"
done
ztcp -c $hostfd
%
```

В программе клиента должен быть указан IP-адрес (или имя хоста) в системе, по которому можно получить доступ к работающей программе сервера, а также номер порта TCP, соответствующий тому, в котором сервер осуществляет прослушивание. После подтверждения сервером установления соединения программа `ztcp` задает дескриптор файла для соединения и сохраняет это значение в переменной `$REPLY`. В программе клиента происходит чтение ответственного сообщения сервера, а затем его отображение:

```

read line <& $hostfd
echo $line
```

После этого в программе клиента осуществляется переход в цикл `while`, в котором пользователю передается запрос на ввод текста, подлежащего передаче серверу, затем введенный текст считывается и передается на сервер. После отправки текста проверяется, не представлял ли собой введенный текст строку `exit`. В случае положительного ответа происходит выход из цикла и закрытие дескриптора файла, что влечет за собой закрытие соединения TCP. Если текст отличался от `exit`, в программе клиента происходит ожидание ответа от сервера, а затем его отображение.

Выполнение программ

Обе эти программы можно вызвать на выполнение либо на двух отдельных системах Linux в сети, либо в двух разных терминальных сеансах в одной и той же системе. Необходимо запускать программу сервера в первую очередь, чтобы она была доступной для прослушивания входящих соединений ко времени запуска программы клиента:

```
% ./server
Waiting for a client...
```

Затем можно приступить к запуску программы клиента:

```
% ./client
Welcome to my server
Enter text: test
Sending test to remote host...
Received: test
```

После подключения клиента об этом можно узнать на сервере:

```
client connected
Received: test
```

Обмен данными между клиентом и сервером продолжается до тех пор, пока пользователь не введет текст `exit` в программе клиента:

```
Enter text: exit
Sending exit to remote host...
%
```

После этого должно быть показано, как происходит автоматический выход из программы сервера:

```
Client disconnected session
%
```

Таким образом, вы уже ознакомились с основами создания полнофункциональных сетевых программ! С помощью командного интерпретатора `zsh` и модуля TSP можно легко обеспечить обмен данными между сценариями командного интерпретатора, работающими на разных системах в сети.

Резюме

В этой главе приведены основные сведения, позволяющие обеспечить взаимодействие сценариев командного интерпретатора с Интернетом. Одним из наиболее широко применяемых при этом инструментов является `Lynx` — программа командной строки, которая позволяет выводить информацию веб-сайта на дисплей в терминальном сеансе с применением графических средств текстового режима. В дополнение к этой возможности программа `Lynx` предоставляет также способ получения с веб-сайта только бесформатных данных и вывод их на устройство `STDOUT`. Программу `Lynx` можно использовать для получения данных с веб-сайта и последующей интерпретации их с помощью стандартных инструментов обработки текста в системе `Linux`, таких как `sed` и `gawk`, для поиска конкретной информации.

Еще одним удобным инструментом, обеспечивающим взаимодействие с Интернетом, является программа `cURL`. Программа `cURL` также позволяет выводить данные, полученные с веб-сайта, в непосредственном виде. Кроме того, эта программа предоставляет возможность легко создавать сценарии загрузки файлов с серверов многих типов.

Наконец, в главе было показано, как использовать модуль TSP командного интерпретатора `zsh` для написания собственных сетевых программ. Командный интерпретатор `zsh` предоставляет удобный способ взаимодействия сценариев командного интерпретатора, выполняемых на отдельных системах в сети.

В следующей главе будет показано, как использовать электронную почту в сценариях командного интерпретатора. Например, во многих ситуациях, когда сценарии командного интерпретатора используются для автоматизации процессов, возникает необходимость получать сообщения, по которым можно судить, был ли процесс завершен успешно или окончился неудачей. Знания о том, как работать с программным обеспечением электронной почты, установленным в конкретной системе, позволяют легко отправлять сообщения автоматически по всем возможным адресам.

Использование электронной почты

В наши дни электронная почта нашла настолько широкое распространение, что адрес электронной почты имеется практически у каждого пользователя. Поэтому у многих сложилась привычка получать и просматривать данные с помощью электронной почты, а не знакомиться с нужными сведениями с помощью файлов или распечаток. Точно так же обстоят дела в мире сценарной поддержки, предоставляемой командным интерпретатором. Если предусмотрено формирование отчета какого-либо типа с помощью сценария командного интерпретатора, то, по всей вероятности, в какой-то момент возникнет необходимость отправить эти полученные результаты какому-то другому лицу по электронной почте. В настоящей главе показано, как осуществляется настройка системы Linux для передачи электронной почты непосредственно из сценариев командного интерпретатора. В этой главе также описано, как проверить способность системы Linux отправлять исходящие почтовые сообщения и добиться нормальной работы почтового клиента с тем, чтобы операции отправки почты можно было выполнять из командной строки. Но прежде всего в главе представлен краткий обзор способов, которые в целом обеспечивают работу с электронной почтой в системе Linux.

Основы электронной почты Linux

Иногда наибольшие трудности, с которыми приходится сталкиваться при организации использования электронной

ГЛАВА

25

В этой главе...

Основы электронной почты
Linux

Установка сервера

Отправка сообщений
с помощью программы Mailx

Программа Mutt

Резюме

почты в сценариях командного интерпретатора, состоят в понимании того, как работает система электронной почты в Linux. Для этого существенно важно знать, какие пакеты программ выполняют конкретные задачи по доставке электронной почты из сценариев командного интерпретатора в папку “Входящие”. В настоящем разделе представлены основы того, как используется электронная почта в различных версиях Linux и какое программное обеспечение должно быть установлено, прежде чем появится возможность ее использовать.

Электронная почта в Linux

Одна из основных целей, которую поставили перед собой разработчики операционной системы Unix, состояла в том, чтобы добиться модульности программного обеспечения. Создатели системы Unix отказались от разработки единственной монолитной программы, которая выполняла бы все возможные задачи, и самые сложные, и самые простые, и занялись подготовкой совокупности небольших программ, каждая из которых способна выполнять отдельные функции из состава общих функциональных возможностей системы.

Этот подход использовался при реализации систем электронной почты, предусмотренных в Unix, а затем был перенесен в среду Linux. В системе Linux функции электронной почты разделены на отдельные наборы функций, а для осуществления каждого из этих наборов функций предусмотрена отдельная программа. На рис. 25.1 показано, как осуществляется модульная организация функций электронной почты в среде Linux в большинстве комплексов программного обеспечения электронной почты с открытым исходным кодом.

Модульная среда электронной почты Linux

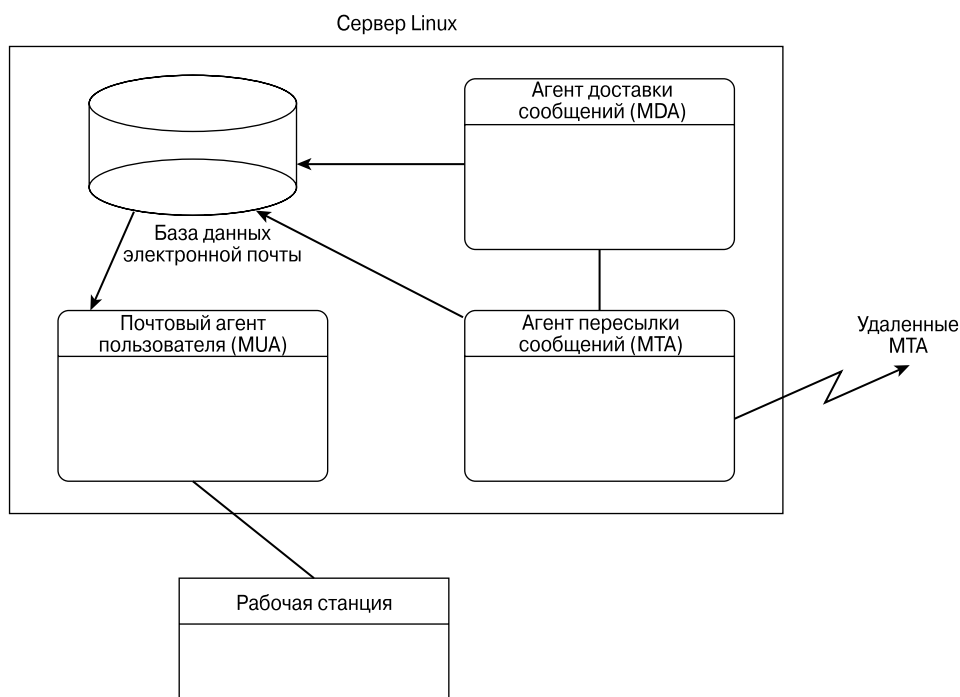


Рис. 25.1. Модульная среда электронной почты в системе Linux

Как показано на рис. 25.1, в среде Linux процесс обработки электронной почты обычно разделен между следующими тремя функциональными единицами:

- агент пересылки сообщений (Mail Transfer Agent — MTA);
- агент доставки сообщений (Mail Delivery Agent — MDA);
- почтовый агент пользователя (Mail User Agent — MUA).

Между этими тремя функциональными единицами часто бывает сложно провести четкий “водораздел”. В некоторых пакетах программ электронной почты объединяются функциональные возможности MDA и MTA, тогда как в других сочетаются функции MUA и MDA. В следующих разделах эти основные компоненты электронной почты описаны более подробно и показано, как они реализованы в системе Linux.

Агент пересылки сообщений

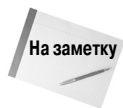
Ядром системы электронной почты в Linux является программное обеспечение MTA, которое отвечает за обработку и входящих, и исходящих почтовых сообщений в системе. Для каждого исходящего почтового сообщения программа MTA должна определить назначение в виде адресов получателей. Если узлом назначения является локальная система, MTA может либо доставить сообщение непосредственно в локальный почтовый ящик, либо передать сообщение локальной программе MDA для доставки.

Но если узлом назначения является удаленный почтовый сервер, то программа MTA должна установить канал связи с программным обеспечением MTA на удаленном узле, чтобы передать сообщение. Предусмотрены два основных метода, которые применяются в пакетах программ MTA для доставки почты на удаленные узлы:

- непосредственная доставка;
- доставка через прокси-сервер.

Если система Linux непосредственно подключена к Интернету, то в ней может часто происходить доставка сообщений, предназначенных для получателей на удаленных узлах, непосредственно на эти удаленные узлы. В программном обеспечении MTA используется система доменных имен (Domain Name System — DNS) для однозначного разрешения сетевого IP-адреса в целях доставки сообщения электронной почты, а затем устанавливается сетевое соединение с помощью протокола SMTP (Simple Mail Transfer Protocol — SMTP).

Но часто возникают ситуации, в которых узел не подключен непосредственно к Интернету, или нежелательно, чтобы обмен с удаленными узлами происходил напрямую. В таких ситуациях обычно используется так называемый *промежуточный узел* — прокси-сервер, который принимает сообщения электронной почты из системы Linux, а затем предпринимает попытки доставить их непосредственно назначенному получателю.



Организация работы в Интернете с применением промежуточных узлов становится все более затруднительной из-за постоянного увеличения количества спама. Одиночный сервер, принадлежащий мошенникам, может рассылать тысячи сообщений так называемой незапрашиваемой рекламной электронной почты (unsolicited commercial e-mail — UCE), скрываясь за промежуточным узлом, чтобы его было сложнее идентифицировать. Поэтому на большинстве промежуточных узлов теперь предъявляется требование пройти аутентификацию в той или иной форме, прежде чем будет предоставлена возможность перенаправлять свои сообщения на другие узлы.

Что касается входящих сообщений, то программа MTA должна быть способна принимать запросы на установление соединений от удаленных почтовых серверов и получать сообщения, предназначенные для локальных пользователей. Опять-таки протоколом, наиболее широко применяемым в этом процессе, остается SMTP.

В среде Linux предусмотрено большое количество программ MTA различных типов с открытым исходным кодом. Каждая из этих программ обладает собственным набором средств, которые позволяют провести различия между разными программами. Но к числу программ, которые превосходят все прочие по своей распространенности, безусловно, относятся следующие:

- sendmail;
- Postfix.

Эти два пакета MTA электронной почты более подробно будут рассматриваться в разделе “Установка сервера”.

Агент доставки сообщений

Назначение программы MDA состоит в том, чтобы доставить сообщение, предназначенное для локального пользователя. Эта программа получает сообщения от программы MTA, а затем она должна точно определить, как и куда доставить эти сообщения. На рис. 25.2 показано, как программа MDA взаимодействует с программой MTA при доставке электронной почты.

Использование программы MDA на почтовом сервере

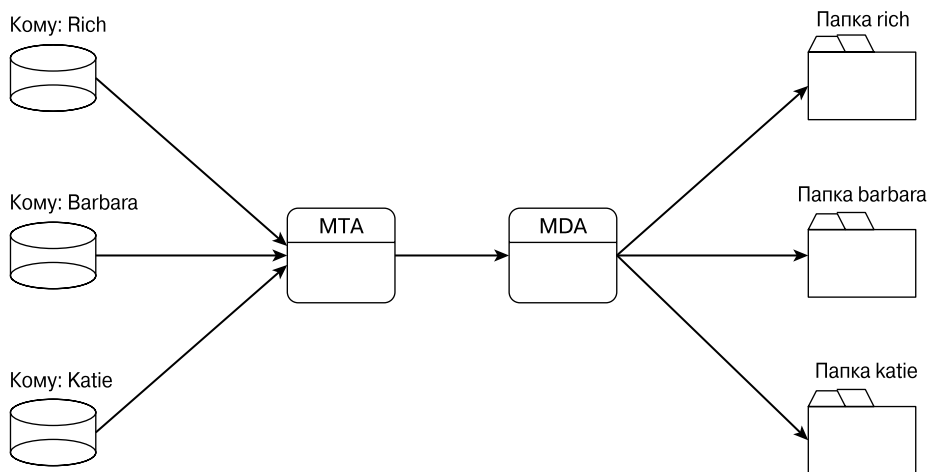


Рис. 25.2. Использование программы MDA на почтовом сервере

Безусловно, иногда функции MDA выполняются непосредственно в самой программе MTA, но чаще всего реализация электронной почты в системе Linux основана на использовании отдельной программы MDA для доставки сообщений локальным пользователям. Функции программ MDA сосредоточены исключительно на доставке почты локальным пользователям, поэтому разработчики этих программ получают возможность оснастить их более привлекательными пользовательскими интерфейсами, что не всегда осуществимо по отношению к программам MTA, включающим функциональные средства MDA. В результате администратор электронной почты может предложить пользователям электронной почты дополнительные

возможности, такие как фильтрация почты на предмет обнаружения спама, перенаправление из-за пределов офиса и автоматическая сортировка почты.

Получив сообщение, программа MDA должна доставить его в надлежащее место — либо в почтовый ящик локального пользователя, либо в альтернативное место, определенное локальным пользователем.

В настоящее время в системе Linux наиболее широко применяются пользовательские почтовые ящики трех типов:

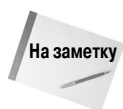
- файлы `/var/spool/mail` или `/var/mail`;
- файлы `$HOME/mail`;
- каталоги почтовых ящиков в стиле Maildir.

Каждый из этих типов почтовых ящиков обладает собственными возможностями, благодаря которым они становятся привлекательными для использования. В большинстве дистрибутивов Linux для размещения отдельных файлов почтовых ящиков применяется каталог `/var/spool/mail` или `/var/mail`, по одному файлу для каждой учетной записи пользователя в системе. Это центральное местоположение всех файлов почтовых ящиков определено в системе, поэтому программы MUA обладают информацией о том, где находятся файлы почтовых ящиков тех или иных пользователей.

Некоторые дистрибутивы Linux предоставляют возможность перемещать отдельные файлы почтовых ящиков в каталоги `$HOME` конкретных пользователей. Это обеспечивает большую безопасность, поскольку каждый файл почтового ящика находится в той области, для которой уже установлены надлежащие права доступа.

Почтовые ящики в стиле Maildir представляют собой относительно новое средство, поддерживаемое некоторыми более развитыми приложениями MTA, MDA и MUA. При этом каждое сообщение не включается в состав файла почтового ящика, а сохраняется в виде отдельного файла в каталоге, который теперь играет роль почтового ящика. Это помогает уменьшить вероятность повреждения почтового ящика, поскольку теперь из-за отдельного сообщения не должна быть нарушена структура всего почтового ящика.

В целом каталоги почтовых ящиков в стиле Maildir обеспечивают более высокую производительность, безопасность и отказоустойчивость, но многие широко применяемые программы MDA и MUA не предоставляют возможность использовать их. Однако почти все программы MDA и MUA обладают способностью работать с файлами почтовых ящиков `/var/spool/mail`.



Исходным местоположением почтовых ящиков в системе Unix является каталог `/var/spool/mail`. Это соглашение об именовании файлов применяется в большинстве дистрибутивов Linux, однако в некоторых из них предусмотрено применение каталога `/var/mail`.

Если в какой-то конкретной системе предусмотрено использование отдельной программы MDA для обработки входящих сообщений электронной почты, то с наибольшей вероятностью таковой является широко распространенная программа Procmail. Она позволяет каждому отдельному пользователю создавать настраиваемый файл конфигурации для определения фильтров электронной почты, создания назначений для перенаправления из-за пределов офиса, а также развертывания отдельных почтовых ящиков.

Почтовый агент пользователя

В предыдущих разделах описывалось, как электронная почта перемещается с удаленного узла на локальный узел, а затем в почтовый ящик отдельного пользователя. Следующим шагом

в этом процессе является предоставление возможности отдельным пользователям просматривать свои сообщения электронной почты.

В модели электронной почты Linux для каждого пользователя предусмотрен локальный файл или каталог почтового ящика, в котором хранятся сообщения этого пользователя. Задача программы MUA — предоставить пользователю возможность обращаться к своим почтовым ящикам для чтения сообщений.

Важно помнить, что программы MUA не получают сообщения; они лишь отображают сообщения, которые уже находятся в почтовом ящике. Многие программы MUA предоставляют также возможность создавать отдельные папки для электронной почты, чтобы пользователь имел возможность перемещать письма из предусмотренного по умолчанию почтового ящика (который часто носит имя Inbox, или “Входящие”) в отдельные папки для лучшей организации.

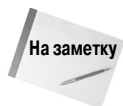
Большинство программ MUA предоставляет также возможность отправлять сообщения. В таком случае распределение функций становится менее четким, поскольку, как уже было сказано, задача отправки сообщений электронной почты возложена на программы MTA.

Для выполнения этой функции в большинстве программ MUA применяются функции промежуточного узла, определяемые протоколом SMTP. Таким образом, должна быть предусмотрена автоматическая передача сообщений программой MUA в локальную программу MTA для доставки или должен быть определен удаленный промежуточный узел в конфигурации MUA, на который должны передаваться сообщения для доставки.

В течение многих лет для платформы Linux было разработано много разных программ MUA с открытым исходным кодом. В следующих разделах описаны некоторые из наиболее широко применяемых программ MUA, с которыми приходится сталкиваться при работе в системе Linux.

Программа Mailx

Наиболее широко применяемой программой MUA с командной строкой для среды Linux является программа Mailx. Во всех установках операционной системы программа Mailx устанавливается с исполняемым файлом mail, а это означает, что Mailx заменяет программу mail, а не разворачивается как отдельная программа.



В дистрибутивах Linux, в основном рассчитанных на применение графического рабочего стола, таких как Ubuntu и openSUSE, программа Mailx с командной строкой не устанавливается по умолчанию. В случае необходимости эту программу следует установить вручную (см. главу 8) в составе пакета почтового клиента. Тем не менее необходимо соблюдать осторожность, поскольку пакеты программ электронной почты в разных дистрибутивах Linux не всегда именуются одинаково. Пакет для Ubuntu именуется mailutils, и в нем используется пакет Mailutils GNU, а не Mailx. Пакет Mailutils предоставляет такие же функциональные возможности, но рассчитан на применение немного других параметров командной строки.

Программа Mailx позволяет пользователям получать доступ к своим почтовым ящикам для чтения хранящихся в них сообщений, а также отправлять сообщения другим пользователям электронной почты, применяя исключительно программы командной строки. Ниже приведен типичный сеанс работы с программой Mailx.

```
$ mail
"/var/mail/rich": 2 messages 2 new
>N  1 Rich Blum      Thu Dec  9 10:07 13/579  Test message
   N  2 Rich Blum      Thu Dec  9 10:08 13/593  This is another test
? 1
```

```

Return-Path: <rich@rich-Parallels-Virtual-Platform>
X-Original-To: rich@rich-Parallels-Virtual-Platform
Delivered-To: rich@rich-Parallels-Virtual-Platform
Received: by rich-Parallels-Virtual-Platform (Postfix, from
↳ userid 1000)
    id 5C03F2606CF; Thu,  9 Dec 2010 10:07:48 -0500 (EST)
To: <rich@rich-Parallels-Virtual-Platform>
Subject: Test message
X-Mailer: mail (GNU Mailutils 2.1)
Message-Id: <20101209150748.5C03F2606CF@rich-Parallels-Virtual-
↳ Platform>
Date: Thu,  9 Dec 2010 10:07:48 -0500 (EST)
From: rich@rich-Parallels-Virtual-Platform (Rich Blum)

```

```

This is a test message
? d
? q
Held 1 message in /var/mail/rich
$

```

В первой строке показан вызов программы Mailx на выполнение без опций командной строки. По умолчанию это позволяет пользователю проверить сообщения в своем почтовом ящике. После ввода команды mail отображается сводка по всем сообщениям в почтовом ящике пользователя. Программа Mailx позволяет читать сообщения только в формате /var/mail или \$HOME/mail. Эта программа не обладает способностью обрабатывать почту с использованием формата, основанного на применении почтовых каталогов Maildir.

Каждый пользователь имеет отдельный файл, в котором содержатся все его сообщения. В некоторых дистрибутивах Linux файл почтового ящика пользователя создается только после получения первого сообщения, предназначенного для учетной записи этого пользователя. Этот файл обычно имеет имя, совпадающее с регистрационным именем пользователя в системе, и находится в каталоге mailbox системы. Таким образом, все сообщения для пользователя rich хранятся в файле /var/mail/rich в системе Linux. По мере поступления новых сообщений для пользователя происходит их добавление к концу файла.

С помощью программы командной строки mail можно также отправлять сообщения электронной почты:

```

$ mail barbara
Cc:
Subject: This is a test sent to Barbara
Hello Barbara -
This is a test message I'm sending from the command line.
$

```

Имя получателя включено в командную строку с именем программы. Программа Mailx запрашивает у пользователя ввод адресов, по которым должны быть разосланы копии (с помощью приглашения Cc:), и темы сообщения, Subject:. После этого программа дает возможность ввести текст сообщения. Чтобы завершить ввод сообщения, необходимо нажать комбинацию клавиш <Ctrl+D>. Затем программа Mailx предпринимает попытки передать сообщение программе MTA для доставки.

Вводные сведения о программе Mutt

По мере дальнейшего усовершенствования среды Unix все более замысловатыми становились и программы MUA. Одна из первых попыток обеспечить применение графических средств в системе Unix состояла в создании графической библиотеки `ncurses`. Программа, в основе которой лежит библиотека `ncurses`, позволяет управлять местоположением курсора на экране терминала и размещать символы почти в любом месте экрана.

Одной из программ MUA, в которой применяется библиотека `ncurses`, является Mutt. После запуска программы Mutt на экране терминала разворачивается удобное меню, в котором перечислены сообщения, по аналогии с тем, как выглядит вывод программы Mailx. Пользователь может выбирать сообщения и просматривать их на дисплее, как показано на рис. 25.3.

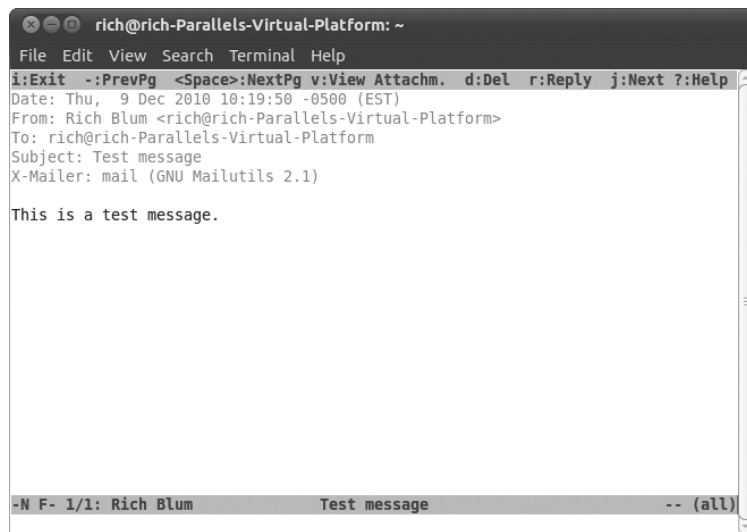


Рис. 25.3. Программа Mutt

В программе Mutt для выполнения стандартных функций, таких как чтение сообщений и инициализация новых сообщений, используются комбинации клавиш. Возможно, наиболее полезным средством для разработчиков сценариев командного интерпретатора является возможность отправлять сообщения непосредственно из командной строки, без перехода в текстовый графический режим. Программа Mutt будет рассматриваться более подробно ниже, в разделе “Программа Mutt”.

Графические почтовые клиенты

Почти во всех системах Linux поддерживается графическая среда X Window. Система X Window используется во многих программах MUA электронной почты для отображения содержимого сообщения. Двумя наиболее широко применяемыми графическими программами MUA являются следующие:

- KMail для среды поддержки окон KDE;
- Evolution для среды поддержки окон GNOME.

Каждый из этих пакетов обеспечивает работу пользователя со своим локальным почтовым ящиком Linux, а также подключение к удаленным почтовым серверам для чтения почтовых сообщений. Типичный пример экрана сеанса Evolution показан на рис. 25.4.

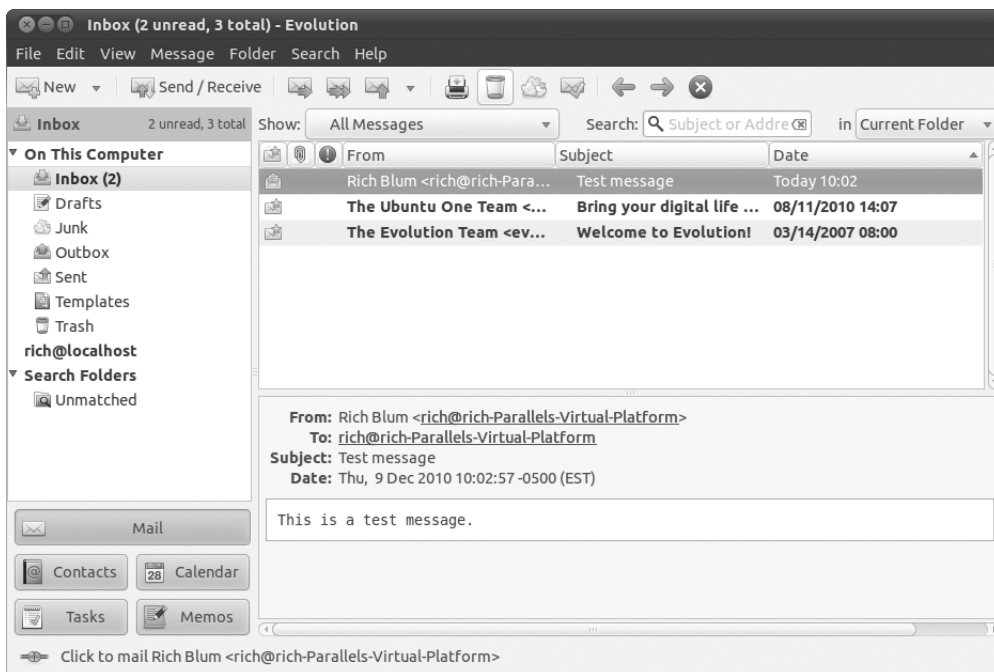


Рис. 25.4. Главный экран программы MUA Evolution

Для подключения к удаленным серверам обе программы, KMail и Evolution, поддерживают протокол POP (Post Office Protocol), а также более развитый по сравнению с ним протокол IMAP (Internet Message Access Protocol). Безусловно, программы MUA KMail и Evolution превосходно подходят для рабочего стола Linux, но они не столь полезны с точки зрения сценарной поддержки командного интерпретатора.

Установка сервера

Прежде чем появится возможность автоматически рассылать свои сообщения электронной почты получателям во всем мире, необходимо обеспечить работу в системе Linux пакета MTA и его правильную настройку. Эта задача сама по себе не столь элементарна, но, к счастью, в некоторых дистрибутивах Linux предусмотрен ряд простых инструментов, которыми можно воспользоваться.

В этом разделе приведены основные сведения о двух наиболее широко применяемых программах MTA электронной почты, которые предусмотрены в Linux: sendmail и Postfix. Безусловно, по вопросам правильной настройки каждого из этих пакетов написаны отдельные книги, поэтому мы рассмотрим лишь основы, чтобы узнать, как обеспечить отправку сообщений электронной почты из системы Linux и их последующее поступление в папку “Входящие”.

sendmail

Пакет MTA sendmail является одним из наиболее широко применяемых пакетов MTA с открытым исходным кодом в числе тех, что используются на почтовых серверах в Интернете. В прошлом на репутации этого пакета отрицательно сказывались многочисленные истории об обнаружении лазеек и пробелов в системе его безопасности; однако в дальнейшем этот пакет был полностью переписан не только для устранения недостатков, касающихся безопасности, но и для включения многих новых средств MTA, таких как управление спамом. Все более новые версии программы sendmail показали себя не только как весьма надежные, но и как достаточно универсальные.

Составные части программы sendmail

Основная исполняемая программа, sendmail, обычно эксплуатируется в фоновом режиме, прослушивая запросы на создание соединений SMTP от удаленных почтовых серверов и перенаправляя исходящие сообщения от локальных пользователей.

Кроме основной программы sendmail, предусмотрены файл конфигурации и несколько таблиц, которые служат для хранения информации, используемой при обработке входящих и исходящих почтовых сообщений. В табл. 25.1 перечислены все компоненты, применяемые в обычной установке sendmail.

Таблица 25.1. Файлы конфигурации sendmail

Файл	Описание
sendmail.cf	Текстовый файл, который управляет поведением программы sendmail
sendmail.cw	Текстовый файл, содержащий список доменных имен, для которых программа sendmail получает сообщения
sendmail.ct	Текстовый файл, содержащий список заслуживающих доверия пользователей, которые могут управлять работой программы sendmail
aliases	Двоичный файл, содержащий список допустимых локальных адресов электронной почты, с которых почта может быть перенаправлена другому пользователю, записана в файл или передана в программу
newaliases	Исполняемая программа, которая создает новый файл базы данных псевдонимов из текстового файла
mailq	Исполняемая программа, которая проверяет очередь почтовых сообщений и выводит сообщения на внешние устройства
mqueue	Каталог, используемый для хранения сообщений, ожидающих доставки
mailertable	Текстовый файл, используемый для указания путей маршрутов, которые относятся к конкретным доменам
domaintable	Текстовый файл, используемый для определения соответствия между старыми доменными именами и новыми
virtusertable	Текстовый файл, используемый для определения соответствия адресов пользователей и доменов с альтернативными адресами
relay-domains	Текстовый файл, используемый для перечисления конкретных узлов, которые имеют разрешение на ретрансляцию сообщений через программу sendmail
access	Текстовый файл, используемый для перечисления конкретных доменов, для которых разрешено или запрещено получение сообщений

Если основной почтовый сервер не эксплуатируется в интересах корпорации или поставщика услуг Интернета, то для обеспечения его нормальной работы достаточно лишь настроить файл конфигурации `sendmail.cf`. В действительности во многих дистрибутивах Linux, предусматривающих применение `sendmail`, основной файл конфигурации `sendmail.cf`, который способен функционировать вполне приемлемо в большинстве простых приложений, создается и настраивается автоматически.

Файл `sendmail.cf`

Программа `sendmail` должна иметь информацию о том, как следует обрабатывать сообщения по мере их получения сервером. Как и программы МТА, программа `sendmail` обрабатывает входящую почту и перенаправляет ее другому пакету электронной почты в удаленной или локальной системе. Для передачи программе `sendmail` указаний, касающихся того, как манипулировать почтовыми адресами назначения, чтобы определить, куда и как должны перенаправляться сообщения, используется файл конфигурации. Заданным по умолчанию местоположением для файла конфигурации является `/etc/mail/sendmail.cf`.

Файл `sendmail.cf` состоит из наборов правил, с помощью которых проводится синтаксический анализ входящего почтового сообщения для определения того, какие действия должны быть выполнены. Каждый набор правил служит для идентификации определенных форматов электронной почты и передачи программе `sendmail` указаний по обработке соответствующих сообщений.

После получения сообщения программа `sendmail` анализирует его заголовок и пропускает это сообщение через различные наборы правил для определения действий, которые должны быть выполнены с сообщением. Файл конфигурации `sendmail` включает правила, которые позволяют программе `sendmail` обрабатывать почту во многих форматах. Почта, полученная от узла SMTP, имеет ряд полей заголовков, отличающихся от тех, что применяются в почте, полученной от локального пользователя. Программа `sendmail` должна знать, как действовать в любых ситуациях, возникающих в процессе обработки почты.

С правилами также связаны вспомогательные функции, которые определены в файле конфигурации. Предусмотрены три типа вспомогательных функций, которые могут быть определены, как описано ниже.

- **Классы.** Определяют общие фразы, которые используются для упрощения идентификации сообщений определенных типов в наборах правил.
- **Макросы.** Значения, предназначенные для упрощения ввода длинных строк в файле конфигурации.
- **Опции.** Ключевые слова, которые применяются для задания параметров операций программы `sendmail`.

Файл конфигурации состоит из ряда классов, макросов, опций и наборов правил. Каждая функция определена как отдельная текстовая строка в файле конфигурации.

Каждая строка в файле конфигурации начинается с одного символа, который определяет действие для этой строки. Строки, которые начинаются с пробела или знака табуляции, рассматриваются как продолжение предыдущей строки с описанием действия. Строки, которые начинаются со знака диеза (`#`), обозначают комментарии и не обрабатываются программой `sendmail`.

Действие, заданное в начале текстовой строки, определяет, для чего используется строка. В табл. 25.2 приведены стандартные действия `sendmail` и их описание.

Таблица 25.2. Строки файла конфигурации sendmail

Строка конфигурации	Описание
C	Определяет классы текста
D	Определяет макрос
F	Определяет файлы, содержащие классы текста
H	Определяет поля заголовка и действия
K	Определяет базы данных, которые содержат искомый текст
M	Определяет обработчики почты
O	Задаёт опции sendmail
P	Определяет значения приоритета sendmail
R	Определяет наборы правил, предназначенные для синтаксического анализа адресов
S	Определяет группы наборов правил

Как уже было сказано, чаще всего после установки программы sendmail не приходится создавать файл конфигурации `sendmail.cf` с нуля; дистрибутив Linux должен создать стандартный шаблонный файл автоматически. Вероятнее всего, единственной задачей, с которой придется столкнуться, будет определение того, должен ли использоваться для перенаправления почты промежуточный узел. Этим средством управляет строка конфигурации DS:

`DSmyisp.com`

Достаточно лишь добавить имя хоста промежуточного узла непосредственно после дескриптора DS.

Postfix

Пакет программ Postfix быстро переходит в разряд наиболее широко применяемых пакетов электронной почты из всех, доступных для систем Unix и Linux. Программа Postfix была разработана Вьетсе Венема (Wietse Venema) в целях предоставления альтернативной программы MTA для стандартных серверов типа Unix. Программное обеспечение Postfix способно превратить любую систему Unix или Linux в полнофункциональный сервер электронной почты.

На этот пакет MTA возложена ответственность по управлению сообщениями, которые поступают на почтовый сервер или отправляются с него. В пакете Postfix такое отслеживание сообщений осуществляется с помощью нескольких модульных программ и системы каталогов очередей электронной почты. Каждая программа обрабатывает сообщения с применением разных очередей сообщений, сопровождая эти сообщения до тех пор, пока они не будут доставлены по месту назначения. Если в любое время в ходе передачи сообщения возникает аварийный отказ почтового сервера, Postfix может определить, в какую очередь было в последний раз успешно помещено сообщение, и попытаться продолжить обработку сообщения.

Части системы Postfix

Система Postfix состоит из нескольких каталогов очередей электронной почты и исполняемых программ и обеспечивает взаимодействие всех этих компонентов в целях создания службы электронной почты.

В пакете Postfix используется ведущая программа, которая всегда эксплуатируется как фоновый процесс. Эта ведущая программа позволяет системе Postfix порождать процессы, при-

меняемые для просмотра очередей электронной почты в целях определения наличия новых сообщений и отправки по надлежащим назначениям.

Ведущая программа использует различные вспомогательные программы, запускаемые по мере необходимости, с учетом их предназначения. Вспомогательные программы можно настроить таким образом, чтобы они продолжали функционировать в течение указанного промежутка времени по завершении их использования. Это позволяет ведущей программе в случае необходимости повторно прибегнуть к услугам функционирующей вспомогательной программы и сократить затраты времени на запуск последней. По истечении заданного времени неиспользуемая вспомогательная программа автоматически останавливается.

Задача всестороннего управления работой системы Postfix возложена на ведущую программу. В табл. 25.3 перечислены вспомогательные программы, используемые в системе Postfix для передачи сообщений электронной почты.

Таблица 25.3. Вспомогательные программы Postfix

<i>Программа</i>	<i>Описание</i>
bounce	Для каждого сбойного сообщения электронной почты выводит запись в очередь сбойных сообщений и возвращает сбойное сообщение отправителю
cleanup	Обрабатывает заголовки входящих почтовых сообщений и помещает сообщения в очередь входящих сообщений
error	Обрабатывает запросы доставки сообщений от программы qmgr, вынуждая возвращать сообщения
flush	Обрабатывает сообщения, получения которых ожидает удаленный почтовый сервер
local	Доставляет сообщения, предназначенные для локальных пользователей
pickup	Обеспечивает ожидание сообщений в очереди maildrop и отправляет их в программу cleanup для обработки
pipe	Перенаправляет сообщения из программы диспетчера очереди во внешние программы
postdrop	Перемещает входящее сообщение в очередь maildrop, если эта очередь не предназначена для записи обычными пользователями
qmgr	Обрабатывает сообщения во входящей очереди, определяя, куда и как они должны быть доставлены, а затем вызывает на выполнение программы для их доставки
sendmail	Предоставляет совместимый с sendmail интерфейс для программ в целях отправки сообщений в очередь maildrop
showq	Выводит сведения о состоянии очереди почтовых сообщений Postfix
smtp	Перенаправляет сообщения на внешние узлы электронной почты с использованием протокола SMTP
smtpd	Получает сообщения от внешних узлов электронной почты с использованием протокола SMTP
trivial-rewrite	Получает сообщения от программы cleanup, обеспечивая при этом, чтобы адреса в заголовке были представлены в стандартном формате для программы qmgr, и используется программой qmgr для разрешения (поиска) адресов удаленных узлов

В программе Postfix используется несколько разных очередей сообщений для управления сообщениями электронной почты в ходе их обработки. Каждая очередь сообщений содержит сообщения в соответствующем состоянии обработки сообщений системы Postfix. В табл. 25.4 перечислены очереди сообщений, которые используются в программе Postfix.

Таблица 25.4. Очереди сообщений Postfix

Очередь	Описание
maildrop	Новые сообщения, полученные от локальных пользователей и ожидающие обработки
incoming	Новые сообщения, ожидающие обработки или полученные от удаленных узлов, а также обработанные сообщения от локальных пользователей
active	Сообщения, готовые для доставки программой Postfix
deferred	Сообщения, для которых первоначальная попытка доставки окончилась неудачей и ожидающих еще одной попытки
flush	Сообщения, предназначенные для удаленных узлов, которые должны подключиться к почтовому серверу для получения этих сообщений
mail	Доставленные сообщения, сохраненные для того, чтобы быть прочитанными локальными пользователями

Если в какое-то время происходит останов системы Postfix, то сообщения остаются в той очереди, в которую они были помещены в последний раз. После перезагрузки программы Postfix обработка сообщений из очередей автоматически возобновляется. Это — великолепная особенность системы Postfix, благодаря которой она является одной из наиболее надежных систем поддержки серверов электронной почты из всех имеющихся!

Файлы конфигурации Postfix

Крайне важной частью системы Postfix являются файлы конфигурации. В системе Postfix предусмотрены три отдельных файла конфигурации, которые позволяют задавать параметры, предназначенные для использования в качестве указаний по обработке сообщений для системы Postfix. В отличие от некоторых других программ МТА, допускается возможность вносить изменения в информацию о конфигурации в ходе работы сервера Postfix и выдавать команду Postfix на загрузку новой информации, без необходимости полностью останавливать работу сервера электронной почты.

Эти три файла конфигурации обычно хранятся в общем каталоге Postfix. Чаще всего заданным по умолчанию местоположением для этого каталога является `/etc/postfix`. Обычно все пользователи получают доступ для просмотра файлов конфигурации, однако только пользователь `root` имеет право вносить изменения в значения параметров в этих файлах. Разумеется, по соображениям безопасности могут быть приняты еще более строгие правила доступа. В табл. 25.5 перечислены файлы конфигурации Postfix.

Таблица 25.5. Файлы конфигурации Postfix

Файл	Описание
<code>install.cf</code>	Содержит информацию из параметров установки, которые использовались при первоначальной установке Postfix
<code>main.cf</code>	Содержит параметры, используемые программами системы Postfix при обработке сообщений
<code>master.cf</code>	Содержит параметры, используемые ведущей программой Postfix при вызове на выполнение основных программ

Файл конфигурации `install.cf` позволяет получить сведения о том, какие параметры установки были заданы в то время, когда осуществлялась первоначальная установка в операционной системе программного обеспечения Postfix. Этот файл предоставляет простой способ

определения того, какие средства доступны или не доступны в данном конкретном варианте установки программного обеспечения.

Файл конфигурации `master.cf` управляет поведением основных программ Postfix. Каждая программа перечислена в отдельной строке наряду с параметрами, управляющими ее работой. Ниже приведен типичный файл `master.cf` с настройками по умолчанию.

```
# =====
#service type private unpriv chroot wakeup maxproc command + args
#          (yes)    (yes)    (yes)    (never) (50)
# =====
smtp      inet      n       -       n       -       -       smtpd
pickup    fifo      n       -       n       60      1       pickup
cleanup   unix      -       -       n       -       0       cleanup
qmgr      fifo      n       -       n       300     1       qmgr
rewrite   unix      -       -       n       -       -       trivial-rewrite
bounce    unix      -       -       n       -       0       bounce
defer      unix      -       -       n       -       0       bounce
trac      unix      -       -       n       -       0       bounce
verify    unix      -       -       n       -       1       verify
flush     unix      n       -       n       1000    0       flush
proxymap  unix      -       -       n       -       -       proxymap
smtp      unix      -       -       n       -       -       smtp
relay     unix      -       -       n       -       -       smtp -o fallback_relay=
showq     unix      n       -       n       -       -       showq
error     unix      -       -       n       -       -       error
local     unix      -       n       n       -       -       local
virtual   unix      -       n       n       -       -       virtual
lmtp      unix      -       -       n       -       -       lmtp
anvil     unix      -       -       n       -       1       anvil
scache    unix      -       -       n       -       1       scache
```

Файл конфигурации `master.cf` включает также строки, которые служат для системы Postfix указанием о том, как должно осуществляться взаимодействие с внешним программным обеспечением MDA, таким как Procmail.

Параметры, управляющие работой системы Postfix, заданы в файле конфигурации `main.cf`. Все параметры, от которых зависит функционирование Postfix, имеют значения по умолчанию, которые выбраны специально для оптимизации системы Postfix. Если значение какого-либо параметра не определено в файле `main.cf`, в качестве его значения выбирается тот вариант, который предварительно задан для Postfix. Если же значение параметра присутствует в файле `main.cf`, то оно перекрывает значение по умолчанию.

Все параметры Postfix перечисляются в отдельной строке в файле конфигурации наряду с их значениями в следующей форме:

```
parameter = value
```

И *parameter*, и *value* представляют собой обычные строки текста, которые можно легко прочитать и изменить в случае необходимости. Ведущая программа Postfix считывает значения параметров из файла `main.cf` сразу после запуска системы Postfix, а также каждый раз после получения команды на перезагрузку Postfix.

В качестве двух примеров параметров Postfix можно привести параметры `myhostname` и `mydomain`. Если эти параметры не заданы в файле конфигурации `main.cf`, то для параметра `myhostname` устанавливается значение результата выполнения команды `gethostname()`

в системе Linux, а для параметра `mydomain` задается доменная часть определяемого по умолчанию параметра `myhostname`. Во многих случаях для обработки электронной почты всего домена применяется единственный сервер электронной почты. Такую настройку системы можно определить в файле конфигурации Postfix достаточно просто:

```
myhostname = mailserver.smallorg.org
mydomain = smallorg.org
```

После запуска система Postfix распознает локальный почтовый сервер как `mailserver.smallorg.org`, а локальный домен — как `smallorg.org` и не рассматривает другие значения, установленные в операционной системе.

Если необходимо указать промежуточный узел, то для этого можно применить параметр `relayhost`:

```
relayhost = myisp.com
```

Здесь можно также указать IP-адрес, но это значение должно быть задано в квадратных скобках.

Отправка сообщений с помощью программы Mailx

Основным инструментом, предназначенным для отправки сообщений электронной почты из сценариев командного интерпретатора, является программа Mailx. Она не только позволяет работать в интерактивном режиме для чтения и отправки сообщений, но и дает возможность использовать параметры командной строки для указания конкретного способа отправки сообщения.

Форматом командной строки программы Mailx для отправки сообщений является следующий:

```
mail [-eIinv] [-a header] [-b addr] [-c addr] [-s subj] to-addr
```

В команде `mail` используются параметры командной строки, которые приведены в табл. 25.6.

Таблица 25.6. Параметры командной строки Mailx

Параметр	Описание
-a	Задать дополнительные строки заголовка SMTP
-b	Добавить поле BCC: с указанием получателя копии сообщения
-c	Добавить поле CC: с указанием получателя копии сообщения
-e	Не отправлять сообщение, если оно пусто
-i	Игнорировать сигналы прерывания от терминала
-I	Принудительно перевести программу Mailx в интерактивный режим
-n	Не читать файл запуска <code>/etc/mail.rc</code>
-s	Указать строку темы
-v	Показывать подробные сведения о доставке на терминале

На основании изучения табл. 25.6 можно прийти к выводу, что почти все необходимое для полного создания сообщения электронной почты можно указать исключительно с помощью параметров командной строки. Единственное, что остается добавить, — это текст самого сообщения.

Для этого необходимо перенаправить текст в команду mail. Ниже приведен простой пример того, как создать и отправить сообщение электронной почты непосредственно из командной строки.

```
$ echo "This is a test message" | mail -s "Test message" rich
```

Программа Mailx отправляет текст из команды echo в качестве текста сообщения. Тем самым обеспечивается простой способ отправки сообщений из сценариев командного интерпретатора. Ниже приведен краткий пример.

```
$ cat factmail
#!/bin/bash
# отправка ответа с результатом вычисления факториала

MAIL='which mail'

factorial=1
counter=1

read -p "Enter the number: " value
while [ $counter -le $value ]
do
    factorial=$((factorial * $counter))
    counter=$((counter + 1))
done

echo "The factorial of $value is $factorial" | mail -s "Factorial
answer" $USER
echo "The result has been mailed to you."
```

В этом сценарии не предполагается, что исполняемый файл программы Mailx имеет стандартное местоположение. В нем используется команда which для определения того, где находится программа mail.

После вычисления результата функции факториала в сценарии командного интерпретатора используется команда mail для отправки сообщения с помощью определяемой пользователем переменной среды \$USER, указывающей на лицо, которым вызван сценарий.

```
$ ./factmail
Enter the number: 5
The result has been mailed to you.
$
```

Пользователю остается лишь проверить свою почту, чтобы узнать, поступил ли ответ:

```
$ mail
"/var/mail/rich": 1 message 1 new
>N 1 Rich Blum Thu Dec 9 10:32 13/586 Factorial answer
?
Return-Path: <rich@rich-Parallels-Virtual-Platform>
X-Original-To: rich@rich-Parallels-Virtual-Platform
Delivered-To: rich@rich-Parallels-Virtual-Platform
Received: by rich-Parallels-Virtual-Platform (Postfix, from
userid 1000)
id B4A2A260081; Thu, 9 Dec 2010 10:32:24 -0500 (EST)
Subject: Factorial answer
To: <rich@rich-Parallels-Virtual-Platform>
```

```

X-Mailer: mail (GNU Mailutils 2.1)
Message-Id: <20101209153224.B4A2A260081@rich-Parallels-Virtual-
↳Platform>
Date: Thu, 9 Dec 2010 10:32:24 -0500 (EST)
From: rich@rich-Parallels-Virtual-Platform (Rich Blum)

The factorial of 5 is 120
?
```

Не всегда бывает удобно отправлять текст сообщения в виде одной строки. Чаще всего возникает необходимость передать в качестве сообщения электронной почты значительной объем вывода. В таких ситуациях можно всегда перенаправить текст во временный файл, а затем воспользоваться командой `cat` для перенаправления вывода в программу `mail`.

Ниже приведен пример отправки в сообщении электронной почты более значительного объема данных.

```

$ cat diskmail
#!/bin/bash
# передача текущих статистических данных об использовании диска
# в виде сообщения электронной почты

date=`date +%m/%d/%Y`
MAIL=`which mail`
TEMP=`mktemp tmp.XXXXXX`

df -k > $TEMP
cat $TEMP | $MAIL -s "Disk stats for $date" $1
rm -f $TEMP
```

Программа `diskmail` получает текущую дату (наряду с некоторыми специальными символами форматирования) с помощью команды `date`, определяет местоположение программы `Mailx`, а затем создает временный файл. В конечном итоге в этой программе используется команда `df` для получения текущих статистических данных о распределении дискового пространства (см. главу 4) и полученный вывод перенаправляется во временный файл.

Затем временный файл перенаправляется в команду `mail`, первый параметр командной строки используется для определения адреса получателя, а в заголовок `Subject:` (Тема:) вводится текущая дата. При выполнении этого сценария не появляется какой-либо вывод в командной строке:

```
$ ./diskmail rich
```

Но после проверки почты должно быть обнаружено отправленное сообщение:

```

$ mail
"/var/mail/rich": 1 message 1 new
>N 1 Rich Blum Thu Dec 9 10:35 19/1020 Disk stats for 12/09/2010
?
Return-Path: <rich@rich-Parallels-Virtual-Platform>
X-Original-To: rich@rich-Parallels-Virtual-Platform
Delivered-To: rich@rich-Parallels-Virtual-Platform
Received: by rich-Parallels-Virtual-Platform (Postfix, from
↳ userid 1000)
    id 3671B260081; Thu, 9 Dec 2010 10:35:39 -0500 (EST)
Subject: Disk stats for 12/09/2010
To: <rich@rich-Parallels-Virtual-Platform>
```

```
X-Mailer: mail (GNU Mailutils 2.1)
Message-Id: <20101209153539.3671B260081@rich-Parallels-Virtual-
Platform>
Date: Thu, 9 Dec 2010 10:35:39 -0500 (EST)
From: rich@rich-Parallels-Virtual-Platform (Rich Blum)
```

Filesystem	1K-blocks	Used	Available	Use%	Mounted on
/dev/sdal	63315876	2595552	57504044	5%	/
none	507052	228	506824	1%	/dev
none	512648	192	512456	1%	/dev/shm
none	512648	100	512548	1%	/var/run
none	512648	0	512648	0%	/var/lock
none	4294967296	0	4294967296	0%	/media/psf
?					

Теперь вам остается лишь запланировать вызов этого сценария на выполнение в каждые сутки с помощью средства `cron`, и отчеты о распределении пространства на жестком диске будут автоматически поступать в ваш ящик для входящей почты! Подобные сценарии позволяют существенно упростить работу системного администратора!

Программа Mutt

Программа Mutt представляет собой еще один широко применяемый пакет почтового клиента для командной строки Linux, разработанный в 1995 году Майклом Элкинсом (Michael Elkins). Эта программа обладает одной особенностью (которая не предусмотрена в программе Mailx), превращающей ее в хороший инструмент, удобный для создания сценариев командного интерпретатора.

Речь идет о том, что в программе Mutt предусмотрена возможность отправлять файлы в виде вложений в сообщениях электронной почты. Таким образом, можно избавиться от необходимости включать большой объем текста из файла в само сообщение электронной почты (как в программе Mailx) и воспользоваться программой Mutt для включения текстового файла в качестве отдельного вложения, которое сопровождает основной текст сообщения. Это средство великолепно подходит для отправки по электронной почте файлов большого объема, таких как файлы журналов.

В настоящем разделе приведены сведения об установке программы Mutt в системе Linux и использовании этой программы для присоединения файлов к сообщениям электронной почты с помощью сценариев командного интерпретатора.

Установка программы Mutt

Программа Mutt не имеет столь широкого распространения в наши дни, когда доступны такие почтовые клиенты с привлекательным графическим оформлением, как KMail или Evolution, поэтому вряд ли можно рассчитывать на то, что она будет устанавливаться по умолчанию в каждом конкретном дистрибутиве Linux. Но в большинстве дистрибутивов Linux эта программа включена в обычный состав файлов дистрибутива для установки с использованием стандартных методов установки программного обеспечения (см. главу 8). Для среды Ubuntu можно установить программу Mutt с помощью следующей команды командной строки:

```
sudo apt-get install mutt
```

Если применяемый вами дистрибутив Linux не включает пакет Mutt или вы желаете установить новейшую версию, то перейдите на веб-сайт Mutt (www.mutt.org), загрузите последние по времени файлы исходного кода, а затем воспользуйтесь методами, описанными в главе 8, чтобы откомпилировать и установить программу Mutt из пакета исходного кода.

Командная строка Mutt

В команде `mutt` предусмотрены параметры, позволяющие управлять работой программы Mutt. В табл. 25.7 приведены доступные параметры командной строки.

Таблица 25.7. Параметры командной строки Mutt

Параметр	Описание
<code>-A alias</code>	Передать развернутую версию указанного псевдонима на устройство STDOUT
<code>-a file</code>	Подключить указанный файл к сообщению с использованием протокола MIME
<code>-b address</code>	Указать получателя слепой копии (BCC)
<code>-c address</code>	Указать получателя точной копии (CC)
<code>-D</code>	Вывести значения всех параметров конфигурации на устройство STDOUT
<code>-e command</code>	Указать команду конфигурации, которая должна быть выполнена после обработки файлов инициализации
<code>-f mailbox</code>	Указать файл почтового ящика, подлежащий загрузке
<code>-F muttrc</code>	Указать файл инициализации, который должен быть считан вместо <code>\$HOME/.muttrc</code>
<code>-h</code>	Вывести текст справки
<code>-H draft</code>	Указать файл черновика, который содержит заголовок и текст, используемые для отправки сообщения
<code>-i include</code>	Указать файл, который должен быть включен в текст сообщения
<code>-m type</code>	Указать заданный по умолчанию тип почтового ящика
<code>-n</code>	Пропускать файл конфигурации системы
<code>-p</code>	Возобновить обработку отложенного сообщения
<code>-Q query</code>	Выполнить запрос на получение значения переменной файла конфигурации. Этот запрос обрабатывается после синтаксического анализа всех файлов конфигурации и выполнения всех команд, заданных в командной строке
<code>-R</code>	Открыть почтовый ящик в режиме только для чтения
<code>-s subject</code>	Указать тему сообщения
<code>-v</code>	Отобразить номер версии Mutt и определения, заданные во время компиляции
<code>-x</code>	Эмулировать режим составления сообщения программы Mailx
<code>-Y</code>	Начать с перечисления всех почтовых ящиков, указанных командой <code>mailboxes</code>
<code>-z</code>	При использовании с параметром <code>-f</code> не производить запуск, если отсутствуют какие-либо сообщения в почтовом ящике
<code>-Z</code>	Открыть первый почтовый ящик, указанный командой <code>mailboxes</code> , который содержит вновь поступившую почту

Очевидно, что такое количество параметров командной строки позволяет полностью задавать способ отправки сообщения электронной почты непосредственно в командной строке, а именно это требуется при работе со сценариями командного интерпретатора.

Во многом аналогично программе Mailx, программа Mutt не позволяет задать в командной строке лишь один компонент письма — основной текст сообщения. Если в программу Mutt не перенаправляется текст, то ее запуск осуществляется в текстовом графическом режиме с окном редактора, в котором должен быть введен текст сообщения.

Это не совсем подходит для сценариев командного интерпретатора, поэтому следует всегда предусматривать перенаправление текста в той или иной форме в качестве текста сообщения, даже если используются возможности программы Mutt, позволяющие указывать файл для присоединения к почтовому сообщению. В следующем разделе будет описано решение этой задачи.

Использование программы Mutt

Теперь рассмотрим, как должна использоваться программа Mutt в сценариях командного интерпретатора. Чтобы задать в своем сценарии командного интерпретатора команду `mutt` в основной форме, необходимо включить опции командной строки, которые указывают тему сообщения, файл вложения и всех получателей сообщения:

```
mutt -s Subject -a file -- recipients
```

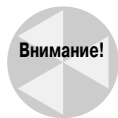
Список *recipients* представляет собой разделенный пробелами список адресов электронной почты, по которым должно быть отправлено сообщение. Если требуется присоединить больше одного файла, то можно задать разделенный пробелами список присоединяемых файлов после опции `-a`; для отделения списка имен файлов от списка адресов получателей служит обозначение в виде двух дефисов (`--`). Параметр *file* должен быть задан как полное составное имя пути или как имя пути, которое определено как относительное применительно к текущему рабочему каталогу, из которого вызывается на выполнение команда `mutt`.

Для работы с командой `mutt` может также применяться еще один способ. Если не осуществляется перенаправление текста для заполнения текста сообщения, то программа Mutt автоматически переходит в полноэкранный режим, чтобы пользователь мог ввести текст в окне редактора. Чаще всего этот вариант развития событий при работе со сценариями является нежелательным, поэтому всегда следует использовать для создания текста сообщения перенаправление из какого-то файла, пусть даже этот файл является пустым:

```
# echo "Here's the log file" | mutt -s "Log file" -a  
/var/log/messages -- rich
```

Эта команда отправляет файл системного журнала в качестве вложения по адресу электронной почты `rich` в локальной системе. Необходимо учитывать, что пользователь должен также иметь надлежащие разрешения для получения доступа к файлу, который требуется подключить. На рис. 25.5 показано полученное сообщение электронной почты в почтовом клиенте Evolution.

Следует отметить, что сообщение включает основной текст из инструкции `echo` наряду с отдельной пиктограммой для присоединенного файла. Присоединенный файл можно сохранить непосредственно из клиентской программы KMail.



Рассматривая имя присоединенного файла, можно отметить, что программа Mutt использует имя этого файла без расширения в качестве имени файла во вложении. При использовании временных файлов необходимо соблюдать осторожность, поскольку Mutt задает имя временного файла как имя файла вложения. Поэтому следует сохранять временные файлы, указывая более описательные имена, а не использовать автоматически сформированные имена временных файлов.

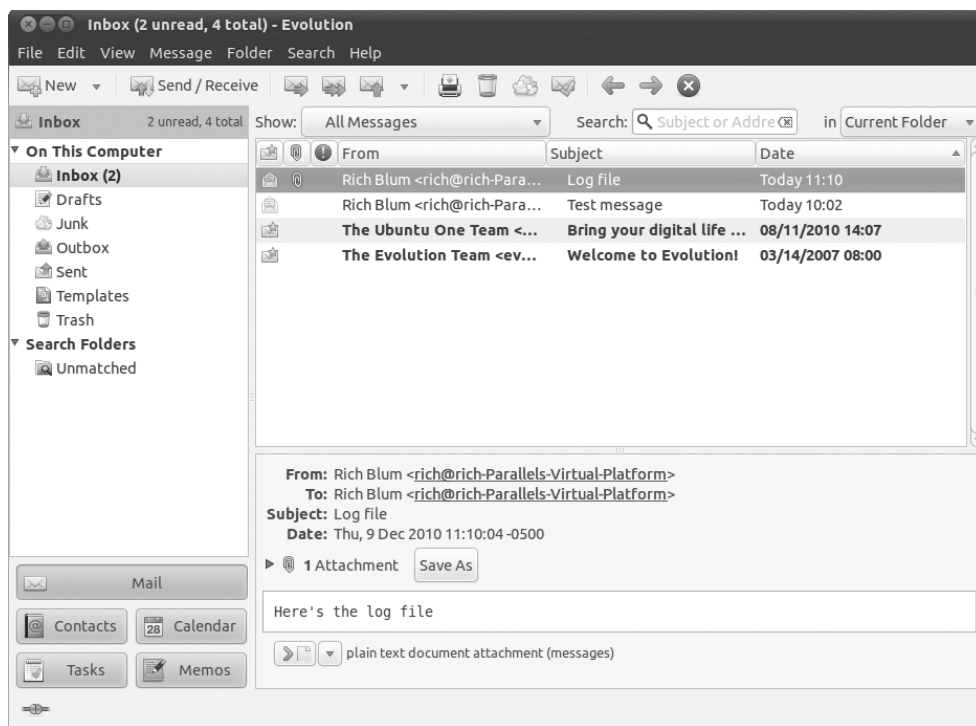


Рис. 25.5. Использование программы Evolution для просмотра сообщения с вложением

Резюме

В настоящей главе рассказывалось о том, как обрабатывать электронную почту в сценариях командного интерпретатора. Например, сценарии командного интерпретатора предоставляют великолепную возможность регулярно отправлять по электронной почте отчеты клиентам.

Прежде чем использовать командную строку для отправки сообщения электронной почты, необходимо ознакомиться с работой электронной почты в среде Linux и с приложениями, которые должны быть установлены и настроены. Среда электронной почты Linux состоит из трех элементов: программы MTA (Mail Transfer Agent — агент пересылки сообщений); программы MDA (Mail Delivery Agent — агент доставки сообщений), которая часто входит в состав программы MTA; а также программы MUA (Mail User Agent — почтовый агент пользователя).

Программа MTA осуществляет функции передачи и получения почтовых сообщений для системы Linux. Эта программа должна иметь данные о том, как следует передавать входящие почтовые сообщения в соответствующие почтовые ящики пользователей и как отправлять исходящие почтовые сообщения, предназначенные для пользователей на удаленных почтовых серверах. В программе MTA для выполнения конкретных операций по доставке почты часто используется прокси-сервер (называемый также *промежуточным узлом*). Данная программа перенаправляет все сообщения, предназначенные для пользователей на удаленном почтовом сервере, промежуточному узлу для доставки. Кроме того, работа программы MTA может быть организована так, чтобы промежуточный узел сам выполнял доставку.

Программа MDA отвечает за то, чтобы почта, предназначенная для локальных пользователей, в конечном итоге отправлялась в соответствующий локальный почтовый ящик. Иногда эта функция выполняется непосредственно программой MTA. Однако, если требуются более усовершенствованные функции доставки почты, такие как получение извещений за пределами офиса или фильтрация спама, можно настроить программу MTA на передачу сообщений в программу MDA, в которую встроены такие возможности.

Программа MUA позволяет отдельным пользователям системы получать доступ к сообщениям в своих почтовых ящиках и передавать исходящие сообщения, адресованные другим пользователям, в программу MTA для доставки. Разработан широкий перечень программ электронной почты, начиная от простых программ командной строки, таких как Mailx и Mutt, и заканчивая графическими программами наподобие KMail и Evolution.

Простейший способ отправки электронной почты из сценария командного интерпретатора состоит в применении программы Mailx, которая позволяет задавать в командной строке заголовки письма с темой и адресами одного или нескольких получателей. Текст сообщения создается путем перенаправления текстовых данных в программу Mailx. Такое перенаправление в почтовое сообщение можно выполнить, используя команду `echo` для передачи отдельных текстовых строк или команду `cat` для ввода содержимого целого файла.

Программа Mutt представляет собой более усовершенствованную программу MUA командной строки, которая позволяет присоединять к почтовому сообщению файлы, не включая содержимое этих файлов непосредственно в текст сообщения. Это позволяет присоединять к сообщениям электронной почты большие текстовые файлы, которые клиенты могут сохранять на диске для просмотра в других программах, допустим, в пакетах обработки текста или электронных таблиц.

В последних двух главах этой книги рассматривается важная область применения сценариев командного интерпретатора — сценарная поддержка функций администрирования. Системным администраторам Linux часто приходится сталкиваться с такой ситуацией, когда необходимо обеспечить регулярный контроль состояния того или иного системного средства. Путем создания сценария командного интерпретатора и включения его в состав заданий `cron` можно легко решать задачи отслеживания всех операций, выполняемых в системе Linux.

ГЛАВА

26

В этой главе...

Контроль над использованием места на диске

Выполнение резервного копирования

Управление учетными записями пользователей

Резюме

Написание программ на основе сценариев

Трудно найти область, в которой разработка сценариев командного интерпретатора приносила бы большую пользу по сравнению с написанием программ на основе сценариев для системного администратора Linux. Системному администратору Linux, как правило, приходится ежедневно выполнять бесчисленное множество заданий, начиная от текущего контроля свободного пространства на диске и резервного копирования важных файлов и заканчивая управлением учетными записями пользователей. Программы на основе сценариев командного интерпретатора позволяют сделать жизнь системного администратора много проще! В настоящей главе демонстрируются некоторые возможности, связанные с написанием программ на основе сценариев в командном интерпретаторе `bash`.

Контроль над использованием места на диске

Одной из самых важных проблем при обслуживании многопользовательских систем Linux является контроль доступного дискового пространства. В некоторых ситуациях, например при работе с файловым сервером совместного доступа, свободное место на диске может быть почти полностью исчерпано в результате действий всего лишь одного неосторожного пользователя.

Рассматриваемая здесь программа на основе сценария командного интерпретатора позволяет определить в указанных каталогах десять пользователей, потребляющих наибольший объем дискового пространства. В результате формируется отчет с отметками дат, который позволяет контролировать тенденцию изменения потребления места на диске.

Обязательные функции

Первым инструментом, который необходимо использовать, является команда `du` (см. главу 4). Эта команда выводит сведения об использовании дискового пространства для отдельных файлов и каталогов. Опция `-s` позволяет получать итоговые сведения на уровне каталогов. Эти данные необходимы для вычисления общего объема дискового пространства, используемого отдельными пользователями. Ниже показаны результаты, формируемые при использовании команды `du` для получения итоговых сведений об объеме каталога `$HOME` каждого пользователя в общем каталоге `/home`.

```
$ du -s /home/*
6174428      /home/consultant
4740         /home/Development
4740         /home/Production
3860         /home/Samantha
7916         /home/Timothy
140376116    /home/user
$
```

Опция `-s` оказалась вполне применимой для каталогов `$HOME` пользователей, но иногда возникает необходимость рассмотреть потребление дискового пространства в системном каталоге, таком, как `/var/log`:

```
$ du -s /var/log/*
4          /var/log/alternatives.log
44         /var/log/alternatives.log.1
4          /var/log/apparmor
176        /var/log/apt
0          /var/log/aptitude
4          /var/log/aptitude.1.gz
4          /var/log/aptitude.2.gz
4          /var/log/aptitude.3.gz
4          /var/log/aptitude.4.gz
4          /var/log/aptitude.5.gz
160        /var/log/auth.log
...
$
```

Очевидно, что формируемый при этом листинг получается слишком подробным. В данном случае для наших целей более удобной является опция `-S`, которая формирует итоги отдельно по каждому каталогу и подкаталогу. Это позволяет быстро и точно выявить проблемные области:

```
$ du -S /var/log
176      /var/log/apt
52       /var/log/exim4
1048     /var/log/dist-upgrade/20101011-1337
6148     /var/log/dist-upgrade
1248     /var/log/installer
```

```

228      /var/log/gdm
4        /var/log/news
4        /var/log/samba/cores/winbindd
4        /var/log/samba/cores
16       /var/log/samba
4        /var/log/unattended-upgrades
4        /var/log/sysstat
4        /var/log/speech-dispatcher
108      /var/log/ConsoleKit
64       /var/log/cups
4        /var/log/apparmor
12       /var/log/fsck
4844     /var/log
$

```

Основной интерес для нас представляет то, какие каталоги занимают наибольшую часть места на диске, поэтому необходимо воспользоваться командой `sort` (см. главу 4) для обработки листинга, сформированного командой `du`:

```

$ du -S /var/log | sort -rn
6148    /var/log/dist-upgrade
4864    /var/log
1248    /var/log/installer
1048    /var/log/dist-upgrade/20101011-1337
228     /var/log/gdm
176     /var/log/apt
108     /var/log/ConsoleKit
64      /var/log/cups
52      /var/log/exim4
16      /var/log/samba
12      /var/log/fsck
4       /var/log/unattended-upgrades
4       /var/log/sysstat
4       /var/log/speech-dispatcher
4       /var/log/samba/cores/winbindd
4       /var/log/samba/cores
4       /var/log/news
4       /var/log/apparmor
$

```

Опция `-n` указывает, что данные, подлежащие сортировке, должны рассматриваться как цифровые. Опция `-r` служит для задания порядка сортировки от большего к меньшему. Такой вариант идеально подходит для выявления самых крупных потребителей дискового пространства.

С помощью редактора `sed` (см. главы 18 и 20) можно улучшить этот листинг еще больше. Чтобы можно было сосредоточиться на десяти самых крупных потребителях дискового пространства, программа `sed` удаляет остальную часть листинга после достижения строки 11. Следующий шаг состоит в добавлении номера для каждой строки в листинге. Как было показано в главе 18, это можно сделать путем добавления знака равенства (=) к команде `sed`. Для того чтобы требуемые номера оказались в той же строке, в которой должен находиться текст с данными о потреблении дискового пространства, можно объединить номера с текстом с помощью команды `N`, как было показано в главе 20. Применяемые команды `sed` выглядят следующим образом:

```
sed '{11,$D; =}' |
sed 'N; s/\n/ /' |
```

После этого можно приступить к дальнейшему усовершенствованию вывода с помощью команды `gawk` (см. главу 21). Вывод редактора `sed` передается по каналу в команду `gawk`, а затем выводится на внешнее устройство с помощью функции `printf`.

```
gawk '{printf $1 ":" "\t" $2 "\t" $3 "\n"}'
```

После каждого номера строки добавлено двоеточие (:), а между отдельными полями в каждой строке текста вывода помещены символы табуляции (`\t`). В результате формируется удобно отформатированный листинг с данными о десяти наиболее крупных потребителях дискового пространства.

```
$ du -S /var/log |
> sort -rn |
> sed '{11,$D; =}' |
> sed 'N; s/\n/ /' |
> gawk '{printf $1 ":" "\t" $2 "\t" $3 "\n"}'
1:      6148      /var/log/dist-upgrade
2:      4864      /var/log
3:      1248      /var/log/installer
4:      1048      /var/log/dist-upgrade/20101011-1337
5:       228      /var/log/gdm
6:       176      /var/log/apt
7:       108      /var/log/ConsoleKit
8:        64      /var/log/cups
9:        52      /var/log/exim4
10:      16       /var/log/samba
$
```

Теперь у вас в руках все необходимые данные! Следующий шаг состоит в использовании этой информации для создания сценария.

Создание сценария

В целях экономии времени и усилий отчет создается в сценарии для нескольких назначенных каталогов. Для решения этой задачи используется переменная `CHECK_DIRECTORIES`. Исходя из поставленной нами задачи, в этой переменной содержатся данные только о двух каталогах:

```
CHECK_DIRECTORIES="/var/log /home"
```

Сценарий содержит цикл `for`, предназначенный для выполнения команды `du` применительно к каждому каталогу, перечисленному в переменной. Мы уже использовали аналогичный способ (см. главу 12) для чтения и обработки значений в списке. При каждой итерации цикла `for`, проходящего по списку значений в переменной `CHECK_DIRECTORIES`, переменной `DIR_CHECK` присваивается очередное значение в списке:

```
for DIR_CHECK in $CHECK_DIRECTORIES
do
...
    du -S $DIR_CHECK
...
done
```

Чтобы можно было проще определить, к какому времени относится отчет, к имени файла отчета добавляется отметка даты с использованием команды `date`. С помощью команды `exec` (см. главу 14) в сценарии происходит перенаправление вывода в файл отчета с отметкой даты:

```
DATE=$(date '+%m%d%y')
exec > disk_space_${DATE}.rpt
```

Затем для формирования удобно отформатированного отчета в сценарии используется команда `echo` для вставки нескольких строк, относящихся к названию отчета:

```
echo "Top Ten Disk Space Usage"
echo "for $CHECK_DIRECTORIES Directories"
```

Теперь мы можем перейти к изучению того, как должен выглядеть рассматриваемый сценарий после определения всех его частей:

```
#!/bin/bash
#
# Big_Users - поиск самых крупных потребителей дискового пространства
# в различных каталогах
#####
# Параметры сценария
#
CHECK_DIRECTORIES=" /var/log /home"      #каталоги, подлежащие проверке
#
##### Основной сценарий #####
#
DATE=$(date '+%m%d%y')                  #дата файла отчета
#
exec > disk_space_${DATE}.rpt            #передача файла отчета
#                                       # в стандартный вывод
#
echo "Top Ten Disk Space Usage"          #Заголовок для всего отчета
echo "for $CHECK_DIRECTORIES Directories"
#
for DIR_CHECK in $CHECK_DIRECTORIES      # цикл обработки каталогов
#                                       # с помощью команды du
do
    echo ""
    echo "The $DIR_CHECK Directory:"      # Заголовок для данных
#                                       # о каждом каталоге
#
# Создание листинга с данными о пользователях, потребляющих дисковое
# пространство больше всех
    du -S $DIR_CHECK 2>/dev/null |
    sort -rn |
    sed '{11,$D; =}' |
    sed 'N; s/\n/ /' |
    gawk '{printf $1 ":" "\t" $2 "\t" $3 "\n"}'
#
done                                     # конец цикла for обработки
#                                       # каталогов с помощью команды du
#
```

Итак, намеченная нами цель достигнута. Мы подготовили простой сценарий командного интерпретатора, который создает отчет с отметкой времени с данными о десяти наиболее крупных потребителях дискового пространства для каждого выбранного каталога.

Выполнение сценария

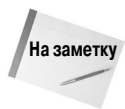
Прежде чем организовать автоматическое выполнение сценария `Big_Users`, необходимо проверить его несколько раз вручную, чтобы убедиться в том, что он действует согласно намеченному замыслу:

```
$
$ ./Big_Users
$
$ cat disk_space_012311.rpt
Top Ten Disk Space Usage
for /var/log /home Directories

The /var/log Directory:
1:          6148      /var/log/dist-upgrade
2:          4892      /var/log
3:          1248      /var/log/installer
4:          1048      /var/log/dist-upgrade/20101011-1337
5:           176      /var/log/apt
6:           108      /var/log/ConsoleKit
7:           64       /var/log/cups
8:           52       /var/log/exim4
9:           16       /var/log/samba
10:          12       /var/log/fsck

The /home Directory:
1:          92365332   /home/user/.VirtualBox/HardDisks
2:          18659720   /home/user/Downloads
3:          17626092   /home/user/archive
4:          6174408    /home/Timothy/Junk/More_Junk
5:          6174408    /home/Timothy/Junk
6:          6174408    /home/consultant/Work
7:          6174408    /home/consultant/Downloads
8:          3227768    /home/user/vmware/Mandriva
9:          3212464    /home/user/vmware/Fedora
10:         104632     /home/user/vmplayer
$
```

Итак, он действительно работает! Теперь, если это потребуется, можно обеспечить автоматическое выполнение данного сценария командного интерпретатора с помощью таблицы `cron` (см. главу 15).



При запуске на выполнение сценария, содержащего команды командного интерпретатора `bash`, для которых необходимо иметь права доступа `root`, следует использовать команду `su` или `sudo`. В противном случае попытка выполнения сценария приведет к непредвиденным результатам.

Предполагаемая частота вызова этого сценария зависит от того, насколько велика активность файлового сервера. Чтобы обеспечить выполнение сценария один раз в неделю, добавьте следующую запись в таблицу cron:

```
15 7 * * 1 /home/user/Big_Users
```

Эта запись указывает, что данный сценарий командного интерпретатора должен вызываться на выполнение каждый понедельник в 7:15. Таким образом, сразу после того, как системный администратор в начале рабочей недели приступает к своим обязанностям, в его распоряжение поступает еженедельный отчет о потреблении дискового пространства.

Выполнение резервного копирования

Независимо от того, эксплуатируется ли система Linux в среде предприятия или дома, потеря данных может оказаться катастрофической. Для предотвращения неблагоприятного развития событий рекомендуется всегда выполнять регулярное резервное копирование.

Однако рекомендации — это одно, а их практическое осуществление — совсем другое. Затруднения могут возникнуть даже при попытке согласовать график резервного копирования для сохранения важных файлов. Но и в этом часто могут помочь сценарии командного интерпретатора.

В настоящем разделе показаны два разных метода использования сценариев командного интерпретатора для резервного копирования данных в системе Linux.

Архивирование файлов данных

Если система Linux используется для работы над важным проектом, можно создать сценарий командного интерпретатора, который автоматически формирует моментальные снимки конкретных каталогов. Перечни таких каталогов можно задавать в файле конфигурации, что позволяет изменять состав резервируемых каталогов после перехода от одного проекта к другому. Выборочное копирование каталогов позволяет сократить затраты времени на осуществление процесса восстановления из основных файлов архива.

В настоящем разделе показано, как создать автоматизированный сценарий командного интерпретатора, который формирует моментальные снимки указанных каталогов и сохраняет архивы предыдущих версий данных.

Обязательные функции

В мире Linux основным инструментом архивирования данных является команда `tar` (см. главу 4). Команда `tar` используется для архивирования целого дерева каталогов в виде одного файла. Ниже приведен пример создания файла архива рабочего каталога с помощью команды `tar`.

```
$ tar -cf archive.tar /home/user/backup_test
tar: Removing leading '/' from member names
$
```

Команда `tar` в ответ выводит предупреждающее сообщение о том, что удаляется ведущая косая черта из имени пути для преобразования из полного составного имени в относительное имя пути (см. главу 3). Благодаря этому появляется возможность извлекать файлы, архивированные с помощью программы `tar`, в любом другом месте файловой системы. Иногда бывает целесообразно исключить появление этого сообщения при выполнении сценария. Для этого достаточно перенаправить вывод в устройство `STDERR` в файл `/dev/null` (см. главу 14):

```
$ tar -cf archive.tar /home/user/backup_test 2>/dev/null
$
```

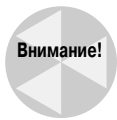
Для размещения файла архива tar может потребоваться много пространства на диске, поэтому рекомендуется проводить сжатие этого файла. Для этого достаточно просто добавить опцию `-z`. Таким образом, файл архива tar будет упакован в файл tar, сжатый программой gzip (такие файлы принято называть tar-файлами). Для указания на то, что данный файл представляет собой tar-файл, следует использовать надлежащие расширения файла, например `.tar.gz` или `.tgz`. Ниже приведен пример создания tar-файла архива рабочего каталога.

```
$ tar -zcf archive.tar.gz /home/user/backup_test 2>/dev/null
$
```

Итак, в нашем распоряжении теперь имеется основной компонент для создания сценария архивирования.

Чтобы исключить необходимость изменять или вновь создавать сценарий архивирования для каждого нового каталога или файла, подлежащего резервированию, можно использовать файл конфигурации. В файле конфигурации должен быть указан каждый каталог или файл, который должен быть включен в архив:

```
$ cat Files_To_Backup
/home/user/Downloads
/home/user/Documents/CLandSS_V2
/home/Samantha/Documents
/home/Does_not_exist
/home/Timothy/Junk
/home/consultant/Work
$
```



Если применяется дистрибутив Linux, который включает графический рабочий стол, необходимо тщательно следить за тем, чтобы каталоги `$HOME` архивировались не частично, а полностью. На первый взгляд может показаться, что в этом нет необходимости, но каталог `$HOME` содержит большое количество файлов конфигурации и временных файлов, необходимых для нормального функционирования графического рабочего стола. К сожалению, в результате создается намного более крупный файл архива по сравнению с тем, на который можно было рассчитывать. Выберите подкаталог для хранения своих рабочих файлов и укажите его в файле конфигурации архива.

Сценарий архивирования должен считывать файл конфигурации и добавлять имена всех каталогов и файлов в список объектов, подлежащих архивированию. Для этого достаточно воспользоваться простой командой `read` (см. главу 13), которая считывает каждую запись из файла. В этом сценарии не используется команда `cat` для перенаправления по каналу в цикл `while`, как в главе 13, а вместо этого осуществляется перенаправление в устройство STDIN с помощью команды `exec` (см. главу 14). Ниже показано, как должен выглядеть этот сценарий.

```
exec < $CONFIG_FILE
read FILE_NAME
```

Заслуживает внимание то, что для хранения содержимого файла конфигурации архива, а также каждой записи, считанной из файла, используется переменная. До тех пор пока команда `read` находит очередную запись, которая может быть считана из файла конфигурации, эта команда возвращает свидетельствующий об успехе код выхода 0 в переменной `?` (см. главу 10). Это значение можно использовать в цикле `while` для проверки того, произошло ли считывание всех записей из файла конфигурации:

```
while [ $? -eq 0 ]
do
...
read FILE_NAME
done
```

После достижения в команде `read` конца файла конфигурации возвращается код выхода, указывающий на неудачу. После этого цикл `while` завершается.

В цикле `while` должны выполняться следующие две операции. Прежде всего к списку архивируемых объектов добавляется имя файла или каталога. Но при этом необходимо предварительно проверить, существует ли вообще этот файл или каталог! Вполне может оказаться так, что пользователь удалит каталог из файловой системы и забудет обновить файл конфигурации архива. Можно проверить существование файла или каталога с помощью простой инструкции `if` или операторов сравнения файлов в команде `test` (см. главу 11). Если файл или каталог существует, он добавляется к списку файлов, подлежащих архивированию, путем изменения значения переменной `FILE_LIST`. В противном случае появляется предупреждающее сообщение, показанное ниже.

```
if [ -f $FILE_NAME -o -d $FILE_NAME ]
then
    # Если файл существует, добавить его имя к списку
    FILE_LIST="$FILE_LIST $FILE_NAME"
else
    # Если файл не существует, сформировать предупреждающее
    # сообщение
    echo
    echo "$FILE_NAME, does not exist."
    echo "Obviously, I will not include it in this archive."
    echo "It is listed on line $FILE_NO of the config file."
    echo "Continuing to build archive list..."
    echo
fi

#
FILE_NO=$((FILE_NO + 1))          # Увеличить номер строки/
номер файла на единицу.
```

Поскольку запись в файле конфигурации архива может указывать имя файла или каталога, в инструкции `if` проводится проверка на наличие и того и другого с использованием опций `-f` и `-d`. Параметр `-o` указывает на то, что операции проверки существования файла или каталога связаны оператором ИЛИ. Таким образом, после получения значения `true` по результатам любой проверки инструкция `if` возвращает значение `true`.

В сценарии предусмотрена переменная `FILE_NO`, которая позволяет проще отслеживать сведения о несуществующих каталогах и файлах. Благодаря этому после завершения работы сценария можно точно определить номер строки в файле конфигурации архива, в которой указан неверный или отсутствующий файл или каталог.

Теперь в нашем распоряжении достаточно информации для того, чтобы мы могли приступить к созданию сценария. В следующем разделе рассматривается поэтапное создание сценария архивирования.

Создание сценария архивирования данных за день

Сценарий `Daily_Archive` автоматически создает архив в указанном местоположении, используя текущую дату для однозначного определения файла. Ниже приведен код этой части сценария.

```
DATE='date +%y%m%d'
#
# задать имя файла архива
#
FILE=archive$DATE.tar.gz
#
# Задать имена файлов конфигурации и назначения
#
CONFIG_FILE=/home/user/archive/Files_To_Backup
DESTINATION=/home/user/archive/$FILE
#
```

Переменная `DESTINATION` служит для присоединения полного имени пути к имени архивированного файла. Переменная `CONFIG_FILE` указывает на файл конфигурации архива, который содержит перечень файлов и каталогов, подлежащих архивированию. В этот перечень в случае необходимости можно легко вносить изменения для указания одних объектов вместо других.

Теперь сценарий `Daily_Archive`, в котором предусмотрены все описанные выше операции, должен выглядеть следующим образом:

```
#!/bin/bash
#
# Daily_Archive - архивирование указанных файлов и каталогов
#####
#
# Определить текущую дату
#
DATE='date +%y%m%d'
#
# Set Archive File Name
#
FILE=archive$DATE.tar.gz
#
# Задать имена файлов конфигурации и назначения
#
CONFIG_FILE=/home/user/archive/Files_To_Backup
DESTINATION=/home/user/archive/$FILE
#
##### Основной сценарий #####
#
# Проверка того, существует ли файл конфигурации резервного копирования
#
if [ -f $CONFIG_FILE ] # Убедиться в том, что файл конфигурации
                        # все еще существует
then                   # Если он существует, просто продолжить работу
    echo
else                   # Если файл не существует, сформировать сообщение
    # об ошибке и выйти из сценария.
```

```

        echo
        echo "$CONFIG_FILE does not exist."
        echo "Backup not completed due to missing Configuration File"
        echo
        exit
    fi
    #
    # Сформировать список имен всех файлов, подлежащих резервному
    # копированию
    #
    FILE_NO=1          # Начать со строки 1 файла конфигурации.
    exec < $CONFIG_FILE # Задать вместо стандартного ввода имя
                        # файла конфигурации

    #
    read FILE_NAME      # Считать первую запись
    #
    while [ $? -eq 0 ]   # Сформировать список файлов, подлежащих
                        # резервному копированию.
    do
        # Убедиться в том, что файл или каталог существует.
        if [ -f $FILE_NAME -o -d $FILE_NAME ]
        then
            # Если файл существует, добавить его имя к списку.
            FILE_LIST="$FILE_LIST $FILE_NAME"
        else
            # Если файл не существует, сформировать предупреждающее
            # сообщение
            echo
            echo "$FILE_NAME, does not exist."
            echo "Obviously, I will not include it in this archive."
            echo "It is listed on line $FILE_NO of the config file."
            echo "Continuing to build archive list..."
            echo

            fi
        #
        FILE_NO=$((FILE_NO + 1)) # Увеличить номер строки/номер файла
                                # на единицу
        read FILE_NAME            # Считать следующую запись.
    done
    #
    #####
    #
    # Создать резервную копию файлов и произвести сжатие архива
    #
    tar -czf $DESTINATION $FILE_LIST 2> /dev/null
    #

```

Выполнение сценария архивирования данных за день

Задача проверки сценария `Daily_Archive` является несложной:

```
$ ./Daily_Archive  
  
/home/Does_not_exist, does not exist.  
Obviously, I will not include it in this archive.  
It is listed on line 4 of the config file.  
Continuing to build archive list...
```

Можно видеть, что в ходе работы сценария обнаружен один несуществующий каталог — `/home/Does_not_exist`. Эти результаты выполнения позволяют определить номера строк в файле конфигурации, содержащие ошибочные сведения о файлах и каталогах, а затем откорректировать перечень архивируемых объектов. Таким образом, теперь данные архивированы в `tar`-файле и находятся в безопасности.

Создание сценария ежечасного архивирования данных

Если приходится работать в крупномасштабной производственной среде, где изменения в файлах происходят часто, то может оказаться недостаточным создание единственного архива за весь день. Но при необходимости повысить частоту архивирования до величины, измеряемой часами, приходится учитывать еще одно соображение.

Применение прежнего подхода с указанием даты для обозначения имен файлов резервных копий, но с учетом того, что с помощью команды `date` в каждом файле должна также быть указана отметка времени, вскоре приводит к возникновению значительной путаницы. Становится очень сложно проводить поиск в каталоге с файлами, имена которых выглядят примерно так:

```
archive010211110233.tar.gz
```

Таким образом, лучше не размещать все файлы архивов в одной и той же папке, а создавать отдельные иерархии каталогов для архивированных файлов. Этот принцип демонстрируется на рис. 26.1.

В каталоге с архивами содержатся каталоги за каждый месяц года, а в качестве имени каждого каталога используется номер месяца. В каталоге за каждый месяц, в свою очередь, содержатся папки за каждый день месяца (как и для имен каталогов, используются числовые обозначения дней месяцев). Это позволяет обозначать отдельные файлы архива с указанием лишь отметки времени, а затем помещать их в каталог, соответствующий текущему дню и месяцу.

Но в связи с этим необходимо решить еще одну задачу. В сценарии должно быть предусмотрено автоматическое создание отдельных каталогов для месяцев и дней, а также автоматическое определение того, что эти каталоги уже существуют и не требуют создания.

Изучая опции командной строки для команды `mkdir` (см. главу 3), можно обнаружить, что в их состав входит опция командной строки `-p`. Эта опция позволяет создавать каталоги и подкаталоги с помощью одной команды и имеет дополнительное преимущество, заключающееся в том, что если каталог уже существует, то сообщение об ошибке не вырабатывается. Это как раз то, что требуется!

Итак, все готово для создания сценария `Hourly_Archive`. Ниже приведена его первая половина.

Создание иерархии архивных каталогов

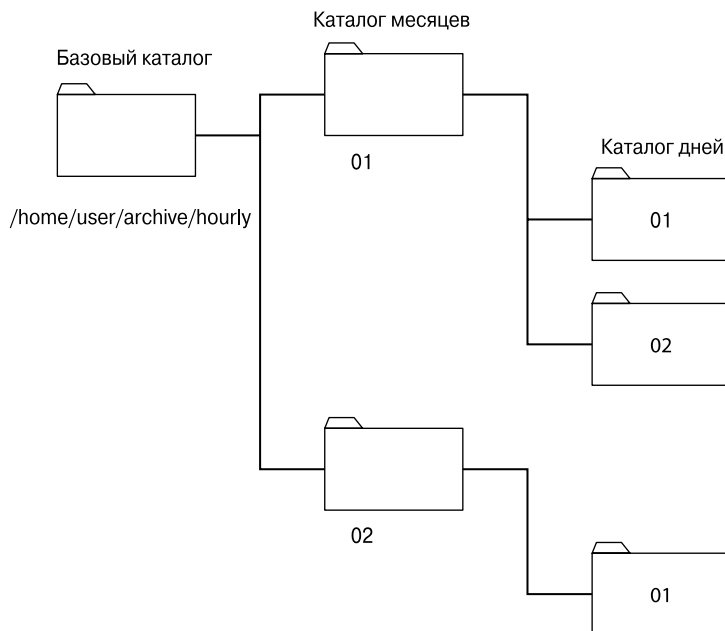


Рис. 26.1. Создание иерархии каталогов с архивами

```

#!/bin/bash
#
# Hourly_Archive - Ежечасное создание архива
#####
#
# Задать файл конфигурации
#
CONFIG_FILE=/home/user/archive/hourly/Files_To_Backup
#
# Задать базовое местоположение каталога с архивами
#
BASEDEST=/home/user/archive/hourly
#
# Получить текущие значения дня, месяца и времени суток
#
DAY='date +%d'
MONTH='date +%m'
TIME='date +%k%M'
#
# Создать каталог назначения для архива
#
mkdir -p $BASEDEST/$MONTH/$DAY
#
# Сформировать имя файл назначения для архива
#

```

```

DESTINATION=$BASEDEST/$MONTH/$DAY/archive$TIME.tar.gz
#
##### Основной сценарий #####
...

```

После того как в сценарии Hourly_Archive достигается часть, обозначенная как “Основной сценарий”, он становится точной копией сценария Daily_Archive. Таким образом, мы можем воспользоваться проделанной ранее работой!

В сценарии Hourly_Archive для получения значений дня и месяца, а также отметки времени, используемой для однозначного определения файла архива, применяется команда `date`. Затем с помощью этой информации создается каталог архива за день (или просто эта операция завершается без дополнительных сообщений, если каталог уже существует). Наконец, в сценарии используется команда `tar` для создания архива и сжатия его в виде `tar`-файла.

Выполнение сценария ежечасного архивирования данных

Как и в случае со сценарием Daily_Archive, следует вначале проверить работу сценария Hourly_Archive и лишь затем помещать его в таблицу `cron`:

```

$ cat /home/user/archive/hourly/Files_To_Backup
/home/Development/Simulation_Logs
/home/Production/Machine_Errors
$
$ ./Hourly_Archive
$
$ ls /home/user/archive/hourly/01/02
archive1601.tar.gz
$
$ ./Hourly_Archive
$
$ ls /home/user/archive/hourly/01/02
archive1601.tar.gz      archive1602.tar.gz
$

```

В первый раз выполнение сценария закончилось вполне успешно, созданы соответствующие каталоги месяца и дня, а затем получен файл архива. Исключительно в качестве дополнительной проверки, сценарий был запущен во второй раз, чтобы можно было понять, будут ли возникать проблемы в связи с тем, что каталоги архивов уже существуют. Сценарий снова был выполнен вполне успешно, что привело к созданию второго файла архива. Очевидно, что теперь сценарий можно включить в таблицу `cron`.

Управление учетными записями пользователей

Задачи управления учетными записями пользователей не сводятся лишь к добавлению, изменению и удалению учетных записей. Необходимо также рассматривать проблемы защиты, обеспечивать сохранение выполненной работы, а также контролировать правильность управления учетными записями. Решение всех этих задач может оказаться очень трудоемким. Это — еще один пример того, как программы на основе сценариев могут действительно способствовать экономии трудозатрат администратора!

Обязательные функции

Одной из наиболее сложных задач управления учетными записями является удаление учетной записи. При удалении учетной записи необходимо выполнить, по меньшей мере, четыре отдельные операции.

1. Проверить, действительно ли имя учетной записи пользователя, подлежащей удалению, указано правильно.
2. Уничтожить все процессы, выполняющиеся в настоящее время в системе, которые принадлежат удаляемой учетной записи.
3. Определить все файлы в системе, принадлежащие учетной записи.
4. Удалить учетную запись пользователя.

Выполняя все эти операции, вполне можно что-либо упустить. Для предотвращения подобных ошибок может использоваться программа на основе сценария командного интерпретатора, приведенная в этом разделе.

Получение правильного имени учетной записи

Первый шаг в процессе удаления учетной записи, который состоит в получении правильного имени учетной записи пользователя, подлежащей удалению, является самым важным. Это — интерактивный сценарий, поэтому для получения имени учетной записи можно использовать команду `read` (см. главу 13). Если есть вероятность того, что пользователь сценария может уйти и оставить запрос на указание имени учетной записи без ответа, можно применить опцию `-t` команды `read` и завершить работу по тайм-ауту после того, как пользователь сценария упустит возможность ответить на вопрос в течение 60 секунд:

```
echo "Please enter the username of the user "  
echo -e "account you wish to delete from system: \c"  
read -t 60 ANSWER
```

В повседневной работе часто приходится сталкиваться с тем, что не удастся сразу выполнить необходимое действие из-за постороннего вмешательства, поэтому лучше предоставить пользователям три попытки для ответа на запрос. Чтобы достичь этой цели, можно воспользоваться циклом `while` (см. главу 12) с опцией `-z` для проверки того, не является ли значение переменной `ANSWER` пустым. Значение переменной `ANSWER` должно иметь нулевую длину, когда сценарий впервые входит в цикл `while`, поскольку оператор вывода запроса находится в конце цикла:

```
while [ -z "$ANSWER" ]  
do  
...  
echo "Please enter the username of the user "  
echo -e "account you wish to delete from system: \c"  
read -t 60 ANSWER  
done
```

Теперь нам нужен способ сообщить пользователю сценария о том, что истек первый тайм-аут и ответ на запрос не получен, но есть еще один шанс ответить на запрос, и т.д. Инструкция `case` (см. главу 11) — это структурированная команда, которая идеально подходит для данного случая. Использование наращиваемых значений переменной `ASK_COUNT` позволяет определить

различные сообщения, отображаемые для пользователя сценария перед каждой попыткой. Код этого раздела сценария может выглядеть следующим образом:

```
case $ASK_COUNT in
2)
    echo
    echo "Please answer the question."
    echo
;;
3)
    echo
    echo "One last try...please answer the question."
    echo
;;
4)
    echo
    echo "Since you refuse to answer the question..."
    echo "exiting program."
    echo
    #
    exit
;;
esac
#
```

Теперь в сценарии предусмотрено все необходимое для того, чтобы с его помощью можно было узнать у пользователя, какая учетная запись должна быть удалена. В этом сценарии нужно будет получить ответы еще на несколько запросов, но уже затрачено так много кода для того, чтобы вывести для пользователя лишь один запрос! Поэтому целесообразно преобразовать эту часть кода в функцию (см. главу 16), чтобы можно было ею воспользоваться в нескольких местах в сценарии `Delete_User`.

Прежде всего необходимо объявить имя функции, в данном случае `get_answer`. Затем проведем очистку от всех предыдущих ответов на запросы, которые мог давать пользователь сценария, с помощью команды `unset` (см. главу 5). Код, необходимый для выполнения этих двух действий, выглядит следующим образом:

```
function get_answer {
#
unset ANSWER
```

В исходном коде требует также замены еще одна часть, касающаяся фактически выводимого запроса пользователю сценария. В сценарии каждый раз приходится получать ответы на разные запросы, поэтому предусмотрим две новые переменные, `LINE1` и `LINE2`, для обработки строк запроса:

```
echo $LINE1
echo -e $LINE2" \c"
```

Однако следует учитывать, что не каждый запрос требует для своего отображения двух строк. Некоторые запросы вполне помещаются в одной строке. Для решения задачи выбора подходящего формата запроса можно воспользоваться инструкцией `if` (см. главу 11). В функции проверяется, не является ли значение `LINE2` пустым, и в случае положительного ответа используется лишь значение `LINE1`:

```

if [ -n "$LINE2" ]
then
    echo $LINE1
    echo -e $LINE2" \c"
else
    echo -e $LINE1" \c"
fi

```

Наконец, в функции необходимо предусмотреть удаление использованных значений, для чего производится очистка переменных LINE2 и LINE1. Таким образом, теперь функция выглядит следующим образом:

```

function get_answer {
#
unset ANSWER
ASK_COUNT=0
#
while [ -z "$ANSWER" ]
do
    ASK_COUNT=$(( ASK_COUNT + 1 ))
#
    case $ASK_COUNT in
        2)
            echo
...
    esac
#
    echo
    if [ -n "$LINE2" ]
    then
        # Вывести 2 строки
        echo $LINE1
        echo -e $LINE2" \c"
    else
        # Вывести 1 строку
        echo -e $LINE1" \c"
    fi
#
    read -t 60 ANSWER
done
#
unset LINE1
unset LINE2
#
} # Конец определения функции get_answer

```

Чтобы вывести для пользователя сценария запрос, какая учетная запись должна быть удалена, необходимо задать значения нескольких переменных, а затем вызвать функцию `get_answer`. Благодаря использованию этой новой функции код сценария становится намного проще:

```

LINE1="Please enter the username of the user "
LINE2="account you wish to delete from system:"
get_answer
USER_ACCOUNT=$ANSWER

```


Остается вероятность, что пользователь сценария допустит опечатку при вводе имени учетной записи, поэтому необходимо предусмотреть проверку правильности введенного имени. Сделать это несложно, поскольку уже подготовлен код, с помощью которого можно выполнять операции вывода запросов и получения ответов:

```
LINE1="Is $USER_ACCOUNT the user account "  
LINE2="you wish to delete from the system? [y/n]"  
get_answer
```

После вывода запроса в сценарии происходит переход к этапу обработки ответа. И в этом случае для сохранения ответа пользователя сценария на запрос применяется переменная ANSWER. Если ответ пользователя является положительным, то можно сделать вывод, что учетная запись, подлежащая удалению, указана правильно и можно перейти в сценарии к дальнейшим действиям. Для обработки ответа может использоваться инструкция case (см. главу 11). В коде инструкции case следует обязательно предусмотреть, чтобы она обеспечивала проверку всех возможных вариантов положительных ответов.

```
case $ANSWER in  
y|Y|YES|yes|Yes|yEs|yeS|YEs|yES )  
#  
;;  
*)  
    echo  
    echo "Because the account, $USER_ACCOUNT, is not "  
    echo "the one you wish to delete, we are leaving the script..."  
    echo  
    exit  
;;  
esac
```

Необходимо отметить, что задачу обработки в сценарии положительных и отрицательных ответов от пользователей приходится решать достаточно часто. Таким образом, имеет смысл создать еще одну функцию, предназначенную для решения именно этой задачи. Для этого достаточно внести в приведенный выше код лишь несколько изменений, которые состоят в объявлении имени функции и добавлении переменных EXIT_LINE1 и EXIT_LINE2 в инструкцию case. Таким образом, после внесения этих изменений, предусмотрев также очистку нескольких переменных перед выходом, получим функцию process_answer:

```
function process_answer {  
#  
case $ANSWER in  
y|Y|YES|yes|Yes|yEs|yeS|YEs|yES )  
;;  
*)  
    echo  
    echo $EXIT_LINE1  
    echo $EXIT_LINE2  
    echo  
    exit  
;;  
esac  
#  
unset EXIT_LINE1
```

```
unset EXIT_LINE2
#
} #Конец определения функции process_answer
```

Теперь для обработки ответа достаточно применить следующий простой вызов функции:

```
EXIT_LINE1="Because the account, $USER_ACCOUNT, is not "
EXIT_LINE2="the one you wish to delete, we are leaving the script..."
process_answer
```

Можно видеть, что пользователь указал имя учетной записи, подлежащей удалению, а затем проверил правильность указанного значения. Тем не менее не лишне провести еще одну проверку и убедиться в том, что удаляемая учетная запись действительно существует в системе. Кроме того, желательно вывести для пользователя сценария все сведения учетной записи, чтобы он мог окончательно убедиться в том, что при указании учетной записи для удаления не произошла путаница. Для выполнения этих задач можно воспользоваться переменной `USER_ACCOUNT_RECORD` и присвоить ей результат поиска с помощью команды `grep` (см. главу 4) интересующей нас учетной записи в файле `/etc/passwd`. Опция `-w` позволяет указать, что при поиске этой конкретной учетной записи среди всех учетных записей выполнялось точное сопоставление слов:

```
USER_ACCOUNT_RECORD=$(cat /etc/passwd | grep -w $USER_ACCOUNT)
```

Если имя удаляемой учетной записи не будет найдено в файле `/etc/passwd`, можно сделать вывод, что интересующая нас учетная запись уже удалена или вообще никогда не существовала. Так или иначе, об этом необходимо сообщить пользователю сценария и завершить работу. Чтобы определить, успешно ли закончилось выполнение команды `grep`, можно проверить статус выхода этой команды. Если имя искомой учетной записи не будет обнаружено, то значение переменной `?` станет равным 1:

```
if [ $? -eq 1 ]
then
    echo
    echo "Account, $USER_ACCOUNT, not found. "
    echo "Leaving the script..."
    echo
    exit
fi
```

Но, даже если учетная запись определена в файле паролей, все равно необходимо проверить, действительно ли пользователь сценария правильно указал удаляемую учетную запись. Именно здесь полностью окупятся все наши усилия по подготовке функций! Остается лишь следующим образом задать требуемые переменные и вызвать функции:

```
echo "I found this record:"
echo $USER_ACCOUNT_RECORD
echo
#
LINE1="Is this the correct User Account? [y/n]"
get_answer
#
EXIT_LINE1="Because the account, $USER_ACCOUNT, is not"
EXIT_LINE2="the one you wish to delete, we are leaving the script..."
process_answer
```

Уничтожение процессов, связанных с учетной записью

К этому моменту сделано все необходимое для получения и проверки правильности имени учетной записи, подлежащей удалению. Чтобы можно было удалить учетную запись пользователя из системы, необходимо выполнить требование, согласно которому этой учетной записи не должны принадлежать какие-либо процессы, выполняющиеся в настоящее время. Таким образом, на следующем этапе мы должны обеспечить поиск и уничтожение таких процессов. Но эта задача значительно сложнее по сравнению с предыдущей!

При этом проще всего произвести поиск самих пользовательских процессов. Для этой цели в сценарии можно воспользоваться командой `ps` (см. главу 4) с опцией `-u` для поиска всех выполняемых процессов, принадлежащих учетной записи. Если таковые процессы имеются, то сведения о них должны быть выведены для пользователя сценария:

```
ps -u $USER_ACCOUNT
```

Чтобы определить, какой следующий шаг должен быть выполнен, можно воспользоваться значением статуса выхода команды `ps`, которое должно быть проанализировано с помощью инструкции `case`:

```
case $? in
1)
    echo "There are no processes for this account currently running."
    echo
;;
0)
    unset ANSWER
    LINE1="Would you like me to kill the process(es)? [y/n]"
    get_answer
...
esac
```

Если статус выхода равен 1, это означает, что в системе отсутствуют какие-либо процессы, которые принадлежат учетной записи, намеченной для удаления. Но если статус выхода равен 0, то в системе есть процессы, принадлежащие интересующей нас учетной записи. В этом случае в сценарии должен быть сформирован запрос к пользователю сценария на подтверждение операции уничтожения этих процессов. Эту задачу можно выполнить с помощью функции `get_answer`.

На первый взгляд кажется, что вслед за этим в сценарии должна быть вызвана функция `process_answer`. Но, к сожалению, теперь должна быть выполнена операция, слишком сложная, чтобы ее можно было поручить функции `process_answer`. Дело в том, что необходимо предусмотреть еще одну инструкцию `case` для обработки ответа пользователя сценария. Первая часть этой инструкции `case` во многом напоминает инструкцию, применяемую в функции `process_answer`:

```
case $ANSWER in
y|Y|YES|yes|Yes|yEs|yeS|YEs|yES ) # Если пользователь ответит
                                   # "yes", уничтожить процессы
                                   # пользователя.
...
;;
*) # Если пользователь дал любой другой ответ, кроме "yes",
   # не уничтожать
   echo
```

```

        echo "Will not kill the process(es)"
    echo
;;
esac

```

Вполне очевидно, что здесь в самой инструкции `case` нет ничего особенно сложного. Основные трудности возникают при подготовке той части инструкции `case`, которая касается обработки положительного ответа пользователя. Именно здесь должны быть выполнены операции, обеспечивающие уничтожение процессов, которые относятся к удаляемой учетной записи. Чтобы решить эту задачу, еще раз вызовем команду `ps`, но на этот раз направим вывод этой команды во временный файл отчета, а не на экран.

```
ps -u $USER_ACCOUNT > $USER_ACCOUNT_Running_Process.rpt
```

Затем для чтения файла отчета можно воспользоваться командой `exec` и циклом `while`, по аналогии с тем, как было организовано чтение файла конфигурации резервного копирования в сценарии `Daily_Archive`.

```

exec < $USER_ACCOUNT_Running_Process.rpt
read USER_PROCESS_REC
while [ $? -eq 0 ]
do
...
    read USER_PROCESS_REC
done

```

В самом цикле `while` необходимо извлечь значение идентификатора процесса (Process ID — PID) из каждой записи статуса каждого функционирующего процесса с помощью команды `cut`. После получения значения PID достаточно применить команду `kill` (см. главу 4) с опцией `-9` для безусловного завершения процесса.

```

USER_PID=$(echo $USER_PROCESS_REC | cut -d " " -f1)
kill -9 $USER_PID
echo "Killed process $USER_PID"

```

После того как будут уничтожены все процессы, принадлежащие учетной записи, в сценарии можно перейти к следующему шагу, который состоит в поиске всех файлов, относящихся к удаляемой учетной записи.

Поиск файлов учетной записи

При удалении учетной записи из системы рекомендуется создать архив всех файлов, принадлежащих этой учетной записи. Наряду с этим должна быть выполнена еще одна важная рекомендация: эти файлы должны быть удалены или переданы во владение другой учетной записи. После удаления учетной записи (допустим, имеющей идентификатор пользователя 1003) и создания новой, принадлежащей совсем другому пользователю, ей может быть снова назначен тот же идентификатор пользователя 1003, и если файлы прежнего пользователя не будут удалены, то окажутся в распоряжении совершенно постороннего лица! Вполне очевидно, какие предпосылки нарушения требований к безопасности данных возникают при таком развитии событий.

В самом сценарии `Delete_User` указанные операции с файлами не предусмотрены, но с его помощью создается отчет, который может использоваться в сценарии `Daily_Archive` в качестве файла конфигурации резервного копирования для получения архива с данными удаляемой учетной записи. Этим отчетом можно также воспользоваться в качестве источника сведений о том, какие файлы должны быть удалены или переданы во владение другой учетной записи.

Для поиска файлов пользователя можно применить команду `find`. Команда `find` с опцией `-u` выполняет во всей файловой системе поиск файлов, принадлежащих указанной учетной записи. Пример применения этой команды приведен ниже.

```
find / -user $USER_ACCOUNT > $REPORT_FILE
```

Очевидно, что выполняемые при этом операции гораздо проще по сравнению с теми, которые приходится выполнять применительно к процессам, принадлежащим учетной записи! А следующий шаг сценария `Delete_User`, который удаляет учетную запись, еще проще.

Удаление учетной записи

Прежде чем окончательно удалить учетную запись из системы, следует сделать все возможное, чтобы убедиться в правильности этого решения. Поэтому необходимо еще раз вывести запрос, не изменил ли пользователь сценария свое намерение удалить рассматриваемую учетную запись.

```
LINE1="Do you wish to remove $User_Account's account from system? [y/n]"
get_answer
#
EXIT_LINE1="Since you do not wish to remove the user account,"
EXIT_LINE2="$USER_ACCOUNT at this time, exiting the script..."
process_answer
```

И наконец, мы подошли к основной цели сценария и можем приступить к фактическому удалению учетной записи из системы. Для этого можно воспользоваться командой `userdel` (см. главу 6):

```
userdel $USER_ACCOUNT
```

Итак, все необходимые фрагменты сценария подготовлены, поэтому можно свести их воедино в виде законченной и удобной программы на основе сценария.

Создание сценария

Напомним, наш замысел состоял в том, чтобы в сценарии `Delete_User` было предусмотрено исчерпывающее взаимодействие с его пользователем. Поэтому необходимо предусмотреть большое количество сообщений, которые позволяли бы постоянно держать пользователя в курсе всего происходящего в ходе выполнения сценария.

В первой части сценария присутствуют объявления двух функций, `get_answer` и `process_answer`. Затем сценарий проходит четыре этапа удаления учетной записи: получение и подтверждение имени учетной записи; поиск и уничтожение процессов пользователя; создание отчета по всем файлам, принадлежащим учетной записи; а затем фактическое удаление учетной записи.

Окончательный вариант сценария `Delete_User` приведен ниже.

```
#!/bin/bash
#
#Delete_User - автоматизация 4 этапов удаления учетной записи
#
#####
# Определить функции
#
#####
```

```

function get_answer {
#
unset ANSWER
ASK_COUNT=0
#
while [ -z "$ANSWER" ]      #Выводить запрос, пока не будет получен ответ.
do
    ASK_COUNT=$(( ASK_COUNT + 1 ])
#
    case $ASK_COUNT in      #Если пользователь не дал ответа за
                            # отведенное время
        2)
            echo
            echo "Please answer the question."
            echo
            ;;
        3)
            echo
            echo "One last try...please answer the question."
            echo
            ;;
        4)
            echo
            echo "Since you refuse to answer the question..."
            echo "exiting program."
            echo
            #
            exit
            ;;
        esac
#
    echo
#
    if [ -n "$LINE2" ]
    then
        # Вывести 2 строки
        echo $LINE1
        echo -e $LINE2" \c"
    else
        # Вывести 1 строку
        echo -e $LINE1" \c"
    fi
#
#    Отвести 60 с на ответ до наступления тайм-аута
    read -t 60 ANSWER
done
# Удалить ненужные переменные
unset LINE1
unset LINE2
#
} #Конец определения функции get_answer
#
#####

```

```

function process_answer {
#
case $ANSWER in
y|Y|YES|yes|Yes|yEs|yeS|YEs|yES )
# Если пользователь ответит "yes", не выполнять никаких действий.
;;
*)
# Если пользователь даст любой другой ответ, кроме "yes", выйти из сценария
echo
echo $EXIT_LINE1
echo $EXIT_LINE2
echo
exit

;;
esac
#
# Удалить ненужные переменные
#
unset EXIT_LINE1
unset EXIT_LINE2
#
} # Конец определения функции process_answer
#
#####
# Конец определений функций
#
##### Основной сценарий #####
# Получить имя пользовательской учетной записи для проверки
#
echo "Step #1 - Determine User Account name to Delete "
echo
LINE1="Please enter the username of the user "
LINE2="account you wish to delete from system:"
get_answer
USER_ACCOUNT=$ANSWER
#
# Проверить с участием пользователя сценария, правильно ли задана
# учетная запись пользователя
#
LINE1="Is $USER_ACCOUNT the user account "
LINE2="you wish to delete from the system? [y/n]"
get_answer
#
# Вызвать функцию process_answer:
#     если пользователь дал любой другой ответ, кроме "yes", выйти из сценария
#
EXIT_LINE1="Because the account, $USER_ACCOUNT, is not "
EXIT_LINE2="the one you wish to delete, we are leaving the script..."
process_answer
#
#####

```

```

# Проверить, действительно ли учетная запись USER_ACCOUNT
# существует в системе
#
USER_ACCOUNT_RECORD=$(cat /etc/passwd | grep -w $USER_ACCOUNT)
#
if [ $? -eq 1 ]          # Если учетная запись не найдена, выйти
                        # из сценария
then
    echo
    echo "Account, $USER_ACCOUNT, not found. "
    echo "Leaving the script..."
    echo
    exit

fi
#
echo
echo "I found this record:"
echo $USER_ACCOUNT_RECORD
echo
#
LINE1="Is this the correct User Account? [y/n]"
get_answer
#
#
# Вызвать функцию process_answer:
#     если пользователь дал любой другой ответ, кроме "yes", выйти
#     из сценария
#
EXIT_LINE1="Because the account, $USER_ACCOUNT, is not "
EXIT_LINE2="the one you wish to delete, we are leaving the script..."
process_answer
#
#####
# Произвести поиск всех работающих процессов, которые принадлежат
# данной учетной записи
#
echo
echo "Step #2 - Find process on system belonging to user account"
echo
echo "$USER_ACCOUNT has the following processes running: "
echo
#
ps -u $USER_ACCOUNT      #Сформировать список работающих
                        # пользовательских процессов

case $? in
1)      # Этой учетной записи не принадлежит ни один работающий процесс
        #
        echo "There are no processes for this account currently running."
        echo
;;

```



```

0)      # Этой учетной записи принадлежат работающие процессы
      # Запросить пользователя сценария, следует ли уничтожить эти
      # работающие процессы
      #
      unset ANSWER
      LINE1="Would you like me to kill the process(es)? [y/n]"
      get_answer
      #
      case $ANSWER in
      y|Y|YES|yes|Yes|yEs|yeS|YES|yES )      # Если пользователь ответит
                                              # "yes", уничтожить процессы пользователя.
          #
          echo
          #
          # Удалить временный файл после получения одного из сигналов
          trap "rm $USER_ACCOUNT_Running_Process.rpt" SIGTERM SIGINT
      & SIGQUIT
          #
          # Сформировать список работающих пользовательских процессов
          ps -u $USER_ACCOUNT > $USER_ACCOUNT_Running_Process.rpt
          #
          exec < $USER_ACCOUNT_Running_Process.rpt      # Задать вместо
                                                         # стандартного ввода имя файла отчета
          #
          read USER_PROCESS_REC      # Первая запись будет пустой
          read USER_PROCESS_REC
          #
          while [ $? -eq 0 ]
          do
              # Получить PID
              USER_PID=$(echo $USER_PROCESS_REC | cut -d " " -f1)
              kill -9 $USER_PID
              echo "Killed process $USER_PID"
              read USER_PROCESS_REC
          done
          #
          echo
          rm $USER_ACCOUNT_Running_Process.rpt      # Удалить временный
                                                         # файл отчета.

      ;;
      *)      # Если пользователь даст любой другой ответ, кроме "yes", не
              # производить уничтожение.
              echo
              echo "Will not kill the process(es)"
              echo

      ;;
      esac
  ;;
esac
#####

```

```

# Создать отчет с перечнем всех файлов, принадлежащих пользовательской
# учетной записи
#
echo
echo "Step #3 - Find files on system belonging to user account"
echo
echo "Creating a report of all files owned by $USER_ACCOUNT."
echo
echo "It is recommended that you backup/archive these files,"
echo "and then do one of two things:"
echo "  1) Delete the files"
echo "  2) Change the files' ownership to a current user account."
echo
echo "Please wait. This may take a while..."
#
REPORT_DATE=`date +%y%m%d`
REPORT_FILE=$USER_ACCOUNT"_Files_"$REPORT_DATE".rpt"
#
find / -user $USER_ACCOUNT > $REPORT_FILE 2>/dev/null
#
echo
echo "Report is complete."
echo "Name of report:      $REPORT_FILE"
echo "Location of report:   `pwd`"
echo
#####
# Удалить учетную запись пользователя
echo
echo "Step #4 - Remove user account"
echo
#
LINE1="Do you wish to remove $User_Account's account from system?"
❏ [y/n]"
get_answer
#
# Вызвать функцию process_answer:
#     если пользователь дал любой другой ответ, кроме "yes",
#     выйти из сценария
#
EXIT_LINE1="Since you do not wish to remove the user account,"
EXIT_LINE2="$USER_ACCOUNT at this time, exiting the script..."
process_answer
#
userdel $USER_ACCOUNT          #удалить учетную запись пользователя
echo
echo "User account, $USER_ACCOUNT, has been removed"
echo
#

```

Очевидно, что мы сумели выполнить весьма значительный объем работы! Зато сценарий Delete_User позволяет сэкономить много усилий в ходе повседневной работы, а также

избежать многочисленных и неприятных ошибок, которые могут возникать при удалении учетных записей.

Выполнение сценария

Сценарий `Delete_User` предназначен для использования в качестве интерактивного, поэтому нет смысла помещать его в таблицу `cron`. Но мы обязаны провести еще одно важное мероприятие — убедиться в том, что сценарий работает в соответствии с ожидаемым. Проверим сценарий, удалив из системы учетную запись `consultant`, которая принадлежала временно назначенному консультанту.

```
$ ./Delete_User
Step #1 - Determine User Account name to Delete

Please enter the username of the user
account you wish to delete from system: consultant

Is consultant the user account
you wish to delete from the system? [y/n] y

I found this record:
consultant:x:1004:1001:Consultant,,,:/home/consultant:/bin/bash

Is this the correct User Account? [y/n] y

Step #2 - Find process on system belonging to user account
consultant has the following processes running:

PID TTY TIME CMD
There are no processes for this account currently running.

Step #3 - Find files on system belonging to user account
Creating a report of all files owned by consultant.

It is recommended that you backup/archive these files,
and then do one of two things:
1) Delete the files
2) Change the files' ownership to a current user account.

Please wait. This may take a while...

Report is complete.
Name of report: consultant_Files_110123.rpt
Location of report: /home/Christine

Step #4 - Remove user account

Do you wish to remove consultant's account from system? [y/n] y

User account, consultant, has been removed
$
```

Очевидно, что задание выполнено вполне успешно! Теперь в нашем распоряжении имеется программа на основе сценария, которой всегда можно воспользоваться при удалении учетных

записей. Еще одной привлекательной стороной этой программы является то, что в нее можно вносить любые изменения в соответствии с текущими потребностями организации.

Резюме

В настоящей главе показано, как использовать средства сценарной поддержки командного интерпретатора, представленные в этой книге, для создания программ Linux на основе сценариев. Администраторы, которым приходится заниматься обеспечением нормальной работы системы Linux, будь то большая многопользовательская система или собственная система, применяемая для личных целей, обязаны контролировать многие аспекты работы. Вместо выполнения необходимых для этого команд вручную можно создать программы на основе сценариев командного интерпретатора, позволяющие, пусть даже частично, автоматизировать эту работу.

В настоящей главе показано, как использовать команду `du` для определения того, как распределено пространство, потребляемое на диске. После этого было описано, как используются команды `sed` и `gawk` для извлечения конкретной информации из данных. Сценарии командного интерпретатора часто применяются для получения вывода с помощью команд `sed` и `gawk` и перенаправления полученных данных для дальнейшей обработки, поэтому важно знать, как это происходит.

Далее в главе рассматривается использование сценариев командного интерпретатора для архивирования и резервного копирования файлов данных в системе Linux. Для архивирования данных широко применяется команда `tar`. В главе показано, как использовать эту команду в сценариях командного интерпретатора для создания файлов архивов и как упростить управление файлами архивов с помощью архивных каталогов.

В завершение главы приведены сведения об использовании сценария командного интерпретатора для реализации четырех этапов удаления учетных записей. При этом еще раз подтвердилось следующее правило: применение функций для включения кода командного интерпретатора, неоднократно повторяющегося в сценарии, способствует созданию сценариев, более удобных для чтения и изменения. В рассматриваемом сценарии объединились многие структурированные команды, такие как `while` и `case`. В главе подчеркиваются различия в структуре сценариев, предназначенных для включения в таблицы `cron`, и интерактивных сценариев.

В следующей главе мы перейдем к изучению еще более развитых сценариев командного интерпретатора, которые позволят решать более сложные проблемы эксплуатации системы Linux.

Более сложные сценарии командного интерпретатора

ГЛАВА

27

В этой главе...

Текущий контроль
статистических данных
системы

Отслеживание проблем,
связанных с базой данных

Резюме

В настоящей главе рассматриваются более сложные методы организации функционирования сценариев командного интерпретатора, которые позволяют работать со сценариями в системе с применением наиболее разнообразных способов работы.

Текущий контроль статистических данных системы

В командном интерпретаторе `bash` предусмотрено большое количество программ, позволяющих следить за производительностью и использованием ресурсов в системе Linux. К сожалению, системному администратору обычно нелегко выделить время для работы с этими программами, особенно если ему приходится управлять несколькими системами. Намного уменьшить трудоемкость работы системного администратора позволяет использование нескольких сценариев командного интерпретатора для создания отчетов на основе выходных данных этих программ.

В настоящем разделе описан процесс создания некоторых усовершенствованных сценариев командного интер-

претатора, способных оказать помощь в текущем контроле над различными статистическими данными по производительности системы.

Отчет с моментальным снимком системы

Хорошим способом, позволяющим оставаться информированным о производительности системы и использовании ресурсов, является получение отчетов с моментальными снимками. *Отчет с моментальным снимком* — это своего рода представление статистических данных системы на определенный момент времени. Отчет такого типа можно сравнить с пояснительной запиской руководству об исправности системы.

Сценарий создания отчета с моментальным снимком может выполняться несколько раз в сутки, когда нужно будет получить сведения о состоянии системы. Мало того, в сценарии можно даже предусмотреть отправку отчета заинтересованным лицам по электронной почте.

Обязательные функции

Для создания отчета с моментальным снимком потребуется использование следующих четырех команд командного интерпретатора `bash`: `uptime`, `df`, `free` и `ps`. Эти разнообразные команды предоставляют все необходимые статистические данные.

Продолжительность работы системы

Наиболее важной среди всех команд получения статистических данных о работе системы является первая из указанных команд, `uptime`:

```
$ uptime
14:15:23 up 1 day, 5:10, 3 users, load average: 0.66, 0.54, 0.33
```

Команда `uptime` предоставляет несколько разных фрагментов информации, которыми можно воспользоваться:

- текущее время;
- количество дней, часов и минут, в течение которых система непрерывно функционировала;
- количество пользователей, зарегистрированных в настоящее время в системе;
- средняя загрузка за одну минуту, пять и пятнадцать минут.

Одним из наиболее важных статистических показателей, который должен быть приведен в отчете с моментальным снимком, является продолжительность непрерывного функционирования системы. Но при попытке получения этого статистического показателя могут возникнуть проблемы. Дело в том, что продолжительность непрерывной работы может измеряться всего лишь несколькими минутами или составлять значительно более продолжительный интервал, измеряемый сутками. (Ходят слухи, что некоторые серверы Linux со времени запуска продолжают работать уже в течение многих лет!) Ниже приведен вывод команды `uptime`, относящийся к системе, которая находилась в рабочем состоянии на протяжении нескольких часов.

```
$ uptime
18:00:20 up 8:55, 3 users, load average: 0.62, 0.45, 0.37
```

А следующий вывод `uptime` относится к системе, которая непрерывно функционировала в течение нескольких суток:

```
$ uptime
13:29:32 up 3 days, 4:24, 4 users, load average: 1.44, 0.83, 0.46
```

Вполне очевидно, что на основании этих листингов довольно сложно правильно определить требуемое значение статистического показателя. Безусловно, команда `gawk` (см. главу 16) позволяет решить эту задачу без каких-либо затруднений применительно к системе, которая находилась в рабочем состоянии в течение нескольких суток:

```
$ uptime | gawk '{print $2,$3,$4,$5}'
up 3 days, 4:27,
```

Но при использовании той же команды `gawk` по отношению к данным системы, которая проработала всего лишь несколько часов, происходит следующее:

```
$ uptime | gawk '{print $2,$3,$4}'
up 8:59, 3 users,
```

Чтобы успешно справиться с указанной задачей, можно воспользоваться дополнительными возможностями команды `gawk`, связанными с применением инструкции `if` (см. главу 19). Если система была в рабочем состоянии в течение нескольких суток, то переменная `$4` в сценарии `gawk` будет содержать слово “days”. Чтобы проверить переменную `$4` на наличие строки “days”, можно попытаться воспользоваться командой `gawk`, которая выглядит следующим образом:

```
$ uptime |
> gawk '{if ($4 == "days") {print $2,$3,$4,$5} else {print $2,$3}}'
up 9:23,
```

Но, возможно, придется столкнуться с ситуацией, в которой система находилась в рабочем состоянии только одни сутки. Это означает, что должна быть также проведена проверка на наличие строки “day”. Для этого можно применить логический оператор ИЛИ (`||`). Таким образом, применяемая команда `gawk` теперь будет выглядеть следующим образом:

```
gawk '{if ($4 == "days" || $4 == "day") {print $2,$3,$4,$5} else {print $2,$3}}'
```

Но после этого в выводе обнаруживаются запятые, которые не совсем соответствуют формату результата. Эти запятые можно просто заменить пробелами с помощью команды `sed`.

```
$ uptime | sed -n '/,/s/,/ /gp' |
> gawk '{if ($4 == "days" || $4 == "day") {print $2,$3,$4,$5}
> else {print $2,$3}}'
up 9:32
```

Теперь полученные данные выглядят намного лучше! Исключительно ради дополнительной проверки проведем испытание этой команды в системе, которая находилась в рабочем состоянии больше чем одни сутки:

```
$ uptime | sed -n '/,/s/,/ /gp' |
> gawk '{if ($4 == "days" || $4 == "day") {print $2,$3,$4,$5}
> else {print $2,$3}}'
up 3 days 4:34
```

Полученные данные также выглядят хорошо. После этого приступим к работе еще над одной командой получения данных о производительности, чтобы воспользоваться ею для создания отчета с моментальным снимком системы.

Использование дисковой памяти

Следующей командой, с помощью которой можно включить необходимые сведения в отчет с моментальным снимком, является команда `df`. Команда `df` отображает статистические данные об использовании места на диске:

```
$ df
Filesystem      1K-blocks    Used    Available  Use%    Mounted on
/dev/sda2       307465076    185930336  105916348   64%     /
none            503116       240       502876      1%      /dev
none            508716       252       508464      1%      /dev/shm
none            508716       100       508616      1%      /var/run
...
```

В частности, команда `df` позволяет получить текущие статистические данные о том, как распределяется пространство на диске, для всех физических и виртуальных дисков в системе. В команду `df` можно передать имя отдельного диска для получения информации только об этом конкретном диске. Эта команда обеспечивает вывод данных в формате, более удобном для чтения, для чего необходимо задать опцию `-h` (сокращение от *human readable* — предназначенный для чтения):

```
$ df -h /dev/sda2
Filesystem Size Used Avail Use% Mounted on
/dev/sda2  294G 178G 102G   64%    /
$
```

Для синтаксического анализа и формирования отчета, который содержит лишь необходимые данные, снова воспользуемся командами `sed` и `gawk`. В частности, в выводе команды `df` присутствует строка заголовка, которая не требуется для включения в отчет, поэтому воспользуемся командой `sed` для поиска лишь уникальных строк данных. Признаком для поиска строк данных является подстрока `% /`, которая присутствует только в них. Однако в программе `sed` косая черта распознается как компонент структуры команд этой программы. Чтобы найти решение этой проблемы, поместим экранирующий символ `\` перед косой чертой. После этого в программе `sed` указанная выше косая черта рассматривается как часть искомой строки и поиск проводится должным образом. Наконец, воспользуемся командой `gawk` для вывода значения использования дисковой памяти в процентах, содержащегося в переменной `$5` в сценарии `gawk`. Вся командная строка будет выглядеть следующим образом:

```
$ df -h /dev/sda2 | sed -n '/% \//p' | gawk '{print $5}'
64%
```

Очевидно, что полученные результаты вполне удовлетворительны. Теперь перейдем к получению моментального снимка данных о распределении оперативной памяти в системе.

Использование памяти

Для получения статистических данных о распределении памяти можно воспользоваться несколькими командами командного интерпретатора `bash`. Исходя из поставленной цели, сосредоточимся на рассмотрении команды `free`.

Команда `free` отображает общий объем физической памяти и показывает сведения о том, какая часть памяти остается свободной и какая ее часть распределена. Кроме того, эта команда выводит такие же данные для памяти страничного обмена и включает сведения о буферах ядра. Ниже приведен пример вывода команды `free`.

```
$ free
              total          used          free      shared    buffers     cached
Mem:        1017436         890148         127288           0         31536         484100
-/+ buffers/cache: 374512      642924
Swap:        200776           0           200776
$
```


Перед этим статистические данные об использовании дисковой памяти были представлены в процентах, поэтому целесообразно применить такой же формат при отображении сведений о распределении оперативной памяти. Но, как показывают полученные результаты, команда `free` не предоставляет сведения об использовании памяти, которые были бы выражены в процентах! Тем не менее это затруднение не сложно преодолеть. Значения в процентах можно вычислить с помощью команды `gawk`.

Вывод команды `free` уже имеет удобный формат с разделением пробелами, поэтому в программе `gawk` остается лишь извлечь данные этих двух полей, необходимые для вычисления, с помощью именованных переменных, а затем воспользоваться данными, содержащимися в этих переменных. Переменная `$2` позволяет определить общий объем имеющейся памяти, а переменная `$3` — общий объем используемой памяти. После деления значения переменной `$3` на значение `$2` получаем с помощью программы `gawk` данные об используемой памяти в процентах:

```
$ free | sed -n '2p' | gawk 'x = ($3 / $2) {print x}'
0.87165
```

Обратите внимание на то, что инструкция `sed` использовалась в этой команде для извлечения только второй строки данных из вывода команды `free`, что позволяет “отбросить” строку заголовка. Полученное числовое значение находится все еще не в том формате, который требуется для отчета. Чтобы сформировать значение в процентах, необходимо умножить это число на 100 и отбросить дробную часть с помощью функции `int` языка `gawk`, предназначенной для преобразования в целое число. Наконец, присоединим знак процента с помощью команды `sed`:

```
$ free | sed -n '2p' |
> gawk 'x = int(($3 / $2) * 100) (print x)' |
> sed 's/$/%/'
87%
```

После этого статистические данные об использовании памяти принимают вид, требуемый для отчета. Перейдем к решению последней задачи — получению сведений о процессах-зомби.

Процессы-зомби

Процессами-зомби в системе Linux называют процессы, находящиеся в неопределенном состоянии. Это — процессы, которые завершили свою работу, но по каким-то причинам остались незаконченными. Таким образом, процессы-зомби, как и персонажи из мира фантастики, которые определили их название, ни живы ни мертвы.

Безусловно, если в системе Linux присутствует один или два процесса-зомби, этому вряд ли следует придавать большое значение. Но процессы-зомби оставляют за собой идентификаторы процессов, которые не возвращаются обратно в таблицу идентификаторов процессов до тех, пока не произойдет их уничтожение. Поэтому необходимо поставить под контроль еще один ряд статистических показателей в системе, которые позволяют отслеживать текущее количество процессов-зомби.

Это можно легко сделать с помощью команды `ps` (см. главу 4). Команда `ps` с опциями `-al` возвращает сведения обо всех процессах в системе, включая то, в каком состоянии обработки они находятся. Состояние `Z` указывает на то, что процесс перешел в категорию процессов-зомби:

```
$ ps -al
F S  UID  PID PPID C PRI NI ADDR  SZ WCHAN  TTY          TIME CMD
0 T 1000 2174 2031 0 80  0 - 2840 signal pts/0 00:00:00 mail
0 T 1000 2175 2031 0 80  0 - 2839 signal pts/0 00:00:00 mail
```

```

1 Z 1000 8779 8797 0 80 0 - 0 exit pts/0 00:00:00 zomb <defunct>
0 R 1000 8781 8051 0 80 0 - 1094 - pts/5 00:00:00 ps
$

```

И в этом случае для получения требуемого формата вывода можно воспользоваться командой `gawk`. После передачи вывода команды `ps` по каналу в команду `gawk` остается только извлечь данные из полей \$2 и \$4, в которых указаны идентификаторы процессов и состояние их обработки. Однако на данном этапе задача состоит в выявлении только процессов-зомби, поэтому необходимо выделить в выводе информацию лишь об этих процессах, для чего можно применить команду `grep`. Требуемая команда может выглядеть следующим образом:

```

$ ps -al | gawk '{print $2,$4}' | grep Z
Z 8779

```

Очевидно, что полученные данные вполне подходят для нашего отчета с моментальным снимком. Теперь, после подготовки всех необходимых частей, нам остается лишь соединить их в рабочем сценарии.

Создание сценария получения моментального снимка

При подготовке сценария необходимо в первую очередь задать различные переменные, которые будут использоваться в тексте сценария. Код определения этих переменных должен быть приведен в начале сценария, чтобы можно было проще вносить в него изменения в будущем.

В отчете с моментальным снимком потребуется отметка даты, поэтому необходимо предусмотреть переменную для хранения значения даты, `DATE`. Что касается статистических данных о производительности, то может потребоваться сбор статистики более чем для одного диска. Поэтому предусмотрена переменная типа массива, `DISKS_TO_MONITOR`, которая включает список дисков во всей системе. Предпочтения системных администраторов в отношении используемых программ электронной почты различаются, поэтому для почты необходимо включить переменную `MAIL`, которая определяет, какая служебная программа используется на локальном компьютере (см. главу 25). Кроме того, может оказаться так, что в конкретной системе команда `mail` расположена не в таком каталоге, как в других системах, поэтому необходимо предусмотреть использование команды `which` (см. главу 23) для определения местоположения команды `mail`. Для указания имени отправителя электронной почты с отчетами предусмотрена переменная `MAIL_TO`. И наконец, имя отчета определяется с помощью переменной `REPORT`.

Итак, раздел с определениями переменных в сценарии получения моментальных снимков может выглядеть следующим образом:

```

# Задание переменных сценария
#
DATE='date +%m%d%Y'
DISKS_TO_MONITOR="/dev/sda1 /dev/sda2"
MAIL='which mutt'
MAIL_TO=user
REPORT=/home/user/Documents/$DATE.rpt

```

Формирование отчета и отправка его по электронной почте осуществляются в одном и том же сценарии, поэтому применяемый в сценарии дескриптор файла необходимо сохранить, а затем восстановить (см. главу 14). Для сохранения дескриптора файла и перенаправления вывода в устройство `STDOUT` применяются следующие команды командного интерпретатора:

```

# Создание файла отчета
#

```

```

ехес 3>&1 #Сохранение дескриптора файла
#
ехес 1> $REPORT #Перенаправление вывода в файл отчета

```

В следующем коде, предшествующем команде mail, восстанавливается сохраненный дескриптор файла устройства STDOUT:

```

# Восстановление дескриптора файла и отправка отчета по почте
#
ехес 1>&3 #Восстановление STDOUT в качестве устройства вывода

```

После соединения всех частей формируется сценарий, который выглядит следующим образом:

```

#!/bin/bash
#
# Snapshot_Stats - формирование отчета со статистическими данными
# системы
#####
# Задание переменных сценария
#
DATE=`date +%m%d%Y`
DISKS_TO_MONITOR="/dev/sda1 /dev/sda2"
MAIL=`which mutt`
MAIL_TO=user
REPORT=/home/user/Documents/Snapshot_Stats_${DATE}.rpt
#
#####
# Создание файла отчета
#
ехес 3>&1 #Сохранение дескриптора файла
#
ехес 1> $REPORT #Перенаправление вывода в файл отчета
#
#####
#
echo
echo -e "\t\tDaily System Report"
echo
#
#####
# Введение отметки времени в отчет
#
echo -e "Today is " `date +%m/%d/%Y`
echo
#
#####
# 1) Сбор статистических данных о производительности работы системы
#
echo -e "System has been \c"
uptime | sed -n '/,/s/,/ /gp' |
    gawk '{if ($4 == "days" || $4 == "day")
        {print $2,$3,$4,$5}
    }'

```

```

else {print $2,$3}}'

#
#####
# 2) Сбор статистических данных об использовании диска
#
echo
for DISK in $DISKS_TO_MONITOR      #Выполнить операции по проверке
                                   # пространства на диске в цикле
do
    echo -e "$DISK usage: \c"
    df -h $DISK | sed -n '/% \//p' | gawk '{print $5}'
done
#
#####
# 3) Сбор статистических данных об использовании памяти
#
echo
echo -e "Memory Usage: \c"
#
free | sed -n '2p' |
    gawk 'x = int(($3 / $2) *100) {print x}' |
    sed 's/$/%/'
#
#####
# 4) Сбор данных о количестве процессов-зомби
#
echo
ZOMBIE_CHECK='ps -al | gawk '{print $2,$4}' | grep Z'
#
if [ "$ZOMBIE_CHECK" = "" ]
then
    echo "No Zombie Process on System at this Time"
else
    echo "Current System Zombie Processes"
    ps -al | gawk '{print $2,$4}' | grep Z
fi
echo
#####
# Восстановление дескриптора файла и отправка отчета по почте
#
exec 1>&3      #Восстановление STDOUT в качестве устройства вывода
#
$MAIL -a $REPORT -s "System Statistics Report for $DATE"
-- $MAIL_TO < /dev/null
#
#####
# Очистка
#
rm -f $REPORT
#

```

Необходимо отметить, что после достижения в сценарии шага #4 вначале происходит проверка на наличие процессов-зомби с использованием результатов команды `ps`, которые присвоены переменной `ZOMBIE_CHECK`. Если обнаруживается, что переменная содержит пустое значение, то в сценарии формируется сообщение о том, что процессы-зомби отсутствуют. В противном случае выводится список всех текущих процессов с состоянием `Z`.

После этого проведем проверку нового сценария получения отчета с моментальным снимком, чтобы определить, насколько успешно он работает.

```
$  
$ ./Snapshot_Stats
```

Очевидно, что объем выводимой информации не очень велик. Но после вызова программы почтового клиента, в данном случае `Mutt`, можно видеть, как будет получен по электронной почте отчет с моментальным снимком (рис. 27.1).

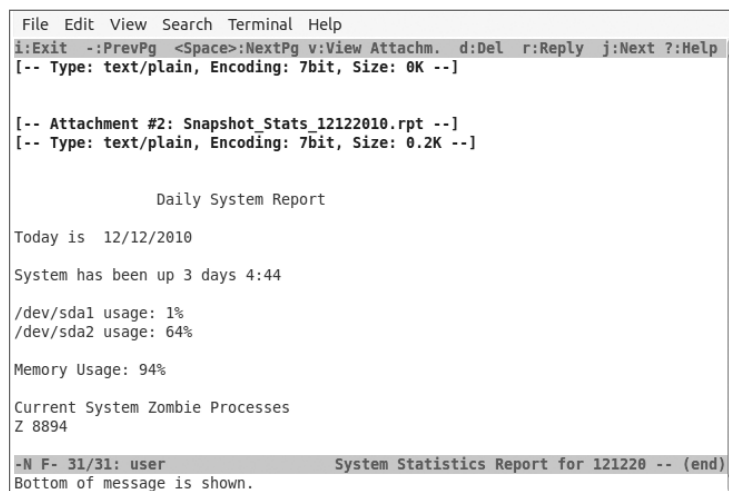


Рис. 27.1. Отображение отчета с моментальным снимком в программе `Mutt`

Безусловно, этот отчет с моментальным снимком может принести большую пользу, поскольку позволяет получить полную картину состояния исправности системы. Но отчет такого типа не позволяет выявлять различные тенденции изменения производительности и использования ресурсов, которые проявляются в течение продолжительного времени. Поэтому перейдем к созданию ряда сценариев, которые позволяют получить именно такую информацию.

Отчет со статистическими данными системы

Основными статистическими показателями любой системы Linux являются данные об использовании памяти и процессора. Если эти значения начинают выходить из-под контроля, то за очень короткое время могут произойти нарушения в работе системы. В настоящем разделе показано, как писать сценарии, позволяющие контролировать и отслеживать использование памяти и процессора в системе Linux в течение продолжительного времени.

Обязательные функции

Назначение этого первого сценария состоит в сборе статистических данных о производительности в файле данных для последующего использования. Прежде всего необходимо точно определить, какие данные должны быть собраны с помощью сценария и какие команды должны для этого использоваться.

Для этого сценария снова будет использоваться команда `uptime`, но на этот раз вместо получения данных о продолжительности времени работы системы будут получены данные о количестве пользователей в системе. Загрузка памяти и процессора непосредственно зависит от количества пользователей. С помощью инструкции `sed` в сценарии будет производиться поиск слова “users” в выводе команды `uptime`. После обнаружения этого слова в текущей строке инструкция `sed` должна удалить остальную часть текстовой строки, включая слово “users”. В результате в строке останется последний элемент — количество пользователей в системе. Для получения этого последнего элемента данных можно использовать переменную `gawk`, `NF`. Вся командная строка и полученный вывод будут выглядеть следующим образом:

```
$ uptime | sed 's/users.*$//' | gawk '{print $NF}'
4
```

Нагрузка системы — еще один важный статистический показатель, который можно извлечь с помощью синтаксического анализа из вывода команды `uptime`. Данные о нагрузке системы показывают, насколько занят работой процессор в системе. Средняя нагрузка, равная 1, означает, что в однопроцессорной системе процессор постоянно занят. Но если в системе имеются два процессора, то показатель 1 указывает, что каждый процессор в среднем загружен лишь на 50%. Нагрузка системы, определяемая с помощью команды `uptime`, вычисляется как среднее за последнюю минуту, за последние пять и пятнадцать минут. В нашем сценарии предусмотрено получение статистических данных о средней нагрузке системы за последние пятнадцать минут; эти данные удобно расположены в конце текстовой строки вывода:

```
$ uptime | gawk '{print $NF}'
0.19
```

Еще одной важной командой, позволяющей получать информацию о системе, является `vmstat`, пример вывода которой приведен ниже.

```
$ vmstat
procs -----memory----- ---swap-- ----io---- -system--
⌘ ----cpu----
r b   swpd   free   buff   cache   si  so        bi  bo        in  cs
⌘ us sy id wa
0 0   11328   42608 165444   502500    0  0          1  2        68  10
⌘ 1  0 99  0
```

При первом вызове команды `vmstat` на выполнение отображаются средние значения, начиная с последней перезагрузки. Для получения текущих статистических данных необходимо выполнить команду `vmstat` с параметрами командной строки:

```
$ vmstat 1 2
procs -----memory----- ---swap-- ----io---- -system--
⌘ ----cpu----
r b   swpd   free   buff   cache   si  so        bi  bo        in  cs
⌘ us sy id wa
0 0   11328   43112 165452   502444    0  0          1  2        68  10
⌘ 1  0 99  0
```

```
0 0 11328 40540 165452 505064 0 0 0 0 58 177
❏ 1 1 98 0
```

Во второй строке содержатся текущие статистические данные для системы Linux. Очевидно, что вывод команды `vmstat` для неискушенного человека выглядит довольно загадочным. В табл. 27.1 описан каждый из символов в выводе.

Таблица 27.1. Символы в выводе команды `vmstat`

Символ	Описание
<code>r</code>	Количество процессов, ожидающих получения процессорного времени
<code>b</code>	Количество процессов в непрерываемом режиме ожидания
<code>swpd</code>	Объем используемой виртуальной памяти (Мбайт)
<code>free</code>	Объем неиспользуемой физической памяти (Мбайт)
<code>buff</code>	Объем памяти, используемой в качестве пространства буферов (Мбайт)
<code>cache</code>	Объем памяти, используемой в качестве пространства кеша (Мбайт)
<code>si</code>	Объем памяти, загруженной с диска (Мбайт)
<code>so</code>	Объем памяти, выгруженной на диск (Мбайт)
<code>bi</code>	Количество блоков, полученных с блочного устройства
<code>bo</code>	Количество блоков, отправленных на блочное устройство
<code>in</code>	Количество прерываний процессора в секунду
<code>cs</code>	Количество переключений контекста процессора в секунду
<code>us</code>	Процентная доля процессорного времени, затраченного на выполнение кода, отличного от кода ядра
<code>sy</code>	Процентная доля процессорного времени, затраченного на выполнение кода ядра
<code>id</code>	Процентная доля процессорного времени, затраченного на простой
<code>wa</code>	Процентная доля процессорного времени, затраченного на ожидание ввода-вывода

Очевидно, что количество предоставляемой информации весьма велико. Но для наших целей вполне подойдут статистические данные об объеме свободной памяти и процентной доле процессорного времени, затраченного на простой.

Заслуживает также внимания то, что в вывод команды `vmstat` включена информация шапки таблицы, которая фактически не требуется в тех данных, которые должны присутствовать в результатах сценария. Для решения проблемы исключения ненужных строк вывода можно воспользоваться командой `sed`, чтобы отобразить только те строки, в которых содержатся числовые значения:

```
$ vmstat 1 2 | sed -n '/[0-9]/p'
1 0 11328 38524 165476 506548 0 0 1 2 68 10 1 0 99 0
0 0 11328 35820 165476 509528 0 0 0 0 82 160 1 1 98 0
```

Безусловно, мы приблизились к искомому результату, но теперь необходимо получить только вторую строку данных. Это требование может быть выполнено с помощью еще одного вызова инструкции `sed`:

```
$ vmstat 1 2 | sed -n '/[0-9]/p' | sed -n '2p'
0 0 11328 36060 165484 509560 0 0 0 0 58 175 1 1 99 0
```

После этого можно легко извлечь именно то значение данных, которое требуется, с помощью инструкции `gawk`.

Наконец, необходимо обозначить каждую запись с данными о производительности системы с помощью отметки даты и времени, чтобы можно было узнать, когда были получены статистические данные. Для этого достаточно воспользоваться командой `date` и в ее вызове указать, в каком формате должны быть представлены данные в записи данных.

```
$ date +"%m/%d/%Y %k:%M:%S"
12/12/2010 13:55:31
```

Теперь рассмотрим, каким образом следует регистрировать значения данных о производительности. Если выборка и регистрация данных должны осуществляться на регулярной основе, то часто лучше всего выводить данные непосредственно в файл журнала. Этот файл можно создать в своем рабочем каталоге `$HOME`, затем добавлять к нему данные после каждого выполнения сценария командного интерпретатора. А когда потребуется просмотреть полученные результаты, можно просто открыть этот файл журнала.

Должно быть сделано все возможное, чтобы данные в файле журнала стали легко доступными для чтения. Предусмотрено много разных методов, которые могут использоваться для форматирования данных в файле журнала. Одним из широко применяемых форматов является формат с разделителями-запятыми (Comma-Separated Values — CSV). Этот формат предусматривает размещение каждой записи данных в отдельной строке и разделение полей данных в записи с помощью запятых. Широкое применение этого формата обусловлено тем, что он позволяет без особых затруднений импортировать данные в программы поддержки электронных таблиц, обслуживания баз данных и формирования отчетов.

Данный сценарий также предусматривает сохранение данных в файле с записями в формате CSV. Этот файл будет использоваться в сценарии формирования отчетов, который будет создан далее в этой главе.

Создание сценария сбора данных

Сценарий, предназначенный для сбора данных, должен быть достаточно простым, поскольку для него уже подготовлены все необходимые функции. Ниже показано, как выглядят результаты нашей работы.

```
#!/bin/bash
#
# Capture_Stats - сбор статистических данных о производительности
# системы
#####
# Задание переменных сценария
#
REPORT_FILE=/home/user/Documents/capstats.csv
DATE='date +%m/%d/%Y'
TIME='date +%k:%M:%S'
#
#####
# Сбор статистических данных о производительности
#
USERS='uptime | sed 's/users.*$//' | gawk '{print $NF}''
LOAD='uptime | gawk '{print $NF}''
#
FREE='vmstat 1 2 | sed -n '/[0-9]/p' | sed -n '2p' |
gawk '{print $4}''
IDLE='vmstat 1 2 | sed -n '/[0-9]/p' | sed -n '2p' |
```



```

gawk '{print $15}'`
#
#####
# Передача статистических данных в файл отчета
#
echo "$DATE,$TIME,$USERS,$LOAD,$FREE,$IDLE" >> $REPORT_FILE
#

```

В этом сценарии осуществляется синтаксический анализ статистических данных, полученных с помощью команд `uptime` и `vmstat`, в ходе чего полученные значения сохраняются в переменных. Затем значения этих переменных сохраняются в файле журнала `REPORT_FILE` с разделением запятыми наряду с отметками даты и времени. Следует отметить, что данные каждый раз присоединяются к файлу журнала, для чего применяется символ перенаправления (`>>`). Это позволяет снова и снова добавлять данные в файл журнала, пока в этом есть необходимость.

После создания сценария, по-видимому, потребуется его проверить, вызвав из командной строки, и только после этого передать на регулярное выполнение в таблицу `cron`:

```

$ cat capstats.csv
12/09/2010, 9:06:50,2,0.29,645988,99
12/09/2010, 9:07:55,2,0.28,620252,100
12/09/2010, 9:40:51,3,0.37,474740,100
12/10/2010,14:36:46,3,0.30,46640,98
12/12/2010, 7:16:26,4,0.25,27308,98
12/12/2010,13:28:53,4,0.42,58832,100

```

В следующем сценарии необходимо создать отчет из данных в формате CSV и отправить его по электронной почте соответствующим лицам. Очевидно, что сценарий формирования отчета должен представлять собой отдельный сценарий, поэтому можно предусмотреть его вызов на выполнение с помощью таблицы `cron` по иному расписанию по сравнению со сценарием сбора данных `Capture_Stats`. Благодаря этому можно будет выбрать, сколько данных должно быть собрано, прежде чем будет сформирован отчет.

Подготовка сценария формирования отчета

Итак, мы имеем файл, заполненный данными в нужном формате (но не предназначенными для непосредственного восприятия), поэтому перейдем к разработке сценария, который позволяет сформировать удобный для чтения отчет. Лучшим инструментом для этого является команда `gawk`.

Команда `gawk` позволяет извлекать исходные данные из файла в формате CSV, а затем представлять их в любом необходимом виде. Вначале проверим такую возможность с помощью командной строки с использованием нового файла `capstats.csv`, созданного сценарием `Capture_Stats`:

```

$ cat capstats.csv |
> gawk -F, '{printf "%s %s - %s\n", $1, $2, $4}'
12/09/2010 9:06:50 - 0.29
12/09/2010 9:07:55 - 0.28
12/09/2010 9:40:51 - 0.37
12/10/2010 14:36:46 - 0.30
12/12/2010 7:16:26 - 0.25
12/12/2010 13:28:53 - 0.42

```

В команде `gawk` необходимо применить опцию `-F` для определения запятой как символа разделения полей в данных. После этого можно извлекать каждое отдельное поле данных и отображать его в требуемом виде с помощью команды `printf`.

Для создания отчета будет использоваться формат HTML, который уже в течение многих лет является стандартным для подготовки веб-страниц. В нем используются простые теги для обозначения типов данных, представленных на веб-странице. Тем не менее код HTML применяется не только в веб-страницах, часто он также используется в сообщениях электронной почты. Но не все программы электронной почты (см. главу 25) форматируют вывод на экран документа электронной почты с внедренными тегами HTML. Поэтому лучшее решение состоит в создании отчета в коде HTML и присоединении его к сообщению электронной почты.

Создание отчета в коде HTML позволяет получить качественно отформатированный отчет с минимальными трудозатратами. Затем отчет может быть отправлен на просмотр в программу, которая отобразит отчет, выполнив всю сложную работу по его форматированию и выводу на экран. Мы обязаны лишь вставить соответствующие теги HTML для форматирования данных.

Проще всего вывести данные в стиле электронной таблицы с помощью кода HTML, применив тег `<table>`, позволяющий создать таблицу со строками и ячейками, которые рассматриваются в языке HTML как средства представления данных. Начало строки обозначается с помощью тега `<tr>`, а конец строки — с помощью тега `</tr>`. Аналогичным образом ячейки определяются с использованием тегов `<td>` и `</td>`.

Вся совокупность кода HTML, необходимого для создания таблицы, выглядит следующим образом:

```
<html>
<body>
<h2>Report title</h2>
<table border="1">
<tr>
  <td>Date</td><td>Time</td><td>Users</td>
  <td>Load</td><td>Free Memory</td><td>%CPU Idle</td>
</tr>
<tr>
  <td>12/09/2010</td><td>11:00:00</td><td>4</td>
  <td>0.26</td><td>57076</td><td>87</td>
</tr>
</table>
</body>
</html>
```

Каждая запись данных обозначена парой тегов — `<tr>...</tr>`. А для каждого поля данных применяется собственная пара тегов: `<td>...</td>`.

При отображении отчета HTML в браузере с помощью его программного обеспечения автоматически создается таблица (рис. 27.2).

Что касается рассматриваемого сценария, то достаточно лишь сформировать код HTML шапки таблицы с помощью команд `echo`, код HTML — для представления данных с помощью команды `gawk`, а затем закрыть таблицу, опять-таки с использованием команд `echo`.

Ниже приведен сценарий `Report_Stats`, который формирует отчет о производительности и отправляет его по электронной почте.

Report for 12/12/2010

Date	Time	Users	Load	Free Memory	%CPU Idle
12/09/2010	9:06:50	2	0.29	645988	99
12/09/2010	9:07:55	2	0.28	620252	100
12/09/2010	9:40:51	3	0.37	474740	100
12/10/2010	14:36:46	3	0.30	46640	98
12/12/2010	7:16:26	4	0.25	27308	98
12/12/2010	13:28:53	4	0.42	58832	100

Рис. 27.2. Данные, отображенные с помощью таблицы HTML

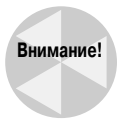
```
#!/bin/bash
#
# Report__Stats - создание отчета на основе полученных данных
# о производительности
#####
# Задание переменных сценария
#
REPORT_FILE=/home/user/Documents/capstats.csv
TEMP_FILE=/home/user/Documents/capstats.html
#
DATE=`date +%m/%d/%Y`
#
MAIL=`which mutt`
MAIL_TO=user
#
#####
# Сформировать заголовок отчета
#
echo "<html><body><h2>Report for $DATE</h2>" > $TEMP_FILE
echo "<table border=\"1\">" >> $TEMP_FILE
echo "<tr><td>Date</td><td>Time</td><td>Users</td>" >> $TEMP_FILE
echo "<td>Load</td><td>Free Memory</td><td>%CPU Idle</td></tr>" >>
$TEMP_FILE
#
#####
# Поместить в отчет статистические данные о производительности
#
cat $REPORT_FILE | gawk -F, '{
printf "<tr><td>%s</td><td>%s</td><td>%s</td>", $1, $2, $3;
printf "<td>%s</td><td>%s</td><td>%s</td>\n</tr>\n", $4, $5, $6;
```

```

}' >> $TEMP_FILE
#
echo "</table></body></html>" >> $TEMP_FILE
#
#####
# Отправить по электронной почте отчет с данными о производительности
# и выполнить очистку
#
$MAIL -a $TEMP_FILE -s "Performance Report $DATE"
-- $MAIL_TO < /dev/null
#
rm -f $TEMP_FILE
#

```

В данном случае для отправки по почте отчета о производительности применяется программа `mutt`, но читатель может внести изменения в сценарий, чтобы вместо этого использовался какой-то другой локальный почтовый клиент. Имя временного файла отчета HTML также можно изменить, если в этом есть необходимость.



Большинство почтовых клиентов позволяет автоматически распознавать тип вложения файла по его расширению. Поэтому необходимо приводить в конце имени файла отчета расширение `.html`.

Выполнение сценария

После создания сценария `Report_Stats` можно выполнить его тестовый прогон из командной строки и проследить за происходящим:

```

$
$ ./Report_Stats

```

Очевидно, что полученные результаты не представляют особого интереса. Действительная проверка состоит в просмотре полученного почтового сообщения, для чего лучше всего может подойти такой графический почтовый клиент, как KMail или Evolution. На рис. 27.3 показано, как выглядит это сообщение в окне почтового клиента Evolution. Заслуживает внимания то, что все данные качественно отформатированы с использованием средств таблицы HTML, по такому же принципу, как это происходит при просмотре в браузере!

Итак, выше были созданы сценарии формирования двух различных отчетов, позволяющие отслеживать показатели производительности системы и своевременно обнаруживать появление проблем, обусловленных нехваткой ресурсов и другими причинами. Тем не менее, каким бы качественным ни был текущий контроль системы, он не позволяет предотвратить возникновение непредвиденных проблем. Поэтому системный администратор должен иметь возможность быстро находить решения таких непредвиденных проблем и извлекать из этого полезный опыт, что позволит свести к минимуму отрицательное воздействие подобных ситуаций в будущем. В связи с этим в следующем разделе рассматривается создание усовершенствованных сценариев баз данных, позволяющих отслеживать возникающие проблемы.

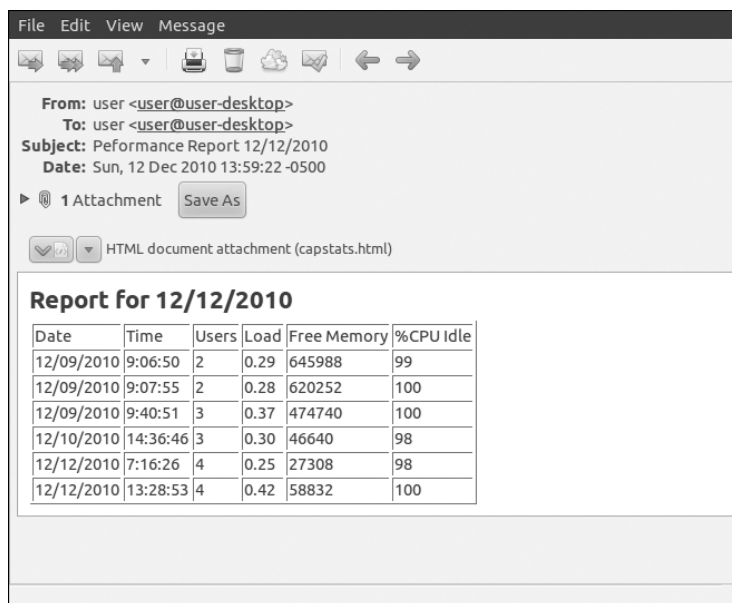


Рис. 27.3. Просмотр вложенного отчета в программе Evolution

Отслеживание проблем, связанных с базой данных

Нет ни одной системы, в которой не может произойти нечто непредвиденное. Тем не менее, анализируя возникающие проблемы, вы сможете внести изменения, которые позволят сократить до минимума вероятность непредвиденных нарушений в работе, которые могут возникнуть в будущем. Большую пользу приносит также документирование процесса устранения проблемы, на тот случай, если придется вновь выходить из подобного положения. Еще большую пользу можно извлечь из того, если каждый член коллектива документирует свой опыт в решении проблем. С информацией, представленной в формате “проблема–решение”, в дальнейшем смогут ознакомиться все члены группы.

В настоящем разделе рассматривается процесс подготовки к работе несложной базы данных отслеживания проблем и их устранения. Кроме того, по материалам этой главы можно написать несколько усовершенствованных сценариев, позволяющих регистрировать, обновлять и просматривать данные о проблемах и связанных с ними решениях.

Создание базы данных

Наиболее важной частью создания базы данных отслеживания проблем и решений является планирование структуры базы данных. Необходимо определить, какого рода информация должна отслеживаться и кто будет иметь доступ, позволяющий модифицировать и использовать данные, хранящиеся в базе данных.

Основное назначение этой базы данных — предоставление возможности регистрировать обнаруженную проблему и приводить данные о том, как было исправлено связанное с этим нарушение в работе, чтобы можно было использовать эти сведения для более быстрого поиска решений аналогичных проблем, которые могут возникнуть в дальнейшем. Для достижения этой цели необходимо отслеживать указанную ниже информацию.

- Дата получения сведений о возникновении проблемы.
- Дата устранения проблемы.
- Описание проблемы и ее признаков.
- Описание способа разрешения проблемы.

Для создания базы данных отслеживания проблем и решений будет использоваться система баз данных `mysql` (см. главу 23). Прежде всего создадим пустую базу данных:

```
$ $ mysql -u root -p
Enter password:
Welcome to the MySQL monitor. Commands end with ; or \g.
...
mysql> CREATE DATABASE Problem_Trek;
Query OK, 1 row affected (0.02 sec)

mysql>
```

Затем можно будет добавить в нее таблицы. В данном случае будет создана одна таблица, `problem_logger`, с четырьмя полями, содержащая сведения о проблемах. Кроме прочих полей, необходимо предусмотреть еще одно поле, `id_number`, которое должно служить в качестве первичного ключа для базы данных. В табл. 27.2 приведено описание пяти необходимых полей.

Таблица 27.2. Поля таблицы `Problem_Loader`

Имя поля	Тип данных	Имя поля	Тип данных
<code>id_number</code>	<code>integer</code>	<code>prob_symptoms</code>	<code>text</code>
<code>report_date</code>	<code>date</code>	<code>prob_solutions</code>	<code>text</code>
<code>fixed_date</code>	<code>date</code>		

Задача создания этой таблицы в базе данных `mysql` является довольно несложной:

```
mysql> USE Problem_Trek;
Database changed
mysql>
mysql> CREATE TABLE problem_logger (
-> id_number int not null,
-> report_date Date,
-> fixed_date Date,
-> prob_symptoms text,
-> prob_solutions text,
-> primary key (id_number));
Query OK, 0 rows affected (0.04 sec)

mysql>
```

Необходимо отметить, что прежде всего была выполнена команда `USE`, которая гарантирует, что новая таблица будет создана именно в той базе данных, в которой требуется. Проведем

дополнительную проверку результатов создания новой таблицы `problem_logger`, воспользовавшись командой `DESCRIBE`. В этом состоит удобный и понятный способ ознакомления с полями, из которых состоит таблица:

```
mysql> DESCRIBE problem_logger;
+-----+-----+-----+-----+-----+-----+
| Field          | Type   | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| id_number      | int(11)| NO   | PRI | NULL    |       |
| report_date    | date   | YES  |     | NULL    |       |
| fixed_date     | date   | YES  |     | NULL    |       |
| prob_symptoms  | text   | YES  |     | NULL    |       |
| prob_solutions | text   | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
5 rows in set (0.01 sec)

mysql>
```

До сих пор мы были вынуждены использовать учетную запись `root` для создания базы данных и таблицы. Но теперь, по завершении этих шагов, учетная запись `root` больше не требуется для дальнейшей работы над сценарием. Как было описано в главе 23, не рекомендуется постоянно использовать учетную запись `root`, работая со сценариями для баз данных. Поэтому необходимо добавить в базу данных новую учетную запись с соответствующими разрешениями, которая будет использоваться в сценариях для работы с базой данных `Problem_Trek`:

```
mysql> GRANT SELECT,INSERT,DELETE,UPDATE ON Problem_Trek.* TO
-> cbres IDENTIFIED BY 'test_password';
Query OK, 0 rows affected (0.03 sec)

mysql>
```

После добавления новой учетной записи пользователя к базе данных следует обязательно проверить, как она действует:

```
$ mysql Problem_Trek -u cbres -p
Enter password:
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Welcome to the MySQL monitor. Commands end with ; or \g.
...
mysql>
```

Новая учетная запись работает идеально! На данном этапе нам осталось сделать еще один шаг, после чего можно будет начать работу над необходимыми функциями для сценариев базы данных. Используя предпочтительный текстовый редактор, необходимо создать специальный файл конфигурации, `$HOME/.my.cnf`, содержащий пароль, который задан для созданной ранее новой учетной записи пользователя. Это позволяет предоставить сценариям доступ к программе `mysql`, не задавая пароль непосредственно в этих сценариях.

```
$ cat $HOME/.my.cnf
[client]
password=test_password
$
```

Кроме того, не следует забывать, что должны быть введены ограничения доступа для всех пользователей, кроме самого пользователя базы данных:

```
$ chmod 400 $HOME/.my.cnf
```

Итак, теперь создана база данных `Problem_Trek` и для нее предусмотрен соответствующий доступ. Мы можем приступать к работе над сценариями для базы данных.

Регистрация проблемы

Первый шаг в отслеживании проблемы состоит в регистрации ее симптомов. Для регистрации проблемы непосредственно с помощью команд `mysql` может потребоваться много времени, поэтому этот процесс будет упрощен путем включения необходимых команд в удобный сценарий.

Обязательные функции

Для ввода информации о проблеме в таблицу `problem_logger` можно воспользоваться командой `INSERT` программы `mysql` (см. главу 23). Для вставки новой записи в таблицу `problem_logger` применяются следующие команды:

```
$
$ mysql Problem_Trek -u cbres -p
Enter password:
...
mysql> INSERT INTO problem_logger VALUES (
-> 1012111322,
-> 20101211,
-> 0,
-> "When trying to run script Capture_Stats, getting message: bash:
./Capture_Stats: Permission denied",
-> "");
Query OK, 1 row affected (0.02 sec)

mysql>
```

Эта операция является довольно несложной. Для просмотра записи, которая была только что помещена в таблицу, можно вызвать команду `SELECT`. Чтобы просмотреть только одну запись, введенную непосредственно перед этим, можно указать идентификационный номер новой записи в предложении `WHERE`:

```
mysql> SELECT * FROM problem_logger WHERE id_number=1012111322;
+-----+-----+-----+-----+-----+
| id_number| report_date | fixed_date | prob_symptoms | prob_solutions|
+-----+-----+-----+-----+-----+
| 1012111322 | 2010-12-11 | 0000-00-00 | When trying to run script
Capture_Stats, getting message: bash:./Capture_Stats: Permission denied| |
+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql>
```

Очевидно, что введенная запись доступна для просмотра, но чтение данных в этом формате затруднительно. Чтобы упростить чтение этой записи, можно добавить в конце команды `\G` вместо точки с запятой `(;)`:


```
mysql> SELECT * FROM problem_logger WHERE id_number=1012111322\G
***** 1. row *****
id_number: 1012111322
report_date: 2010-12-11
fixed_date: 0000-00-00
prob_symptoms: When trying to run script Capture_Stats,
  getting message: bash:./Capture_Stats: Permission denied
prob_solutions:
1 row in set (0.00 sec)

mysql>
```

Полученный результат выглядит лучше и в большей степени применим для работы сценария. Две приведенные выше команды `mysql` воплощают в себе две основные функции, необходимые для создания сценария. Итак, можно приступить к формированию сценария `Record_Problem`.

Создание сценария

В главе 23 было показано, что для поиска команды `mysql` в конкретной системе следует использовать команду `which` и полученный с ее помощью результат присваивать переменной среды. В этом направлении можно пройти на шаг дальше и добавить имя базы данных и учетную запись пользователя, предназначенные для применения в программе `mysql`:

```
#
MYSQL='which mysql'" Problem_Trek -u cbres"
#
```

Сценарий `Record_Problem` является интерактивным, т.е. в нем предусмотрен вывод вопросов к пользователю для получения необходимой информации. После получения в сценарии ответов различным переменным присваиваются соответствующие значения. После сбора данных вызывается команда `INSERT` с приведенными в ней именами необходимых переменных для ввода информации в базу данных. Это позволяет построить сценарий таким образом, чтобы пользователь мог с ним работать, не задумываясь над тем, какие команды `mysql` фактически применяются для выполнения необходимых действий:

```
INSERT INTO problem_logger VALUES (
  $ID_NUMBER,
  $REPORT_DATE,
  $FIXED_DATE,
  "$PROB_SYMPTOMS",
  "$PROB_SOLUTIONS");
```

Переменная `ID_NUMBER` будет использоваться в качестве поля первичного ключа в таблице. Поскольку идентификационные номера должны составлять содержимое поля первичного ключа, каждый идентификационный номер проблемы должен быть уникальным и обозначать только определенную зарегистрированную проблему. Для пользователя сценария необходимо упростить задачу присваивания идентификационных номеров, поэтому должны использоваться автоматически сформированные идентификационные номера. Чтобы обеспечить уникальность идентификационных номеров, можно использовать сочетание значений текущей даты и времени для формирования требуемого ключа, состоящего из 10 цифр:

```
#
ID_NUMBER='date +%y%m%d%H%M'
#
```

На данном этапе создается сценарий, который служит лишь для регистрации проблемы, поэтому пока еще не нужно присваивать данные переменным `FIXED_DATE` (дата исправления) и `PROB_SOLUTIONS` (реализованные решения). Поэтому перед вставкой записи в таблицу необходимо предусмотреть присваивание двум этим переменным соответствующих значений `NULL`:

```
#
# Задать на время пустые значения параметров FIXED_DATE и
# PROB_SOLUTIONS
FIXED_DATE=0
PROB_SOLUTIONS=""
#
```

Чтобы можно было показать пользователю, что добавление записи с данными регистрации проблемы прошло успешно, в сценарии будет предусмотрена инструкция `SELECT`. Чтобы в результате выполнения этой инструкции не выводилось все содержимое базы данных, в предложении `WHERE` будет использоваться переменная `ID_NUMBER`, которая указывает только ту запись, которая была добавлена перед этим:

```
SELECT * FROM problem_logger WHERE id_number=$ID_NUMBER\G
```

После соединения всех этих фрагментов окончательный вариант сценария будет выглядеть следующим образом:

```
#!/bin/bash
#
# Record_Problem - регистрация проблем в системе в базе данных
#####
# Определить местоположение mysql и сохранить эти данные в переменной
#
MYSQL='which mysql'" Problem_Trek -u cbres"
#
#####
# Создать идентификатор записи и присвоить значение переменной
# REPORT_DATE
#
ID_NUMBER='date +%y%m%d%H%M\'
#
REPORT_DATE='date +%Y%m%d\'
#
#####
# Собрать информацию для ввода в таблицу
#
echo
echo -e "Briefly describe the problem & its symptoms: \c"
#
read ANSWER
PROB_SYMPTOMS=$ANSWER
#
# Задать на время пустые значения параметров FIXED_DATE и
# PROB_SOLUTIONS
FIXED_DATE=0
PROB_SOLUTIONS=""
#
```

```
#####
# Вставить полученную информацию в таблицу
#
#
echo
echo "Problem recorded as follows:"
echo
$MYSQL <<EOF
INSERT INTO problem_logger VALUES (
    $ID_NUMBER,
    $REPORT_DATE,
    $FIXED_DATE,
    "$PROB_SYMPTOMS",
    "$PROB_SOLUTIONS");
SELECT * FROM problem_logger WHERE id_number=$ID_NUMBER\G
EOF
#
#####
```

После этого проверим сценарий `Record_Problem`, чтобы определить, готов ли он к эксплуатации:

```
$ ./Record_Problem
```

```
Briefly describe the problem & its symptoms:
Running yum to install software and
getting 'not found' message.
```

```
Problem recorded as follows:
```

```
***** 1. row *****
id_number: 1012111510
report_date: 2010-12-11
fixed_date: 0000-00-00
prob_symptoms:
Running yum to install software and getting 'not found' message.
prob_solutions:
$
```

Сценарий действует превосходно! В частности, он позволяет убедиться в том, что для регистрации проблем гораздо проще использовать примерно такой сценарий, чем пытаться выполнить такую же работу с помощью отдельных команд `mysql`.

К этому времени мы ввели в действие способ регистрации проблем, а теперь нам нужен способ обновления записей с данными о проблемах. Перейдем к созданию сценария `Update_Problem`.

Обновление сведений о проблеме

Следующий шаг в отслеживании проблем — зарегистрировать их решение. Сценарий, который должен быть создан в этом разделе, предназначен для ввода дополнительных данных в запись, содержащую сведения о проблеме.

Обязательные функции

Задача обновления записей с использованием программы `mysql` решается очень просто. После получения всех необходимых данных достаточно воспользоваться командой команды `UPDATE` программы `mysql`:

```
UPDATE problem_logger SET
    prob_solutions="$PROB_SOLUTIONS",
    fixed_date=$FIXED_DATE
WHERE id_number=$ID_NUMBER;
```

Как вы могли убедиться, здесь нет ничего сложного. Однако для подготовки сценария `Update_Problem` недостаточно просто предусмотреть своевременный вызов команды `UPDATE`.

Создание сценария

Как и в предыдущем сценарии, необходимо позаботиться о том, чтобы максимально упростить работу для пользователя сценария. И для этого должны быть предусмотрены определенные дополнения.

В частности, требуется получение первичного ключа для базы данных `Problem_Trek`, или `id_number`, чтобы с его помощью идентифицировать обновляемую запись. После того как идентификационный номер станет известен, его можно передать в качестве параметра для сценария (см. главу 13). Для этого достаточно лишь проверить наличие параметра и присвоить параметр переменной `ID_NUMBER`. Если идентификационный номер не передан как параметр, то необходимо его запросить в сценарии:

```
if [ $# -eq 0 ] #Проверить, передан ли идентификационный номер
then
    #Если он не передан, запросить его
    #
    ...
    echo
    echo "What is the ID number for the"
    echo -e "problem you want to update?: \c"
    read ANSWER
    ID_NUMBER=$ANSWER
else
    ID_NUMBER=$1
fi
```

Тем не менее на запрос к вводу идентификационного номера проблемы пользователь вряд ли сможет найти ответ. Мало найдется таких пользователей, которые могут запоминать десятизначные номера, чтобы ссылаться с их помощью на конкретную проблему, которую они решают. Поэтому мы обязаны упростить работу для пользователя сценария путем вывода списка нерешенных проблем, зарегистрированных в базе данных.

Для этого необходимо прежде всего проверить, есть ли в базе данных записи, относящиеся к нерешенным проблемам. Признаком такой записи является то, что в ней не заполнены поля `fixed_date` и `prob_solutions`. Если хотя бы одно из этих полей пусто, то запись требует обновления. Чтобы провести поиск по обоим этим полям с помощью инструкции `SELECT`, можно применить оператор `OR` в предложении `WHERE`. Код, применяемый для выполнения этой задачи, выглядит примерно так:

```
RECORDS_EXIST=`$MYSQL -Bse 'SELECT id_number FROM problem_logger
WHERE fixed_date="0000-00-00" OR prob_solutions=""`
```

Заслуживает внимания то, что вывод применяемой команды присваивается переменной RECORDS_EXIST. Для этого необходимо использовать опции -Bse в команде mysql. Это позволяет выполнить команду в пакете (вместе со всеми прочими командами), а затем выйти. Если переменная RECORDS_EXIST содержит данные, это означает, что имеются записи, требующие обновления. Эти записи можно вывести на экран следующим образом:

```
#
    if [ "$RECORDS_EXIST" != "" ]
    then
    echo
    echo "The following record(s) need updating..."
    $MYSQL <<EOF
    SELECT id_number, report_date, prob_symptoms
        FROM problem_logger
        WHERE fixed_date="0000-00-00" OR
            prob_solutions=""\G
EOF
    fi
#
```

Итак, формируется удобный список записей с регистрацией проблем, работа над которыми еще не закончена, наряду с их идентификационными номерами. Пользователю остается лишь выбрать идентификационный номер из списка, а это намного проще, чем пытаться вспомнить применявшийся ранее номер!

Теперь можно соединить все фрагменты сценария обновления записей, который должен упростить выполнение этой функции. Этот сценарий будет выглядеть примерно так:

```
#!/bin/bash
#
# Update_Problem - обновление записи с проблемой в базе данных
#####
# Определить местоположение mysql и сохранить эти данные в переменной
#
MYSQL=`which mysql` Problem_Trek -u cbres"
#
#####
# Получить идентификатор записи
#
if [ $# -eq 0 ] #Проверить, передан ли идентификационный номер
then
    #Если он не передан, запросить его
#
# Проверить наличие незаполненных записей
RECORDS_EXIST=$MYSQL -Bse 'SELECT id_number FROM problem_logger
WHERE fixed_date="0000-00-00" OR prob_solutions=""'\
#
    if [ "$RECORDS_EXIST" != "" ]
    then
    echo
    echo "The following record(s) need updating..."
    $MYSQL <<EOF
    SELECT id_number, report_date, prob_symptoms
```

```

        FROM problem_logger
        WHERE fixed_date="0000-00-00" OR
              prob_solutions=""\G

EOF
    fi
#
    echo
    echo "What is the ID number for the"
    echo -e "problem you want to update?: \c"
    read ANSWER
    ID_NUMBER=$ANSWER
else
    ID_NUMBER=$1
fi
#
#####
# Получить дату решения (или значение переменной FIXED_DATE)
#
echo
echo -e "Was problem solved today? (y/n) \c"
read ANSWER
#
case $ANSWER in
y|Y|YES|yes|Yes|yEs|yeS|YEs|yES )
#
    FIXED_DATE=`date +%Y%m%d`
;;
*)
# Если дан любой ответ, кроме "yes", запросить ввод даты
    echo
    echo -e "What was the date of resolution? [YYYYMMDD] \c"
    read ANSWER
#
    FIXED_DATE=$ANSWER
;;
esac
#
#####
# Получить сведения о решении проблемы
#
echo
echo -e "Briefly describe the problem solution: \c"
#
read ANSWER
PROB_SOLUTIONS=$ANSWER
#
#####
# Обновить запись с данными о проблеме
#
#

```

```

echo
echo "Problem record updated as follows:"
echo
$MYSQL <<EOF
UPDATE problem_logger SET
    prob_solutions="$PROB_SOLUTIONS",
    fixed_date=$FIXED_DATE
    WHERE id_number=$ID_NUMBER;
SELECT * FROM problem_logger WHERE id_number=$ID_NUMBER\G
EOF
#
    Проверим данный сценарий, чтобы определить, успешно ли он функционирует.

$ ./Update_Problem

The following record(s) need updating...
***** 1. row *****
id_number: 1012111624
report_date: 2010-12-11
prob_symptoms: Running yum to install software and getting
'not found' message.

What is the ID number for the
problem you want to update?: 1012111624

Was problem solved today? (y/n) y

Briefly describe the problem solution: Network service was down.
    Issued the command: service network restart

Problem record updated as follows:

***** 1. row *****
id_number: 1012111624

report_date: 2010-12-11
fixed_date: 2010-12-11
prob_symptoms: Running yum to install software and getting
'not found' message.
prob_solutions: Network service was down.
Issued the command: service network restart
$

```

Очевидно, что сценарий работает очень хорошо и довольно прост в использовании.

Дальнейшее развитие сценариев

Теперь, чтобы еще больше упростить процесс работы со сведения о проблемах для пользователей сценария, внесем изменения в сценарий `Record_Problem`, предусмотрев в нем вызов сценария `Update_Problem` в случае необходимости. Дело в том, что иногда, в обстановке нехватки времени, проще зарегистрировать проблемы в “пакетном” режиме. В самом конце исходного сценария `Record_Problem` можно добавить следующий код:

```

# Проверить, желает ли пользователь сейчас ввести данные о решении
#

```

```

echo
echo -e "Do you have a solution yet? (y/n) \c"
read ANSWER
#
case $ANSWER in
y|Y|YES|yes|Yes|yEs|yeS|YES|yES )
    ./$HOME/scripts/Update_Problem $ID_NUMBER
#
;;
*)
# Если дан любой ответ, кроме "yes", просто выйти из сценария
;;
esac
#####

```

Теперь проверим этот вариант сценария, чтобы узнать, успешно ли он функционирует:

```

$ ./Record_Problem

Briefly describe the problem & its symptoms:
Moved script from Fedora to Ubuntu and was not working properly.

Problem recorded as follows:

***** 1. row *****
id_number: 1012111631
report_date: 2010-12-11
fixed_date: 0000-00-00
prob_symptoms:
Moved script from Fedora to Ubuntu and was not working properly.
prob_solutions:
Do you have a solution yet? (y/n) y

Was problem solved today? (y/n) y

Briefly describe the problem solution:
Added #!/bin/bash to the top of the script.
This is needed because Ubuntu's default shell is dash.

Problem record updated as follows:

***** 1. row *****
id_number: 1012111631
report_date: 2010-12-11
fixed_date: 2010-12-11
prob_symptoms:
Moved script from Fedora to Ubuntu and was not working properly.
prob_solutions: Added #!/bin/bash to the top of the script.
This is needed because Ubuntu's default shell is dash.
$

```

Сценарий работает просто идеально! Итак, нам удалось существенно упростить регистрацию и обновление записей в базе данных `Problem_Trek`.

Поиск проблемы

В полной мере удобство использования базы данных, подобной `Problem_Trek`, можно оценить, проведя с ее помощью обзор записей. Просмотр встретившихся ранее проблем и их решений позволяет упростить решение повторно возникающих аналогичных проблем и следить за тенденциями.

Обязательные функции

Чтобы найти в таблице записи, по которым можно сделать обзор, снова воспользуемся командой `SELECT`. Выше в этой главе такая команда применялась для другой цели — для просмотра вновь добавленных записей в таблице `problem_logger`:

```
SELECT * FROM problem_logger WHERE id_number=$ID_NUMBER\G
```

А при составлении сценария `Find_Problem` необходимо внести небольшое изменение в команду `SELECT`, которая использовалась ранее. Чтобы было проще находить записи с описанием проблем, встречавшихся ранее, предусмотрим вывод в сценарии запроса к вводу ключевого слова, которое характеризует искомые проблемы. Таким образом, если пользователь вспомнит, что в прошлом месяце уже сталкивался с проблемой, которая касается программы `yum`, то может ввести ключевое слово `"yum"`. Чтобы обеспечить поиск в базе данных записей, содержащих это ключевое слово, можно задать в инструкции `SELECT` сценария предложение `LIKE`. Применяемый при этом код может выглядеть так:

```
mysql> SELECT * FROM problem_logger
-> WHERE prob_symptoms LIKE 'yum'\G
Empty set (0.00 sec)
```

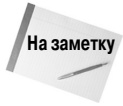
Но, к сожалению, этот код не действует должным образом! Дело в том, что мы не предусмотрели применение подстановочных знаков. Чтобы с помощью предложения `LIKE` можно было найти слово `"yum"`, глубоко спрятанное в поле `prob_symptoms`, следует добавить подстановочные знаки с обеих сторон от ключевого слова. Но, по существу, именно так программе `mysql` передается указание о том, что должна быть найдена запись, которая содержит слово `"yum"` в поле `prob_symptoms`. Применяемым при этом подстановочным знаком является символ процента (%). Итак, рассматриваемый код должен выглядеть следующим образом:

```
mysql> SELECT * FROM problem_logger
-> WHERE prob_symptoms LIKE '%yum%'\G
***** 1. row *****
id_number: 1012111624
report_date: 2010-12-11
fixed_date: 2010-12-11
prob_symptoms: Running yum to install software and getting
'not found' message.
prob_solutions: Network service was down.
Issued the command: service network restart
1 row in set (0.00 sec)
```

При этом желательно, чтобы команда `SELECT` выполняла поиск не только в поле `prob_solutions`, но и в поле `prob_symptoms`, поскольку это позволяет получить более полный список записей, представляющих интерес:

```
mysql> SELECT * FROM problem_logger WHERE
-> prob_symptoms LIKE '%yum%'
```

```
-> OR
-> prob_solutions LIKE '%yum%'\G
***** 1. row *****
id_number: 1012111624
...
```



Приступая к использованию этой операции поиска, можно не задумываться о проблемах чувствительности к регистру. По умолчанию в программе `mysql` регистр игнорируется, поэтому одинаково успешно будет выполняться поиск записей, содержащих ключевое слово "yum" или "YUM".

Но для повышения эффективности поиска в базе данных `Problem_Trek` следует предусмотреть применение в случае необходимости не одного ключевого слова, а нескольких. Но программа `mysql` не позволяет использовать предложение `LIKE` для задания нескольких искомых ключевых слов. Тем не менее остается возможность воспользоваться для этой цели предложением `REGEXP`, позволяющим задавать регулярные выражения. Одной из удобных особенностей предложения `REGEXP` является то, что оно обеспечивает поиск заданного ключевого слова во всем указанном поле, не требуя применения подстановочных знаков. Чтобы задать в качестве критерия поиска нескольких ключевых слов, достаточно поместить между ними обозначение логической инструкции ИЛИ, символ канала (`|`). Таким образом, применяемый код после внесения рассматриваемых изменений будет выглядеть следующим образом:

```
mysql> SELECT * FROM problem_logger WHERE
-> prob_symptoms REGEXP 'yum|dash'
-> OR
-> prob_solutions REGEXP 'yum|dash'\G
***** 1. row *****
id_number: 1012111624
report_date: 2010-12-11
fixed_date: 2010-12-11
prob_symptoms: Running yum to install software and getting
'not found' message.
prob_solutions: Network service was down.
Issued the command: service network restart
***** 2. row *****
id_number: 1012111631
report_date: 2010-12-11
fixed_date: 2010-12-11
prob_symptoms:
Moved script from Fedora to Ubuntu and was not working properly.
prob_solutions: Added #!/bin/bash to the top of the script.
This is needed because Ubuntu's default shell is dash.
2 rows in set (0.00 sec)
```

Итак, в нашем распоряжении теперь имеется самая важная функция, вокруг которой может быть построен сценарий `Find_Problem`. Приступим к подготовке кода сценария, соединив отдельные его фрагменты.

Создание сценария

Наилучший вариант состоит в том, чтобы ключевые слова передавались в качестве параметров при вызове сценария. Однако невозможно определить заранее, сколько ключевых слов

потребуется задать при том или ином вызове сценария. Это может быть пять ключевых слов или одно ключевое слово; но может произойти так, что сценарий будет вызван вообще без указания ключевых слов! Тем не менее задача вызова сценария с различным количеством ключевых слов легко поддается решению. Достаточно воспользоваться проверяемым условием `if` для определения того, произошла ли передача каких-либо ключевых слов. Если таковые имеются, то с помощью параметра `$@` можно перехватить сразу весь массив ключевых слов (см. главу 13). Если же передача пользователем ключевых слов не произошла, то в сценарии потребуется вывести запрос для их указания. Код, обеспечивающий выполнение этой задачи, может выглядеть так:

```
# Получить ключевые слова
#
if [ -n "$1" ]      #Проверить, введено ли ключевое слово
then               #Прочитать все заданные ключевые слова
#
    KEYWORDS=$@     #Прочитать все параметры как отдельные слова
                    # в одной и той же строке
#
else               #Ключевые слова не заданы. Запросить их ввод
    echo
    echo "What keywords would you like to search for?"
    echo -e "Please separate words by a space: \c"
    read ANSWER
    KEYWORDS=$ANSWER
fi
#
```

После получения искомых ключевых слов можно приступить к осуществлению следующего шага — формированию инструкции поиска. Напомним: между всеми подряд идущими ключевыми словами необходимо поместить логический оператор ИЛИ (символ канала `|`), поскольку без этого не может быть выполнен должным образом поиск с помощью предложения `REGEXP`. Такую задачу можно легко решить с помощью команды `sed`, в которой можно заменить знаком `|` каждый пробел между элементами массива `KEYWORDS`.

```
KEYWORDS='echo $KEYWORDS | sed 's/ /|/g'
```

Теперь сценарий выглядит таким образом:

```
#!/bin/bash
#
# Find_Problem - поиск записей с данными о проблеме с использованием
# ключевых слов
#####
# Определить местоположение mysql и сохранить эти данные в переменной
#
MYSQL='which mysql' Problem_Trek -u cbres"
#
#####
# Получить ключевые слова
#
if [ -n "$1" ]      #Проверить, введено ли ключевое слово
then               #Прочитать все заданные ключевые слова
#
```

```

KEYWORDS=$@      #Прочитать все параметры как отдельные слова
                  # в одной и той же строке
#
else              #Ключевые слова не заданы. Запросить их ввод
    echo
    echo "What keywords would you like to search for?"
    echo -e "Please separate words by a space: \c"
    read ANSWER
    KEYWORDS=$ANSWER
fi
#
#####
# Выполнить поиск записи с данными о проблеме
#
echo
echo "The following was found using keywords: $KEYWORDS"
echo
#
KEYWORDS='echo $KEYWORDS | sed 's/ /\|/g`'
#
$MYSQL <<EOF
SELECT * FROM problem_logger WHERE
    prob_symptoms REGEXP '($KEYWORDS)'
    OR
    prob_solutions REGEXP '($KEYWORDS)'\G
EOF
#

```

Безусловно, следующий шаг должен состоять в проверке нового сценария. На данный момент нас интересует, насколько успешно решена задача применения нескольких ключевых слов, поэтому вначале проверим сценарий в условиях передачи нескольких ключевых слов в командной строке:

```
$ ./Find_Problem yum dash
```

```
The following was found using keywords: yum dash
```

```

***** 1. row *****
id_number: 1012111624
...
prob_solutions: Added #!/bin/bash to the top of the script.
This is needed because Ubuntu's default shell is dash.

```

Затем проверим сценарий, не задавая ни одного ключевого слова в качестве параметра. В таком случае запрос на ввод ключевых слов должен поступить от сценария.

```
$ ./Find_Problem
```

```

What keywords would you like to search for?
Please separate words by a space: Ubuntu

```

```
The following was found using keywords: Ubuntu
```

```
***** 1. row *****
```

```
id_number: 1012111631
...
prob_solutions: Added #!/bin/bash to the top of the script.
This is needed because Ubuntu's default shell is dash.
```

Обе проверки прошли без каких-либо проблем. Итак, введен в действие удобный способ поиска записей с описанием проблем в базе данных `Problem_Trek`.

Имея в своем распоряжении все три описанных сценария, `Record_Problem`, `Update_Problem` и `Find_Problem`, можно приступить к созданию собственной базы данных отслеживания проблем. Автор выражает надежду, что время, затраченное читателем на изучение этих сценариев, окупится для него, когда настанет время накапливать сведения о том, какие проблемы перед ним возникали и как были решены.

Резюме

В этой главе приведен краткий обзор некоторых из наиболее сложных функций, которые могут быть реализованы с помощью сценариев командного интерпретатора. Пользователи часто упускают возможности, открывающиеся при использовании сценариев командного интерпретатора, для выполнения конкретных задач системного администрирования, сталкиваясь с тем, что для полноценной организации работы этих сценариев требуется применение весьма сложных функций.

В настоящей главе показано, как использовать поддерживаемые командным интерпретатором программы контроля производительности системы, такие как `uptime`, `df` и `free`, для сбора данных к моментальному снимку использования ресурса и производительности системы. Необходимая информация, полученная с помощью указанных команд, была представлена в наиболее удобном виде с применением команд `sed` и `gawk`, включая некоторые из наиболее сложных функций этих команд. После создания отчета на основании полученной информации была предусмотрена отправка этого отчета в указанную учетную запись пользователя через почтового клиента.

В следующем разделе рассматривалось создание более усовершенствованного сценария контроля производительности на основе программ `uptime` и `vmstat`. С помощью этих программ осуществлялся сбор данных и сохранение их в файле CSV для использования в других сценариях и (или) программах. В связи с этим был создан еще один сценарий, предназначенный для чтения данных о производительности из файла CSV и создания отчета в формате HTML. Само по себе применение формата HTML открывает возможность чтения отчета с помощью программы электронной почты или предпочтительного браузера.

В конце главы описано создание базы данных отслеживания проблем и усовершенствованных сценариев командного интерпретатора, предназначенных для заполнения и обновления базы данных, а также для создания отчетов на основе накопленных данных. В главе использовалось программное обеспечение базы данных `mysql` и в сценариях введены в действие такие команды, как `INSERT`, `UPDATE` и `SELECT`. Кроме того, рассматривались сложные вопросы, такие как поиск записей с помощью предложения `LIKE` в сравнении с поиском с применением предложения `REGEXP`, поддерживающего регулярные выражения.

Все сценарии, приведенные в главе, могут быть легко изменены в соответствии с требованиями, предъявляемыми в той или иной уникальной среде. Раскрывая перед читателем некоторые уникальные и усовершенствованные способы организации работы сценариев командного интерпретатора, авторы надеются, что это описание позволит понять, в чем состоят перспекти-

вы продуктивного использования существующих сценариев командного интерпретатора и создания новых.

Выражаем свою благодарность за то, что вы до конца прошли с нами сложный путь изучения и применения командной строки Linux и сценарной поддержки командного интерпретатора. Надеемся, что вам понравилось это путешествие и вы полностью освоили работу с командной строкой и способы создания сценариев командного интерпретатора, с помощью которых часто можно упростить свою работу. Но не останавливайтесь на этом в своем стремлении к изучению возможностей командной строки. В мире разработок с открытым исходным кодом постоянно появляются новинки, будь то новая программа для командной строки или полномасштабный командный интерпретатор. Поддерживайте связи с сообществом пользователей Linux и следите за новыми усовершенствованиями и достижениями.

Краткое руководство по командам bash

Как было показано в настоящей книге, командный интерпретатор `bash` предоставляет широкий набор средств и поэтому поддерживает большое количество команд. В данном приложении представлено краткое руководство, позволяющее быстро найти функцию или команду, которую можно использовать в командной строке `bash` или в сценарии командного интерпретатора `bash`.

Встроенные команды

Многие широко применяемые команды командного интерпретатора `bash` относятся к категории встроенных команд. Благодаря этому использование таких команд приводит к значительному сокращению затрат времени на выполнение. В табл. А.1 приведен перечень встроенных команд, которые могут быть вызваны непосредственно из командного интерпретатора `bash`.

Таблица А.1. Встроенные команды `bash`

Команда	Описание
<code>alias</code>	Определяет псевдоним для указанной команды
<code>bg</code>	Выполняет задание в фоновом режиме
<code>bind</code>	Связывает клавиатурную последовательность с функцией <code>readline</code> или макросом
<code>break</code>	Обеспечивает выход из цикла <code>for</code> , <code>while</code> , <code>select</code> или <code>until</code>
<code>builtin</code>	Выполняет указанную встроенную команду командного интерпретатора

ПРИЛОЖЕНИЕ



В этом приложении...

Встроенные команды

Команды `bash`

Переменные среды

Команда	Описание
<code>cd</code>	Обеспечивает переход из текущего каталога в указанный каталог
<code>caller</code>	Возвращает контекст любого активного вызова подпрограммы
<code>command</code>	Выполняет указанную команду без обычного поиска командного интерпретатора
<code>compgen</code>	Формирует возможные сопоставляемые завершения для заданного слова
<code>complete</code>	Показывает, как должны быть завершены указанные слова
<code>continue</code>	Обеспечивает переход к выполнению следующей итерации цикла <code>for</code> , <code>while</code> , <code>select</code> или <code>until</code>
<code>declare</code>	Объявляет переменную или тип переменной
<code>dirs</code>	Отображает список каталогов, хранящийся в настоящее время в памяти
<code>disown</code>	Удаляет указанные задания из таблицы заданий, относящейся к процессу
<code>echo</code>	Выводит указанную строку на устройство <code>STDOUT</code>
<code>enable</code>	Включает или отключает указанную встроенную команду командного интерпретатора
<code>eval</code>	Соединяет заданные аргументы в одну команду, а затем вызывает ее на выполнение
<code>exec</code>	Заменяет процесс командного интерпретатора указанной командой
<code>exit</code>	Вынуждает командный интерпретатор завершить свою работу с указанным статусом выхода
<code>export</code>	Задаёт указанные переменные как доступные для дочерних процессов командного интерпретатора
<code>fc</code>	Выбирает список команд из журнала выполненных команд
<code>fg</code>	Возобновляет выполнение задания в режиме переднего плана
<code>getopts</code>	Интерпретирует указанные позиционные параметры
<code>hash</code>	Находит и записывает в память полное имя пути для указанной команды
<code>help</code>	Отображает справочное меню
<code>history</code>	Отображает журнал выполненных команд
<code>jobs</code>	Формирует список активных заданий
<code>kill</code>	Отправляет сигнал системы процессу с указанным идентификатором процесса (process ID — PID)
<code>let</code>	Вычисляет каждый аргумент в математическом выражении
<code>local</code>	Создает в функции переменную с ограниченной областью определения
<code>logout</code>	Обеспечивает выход из интерактивного сеанса работы с командным интерпретатором
<code>popd</code>	Удаляет записи из стека каталогов
<code>printf</code>	Выводит текст с использованием строк форматирования
<code>pushd</code>	Добавляет каталог к стеку каталогов
<code>pwd</code>	Выводит на внешнее устройство имя пути текущего рабочего каталога
<code>read</code>	Считывает одну строку данных из устройства <code>STDIN</code> и присваивает полученное значение переменной
<code>readonly</code>	Считывает одну строку данных из устройства <code>STDIN</code> и присваивает полученное значение переменной, предназначенной только для чтения
<code>return</code>	Принудительно завершает работу функции со значением, которое может быть получено в вызывающем сценарии
<code>set</code>	Задаёт и отображает значения переменных среды и атрибутов командного интерпретатора
<code>shift</code>	Сдвигает позиционные параметры вниз на одну позицию
<code>shopt</code>	Вводит в действие или отменяет значения переменных, управляющих альтернативными функциями командного интерпретатора

Команда	Описание
<code>suspend</code>	Приостанавливает выполнение программы командного интерпретатора до получения сигнала SIGCONT
<code>test</code>	Возвращает статус выхода 0 или 1 с учетом указанного условия
<code>times</code>	Отображает накопленные данные о пользователе и системе
<code>trap</code>	Выполняет указанную команду после получения указанного сигнала системы
<code>type</code>	Показывает, как будет интерпретироваться заданное слово, будучи используемым в качестве команды
<code>ulimit</code>	Задаёт предел использования указанного ресурса для пользователей системы
<code>umask</code>	Задаёт разрешения по умолчанию для вновь созданных файлов и каталогов
<code>unalias</code>	Удаляет указанный псевдоним
<code>unset</code>	Удаляет указанную переменную среды или атрибут командного интерпретатора
<code>wait</code>	Ожидает завершения указанного задания и возвращает статус выхода

Встроенные команды обеспечивают более высокую производительность по сравнению с внешними командами, но по мере увеличения количества встроенных команд, реализованных непосредственно в командном интерпретаторе, увеличивается объем памяти, распределенной для команд, которые, возможно, так никогда и не потребуются. Командный интерпретатор `bash` поддерживает также внешние команды, применение которых способствует расширению функциональных возможностей командного интерпретатора. Эти команды рассматриваются в разделе “Команды `bash`”.

Команды `bash`

Кроме встроенных команд, в командном интерпретаторе `bash` используются внешние команды, предназначенные в основном для перехода по каталогам файловой системы, а также для управления файлами и каталогами. В табл. А.2 приведены основные внешние команды, с которыми чаще всего приходится сталкиваться при работе в командном интерпретаторе `bash`.

Таблица А.2. Внешние команды командного интерпретатора `bash`

Команда	Описание
<code>bzip2</code>	Использует алгоритм сжатия текста с блочной сортировкой Бэрроуза–Уилера (Burrows-Wheeler) и кодирование по алгоритму Хаффмена (Huffman)
<code>cat</code>	Выводит на внешнее устройство содержимое указанного файла
<code>chage</code>	Изменяет дату истечения срока действия пароля для указанной учетной записи пользователя системы
<code>chfn</code>	Изменяет содержимое комментария к указанной учетной записи пользователя
<code>chgrp</code>	Изменяет заданную по умолчанию группу для указанного файла или каталога
<code>chmod</code>	Изменяет права доступа в системе для указанного файла или каталога
<code>chown</code>	Устанавливает вместо заданного по умолчанию владельца указанного файла или каталога другого владельца
<code>chpasswd</code>	Считывает файл с заданными в виде отдельных пар регистрационными именами и паролями и обновляет пароли
<code>chsh</code>	Изменяет применяемый по умолчанию командный интерпретатор для указанной учетной записи пользователя

Команда	Описание
<code>compress</code>	Вызывает оригинальную версию программы сжатия файлов Unix
<code>cp</code>	Копирует указанные файлы в другое местоположение
<code>date</code>	Отображает даты в различных форматах
<code>df</code>	Отображает текущие статистические сведения о распределении пространства на диске для всех смонтированных устройств
<code>du</code>	Отображает статистические сведения об использовании дисковой памяти для указанного пути к файлу
<code>file</code>	Показывает тип файла для указанного файла
<code>find</code>	Выполняет рекурсивный поиск файлов
<code>finger</code>	Отображает информацию об учетных записях пользователей в локальной или удаленной системе Linux
<code>free</code>	Проверяет наличие доступной и используемой памяти в системе
<code>grep</code>	Обеспечивает поиск указанной текстовой строки в файле
<code>groupadd</code>	Создает новую системную группу
<code>groupmod</code>	Изменяет существующую системную группу
<code>gzip</code>	Обеспечивает сжатие по алгоритму Лемпеля–Зива (Lempel-Ziv) с помощью программы, разработанной в рамках проекта GNU
<code>head</code>	Отображает начальную часть содержимого указанного файла
<code>killall</code>	Отправляет сигнал системы выполняющемуся процессу с указанным именем процесса
<code>less</code>	Просматривает содержимое файла с применением дополнительных функций
<code>link</code>	Создает ссылку на файл с применением псевдонима
<code>ls</code>	Формирует листинг содержимого каталога
<code>mkdir</code>	Создает указанный подкаталог в текущем каталоге
<code>more</code>	Выводит на внешнее устройство содержимое указанного файла, приостанавливая вывод после заполнения каждого экрана с данными
<code>mount</code>	Отображает сведения о смонтированных дисковых устройствах или монтирует дисковые устройства в виртуальной файловой системе
<code>mv</code>	Переименовывает файл
<code>nice</code>	Выполняет команду с отличным от применяемого по умолчанию уровнем приоритета в системе
<code>passwd</code>	Изменяет пароль для учетной записи пользователя системы
<code>ps</code>	Отображает информацию о выполняемых процессах в системе
<code>pwd</code>	Отображает текущий каталог
<code>renice</code>	Изменяет приоритет приложения, выполняемого в системе
<code>rm</code>	Удаляет указанный файл
<code>rmdir</code>	Удаляет указанный каталог
<code>sort</code>	Изменяет организацию данных в файле данных на основе указанного порядка
<code>stat</code>	Обеспечивает просмотр статистических данных для указанного файла
<code>sudo</code>	Вызывает приложение на выполнение в учетной записи пользователя root
<code>tail</code>	Отображает конечную часть содержимого указанного файла
<code>tar</code>	Создает состоящий из одного файла архив с файлами и каталогами

Команда	Описание
touch	Создает новый пустой файл или обновляет отметку времени существующего файла
top	Отображает активные процессы наряду с наиболее важными статистическими данными системы
umount	Удаляет смонтированное дисковое устройство из виртуальной файловой системы
uptime	Отображает информацию о том, какова продолжительность непрерывной работы системы
useradd	Создает новую учетную запись пользователя системы
userdel	Удаляет существующую учетную запись пользователя системы
usermod	Изменяет существующую учетную запись пользователя системы
vmstat	Формирует подробный отчет об использовании памяти и процессора в системе
which	Определяет местоположение исполняемого файла
zip	Применяет версию программы PKZIP операционной системы Windows для Unix

Задавая в командной строке эти команды, можно решить почти любую необходимую задачу по управлению системой.

Переменные среды

Отличительной особенностью в работе командного интерпретатора `bash` является то, что он использует многие переменные среды. Безусловно, переменные среды сами по себе не имеют ничего общего с командами командного интерпретатора, но от их значений часто зависит то, как работают эти команды, поэтому важно знать, какие переменные среды применяются командным интерпретатором. В табл. А.3 перечислены переменные среды, применяемые по умолчанию в командном интерпретаторе `bash`.

Таблица А.3. Переменные среды командного интерпретатора `bash`

Переменная	Описание
<code>\$*</code>	Содержит все параметры командной строки в виде одного текстового значения
<code>\$@</code>	Содержит все параметры командной строки в виде отдельных текстовых значений
<code>\$#</code>	Показывает количество представленных параметров командной строки
<code>\$?</code>	Показывает статус выхода последнего по времени использованного процесса переднего плана
<code>\$-</code>	Отображает текущие флаги опций командной строки
<code>\$\$</code>	Отображает идентификатор процесса (process ID — PID) текущего командного интерпретатора
<code>\$_</code>	Отображает идентификатор процесса последнего по времени выполняемого фонового процесса
<code>\$0</code>	Показывает имя команды, вызванной на выполнение в командной строке
<code>\$_</code>	Показывает полное составное имя программы командного интерпретатора
<code>BASH</code>	Показывает полное имя файла, которое было задано для вызова командного интерпретатора
<code>BASH_ARGC</code>	Отображает количество параметров в текущей подпрограмме
<code>BASH_ARGV</code>	Представляет массив, содержащий все указанные параметры командной строки

Переменная	Описание
BASH_COMMAND	Представляет имя команды, выполняемой в настоящее время
BASH_ENV	Если задано значение переменной BASH_ENV, то в каждом сценарии bash предпринимается попытка выполнить файл запуска, определенный этой переменной, перед переходом к дальнейшей работе
BASH_EXECUTION_STRING	Представляет команду, используемую в опции командной строки -c
BASH_LINENO	Представляет массив, содержащий номера строк каждой команды в сценарии
BASH_REMATCH	Представляет массив, содержащий текстовые элементы, которые сопоставляются с указанным регулярным выражением
BASH_SOURCE	Представляет массив, содержащий имена файлов исходного кода для функций, объявленных в командном интерпретаторе
BASH_SUBSHELL	Показывает количество вторичных командных интерпретаторов, вызванных на выполнение текущим командным интерпретатором
BASH_VERSION	Отображает номер версии текущего экземпляра командного интерпретатора bash
BASH_VERSINFO	Представляет массив переменных, который содержит отдельные номера версий текущего экземпляра командного интерпретатора bash, старший номер и младший номер
COLUMNS	Задаёт параметр COLUMNS, который содержит значение ширины окна терминала, используемого для текущего экземпляра командного интерпретатора bash
COMP_CWORD	Определяет индекс в переменной COMP_WORDS, который содержит значение текущей позиции курсора
COMP_LINE	Показывает текущую командную строку
COMP_POINT	Содержит индекс текущей позиции курсора относительно начала текущей команды
COM_WORDBREAKS	Задаёт набор символов, используемых в качестве разделителей слов при выполнении операции дополнения слова
COMP_WORDS	Представляет массив переменных, содержащий отдельные слова из текущей командной строки
COMPREPLY	Представляет массив переменных, который содержит возможные коды завершения, сформированные функцией командного интерпретатора
DIRSTACK	Представляет массив переменных, который содержит текущее содержимое стека каталогов
EUID	Представляет действительный числовой идентификатор текущего пользователя
FCEDIT	Представляет заданный по умолчанию редактор, используемый командой fc
FIGIGNORE	Представляет разделенный двоеточиями список суффиксов, которые следует пропускать при выполнении операции завершения имени файла
FUNCNAME	Определяет имя выполняющейся в настоящее время функции командного интерпретатора
GLOBIGNORE	Представляет разделенный двоеточиями список шаблонов, определяющих набор имен файлов, которые следует пропускать при дополнении имени файла
GROUPS	Представляет массив переменных, содержащий список групп, членом которых является текущий пользователь
histchars	Представляет параметр, содержащий до трех символов, которые управляют развертыванием журнала

Переменная	Описание
HISTCMD	Представляет номер текущей команды в журнале
HISTCONTROL	Представляет параметр, управляющий тем, какие команды должны быть введены в список журнала командного интерпретатора
HISTFILE	Определяет имя файла, предназначенного для сохранения списка журнала командного интерпретатора (по умолчанию — <code>.bash_history</code>)
HISTFILESIZE	Задаёт максимальное количество строк, предназначенных для сохранения в файле журнала
HISTIGNORE	Представляет разделённый двоеточиями список шаблонов, используемый для принятия решения о том, какие команды пропускаются при включении в файл журнала
HISTSIZE	Задаёт максимальное количество команд, хранимых в файле журнала
HOSTFILE	Представляет параметр, содержащий имя файла, который должен быть считан при необходимости дополнения в командном интерпретаторе имени хоста в процессе его ввода
HOSTNAME	Задаёт имя текущего хоста
HOSTTYPE	Задаёт строку, описывающую компьютер, на котором работает командный интерпретатор <code>bash</code>
IGNOREEOF	Задаёт количество последовательных символов признака конца файла EOF, которые должны быть получены командным интерпретатором для завершения работы. Если этот параметр не задан, то по умолчанию применяется значение 1
INPUTRC	Задаёт имя файла инициализации Readline (значение по умолчанию — <code>.inputrc</code>)
LANG	Определяет категорию региональной установки для командного интерпретатора
LC_ALL	Переопределяет переменную <code>LANG</code> , которая задаёт категорию региональной установки
LC_COLLATE	Задаёт порядок сортировки, используемый при сортировке строковых значений
LC_CTYPE	Определяет интерпретацию символов, используемых в расширениях имен файлов и в шаблонах согласования
LC_MESSAGES	Определяет региональную установку, используемую при интерпретации строк в двойных кавычках, которым предшествует знак доллара
LC_NUMERIC	Определяет региональную установку, которая используется при форматировании чисел
LINENO	Представляет номер строки в сценарии, выполняющемся в настоящее время
LINES	Определяет количество строк, имеющихся на терминале
MACHTYPE	Представляет строку, определяющую тип системы в формате <code>cpu-company-system</code>
MAILCHECK	Представляет параметр, от которого зависит, с какой периодичностью (в секундах) командный интерпретатор должен проводить проверку на наличие новой почты (значение по умолчанию — 60)
OLDPWD	Представляет предыдущий рабочий каталог, который использовался в командном интерпретаторе
OPTERR	Если задано значение 1 параметра <code>OPTERR</code> , командный интерпретатор <code>bash</code> отображает сообщения об ошибках, сформированные командой <code>getopts</code>
OSTYPE	Представляет строку, определяющую операционную систему, в которой работает командный интерпретатор

Переменная	Описание
PIPESTATUS	Представляет массив переменных, содержащий список значений статуса выхода из процессов в данном процессе переднего плана
POSIPLY_CORRECT	Если этот параметр задан, командный интерпретатор <code>bash</code> начинает свою работу в режиме POSIX
PPID	Представляет идентификатор родительского процесса командного интерпретатора <code>bash</code>
PROMPT_COMMAND	Если этот параметр задан, он определяет команду, выполняемую перед отображением первичного приглашения к вводу информации
PS1	Представляет строку первичного приглашения командного интерпретатора
PS2	Представляет строку вторичного приглашения командного интерпретатора
PS3	Представляет приглашение к вводу информации, используемое для команды <code>select</code>
PS4	Представляет приглашение к вводу информации, отображаемое перед повтором командной строки, если используется параметр <code>bash -x</code>
PWD	Представляет текущий рабочий каталог
RANDOM	Представляет переменную, которая возвращает случайное число от 0 до 32767. Присваивание значения этой переменной равносильно заданию начального значения для генератора случайных чисел
REPLY	Представляет заданную по умолчанию переменную для команды <code>read</code>
SECONDS	Представляет продолжительность времени (в секундах), истекшего с момента запуска командного интерпретатора. Присваивание значения этому параметру приводит к переустановке таймера в это значение
SHELLOPTS	Представляет разделенный двоеточиями список включенных параметров командного интерпретатора <code>bash</code>
SHLVL	Представляет параметр, который указывает уровень командного интерпретатора, увеличивающийся на единицу после каждого запуска нового командного интерпретатора <code>bash</code>
TIMEFORMAT	Представляет формат, указывающий способ отображения значений времени в командном интерпретаторе
TMOUТ	Показывает значение, определяющее продолжительность времени (в секундах) ожидания ввода командами <code>select</code> и <code>read</code> . По умолчанию предусмотрено значение нуль, которое задает неопределенно долгое время ожидания
UID	Представляет реальный числовой идентификатор текущего пользователя

Для вывода значений переменных среды предназначена встроенная команда `set`. В разных дистрибутивах Linux обнаруживаются существенные различия между значениями переменных среды командного интерпретатора, задаваемых по умолчанию во время начальной загрузки.

Краткое руководство по программам sed и gawk

В сценариях командного интерпретатора часто приходится обрабатывать данные в определенной форме и при этом трудно обойтись без программы sed или gawk (а иногда и той и другой). В настоящем приложении приведен справочник по командам sed и gawk, которые могут найти применение при работе с данными в сценариях командного интерпретатора.

Редактор sed

Редактор sed позволяет манипулировать данными в потоке данных на основе команд, заданных в командной строке или сохраненных в текстовом файле с командами. Этот редактор считывает с устройства ввода одновременно по одной строке данных, согласовывает содержимое строки с данными, представленными в командах редактора, вносит в данные изменения в режиме потоковой обработки согласно указаниям в командах, а затем выводит вновь сформированные данные в поток STDOUT.

Начало работы с редактором sed

Команда sed имеет следующий формат:

```
sed options script file
```

ПРИЛОЖЕНИЕ



В этом приложении...

Редактор sed

Программа gawk

Опции *options*, для которых предусмотрены параметры, позволяют настраивать поведение команды *sed*. К ним относятся опции, приведенные в табл. Б.1.

Таблица Б.1. Опции команды *sed*

Опция	Описание
<code>-e script</code>	Добавить команды, указанные в сценарии <i>script</i> , к командам, выполняемым при обработке входных данных
<code>-f file</code>	Добавить команды, приведенные в файле <i>file</i> , к командам, выполняемым при обработке входных данных
<code>-n</code>	Не вырабатывать выходные данные после выполнения каждой команды, а ожидать команду <i>print</i>

Параметр *script* указывает отдельную команду, применяемую к потоковым данным. Если требуется больше одной команды, то следует воспользоваться либо опцией `-e` для задания команд в командной строке, либо опцией `-f` для определения их в отдельном файле.

Команды *sed*

Сценарий редактора *sed* содержит команды, выполняемые программой *sed* для каждой строки данных во входном потоке. В настоящем разделе описаны некоторые из наиболее распространенных команд *sed*, которые могут найти применение практически в любом сценарии.

Подстановка

Команда *s* (сокращение от *substitution*) применяется для подстановки одного фрагмента текста вместо другого во входном потоке и имеет следующий формат:

```
s/pattern/replacement/flags
```

где *pattern* обозначает текст, подлежащий замене, а *replacement* — новый текст, который должен быть вставлен редактором *sed* на его место.

Параметр *flags* управляет тем, как выполняется подстановка. Предусмотрены флаги подстановки четырех указанных ниже типов.

- Число, указывающее, какое вхождение шаблона *pattern* должно быть заменено.
- *g*. Указывает, что должны быть заменены все вхождения текста.
- *p*. Указывает, что содержимое исходной строки должно быть выведено на внешнее устройство.
- *w file*. Указывает, что результаты подстановки должны быть записаны в файл.

При подстановке первого типа можно указать, какое вхождение шаблона сопоставления должно быть заменено редактором *sed*. Например, чтобы заменить только второе вхождение шаблона, можно задать значение 2.

Адресование

По умолчанию команды, заданные для выполнения в редакторе *sed*, применяются ко всем строкам текстовых данных. Но иногда требуется применить команду только к конкретной строке или группе строк, для чего предназначена *построчная адресация*.

В редакторе *sed* предусмотрены две формы построчной адресации:

- задание диапазона номеров строк;

- применение текстового шаблона, позволяющего отфильтровывать только нужные строки.

В обеих формах для указания адреса используется одинаковый формат:

```
[address] command
```

Если используется числовая построчная адресация, то для ссылки на строки служат данные об их положении в текстовом потоке. Редактор `sed` присваивает первой строке в текстовом потоке номер один, затем последовательно наращивает номера строк после обработки каждого символа перевода на новую строку.

```
$ sed '2,3s/dog/cat/' data1
```

Предусмотрен еще один метод сокращенного обозначения строк, к которым применяется команда, но он немного сложнее. Редактор `sed` позволяет задавать текстовый шаблон, используемый для фильтрации строк, которые предназначены для обработки с помощью команды. Для этого применяется следующий формат:

```
/pattern/command
```

Параметр *pattern* (шаблон) должен быть задан между символами косой черты. Редактор `sed` будет применять команду только к строкам, в которых содержится указанный текстовый шаблон.

```
$ sed '/rich/s/bash/csh/' /etc/passwd
```

Этот фильтр находит строку, в которой содержится текст `rich`, и заменяет подстроку `bash` подстрокой `csh`.

Предусмотрена также возможность собрать в группу несколько команд для применения к строкам с конкретным адресом:

```
address {  
    command1  
    command2  
    command3}
```

Редактор `sed` обрабатывает с помощью заданных команд только те строки, которые согласуются с указанным адресом *address*. Редактор `sed` будет обрабатывать каждую команду, перечисленную в строках адреса:

```
$ sed '2{  
> s/fox/elephant/  
> s/dog/cat/  
> }' data1
```

В данном примере редактор `sed` применяет каждую из подстановок ко второй строке в файле данных.

Удаление строк

Действие, выполняемое командой `delete` (сокращенно `d`), в основном соответствует ее названию: она удаляет все текстовые строки, сопоставляемые с заданной схемой адресации. При использовании команды `delete` необходимо соблюдать осторожность, поскольку, если не будет задана схема адресации, произойдет удаление всех строк из потока:

```
$ sed 'd' data1
```

Очевидно, что команда `delete` становится наиболее удобной при использовании в сочетании с указанным адресом. С ее помощью можно удалять конкретные строки текста из потока данных либо по номеру строки:

```
$ sed '3d' data6
```

либо с указанием конкретного диапазона строк:

```
$ sed '2,3d' data6
```

В команде `delete` может применяться и средство сопоставления с шаблонами редактора `sed`:

```
$ sed '/number 1/d' data6
```

Из потока будут удалены только строки, сопоставляемые с указанным текстом.

Вставка и добавление текста

Как и следовало ожидать, редактор `sed`, подобно любому другому редактору, позволяет вставлять текстовые строки в поток данных и присоединять к концу потока данных дополнительные строки. Изучая различия между этими двумя действиями, необходимо учитывать следующее, чтобы избежать путаницы:

- команда `insert` (сокращенно `i`) добавляет символ перевода на новую строку **ПЕРЕД** указанной строкой;
- команда `append` (сокращенно `a`) добавляет символ перевода на новую строку **ПОСЛЕ** указанной строки.

Изучение формата этих двух команд может оказаться затруднительным в связи с возможной путаницей: задавая команду вставки или добавления, нельзя приводить саму команду и ее данные в одной командной строке. Строку, подлежащую вставке или добавлению, необходимо задавать отдельно, на отдельной строке. Для этого применяется следующий формат:

```
sed '[address]command\  
new line'
```

Текст, приведенный в строке `new line`, появляется в выходных данных редактора `sed` в том месте, которое указано в команде. Напомним, что при использовании команды `insert` текст появляется перед текстовым содержимым потока данных:

```
$ echo "testing" | sed 'i\  
> This is a test'  
This is a test  
testing  
$
```

А при использовании команды `append` добавляемая строка появляется после указанного текста в потоке данных:

```
$ echo "testing" | sed 'a\  
> This is a test'  
testing  
This is a test  
$
```

Это позволяет вставлять текст в конце обычного текста.

Внесение изменений в строки

Команда `change` позволяет изменить содержимое всей строки текста в потоке данных. Она имеет формат, аналогичный формату команд `insert` и `append`, поскольку в ней вновь вводимая строка также должна быть задана отдельно от остальной части команды `sed`:

```
$ sed '3c\  
> This is a changed line of text.' data6
```

Символ обратной косой черты используется для обозначения символа перевода на новую строку данных в сценарии.

Команда преобразования `transform`

Команда `transform` (сокращенно `y`) — это единственная команда редактора `sed`, действие которой распространяется на отдельные символы. В команде `transform` используется следующий формат:

```
[address]y/inchars/outchars/
```

Команда `transform` выполняет взаимно однозначное отображение значений символов `inchars` в значения символов `outchars`. Первый символ из `inchars` преобразуется в первый символ из `outchars`. Второй символ из `inchars` преобразуется во второй символ из `outchars`. Такое отображение указанных символов продолжается по всей длине заданных строк `inchars` и `outchars`. Если строки `inchars` и `outchars` не имеют одинаковую длину, то редактор `sed` формирует сообщение об ошибке.

Вывод строк

Команда `p` передает строку в вывод редактора `sed`, по аналогии с тем, для чего служит флаг `p` в команде подстановки. Чаще всего команда `print` (сокращенно `p`) применяется для вывода строк, которые содержат текст, сопоставленный с текстовым шаблоном:

```
$ sed -n '/number 3/p' data6  
This is line number 3.  
$
```

Команда `print` позволяет фильтровать только определенные строки данных из входного потока.

Запись в файл

Для записи строк в файл предназначена команда `w`. Она имеет следующий формат:

```
[address]w filename
```

Значение имени файла `filename` может быть задано с указанием относительного или полного имени пути, но в любом случае пользователь редактора `sed` должно иметь разрешение на запись в файл. Для обозначения адреса (параметр `address`) может применяться метод адресования любого типа, используемый в редакторе `sed`, будь то метод, основанный на указании единственного номера строки, текстового шаблона, диапазона номеров строк или текстовых шаблонов.

Ниже приведен пример, в котором в текстовый файл выводятся только первые две строки из потока данных:

```
$ sed '1,2w test' data6
```

Выходной файл `test` содержит только первые две строки из входного потока.

Чтение из файла

Выше уже было описано, как вставлять данные и добавлять текст к потоку данных из командной строки `sed`. Команда `read` (сокращенно `r`) позволяет вставлять данные, содержащиеся в отдельном файле, и имеет следующий формат:

```
[address]r filename
```

Параметр `filename` указывает абсолютное или относительное имя пути для файла, который содержит данные. При использовании команды `read` не предусмотрена возможность задавать диапазон адресов. Разрешается задавать только единственный номер строки или адрес в виде текстового шаблона. Редактор `sed` вставляет текст из файла после строки, указанной этим адресом.

```
$ sed '3r data' data2
```

Редактор `sed` вставляет весь объем текста из файла данных в файл `data2`, начиная со строки 3 файла `data2`.

Программа gawk

Программа `gawk` представляет собой версию GNU исходной программы `awk`, предусмотренной в Unix. Программа `gawk` позволяет поднять на новый уровень сложность решаемых задач по сравнению с редактором `sed`, поскольку в ней вместо простых команд редактора применяются целые сценарии на полноценном языке программирования. В настоящем разделе содержатся основные сведения о программе `gawk`, и он может служить как справочник по его функциям.

Формат команды gawk

Программа `gawk` имеет следующий основной формат:

```
gawk options program file
```

В табл. Б.2 приведены опции `options`, поддерживаемые программой `gawk`.

Таблица Б.2. Опции gawk

Опция	Описание
<code>-F fs</code>	Задать для файла разделитель, который служит для разграничения полей данных в строке
<code>-f file</code>	Указать имя файла, из которого должна быть считана программа
<code>-v var=value</code>	Определить переменную и значение по умолчанию для использования в программе <code>gawk</code>
<code>-mf N</code>	Задать максимальное количество полей, подлежащих обработке в файле данных
<code>-mr N</code>	Задать максимальный размер записи в файле данных
<code>-W keyword</code>	Указать режим совместимости или уровень предупреждения для программы <code>gawk</code> . Чтобы получить список всех имеющихся ключевых слов <code>keyword</code> , можно воспользоваться опцией <code>help</code>

Опции командной строки предоставляют простой способ настройки средств, применяемых в программе `gawk`.

Использование программы gawk

Для использования программы `gawk` можно вызвать ее на выполнение непосредственно из командной строки или из сценария командного интерпретатора. В настоящем разделе описано, как использовать программу `gawk` и подготавливать сценарии, предназначенные для обработки этой программой.

Чтение программного сценария из командной строки

Сценарий программы `gawk` заключен в открывающие и закрывающие фигурные скобки. Между этими двумя фигурными скобками должны быть помещены команды сценария. Поскольку предполагается, что в командной строке `gawk` весь сценарий представлен в виде одной текстовой строки, необходимо также заключить сценарий в одинарные кавычки. Ниже приведен пример простого сценария для программы `gawk`, заданного в командной строке.

```
$ gawk '{print $1}'
```

Этот сценарий отображает первое поле данных из каждой строки входного потока.

Использование в программном сценарии нескольких команд

Язык программирования был бы слишком маловыразительным, если с его помощью можно было задавать в программе только одну команду. Поэтому и в языке программирования `gawk` предусмотрена возможность комбинировать команды, создавая полноценную программу. Чтобы воспользоваться несколькими командами в программном сценарии, заданном в командной строке, достаточно разграничить команды точками с запятой:

```
$ echo "My name is Rich" | gawk '{ $4="Dave"; print $0 }'
My name is Dave
$
```

В этом сценарии выполняются две команды: вначале четвертое поле данных заменяется другим значением, затем отображается вся строка данных в потоке.

Чтение программы из файла

Как и в случае редактора `sed`, редактор `gawk` позволяет сохранять программы в файле и вызывать их из командной строки:

```
$ cat script2
{ print $5 "'s userid is " $1 }
$ gawk -F: -f script2 /etc/passwd
```

Программа `gawk` применяет все команды, заданные в файле, к данным из входного потока.

Выполнение сценариев перед обработкой данных

Программа `gawk` позволяет также указывать, когда должен быть выполнен программный сценарий. По умолчанию программа `gawk` считывает строку текста из входного потока, а затем выполняет программный сценарий применительно к данным в строке текста. Иногда может потребоваться выполнить некоторый сценарий перед обработкой данных, например, чтобы создать раздел с заголовком для отчета. Для этого можно воспользоваться ключевым словом `BEGIN`. Это вынуждает программу `gawk` выполнить программный сценарий, указанный после ключевого слова `BEGIN`, перед чтением данных:

```
$ gawk 'BEGIN {print "This is a test report"}'
This is a test report
$
```

В раздел BEGIN можно поместить команды gawk любого типа, но чаще всего применяются команды, которые присваивают переменным значения по умолчанию.

Выполнение сценариев после обработки данных

В отличие от ключевого слова BEGIN, ключевое слово END позволяет задать фрагмент программного сценария, выполняемого программой gawk после чтения данных:

```
$ gawk 'BEGIN {print "Hello World!"} {print $0} END {print
    "byebye"}'
Hello World!
This is a test
This is a test
This is another test.
This is another test.
byebye
$
```

Программа gawk сначала выполняет код в разделе BEGIN, затем обрабатывает все данные во входном потоке и, наконец, выполняет код в разделе END.

Переменные gawk

Программа gawk представляет собой намного большее, чем просто редактор; это — интерпретатор, создающий полноценную среду программирования. Широкие возможности gawk обусловлены также тем, что с этой программой связано много команд и функций. В настоящем разделе приведены основные сведения, с которыми необходимо ознакомиться, чтобы успешно разрабатывать программы для интерпретатора gawk.

Встроенные переменные

В программе gawk встроенные переменные используются для ссылки на конкретные средства, применяемые при работе с данными в этой программе. В настоящем разделе описаны встроенные переменные, доступные для использования в программах gawk, и показано, как с ними обращаться.

Программа gawk определяет данные как записи и поля данных. При этом *запись* рассматривается как строка данных (по умолчанию разграничиваемая символами обозначения конца строки), а под *полем данных* подразумевается отдельный элемент данных в строке (по умолчанию разграничиваемый пробельным символом, таким как пробел или знак табуляции).

В программе gawk переменные поля данных используются для ссылки на элементы данных в каждой записи. Описание этих переменных приведено в табл. Б.3.

Таблица Б.3. Переменные полей и записей данных программы gawk

Переменная	Описание
\$0	Вся запись данных
\$1	Первое поле данных в записи
\$2	Второе поле данных в записи

Переменная	Описание
\$n	Поле данных в записи, которое находится в позиции <i>n</i>
FIELDWIDTHS	Разделенный пробелами список чисел, определяющих точную ширину (в пробелах) каждого поля данных
FS	Символ разделителя полей ввода
RS	Символ разделителя записей ввода
OFS	Символ разделителя полей вывода
ORS	Символ разделителя записей вывода

Кроме переменных, которые задают значения разделителей полей и записей, в программе `gawk` предусмотрены некоторые другие встроенные переменные, позволяющие определить, что происходит с данными, и извлечь информацию из среды командного интерпретатора. В табл. Б.4 приведены дополнительные сведения о таких встроенных переменных `gawk`.

Таблица Б.4. Дополнительные сведения о встроенных переменных `gawk`

Переменная	Описание
ARGC	Количество представленных параметров командной строки
ARGIND	Индекс в <code>ARGV</code> текущего обрабатываемого файла
ARGV	Массив параметров командной строки
CONVFMT	Формат преобразования для чисел (см. инструкцию <code>printf</code>). Значением по умолчанию является <code>%.6 g</code>
ENVIRON	Ассоциативный массив текущих переменных среды командного интерпретатора и их значений
ERRNO	Зафиксированная в системе ошибка, если при чтении или закрытии входных файлов произошла ошибка
FILENAME	Имя файла данных, используемого для ввода данных в программу <code>gawk</code>
FNR	Текущий номер записи в файле данных
IGNORECASE	Если установлено ненулевое значение, выполнять с помощью <code>gawk</code> все строковые функции (включая регулярные выражения); игнорировать регистр символов
NF	Общее количество полей данных в файле данных
NR	Количество обработанных входных записей
OFMT	Формат вывода для отображения чисел. Значением по умолчанию является <code>%.6 g</code>
RLENGTH	Длина подстроки, сопоставленной в функции <code>match</code>
RSTART	Индекс начала подстроки, сопоставленной в функции <code>match</code>

Встроенные переменные можно использовать в любом месте в программном сценарии `gawk`, включая разделы `BEGIN` и `END`.

Присваивание значений переменным в сценариях

Значения переменным в программах `gawk` присваиваются так же, как в сценариях командного интерпретатора, с использованием *оператора присваивания*:

```
$ gawk '
> BEGIN{
> testing="This is a test"
```

```
> print testing
> }'
This is a test
$
```

После присваивания значения переменной можно использовать эту переменную в сценарии `gawk` каждый раз, когда потребуется ее значение.

Присваивание значений переменным в командной строке

Для присваивания значений переменным в целях применения в программе `gawk` может также использоваться командная строка `gawk`. Это позволяет задавать значения вне обычного кода, изменяя при этом значения динамически. Ниже приведен пример использования переменной командной строки в целях выбора для отображения конкретного поля данных в файле:

```
$ cat script1
BEGIN{FS=","}
{print $n}
$ gawk -f script1 n=2 data1
$ gawk -f script1 n=3 data1
```

Такая возможность может стать основой превосходных способов обработки данных с помощью таких сценариев командного интерпретатора, как сценарии `gawk`.

Средства программы `gawk`

Некоторые средства программы `gawk` особенно хорошо подходят для манипулирования данными, позволяя создавать сценарии `gawk`, с помощью которых можно осуществлять синтаксический анализ текстовых файлов почти любого типа, включая файлы журналов.

Регулярные выражения

Для задания фильтров, определяющих, к каким строкам данных в потоке данных должен применяться программный сценарий, может использоваться формат базовых регулярных выражений (Basic Regular Expression — BRE) или формат расширенных регулярных выражений (Extended Regular Expression — ERE).

Если используется регулярное выражение, то оно должно быть приведено перед левой фигурной скобкой управляемого им программного сценария:

```
$ gawk 'BEGIN{FS=","} /test/{print $1}' data1
This is a test
$
```

Оператор сопоставления

Оператор сопоставления позволяет ограничить применение регулярного выражения лишь конкретным полем данных в записях. В качестве оператора сопоставления применяется знак циркумфлекс (^). При этом необходимо задать оператор сопоставления наряду с переменной поля данных и регулярным выражением, применяемым для сопоставления:

```
$1 ~ /^data/
```

Это выражение служит для фильтрации записей, в которых первое поле данных начинается с текстовых данных.

Математические выражения

В шаблоне сопоставления могут применяться не только регулярные, но и математические выражения. Данное средство становится применимым, если возникает необходимость обеспечить сопоставление с числовыми значениями в полях данных. Например, предположим, что необходимо вывести сведения обо всех пользователях системы, принадлежащих к группе пользователей root (к группе с номером 0). Для этого может использоваться следующий сценарий:

```
$ gawk -F: '$4 == 0{print $1}' /etc/passwd
```

Этот сценарий отображает первое значение поля данных для всех строк, которые содержат значение 0 в четвертом поле данных.

Структурированные команды

Программа `gawk` поддерживает следующие структурированные команды:

Инструкция `if-then-else`:

```
if (condition) statement1; else statement2
```

Инструкция `while`:

```
while (condition)
{
    statements
}
```

Инструкция `do-while`:

```
do {
    statements
} while (condition)
```

Инструкция `for`:

```
for(variableassignment; condition; iteration process)
```

Очевидно, что перед программистом, занимающимся разработкой сценариев `gawk`, открываются широкие возможности создания программ. Язык `gawk` позволяет конструировать такие программы, которые по своему функциональному назначению не уступают программам почти на любом языке высокого уровня.

Предметный указатель

A

ANSI
American National Standards
Institute 53
ASCII
American Standard Code for
Information Interchan-
ge 53

B

BRE
Basic Regular Expression 509
BSD
Berkeley Software Distribu-
tion 124

C

CLI
Command Line Interface 51
Cookie-файл 649
csh
C shell 604

D

dash
Debian ash 589
DEC
Digital Equipment Corpora-
tion 55
DNS
Domain Name System 671

E

EOF
End-of-File 485
ERE
Extended Regular Expres-
sion 509

G

GID
Group ID 177
GNOME
GNU Network Object Model
Environment 42
GNU 28

I

IMAP
Internet Message Access
Protocol 677
ISO
International Organization for
Standardization 53

J

JFS
Journaled File System 204

K

KDE
K Desktop Environment 41
ksh
Korn shell 604

L

LAMP
Linux-Apache-MySQL-PHP 619
Linus Torvalds 29
LiveCD 28
LV
Logical Volume 211
LVM
Logical Volume Manager 211

M

MDA
Mail Delivery Agent 671

MTA
Mail Transfer Agent 671
MUA
Mail User Agent 671

O

OSS
Open Source Software 38

P

PID
Process ID 35; 408
PMS
Package Management Sys-
tem 221
POP
Post Office Protocol 677
PV
Physical Volume 211

R

RPM
Red Hat Package Management
system 282

S

SGI
Silicon Graphics Incorpora-
ted 204
SGID
Set Group ID 195
SMTP
Simple Mail Transfer Proto-
col 671
SQL
Structured Query Language 624
SSH
Secure Shell 124
SUID
Set User ID 195

T
tar-файл 699

U
UCE
 Unsolicited Commercial
 E-mail 671
UID
 User ID 175; 603
URI
 Uniform Resource Identifier 263

V
VFS
 Virtual File System 37
VG
 Volume Group 211
vi
 improved vim 243

W
widget 460

X
XML
 Extensible Markup Language 646

Y
Yanking 250

Z
zle
 zsh line editor 603

A
Автономность 444
Агент
 доставки сообщений 671
 пересылки сообщений 671
 почтовый пользователя 671
Администратор
 системный 692

Адрес
 электронной почты 529; 669
Адресация
 построчная 492
 построчная числовая 493
Анимация 56
Апплет 42
Архивирование
 данных ежечасное 703
 данных за день 701
 файлов данных 698

Б
База
 данных terminfo 58
База данных 626
 MySQL 619
 postgres 632
 PostgreSQL 628
 template0 632
 template1 632
 реляционная 634
Безопасность
 данных 175
Библиотека 446
 KDE 43
 ncurses 676
 графическая текстовая curses 648
 функций 585
 функций 447
Бит
 SGID 195
 фиксации 190; 195
Блок 299
Браузер
 текстовый 647
Буфер
 обратной прокрутки 77
Буферизация
 данных 55

В
Веб-сайт
 с URL 661
 Lynx 648
 Mutt 688
 MySQL 620

PostgreSQL 628
Ведение
 журнала 55; 202
Вектор 55
Владелец
 файла 194
Вложенность
 циклов 339
Возврат
 каретки 54
 значения 436
 из функции 436
Возобновление
 выполнения 413
Восстановление
 файловой системы 210
Вставка
 данных 250
 текста 497
Выборка
 значений параметров 437
Вывод
 номеров строк 502
 ошибок 382
 последних строк 552
 форматированный 577
 из функции 436
Выгрузка 30
Вызов
 функции 431
 функции рекурсивный 444
Выполнение
 перехода 543
Выработка
 сигнала 402
Выражение
 группирования 525
 математическое 572
 регулярное 494; 507
 регулярное расширенное 521
 регулярное 143
 регулярное расширенное 144
Выход
 из цикла 343
 по тайм-ауту 374
 из сценария 292
Вычисление
 факториала 445

Вычисления

математические 580
математические 287

Г

График

резервного копирования 698

Графика векторная 55

Группа 186

пользователей 186
томов 211

Группирование команд 494

Д

Данные хранимые 618

Демон

atd 416
команды at 416

Дескриптор

файла 380
файла STDERR 382
файла STDIN 380
файла STDOUT 381
файла открытый 392
файла стандартный 380

Диапазон 518

адресов 554
символов 518

Диск

корневой 93

Диспетчер

логических томов 211

Дисплей

текстовый 52

Дистрибутив 28

Linux 45
LiveCD 48
основной 46
специализированный 47
программного обеспечения
Беркли 124

Добавление

пользователя 179
текста 497

Доставка

непосредственная 671
почты 671
через прокси-сервер 671

Ж

Журнал 201

ошибок 386
сообщений 398

З

Зависимость 222

нарушенная 235

Заголовок HTTP 649

Загрузка начальная 423

Задание 403

низкоприоритетное 408
остановленное 403
отложенное 419
переменной среды 155
пропущенное 422
шрифта 71
статуса выхода 435

Задача 128

Закладка 78

Заккрытие

дескриптора файла 391

Замена

символов 492
шаблона 545

Запись

учетная пользователя 175; 625
учетная системная 177

Запуск сценария 401

Знак

вопросительный 522
вставки 512
подстановочный 508
равенства двойной 597
табуляции 503
тильды 244

И

Идентификатор

группы 177; 186
группы установленный 195
пользователя 175; 603
пользователя установлен-
ный 195
процесса 35; 408
пользователя 275

Изменение

группы 188

пользователя 182

потока управления 542

приоритета 123

разрешений 192

Изображение

цветное 57

Имя

группы 186
регистрационное 176
сценария базовое 356
функции 433

Инициализация 605

Инструкция

echo 274
assign 593
do 324
done 324
do-while 576
for 569; 576
function 599
GRANT 627
if 298; 573
if-then 298
SELECT 637
SQL 622
while 574

Интервал 523

Интернет 647

Интерпретатор

командный ; 38
командный bash 87
командный Bourne 39
командный dash 588
командный zsh 601; 662
командный дочерний 157
командный интерактив-
ный 168

Интерфейс

клиентский MySQL 621
командной строки 51
командной строки lpx 649

Информация

о погоде 660

Использование

дисковой памяти 132; 723
файлов совместное 195
функции 431

К

- Кабель
 - связи 52
- Кавычки 273
 - двойные 273
 - одинарные 273
 - одинарные обратные 278
- Калькулятор 288
 - bash 288
 - bc 288
- Канал
 - связи 54
- Каталог 674
 - /etc/shadow 178
 - HOME 177
 - виртуальный 93
 - почтового ящика 674
 - текущий 164
- Клавиатура 57
- Клавиша
 - <Alt> 248
 - <Ctrl> 248
 - <Meta> 248
- Комбинация
 - клавиш <Ctrl+C> 402
 - клавиш <Ctrl+Z> 403
 - клавиш <Ctrl+D> 485
- Клавиша
 - блокировки прокрутки 57
 - ввода 57
 - возврата 57
 - останова прокрутки 57
 - перелистывания страниц 54
 - повторения 57
 - прерывания 57
 - со стрелкой 54; 58
 - удаления 57
 - функциональная 58
- Класс
 - символов 515
 - символов специальный 519
- Клиент 663
 - почтовый 669
- Ключ 567
- Код
 - разрешения 189
 - управляющий 54
 - статуса выхода 292
- Кодировка 53
- ANSI 53
- ASCII 53
- ISO-8859-1 53
- ISO-10646 54
- Latin-1 53
- Колесико
 - прокрутки мыши 56
- Команда
 - exit 294
 - expr 285
 - read 436
 - return 435
 - rpm 282
 - set 275
 - source 447
 - wc 281
 - alias 173
 - append 497
 - aptitude 223
 - at 416
 - atq 419
 - atrm 419
 - autoload 615
 - basename 355
 - batch 417
 - BEGIN 488
 - bg 413
 - branch 543
 - break 343; 458
 - bzcat 145
 - bzip2 145
 - case 301; 321; 454
 - cat 112
 - cd 95
 - chage 185
 - change 499
 - chfn 184
 - chgrp 194
 - chmod 192; 314
 - chown 194
 - chpasswd 184
 - chsh 184
 - clear 455
 - configure 240
 - continue 343
 - cp 109
 - date 91; 298
 - delete 495
 - DELETE 637
 - delete многострочная 536
 - delete однострочная 536
 - df 136; 723
 - dialog 460
 - dpkg 222
 - du 137; 693
 - egrep 144
 - emacs 591
 - export 594
 - fdisk 205
 - fg 414
 - fgrep 144
 - file 112
 - finger 184
 - for 323
 - free 725
 - function 599
 - getopt 366
 - getopts 369
 - grep 142; 726
 - groupadd 187
 - groupmod 188
 - gzip 146
 - head 117
 - history 589
 - infocmp 59
 - insert 497
 - INSERT 636
 - ipcs 32
 - jobs 411
 - kdialog 471
 - kill 131; 404
 - killall 132
 - kwrite 256
 - less 115
 - let 611
 - list 502
 - ls 97
 - lsf 392
 - lvcreate 216
 - lvdisplay 217
 - lynx 649
 - mail 685
 - make 240
 - man 91; 208
 - mkdir 196; 209
 - mktemp 395
 - more 114
 - mount 133; 209
 - mutt 689
 - mv 107
 - mysql 622

- next 533
- next многострочная 534
- next однострочная 533
- nice 414
- nohup 410
- passwd 183
- print 485
- printf 577
- print многострочная 537
- print однострочная 537
- ps ; 120; 33; 403; 725
- psql 631
- pvcreate 215
- pvdisplay 215
- read 372; 455; 504
- renice 415
- repeat 613
- rm 108
- rmdir 110
- rpm 222
- select 458
- SELECT 638
- set 153; 367; 602
- shift 361
- SHOW 624
- SHOW TABLES 627
- sleep 403
- sort 138; 694
- stat 111
- substitute 492
- substitution 503; 534
- sudo 227; 240
- tac 542
- tail 116
- tar 147; 698
- tee 398
- test 301; 544
- top 127
- touch 103; 190
- transform 500
- trap 404
- typeset 612
- umask 190
- umount 135
- unset 158
- until 338
- uptime 722
- USE 625
- useradd 179
- userdel 182
- usermod 187

- vgcreate 215
- vgdisplay 215
- vi 243
- vim 243
- vmstat 730
- which 726
- while 335
- zcompile 616
- zenity 474
- zftp 610
- zmodload 609
- zshmodules 612
- ztcp 663
- zupper 232
- встроенная 606
- встроенная dash 594
- замены vim 247
- командного интерпретатора 91
- многострочная 532
- поиска vim 246
- редактирования 245
- событий X 65
- структурированная 297; 572; 613
- структурированного программирования 572
- Комментарий 272
- Компоновка
 - меню 454
- Консоль
 - Linux 52; 61
 - виртуальная 62
- Конструкция
 - ORDER BY 638
 - WHERE 637
- Контроль
 - доступного дискового пространства 692
- Концепция OSS 38
- Копирование
 - в память 250
 - резервное 213; 698
 - файла 104
- Копия
 - зеркальная 214
- Корень 93
- Корпорация
 - SGI 204

Л

- Линейка
 - прокрутки 56
- Линус Торвальдс 29
- Листинг 695
- Ловушка 406

М

- Манипулирование
 - Манипулирование файлами 138
- Маркер
 - Маркер сворачивания 255
- Маска 191
 - Маска пользовательская 190
- Массив ; 171; 564
 - Массив ассоциативный 564
 - Массив переменных 171; 594
 - Массив с числовыми индексами 567
- Меню 453
 - К 42
 - сценария 454
 - текстовое 453
- Метакоманда 631
- Метод
 - POST 655
- Моделайн 267
- Модификатор
 - числовой 245
- Модификация
 - несанкционированная 175
- Модуль 294
 - TCP 663
 - zsh 609
 - загружаемый 601
 - командный 601
- Модульность 670
 - программного обеспечения 670
- Монтирование 132; 93
 - файла образа CD 135
- Мышь 52

Н

- Набор
 - разрешений 190

Написание
сценариев 242

Настройка
клавиатуры 67

Номер
задания 403; 408
индексного узла 200
телефона 527
устройства младший 36
устройства старший 36

Носитель
информации 237
информации сменный 132

Нумерация
строк в файле 551

О

Область
буферная 243
буферов 538
подкачки 30
прокрутки 56
просмотра 56
определения 439

Обозначение
восьмеричное 191
конца строки 503
символическое 193
переменной 276

Оболочка 548
сценария 548

Обработка
данных 480
многострочного текста 533
опций командной строки 362
параметров автоматическая 368
сигнала 401
сигналов 130
переменных 439

Обработчик
регулярных выражений 509
регулярных выражений базовых 509
регулярных выражений расширенных 509

Обращение 518
класса символов 518
команды 540

текстового файла 542

Объект
базы данных MySQL 625

Объявление
типа данных 612

Ограничение
данных 635
данных not null 635

Окно
прокручиваемое 464
регистрации 63
редактирования 243
редактирования KWrite 255
с вкладками 73
скользящее 552
терминального режима 249

Оператор
одинарной точки 272
if 573
логический 317
перенаправления 644
присваивания 566
сопоставления 571
точки 447; 449

Операция математическая 566

Определение функции
449 ; 584

Опция
инициализации 605
командной строки 352

Организация
модульная 670
цикла 545

Останов
процесса 130

Отображение
зеркальное 214
шрифта 71
сообщений 273

Отслеживание
проблем и решений 737
процессов 127
работы программ 119
файла 116

Отсчет
значений индексов 172

Отчет 722
с моментальным снимком 722

Очередь

заданий 417; 419

Очистка
экрана монитора 454

П

Пакет 221
coreutils 38
DEB 620
dialog 459
gdialog 474
Lynx 648
Postfix 680
RPM 620
zenity 474
программ 221
программ XFree86 40
программ X.org 40
эмуляции терминала 51; 52

Память
виртуальная 30
физическая 30

Панель 42
задач 42

Параметр 148
--forest 127
GNU длинный 126
в стиле BSD 124
в стиле Unix 121
безопасности 192
командной строки 352
команды lynx 650
опции 366
позиционный 353
прокси-сервера 658

Пароль 176

Перевод
строки 54

Передача
данных по каналу 282
параметров 437

Перезапуск
задания 413

Переименование
файла 107

Переключение перенаправления 386

Переменная 275
глобальная 439
локальная 439

- ARGC 563
- ARGV 563
- ENVIRON 564
- FIELDWIDTHS 561
- ORS 562
- RS 562
- USER_ACCOUNT_RECORD 710
- встроенная 559
- встроенная gawk 563
- глобальная 151
- локальная 151
- определяемая пользователем 559
- параметра специальная 357
- пользовательская 565
- поля данных 560
- специальная 357
- среды 150; 591
- среды Bourne 160
- среды fpath 614
- среды Linux 150
- среды Lynx 657
- среды OPTARG 369
- среды OPTIND 369
- среды PATH 163
- среды REPLY 374
- среды глобальная 151
- среды локальная 153
- среды пользовательская 151
- среды системная 151
- среды собственная 155
- с типом массива 568
- пользовательская 277
- системы 277
- среды 275
- среды PATH 272
- Перемещение файла 107
- Перенаправление 280; 381
 - ввода 280
 - ввода через буфер 281
 - вывода 280
 - временное 384
 - вывода sed 549
 - вывода цикла 349
 - дескриптора файла 388
 - постоянное 384
 - собственное 387
- Переопределение 433
- Переформатирование файловой системы 219
- Перехват
 - вывода 66
 - команды выхода 405
 - сигналов 404
- Переход
 - по каталогам 95
- Планирование запуска 416
- Погода 658
- Подавление
 - вывода команды 394
- Подстановка имен файлов 331
- Подсчет параметров 357
- Поиск
 - данных 142
 - инкрементный 251
 - неинкрементный 251
 - функций 614
- Поле 561
 - комментария 177
- Получение
 - данных от пользователей 352
- Поток
 - команд 542
 - управления 542
- Потребитель
 - дискового пространства 694
- Потребление
 - места на диске 693
- Почта
 - электронная 669
 - электронная рекламная незапрашиваемая 671
- Право
 - владения 194
 - доступа 634
 - доступа к файлам 175
 - доступа к файлу 188
- Прерывание
 - процесса 402
- Приглашение
 - вторичное 487
 - к вводу информации 89
 - к вводу пароля 622
- Применение
 - дополнительного экрана 55
 - сдвига 361
 - функции 430
- Принцип
 - \“клиент/сервер\” 665
- Приоритет
 - планирования 414
 - процесса 414
 - процесса 123
- Приостановка процесса 403
- Проверка
 - даты файла 316
 - каталогов 309
 - на наличие файла 311
 - номера телефона 527
 - параметров 356
 - права владения 315
 - существования объекта 310
- Программа 487
 - anacron 422
 - awk 484
 - bunzip2 145
 - bzcat 145
 - bzip2 144
 - bzip2recover 145
 - cron 420
 - cURL 661
 - gedit 262
 - GNU 29
 - gunzip 145
 - gzcat 145
 - gzip 145
 - LVM 212
 - Lynx 647
 - mail 674
 - Mailx 674; 684
 - MDA 672
 - MTA 671
 - MUA 674
 - Mutt 676; 687
 - mysql 621
 - PMS 222
 - Postfix 680
 - psql 631
 - resize2fs 219
 - sendmail 678
 - unzip 146
 - usermod 183
 - yum 232
 - zip 146

- zipcloak 146
- zipnote 146
- zipsplit 146
- клиента 666
- сервера 665
- с открытым исходным ко-
дом 38
- управления данными 559
- Программное обеспечение
прикладное 29
- Прокрутка 56
 - обратная 77
- Прокси-сервер 656
- Пропуск команд 346
- Прослушивание 124
- Просмотр
 - заданий 411
 - несанкционированный 175
 - файла 115
 - части файла 116
- Пространство
 - дисковое 693
 - обработки 534
 - хранения 538
 - шаблонов 534; 538
- Протокол 664
 - HTTP 655
 - SMTP 671
- Профиль 74
 - Shell 74
- Процесс ; 33
 - init 33; 423
 - init типа System V 424
 - init типа Upstart 425
 - дочерний 157
 - начальной загрузки 423
 - Процесс фоновый 408
- Процесс-зомби 725
- Псевдоним
 - команды 173
- Путь
 - абсолютный 95
 - относительный 96

П

- Работа
 - с файлами данных 138
- Раздел
 - Раздел основной 206

- Раздел первичный 207
- Раздел расширенный 206
- Разделение
 - опций и параметров 363
- Разделитель
 - полей внутренний 329
- Размещение
 - в последнем блоке 203
- Размонтирование 132; 135
- Распределение
 - блоков предварительное 203
 - памяти 724
 - полосовое 213
- Регистрация на сервере 639
- Редактирование данных
 - 245; 250
- Редактор
 - emacs 247; 591
 - gawk 484
 - GNOME 261
 - Kate 254
 - KDE 254
 - KWrite 255
 - sed 481
 - vi 242
 - vim 242; 243
 - построчный zsh 603
 - поточковый 481
 - текстовый 242; 481
 - текстовый интерактивный 481
- Режим
 - визуальный 246
 - вставки vim 244
 - графического рабочего
стола 52
 - командной строки 242; 244
 - консоли Linux 52
 - консоли графический 242
 - нормальный vim 244
 - однопользовательский 33
 - переднего плана 414
 - полноэкранный 243
 - работы системы 424
 - текстовый 52
 - фоновый 407
- Рекурсия 444
- Репозиторий 222
 - aptitude 229
 - yum 236
 - утвержденный 237

- Роль
 - группы 633
 - регистрации 633; 636
 - регистрации PostgreSQL 633

С

- Сбор
 - данных с экрана 658
 - статистических данных 730
- Сеанс
 - терминальный 409; 647
- Сегмент
 - общей памяти 32
- Сервер 663
 - MySQL 626
 - PostgreSQL 629
- Сжатие
 - данных 144
- Сигнал 130; 401
 - HUP 131
 - INT 131
 - KILL 131
 - процесса 130
 - процесса Linux 131
 - EXIT 405
 - Linux 401
 - SIGCONT 411
 - SIGHUP 402
 - SIGINT 402
 - SIGKILL 404
 - SIGTERM 404
- Символ
 - канала 282
 - передачи данных по кана-
лу 282
 - перенаправления 281
 - графический 55
 - канала 524
 - конца файла 485
 - одинарной точки 164
 - перенаправления 380; 381
 - пробельный 486
 - разделения полей 486
 - разделителя полей 561
 - специальный 511
 - точки 514
 - экранирующий 492
 - экранирующий 286
- Символ-шаблон 102

- Система
 - GNU/Linux 38
 - Postfix 680
 - X Window 40
 - безопасности файлов 175
 - ввода и вывода 379
 - доменных имен 671
 - инициализации 33
 - операционная Linux 28
 - управления пакетами 221
 - файловая 36; 199
 - файловая ext 200
 - файловая ext2 200
 - файловая ext3 202
 - файловая ext4 203
 - файловая JFS 204
 - файловая Linux 92
 - файловая ReiserFS 203
 - файловая XFS 204
 - файловая виртуальная 37
 - файловая расширенная 200
 - файловая с ведением журнала 201
 - управления пакетами 282
 - файловая iso9660 133
 - файловая vfat 133
- Скобка
 - квадратная 515
 - фигурная 494
- Скобки
 - квадратные 287
 - квадратные двойные 319
 - круглые двойные 318; 611
- Словарь 568
- Слово
 - ключевое function 431
 - ключевое local 441
 - ключевое BEGIN 488
 - ключевое END 489
 - ключевое function 584
 - ключевое primary key 635
- Снимок 722
 - моментальный 213; 722
 - моментальный каталога 698
- Событие 425
 - запуска 425
- Содержимое
 - файла 243
- Создание
 - библиотеки 446
- базы данных 626
- временного файла 395
- групп томов 215
- группы 187
- дескриптора файла 387
- журнала 209
- каталога 110
- логического тома 216
- раздела 205
- файла 103
- файловой системы 208
- функции меню 455
- функции 430; 431
- Сопоставление 521
 - многострочное 538
 - с шаблоном 320; 521
 - с шаблоном 142
- Сортировка 139
 - данных 138
 - лексикографическая 139
 - по месяцам 139
 - почты автоматическая 673
 - с учетом числовых значений 139
- Состояние процесса 35
- Сочетание
 - циклов until и while 341
- Спам 671
- Спецификатор
 - формата 577
- Способ
 - запуска bash 164
- Сравнение
 - строк 304
 - строк на равенство 305
 - файлов 309
 - чисел 302
- Среда
 - X Window 63
 - графического рабочего стола 29
 - командной строки 51
- Средство bash 596
- Ссылка
 - гибкая 106
 - жесткая 106
 - символическая 106
- Стандарт POSIX 509
- Стандартизация параметров 371
- Статус
 - выхода ; 292
 - выхода ненулевой 299
 - задания 408
 - кода завершения 461
 - файла 111
- Стол рабочий
 - рабочий GNOME 42
 - рабочий KDE 41
 - рабочий графический 40
- Страница 30
 - общей памяти 32
- Строка
 - данных 724
 - заголовка 724
 - командная bash 88
 - командная lpx 649
 - командная Mutt 688
 - командная mysql 621; 623
 - командная xterm 63
 - командная интерактивная 244
 - меню 75
 - пустая 514
 - сообщения 243
 - формата 577
- Структура
 - виртуального каталога 93
- Схема 633
 - public 633
- Сценарий
 - командного интерпретатора 270
 - рекурсивный 445
 - для системного администратора 692
 - запуска 424
 - интерактивный 453
 - командного интерпретатора 39
 - программный 487

Т

- Таблица
 - stop 416; 420
 - данных 625
 - индексных узлов 200
 - процессов 33
 - разделов 205
 - страниц памяти 30

Табуляция
 горизонтальная 54
 Тер
 форматирования 645
 Терминал
 DEC VT100 55
 DEC VT102 55
 GNOME 80
 Konsole 72
 Tektronix 4010 55
 xterm 63
 ввода-вывода 52
 Тип терминала 243
 Тире двойное 363
 Том
 логический 211
 физический 211
 Точка
 монтажирования 133
 монтажирования 93; 209
 привязки 512

У

Удаление
 задания 419
 каталога 110
 ловушки 406
 переменной среды 158
 пользователя 182
 пустых строк 553
 строк 495
 файла 108
 Узел
 индексный 101; 109; 200
 промежуточный 671
 Указатель 390
 внутренний 390
 Уничтожение
 задания 404
 процесса 711
 файла 108
 Упорядочение
 строк 305
 Управление
 данными 559
 заданиями 411
 логическими томами 219
 пакетами 221
 потоком данных 297

 сценарием 401
 учетными записями 705
 циклом 343
 Уровень
 безопасности 191
 Условие 302
 составное 317
 Установка
 MySQL 619
 из исходного кода 237
 локальная 233
 программного обеспече-
 ния 221

Ф

Файл
 zip 144
 библиотеки 446
 образа 134
 сценария 271
 .bashrc 88; 169
 /etc/group 186; 187
 /etc/passwd 88; 176
 /etc/profile 165
 inittab 424
 lynx.cfg 656
 nohup.out 410
 null 394
 /proc/meminfo 30
 README 239
 sendmail.cf 679
 shadow 178
 terminfo 58
 блочный 35
 временный 395
 данных 112
 запуска 164
 исполняемый 112
 конфигурации Lynx 656
 конфигурации master.cf 683
 конфигурации Postfix 682
 конфигурации sendmail 678
 почтового ящика 673
 профиля 167
 символьный 35
 скрытый 97
 специальный 36
 сценария 427
 текстовый 112; 480
 теневого 178
 устройства 35
 Факториал 445
 Фильтр 102; 494
 Фильтрация 509
 данных 509
 почты 673
 текста 509
 Фишинг 656
 Флаг
 подстановки 490
 Флеш-диск 49
 Флешка 49
 Флеш-накопитель
 USB 49; 133
 Формат
 времени 417
 даты 417
 команды getopt 366
 листинга 99
 приглашения 89
 Форматирование
 вывода 578
 раздела 208
 Формирование
 ссылки на файл 106
 Функция 430
 factorial 445
 автономная 444
 сценария 448
 int 725
 split 583
 strftime 584
 systemtime 584
 встроенная 580
 математическая 580
 побитовой обработки 581
 строковая 581
 ядра 30
 Функция-заглушка 455

Х

Хеш-массив 568

Ц

Цвет
 переднего плана 57
 текста 57
 фона 57

Цикл

while 458; 574
вложенный 339
внутренний 340

Чтение

данных 372
данных из файла 504
из файла 376
имени программы 355
параметров 353

Ш

Шаблон 142; 507
согласования 143
сопоставленный 143
BRE 509
регулярного выражения 508
сопоставления 572
текстовый 494

Э

Экран

дополнительный 56

Экранирование 511

специальных символов 511

Экстент 203

Элемент

графический 460
графический checklist 472
графический fselect 466
графический inputbox 463
графический menu 465
графический msgbox 461
графический radiolist 472
графический textbox 464
графический yesno 462

Эмуляция

клавиатуры 58
окна редактирования текста 242
терминала 51; 52

Ю

Юникод 54

Я

Ядро

Linux 29
виртуального каталога 93

Язык

программирования 484
программирования curl 661
программирования gawk 484

Ярлык

программы 42

Ящик почтовый 673