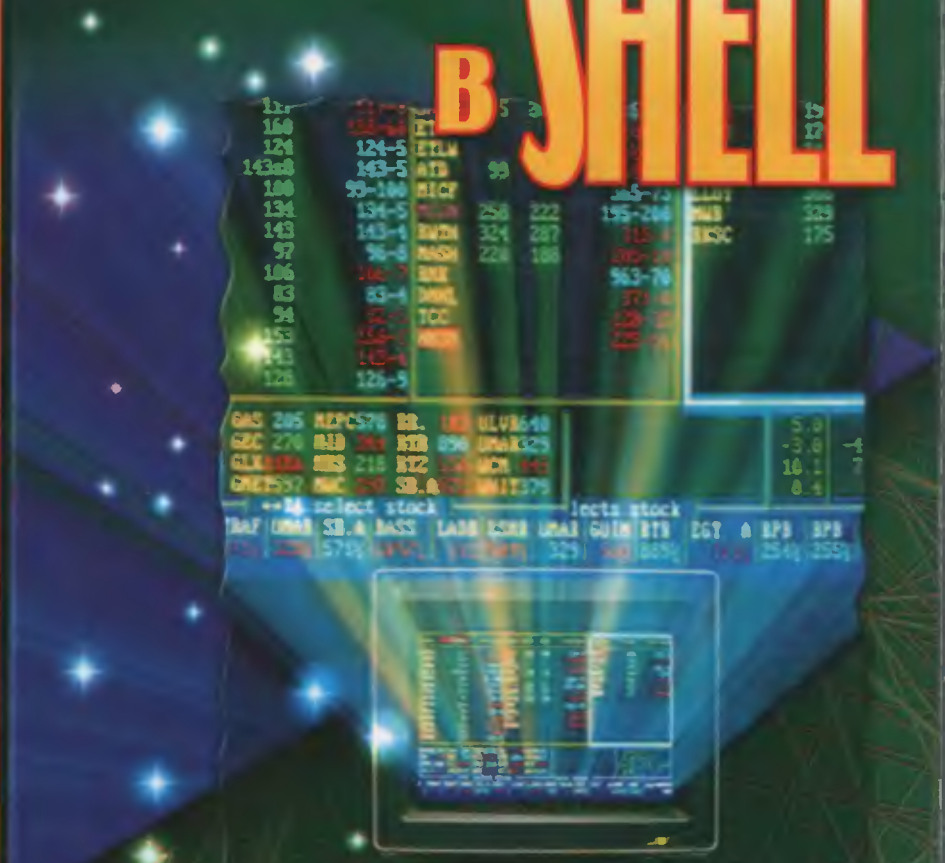
LINUX и UNIX: программирование в SHELL

Руководство разработчика

Дэвид Тейнсли



LINUX and UNIX Shell Programming

David Tansley

Addison-Wesley

An imprint of Pearson Education

Harlow, England • London • New York • Reading, Massachusetts • San Francisco • Toronto • Don Mills, Ontario • Sydney
Tokyo • Singapore • Hong Kong • Seoul • Taipei • Cape Town • Madrid • Mexico City • Amsterdam • Munich • Paris • Milan

Дэвид Тейнсли

Linux и UNIX: программирование в shell

Руководство разработчика

Перевод с английского

BHV, "Ирина", Киев
2001

УДК 681.3.06

Д. Тейнсли
Linux и UNIX: программирование в shell. Руководство разработчика: Пер. с англ. — К.: Издательская группа BHV, 2001. — 464 с.
ISBN 966-552-085-7
ISBN 5-7315-0114-9

Данная книга является практическим руководством по программированию интерпретатора Bourne shell — стандартного командного интерпретатора в UNIX, полностью совместимого с интерпретатором BASH shell в Linux. Книга предназначена для начинающих и опытных программистов и содержит множество полезных примеров, советов и подсказок. С ее помощью читатель сможет быстро научиться создавать shell-сценарии для реальных задач и ситуаций, возникающих в большинстве систем UNIX и Linux.

Обложка А.А.Стеценко

ISBN 0-201-67472-6
ISBN 966-552-085-7
ISBN 5-7315-0114-9

Введение

Настоящая книга посвящена shell-программированию, или, точнее, программированию в интерпретаторе Bourne shell.

Программирование на языке интерпретатора shell приобретает все большую популярность по мере утверждения Linux в качестве удобной в работе и отказоустойчивой операционной системы. Трудно оценить, какое количество пользователей работают с Linux. Эта операционная система распространяется бесплатно, хотя многие компании разрабатывают ее коммерческие варианты. Кроме того, несмотря на сделанные несколько лет назад неутешительные прогнозы специалистов относительно будущего UNIX, данная ОС также не теряет популярности, и число ее приверженцев продолжает расти.

Если вы хотите изучить shell-программирование, эта книга — для вас. Если же у вас имеется определенный опыт работы с интерпретатором shell, она послужит для вас хорошим справочным руководством. Кроме того, настоящее издание можно рассматривать как сборник советов по администрированию системы и кратких подсказок на все случаи жизни.

При написании книги во внимание принимались шесть основных положений:

- читатель должен быстро освоить инструментальные средства интерпретатора shell, а также изучить основы программирования на языке shell;
- книга должна служить не только руководством, но и справочником;
- для повышения производительности системы читателю следует научиться писать shell-сценарии;
- необходимо, чтобы shell-сценарии имели четкую и понятную структуру;
- главы, по возможности, должны быть самодостаточными и независимыми друг от друга, что упрощает их изучение;
- читатель должен уметь решать ряд административных задач, в частности создавать CGI-сценарии.

Обычно книги данной тематики имеют одну характерную особенность: некоторые примеры сценариев усложнены только ради того, чтобы занимать на несколько строчек меньше. Предлагаемая вашему вниманию книга свободна от подобных ограничений: все программные коды в ней просты, но эффективны.

Если вы приступили к изучению shell-программирования, очевидно, у вас есть на то веские причины. Основные причины изучения shell-программирования заключаются в следующем:

- язык программирования интерпретатора shell вполне самодостаточен, он содержит все необходимые управляющие конструкции и удобен в применении;
- открывается возможность быстрого создания сценариев;
- сценарии позволяют автоматизировать рутинные операции, выполняемые обычно вручную.

Интерпретатор Bourne shell

Bourne shell является стандартным интерпретатором команд, который входит в состав всех систем UNIX и совместим с интерпретатором bash в Linux. В книге, посвященной shell-программированию и не привязанной к конкретной операционной системе, в качестве общего знаменателя должен рассматриваться именно Bourne shell. Учтите, что существуют и другие интерпретаторы, такие как bash, Korn shell и C shell. Если в вашей системе установлен интерпретатор bash, сценарии из этой книги будут выполняться в нем, поскольку bash совместим с Bourne shell. Синтаксис интерпретатора Korn shell близок к синтаксису рассматриваемого здесь языка.

Если внимательно изучить сценарии системной инсталляции, то можно обнаружить, что более чем на 95 процентов они являются сценариями Bourne shell. Это объясняется тем, что создатели сценариев знали: они будут выполняться в любой системе UNIX и Linux.

Переносимость интерпретатора shell

Если необходимо, чтобы создаваемый сценарий выполнялся под управлением любой системы, он должен обладать свойством переносимости. Переносимость сценариев определяется двумя основными факторами:

- синтаксисом языка применяемого интерпретатора shell;
- используемыми командами интерпретатора.

Проблемы, связанные с первым фактором, решаются автоматически, если сценарии создаются для интерпретатора Bourne shell.

Что касается второго фактора, то для большинства shell-сценариев характерна следующая особенность: по крайней мере, 20 процентов (а то и больше) их рабочего времени затрачивается на выполнение таких shell-команд, как `cp`, `mv`, `mkdir` и др. Здесь и заключена проблема переносимости. Дело в том, что поставщики разных операционных систем оснащают эти команды различными наборами опций. В нашей книге используются сценарии общего характера и применяются только те опции и команды, которые присутствуют как в System V, так и в BSD. Если же имеются какие-либо различия, то приводятся альтернативные команды, но это случается сравнительно редко.

Структура книги

Книга представляет собой как руководство, так и справочник, поэтому ее главы можно читать в произвольном порядке. Если, например, вы хотите научиться включать в свои HTML-страницы CGI-сценарии, можете начать изучение материала даже с последней главы.

Книга состоит из пяти частей, посвященных различным аспектам программирования на языке интерпретатора shell.

Интерпретатор shell

В первой части книги рассматриваются общие вопросы, связанные с работой в интерпретаторе shell.

В главе 1 рассказывается о правах доступа к файлам и каталогам, а также о том, как создавать символические ссылки.

Чтобы на поиск созданного вами файла не уходило много времени, прочитайте главу 2, посвященную команде `find`.

Вполне вероятно, что вы захотите выполнять сценарии в ночное или вечернее время. В этом случае вам следует изучить главу 3, в которой изложены сведения о планировании выполнения сценариев.

Первая часть содержит также информацию о том, как работают механизмы ввода-вывода в интерпретаторе shell, как принимать данные с терминала и отображать на нем содержимое файлов, как перенаправлять потоки ввода-вывода и многое другое.

Фильтрация текста

Во второй части дается подробный обзор важнейших инструментов фильтрации текста. Фильтрация может выполняться в разное время: до того как данные поступят на вход сценария, в процессе выполнения сценария и при выводе текста на экран.

В отдельных главах описаны: `grep` — утилита поиска текстовых файлов (глава 8), `awk` — текстовый редактор, имеющий собственный язык сценариев (глава 9), `sed` — потоковый редактор, позволяющий выполнять быстрое редактирование (глава 10), и `tr` — утилита трансляции символов (глава 12). В главе 11 обсуждаются вопросы сортировки и объединения файлов, а также принципы работы с отдельными фрагментами текста.

Регистрация в системе

Третья часть посвящена вопросам регистрации в системе и настройки пользовательской среды. Каждый пользователь UNIX и Linux должен хорошо представлять, какие файлы выполняются в процессе регистрации.

В главе 14 подробно описаны механизмы управления локальными переменными интерпретатора shell и глобальными переменными среды.

В главе 15 даны рекомендации относительно того, как правильно употреблять различного рода кавычки и другие специальные символы интерпретатора shell, что позволит вам эффективно работать с системными переменными.

Основы shell-программирования

В четвертой части мы приступим к написанию сценариев интерпретатора shell. Здесь приведены указания по созданию исполняемых файлов и по их выполнению в среде интерпретатора shell, а также продемонстрированы принципы применения в сценариях условных, циклических и других управляющих конструкций.

Глава 19 содержит описания функций. Кроме того, в ней представлены способы их многократного вызова из различных сценариев.

Важным моментом является передача аргументов сценарию интерпретатора shell. В главе 20 показано, как обрабатывать аргументы, переданные сценарию.

В главе 21 вы узнаете, как форматировать текст при выводе на экран. Отдельная глава, 22, посвящена вопросам обновления файлов. В главе 23 сосредоточены основные сведения об отладке сценариев.

Завершает данную часть обзор встроенных shell-команд, которые до этого не рассматривались.

В пятой части состоится знакомство с современными методами создания сценариев.

В главе 26 рассказывается, как посылать сигналы различным процессам и как перехватывать их, а также описывается специфическая конструкция "документ здесь", которая позволяет вручную вводить данные, ожидаемые сценарием.

Вам известно, почему при запуске системы автоматически загружаются некоторые программы? Это не будет для вас тайной, если вы ознакомитесь с главой 28, в которой мы поговорим об уровнях выполнения сценариев.

Глава 27 включает небольшую коллекцию полезных сценариев. Один из них позволяет запрещать другим пользователям доступ к вашим файлам, не модифицируя файл */etc/passwd*.

Если вы интересуетесь вопросами разработки HTML-страниц, прочитайте главу 29, в которой рассматривается создание CGI-сценариев, но не с помощью языка Perl, а с помощью интерпретатора Bourne shell.

В книгу включены два приложения: приложение А, содержащее таблицу кодов ASCII, и приложение Б, в котором представлен ряд интересных shell-команд.

Что нужно знать

Желательно, чтобы читатель знал, как зарегистрироваться в системе, перейти в другой каталог и отредактировать файл в текстовом редакторе.

Чтобы запускать CGI-сценарии из главы 29, нужно располагать инсталлированным Web-сервером и иметь право выполнять CGI-сценарии.

Соглашения, принятые в книге

В книге употребляются следующие обозначения и шрифтовые выделения:

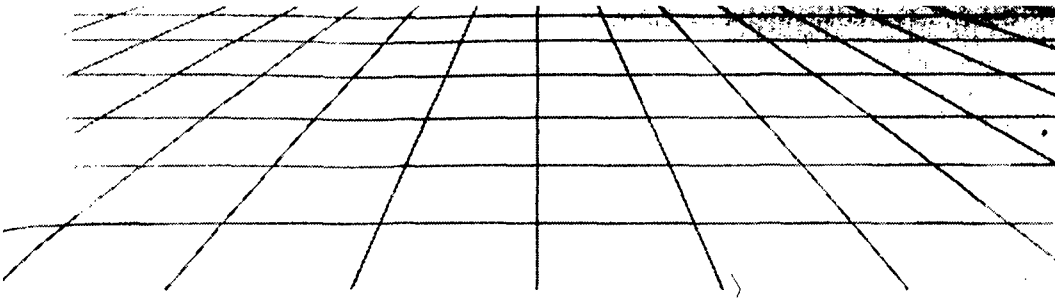
[Ctrl+клавиша]	Означает нажатие указанной клавиши одновременно с клавишей [Ctrl]; например, [Ctrl+O] — это указание одновременно нажать клавиши [Ctrl] и [O]
`Courier New`	Применяется во всех листингах сценариев, а также для обозначения результатов выполнения команд
`Courier New`	Используется для выделения командной строки

В первых двух частях книги можно встретить примечания наподобие следующего:

> **В Linux...**

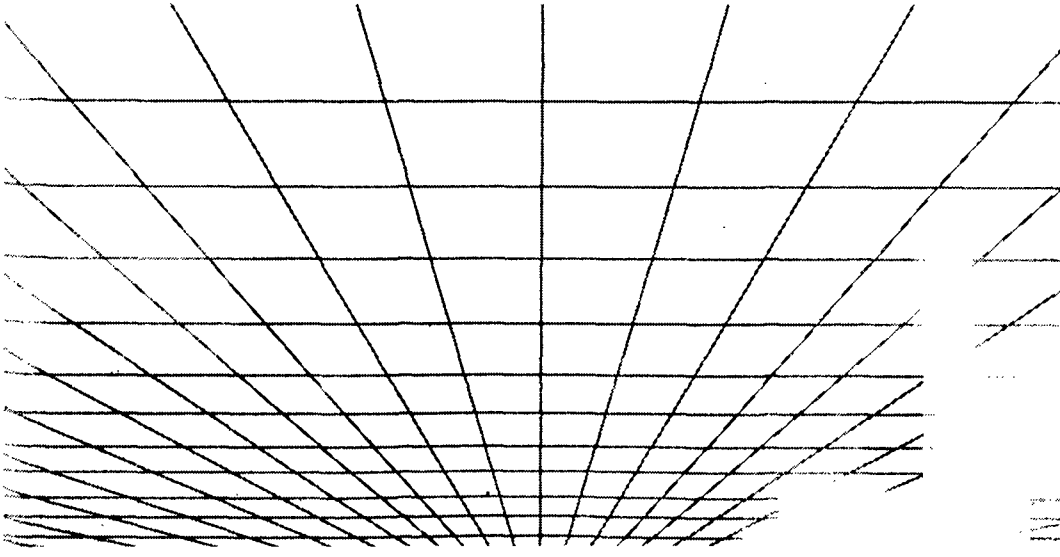
Они служат для того, чтобы кратко описать различия между синтаксисом рассматриваемой команды в BSD/Linux и System V.

Приводимые в книге сценарии протестированы в Linux (Red Hat) и в AIX. Некоторые сценарии протестированы в системе Data Generals.



ЧАСТЬ 1

Интерпретатор shell



ГЛАВА 1

Файлы и права доступа к ним

Если вы не хотите, чтобы кто угодно получал доступ к вашим файлам, изучите назначение битов режима. Благодаря им можно управлять доступом к файлам и каталогам, а также указывать тип доступа к создаваемым файлам. Это лишь небольшая часть системы безопасности в UNIX и Linux. Но на данный момент нас интересует не система безопасности в целом, а только та ее часть, которая имеет отношение к файлам и каталогам.

В этой главе рассматриваются следующие темы:

- права доступа к файлам к каталогам;
- биты смены идентификаторов (SUID и SGID);
- изменение владельца файла или каталога с помощью команд `chown` и `chgrp`;
- команда `umask`;
- символические ссылки.

Файл принадлежит создавшему его пользователю, а также группе, членом которой данный пользователь является. Владелец файла может самостоятельно определять, кому позволено производить запись в файл, читать его содержимое, а также запускать файл на выполнение, если он является исполняемым.

Примечание:

Пользователь `root` (системный администратор) может отменить практически все ограничения, заданные рядовым пользователем.

Доступ к созданному файлу может осуществляться тремя способами:

1. Путем чтения, при этом содержимое файла отображается на экране.
2. Путем записи, при этом содержимое файла редактируется или удаляется.
3. Путем выполнения, если файл содержит сценарий интерпретатора shell либо является программой.

Пользователи, имеющие доступ к файлу, делятся на три категории:

1. Владелец файла, создавший его.
2. Члены группы, к которой относится владелец файла.
3. Остальные пользователи.

1.1. Информация о файлах

После создания файла система сохраняет о нем всю информацию, которая может когда-либо понадобиться, в частности:

- раздел диска, где физически находится файл;

- тип файла;
- размер файла;
- идентификатор владельца файла, а также тех, кому разрешен доступ к файлу;
- индексный дескриптор;
- дата и время последнего изменения файла;
- режим доступа к файлу.

Рассмотрим типичный список файлов, полученный в результате выполнения команды `ls -l`.

```
$ ls -l
total 4232
-rwxr-xr-x   1 root     root         3756 Oct 14 04:44 dmesg
-r-xr-xr-x   1 root     root        12708 Oct  3 05:40 ps
-rwxr-xr-x   1 root     root         5388 Aug  5  1998 pwd
...
```

Информацию, предоставляемую командой `ls -l`, можно разбить на следующие части:

`total 4232`	Суммарный размер файлов в каталоге
`-rwxr-xr-x`	Режим доступа к файлу, отображаемый в виде строки из десяти символов. Первый символ ('-') указывает на то, что текущая запись относится к файлу (если на его месте стоит символ `d`, значит, запись относится к каталогу). Остальные символы делятся на три категории: `rwx` — права владельца (первая триада); `r-x` — права группы (вторая триада); `r-x` — права остальных пользователей (последняя триада). Символ `r` означает право чтения, символ `w` — право записи, символ `x` — право выполнения, символ '-' — отсутствие соответствующего права. Система разрешений подробно описана ниже
`1`	Количество жестких ссылок на файл
`root`	Идентификатор владельца файла
`root`	Идентификатор группы, в которую входит владелец файла
`3756`	Размер файла в байтах
`oct 14 04:44`	Дата и время последнего изменения файла
`dmesg`	Имя файла

1.2. Типы файлов

В начале строки режима может стоять не только символ '-' или `d`, ведь в каталоге насчитывается до семи различных типов записей (табл. 1.1):

Таблица 1.1. Типы файлов

`d`	Каталог
`l`	Символическая ссылка (указатель на другой файл)
`s`	Сокет

b	Специальный блочный файл
c	Специальный символьный файл
p	Файл именованного канала
-	Обычный файл или, если выразиться точнее, ни один из файлов, перечисленных выше

1.3. Права доступа к файлам

Давайте создадим файл, используя команду `touch`:

```
$ touch myfile
```

Теперь выполним команду `ls -l`:

```
$ ls -l myfile
-rw-r--r--    1 dave     admin           0 Feb 19 22:05 myfile
```

Мы получили пустой файл, и, как и ожидалось, первый символ в строке режима свидетельствует о том, что это обычный файл. В результате выполнения большинства операций по созданию файлов образуются либо обычные файлы, либо символические ссылки (о них говорится ниже).

Права владельца	Права группы	Права остальных пользователей
rw-	r--	r--

Следующие три символа в строке режима (`rw-`) описывают права доступа к созданному файлу со стороны его владельца (пользователь `dave`). За ними следуют символы `r--`, указывающие на права группы, в которую входит этот пользователь (в данном случае он является членом группы `admin`). Последние три символа (`r--`) представляют собой права пользователей, не принадлежащих к данной группе.

Существует три вида разрешений:

r	Право чтения данного файла
w	Право записи/изменения данного файла
x	Право выполнения данного файла, если он является сценарием или программой

Следовательно, строку режима для файла *myfile* необходимо интерпретировать следующим образом:

-	Rw-	r--	r--
Обычный файл	Владелец может осуществлять чтение и запись этого файла	Пользователи указанной группы могут осуществлять только чтение этого файла	Остальные пользователи также могут осуществлять только чтение этого файла

Возможно, вы обратили внимание на то, что при создании файла *myfile* владелец не получил право выполнять данный файл. Это связано с ограничениями, установленными по умолчанию в системе. Ситуация прояснится чуть позже, когда мы изучим команду `umask`.

Рассмотрим несколько дополнительных примеров (табл. 1.2).

Таблица 1.2. Примеры строк режима

Строка режима	Результат
r--------	Доступ к файлу разрешен только владельцу, который может читать содержимое файла, но не имеет права осуществлять запись в файл и выполнять его
r--r-----	Доступ к файлу возможен только для чтения и разрешен владельцу и всем пользователям группы, в которую он входит
r--r--r--	Любой пользователь может получить доступ к файлу для чтения, остальные действия запрещены
rwx------	Владелец имеет полный доступ к файлу, для остальных пользователей файл недоступен
rwxr-x---	Владелец имеет полный доступ к файлу; пользователи группы, в которую входит владелец, могут читать файл и запускать его на выполнение; для остальных пользователей файл недоступен
rwxr-xr-x	Владелец имеет полный доступ к файлу; остальные пользователи могут читать файл и запускать его на выполнение
rw-rw----	Владелец и пользователи группы, в которую он входит, могут осуществлять чтение и запись файла; для остальных пользователей файл недоступен
rw-rw-r--	Владелец и пользователи группы, в которую он входит, могут осуществлять чтение и запись файла; остальным пользователям разрешено только чтение файла
rw-rw-rw-	Все пользователи могут осуществлять чтение и запись файла

1.4. Изменение прав доступа к файлу

Вы можете изменять режим доступа к файлам, которыми владеете, с помощью команды `chmod`. Аргументы этой команды могут быть заданы либо в числовом виде (абсолютный режим), либо в символьном (символьный режим). Сначала рассмотрим символьный режим.

1.4.1. Символьный режим

Общий формат команды `chmod` для символьного режима таков:

```
chmod [кто] оператор [разрешения] файл
```

Значения параметра *кто*:

- `u` Владелец
- `g` Группа
- `o` Другие пользователи
- `a` Все (владелец, группа и другие пользователи)

Значения параметра *оператор*:

- `+` Добавление разрешения
- `-` Удаление разрешения
- `=` Установка заданного разрешения

Значения параметра *разрешения*:

- `r` Право чтения
- `w` Право записи
- `x` Право выполнения
- `X` Установка права выполнения только в том случае, если для какой-либо категории пользователей уже задано право выполнения
- `s` Установка бита SUID или SGID для владельца или группы
- `t` Установка sticky-бита*
- `u` Установка тех же прав, что и у владельца
- `g` Установка тех же прав, что и у группы
- `o` Установка тех же прав, что и у других пользователей

* Если символ `t` установлен для каталога, то это означает, что только владелец файлов, содержащихся в данном каталоге, может удалять их, даже если член группы имеет те же права, что и владелец файла.

Если символ `t` установлен для исполняемого файла (программы или сценария), то после завершения программы ее следует оставить на диске подкачки (в виртуальной памяти), чтобы ускорить последующий ее запуск другими пользователями. Поскольку в наши дни проблема оперативной памяти не стоит так остро, как раньше, в использовании sticky-бита при работе с файлами нет особой необходимости.

1.4.2. Примеры использования команды chmod

Рассмотрим несколько примеров изменения режима доступа к файлу с помощью команды `chmod`. Предполагается, что строка режима для нашего файла имеет такой вид: `rwxrwxrwx`.

Команда	Строка режима	Результат
`chmod a-x myfile`	`rw-rw-rw-`	Отмена всех разрешений на выполнение
`chmod og-w myfile`	`rw-r--r--`	Отмена разрешений на запись для группы и других пользователей
`chmod g+w myfile`	`rw-rw-r--`	Добавление разрешения на запись для группы
`chmod u+x myfile`	`rwxrw-r--`	Добавление разрешения на выполнение для владельца
`chmod go+x myfile`	`rwxrwxr-x`	Добавление разрешения на выполнение для группы и других пользователей
`chmod g=o myfile`	`rwxr-xr-x`	Предоставление группе тех прав, которые уже установлены для других пользователей

1.4.3. Абсолютный режим

Общий формат команды `chmod` для абсолютного режима таков:

`chmod [`*режим*`] `*файл*

Здесь параметр *режим* представляет собой восьмеричное число. В простейшем случае оно состоит из трех трехбитовых наборов, каждый из которых относится к соответствующей категории разрешений (владельца, группы, других пользователей). Старший бит

соответствует разрешению на чтение (1 — установлено, 0 — снято), средний — разрешению на запись, а младший — разрешению на выполнение. Рассмотрим примеры:

Таблица 1.3. Восьмеричные значения режима

Восьмеричное число	Результат
400	Владелец имеет право чтения
200	Владелец имеет право записи
100	Владелец имеет право выполнения
040	Группа имеет право чтения
020	Группа имеет право записи
010	Группа имеет право выполнения
004	Другие пользователи имеют право чтения
002	Другие пользователи имеют право записи
001	Другие пользователи имеют право выполнения

Чтобы получить итоговое значение режима, который вы хотите задать для своего файла, определите требуемый набор разрешений и сложите соответствующие числа в левой колонке таблицы.

Обратимся к примеру файла, который рассматривался ранее:

```
-rw-r--r--   1 dave     admin           0 Feb 19 22:05 myfile
```

Его строка режима эквивалентна числу 644, сформированного таким образом:

```
право чтения и записи для владельца    — 400 + 200 = 600
                                                   +
право чтения для группы                — 040       = 040
                                                     +
право чтения для других пользователей  — 004       = 004
                                                   = 644
```

Правило формирования восьмеричного режима доступа проще всего сформулировать с помощью следующей таблицы:

Таблица 1.4. Определение режима доступа к файлу

Владелец	Группа	Другие пользователи
r w x	r w x	r w x
4+2+1	4+2+1	4+2+1

1.4.4. Дополнительные примеры использования команды *chmod*

Ниже приведен ряд примеров, иллюстрирующих применение команды `chmod` в абсолютном режиме:

Команда	Строка режима	Результат
`chmod 666`	`rw-rw-rw-`	Установка разрешений на чтение и запись для владельца, группы и других пользователей
`chmod 644`	`rw-r--r--`	Установка разрешений на чтение и запись для владельца; группа и остальные пользователи получают право чтения

`chmod 744`	`rwxr--r--`	Предоставление полного доступа владельцу; группа и другие пользователи имеют право чтения
`chmod 664`	`rw--rw-r--`	Установка разрешений на чтение и запись для владельца и группы; другим пользователям предоставляется право чтения
`chmod 700`	`rwx------`	Предоставление полного доступа только владельцу; остальным пользователям доступ запрещен
`chmod 444`	`r--r--r--`	Все пользователи получают разрешение только на чтение

В качестве примера изменим права доступа к файлу *myfile*:

```
-rw-r--r--   1 dave      admin           0 Feb 19 22:05 myfile
```

Необходимо, чтобы владелец имел полный доступ к файлу, а пользователи группы — только разрешение на чтение. Для этого введите следующую команду:

```
$ chmod 740 myfile
$ ls -l myfile
-rwxr-----   1 dave      admin           0 Feb 19 22:05 myfile
```

Если другим пользователям также нужно дать разрешение на чтение, воспользуйтесь такой командой:

```
$ chmod 744 myfile
$ ls -l myfile
-rwxr--r--   1 dave      admin           0 Feb 19 22:05 myfile
```

Для изменения режима доступа ко всем файлам, находящимся в каталоге, предназначена команда, подобная приведенной ниже:

```
$ chmod 664 *
```

В результате выполнения этой команды владелец и группа получат разрешения на чтение и запись всех файлов текущего каталога, а другие пользователи — только право чтения файлов. Чтобы действие данной команды рекурсивно распространилось на все подкаталоги, воспользуйтесь опцией `-R`:

```
$ chmod -R 664 *
```

Следствием применения этой команды является рекурсивный обход всех подкаталогов, которые содержатся в текущем каталоге. При этом владелец и группа получают разрешение на чтение и запись, а другие пользователи — разрешение на чтение. Используйте опцию `-R` с осторожностью: убедитесь в том, что действительно требуется изменить разрешения для всех файлов из дерева подкаталогов.

1.5. Каталоги

Установка битов режима приобретает несколько иной смысл, когда команда `chmod` применяется по отношению к каталогу. Бит "чтения" означает возможность просмотра списка файлов в каталоге. Бит "записи" свидетельствует о том, что вам разрешается создавать и удалять файлы в данном каталоге. Наконец, бит "выполнения" указывает на возможность осуществления поиска файлов в каталоге и перехода в него.

Таблица 1.5. Права доступа к каталогу

r	w	x
Возможность просмотра списка файлов в каталоге	Возможность создания/удаления файлов в каталоге	Возможность поиска файлов в каталоге и перехода в него

Таблица 1.6. Примеры режимов доступа к каталогу

Строка режима	Владелец	Группа	Другие пользователи
`drwxrwxr-x` (775)	Чтение, запись, поиск	Чтение, запись, поиск	Чтение, поиск
`drwxr-xr--` (754)	Чтение, запись, поиск	Чтение, поиск	Чтение
`drwxr-xr-x` (755)	Чтение, запись, поиск	Чтение, поиск	Чтение, поиск

Когда строка режима для группы и других пользователей имеет значение `--x`, никто не может просматривать содержимое каталога, кроме его владельца. Если каталог содержит сценарий или программу с установленным битом выполнения, пользователи по-прежнему могут выполнять их, указывая точное имя файла. При этом не имеет значения, может ли пользователь перейти в данный каталог.

Разрешения, установленные для каталога, имеют более высокий приоритет, чем разрешения, установленные для файлов этого каталога. Например, если есть каталог *docs*:

```
drwxr--r--   1 louise   admin        2390 Jul 23 09:44 docs
```

а в нем — файл *pay*:

```
-rwxrwxrwx   1 louise   admin        5567 Oct 3 05:40 pay
```

пользователь, который является членом группы `admin` и собирается редактировать файл *pay*, не сможет этого сделать из-за отсутствия соответствующих прав доступа к каталогу. Этот файл доступен каждому, но поскольку бит поиска не установлен для группы `admin`, владеющей каталогом *docs*, ни один из пользователей группы не может получить доступ к каталогу. Если предпринимается попытка доступа, отображается сообщение "Permission denied" (доступ не разрешен).

1.6. Биты смены идентификаторов (SUID и SGID)

Биты SUID (Set User ID — установить идентификатор пользователя) и SGID (Set Group ID — установить идентификатор группы) были предметом жарких споров на протяжении многих лет. В некоторых системах установка этих битов не допускается либо они полностью игнорируются, даже если установлены. Это связано с тем, что при использовании данных битов возникает дополнительная угроза безопасности системы.

Идея, лежащая в основе применения бита SUID, состоит в том, что пользователь, запустивший программу, для которой владелец установил бит SUID, на время выполнения программы получает все права ее владельца. Если, например, администратор создал сценарий и установил для него бит SUID, а другой пользователь запускает этот сценарий, привилегии администратора на время выполнения сценария переходят к пользователю. Тот же принцип применим и к биту SGID, только в данном случае меняются привилегии группы, владеющей сценарием.

1.6.1. Для чего нужны биты SUID и SGID?

Для чего нужны сценарии, при запуске которых происходит смена идентификаторов? Сейчас я попытаюсь объяснить. Я отвечаю за администрирование нескольких больших баз данных. Чтобы выполнить операцию по их резервированию, требуется специальный профильный файл администратора. Я создал несколько сценариев и установил для них бит SGID, благодаря чему пользователи, которым разрешен запуск этих сценариев, не обязаны регистрироваться в качестве администраторов баз данных. А это, в свою очередь, уменьшает риск случайного повреждения информации на сервере. При запуске указанных сценариев пользователи получают разрешение на выполнение операций по выгрузке базы данных, хотя обычно такое право предоставляется только административному персоналу. После завершения сценариев восстанавливаются изначальные права пользователей.

Существует несколько системных команд UNIX, для которых установлен бит SUID или SGID. Чтобы найти эти команды, перейдите в каталог */bin* или */sbin* и введите:

```
$ ls -l | grep '^...s'
```

Вы получите список команд с установленным битом SUID.

```
$ ls -l | grep '^...s..s'
```

В результате выполнения этой команды выводится список команд, у которых установлен как бит SUID, так и бит SGID.

1.6.2. Установка битов SUID и SGID

Чтобы установить бит SUID, вставьте цифру 4 перед числом, задающим режим доступа. Биту SGID соответствует цифра 2. Если одновременно устанавливаются оба бита, следует ввести цифру 6 (4 + 2).

В строке режима установленные биты SUID и SGID обозначаются символом `s`, который помещается на место символа `x` в первую или вторую триаду соответственно. Помните, что сам бит выполнения (`x`) также должен быть установлен. Если, например, вы хотите для какой-либо программы установить бит SGID, убедитесь в том, что группа обладает правом выполнения этой программы.

Примечание:

Команда `chmod` не запрещает вам установить бит SUID или SGID при отсутствии соответствующего разрешения на выполнение файла. В этом случае при выполнении команды `ls -l` в строке режима будет указан символ `S`, а не `s`. Система таким образом информирует вас о неправильной установке прав доступа к файлу.

Рассмотрим несколько примеров:

Команда	Строка режима	Результат
`chmod 4755`	`rwsr-xr-x`	Для файла установлен бит SUID; владелец имеет право чтения, записи и выполнения; группа и другие пользователи имеют право чтения и выполнения
`chmod 6711`	`rws--s--x`	Для файла установлены биты SUID и SGID; владелец имеет право чтения, записи и выполнения; группа и другие пользователи имеют право выполнения

`chmod 2751`	`rwxr-s--x`	Для файла установлен бит SGID; владелец имеет право чтения, записи и выполнения; группа имеет право чтения и выполнения; другие пользователи имеют право выполнения

Для установки битов SUID и SGID можно также воспользоваться символьными операторами команды `chmod`. Вот как это делается:

```
chmod u+s имя_файла — SUID
chmod g+s имя_файла — SGID
```

Заметьте, что команда `chmod` не выполняет проверку корректности установок. Даже если для файла установлен бит выполнения, это еще не означает, что мы имеем дело с программой или сценарием.

1.7. Команды chown и chgrp

Создав файл, вы автоматически становитесь его владельцем, но можете передать право владения другому пользователю, у которого есть запись в файле */etc/passwd*. Только системный администратор либо фактический владелец может передавать права на файл другому пользователю. Если вы отказались от владения файлом, для того чтобы восстановить свои права на него, вам придется обратиться к системному администратору.

Для передачи прав владельца предназначена команда `chown`. Команда `chgrp` задает группу, которой принадлежит файл. Общий формат этих команд таков:

```
chown владелец файл
chgrp владелец файл
```

Опция `-R` позволяет выполнить рекурсивное изменение файлов в указанном каталоге и всех его подкаталогах.

1.7.1. Пример использования команды chown

Вот как можно поменять владельца файла с помощью команды `chown`:

```
$ ls -l
-rwxrwxrwx   1 louise   admin          345 Sep 20 14:33 project
$ chown pauline project
$ ls -l
-rwxrwxrwx   1 pauline  admin          345 Sep 20 14:33 project
```

Право владения файлом *project* переходит от пользователя `louise` к пользователю `pauline`.

1.7.2. Пример использования команды chgrp

Следующий пример демонстрирует, как поменять группу, которой принадлежит файл:

```
$ ls -l
-rwxrwxrwx   1 pauline  admin          345 Sep 20 14:33 project
$ chgrp sysadmin project
$ ls -l
-rwxrwxrwx   1 pauline  sysadmin       345 Sep 20 14:33 project
```

Пользователь pauline передал группе sysadmin право владения файлом *project*, которое до этого принадлежало группе admin.

1.7.3. Определение групп, в состав которых вы входите

Если вы хотите узнать, к какой группе принадлежите, введите команду groups:

```
$ groups
admin sysadmin appsgen general
```

либо воспользуйтесь командой id:

```
$ id
uid=0(root) gid=0(root) groups=0(root), 1(bin), 2(daemon), 3(sys), 4(ado)
```

1.7.4. Определение групп, в состав которых входят другие пользователи

Определить, в состав каких групп входит другой пользователь, позволяет команда groups, в качестве аргумента которой указано имя пользователя:

```
$ groups matty
sysadmin appsgen post
```

Данная команда сообщает о том, что пользователь matty входит в состав групп sysadmin, appsgen и post.

1.8. Команда umask

Когда вы регистрируетесь в системе, команда umask устанавливает стандартный режим доступа к создаваемым файлам и каталогам. Задайте с помощью этой команды подходящий для вас режим, чтобы пользователи, не являющиеся членами вашей группы, не могли выполнять нежелательные операции над вашими файлами. Действие команды длится до тех пор, пока вы не выйдете из системы либо не выполните команду umask еще раз.

Как правило, значение umask устанавливается в файле */etc/profile*, доступ к которому имеют все пользователи. Поэтому, если вы хотите установить общесистемное значение umask, отредактируйте данный файл (для этого нужно иметь права администратора). Свое собственное значение umask можно задать в файле *.profile* или *.bash_profile*, находящемся в каталоге */home*.

1.8.1. Обработка значений umask

Команда umask задает восьмеричное число, которое при создании каждого файла и каталога вычитается из стандартного значения режима доступа. Полученное значение режима присваивается файлу или каталогу. Стандартному режиму доступа к каталогам соответствует число 777, а режиму доступа к файлам — 666 (система не позволяет создавать текстовые файлы с установленными битами выполнения, эти биты следует добавлять отдельно с помощью команды chmod). Значение umask также состоит из трех трехбитовых наборов: для владельца, группы и других пользователей.

Общий формат команды umask таков:

```
umask nnn
```

где *nnn* — это маска режима в диапазоне от 000 до 777.

Ниже показано, как на основании значения umask определить режим доступа к файлу или каталогу (табл. 1.8).

Таблица 1.8. Интерпретация значения umask

Цифра в значении umask	Результат для файла	Результат для каталога
0	6	7
1	6	6
2	4	5
3	4	4
4	2	3
5	2	2
6	0	1
7	0	0

Из таблицы следует, что, например, значению umask, равному 002, соответствует режим 664 для файлов и 775 для каталогов.

Если вам удобнее работать со строками режима, руководствуйтесь описанной ниже последовательностью действий. Предположим, значение umask равно 002.

1. Сначала запишите полную строку режима, эквивалентную числу 777.
2. Под ней запишите строку режима, соответствующую значению umask (002).
3. Вычеркните из первой строки те символы, которые дублируются в тех же позициях во второй строке. Вы получите строку режима для каталогов.
4. Вычеркните из полученной строки все символы x. Вы получите строку режима для файлов.

1. Полная строка режима `rwxrwxrwx` (777)
2. Значение umask (002) `-------w-`
3. Строка режима для каталогов `rwxrwxr-x` (775)
4. Строка режима для файлов `rw-rw-r--` (664)

1.8.2. Примеры установки значений umask

В табл. 1.9 представлены некоторые возможные значения umask и указаны соответствующие им режимы доступа к файлам и каталогам.

Таблица 1.9. Примеры значений umask

Значение umask	Режим доступа к каталогам	Режим доступа к файлам
022	755	644
027	750	640
002	775	664
006	771	660
007	770	660

Для просмотра текущего значения umask введите команду umask без параметров.

```
$ umask
022
$ touch file1
$ ls -l file1
-rw-r--r--   1 dave     admin           0 Feb 18 42:05 file1
```

Чтобы изменить существующую установку, просто укажите новый аргумент команды `umask`:

```
$ umask 002
```

Убедимся в том, что система приняла изменения:

```
$ umask
002
$ touch file2
$ ls -l file2
-rw-rw-r--   1 dave     admin           0 Feb 18 45:07 file2
```

1.9. Символические ссылки

Существует два типа ссылок: жесткие и символические (мягкие). Мы рассмотрим последние. Символическая ссылка представляет собой файл, содержащий имя другого файла и в действительности являющийся указателем на файл.

1.9.1. Применение символических ссылок

Предположим, у нас есть файл с информацией о продажах, находящийся в каталоге */usr/local/admin/sales*. Необходимо, чтобы каждый пользователь мог работать с этим файлом. Вместо того чтобы создавать множество копий файла в пользовательских каталогах, можно образовать в них символические ссылки, которые указывают на исходный файл в каталоге */usr/local/admin/sales*. Тогда о всех изменениях, производимых в файле любым пользователем, немедленно узнают остальные пользователи. Имена символических ссылок могут быть произвольными и не обязаны совпадать с именем исходного файла.

Ссылки удобны в том случае, когда для получения доступа к файлу необходимо пройти через большое количество подкаталогов. Вместо этого можно создать ссылку в текущем каталоге, которая будет указывать на подкаталог, глубоко "спрятанный" в недрах других подкаталогов. Это также избавляет вас от необходимости запоминать местоположение таких файлов.

1.9.2. Примеры создания символических ссылок

Символическая ссылка создается с помощью команды `ln -s`, формат которой таков:

```
ln -s исходный_файл [имя_ссылки]
```

Если имя ссылки не указано, будет создана ссылка, имя которой совпадает с именем исходного файла.

Рассмотрим случай, когда в системе регистрируются 40 пользователей, относящихся к двум группам — менеджеры (`sales`) и административный персонал (`admin`). Для каждой группы при входе в систему должны быть заданы свои установки. Сначала

я удалю все профильные файлы (*.profile*) во всех пользовательских начальных каталогах, а затем создам в каталоге */usr/local/menus* два новых профильных файла — *sales.profile* и *admin.profile*. В начальном каталоге каждого пользователя необходимо создать символическую ссылку на один из этих двух файлов. Вот как это делается для пользователя `matty`, являющегося членом группы `sales`:

```
$ cd /home/sales/matty
$ rm .profile
$ ln -s /usr/local/menus/sales.profile .profile
$ ls -la .profile
lrwxrwxrwx   1 matty    sales        5567 Oct 3 05:40 .profile ->
/usr/local/menus/sales.profile
```

Аналогичные действия выполняются для всех пользователей. Теперь для изменения любого из профилей достаточно поменять всего один файл — либо *sales.profile*, либо *admin.profile*.

Когда ссылка больше не нужна, ее можно удалить. Однако помните, что при удалении исходного файла символическая ссылка не удаляется автоматически, а лишь становится недействительной. При обращении к ней выводится сообщение "No such file or directory" (нет такого файла или каталога).

Символическая ссылка может указывать на файл или каталог, находящийся в другой файловой системе. Следует только учитывать, что у вас должно быть право поиска во всех каталогах, перечисленных в путевом имени к исходному файлу. Режим доступа к ссылке устанавливается равным 777 (`rwxrwxrwx`), но режим доступа к исходному файлу не меняется.

После инсталляции новой системы мне часто приходится создавать ссылку на каталог */tmp* из каталога */var*, так как некоторые приложения рассчитывают на каталог */var/tmp* (а он не всегда существует) для размещения в нем своих временных файлов. Чтобы хранить все временные файлы в одном месте и не создавать новый каталог, я формирую символическую ссылку:

```
$ ln -s /tmp /var/tmp
$ cd /var
$ ls -l
...
lrwxrwxrwx   1 root     root         5567 Sep 9 10:40 tmp -> /tmp
...
```

1.10. Заключение

Эта глава содержит базовые сведения о системе безопасности файлов. Будьте предельно внимательны, реализуя наши рекомендации. Небольшая ошибка при вводе команды `chmod -R` из корневого каталога может привести к огромным затратам времени на то, чтобы восстановить для файлов корректный режим доступа.

Создание сценариев с установленным битом SUID является весьма удобным. Но старайтесь осторожно применять SUID-сценарии, обеспечивающие предоставление прав пользователя `root`.

ГЛАВА 2

Команды find и xargs

Часто в процессе работы возникает необходимость осуществить поиск файлов с определенными характеристиками, такими как права доступа, размер, тип и т.д. Команда `find` представляет собой универсальный инструмент поиска: она позволяет искать файлы и каталоги, просматривать все каталоги в системе или только текущий каталог.

В этой главе рассматриваются следующие темы, связанные с применением команды `find`:

- параметры команды `find`;
- примеры использования различных опций команды `find`;
- примеры совместного использования команд `xargs` и `find`.

Возможности команды `find` обширны, велик и список предлагаемых опций. В настоящей главе описаны наиболее важные из них. Команда `find` может проводить поиск даже на дисках NFS (Network File System — сетевая файловая система), конечно, при наличии соответствующих разрешений. В подобных случаях команда обычно выполняется в фоновом режиме, поскольку просмотр дерева каталогов требует значительных затрат времени.

Общий формат команды `find` таков:

```
find путевое_имя -опции
```

где *путевое_имя* — это каталог, с которого необходимо начинать поиск. Символ '.' служит для обозначения текущего каталога, символ '/' — корневого каталога, а символ '~' — записанного в переменной `$HOME` начального каталога текущего пользователя.

2.1. Опции команды find

Остановимся на описании основных опций команды `find`.

Таблица 2.1. Основные опции команды `find`

`-name`	Поиск файлов, имена которых соответствуют заданному шаблону
`-print`	Запись полных имен найденных файлов в стандартный поток вывода
`-perm`	Поиск файлов, для которых установлен указанный режим доступа
`-prune`	Применяется для того, чтобы команда `find` не выполняла рекурсивный поиск по уже найденному путевому имени; если указана опция `-depth`, опция `-prune` игнорируется

`-user`	Поиск файлов, принадлежащих указанному пользователю
`-group`	Поиск файлов, которые принадлежат данной группе
`-mtime` *−n +n*	Поиск файлов, содержимое которых модифицировалось менее чем (-) или более чем (+) `n` дней назад; имеются также опции `-atime` и `-ctime`, которые позволяют осуществлять поиск файлов соответственно по дате последнего чтения и дате последнего изменения атрибутов файла
`-nogroup`	Поиск файлов, принадлежащих несуществующей группе, для которой, иначе говоря, отсутствует запись в файле `/etc/groups`
`-nouser`	Поиск файлов, принадлежащих несуществующему пользователю, для которого, другими словами, отсутствует запись в файле `/etc/passwd`
`-newer` *файл*	Поиск файлов, которые созданы позднее, чем указанный файл
`-type`	Поиск файлов определенного типа, а именно: `b` — специальный блочный файл; `d` — каталог; `c` — специальный символьный файл; `p` — именованный канал; `l` — символическая ссылка; `s` — сокет; `f` — обычный файл
`-size` *n*	Поиск файлов, размер которых составляет `n` единиц; возможны следующие единицы измерения: `b` — блок размером 512 байтов (установка по умолчанию); `c` — байт; `k` — килобайт (1024 байта); `w` — двухбайтовое слово
`-depth`	При поиске файлов сначала просматривается содержимое текущего каталога и лишь затем проверяется запись, соответствующая самому каталогу
`-fstype`	Поиск файлов, которые находятся в файловой системе определенного типа; обычно соответствующие сведения хранятся в файле */etc/fstab*, который содержит данные о файловых системах, используемых на локальном компьютере
`-mount`	Поиск файлов только в текущей файловой системе; аналогом этой опции является опция `-xdev`
`-exec`	Выполнение команды интерпретатора shell для всех обнаруженных файлов; выполняемые команды имеют формат `команда {} \;` (обратите внимание на наличие пробела между символами `{}` и `\;`)
`-ok`	Аналогична опции `-exec`, но перед выполнением команды отображается запрос

2.1.1. Опция -name

При работе с командой `find` чаще всего используется опция `-name`. После нее в кавычках должен быть указан шаблон имени файла.

Если необходимо найти все файлы с расширением `txt` в вашем начальном каталоге, укажите символ '~' в качестве путевого имени. Имя начального каталога будет извлечено из переменной `$HOME`.

```
$ find ~ -name "*.txt" -print
```

Чтобы найти все файлы с расширением `txt`, находящиеся в текущем каталоге, следует воспользоваться такой командой:

```
$ find . -name "*.txt" -print
```

Для нахождения в текущем каталоге всех файлов, в именах которых встречается хотя бы один символ в верхнем регистре, введите следующую команду:

```
$ find . -name "[A-Z]*" -print
```

Найти в каталоге */etc* файлы, имена которых начинаются с символов "host", позволяет команда

```
$ find /etc -name "host*" -print
```

Поиск в начальном каталоге всех файлов с расширением `txt`, а также файлов, имена которых начинаются с точки, производит команда

```
$ find ~ -name "*.txt" -print -o -name ".*" -print
```

Опция `-o` является обозначением операции логического ИЛИ. В случае ее применения помимо файлов с обычными именами будут найдены файлы, имена которых начинаются с точки.

Если вы хотите получить список всех файлов в системе, не имеющих расширения, выполните представленную ниже команду, но будьте осторожны, так как она может существенно замедлить работу системы:

```
$ find / -name "*" -print
```

Ниже показано, как найти все файлы, в именах которых сначала следуют символы нижнего регистра, а за ними — две цифры и расширение `.txt` (например, `ax37.txt`):

```
$ find . -name "[a-z][a-z][0--9][0--9].txt" -print
```

2.1.2. Опция -perm

Опция `-perm` позволяет находить файлы с заданным режимом доступа. Например, для поиска файлов с режимом доступа 755 (их может просматривать и выполнять любой пользователь, но только владелец имеет право осуществлять запись) следует воспользоваться такой командой:

```
$ find . -perm 755 -print
```

Если перед значением режима вставить дефис, будет произведен поиск файлов, для которых установлены все указанные биты разрешений, остальные биты при этом

игнорируются. Например, следующая команда ищет файлы, к которым другие пользователи имеют полный доступ:

```
$ find . -perm -007 -print
```

Если же перед значением режима введен знак "плюс", осуществляется поиск файлов, для которых установлен хотя бы один из указанных битов разрешений, при этом остальные биты игнорируются.

2.1.3. Опция -prune

Когда вы не хотите вести поиск в том или ином каталоге, воспользуйтесь опцией `-prune`. Она служит указанием остановить поиск на текущем путевом имени. Если путевое имя указывает на каталог, команда `find` не будет заходить в него. При наличии опции `-depth` опция `-prune` игнорируется.

Следующая команда проводит поиск в текущем каталоге, не заходя в подкаталог */bin*:

```
$ find . -name "bin" -prune -o -print
```

2.1.4. Опции -user и -nouser

Чтобы найти файлы, принадлежащие определенному пользователю, укажите в команде `find` опцию `-user`, а также имя пользователя. Например, поиск в начальном каталоге файлов, принадлежащих пользователю `dave`, осуществляется посредством такой команды:

```
$ find ~ -user dave -print
```

Поиск в каталоге */etc* файлов, принадлежащих пользователю `uucp`, выполняет следующая команда:

```
$ find /etc -user uucp -print
```

Благодаря опции `-nouser` возможен поиск файлов, принадлежащих несуществующим пользователям. При ее использовании производится поиск файлов, для владельцев которых нет записи в файле */etc/passwd*. Конкретное имя пользователя указывать не нужно: команда `find` выполняет всю необходимую работу сама. Чтобы найти все файлы, которые принадлежат несуществующим пользователям и находятся в каталоге */home*, задайте вот такую команду:

```
$ find /home -nouser -print
```

2.1.5. Опции -group и -nogroup

Опции `-group` и `-nogroup` аналогичны опциям `-user` и `-nouser` и позволяют искать файлы, принадлежащие заданной группе или несуществующим группам. Ниже приведена команда для нахождения в каталоге */apps* всех файлов, которыми владеют пользователи группы `accts`:

```
$ find /apps -group accts -print
```

Следующая команда ищет во всей системе файлы, принадлежащие несуществующим группам:

```
$ find / -nogroup -print
```

2.1.6. Опция -mtime

Опцию `-mtime` следует применять при поиске файлов, доступ к которым осуществлялся x дней назад. Если аргумент опции снабдить знаком '-', будут отобраны файлы, к которым не было доступа в течение x дней. Аргумент со знаком '+' приводит к противоположному результату — производится отбор файлов, доступ к которым осуществлялся на протяжении последних x дней.

Найти все файлы, которые не обновлялись в течение последних пяти дней, позволяет следующая команда:

```
$ find / -mtime -5 -print
```

Ниже приведена команда, выполняющая поиск в каталоге */var/adm* файлов, которые обновлялись в течение последних трех дней:

```
$ find /var/adm -mtime +3 -print
```

2.1.7. Опция -newer

Если необходимо найти файлы, доступ к которым осуществлялся в промежутке времени между обновлениями двух заданных файлов, воспользуйтесь опцией `-newer`. Общий формат ее применения таков:

```
-newer старый_файл ! -newer новый_файл
```

Знак '!' является оператором логического отрицания. Он означает: найти файлы, которые новее, чем *старый_файл*, но старее, чем *новый_файл*.

Предположим, у нас есть два файла, которые обновлялись с интервалом немногим более двух дней:

```
-rwxr-xr-x   1 root     root           92 Apr 18 11:18 age.awk
-rwxrwxr-x   1 root     root         1054 Apr 20 19:37 belts.awk
```

Для нахождения всех файлов, которые обновлялись позже, чем *age.awk*, но раньше, чем *belts.awk*, выполните следующую команду (применение опции `-exec` описано чуть ниже):

```
$ find . -newer age.awk ! -newer belts.awk -exec ls -l {} \;
-rwxrwxr-x   1 root     root           62 Apr 18 11:32 ./who.awk
-rwxrwxr-x   1 root     root           49 Apr 18 12:05 ./group.awk
-rw-r--r--   1 root     root          201 Apr 20 19:30 ./grade2.txt
-rwxrwxr-x   1 root     root         1054 Apr 20 19:37 ./belts.awk
```

Но как быть, если необходимо найти файлы, созданные, скажем, в течение последних двух часов, а у вас нет файла, сформированного ровно два часа назад, с которым можно было бы сравнивать? Создайте такой файл! Для этой цели предназначена команда `touch -t`, которая создает файл с заданной временной меткой в формате *ММДДччмм* (месяц-день-часы-минуты). Например:

```
$ touch -t 05042140 dstamp
$ ls -l dstamp
-rw-r--r--   1 dave     admin           0 May 4 21:40 dstamp
```

В результате будет получен файл, дата создания которого — 4 мая, время создания —21:40 (предполагается, что текущее время — 23:40). Теперь можно применить

команду find с опцией -newer для нахождения всех файлов, которые обновлялись в течение последних двух часов:

```
$ find . -newer datamp -print
```

2.1.8. Опция -type

Операционные системы UNIX и Linux поддерживают различные типы файлов (см. главу 1). Поиск файлов нужного типа осуществляется посредством команды find с опцией -type. Например, для нахождения всех подкаталогов в каталоге */etc* воспользуйтесь такой командой:

```
$ find /etc -type d -print
```

Чтобы получить список всех файлов, но не каталогов, выполните следующую команду:

```
$ find . ! -type d -print
```

Ниже приведена команда, которая предназначена для поиска всех символических ссылок в каталоге */etc*:

```
$ find /etc -type l -print
```

2.1.9. Опция -size

В процессе поиска размер файла указывается с помощью опции -size N, где *N* — размер файла в блоках по 512 байтов. Возможные аргументы имеют следующие значения: +N — поиск файлов, размер которых больше заданного, -N — меньше заданного, N — равен заданному. Если в аргументе дополнительно указан символ c, то размер считается заданным в байтах, а не в блоках, а если символ k — в килобайтах.

Для поиска файлов, размер которых превышает 1 Мб, предназначена команда

```
$ find . -size +1000k -print
```

Следующая команда выполняет поиск в каталоге */home/apache* файлов, размер которых в точности равен 100 байтам:

```
$ find /home/apache -size 100c -print
```

Произвести поиск файлов, размер которых превышает 10 блоков (5120 байтов), позволяет приведенная ниже команда:

```
$ find . -size +10 -print
```

2.1.10. Опция -depth

Опция -depth позволяет организовать поиск таким образом, что сначала проверяются все файлы текущего каталога (и рекурсивно все его подкаталоги) и только в конце — запись самого каталога. Эта опция широко применяется при создании списка файлов, помещаемых в архив на магнитной ленте с помощью команды cpio или tar, так как в этом случае сначала записывается на ленту образ каталога и лишь после этого задаются права доступа к нему. Это позволяет пользователю архивировать те каталоги, для которых у него нет разрешения на запись.

Следующая команда выводит список всех файлов и подкаталогов текущего каталога:

```
$ find . -name "*" -print -o -name ".*" -print -depth
```

Вот как могут выглядеть результаты ее работы:

```
./.Xdefaults
./.bash_logout
./.bash_profile
./.bashrc
./.bash_history
./file
./Dir/file1
./Dir/file2
./Dir/file3
./Dir/Subdir/file4
./Dir/Subdir
./Dir
.
```

2.1.11. Опция -mount

Поиск файлов только в текущей файловой системе, исключая другие смонтированные файловые системы, обеспечивает опция `-mount` команды `find`. В следующем примере осуществляется поиск всех файлов с расширением `XC` в текущем разделе диска:

```
$ find / -name "*.XC" -mount -print
```

2.1.12. Поиск файлов с последующей архивацией командой cpio

Команда `cpio` применяется главным образом для записи файлов на магнитную ленту и чтения их с ленты. Очень часто она используется совместно с командой `find`, по каналу принимая от нее список файлов.

Вот как выполняется запись на магнитную ленту содержимого каталогов */etc*, */home* и */apps*:

```
$ cd /
$ find etc home apps -depth -print | cpio -ov > dev/rmt0
```

Опция `-o` команды `cpio` задает режим записи файлов на ленту. Опция `-v` (verbose — словесный режим) является указанием команде `cpio` сообщать о каждом обрабатываемом файле.

Обратите внимание на то, что в именах каталогов отсутствует начальный символ '/'. Таким образом задаются *относительные* путевые имена архивируемых каталогов, что при последующем чтении файлов из архива позволит воссоздавать их в любой части операционной системы, а не только в корневом каталоге.

2.1.13. Опции -exec и -ok

Предположим, вы нашли нужные файлы и хотите выполнить по отношению к ним определенные действия. В этом случае вам понадобится опция `-exec` (некоторые системы позволяют с помощью опции `-exec` выполнять только команды `ls` или

ls -l). Многие пользователи применяют опцию -exec для нахождения старых файлов, подлежащих удалению. Я рекомендую вместо команды rm выполнить сначала команду ls, чтобы убедиться в том, что команда find нашла именно те файлы, которые нужно удалить.

После опции -exec следует указать выполняемую команду, а затем ввести фигурные скобки, пробел, обратную косую черту и, наконец, точку с запятой. Рассмотрим пример:

```
$ find . -type f -exec ls -l {} \;
-rwxr-xr-x  10 root     wheel        1222 Jan 4 1993 ./sbin/C80
-rwxr-xr-x  10 root     wheel        1222 Jan 4 1993 ./sbin/Normal
-rwxr-xr-x  10 root     wheel        1222 Jan 4 1993 ./sbin/Revvid
```

Здесь выполняется поиск обычных файлов, список которых отображается на экране с помощью команды ls -l.

Чтобы найти файлы, которые не обновлялись в каталоге */logs* в течение последних пяти дней, и удалить их, выполните следующую команду:

```
$ find /logs -type f -mtime +5 -exec rm {} \;
```

Следует соблюдать осторожность при перемещении и удалении файлов. Пользуйтесь опцией -ok, которая позволяет выполнять команды mv и rm в безопасном режиме (перед обработкой очередного файла выдается запрос на подтверждение). В следующем примере команда find находит файлы с расширением LOG, и если какой-то файл создан более пяти дней назад, она удаляет его, но сначала просит вас подтвердить эту операцию:

```
$ find . -name "*.LOG" -mtime +5 -ok rm {} \;
< rm ... ./nets.LOG > ? y
```

Для удаления файла введите y, а для предотвращения этого действия — n.

2.1.14. Дополнительные примеры использования команды find

Рассмотрим еще несколько примеров, иллюстрирующих применение команды find. Ниже показано, как найти все файлы в своем начальном каталоге:

```
$ find ~ -print
```

Найти все файлы, для которых установлен бит SUID, позволяет следующая команда:

```
$ find . -type f -perm +4000 -print
```

Чтобы получить список пустых файлов, воспользуйтесь такой командой:

```
$ find / -type f -size 0 -exec ls -l {} \;
```

В одной из моих систем каждый день создается системный журнал аудита. К имени журнального файла добавляется номер, что позволяет сразу определить, какой файл создан позже, а какой — раньше. Например, версии файла *admin.log* нумеруются последовательно: *admin.log.001*, *admin.log.002* и т.д. Ниже приведена команда find, которая удаляет все файлы *admin.log*, созданные более семи дней назад:

```
$ find /logs -name 'admin.log.[0-9][0-9][0-9]' -atime +7 exec \ rm {} \;
```

2.2. Команда xargs

При наличии опции `-exec` команда `find` передает указанной команде все найденные файлы, которые обрабатываются за один раз. К сожалению, в некоторых системах длина командной строки ограниченна, поэтому при обработке большого числа файлов может быть выдано сообщение об ошибке, которое обычно гласит: "Too many arguments" (слишком много аргументов) или "Arguments too long" (слишком большой список аргументов). В этой ситуации на помощь приходит команда `xargs`. Файлы, полученные от команды `find`, она обрабатывает порциями, а не все сразу.

Рассмотрим пример, в котором команда `find` возвращает список всех файлов, имеющихся в системе, а команда `xargs` выполняет для них команду `file`, проверяющую тип каждого файла:

```
$ find / -type f -print | xargs file
/etc/protocols: English text
/etc/securetty: ASCII text
...
```

Ниже приведен пример, демонстрирующий поиск файлов дампа, имена которых команда `echo` помещает в файл */tmp/core.log*:

```
$ find / -name core -print | xargs echo > /tmp/core.log
```

В следующем примере в каталоге */apps/audit* выполняется поиск всех файлов, к которым другие пользователи имеют полный доступ. Команда `chmod` удаляет для них разрешение на запись:

```
$ find /apps/audit -perm -7 -print | xargs chmod o-w
```

Завершает наш перечень пример, в котором команда `grep` ищет файлы, содержащие слово "device":

```
$ find / -type f -print | xargs grep "device"
```

2.3. Заключение

Команда `find` представляет собой прекрасный инструмент поиска различных файлов по самым разнообразным критериям. Благодаря опции `-exec`, а также команде `xargs` найденные файлы могут быть обработаны практически любой системной командой.

ГЛАВА 3

Выполнение команд в фоновом режиме

Когда вы работаете за терминалом, во многих случаях неудобен вывод на экран результатов выполнения заданий в системе, ведь в это время вы можете быть заняты другими важными делами, например просмотром сообщений электронной почты. Кроме того, иногда возникает необходимость выполнять задания, интенсивно использующие дисковые ресурсы или ресурсы центрального процессора, в то время, когда загрузка системы минимальна. Для этих целей специально разработаны команды, которые позволяют запускать задания в фоновом режиме, а не на экране терминала.

В этой главе рассматриваются следующие темы:

- планирование заданий с помощью программы `cron`;
- планирование заданий с помощью команды `at`;
- выполнение заданий в фоновом режиме;
- выполнение заданий с помощью команды `nohup`.

`cron`	Системный планировщик, применяемый для многократного выполнения заданий в указанные периоды времени; является демоном, т.е. работает только в фоновом режиме
`at`	Команда, которая служит для однократного выполнения заданий в назначенное время
`&`	Оператор, позволяющий перевести задание в фоновый режим
`nohup`	Команда для перевода задания в фоновый режим таким образом, чтобы оно не реагировало на сигнал HUP (hang-up — отбой) и продолжало выполняться, даже если запустивший его пользователь выйдет из системы

3.1. Планировщик cron и команда crontab

Программа `cron` является основным системным планировщиком, служащим для выполнения различных заданий в фоновом режиме. Команда `crontab` позволяет пользователям создавать, редактировать и удалять инструкции для программы `cron` посредством специального `crontab`-файла. У каждого пользователя может быть свой `crontab`-файл, но в крупных системах администратор (пользователь `root`) обычно исключает данную возможность. В этом случае администратор создает вспомогательные файлы *cron.deny* и *cron.allow*, содержащие списки пользователей, которым соответственно запрещено и разрешено выполнять команду `crontab`.

3.1.1. Структура crontab-файла

Чтобы планировать выполнение заданий в определенное время, нужно знать формат записей в `crontab`-файле. Каждая запись содержит шесть полей:

1 Минуты — 0—59
2 Часы — 0—23 (0 означает полночь)
3 День месяца — 1—31
4 Месяц — 1—12
5 День недели — 0—7 (0 и 7 означает воскресенье)
6 Команда, которая должна быть выполнена

Общий формат записи таков:

```
минуты часы день_месяца месяц день_недели команда
```

Все поля разделяются пробелами.

В первых пяти полях записи могут быть указаны диапазоны значений. Например, для того чтобы обеспечить выполнение задания с понедельника по пятницу, надлежит указать в пятом поле значение `1-5`.

Допускается также задание списка значений, разделенных запятыми. Если, например, задание должно быть выполнено только в понедельник и четверг, следует ввести `1, 4`.

Символ звездочки (*) — это обозначение диапазона "от первого до последнего", т.е. каждую минуту, каждый день и т.д. Если указан диапазон, то можно задать для него шаг пропуска с помощью символа /. Например, запись `*/2` означает "каждый второй".

В `crontab`-файле допускаются комментарии. В начале строки комментария должен стоять символ #.

3.1.2. Примеры записей в crontab-файле

Запись

```
30 21 * * * /apps/bin/cleanup.sh
```

означает выполнение сценария *cleanup.sh* в каталоге */apps/bin* каждый вечер в 21:30.

Запись

```
45 4 1,10,22 * * /apps/bin/backup.sh
```

означает выполнение сценария *backup.sh* в каталоге */apps/bin* в 4:45 утра 1-го, 10-го и 22-го числа каждого месяца.

Запись

```
10 1 * * 6,0 /bin/find -name "core" -exec rm {} \;
```

означает выполнение команды `find` для удаления файлов дампа в 1:10 ночи по субботам и воскресеньям.

Запись

```
0,30 18-23 * * * /apps/bin/dbcheck.sh
```

означает выполнение сценария *dbcheck.sh* в каталоге */apps/bin* каждые полчаса между 18:00 и 23:00.

Запись

```
0 23 * * 6 /apps/bin/qtrend.sh
```

означает выполнение сценария *qtrend.sh* в каталоге */apps/bin* в 23:00 каждую субботу.

При выполнении команд и сценариев, указанных в crontab-файле, следует убедиться, что корректно заданы все необходимые переменные среды. Программа cron не сделает это за вас: это не входит в ее компетенцию. Поэтому локальные переменные среды должны быть установлены вручную, в отличие от глобальных переменных, которые устанавливаются автоматически. Данная задача может быть решена непосредственно в crontab-файле за счет создания записи следующего вида:

```
имя_переменной = значение
```

Если программа cron не сможет выполнить поступившую команду, пользователь получит электронное сообщение, в котором будут указаны причины неудачи.

3.1.3. Опции команды crontab

Общий формат команды crontab таков:

```
crontab [-u пользователь] -e -l -r
```

Опция	Назначение
-u *пользователь*	Установка имени пользователя, для которого нужно создать crontab-файл
-e	Активизация режима редактирования crontab-файла
-l	Отображение содержимого crontab-файла
-r	Удаление crontab-файла

Если вы хотите работать с собственным crontab-файлом, то указывать опцию -u нет необходимости.

3.1.4. Создание crontab-файла

Сначала, еще до того как crontab-файл будет помещен в очередь заданий программы cron, необходимо установить переменную среды EDITOR. Благодаря этому планировщик получит указание относительно того, какой редактор следует использовать при обработке crontab-файлов. Если вы предпочитаете редактор vi, откройте файл *.profile* или *.bash_profile*, находящийся в вашем начальном каталоге, и поместите в него следующие команды:

```
EDITOR = vi; export EDITOR
```

Далее создайте новый файл *<имя_пользователя>cron*, где *<имя_пользователя>* — ваше регистрационное имя. Вот пример содержимого такого файла:

```
# вывод текущей даты на экран
# каждые 15 минут между 18:00 и 6:00
0,15,30,45 18-06 * * * /bin/echo `date` > /dev/console
```

Приведенная выше запись задает отображение на экране текущей даты каждые 15 минут в указанном интервале времени. Теперь, если система вдруг "зависнет", вы сможете определить, когда это произошло.

Чтобы поместить в очередь заданий планировщика cron свой crontab-файл, выполните команду crontab, указав в ней имя созданного файла:

```
$ crontab davecron
```

Копия файла будет помещена в каталог */var/spool/cron*, а имя копии совпадет с вашим регистрационным именем (в данном случае — dave).

3.1.5. Вывод на экран содержимого crontab-файла

Для вывода на экран содержимого crontab-файла предназначена команда crontab -l:

```
$ crontab -l
# (davecron installed on Tue May 4 13:07:43 1999)
# вывод текущей даты на экран
# каждые 15 минут между 18:00 и 6:00
0,15,30,45 18-06 * * * /bin/echo `date` > /dev/console
```

Вот как легко можно создать резервную копию crontab-файла в своем начальном каталоге:

```
$ crontab -l > $HOME/davecron
```

3.1.6. Редактирование crontab-файла

Для добавления, редактирования или удаления записей в crontab-файле используется тот редактор, который указан в переменной среды EDITOR. Чтобы отредактировать файл, выполните команду

```
$ crontab -e
```

При сохранении файла программа cron проверяет значения полей и информирует пользователя об обнаруженных ошибках. Если какая-либо запись содержит ошибку, файл не будет принят.

В процессе редактирования crontab-файла можно добавлять в него новые записи. Добавим, например, следующую запись:

```
# удаление файлов дампа в 3.30 утра в 1-й, 7-й, 14-й,
# 21-й и 26-й день каждого месяца
30 3 1,7,14,21,26 * * /bin/find -name "core" -exec rm {} \;
```

Желательно размещать перед каждой записью комментарий, объясняющий ее назначение.

Теперь сохраним файл, выйдем из редактора и проверим результат:

```
$ crontab -l
# (davecron installed on Tue May 4 13:07:43 1999)
# вывод текущей даты на экран
# каждые 15 минут между 18:00 и 6:00
0,15,30,45 18-06 * * * /bin/echo `date` > /dev/console
# удаление файлов дампа в 3.30 утра в 1-й, 7-й, 14-й,
# 21-й и 26-й день каждого месяца
30 3 1,7,14,21,26 * * /bin/find -name "core" -exec rm {} \;
```

3.1.7. Удаление crontab-файла

Для удаления своего crontab-файла введите команду

```
$ crontab -r
```

3.1.8. Восстановление утерянного crontab-файла

Если crontab-файл случайно удален, инсталлируйте заново исходный файл из вашего начального каталога:

```
$ crontab <имя_файла>
```

Именно по этой причине в документации к программе cron говорится о том, что не рекомендуется прибегать к непосредственному редактированию crontab-файла. Следует вносить все изменения в копию файла и инсталлировать ее заново.

3.2. Команда at

Команда at позволяет передавать задания демону cron для одноразового выполнения в назначенное время. Выдавая задание, команда at сохраняет в отдельном файле как его текст, так и все текущие переменные среды. Заметим, что команда crontab не делает этого. По умолчанию все результаты выполнения задания направляются пользователю в виде электронного сообщения.

Как и в случае с программой cron, пользователь root может контролировать, кому разрешено или запрещено выполнять команду at. Соответствующие списки пользователей содержатся в файлах *at.allow* и *at.deny*, находящихся в каталоге */etc*.

Базовый формат команды at таков:

```
at [-f файл] [-l -d -m] время
```

Опция	Назначение
-f *файл*	Список заданий должен быть взят из указанного файла
-l	Вывод на экран списка заданий, которые ожидают выполнения; аналогична команде atq
-d	Удаление задания с указанным номером; аналогична команде atrm (в некоторых системах заменяется опцией -r)
-m	Выдача пользователю электронного сообщения о завершении задания
время	Спецификация времени, когда будет выполнено задание. Эта спецификация может быть довольно сложной. Допускается указание не только времени в формате *часы:минуты*, но и даты, а также многочисленных ключевых слов, таких как названия дней недели, месяцы, наречий today (сегодня), tomorrow (завтра), now (сейчас) и др. Наиболее удобна запись вида now + 3 hours (через три часа).

3.2.1. Запуск команд и сценариев с помощью команды at

Текст задания можно передать команде at двумя способами: в файле или в режиме командной строки at. Если задание состоит из одной команды или двух-трех команд, объединенных каналом, то удобнее воспользоваться вторым способом. Для запуска сценариев интерпретатора shell предпочтительнее первый вариант.

В случае необходимости выполнить одиночную команду вызовите команду at, указав требуемое время. Отобразится приглашение at>. Введите свою команду, а затем нажмите [Enter] и [Ctrl+D]. Рассмотрим пример.

```
$ at 21:10
at> find / -name "passwd" -print
at> <EOT>
warning: commands will be executed using /bin/sh
job 1 at 1999-05-05 21:10
```

Запись <EOT> появляется после нажатия [Ctrl+D]. Теперь в 21:10 будет выполнена команда find, ищущая в системе файлы с именем *passwd*. Обратите внимание на то, что команда at присваивает заданию уникальный идентификатор 1. Результаты выполнения команды find будут направлены вам по электронной почте. Вот фрагмент соответствующего электронного сообщения:

```
Subject: Output from your job 1
/etc/passwd
/etc/pam.d/passwd
/etc/uucp/passwd
/tmp/passwd
/root/passwd
/usr/bin/passwd
/usr/doc/uucp-1.06.1/sam ple/passwd
```

Ниже приведены примеры корректного указания времени при вызове команды at:

```
at 6.45am May 12 - 12-го мая в 6:45 утра
at 11.10pm       - в 23:10 (сегодня или завтра, если
                           это время уже прошло)
at now + 1 hour  - через час
at 9am tomorrow  - завтра в 9:00 утра
at 15:00 May 24  - 24 мая в 15:00
at 4am + 3 days  - через 3 дня в 4:00 утра
```

Если необходимо запустить с помощью команды at файл сценария, укажите его имя после опции -f, как это сделано ниже:

```
$ at 3.00pm tomorrow -f /apps/bin/db_table.sh
warning: commands will be executed using /bin/sh
job 8 at 1999-05-06 15:00
```

Сценарий *db_table.sh* будет выполнен завтра в 15:00.

Передать задание команде at позволяет также команда echo:

```
$ echo find /etc -name "passwd" -print | at now +1 minute
```

3.2.2. Просмотр списка запланированных заданий

Для того чтобы просмотреть полный список запланированных заданий, введите команду at -l или atq:

```
$ atq
2 1999-05-05 23:00 a
3 1999-05-06 06:00 a
4 1999-05-21 11:20 a
```

В первом столбце содержится идентификатор задания, за ним следуют дата и время выполнения задания. В последнем столбце находится символ `a`, указывающий на то, что задание получено от команды `at`. Существует также команда `batch`, которая планирует выполнение задания в период наименьшей загруженности системы. Задания, полученные от этой команды, помечаются в выводе команды `atq` символом `b`.

Получив задание, команда `at` создает в каталоге */var/spool/at* файл, в который помещает текст задания и заносит текущие установки всех переменных среды:

```
$ pwd
/var/spool/at
$ ls
a0000200eb7ae4 a0000400ebd228 a0000800eb7ea4 spool
```

3.2.3. Удаление запланированного задания

Для удаления задания предназначена команда `atrm` (синоним команды `at -d` или `at -r`), имеющая следующий формат:

```
atrm номер_задания
```

Чтобы удалить задание, нужно сначала получить его идентификатор. Поэтому сначала введите команду `at -l` и узнайте идентификатор интересующего вас задания, а затем выполните команду `atrm`:

```
$ at -l
2 1999-05-05 23:00 a
3 1999-05-06 06:00 a
4 1999-05-21 11:20 a

$ atrm 3
$ at -l
2 1999-05-05 23:00 a
4 1999-05-21 11:20 a
```

3.3. Оператор &

При выполнении задания в экранном режиме происходит "захват" терминала на весь этот период. Перевод задания в фоновый режим позволяет освободить терминал для других целей. Чтобы выполнить команду в фоновом режиме, укажите после нее оператор `&`:

```
команда &
```

В таком режиме удобно выполнять команду `find`, посылать задания на принтер, сортировать записи больших списков с помощью команды `sort` и т.д. Не переводите в фоновый режим те команды, которые требуют ввода информации пользователем, поскольку в этом случае работа команды будет приостановлена, а вы не узнаете об этом.

Недостаток выполнения команды в фоновом режиме заключается в том, что весь ее вывод по-прежнему направляется на терминал. По этой причине выходные потоки таких команд часто перенаправляют в файл с помощью следующей конструкции:

```
команда > выходной_файл 2>&1 &
```

Данная конструкция задает переадресацию стандартных потоков вывода и ошибок в указанный файл.

При запуске задания в фоновом режиме на экране отображается номер соответствующего процесса. Впоследствии этот номер можно использовать как для уничтожения процесса с помощью команды `kill`, так и для перевода задания в экранный режим посредством команды `fg`.

3.3.1. Запуск команды в фоновом режиме

Выполним в фоновом режиме команду `find`, которая ищет в каталоге */etc* файл *srm.conf*, и перенаправим ее вывод в файл *find.dt*:

```
$ find /etc -name "srm.conf" -print > find.dt 2>&1 &
[1] 27015
```

Номер процесса в данном случае равен 27015. После завершения выполнения задания, когда вы в очередной раз нажмете клавишу [Return], на экране отобразится такое сообщение:

```
[1]+ Done                    find /etc "srm.conf" -print
```

3.3.2. Получение списка выполняющихся процессов с помощью команды ps

Предположим, имеется сценарий *ps1*, который выполняется довольно долго:

```
$ ps1 &
[2] 28305
```

Узнать о состоянии этого задания можно с помощью команды `ps`, которая по умолчанию выводит список всех запущенных в данный момент процессов, принадлежащих текущему пользователю:

```
$ ps
   PID TTY          TIME CMD
   679 pts/0    00:00:01 bash
...
 28305 pts/0    00:02:07 ps1
...
 28310 pts/0    00:00:00 ps
```

Здесь в четырех столбцах приведена следующая информация: первый — идентификатор процесса, второй — идентификатор терминала, с которого он запущен, третий — суммарное время использования процессора, четвертый — выполняемая команда.

Если процессов слишком много, воспользуйтесь командой `grep`, указав в ней номер нужного задания:

```
$ ps | grep 28305
 28305 pts/0    00:02:20 ps1
```

Обратите внимание: команда `ps` не показывает, в каком режиме выполняется задание — в фоновом или экранном.

3.3.3. Уничтожение фонового задания

Сигнал о завершении посылается процессу командой `kill`:

```
kill [-сигнал] номер_процесса
```

Далее в этой книге мы рассмотрим, какие существуют сигналы. Пока же достаточно знать, что по умолчанию команда `kill` посылает сигнал номер 1 — HUP (hang-up — отбой). На экран выводится сообщение о прекращении задания:

```
$ kill 28305
[1]+ Terminated              ./ps1
```

Многие команды и сценарии перехватывают сигнал HUP, поэтому команда `kill -1` не уничтожает их. В этом случае нужно воспользоваться командой `kill -9`, которая посылает процессу сигнал KILL (уничтожить). Этот сигнал не перехватывается и означает безусловное уничтожение процесса.

```
$ kill -9 28305
[1] + Killed                 ./ps1
```

3.4. Команда nohup

Задание, выполняющееся в фоновом режиме, уничтожается, когда запустивший его пользователь выходит из системы. Вы можете обеспечить, чтобы после завершения сеанса работы в системе продолжилось автономное выполнение вашего задания. Для этого запустите его с помощью команды `nohup`. Общий формат этой команды таков:

```
nohup команда &
```

3.4.1. Запуск задания с помощью команды nohup

По умолчанию все выходные данные задания, запущенного с помощью команды `nohup`, направляются в файл *nohup.out*, но можно указать другой файл:

```
nohup команда > выходной_файл 2>&1 &
```

Давайте проверим работу команды `nohup` на примере упомянутого выше сценария *ps1*.

```
$ nohup ps1 &
[1] 179
nohup: appending output to 'nohup.out'
```

Теперь выйдите из интерпретатора shell, выполнив команду `logout`, снова зарегистрируйтесь и введите следующую команду:

```
$ ps x | grep ps1
 30004 ?        RN     4:01 sh ./ps1
 30006 pts/1    S      0:00 grep ps1
```

Опция `x` предназначена для вывода списка заданий, не связанных с терминалом (обратите внимание на знак вопроса во втором столбце). В третьем столбце указан статус задания. Статус `R` означает, что процесс выполняется, статус `N` — это признак снижения

приоритета у выполняемого процесса. В четвертом столбце приведено суммарное время использования процессора. Как видите, сценарий продолжает свою работу.

Если система не поддерживает команду `ps x`, воспользуйтесь опцией `-e`, которая предназначена для получения списка всех системных процессов:

```
$ ps -e | grep ps1
 30004 ?        00:04:01 sh ./ps1
```

3.4.2. Одновременный запуск нескольких заданий

Если необходимо одновременно выполнить несколько команд, можно объединить их в файле сценария, а затем запустить его с помощью команды `nohup`. Предположим, имеется следующая цепочка команд:

```
cat /home/accounts/qtr_0499 | /apps/bin/trials.awk | sort | lp
```

Поместим ее в файл:

```
$ cat > quarterend
cat /home/accounts/qtr_0499 | /apps/bin/trials.awk | sort | lp
<CTRL-D>
```

Сделаем файл исполняемым:

```
$ chmod 744 quarterend
```

Теперь запустим этот файл в фоновом режиме посредством команды `nohup`, направив результаты работы сценария в файл *qtr.out*:

```
$ nohup ./quarterend > qtr.out 2>&1 &
[5] 182
```

Обратите внимание на то, что при вызове файла *quarterend* указано путевое имя `./`. Тем самым интерпретатору shell дается указание искать данную команду в текущем каталоге.

3.5. Заключение

Мы рассмотрели различные способы выполнения заданий в фоновом режиме. Необходимость в этом может возникнуть при пакетном обновлении больших файлов или выполнении сложной операции поиска. Подобную работу лучше не делать в часы пиковой загрузки системы.

Можно создать сценарии, которые автоматически модифицируют журнальные файлы. Вам останется лишь просматривать эти файлы по мере необходимости. Таким образом, программа `cron` и другие представленные в этой главе команды позволяют существенно упростить администрирование системы.

ГЛАВА 4

Подстановка имен файлов

При работе в режиме командной строки довольно много времени уходит на поиск необходимых файлов. Интерпретатор shell предлагает набор метасимволов, позволяющих находить файлы, имена которых соответствуют предложенному шаблону.

Вот список основных метасимволов:

*	Соответствует произвольной строке, содержащей ноль и более символов
?	Соответствует любому символу
[...]	Соответствует любому символу из числа заключенных в скобки
[!...]	Соответствует любому символу за исключением тех, которые указаны в скобках

Когда интерпретатор shell встречает указанные символы в командной строке, он обрабатывает их особым образом, если только вы не защитили их с помощью кавычек, о чем говорится в главе 15.

4.1. Применение метасимвола *

Символ звездочки может служить заменой любой части имени файла. Следующая команда выводит список всех файлов, имена которых начинаются со строки "app":

```
$ ls app*
appdva          app_tapes
appdva_SLA
```

Вот как можно получить список файлов, имеющих расширение *doc*:

```
$ ls *.doc
accounts.doc    qtr_end.doc
```

Представленная ниже команда находит файлы, у которых имя начинается со строки "cl", а расширение равно *sed*:

```
$ ls cl*.sed
cleanlogs.sed       cleanmeup.sed
cleanmessages.sed
```

Звездочку удобно применять, когда, например, не хочется вводить полное имя каталога при переходе в него:

```
$ cd /etc
$ ls -l | grep ^d
```

```
...
drwxr-xr-x   2 root     root         1024 Jan 26 14:41 cron.daily
drwxr-xr-x   2 root     root         1024 Jan 27  1998 cron.hourly
drwxr-xr-x   2 root     root         1024 Jan 27  1998 cron.monthly
drwxr-xr-x   2 root     root         1024 Jan 26 14:37 cron.weekly
 ...
$ cd cron.w*
$ pwd
/etc/cron.weekly
```

4.2. Применение метасимвола ?

Знак вопроса служит для замены любого отдельного символа. Следующая команда выводит список файлов, имена которых содержат не менее трех символов, причем третьим является символ 'R':

```
$ ls ??R*
BAREAD
```

Следующая команда ищет файлы с именами, начинающимися со строки "conf", за которой следуют два произвольных символа и расширение *log*:

```
$ ls conf??.log
conf12.log    conf.2.log
conf25.log
```

Вот как можно найти файлы, имена которых состоят не менее чем из четырех символов, причем первым является символ 'f', а последним — 's':

```
$ ls f??*s
ftpaccess      ftphosts
ftpconversions ftpusers
ftpgroups
```

4.3. Применение метасимволов [...] и [!...]

Метасимволы `[...]` соответствуют любому символу из числа тех, что указаны в квадратных скобках. Представленная ниже команда выводит список файлов, имена которых начинаются с символа 'i' или 'o':

```
$ ls [io]*
inetd.conf    ioctl.save    outputrc
info-dir      inputrc
initrunlvl    issue
inittab       issue.net
```

В скобках можно задавать диапазон символов. Начальный и конечный символы при этом разделяются дефисом. Например, следующая команда ищет файлы с именем *log*, в расширении которых первый символ — цифра:

```
$ ls log.[0-9]*
log.0323      log.0325
log.0324      log.0326
```

Метасимволы `[!...]` соответствуют любому символу из числа тех, что не указаны в квадратных скобках. Если в предыдущем примере на первом месте в скобках поставить восклицательный знак, команда будет искать файлы с именами *log*, в расширении которых первый символ не является цифрой:

```
$ ls log.[!0-9]*
log.sybase
```

Приведенная ниже команда ищет файлы, имена которых начинаются со строки "LPS", два следующих символа могут быть произвольными, затем идет символ, не являющийся цифрой, а за ним — произвольная строка:

```
$ ls LPS??[!0-9]*
LPSILP          LPSOSI
LPSOPS          LPSPOPQTR
```

Поиск файлов, имена которых начинаются с символа верхнего регистра, производится посредством такой команды:

```
$ ls [A-Z]*
```

Следующая команда ищет файлы, имена которых, наоборот, начинаются с символа нижнего регистра:

```
$ ls [a-z]*
```

А эта команда находит файлы, в начале имени которых стоит цифра:

```
$ ls [0-9]*
```

Вот как можно найти все скрытые файлы (такие как *.profile*, *.rhosts*, *.history* и т.д.):

```
$ ls .*
```

4.4. Заключение

Метасимволы представляют собой универсальный инструмент поиска строк по шаблону. С их помощью легко находить нужные файлы и каталоги. Далее мы более подробно рассмотрим способы применения метасимволов в составе регулярных выражений.

ГЛАВА 5

Ввод и вывод данных в интерпретаторе shell

Команды и сценарии могут получать входные данные двумя способами: из стандартного входного потока (связан с клавиатурой) или из файла. Аналогичное разделение существует и при выводе данных: результаты работы команды или сценария по умолчанию направляются на экран терминала, но можно перенаправить их в файл. Если в процессе работы возникают ошибки, сообщения о них тоже отображаются на экране. Чтобы избежать этого, нужно перенаправить поток ошибок в файл.

В этой главе рассматриваются следующие темы:

- работа со стандартными потоками ввода, вывода и ошибок;
- переадресация ввода и вывода.

5.1. Команда echo

Команда `echo` отображает на экране указанную строку текста. Общий ее формат таков:

```
echo строка
```

В строке могут встречаться различные управляющие символы, ниже перечислены основные из них:

`\c`	запрет отображения концевого символа новой строки
`\f`	прогон страницы
`\n`	новая строка
`\t`	горизонтальная табуляция

Например, если указать управляющий символ `\c`, то по завершении вывода не будет осуществлен переход в новую строку:

```
$ echo "Как вас зовут?\t\c"
Как вас зовут?      $
```

Здесь `$` — символ приглашения.

По умолчанию подразумевается, что в конце строки находится символ новой строки:

```
$ echo "Как вас зовут?"
Как вас зовут?
$
```

В строке можно вычислять значения переменных интерпретатора shell и даже других команд. Например, следующая команда сообщает о том, каков начальный каталог текущего пользователя (переменная среды $HOME) и к какому терминалу он подключен (команда tty заключена в обратные кавычки, чтобы интерпретатор shell поместил в строку результат ее выполнения):

```
$ echo "Ваш начальный каталог $HOME, вы подключены к терминалу `tty`"
Ваш начальный каталог /home/dave, вы подключены к терминалу /dev/ttyp1
```

В Linux...

Чтобы запретить вывод символа новой строки, укажите опцию -n:

```
$ echo -n "Как вас зовут?"
```

Управляющие символы по умолчанию не распознаются. Чтобы активизировать их, задайте опцию -e:

```
$ echo -e "Как вас зовут?\t\c"
Как вас зовут?      $
```

Для вывода дополнительных пустых строк используйте управляющий символ \n:

```
$ echo "Выводим 3 пустые строки\n\n\nOK"
Выводим 3 пустые строки


OK
```

В любом месте строки можно размещать символы табуляции:

```
$ echo "Один символ табуляции\tДва табуляции\t\tOK"
Один символ табуляции   Два символа табуляции   OK
```

Чтобы перенаправить результаты работы команды echo в файл, воспользуйтесь оператором >:

```
$ echo "Строка занесена в файл." > myfile
```

В этом случае содержимое файла *myfile* будет заменено. Существует также оператор >>, который позволяет добавить строку в конец файла:

```
$ echo "Отчет создал пользователь $LOGNAME. `date` " >> myfile
```

Здесь используется переменная среды $LOGNAME, которая содержит регистрационное имя текущего пользователя.

Рассмотрим содержимое файла *myfile*:

```
$ cat myfile
Строка занесена в файл.
Отчет создал пользователь root. Sat May 22 18:25:06 GMT 1999
```

Одной из проблем, с которыми часто сталкиваются начинающие пользователи при работе с командой echo, является включение в строку двойных кавычек. Символы двойных кавычек имеют специальное назначение в интерпретаторе shell,

поэтому должны быть защищены с помощью обратной косой черты. Вот как выводится на экран строка "/dev/rmt0":

```
$ echo "\"/dev/rmt0\""
"/dev/rmt0"
```

5.2. Команда read

Команда read читает одну строку из стандартного входного потока и записывает ее содержимое в указанные переменные. Если задана единственная переменная, в нее записывается вся строка. В результате ввода команды read без параметров строка помещается в переменную среды $REPLY. При указании нескольких переменных в первую из них записывается первое слово строки, во вторую — второе слово и т.д. Последней переменной присваивается остаток строки.

Общий формат команды таков:

```
read переменная1 переменная2...
```

В следующем примере в переменную name записывается весь вводимый с клавиатуры текст до тех пор, пока не будет нажата клавиша [Enter]:

```
$ read name
Джон Алан Доу
$ echo $name
Джон Алан Доу
```

Представленная ниже команда заносит введенные имя и фамилию в две переменные. В качестве разделителя между словами используется пробел.

```
$ read name surname
Джон Доу
$ echo $name $surname
Джон Доу
```

Если во входной строке больше слов, чем указано переменных, в последнюю переменную записываются все оставшиеся слова:

```
$ read name surname
Джон Алан Доу
$ echo $name
Джон
$ echo $surname
Алан Доу
```

Следующий сценарий вызывает отдельную команду read для чтения каждой переменной:

```
$ cat var_test
#!/bin/sh
# var_test
echo "Имя:\c"
read name
echo "Отчество:\c"
read middle
echo "Фамилия:\c"
read surname
```

В Linux...

Вместо управляющего символа \c в команде echo следует указывать опцию -n:

```
$ cat var_test
#!/bin/sh
# var_test
echo -n "Имя:"
read name
echo -n "Отчество:"
read middle
echo -n "Фамилия:"
read surname
```

5.3. Команда cat

Команда cat довольно проста, но универсальна. Эту команду удобно применять как для отображения файла, так и для его создания, а также при отображении файлов, содержащих управляющие символы. Используя команду cat, следует учитывать, что процесс вывода не останавливается по достижении конца страницы — файл пролистывается до конца. Если необходимо просмотреть файл постранично, передайте вывод команды cat какой-нибудь программе постраничной разбивки:

```
$ cat myfile | more
```

или

```
$ cat myfile | pg
```

Общий формат команды cat таков:

```
cat [опции] имя_файла1...имя_файла2...
```

Из опций команды cat в первую очередь заслуживает внимания опция -v, активизирующая режим отображения непечатаемых символов.

Вывести файл *myfile* позволяет вот такая простая команда:

```
$ cat myfile
```

Для отображения сразу трех файлов — *myfile1*, *myfile2* и *myfile3* — нужно выполнить команду

```
$ cat myfile1 myfile2 myfile3
```

Чтобы сформировать файл *bigfile*, включающий содержимое файлов *myfile1*, *myfile2* и *myfile3*, следует перенаправить выходной поток предыдущей команды в новый файл:

```
$ cat myfile1 myfile2 myfile3 >> bigfile
```

Если необходимо создать новый файл и ввести в него текст, не указывайте имя файла. В таком случае команда `cat` читает данные не из файла, а из стандартного входного потока (клавиатуры), и вам остается лишь перенаправить его в новый файл:

```
$ cat >> myfile
Это новый файл
<CTRL-D>
$ pg myfile
Это новый файл
```

По завершении ввода данных нажмите [Ctrl+D].

Для просмотра управляющих символов в файле воспользуйтесь опцией `-v`. Следующая команда отображает содержимое файла, в котором встречаются символы `<CTRL-M>` (представлены как `^M`):

```
$ cat -v life.tct
ERROR ON REC AS12^M
ERROR ON REC AS31^M
```

5.4. Каналы

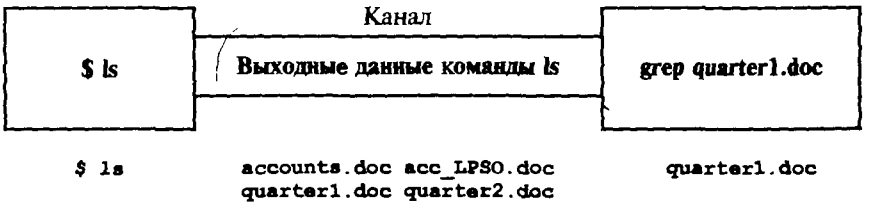
Каналом называется способ переадресации данных, при котором результаты работы одной команды передаются другой команде в виде входных данных. Канал организуется с помощью оператора `|`:

```
команда1 | команда2
```

В следующем примере команда `ls` формирует список всех файлов из текущего каталога. Этот список был бы выведен на экран, если бы не символ канала. Интерпретатор shell обнаруживает канал, перехватывает все выходные потоки команды, стоящей слева от оператора `|`, и направляет их команде, которая расположена справа от оператора. В данном случае утилита фильтрации `grep` отбирает в списке файл с именем *quarter1.doc*:

```
$ ls | grep quarter1.doc
quarter1.doc
```

Представим этот пример схематически:



При обработке строковых данных можно объединять каналами такие мощные программы фильтрации, как потоковый редактор `sed`, редактор `awk` и утилита `grep`, создавая сложные критерии отбора информации. В показанной ниже командной строке команда `who` выводит информацию о пользователях, зарегистрированных в

данный момент в системе, а программа awk выбирает из каждой строки имя пользователя (первое поле) и идентификатор терминала (второе поле):

```
$ who | awk '{print $1"\t"$2}'
matthew    pts/0
louise     pts/1
```

Следующая командная строка служит для вывода списка всех смонтированных файловых систем. Команда df формирует расширенный список с указанием всевозможных статистических данных об использовании каждой файловой системы. Программа awk извлекает из этого списка только первый столбец с именами файловых систем, а команда grep -v удаляет заголовок этого столбца, оставляя только имена.

```
$ df | awk '{print $1}' | grep -v "Filesystem"
/dev/hda5
/dev/hda8
/dev/hda6
/dev/hdb5
/dev/hdb1
/dev/hda7
/dev/hda1
```

С помощью редактора sed можно удалить из полученного списка повторяющуюся подстроку /dev/, оставив только имя раздела. Вот как это делается:

```
$ df | awk '{print $1}' | grep -v "Filesystem" | sed s'/\/dev\///g'
hda5
hda8
hda6
hdb5
hdb1
hda7
hda1
```

Команда s редактора sed предназначена для замены указанного шаблона (в данном случае \/dev\/; символы / имеют специальное назначение, поэтому защищены символами \) заданной строкой (в нашем случае это пустая строка). Флаг g означает, что замену нужно производить каждый раз, когда обнаружено совпадение, а не только первый раз.

В следующем примере команда sort сортирует строки текстового файла myfile, а результат посылается на принтер:

```
$ sort myfile | lp
```

5.5. Команда tee

Команда tee функционирует следующим образом: входные данные копируются, при этом одна копия направляется в стандартный поток вывода, а другие копии — в указанные файлы. Общий формат этой команды таков:

```
tee [-a] файлы
```

Опция -a задает добавление выводимых данных в конец файла (по умолчанию производится замена содержимого файла). Команду tee удобно применять в том случае, когда необходимо вести журнал выводимых данных или сообщений.

Рассмотрим пример. Команда who формирует список пользователей, которые зарегистрированы в данный момент в системе, а команда tee отображает этот список на экране, направляя копию в файл *who.out*.

```
$ who | tee who.out
louise    pts/1    May 20 12:58 (193.132.90.9)
matthew   pts/0    May 20 10:18 (193.132.90.1)

$ cat who.out
louise    pts/1    May 20 12:58 (193.132.90.9)
matthew   pts/0    May 20 10:18 (193.132.90.1)
```

В следующем примере команда cpio выполняет резервирование файлов из каталога */home* на магнитную ленту, а список помещаемых в архив файлов фиксируется в файле *tape.log*. Поскольку с помощью команды cpio производится последовательное добавление данных в архив, воспользуйтесь опцией -a команды tee:

```
$ find /home -depth -print | cpio -ov -0 /dev/rmt0 | tee -a tape.log
```

Чтобы сообщить пользователю о том, кто именно выполнил сценарий myscript, сохраняющий выводимые данные в файле *myscript.log*, можно перед вызовом сценария задать несложную команду echo:

```
$ echo "Сценарий myscript запущен пользователем dave" | tee -a myscript.log
$ myscript | tee -a myscript.log
```

Можно направлять вывод нескольких команд в один и тот же файл, но не забывайте применять опцию -a.

```
$ sort myfile | tee -a accounts.log
$ myscript | tee -a accounts.log
```

5.6. Стандартные потоки ввода, вывода и ошибок

С каждым процессом (командой, сценарием и т.п.), выполняемым в интерпретаторе shell, связан ряд открытых файлов, из которых процесс может читать свои данные и в которые он может записывать их. Каждый из этих файлов идентифицируется числом, называемым *дескриптором файла*, но у первых трех файлов есть также имена, которые легче запоминать:

Файл	Дескриптор
Стандартный поток ввода (*stdin*)	0
Стандартный поток вывода (*stdout*)	1
Стандартный поток ошибок (*stderr*)	2

В действительности создается 12 открытых файлов, но, как видно из таблицы, файлы с дескрипторами 0, 1 и 2 резервируются для стандартных потоков ввода, вывода и ошибок. Пользователи могут также работать с файлами, имеющими дескрипторы от 3 до 9.

5.6.1. Стандартный поток ввода

Файл стандартного потока ввода (*stdin*) имеет дескриптор 0. Из этого файла процессы извлекают свои входные данные. По умолчанию входной поток ассоциирован с клавиатурой (устройство */dev/tty*), но чаще всего он поступает по каналу от других процессов или из обычного файла.

5.6.2. Стандартный поток вывода

Файл стандартного потока вывода (*stdout*) имеет дескриптор 1. В этот файл записываются все выходные данные процесса. По умолчанию данные выводятся на экран терминала (устройство */dev/tty*), но их можно также перенаправить в файл или послать по каналу другому процессу.

5.6.3. Стандартный поток ошибок

Файл стандартного потока ошибок (*stderr*) имеет дескриптор 2. В этот файл записываются сообщения об ошибках, возникающих в ходе выполнения команды. По умолчанию сообщения об ошибках выводятся на экран терминала (устройство */dev/tty*), но их также можно перенаправить в файл. Зачем же для регистрации ошибок выделять специальный файл? Дело в том, что это очень удобный способ выделения из результатов работы команды собственно выходных данных, а также хорошая возможность эффективно организовать ведение различного рода журнальных файлов.

5.7. Файловый ввод-вывод

При вызове команд можно указывать, откуда следует принимать входные данные и куда необходимо направлять выходные данные, а также сообщения об ошибках. По умолчанию, если не указано иное, подразумевается работа с терминалом: данные вводятся с клавиатуры и выводятся на экран. Но интерпретатор shell располагает механизмом переадресации, позволяющим ассоциировать стандартные потоки с различными файлами. В табл. 5.1 приведены наиболее распространенные операторы переадресации.

Во время перенаправления стандартного потока ошибок следует указывать дескриптор файла (2). Для потоков ввода и вывода делать это не обязательно.

Таблица 5.1. Основные операторы переадресации

`команда > файл`	Направляет стандартный поток вывода в новый файл
`команда 1> файл`	Направляет стандартный поток вывода в указанный файл
`команда >> файл`	Направляет стандартный поток вывода в указанный файл (режим присоединения)
`команда > файл 2>&1`	Направляет стандартные потоки вывода и ошибок в указанный файл
`команда 2> файл`	Направляет стандартный поток ошибок в указанный файл
`команда 2>> файл`	Направляет стандартный поток ошибок в указанный файл (режим присоединения)
`команда >> файл 2>&1`	Направляет стандартные потоки вывода и ошибок в указанный файл (режим присоединения)
`команда < файл1 > файл2`	Получает входные данные из первого файла и направляет выходные данные во второй файл

`команда < файл`	В качестве стандартного входного потока получает данные из указанного файла
`команда << разделитель`	Получает данные из стандартного потока ввода до тех пор, пока не встретится разделитель
`команда <&m`	В качестве стандартного входного потока получает данные из файла с дескриптором *m*
`команда >&m`	Направляет стандартный поток вывода в файл с дескриптором *m*

5.7.1. Переадресация стандартного потока вывода

Рассмотрим, как осуществляется переадресация стандартного потока вывода. В следующей командной строке из файла */etc/passwd* извлекаются имена пользователей, известных в системе, полученный список сортируется по алфавиту, а результаты направляются в файл *sort.out*.

```
$ cat /etc/passwd | awk -F: '{print $1}'| sort > sort.out
```

Опция `-F` программы `awk` свидетельствует о том, что указанный после нее символ двоеточия является разделителем полей в файле */etc/passwd*.

В один и тот же файл в режиме присоединения можно направлять результаты работы сразу нескольких команд:

```
$ ls -l | grep ^d >> files.out
$ ls account* >> files.out
```

В первой строке в файл *files.out* помещается список каталогов, а во второй — список файлов, имена которых начинаются со строки "account".

Для создания пустого файла нулевой длины выполните следующую команду:

```
$ > myfile
```

5.7.2. Переадресация стандартного потока ввода

Рассмотрим несколько примеров переадресации стандартного потока ввода. Чтобы из командной строки отправить пользователю электронное сообщение, которое находится в файле, следует направить файл программе `mail`. Посредством следующей команды пользователь `louise` получит сообщение, которое содержится в файле *contents.txt*:

```
$ mail louise < contents.txt
```

Переадресация вида `команда << разделитель` называется конструкцией "документ здесь". Более подробно мы поговорим о ней позже. Пока же рассмотрим общие принципы ее функционирования. Встречая в командной строке оператор <<, интерпретатор shell воспринимает все данные, вводимые с клавиатуры, как входной поток, пока в отдельной строке не будет введено слово-разделитель, указанное в командной строке после оператора. Разделителем может служить любое слово. Вот как, например, можно создать файл в режиме командной строки:

```
$ cat >> myfile << Пока
> Привет! Я работаю за терминалом $TERM
```

```
> и мое имя $LOGNAME.
> Пока!
> Пока

$ cat myfile
Привет! Я работаю за терминалом vt100
и мое имя dave.
Пока!
```

Признаком окончания ввода является слово "Пока" в отдельной строке, за которым нет никаких других символов. Вот почему предпоследняя строка не послужила командой окончания: после слова "Пока" стоит восклицательный знак.

5.7.3. Переадресация стандартного потока ошибок

При переадресации стандартного потока ошибок указывается дескриптор 2. Рассмотрим пример. Утилита `grep` ищет в файле *missiles* строку "trident":

```
$ grep "trident" missiles
grep: missiles: No such file or directory
```

Однако в текущем каталоге нет такого файла, и соответствующее сообщение об ошибке по умолчанию выводится на экран. Можно переслать все сообщения об ошибках в системную корзину (устройство */dev/null*):

```
$ grep "trident" missiles 2> /dev/null
```

Теперь никакие сообщения на экране отображаться не будут.

Подобный режим работы не всегда желателен. Часто сообщения об ошибках необходимо фиксировать в файле для последующего анализа. В следующей командной строке сообщения об ошибках пересылаются в файл *grep.err*:

```
$ grep "trident" missiles 2> grep.err
$ cat grep.err
grep: missiles: No such file or directory
```

Во многих случаях создается общий журнальный файл, в который добавляются сообщения об ошибках, поступающие от многих команд. При этом следует использовать оператор >>, чтобы не перезаписывать файл:

```
$ grep "LPSO" * 2>> account.err
$ grep "SILO" * 2>> account.err
```

5.7.4. Переадресация обоих выходных потоков

В одной командной строке можно последовательно переадресовывать как стандартный поток вывода, так и стандартный поток ошибок. Ниже приведен пример, в котором команда `cat` обрабатывает два файла, направляя вывод в файл *accounts.out*, а сообщения об ошибках — в файл *accounts.err*:

```
$ cat account_qtr.doc account_end.doc 1> accounts.out 2> accounts.err

$ cat accounts.out
AVBD 34HJ OUT
AVFJ 31KO OUT
...
```

```
$ cat accounts.err
cat: account_end.doc: No such file or directory
```

Просмотрев файл *accounts.err*, обнаруживаем, что исходного файла *account_end.doc* не существует.

5.7.5. Объединение выходных потоков в файле

Оператор `n>&m` позволяет перенаправить файл с дескриптором *n* туда, куда направлен файл с дескриптором *m*. Подобных операторов в командной строке может быть несколько, в этом случае они вычисляются слева направо. Рассмотрим пример:

```
$ cleanup > cleanup.out 2>&1
```

Здесь сценарий `cleanup` направляет все свои выходные данные (как поток вывода, так и поток ошибок) в файл *cleanup.out*.

В следующем примере все результаты работы команды `grep` направляются в файл *grep.out*:

```
$ grep "standard" * > grep.out 2>&1
```

5.8. Команда exec

Команда `exec` заменяет текущий интерпретатор shell указанной командой. Обычно она используется для того, чтобы закрыть текущий интерпретатор и запустить другой. Но у нее есть и другое применение. Например, команда вида

```
exec < файл
```

делает указанный файл стандартным входным потоком всех команд. Выполнять ее в интерактивном режиме нет смысла — она предназначена для использования в сценариях, чтобы все идущие после нее команды читали свои входные данные из файла. В этом случае в конце сценария обязательно должна стоять команда

```
exec <&-
```

которая закрывает стандартный входной поток (в данном случае файл). Подобный прием применяется преимущественно в сценариях, выполняющихся при выходе из системы. Более общий способ использования команды `exec` описан в следующем параграфе.

5.9. Применение дескрипторов файлов

Рассмотренная в предыдущем параграфе команда `exec < файл` не только назначает файл стандартным входным потоком всех команд сценария, но и перезаписывает указатель на файл с дескриптором 0 (*stdin*). Восстановить этот указатель можно будет только по завершении работы сценария. Если же в сценарии предполагается продолжить чтение данных с клавиатуры, то необходимо каким-то образом сохранить указатель на прежний входной поток. Ниже приведен небольшой сценарий, в котором демонстрируется, как это сделать.

```
$ cat f_desc
#!/bin/sh
```

```
# f_desc
exec 3<&0 0<stock.txt
read line1
read line2
exec 0<&3
echo $line1
echo $line2
```

Первая команда `exec` сохраняет указатель на стандартный входной поток (*stdin*) в файле с дескриптором 3 (допускается любое целое число в диапазоне от 3 до 9), а затем открывает файл *stock.txt* для чтения. Следующие две команды `read` читают из файла две строки текста. Вторая команда `exec` восстанавливает указатель на стандартный входной поток: теперь он связан с файлом *stdin*, а не *stock.txt*. Завершающие команды `echo` отображают на экране содержимое прочитанных строк, которые были сохранены в переменных `line1` и `line2`.

Предположим, файл *stock.txt* содержит такой текст:

```
$ cat stock.txt
Crayons Assorted 34
Pencils Light 12
```

Ниже показаны результаты работы сценария:

```
$ f_desc
Crayons Assorted 34
Pencils Light 12
```

В следующих главах мы рассмотрим более сложные примеры работы с дескрипторами файлов. В частности, будет показано, как с их помощью копировать текстовые файлы, не пользуясь командой `cp`.

5.10. Заключение

В процессе чтения книги вы встретите множество примеров переадресации. Механизм переадресации является важной частью интерпретатора shell, так как позволяет соединять команды с выходными и входными файлами, а также разделять потоки вывода и ошибок.

ГЛАВА 6

Порядок выполнения команд

При выполнении команды иногда требуется знать, как была выполнена другая команда. Предположим, необходимо скопировать все файлы из одного каталога в другой, а затем удалить исходный каталог. Прежде чем удалять исходный каталог, важно убедиться, что копирование прошло успешно. В противном случае содержимое исходного каталога может оказаться утерянным.

В этой главе рассматриваются следующие темы:

- выполнение команды в зависимости от результата выполнения другой команды;
- группирование команд.

Интерпретатор shell располагает операторами `&&` и `||`, которые группируют команды по принципу логического И/ИЛИ. Существуют также операторы `()` и `{}`, объединяющие заключенные в них команды в группу, выполняемую в текущем или порожденном интерпретаторе shell.

6.1. Оператор &&

Общий формат оператора `&&` таков:

```
команда1 && команда2
```

Эта инструкция обрабатывается следующим образом: правый операнд интерпретируется только тогда, когда левый операнд равен `TRUE`. Иными словами, вторая команда выполняется в том случае, если первая завершилась успешно.

Рассмотрим простой пример:

```
$ cp justice.doc justice.bak && echo "копирование прошло успешно"
копирование прошло успешно
```

Сообщение, заданное в команде `echo`, появилось на экране, значит, команда `cp` успешно выполнила копирование файла.

А вот более практичный пример:

```
$ mv /apps/bin /apps/dev/bin && rm -r /apps/bin
```

Каталог */apps/bin* с помощью команды `mv` перемещается в каталог */apps/dev/bin*. Если перемещение завершится удачно, то исходный каталог — */apps/bin* — будет удален.

В следующем примере команда `sort` сортирует содержимое текстового файла *quarter_end.txt*, записывая результат в файл *quarter.sorted*. Если запись прошла успешно, полученный файл выводится на печать с помощью команды `lp`.

```
$ sort quarter_end.txt > quarter.sorted && lp quarter.sorted
```

6.2. Оператор ||

Рассмотрим общий формат оператора ||:

```
команда1 || команда2
```

Эта инструкция обрабатывается следующим образом: правый операнд интерпретируется только тогда, когда левый операнд равен FALSE. Иными словами, вторая команда выполняется в том случае, если первая завершилась неуспешно.

Приведем простой пример, иллюстрирующий применение оператора ||:

```
$ cp wopper.txt oops.txt || echo "копирование не выполнено"
cp: wopper.txt: No such file or directory
копирование не выполнено
```

Операция копирования завершилась неудачно (исходного файла не существует), поэтому была выполнена команда echo.

В следующем примере из файла *acc.qtr* извлекаются первое и пятое поля, а результат помещается во временный файл *qtr.tmp*. Если по какой-то причине извлечь нужные данные не удастся, пользователь dave получит электронное сообщение.

```
$ awk '{print$1,$5}' acc.qtr > qtr.tmp || echo "Получить данные не удалось." | \
                                                    mail dave
```

6.3. Группирование команд с помощью скобок

Существует возможность объединить несколько команд в группу и выполнить ее как единое целое в текущем или порожденном интерпретаторе shell. Во втором случае создается копия текущего интерпретатора, а все изменения среды интерпретатора отменяются по завершении последней команды в группе.

Для выполнения группы команд в текущем интерпретаторе shell следует заключить их список в фигурные скобки, разделив команды точкой с запятой:

```
{команда1; команда2;...}
```

Чтобы выполнить группу команд в порожденном интерпретаторе shell, поступите аналогичным образом, но вместо фигурных скобок поставьте круглые скобки:

```
(команда1; команда2;...)
```

Сам по себе подобный метод применяется не часто. Обычно группа команд выполняется в составе более крупных конструкций с операторами && или ||. Результатом работы группы команд считается результат выполнения последней команды в группе.

Предположим, я выполняю сценарий comet, обрабатывающий файл *month_end.txt*, и в случае неудачного завершения сценария хочу прекратить работу интерпретатора shell. Простейший способ сделать это состоит в следующем:

```
$ comet month_end.txt || exit
```

Но можно также перед выходом из интерпретатора послать отвечающему за этот сценарий пользователю электронное сообщение:

```
$ comet month_end.txt || (echo "Привет! Твой сценарий не работает." | \
                          mail dave; exit)
```

Символ \ в конце первой строки означает, что команда будет продолжена в следующей строке.

Вспомним рассмотренный ранее пример, в котором команда `sort` сортирует содержимое текстового файла *quarter_end.txt*, записывая результат в файл *quarter.sorted* с последующим выводом его на печать:

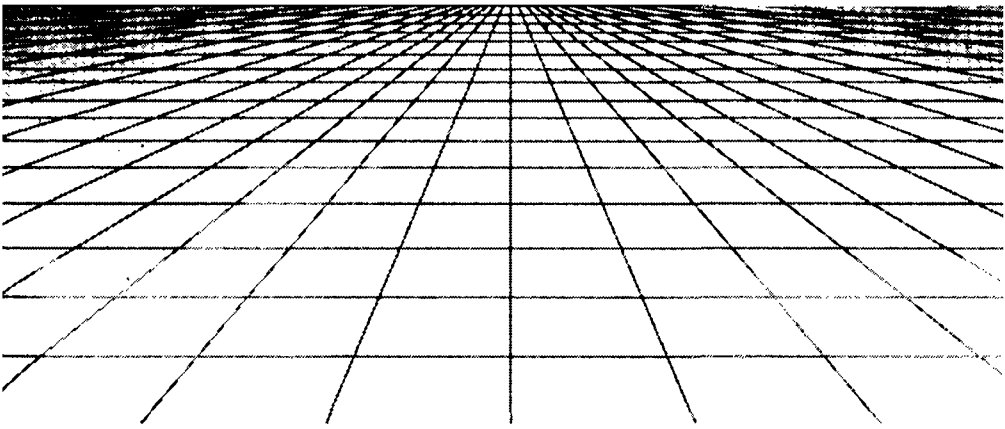
```
$ sort quarter_end.txt > quarter.sorted && lp quarter.sorted
```

Применив метод группировки команд, можно одновременно с печатью файла скопировать его в каталог */logs*:

```
$ sort quarter_end.txt > quarter.sorted && \
  (cp quarter.sorted /logs/quarter.sorted; lp quarter.sorted)
```

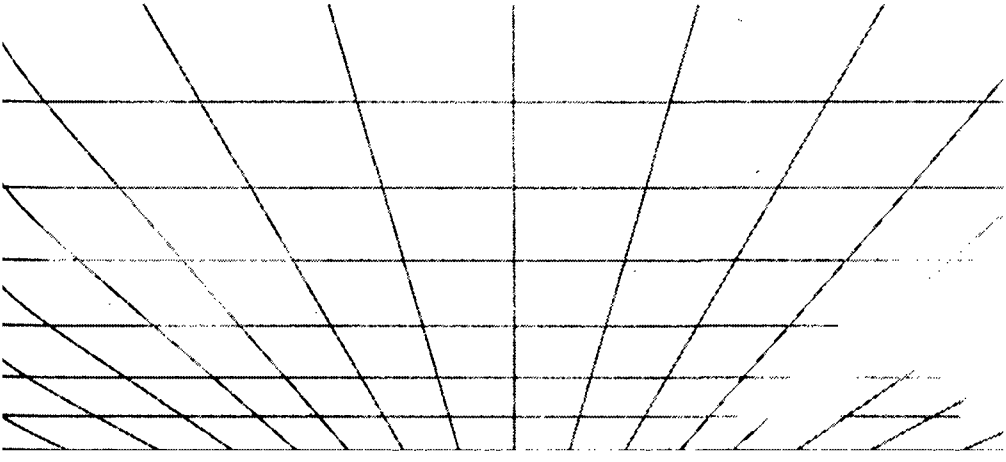
6.4. Заключение

При создании сложных инструкций большую роль играют операторы `&&` и `||`. Они позволяют выполнить указанную после них команду или группу команд только в том случае, если была (или не была) успешно выполнена предыдущая команда или группа команд.



ЧАСТЬ 2

Фильтрация текста



ГЛАВА 7

Регулярные выражения

При работе в UNIX или Linux часто используются регулярные выражения — мощное средство текстового поиска. Если, например, требуется найти слово, у которого первые два символа являются прописными буквами, а следующие четыре символа — цифрами, сформировать правильный шаблон поиска помогут регулярные выражения.

В этой главе рассматриваются следующие темы:

- создание шаблонов для поиска выражений, стоящих в начале или в конце строки;
- создание шаблонов для поиска символов, встречающихся неопределенное число раз;
- создание шаблонов для поиска специальных символов;
- создание шаблонов для поиска символов из указанного набора или диапазона;
- создание шаблонов для поиска символов, встречающихся указанное число раз подряд.

Регулярные выражения можно применять для фильтрации текстовых файлов или выходных данных команды либо сценария. Они представляют собой набор шаблонов, состоящих как из специальных, так и обычных символов.

Регулярные выражения в той или иной форме используются всеми основными текстовыми редакторами и утилитами, выполняющими фильтрацию текста. К сожалению, наборы поддерживаемых выражений несколько различаются от программы к программе, но существуют так называемые базовые регулярные выражения, которые во всех программах обрабатываются одинаково. Именно их мы и рассмотрим в настоящей главе. Единственное исключение — оператор `\{ \}`, который поддерживается в программах `sed` и `grep`, но не в `awk`.

В табл. 7.1 перечислены метасимволы и операторы, применяемые в базовых регулярных выражениях.

Таблица 7.1. Метасимволы и операторы базовых регулярных выражений

`^`	Соответствует началу строки
`$`	Соответствует концу строки
`[]`	Соответствует любому символу из числа заключенных в скобки; чтобы задать диапазон символов, укажите первый символ диапазона, дефис и последний символ (например, вместо шаблона `[12345]` можно ввести `[1-5]`)
`[^]`	Соответствует любому символу, кроме тех, что указаны в скобках
`\`	Отменяет специальное значение следующего за ним метасимвола
`.`	Соответствует любому отдельному символу

*	Указывает на то, что предыдущий шаблон встречается ноль или более раз; в программах `awk` и `egrep`, где используются расширенные регулярные выражения, существует два дополнительных оператора: `?` (означает, что предыдущий шаблон встречается не более одного раза) и + (означает, что предыдущий шаблон встречается один или более раз)
`\{n\}`	Указывает на то, что предыдущий шаблон встречается ровно *n* раз
`\{n,\}`	Указывает на то, что предыдущий шаблон встречается не менее *n* раз
`\{,m\}`	Указывает на то, что предыдущий шаблон встречается не более *m* раз
`\{n,m\}`	Указывает на то, что предыдущий шаблон встречается не менее *n* раз и не более *m* раз

7.1. Поиск одиночных символов с помощью метасимвола '.'

Метасимвол '.' соответствует любому одиночному символу. Если, например, требуется найти слово, начинающееся с подстроки "beg", после которой стоит произвольный символ, а за ним — символ 'n', задайте шаблон `beg.n`. Будут найдены такие слова, как "begin", "began" и т.д.

Данный метасимвол удобно применять при фильтрации результатов работы команды `ls -l` для поиска файлов, имеющих требуемые права доступа. Следующий шаблон соответствует файлам, выполнять которые могут все пользователи:

```
...x..x..x
```

Вот примеры отбора строк режима по этому шаблону:

```
drwxrwxrw- - не соответствует
-rw-rw-rw- - не соответствует
-rwxrwxr-x - соответствует
-rwxr-xr-x - соответствует
```

Предположим, выполняется фильтрация текстового файла. Необходимо найти в нем строки, состоящие из десяти символов, из которых пятый и шестой — "XC". Данная задача решается с помощью такого шаблона:

```
....XC....
```

Он означает, что первые четыре символа могут быть произвольными, следующие два — "XC", а последние четыре — тоже произвольные. Вот несколько примеров сравнения:

```
1234XC9088 - соответствует
4523XX9001 - не соответствует
0011XA9912 - не соответствует
9931XC3445 - соответствует
```

7.2. Поиск выражений в начале строки с помощью метасимвола '^'

Метасимвол '^' позволяет искать слова или символы, стоящие в начале строки. Например, благодаря шаблону `^d` можно отобрать из списка, выводимого командой `ls -l`, только те записи, которые соответствуют каталогам:

```
drwxrwxrw- - соответствует
-rw-rw-rw- - не соответствует
```

```
drwxrwxr-x – соответствует
-rwxr-xr-x – не соответствует
```

Вернемся к рассмотренному в предыдущем параграфе примеру фильтрации текстового файла. Шаблон `^001` соответствует строкам, начинающимся с символов "001". Результат его применения может быть таким:

```
1234XC9088 – не соответствует
4523XX9001 – не соответствует
0011XA9912 – соответствует
9931XC3445 – не соответствует
```

Для поиска строк, у которых в четвертой позиции от начала стоит символ '1', можно воспользоваться следующим шаблоном:

```
^...1
```

В результате получим:

```
1234XC9088 – не соответствует
4523XX9001 – не соответствует
0011XA9912 – соответствует
9931XC3445 – соответствует
```

Чтобы найти строки, начинающиеся с символов "comp", следует указать:

```
^comp
```

Давайте немного усложним этот шаблон. Предположим, после символов "comp" могут идти любые две буквы, но завершать последовательность должны символы "ing":

```
^comp..ing
```

Этот шаблон обеспечивает поиск таких слов, как "computing", "complaining" и т.д. Как показывает данный пример, в регулярном выражении можно сочетать различные шаблоны поиска.

7.3. Поиск выражений в конце строки с помощью метасимвола '$'

Метасимвол '$' предназначен для поиска слов или символов, находящихся в конце строки. Он указывается в конце шаблона. Предположим, требуется найти строки, заканчивающиеся словом "trouble". Эту задачу позволяет решить такой шаблон:

```
trouble$
```

Следующий шаблон соответствует пустой строке, не содержащей символов:

```
^$
```

А с помощью показанного ниже шаблона можно найти строки, включающие только один символ:

```
^.$
```

7.4. Поиск символов, встречающихся неопределенное число раз, с помощью метасимвола '*'

Метасимвол '*' означает, что предыдущий символ в регулярном выражении либо отсутствует, либо встречается произвольное число раз подряд (1, 2 и т.д.). Например, шаблон

```
compu*t
```

отвечает таким словам:

```
computer
computing
compuuuuute
```

А шаблон

```
10133*
```

соответствует следующему:

```
101333
10133
10134
```

7.5. Поиск специальных символов с помощью метасимвола '\'

Ряд символов, попадая в состав регулярных выражений, приобретает специальное значение. В общем случае специальными являются следующие символы:

```
$ . ' " * [ ] ^ | ( ) \ + ?
```

Когда требуется найти строку, содержащую один из таких символов, его необходимо "защитить" в шаблоне с помощью обратной косой черты, которая отменяет специальное значение следующего за ней метасимвола. Предположим, строка содержит символ '.', который, как известно, в регулярном выражении соответствует произвольному символу. Вот шаблон для него:

```
\.
```

Если необходимо найти файлы, допустим, с расширением *pas*, можно применить следующий шаблон:

```
\*\.pas
```

7.6. Поиск символов, входящих в заданный набор или диапазон

Шаблон `[]` соответствует списку или диапазону символов, указанных в квадратных скобках.

Символы в списке можно разделять запятыми. Это облегчает восприятие шаблона, хотя и не является обязательным требованием.

Для задания диапазона символов используется дефис (-). Слева от него указывается первый символ диапазона, а справа — последний. Предположим, необходимо найти символ, являющийся цифрой. Можно применить такой шаблон:

```
[0123456789]
```

Однако проще задать диапазон:

```
[0-9]
```

Следующий шаблон соответствует любой строчной букве:

```
[a-z]
```

Чтобы найти любую букву произвольного регистра, воспользуйтесь шаблоном

```
[A-Za-z]
```

Здесь формируется список из двух диапазонов: прописные буквы от 'A' до 'Z' и строчные буквы от 'a' до 'z'.

Представленный ниже шаблон соответствует любому алфавитно-цифровому символу:

```
[A-Za-z0-9]
```

Далее показан шаблон, предназначенный для поиска трехсимвольных комбинаций следующего типа: в начале находится буква 's', за ней может следовать любая прописная или строчная буква, а завершает последовательность буква 't':

```
s[a-zA-Z]t
```

Если же комбинация состоит только из букв нижнего регистра, воспользуйтесь таким шаблоном:

```
s[a-z]t
```

Чтобы найти слово "computer" независимо от того, расположено оно в начале предложения или нет, примените такой шаблон:

```
[Cc]omputer
```

Следующий шаблон соответствует слову "system", которое начинается с прописной или строчной буквы и за которым следует точка:

```
[S,s]ystem\.
```

Запятая в квадратных скобках поставлена для того, чтобы сделать шаблон удобным для зрительного восприятия.

Метасимвол '*', размещенный после квадратных скобок, указывает на то, что символы в скобках могут повторяться неопределенное число раз. Например, следующий шаблон соответствует любому слову:

```
[A-Za-z]*
```

Метасимвол '^' после открывающей квадратной скобки — это признак того, что шаблон соответствует любым символам, кроме указанных в скобках. Так, шаблон

```
[^a-zA-Z]
```

соответствует всем символам, кроме букв, а шаблон

```
[^0-9]
```

отвечает всем символам, которые не являются числами.

7.7. Поиск символов, встречающихся заданное число раз

Метасимвол '*' позволяет находить символы, встречающиеся несколько раз подряд, но число повторений при этом не определяется. Если же необходимо в процессе поиска учитывать точное количество последовательных вхождений символа в строку, следует применить шаблон \{ \}. Существует четыре варианта этого шаблона:

`шаблон\{n\}`	Соответствует шаблону, встречающемуся ровно *n* раз подряд
`шаблон\{n,\}`	Соответствует шаблону, встречающемуся не менее *n* раз подряд
`шаблон\{,m\}`	Соответствует шаблону, встречающемуся не более *m* раз подряд
`шаблон\{n,m\}`	Соответствует шаблону, встречающемуся не менее *n* и не более *m* раз подряд, где *n* и *m* — целые числа из интервала от 0 до 255

Представленный ниже шаблон соответствует последовательности из двух букв 'A', за которыми следует буква 'B':

```
A\{2\}B
```

В результате получим "AAB".

В следующем шаблоне задано, что буква 'A' встречается не менее четырех раз подряд:

```
A\{4,\}B
```

Возможные результаты поиска — "AAAAB" или "AAAAAAAB", но не "AAAB".

Поиск последовательности, в которой буква 'A' встречается от двух до четырех раз, выполняется по такому шаблону:

```
A\{2,4\}B
```

Будут найдены строки "AAB", "AAAB", "AAAAB", но не "AB" или "AAAAAB".

Вернемся к уже рассматривавшемуся примеру фильтрации текстового файла, фрагмент которого представлен ниже:

```
1234XC9088
4523XX9001
0011XA9912
9931XC3445
```

Допустим, требуется найти строки, в которых первые четыре символа — цифры, за ними идут символы "XX", а затем — еще четыре цифры. Решить данную задачу позволит такой шаблон:

```
[0-9]\{4\}XX[0-9]\{4\}
```

Применив этот шаблон к приведенному выше фрагменту, получим:

```
1234XC9088 - не соответствует
4523XX9001 - соответствует
0011XA9912 - не соответствует
9931XC3445 - не соответствует
```

7.8. Примеры

В табл. 7.2 приведен ряд дополнительных примеров использования регулярных выражений.

Таблица 7.2. Полезные регулярные выражения

`^the`	Соответствует строкам, которые начинаются символами "the"
`[Ss]igna[lL]`	Соответствует словам "signal", "signaL", "Signal" и "SignaL"
`[Ss]igna[lL]\.`	То же, что и в предыдущем случае, но слово, к тому же, должно завершаться точкой
`tty$`	Соответствует строкам, которые завершаются символами "tty"
`^USER$`	Соответствует слову "USER", которое является единственным в строке
`\.`	Соответствует точке
`^d..x..x..x`	Соответствует каталогам с установленным правом на выполнение для владельца, группы и других пользователей
`^[^l]`	Исключает из списка файлов записи, соответствующие символическим ссылкам
`000*`	Находит строки, содержащие два или больше нулей подряд
`[iI]`	Соответствует прописной и строчной букве 'I'
`[iInN]`	Соответствует прописным и строчным буквам 'I' и 'n'
`^$`	Соответствует пустой строке
`^.*$`	Соответствует строке, состоящей из любого числа символов
`^......$`	Соответствует строке, состоящей из шести символов
`[a-zA-Z]`	Соответствует любой прописной или строчной букве
`[a-z][a-z]*`	Соответствует по крайней мере одной строчной букве
`[^0-9\$]`	При рассмотрении цифры и знаки доллара не учитываются
`[^0-9A-Za-z]`	При рассмотрении не учитываются буквы и цифры
`[123]`	Соответствует цифрам 1, 2 и 3
`[Dd]evice`	Соответствует словам "Device" и "device"
`De..ce`	Соответствует слову, в котором первые два символа — "De", за ними идут любые два символа, а затем — символы "ce"

`\^q`	Соответствует символам "^q"
`^.$`	Соответствует строке, содержащей только один символ
`^\.[0-9][0-9]`	Соответствует строке, которая начинается с точки и двух цифр
`"Device"`	Соответствует слову "Device"
`De[Vv]ice\.`	Соответствует слову "DeVice" или "Device", после которого стоит точка
`[0-9]\{2\}-[0-9]\{2\}-[0-9]\{4\}`	Соответствует шаблону даты в формате *dd-mm-yyyy*
`[0-9]\{3\}\.[0-9]\{3\}\.[0-9]\{3\}\.[0-9]\{3\}`	Соответствует шаблону IP-адреса в формате *nnn.nnn.nnn.nnn*

7.9. Заключение

Регулярные выражения и методы работы с ними — важный аспект shell-программирования. Знакомство этой с методикой позволит повысить качество создаваемых сценариев, так как во многих случаях три-четыре команды фильтрации текста можно заменить одной командой с регулярным выражением.

В следующих главах мы рассмотрим примеры применения регулярных выражений в программах `grep`, `sed` и `awk`.

ГЛАВА 8

Семейство команд grep

Команда `grep` (global regular expression print — печать глобальных регулярных выражений) является наиболее известным инструментальным средством в UNIX и Linux. Она выполняет в текстовых файлах или стандартном входном потоке поиск выражений, соответствующих шаблону, с последующим отображением результата на экране. Команда `grep` может работать как с базовыми, так и с расширенными регулярными выражениями. Существует три разновидности этой команды:

- `grep` — стандартный вариант, которому уделено основное внимание в данной главе.
- `egrep` — работает с расширенными регулярными выражениями (не поддерживает только оператор `\{ \}`).
- `fgrep` — быстрый вариант команды `grep`. Вместо поиска выражений, соответствующих шаблону, выполняет поиск фиксированных строк из указанного списка. Пусть вас не вводит в заблуждение слово "быстрый". На самом деле это наиболее медленная из команд семейства `grep`.

Конечно, хотелось бы, чтобы существовала только одна, универсальная, команда `grep`, и с этой ролью, в принципе, справляется GNU-версия `grep`. К сожалению, нет единого способа задания аргументов для всех трех разновидностей команды `grep`, к тому же скорость их работы заметно различается.

В этой главе рассматриваются следующие темы:

- параметры команды `grep`;
- применение регулярных выражений в команде `grep`;
- особенности поиска алфавитно-цифровых символов.

В предыдущей главе указывалось, что в данной книге описываются только базовые регулярные выражения. Но мы все же рассмотрим несколько примеров употребления расширенных регулярных выражений во время знакомства с командой `egrep`. Основное же внимание будет уделено команде `grep`. Все применяемые в ней шаблоны могут без изменений использоваться и в команде `egrep`.

Прежде чем вы приступите к прочтению главы, создайте представленный ниже файл с именем *data.f*, содержащий информацию о заказах товаров. Структура записей этого файла такова:

1-й столбец — код города;

2-й столбец — код месяца, когда был сделан заказ;

3-й столбец — код заказа, включающий год, когда он был сделан;

4-й столбец — код товара;

5-й столбец — цена за единицу товара;

6-й столбец — код фирмы;

7-й столбец — количество заказанного товара.

```
$ cat data.f
48   dec   3BC1997    LPSX   68.00   LVX2A   138
483  sept  5AP1996    USP    65.00   LVX2C   189
47   oct   3ZL1998    LPSX   43.00   KVM9D   512
219  dec   2CC1999    CAD    23.00   PLV2C   68
484  nov   7PL1996    CAD    49.00   PLV2C   234
483  may   5PA1998    USP    37.00   KVM9D   644
216  sept  3ZL1998    USP    86.00   KVM9E   234
```

Разделителем полей является символ табуляции.

8.1. Команда grep

Общий формат команды `grep` таков:

```
grep [параметры] базовое_регулярное_выражение [файл]
```

В качестве регулярного выражения может быть указана обычная строка. Если не задан файл, текст берется из стандартного входного потока.

8.1.1. Употребление кавычек

Строку, которая задана в качестве регулярного выражения и состоит из нескольких слов, следует заключить в двойные кавычки. В противном случае первое слово строки будет воспринято как образец поиска, а все остальные слова будут считаться именами файлов. В итоге отобразится сообщение о том, что указанные файлы не найдены.

Если образец поиска состоит из какой-нибудь системной переменной, например `$PATH`, рекомендуется тоже взять ее в двойные кавычки. Это связано с тем, что, прежде чем передавать аргументы команде `grep`, интерпретатор shell выполняет подстановку переменных, и команда `grep` получает *значение* переменной, которое может содержать пробелы. В этом случае будет выдано то же сообщение об ошибке, о котором говорилось в предыдущем абзаце.

Шаблон поиска должен быть заключен в одинарные кавычки, если в состав регулярного выражения входят метасимволы.

8.1.2. Параметры команды grep

Ниже перечислены основные параметры команды `grep`:

- `-c` Задает отображение только числового значения, указывающего, сколько строк соответствуют шаблону
- `-i` Дает указание игнорировать регистр символов
- `-h` Подавляет вывод имен файлов, включающих найденные строки (по умолчанию в выводе команды `grep` каждой строке предшествует имя файла, в котором она содержится)
- `-l` Задает отображение только имен файлов, содержащих найденные строки
- `-n` Задает нумерацию выводимых строк
- `-s` Подавляет вывод сообщений о несуществующих или нетекстовых файлах
- `-v` Задает отображение строк, не соответствующих шаблону

8.1.3. Поиск среди нескольких файлов

Если в текущем каталоге требуется найти последовательность символов "sort" во всех файлах с расширением *doc*, выполните такую команду:

```
$ grep sort *.doc
```

Следующая команда осуществляет поиск фразы "sort it" во всех файлах текущего каталога:

```
$ grep "sort it" *
```

8.1.4. Определение числа строк, в которых найдено совпадение

Опция `-c` позволяет узнать, сколько строк соответствуют заданному шаблону. Это может оказаться полезным в том случае, когда команда `grep` находит слишком много строк, и их лучше вывести в файл, а не на экран. Рассмотрим пример:

```
$ grep -c "48" data.f
4
```

Команда `grep` возвращает число 4. Это означает, что в файле *data.f* обнаружены 4 строки, содержащие последовательность символов "48". Следующая команда отображает эти строки:

```
$ grep "48" data.f
48   dec   3BC1997    LPSX   68.00   LVX2A   138
483  sept  5AP1996    USP    65.00   LVX2C   189
484  nov   7PL1996    CAD    49.00   PLV2C   234
483  may   5PA1998    USP    37.00   KVM9D   644
```

8.1.5. Вывод номеров строк

С помощью опции `-n` выводимые строки можно пронумеровать. В результате вы сможете с легкостью устанавливать, в какой позиции файла находится требуемая строка. Например:

```
$ grep -n "48" data.f
1:48   dec   3BC1997    LPSX   68.00   LVX2A   138
2:483  sept  5AP1996    USP    65.00   LVX2C   189
5:484  nov   7PL1996    CAD    49.00   PLV2C   234
6:483  may   5PA1998    USP    37.00   KVM9D   644
```

Номера строк отображаются в первом столбце.

8.1.6. Поиск строк, не соответствующих шаблону

Благодаря опции `-v` можно отобрать те строки, которые не соответствуют шаблону. Следующая команда извлекает из файла *data.f* строки, не содержащие последовательность символов "48":

```
$ grep -v "48" data.f
47   oct   3ZL1998    LPSX   43.00   KVM9D   512
219  dec   2CC1999    CAD    23.00   PLV2C   68
216  sept  3ZL1998    USP    86.00   KVM9E   234
```

8.1.7. Поиск символов на границе слов

Вы, наверное, заметили, что при поиске строк, содержащих последовательность символов "48", были найдены строки заказов с кодом города не только 48, но также 483 и 484. Если необходимо найти заказ, у которого код города равен 48, добавьте в шаблон поиска символ табуляции:

```
$ grep "48<tab>" data.f
48   Dec   3BC1997    LPSX   68.00   LVX2A   138
```

Здесь запись <tab> означает нажатие клавиши [Tab].

Если не известно, какой пробельный символ является разделителем полей в файле, можно воспользоваться регулярным выражением \ > — *признаком конца слова*:

```
$ grep '48\>' data.f
48   Dec   3BC1997    LPSX   68.00   LVX2A   138
```

8.1.8. Игнорирование регистра символов

По умолчанию команда grep чувствительна к изменению регистра символов. Чтобы провести поиск без учета регистра, воспользуйтесь опцией -i. В файле *data.f* обозначение месяца Sept встречается как в верхнем, так и в нижнем регистре. Поэтому для отбора строк обоих видов следует применить такую команду:

```
$ grep -i "sept" data.f
483  Sept  5AP1996    USP    65.00   LVX2C   189
216  sept  3ZL1998    USP    86.00   KVM9E   234
```

8.2. Команда grep и регулярные выражения

С помощью регулярных выражений можно задавать более сложные критерии фильтрации информации. При работе с регулярными выражениями следует заключать шаблон поиска в одинарные кавычки. Это позволит защитить все встречающиеся в нем специальные символы от интерпретатора shell, который в противном случае может "перехватывать" их у команды grep.

8.2.1. Выбор символов из списка

В предыдущей главе мы уже отмечали, что с помощью оператора [] можно задать диапазон или список символов, включаемых в шаблон поиска. Предположим, требуется извлечь из файла *data.f* строки заказов, сделанных в городах, код которых равен 483 или 484. Поставленную задачу решает следующая команда:

```
$ grep '48[34]' data.f
483  Sept  5AP1996    USP    65.00   LVX2C   189
484  nov   7PL1996    CAD    49.00   PLV2C   234
483  may   5PA1998    USP    37.00   KVM9D   644
```

8.2.2. Инверсия шаблона с помощью метасимвола '^'

Следующая команда находит в файле *data.f* строки, не начинающиеся с цифры 4 или 8:

```
$ grep '^[^48]' data.f
219  dec   2CC1999    CAD    23.00   PLV2C   68
216  sept  3ZL1998    USP    86.00   KVM9E   234
```

Символ '^', заключенный в квадратные скобки, говорит о том, что шаблон соответствует любому символу, кроме цифр 4 и 8. Символ '^' в начале шаблона — это признак того, что поиск производится с начала каждой строки.

8.2.3. Шаблон, соответствующий любому символу

Предположим, в файле *data.f* требуется найти коды фирм, которые начинаются на букву 'K' и заканчиваются буквой 'D'. Реализуется это следующим образом:

```
$ grep 'K...D' data.f
47   Oct   3ZL1998    LPSX   43.00   KVM9D   512
483  may   5PA1998    USP    37.00   KVM9D   644
```

Данный шаблон рассчитан на то, что коды фирм в файле состоят из пяти символов.

Ниже представлена небольшая вариация вышеприведенного примера. На этот раз осуществляется поиск всех кодов со следующей структурой: первые два символа — буквы в верхнем регистре, далее следуют два произвольных символа, а завершает последовательность буква 'C':

```
$ grep '[A-Z][A-Z]..C' data.f
483  Sept  5AP1996    USP    65.00   LVX2C   189
219  dec   2CC1999    CAD    23.00   PLV2C   68
484  nov   7PL1996    CAD    49.00   PLV2C   234
```

8.2.4. Поиск по дате

Представленная ниже команда находит все заказы, которые были сделаны в 1996 или 1998 году и коды которых начинаются с цифры 5:

```
$ grep '5..199[68]' data.f
483  Sept  5AP1996    USP    65.00   LVX2C   189
483  may   5PA1998    USP    37.00   KVM9D   644
```

Структура используемого здесь шаблона такова: первым символом является цифра 5, за ней следует два произвольных символа, затем число 199, а последним символом может быть либо цифра 6, либо цифра 8.

Поиск всех заказов, сделанных в 1998 году, выполняется посредством команды:

```
$ grep '[0-9]\{3\}8' data.f
47   Oct   3ZL1998    LPSX   43.00   KVM9D   512
483  may   5PA1998    USP    37.00   KVM9D   644
216  sept  3ZL1998    USP    86.00   KVM9E   234
```

Примененный в этом примере шаблон означает: найти любую последовательность из трех цифр, за которой идет цифра 8.

8.2.5. Комбинированные диапазоны

Допустим, необходимо найти строки, в которых код города имеет следующий формат: первым символом является произвольная цифра, второй символ выбирается из диапазона от 0 до 5, а третий символ принадлежит диапазону от 0 до 6. Воспользуемся следующей командой:

```
$ grep '[0-9][0-5][0-6]' data.f
47   Oct   3ZL1998    LPSX   43.00   KVM9D   512
```

```
484  nov  7PL1996    CAD    49.00   PLV2C   234
483  may  5PA1998    USP    37.00   KVM9D   644
216  sept 3ZL1998    USP    86.00   KVM9E   234
```

Как видите, отображается больше информации, чем нам необходимо. Значит, в шаблоне поиска недостаточно уточнен критерий отбора информации. Очевидно, следует указать, что поиск нужно начинать в начале строки. Для этого применим метасимвол '^':

```
$ grep '^[0-9][0-5][0-6]' data.f
216  sept 3ZL1998    USP    86.00   KVM9E   234
```

8.2.6. Поиск повторяющихся последовательностей

Если необходимо найти какую-либо строку, которая содержит цифру 4, повторенную минимум дважды, задайте такую команду:

```
$ grep '4\{2,\}' data.f
483  may  5PA1998    USP    37.00   KVM9D   644
```

Запятая указывает, что предыдущий символ встречается не менее двух раз.

Вот как можно найти все записи, содержащие по крайней мере три девятки:

```
$ grep '9\{3,\}' data.f
219  dec  2CC1999    CAD    23.00   PLV2C   68
```

Иногда точное число повторений символа не известно. В таких случаях окажется полезной команда со следующей структурой:

```
$ grep '8\{2,6}3' myfile

83     - не соответствует
888883 - соответствует
8884   - не соответствует
88883  - соответствует
```

Здесь задан поиск строк, в которых цифра 8 встречается от двух до шести раз подряд и предшествует цифре 3.

8.2.7. Выбор из нескольких шаблонов

Опция -E позволяет использовать в команде grep синтаксис расширенных регулярных выражений. Предположим, необходимо найти все заказы с кодами городов 216 или 219. В этом случае можно воспользоваться метасимволом '|', задающим выбор из двух шаблонов:

```
$ grep -E '219|216' data.f
219  dec  2CC1999    CAD    23.00   PLV2C   68
216  sept 3ZL1998    USP    86.00   KVM9E   234
```

8.2.8. Поиск пустых строк

Для поиска в файле пустых строк можно составить шаблон из метасимволов '^' и '$':

```
$ grep '^$' myfile
```

8.2.9. Поиск специальных символов

Если шаблон поиска включает какой-либо метасимвол в качестве литерала, т.е. обычного символа, следует поставить перед ним обратную косую черту, чтобы отменить его специальную интерпретацию. Предположим, требуется отыскать все строки, содержащие точку. Эту задачу решает такая команда:

```
$ grep '\.' myfile
```

Следующая команда ищет в файле *myfile* двойные кавычки:

```
$ grep '\" ' myfile
```

А вот эта команда отбирает в листинге команды `ls -l` запись, соответствующую файлу *control.conf*:

```
$ ls -l | grep 'control\.conf'
```

8.2.10. Поиск имен файлов, соответствующих заданному формату

В моей системе применяется следующий формат наименования файлов с документами: до шести символов, расположенных в начале, являются буквами нижнего регистра, далее следует точка, а завершают последовательность два символа верхнего регистра. Если требуется найти имена файлов подобного типа, записанные в файле *filename.deposit*, следует применить такую команду:

```
$ grep '[a-z]\{1,6\}\.[A-Z]\{1,2\}' filename.deposit

yrend.AS - соответствует
mothdf   - не соответствует
soa.PP   - соответствует
qp.RR    - соответствует
```

8.2.11. Поиск IP-адресов

Администратору DNS-сервера приходится поддерживать большое количество IP-адресов, относящихся к различным сетям. В моей системе файл *ipfile* может содержать свыше 200 адресов. Мне часто приходится выполнять поиск всех адресов в формате "*nnn.nnn.*" (т.е. адресов, содержащих две трехзначные последовательности, оканчивающиеся точкой). Для этой цели я пользуюсь следующей командой:

```
$ grep '[0-9]\{3\}\.[0-9]\{3\}\.' ipfile
```

8.2.12. Поиск строк с использованием подстановочных знаков

Предположим, имеется такой файл:

```
$ cat testfile
looks
likes
looker
long
```

Следующая команда находит в нем слова, начинающиеся с буквы 'l', после которой идет произвольное число символов, а за ними — буква 's':

```
$ grep 'l.*s' testfile
looks
likes
```

Показанная ниже команда отбирает слова, начинающиеся с буквы 'l', после которой идет произвольное число символов, затем — буква 'k', а после нее — еще один символ:

```
$ grep 'l.*k.' testfile
looks
likes
```

Следующая команда находит слова, в которых буква 'o' встречается не менее двух раз подряд:

```
$ grep 'ooo*' testfile
looks
```

Если требуется найти слово, которое стоит в конце строки, воспользуйтесь командой следующего вида:

```
$ grep 'device$' *
```

Эта команда ищет во всех файлах текущего каталога строки, завершающиеся словом "device".

8.3. Классы символов

Команда `grep` поддерживает целый ряд предопределенных диапазонов символов, называемых *классами* (табл. 8.1). Обозначение класса состоит из открывающей квадратной скобки, двоеточия, собственно имени класса, двоеточия и закрывающей квадратной скобки. Поскольку класс представляет собой диапазон, обозначение класса дополнительно должно заключаться в квадратные скобки.

Таблица 8.1. Основные классы символов и эквивалентные им регулярные выражения

Класс	Эквивалентное регулярное выражение
`[:upper:]`	[A-Z]
`[:lower:]`	[a-z]
`[:digit:]`	[0-9]
`[:alnum:]`	[0-9a-zA-Z]
`[:space:]`	символы пробела
`[:alpha:]`	[a-zA-Z]

Отличие классов от эквивалентных регулярных выражений, приведенных в табл. 8.1, состоит в том, что класс включает не только символы английского алфавита из стандартной таблицы ASCII-кодов, но также символы того национального языка, который установлен в данной конкретной системе.

Рассмотрим несколько примеров на базе нашего файла *data.f*. Предположим, требуется найти все коды заказов, которые содержат цифру 5, сопровождаемую по крайней мере двумя буквами в верхнем регистре:

```
$ grep '5[[:upper:]][[:upper:]]' data.f
483   Sept   5AP1996     USP     65.00    LVX2C    189
483   may    5PA1998     USP     37.00    KVM9D    644
```

Вот как осуществляется поиск всех кодов товара, которые оканчиваются буквой 'P' или 'D':

```
$ grep '[[:upper:]][[:upper:]][PD]' data.f
483   Sept   5AP1996     USP     65.00    LVX2C    189
219   dec    2CC1999     CAD     23.00    PLV2C    68
484   nov    7PL1996     CAD     49.00    PLV2C    234
483   may    5PA1998     USP     37.00    KVM9D    644
216   sept   3ZL1998     USP     86.00    KVM9E    234
```

8.4. Дополнительные примеры использования команды grep

В следующих примерах команда `grep` принимает по каналу результаты работы других команд, фильтруя их надлежащим образом.

8.4.1. Фильтрация списка файлов

Если требуется извлечь из списка файлов текущего каталога только подкаталоги, воспользуйтесь следующей командой:

```
$ ls -l | grep '^d'
```

Представленная ниже команда выбирает из списка файлов текущего каталога только те записи, которые относятся к файлам, а не к подкаталогам:

```
$ ls -l | grep '^[^d]'
```

Выполнить поиск каталогов, в которых установлены биты поиска для группы и других пользователей, позволяет такая команда:

```
$ ls -l | grep '^d.....x..x'
```

8.4.2. Подавление вывода сообщений об ошибках

Допустим, вы хотите найти запись пользователя `louise` в системном файле паролей:

```
$ grep louise /etc/passwd
louise:1xAL6GW9G.ZyY:501:501:Accounts Sect1C:/home/accts/louise:
/bin/sh
```

Не исключена возможность, что вы забудете, как называется этот файл. В таком случае воспользуйтесь следующей командой:

```
$ grep louise /etc/password
grep: /etc/password: No such file or directory
```

Команда grep выводит сообщение об ошибке, в которой говорится о том, что указанного файла не существует. Можно попробовать провести поиск во всех файлах каталога */etc*:

```
$ grep louise /etc/*
```

Однако в результате будут выведены многочисленные сообщения об ошибках, гласящие, что к определенным файлам доступ запрещен, а некоторые элементы каталога являются подкаталогами и поиск в них не ведется.

В подобной ситуации можно воспользоваться опцией -s, которая подавляет вывод сообщений об ошибках:

```
$ grep -s louise /etc/*
```

Если ваша версия команды grep не поддерживает данную опцию, воспользуйтесь следующей командой:

```
$ grep louise /etc/* 2> /dev/null
```

Эта команда направляет поток ошибок (2>) в системную корзину (устройство */dev/null*). На жаргоне системных администраторов это устройство называется битодробилкой.

8.4.3. Фильтрация списка процессов

Совместное применение команд grep и ps a позволяет выяснить, выполняется ли в системе некоторый процесс. Опция a команды ps задает вывод списка всех процессов, включая процессы других пользователей. Например, следующая команда проверяет, выполняется ли в данный момент процесс named:

```
$ ps a | grep named
211  ?    S    4.56  named
303  3    S    0.00  grep named
```

Выводимый результат включает также саму команду grep, поскольку она создает процесс, выполняющий фильтрацию текста в канале, и команда ps распознает этот процесс. Чтобы исключить команду grep из результата, задайте дополнительную команду grep с опцией -v:

```
$ ps ax | grep named | grep -v "grep"
211  ?    S    4.56  named
```

8.5. Команда egrep

Команда egrep (extended grep) воспринимает как базовые, так и расширенные регулярные выражения. Одной из привлекательных ее особенностей является возможность сохранения шаблонов поиска в файле. Подключается этот файл с помощью опции -f. Рассмотрим пример:

```
$ cat grepstrings
484
47
$ egrep -f grepstrings data.f
```

В этом случае в файле *data.f* осуществляется поиск записей, которые содержат последовательность символов "484" или "47". Создание файла шаблонов удобно в том случае, когда шаблонов очень много и вводить их в командной строке затруднительно. Если шаблонов немного, воспользуйтесь метасимволом '|', который позволяет сделать выбор между несколькими шаблонами. Например, следующая команда ищет в файле *data.f* записи, содержащие последовательность символов "3ZL" или "2CC":

```
$ egrep '3ZL|2CC' data.f
47   Oct   3ZL1998    LPSX   43.00   KVM9D   512
219  dec   2CC1999    CAD    23.00   PLV2C   68
216  sept  3ZL1998    USP    86.00   KVM9E   234
```

Метасимвол '|' можно применять более одного раза. Например, чтобы узнать, зарегистрированы ли в системе пользователи `louise`, `matty` и `pauline`, выполните команду `who` и направьте результаты ее работы команде `egrep`:

```
$ who | egrep 'louise|matty|pauline'
louise   pty8
matty    tty02
pauline  pty2
```

Круглые скобки позволяют представить выражение с несколькими шаблонами как один шаблон. Так, с помощью представленной ниже команды можно найти в текущем каталоге файлы, в которых встречаются слова из ряда "shutdown", "shutdowns", "reboot" и "reboots":

```
$ egrep '(shutdown|reboot)s?' *
```

8.6. Заключение

В настоящей главе продемонстрированы лишь некоторые из многочисленных возможностей команды `grep`, которая является универсальным инструментом фильтрации, очень популярным среди пользователей UNIX и Linux. Существует несколько ее разновидностей, которые в некоторых системах заменены единой GNU-командой `grep`. Как будет показано далее в этой книге, команда `grep` также является важным инструментом shell-программирования, используемым в связке с другими утилитами UNIX.

ГЛАВА 9

Утилита awk

При форматировании отчетов и извлечении информации из больших текстовых файлов неоценимую помощь оказывает утилита awk, которая обладает мощными средствами обработки текста. Как показывает опыт, среди всех инструментов фильтрации, имеющихся в интерпретаторе shell, труднее всего освоить работу именно с awk. Причина этого явления не ясна. Может быть, дело в синтаксисе утилиты или не совсем понятных сообщениях об ошибках, таких как `bailing out` и `awk: cmd. line:`. Подобные сообщения довольно часто встречаются при программировании на языке awk. Именно так — *язык* awk, поскольку это совершенно самостоятельный язык программирования. Возможно, изучать его нелегко, но его совместное применение с другими инструментальными средствами, такими как команда grep и редактор sed, позволяет значительно упростить программирование в интерпретаторе shell.

В главе не делается попытка изучить все возможности утилиты awk и не рассматриваются методы написания сложных сценариев на языке awk, так как для этого потребовалась бы отдельная книга. Мы научимся создавать эффективные однострочные команды и небольшие сценарии, выполняющие фильтрацию текстовых файлов и строк. Итак, в этой главе рассматриваются следующие темы:

- выборка текстовых полей;
- использование регулярных выражений;
- сравнение текстовых полей;
- передача параметров утилите awk;
- базовые команды и сценарии awk.

Название утилиты awk составлено из начальных букв фамилий разработчиков языка: Ахо (Aho), Вайнбергера (Weinberger) и Кернигана (Kernighan). Существуют также утилиты nawk и gawk, обладающие усовершенствованными возможностями обработки текста. Но эти утилиты здесь не рассматриваются.

Основная задача утилиты awk заключается в просмотре текстового файла или строки с целью нахождения в них информации, соответствующей заданному критерию отбора. После выборки нужных данных можно применить к ним функции, выполняющие обработку текста. Сложные сценарии awk обычно применяются для форматирования отчетов на базе текстовых файлов.

9.1. Вызов awk

Вызвать утилиту awk можно тремя способами. Первый заключается в передаче ей требуемых команд непосредственно в командной строке:

```
awk [-F разделитель_полей] 'сценарий' входной_файл...
```

В одинарных кавычках указывается список инструкций языка awk. Именно этому способу отдается преимущество в примерах настоящей главы.

Задавать разделитель полей с помощью опции -F не обязательно, так как по умолчанию утилита awk использует для этих целей пробел. Но, например, в файле */etc/passwd* поля отделяются друг от друга двоеточием. В данном случае вызов утилиты выглядит так:

```
awk -F: 'сценарий' входной_файл...
```

Второй способ вызова утилиты awk состоит в создании отдельного файла сценария, содержащего список инструкций awk. При этом утилита awk указывается в первой строке сценария в качестве интерпретатора команд. Созданный файл делается исполняемым и может быть вызван из командной строки.

Согласно третьему способу все инструкции awk помещаются в отдельный файл, после чего осуществляется вызов этого файла:

```
awk -f файл_сценария входной_файл...
```

Опция -f свидетельствует о том, что инструкции awk содержатся в файле сценария.

Утилита awk анализирует информацию, содержащуюся в одном или нескольких входных файлах.

9.2. Сценарии

Сценарий awk — это набор инструкций, состоящих из шаблонов и связанных с ними процедур. Когда утилита просматривает записи входного файла, она проверяет, установлена ли опция -F или переменная FS (о ней мы поговорим ниже), задающие разделители полей записи. По умолчанию в качестве разделителя принят пробел. При обнаружении символа новой строки прочитанная строка классифицируется как запись, и к ней применяются инструкции сценария. Процесс чтения записей продолжается до тех пор, пока не будет обнаружен признак конца файла.

В табл. 9.1 приведен образец входного файла и продемонстрировано, как утилита awk его анализирует. Утилита последовательно просматривает строки файла. Отыскав первый символ-разделитель, она помечает все предыдущие символы как поле номер 1. Символы между первым и вторым разделителями обозначаются как поле номер 2 и т.д. Процесс анализа завершается при обнаружении символа новой строки, который по умолчанию считается признаком конца записи. После этого содержимое полей сбрасывается и утилита переходит к следующей строке файла.

Таблица 9.1. Образец анализа входного файла

Поле 1	Разделитель	Поле 2	Разделитель	Поле 3	Разделитель	Поле 4 и символ новой строки
P.Bunny (запись 1)	#	02/99	#	48	#	Yellow\n
J.Troll (запись 2)	#	07/99	#	4842	#	Brown-3\n

9.2.1. Шаблоны и процедуры

Инструкция awk состоит из шаблона и связанной с ним процедуры, причем количество инструкций в сценарии awk может быть очень велико. Шаблонная часть уточняет, когда следует выполнять инструкцию либо к каким входным данным она должна применяться. Процедура определяет порядок обработки данных.

Шаблоном может служить любая условная или составная конструкция либо регулярное выражение. Существует также два специальных шаблона: BEGIN и END. Шаблон BEGIN применяется для инициализации переменных и создания заголовков отчета. Связанная с ним процедура выполняется перед началом обработки входного файла. Шаблон END употребляется для вывода итоговых данных и выполнения инструкций по завершении обработки файла. Если никакой шаблон не указан, процедура *выполняется для каждой записи из входного файла*.

Тело процедуры заключается в фигурные скобки. Чаще всего процедура осуществляет вывод информации на экран, но она может также содержать операторы присваивания, управляющие конструкции и встроенные функции. Если процедура не задана, утилита awk выводит на экран все содержимое записи, соответствующей шаблону.

9.2.2. Работа с полями и записями

На поля текущей записи можно ссылаться следующим образом: $1, $2 ... $n. Этот метод называется идентификацией полей. Подобная схема обозначений значительно облегчает работу с полями. Например, в качестве ссылки на первое и третье поля достаточно указать:

```
$1, $3
```

Обратите внимание на то, что идентификаторы полей разделяются запятой. Чтобы сослаться на все поля записи, содержащей пять полей, можно указать:

```
$1, $2, $3, $4, $5
```

Однако проще воспользоваться идентификатором $0, который служит для обозначения всех полей текущей записи.

Когда в процессе просмотра утилита awk встречает символ новой строки, считается, что найден конец записи, после чего выполняется переход к новой строке, анализ новой записи и инициализация полей.

Чтобы вывести на экран содержимое записи, задайте в теле процедуры команду print со списком нужных полей.

Тестовый файл

Прежде чем мы перейдем к практической части, создайте файл *grade.txt*, на котором основано большинство примеров настоящей главы. Файл этот должен содержать несколько записей из локальной базы данных секции каратистов.

```
$ cat grade.txt
M.Tansley     05/99     48311     Green     8     40     44
J.Lulu        06/99     48317     green     9     24     26
P.Bunny       02/99     48        Yellow    12    35     28
J.Troll       07/99     4842      Brown-3   12    26     26
L.Tansley     05/99     4712      Brown-2   12    30     28
```

Назначение полей таково: 1 — имя; 2 — дата получения пояса; 3 — порядковый номер ученика; 4 — полученный пояс; 5 — возраст; 6 — текущий рейтинг; 7 — максимальное количество рейтинговых очков, которое можно было получить за участие в прошедшем соревновании.

Поля разделяются пробелами, поэтому при вызове утилиты `awk` можно не указывать опцию `-F`.

Сохранение выходных данных

В качестве небольшого отступления напомним, что существует два способа, позволяющих сохранить результаты работы утилиты `awk`. Первый, более простой, способ состоит в том, чтобы перенаправить выходной поток в требуемый файл:

```
$ awk '{print $0}' grade.txt > grade.out
```

При этом выходные данные не будут отображаться на экране.

Второй способ заключается в применении команды `tee`, о которой рассказывалось в главе 5. Выходные данные передаются в файл и *одновременно* отображаются на экране. Например:

```
$ awk '{print $0}' grade.txt | tee grade.out
```

Отображение всех записей

В приведенной ниже команде утилита `awk` просматривает файл *grade.txt* и, поскольку шаблон не указан, отображает содержимое всех записей:

```
$ awk '{print $0}' grade.txt
M.Tansley     05/99     48311     Green     8     40     44
J.Lulu        06/99     48317     green     9     24     26
P.Bunny       02/99     48        Yellow    12    35     28
J.Troll       07/99     4842      Brown-3   12    26     26
L.Tansley     05/99     4712      Brown-2   12    30     28
```

Отображение отдельных полей всех записей

Предположим, требуется отобразить на экране только имена спортсменов и названия поясов, которыми они обладают. Соответствующие данные хранятся в полях `$1` и `$4`, *поэтому введем такую команду:*

```
$ awk '{print $1, $4}' grade.txt
M.Tansley Green
J.Lulu green
P.Bunny Yellow
J.Troll Brown-3
L.Tansley Brown-2
```

Отображение заголовка отчета

Результат работы предыдущей команды выглядит не слишком привлекательно. Рассмотрим, какие шаги можно предпринять, чтобы улучшить его. Прежде всего выровняем границы полей посредством символов табуляции. Табуляция создается с помощью Escape-последовательности `\t` (об управляющих последовательностях речь пойдет ниже). Кроме того, для придания отчету солидности добавим к нему заголовок, включающий названия полей, а также разделительную линию, которая отображается

в отдельной строке благодаря Escape-последовательности \n. Заголовок отчета формируется в процедурной части шаблона BEGIN.

```
$ awk 'BEGIN {print "Name          Belt\n--------------------"} \
  {print $1 "  \t" $4}' grade.txt

Name          Belt
--------------------
M.Tansley     Green
J.Lulu        green
P.Bunny       Yellow
J.Troll       Brown-3
L.Tansley     Brown-3
```

Отображение резюме отчета

Чтобы добавить в конец отчета строку "end-of-report", следует воспользоваться шаблоном END. Этот шаблон употребляется для обозначения действий, которые выполняются после обработки последней записи входного файла.

```
$ awk 'BEGIN {print "Name\n---------"} {print $1} \
  END {print "\nend-of-report"}' grade.txt

Name
---------
M.Tansley
J.Lulu
P.Bunny
J.Troll
L.Tansley

end-of-report
```

Обработка сообщений об ошибках

При работе с утилитой awk почти невозможно избежать синтаксических ошибок. Эта утилита выводит на экран искомую строку и указывает, в каком ее месте возникла ошибка. Но сами сообщения об ошибках не слишком информативны и не всегда могут помочь в разрешении проблем.

Давайте смоделируем ситуацию, при которой возникает синтаксическая ошибка, например, пропустим двойную кавычку в предыдущей команде:

```
$ awk 'BEGIN {print "Name\n---------"} {print $1} \
  END {print "\nend-of-report}' grade.txt
awk: cmd. line:2: END {print "\nend-of-report}
awk: end. line:2:              ^ unterminated string
```

Если вы впервые сталкиваетесь с утилитой awk, краткость подобных сообщений может вас смутить. Предлагаем вам перечень правил обнаружения ошибок:

- убедитесь, что сценарий awk целиком заключен в одинарные кавычки;
- удостоверьтесь, что все кавычки внутри сценария являются парными;
- проверьте, заключены ли процедуры в фигурные скобки, а условные конструкции — в круглые скобки.

Более понятным является сообщение об ошибке, возникающей при создании ссылки на несуществующий файл:

```
$ awk 'END {print "End-of-report"}' grades.txt
awk: cmd. line:2: fatal: cannot open file 'grades.txt' for reading
(No such file or directory)
```

Ввод данных с клавиатуры

Давайте посмотрим, что произойдет, если не указать файл *grade.txt* в командной строке:

```
$ awk 'BEGIN {print "Name          Belt\n--------------------"} \
 {print $1"  \t"$4}'

Name          Belt
--------------------
>
```

С помощью шаблона `BEGIN` на экран выводится заголовок отчета, при этом сам отчет пуст, а утилита `awk` ожидает получения входных данных с клавиатуры (об этом свидетельствует строка приглашения >). Вы должны ввести их вручную. После нажатия клавиши [Enter] введенная строка интерпретируется как входная запись и по отношению к ней выполняются соответствующие инструкции. По завершении ввода данных нажмите [Ctrl+D]. Подобный метод работы применяется довольно редко, поскольку чреват большим количеством опечаток и ошибок.

9.2.3. Регулярные выражения

При изучении возможностей команды `grep` приводилось большое количество примеров регулярных выражений, поэтому мы не будем еще раз останавливаться на описании их синтаксиса. Ниже, когда будут рассматриваться операторы, вы встретите много примеров команд `awk` с регулярными выражениями.

В сценарии `awk` регулярное выражение выделяется с обеих сторон символами косой черты: `/регулярное_выражение/`. Например, если в текстовом файле нужно найти строку, содержащую слово "Green", следует задать шаблон `/Green/`.

9.2.4. Метасимволы

Перечисленные ниже метасимволы могут встречаться в регулярных выражениях утилиты `awk`:

```
\ ^ $ . [ ] | ( ) * + ?
```

Следует остановиться на описании двух метасимволов, которые не рассматривались в главе 7, поскольку они специфичны для `awk` и не применяются в команде `grep` и редакторе `sed`.

- `+` Указывает на то, что предыдущий символ встречается один или несколько раз. Например, выражение `/t+/` соответствует одной или нескольким буквам 't', а выражение `/[a-z]+/` — любой последовательности строчных букв.
- `?` Указывает на то, что предыдущий символ встречается не более одного раза. Например, выражение `/XY?Z/` соответствует строкам "XYZ" и "XZ".

9.2.5. Операторы

В awk существует достаточно много операторов, манипулирующих числами, строками, переменными, полями и элементами массива. Ниже приведен список основных операторов.

= += *= /= %=	Операторы присваивания (простого и составного)
?:	Условный оператор
|| && !	Логические операторы ИЛИ, И, НЕ
~ !~	Операторы сравнения с регулярным выражением (совпадение, несовпадение)
< <= == != > >=	Операторы простого сравнения
+ - * / %	Арифметические операторы (сложение, вычитание, умножение, деление, деление по модулю)
++ --	Инкремент и декремент (могут быть префиксными и постфиксными)

9.2.6. Операторы сравнения

Простейшие инструкции awk создаются с помощью операторов сравнения, перечисленных в табл. 9.2.

Таблица 9.2. Операторы сравнения утилиты awk

Оператор	Проверка
<	Меньше
<=	Меньше или равно
==	Равно
!=	Не равно
>	Больше
>=	Больше или равно
~	Соответствие регулярному выражению (фрагмент строки совпадает с шаблоном)
!~	Несоответствие регулярному выражению (в строке не обнаружено совпадений с шаблоном)

Проверка на совпадение

Оператор ~ (тильда) позволяет находить поля, соответствующие заданному шаблону (регулярному выражению). Обычно он применяется в конструкции if, условиая часть которой заключается в круглые скобки.

Предположим, из файла *grade.txt* требуется извлечь информацию о владельцах коричневых поясов. Для этого нужно найти строки, содержащие слово "Brown" (коричневый):

```
$ awk '{if($4 ~ /Brown/) print $0}' grade.txt
J.Troll      07/99    4842     Brown-3  12   26   26
L.Tansley    05/99    4712     Brown-2  12   30   28
```

Конструкция `if` является частью сценария `awk` и помещается в фигурные скобки. Поставленную задачу можно решить намного проще, если вспомнить, что при нахождении строки, соответствующей шаблонной части инструкции, утилита `awk` по умолчанию отображает всю строку. Таким образом, можно вообще не указывать команду `print`, а условную часть конструкции `if` представить в виде шаблона:

```
$ awk '$0 ~ /Brown/' grade.txt
J.Troll      07/99    4842      Brown-3  12   26   26
L.Tansley    05/99    4712      Brown-2  12   30   28
```

Приведенная команда означает следующее: если в строке встречается слово "Brown", вывести ее на экран.

Проверка на равенство

Допустим, необходимо получить информацию об ученике с номером 48. Показанная ниже команда не позволит решить данную задачу, поскольку в файле есть множество номеров, содержащих последовательность цифр "48":

```
$ awk '{if($3 ~ /48/) print $0}' grade.txt
M.Tansley    05/99    48311     Green    8    40   44
J.Lulu       06/99    48317     green    9    24   26
P.Bunny      02/99    48        Yellow   12   35   28
J.Troll      07/99    4842      Brown-3  12   26   26
```

Чтобы найти точное совпадение, воспользуйтесь оператором ==:

```
$ awk '$3 == "48"' grade.txt
P.Bunny      02/99    48        Yellow   12   35   28
```

Проверки на несовпадение и неравенство

Иногда требуется извлечь те строки, которые не соответствуют шаблону. Для этих целей предназначен оператор `!~`, выполняющий проверку на неравенство регулярному выражению. Давайте, например, выведем список всех учеников, не являющихся обладателями коричневого пояса:

```
$ awk '$0 !~ /Brown/' grade.txt
M.Tansley    05/99    48311     Green    8    40   44
J.Lulu       06/99    48317     green    9    24   26
P.Bunny      02/99    48        Yellow   12   35   28
```

Не забывайте, что по умолчанию утилита `awk` выводит на экран все записи, отвечающие указанному критерию отбора, поэтому нет необходимости задавать какое-либо действие. Если вместо предыдущей команды указать

```
$ awk '$4 != "Brown"' grade.txt
```

получим ошибочный результат. Эта команда означает, что требуется найти строки, в которых четвертое поле не равно "Brown". Такому критерию удовлетворяют все строки в файле. Конечно, если нужно найти обладателей поясов, отличных от "Brown-2", можно применить следующую команду:

```
$ awk '$4 != "Brown-2"' grade.txt
M.Tansley    05/99    48311     Green    8    40   44
J.Lulu       06/99    48317     green    9    24   26
P.Bunny      02/99    48        Yellow   12   35   28
J.Troll      07/99    4842      Brown-3  12   26   26
```

Обратите внимание на один важный момент: строка "Brown-2" заключена в двойные кавычки. Если этого не сделать, четвертое поле будет сравниваться с содержимым переменной Brown-2. Вряд ли в вашем интерпретаторе существует такая переменная, поэтому вместо нее будет подставлена пустая строка, и вы получите совершенно другой результат.

Проверка "меньше чем"

Допустим, нужно определить, кто из учеников не смог набрать максимального количества очков на соревновании. Для выполнения проверки достаточно сравнить набранный рейтинг (поле 6) с общей суммой возможных очков (поле 7). Также поместим в отчет небольшое сообщение.

```
$ awk '{if($6 < $7) print $1 " -- try better at the next competition"}'
   grade.txt
M.Tansley -- try better at the next competition
J.Lulu -- try better at the next competition
```

Проверка "меньше или равно"

Чтобы включить в отчет тех учеников, рейтинг которых не выше значения в седьмом поле, нужно лишь незначительно видоизменить предыдущий пример:

```
$ awk '{if($6 <= $7) print $1}' grade.txt
M.Tansley
J.Lulu
J.Troll
```

Проверка "больше чем"

Следующая команда формирует список лидеров соревнования:

```
$ awk '{if ($6 > $7) print $1}' grade.txt
L.Tansley
P.Bunny
```

9.2.7. Логические операторы

Логические операторы позволяют формировать сложные выражения, позволяющие выполнять проверку нескольких условий. Существует три логических оператора:

- `&&` И: чтобы результат был истинным, оба операнда должны быть истинными
- `||` ИЛИ: чтобы результат был истинным, хотя бы один из операндов должен быть истинным
- `!` НЕ: результат проверки инвертируется

Оператор логического И

Предположим, перед нами поставлена задача выяснить, кому был присвоен зеленый пояс в мае 1999 года. Строки, отвечающие этому условию, должны в первом поле содержать значение "05/99", а во втором — "Green":

```
$ awk '{if($2 == "05/99" && $4 == "Green") print $0}' grade.txt
M.Tansley    05/99    48311    Green    8    40    44
```

Оператор логического ИЛИ

Чтобы получить список учеников, обладающих желтым или коричневым поясом, воспользуйтесь оператором ||:

```
$ awk '{if($4 == "Yellow" || $4 ~ /Brown/) print $0}' grade.txt
P.Bunny      02/99    48       Yellow   12   35   28
J.Troll      07/99    4842     Brown-3  12   26   26
L.Tansley    05/99    4712     Brown-2  12   30   28
```

Приведенную команду можно упростить с помощью метасимвола |, означающего выбор любого из двух шаблонов:

```
$ awk '$4 ~ /Yellow|Brown/' grade.txt
P.Bunny      02/99    48       Yellow   12   35   28
J.Troll      07/99    4842     Brown-3  12   26   26
L.Tansley    05/99    4712     Brown-2  12   30   28
```

9.2.8. Операторы присваивания и арифметические операторы

С помощью операторов присваивания и арифметических операторов можно создавать в сценариях awk локальные переменные и манипулировать их значениями.

Создание локальных переменных

При разработке сценариев awk не всегда удобно работать с идентификаторами полей. Лучше создать переменную, содержащую значение поля, и присвоить ей выразительное имя, чтобы в дальнейшем ссылаться на поле по имени. Подобную переменную можно получить с помощью конструкции следующего вида:

```
имя_переменной = $n
```

где *n* — существующий номер поля.

В следующем примере мы создадим две переменные: name, содержащую имена учеников (поле 1), и belts, содержащую названия поясов (поле 4). Затем будет произведен поиск учеников, обладающих желтым поясом.

```
$ awk '{name=$1; belts=$4; if(belts ~ /Yellow/) print name" is belt "belts}'
   grade.txt
P.Bunny is belt Yellow
```

Обратите внимание на то, что команды сценария отделяются друг от друга точкой с запятой.

Проверка значения поля

В следующем примере мы проверим, кто из учеников набрал в соревнованиях менее 27 очков. В первом варианте команды значение поля $6 непосредственно сравнивается с числом 27:

```
$ awk '$6 < 27' grade.txt
J.Lulu       06/99    48317    green    9    24   26
J.Troll      07/99    4842     Brown-3  12   26   26
```

Более универсальный способ заключается в том, чтобы перед обработкой входного файла, в процедурной части шаблона BEGIN, создать, как в настоящей программе, набор локальных переменных с нужными значениями и ссылаться на них, когда

потребуется. Конечно, этот прием неэффективен в случае одноразовых команд, зато очень полезен в больших сценариях, так как позволяет легко вносить в них изменения. Во втором примере мы создадим переменную BASELINE и присвоим ей значение 27, а затем сравним с ней интересующее нас поле $6:

```
$ awk 'BEGIN {BASELINE=27} (if($6 < BASELINE) print $0}' grade.txt
J.Lulu        06/99     48317     green     9     24    26
J.Troll       07/99     4842      Brown-3   12    26    26
```

Изменение значения числового поля

Изменяя значение поля, следует помнить о том, что содержимое входного файла на самом деле не меняется. Изменению подвергается только копия файла, которая хранится в буфере awk.

В следующем примере на экран выводятся имена учеников и их рейтинговые очки, а рейтинг ученика по имени M.Tansley уменьшается на единицу:

```
$ awk '(if($1 == "M.Tansley") $6=$6-1; print $1, $6}' grade.txt
M.Tansley   39
J.Lulu      24
P.Bunny     35
J.Troll     26
L.Tansley   30
```

Изменение значения текстового поля

Для изменения значения текстового поля достаточно применить к нему оператор присваивания. Следующая команда при выводе на экран добавляет к имени ученика J.Troll дополнительный инициал:

```
$ awk '{if($1 == "J.Troll") $1="J.L.Troll"; print $1}' grade.txt
M.Tansley
J.Lulu
P.Bunny
J.L.Troll
L.Tansley
```

Не забывайте о том, что строковые константы следует заключать в двойные кавычки. Поскольку имена учеников в данном случае содержат точки, утилита awk выдаст сообщение об ошибке при отсутствии кавычек, так как точка является метасимволом и встречается в непонятном контексте.

Отображение только измененных записей

При работе с файлами большого объема часто нет необходимости отображать все записи, а достаточно вывести лишь те из них, которые подверглись изменениям. По отношению к предыдущему примеру это означает, что все команды после конструкции if следует дополнительно заключить в фигурные скобки:

```
$ awk '{if($1=="J.Troll") {$1="J.L.Troll"; print $1}}' grade.txt
J.L.Troll
```

Создание нового поля

Аналогично локальным переменным в сценарии awk можно создавать новые поля. Например, мы можем создать поле $8, содержащее разницу полей $6 и $7 в том случае, если значение в поле $7 больше, чем в поле $6:

```
$ awk 'BEGIN {print "Name\t\tDifference"} {if($6 < $7) \
  {$8=$7-$6; print $1"  \t"$8}}' grade.txt
Name        Difference
M.Tansley   4
J.Lulu      2
```

Суммирование столбцов

Для вычисления суммарного рейтинга учеников секции мы создадим переменную tot и с помощью выражения tot+=$6 будем прибавлять к ней значение поля $6 при обработке каждой записи. По завершении обработки записей в процедурной части шаблона END итоговое значение переменной tot будет выведено на экран.

```
$ awk 'tot+=$6; END {print "Club student total points: " tot}' grade.txt
M.Tansley    05/99    48311    Green     8    40    44
J.Lulu       06/99    48317    green     9    24    26
P.Bunny      02/99    48       Yellow    12   35    28
J.Troll      07/99    4842     Brown-3   12   26    26
L.Tansley    05/99    4712     Brown-2   12   30    28
Club student total points: 155
```

Вероятно, вы заметили, что утилите awk не было дано указание выводить на экран все записи — она сделала это сама. Причина такого поведения заключается в том, что выражение tot+=$6 относится к шаблонной части инструкции и не задает критерия отбора строк, т.е. применяется ко всем записям. А поскольку процедурная часть этого шаблона отсутствует, выполняется действие по умолчанию — команда print $0.

Если файл велик, можно не выводить на экран все записи, а лишь отобразить итог. Для этого достаточно взять выражение tot+=$6 в фигурные скобки, чтобы перенести его в процедурную часть инструкции:

```
$ awk '{tot+=$6}; END {print "Club student total points: " tot}' grade.txt
Club student total points: 155
```

Суммирование размеров файлов

При просмотре содержимого каталога часто требуется узнать общий размер всех файлов в нем, исключая файлы в подкаталогах и скрытые файлы. Алгоритм решения этой задачи таков: результаты работы команды ls -l (формирует список файлов с расширенной информацией о них; см. главу 1) направляются утилите awk, которая удаляет записи, начинающиеся с символа 'd' (признак каталога), и вычисляет сумму по 5-му столбцу (содержит размер файла).

Представленная ниже команда отображает список файлов текущего каталога (имя файла берется из 9-го столбца), указывая размер каждого из них, а в конце выводит суммарный размер файлов, накопленный в переменной tot:

```
$ ls -l | awk '/^[^d]/ {print $9"\t"$5; tot+=$5} END {print "total KB: "tot}'
dev_pkg.fail    345
failedlogin     12416
```

```
messages        4260
sulog           12810
utap            1856
wtap            7104
total KB: 38791
```

Если необходимо включить в список скрытые файлы, следует вместо команды `ls -l` задать команду `ls -la`.

9.2.9. Встроенные переменные

Утилита `awk` имеет ряд встроенных переменных, которые позволяют получить подробную информацию о входном потоке и настройках `awk`. Значения некоторых переменных можно изменять. В табл. 9.3 кратко описаны основные переменные.

Таблица 9.3. Встроенные переменные `awk`

Переменная	Что содержит
ARGC	Количество аргументов в командной строке (поддерживается только в `nawk` и `gawk`)
ARGV	Массив аргументов командной строки (поддерживается только в `nawk` и `gawk`)
ENVIRON	Массив переменных среды (поддерживается только в `nawk` и `gawk`)
FILENAME	Имя файла, обрабатываемого в текущий момент
FNR	Количество уже обработанных записей в текущем файле (поддерживается только в `nawk` и `gawk`)
FS	Разделитель полей во входном потоке (по умолчанию пробел); аналогична опции `-F` командной строки
NF	Количество полей в текущей записи
NR	Количество обработанных записей во входном потоке
OFS	Разделитель полей в выходном потоке (по умолчанию пробел)
ORS	Разделитель записей в выходном потоке (по умолчанию символ новой строки)
RS	Разделитель записей во входном потоке (по умолчанию символ новой строки)

Переменная ARGC хранит число аргументов командной строки, переданной сценарию `awk` (точнее, `nawk` или `gawk`, т.к. эта переменная появилась только в новых версиях утилиты). Переменная ARGV хранит значения аргументов командной строки. Доступ к нужному аргументу осуществляется с помощью ссылки `ARGV[n]`, где *n* — порядковый номер аргумента в командной строке.

Переменная ENVIRON хранит значения всех текущих переменных среды. Чтобы получить доступ к нужной переменной, следует указать ее имя, например:

```
ENVIRON["EDITOR"]=="vi"
```

Поскольку сценарий `awk` может обрабатывать большое количество файлов, предусмотрена переменная FILENAME, которая указывает на то, какой файл просматривается в текущий момент.

Переменная FNR хранит номер записи, которую утилита awk обрабатывает в текущий момент; ее значение меньше или равно значению переменной NR, которая отслеживает общее число обработанных записей входного потока. Если сценарий получает доступ более чем к одному файлу, переменная FNR сбрасывается в ноль при открытии каждого нового файла. В переменную NF записывается количество полей текущей записи. Ее значение сбрасывается по достижении конца записи.

Переменная FS содержит символ, используемый в качестве разделителя полей входного потока. Эту переменную можно установить из командной строки с помощью опции -F. По умолчанию разделителем полей служит пробел. Переменная OFS содержит символ, являющийся разделителем полей в выходном потоке. По умолчанию это тоже пробел.

В переменной ORS хранится разделитель записей в выходном потоке. По умолчанию им является символ новой строки (\n). Переменная RS содержит разделитель записей во входном потоке (в большинстве случаев это тоже символ \n).

Переменные NF, NR и FILENAME

Представленная ниже команда позволяет быстро определить число записей во входном файле *grade.txt*. Значение переменной NR отображается по завершении обработки файла.

```
$ awk 'END {print NR}' grade.txt
```

В следующем примере на экран выводятся все записи исходного файла. Каждой из них предшествуют два числа: количество полей в записи (переменная NF) и номер записи в файле (переменная NR). В самом конце отображается имя входного файла (переменная FILENAME).

```
$ awk '{print NF, NR, $0} END {print FILENAME}' grade.txt
7  1 M.Tansley    05/99    48311    Green    8    40    44
7  2 J.Lulu       06/99    48317    green    9    24    26
7  3 P.Bunny      02/99    48       Yellow   12   35    28
7  4 J.Troll      07/99    4842     Brown-3  12   26    26
7  5 L.Tansley    05/99,   4712     Brown-2  12   30    28
grade.txt
```

Переменную NF удобно использовать, когда требуется извлечь из путевого имени последнюю часть, т.е. имя файла или каталога. В этом случае необходимо указать, что разделителем полей является символ '/', и использовать ссылку $NF, которая является обозначением последнего поля текущей записи. Например:

```
$ pwd
/usr/local/etc
$ echo $PWD | awk -F/ '{print $NF}'
etc
```

Переменная среды $PWD хранит путевое имя текущего каталога.

9.2.10. Встроенные функции работы со строками

Утилита awk располагает набором универсальных функций преобразования строк. В табл. 9.4 перечислены основные из них.

Таблица 9.4. Функции работы со строками

Функция	Назначение
gsub(*r*,*s*)	Выполняет глобальную замену каждой строки, соответствующей регулярному выражению *r*, строкой *s* в пределах текущей записи; появилась в nawk
index(*s*,*t*)	Возвращает позицию первого вхождения подстроки *t* в строку *s*
length(*s*)	Возвращает длину строки *s*
match(*s*,*r*)	Проверяет, содержит ли строка *s* подстроку, соответствующую регулярному выражению *r*; появилась в nawk
split(*s*,*a*,*fs*)	Разбивает строку *s* на элементы, разделенные символом *fs*, и помещает полученные элементы в массив *a*
sub(*r*,*s*)	Выполняет замену самой первой строки, соответствующей регулярному выражению *r*, строкой *s* в пределах текущей записи; появилась в nawk
substr(*s*,*p*[,*n*])	Возвращает подстроку строки *s*, начинающуюся с позиции *p* и имеющую длину *n*; если аргумент *n* не задан, концом подстроки считается символ \0 (признак конца строки)

Функция gsub()

Благодаря функции gsub() вы сможете выполнить в текущей записи глобальную замену строк, соответствующих заданному регулярному выражению. Например, для изменения номера ученика с 4842 на 4899 введите такую команду:

```
$ awk 'gsub(4842,4899) {print $0}' grade.txt
J.Troll      07/99    4899    Brown-3  12   26   26
```

Функция index()

Чтобы узнать позицию первого вхождения подстроки *t* в строку *s*, воспользуйтесь функцией index(), только не забудьте взять ее аргументы в двойные кавычки, иначе они будут восприниматься как имена переменных среды. Например, следующая команда возвращает число, определяющее позицию подстроки "ny" в строке "Bunny":

```
$ awk 'BEGIN {print index("Bunny","ny")}' grade.txt
4
```

Функция length()

Функция length() возвращает длину переданного ей текстового аргумента. В показанном ниже примере производится поиск информации об ученике с номером 4842, а затем определяется длина имени ученика:

```
$ awk '$3==4842 {print length($1)" "$1}' grade.txt
7 J.Troll
```

Следующая команда демонстрирует применение утилиты awk для вычисления длины текстовой строки:

```
$ awk 'BEGIN {print length("A FEW GOOD MEN")}'
14
```

Функция match()

Функция `match()` позволяет проверить, содержит ли строка заданную подстроку. Последняя может быть представлена как литералом в двойных кавычках, так и регулярным выражением. Если поиск прошел успешно, возвращается число, определяющее позицию, с которой начинается вхождение подстроки в искомую строку. В случае неудачи возвращается ноль. Следующая команда проверяет, содержит ли имя ученика с номером 48317 символ 'u':

```
$ awk '$3==48317 {print match($1,"u"), $1}' grade.txt
4 J.Lulu
```

Функция split()

Функция `split()` преобразует переданную ей строку в массив и возвращает число элементов в полученном массиве. В следующем примере заданная строка разбивается на три элемента, которые помещаются в массив `myarray`. Разделителем элементов в данном случае является символ '#'.

```
$ awk 'BEGIN {print split("123#456#678",myarray,"#")}'
3
```

Массив `myarray` будет иметь такую структуру:

```
myarray[1]="123"
myarray[2]="456"
myarray[3]="678"
```

Функция sub()

Функция `sub()` применяется для поиска строки, соответствующей заданному шаблону, и ее замены при первом появлении. В этом состоит отличие данной функции от функции `gsub()`, которая находит все случаи вхождения подстроки в строку, производя соответствующее число замен. Приведенная ниже команда находит запись ученика J.Troll и меняет его рейтинг с 26 на 29 (поле 6), при этом значение поля 7 (тоже 26) остается неизменным:

```
$ awk '$1=="J.Troll" {sub(26,29,$0)}' grade.txt
J.Troll      07/99    4842     Brown-3  12   29   26
```

Функция substr()

Функция `substr()` возвращает указанную часть строки. Вам нужно задать позицию, с которой начинается вхождение подстроки в искомую строку, и длину подстроки. Рассмотрим пример:

```
$ awk '$1=="L.Tansley" {print substr($1,1,5)}' grade.txt
L.Tan
```

Эта команда возвращает из строки "L.Tansley" подстроку, начинающуюся с первого символа и занимающую пять символов.

Если значение третьего аргумента значительно превышает реальную длину строки, функция `substr()` возвращает все символы строки, начиная с указанной позиции:

```
$ awk '$1=="L.Tansley" {print substr($1,3,99)}' grade.txt
Tansley
```

То же самое происходит, когда третий аргумент [illegible]ан. Например, следующая команда формирует список фамилий учеников:

```
$ awk '{print substr($1,3)}' grade.txt
Tansley
Lulu
Bunny
Troll
Tansley
```

Передача строк из интерпретатора shell утилите awk

Очень часто утилита awk получает входные данные не из файла, а по каналу от других команд. Рассмотрим несколько примеров обработки строк, поступающих из канала.

В первом примере команда echo передает строку "Stand-by" утилите awk, которая вычисляет ее длину:

```
$ echo "Stand-by" | awk '{print length($0)}'
8
```

Во втором примере утилита awk получает строку с именем файла и возвращает имя файла без расширения:

```
$ echo "mydoc.txt" | awk '{print substr($STR,1,5)}'
mydoc
```

Следующая команда возвращает только расширение файла:

```
$ echo "mydoc.txt" | awk '{print substr($STR,7)}'
txt
```

9.2.11. Escape-последовательности

При работе со строками и регулярными выражениями нередки случаи включения в шаблон поиска непечатаемых символов (таких как символ новой строки либо табуляции) или же символов со специальным значением в утилите awk (любой из метасимволов). Такие символы создаются с помощью управляющих Escape-последовательностей, признаком которых является обратная косая черта в начале.

Например, шаблон поиска открывающей фигурной скобки выглядит так:

```
/\{/
```

В табл. 9.5 перечислены Escape-последовательности, распознаваемые утилитой awk.

Таблица 9.5. Escape-последовательности утилиты awk

\b	Возврат на одну позицию (забой)
\f	Прокрутка страницы
\n	Новая строка
\r	Возврат каретки
\t	Горизонтальная табуляция
\ddd	Восьмеричный код символа
\c	Любой другой специальный символ. Например, запись \\ соответствует символу обратной косой черты

В следующей команде сначала отображается фраза "May Day", в которой слова разделены символом табуляции, а затем выводятся два символа новой строки, вследствие чего образуется пустая строка. Потом отображается слово "May", а за ним — слово "Day", каждая буква которого представлена ASCII-кодом: 'D' — 104, 'a' — 141, 'y' — 171.

```
$ awk 'BEGIN {print "May\tDay\n\nMay \104\141\171"}'
May     Day

May     Day
```

9.2.12. Команда printf

Во всех примерах, с которыми мы ознакомились, данные выводились на экран с помощью команды `print` без какого-либо форматирования. В `awk` имеется намного более мощная команда `printf`, аналог одноименной функции языка C, позволяющая задавать правила форматирования выходных данных.

Базовый синтаксис команды таков:

```
printf "строка_форматирования", аргументы
```

Строка форматирования может включать как литеральные символы, записываемые в выходной поток без изменения, так и спецификаторы форматирования (табл. 9.6), каждый из которых задает способ интерпретации соответствующего ему аргумента.

Таблица 9.6. Спецификаторы форматирования

`%c`	Символ ASCII; если аргумент является строкой, выводится первый символ строки
`%d, %i`	Целое число
`%e`	Число с плавающей точкой в формате [-]*d.dddddde*[+-]*dd*
`%E`	Аналогичен спецификатору `%e`, но знак экспоненты представлен символом 'E', а не 'e'
`%f`	Число с плавающей точкой в формате [-]*ddd.dddddd*
`%g`	Тип преобразования будет `%e` или `%f` в зависимости от того, какой результат короче; вывод незначащих нулей подавляется
`%G`	Аналогичен спецификатору `%g`, но экспоненциальный формат будет представлен спецификатором `%E`, а не `%e` .
`%o`	Восьмеричное число без знака
`%s`	Строка символов
`%x`	Шестнадцатеричное число без знака (используются шестнадцатеричные цифры `a`, `b`, `c`, `d`, `e`, `f`)
`%X`	Аналогичен спецификатору `%x`, но используются шестнадцатеричные цифры `A`, `B`, `C`, `D`, `E`, `F`
`%%`	Отображается символ '%', интерпретации аргумента не происходит

В состав спецификаторов форматирования могут входить различные модификаторы, определяющие дополнительные особенности форматирования (табл. 9.7). Модификатор помещается между символом '%' и управляющим символом.

Таблица 9.7. Дополнительные модификаторы

-	Аргумент выравнивается по левому краю поля вывода (по умолчанию принято правостороннее выравнивание)
пробел	Если аргумент является положительным числом, перед ним ставится пробел, а если отрицательным — знак минус
+	Если аргумент является числом, ему всегда предшествует знак плюс, даже если это положительное число
#	Выбирается альтернативная форма спецификатора: %o — восьмеричному числу предшествует ведущий ноль; %x — шестнадцатеричному числу предшествует запись 0x; %X — шестнадцатеричному числу предшествует запись 0X; %e, %E, %f — число всегда содержит десятичную точку; %g, %G — вывод незначащих нулей не подавляется
0	Если длина поля вывода больше, чем число символов в представлении аргумента, аргумент дополняется нулями, а не пробелами
ширина	Ширина поля вывода; если длина поля больше, чем число символов в представлении аргумента, аргумент по умолчанию дополняется пробелами
.точность	Точность представления аргумента: %e, %E, %f — число символов после десятичной точки; %g, %G — максимальное число значащих цифр; %d, %o, %i, %u, %x, %X — минимальное число выводимых цифр; %s — максимальное число выводимых символов

Преобразование символов

Чтобы узнать, ASCII-код какого символа равен 65, можно с помощью команды echo направить строку "65" утилите awk и затем передать ее в качестве аргумента команде printf со спецификатором %c. Команда printf выполнит преобразование автоматически:

```
$ echo "65" | awk '{printf "%c\n", $0}'
A
```

Как видите, это символ 'A'. Обратите внимание на наличие в команде символа новой строки (\n). Необходимость в нем объясняется тем, что команда printf по умолчанию не записывает этот символ в выходной поток. Если не указать Escape-последовательность \n, сразу после буквы 'A' в той же строке будет отображено приглашение интерпретатора shell, что может смутить пользователя, работающего в данный момент за терминалом.

Конечно, код символа может быть указан непосредственно в команде printf:

```
$ awk 'BEGIN {printf "%c\n", 65}'
A
```

Форматированный вывод

Предположим, необходимо отобразить имена учеников и их идентификаторы, причем имена должны быть выровнены по левому краю и размещаться в поле длиной 15 символов. В конце строки форматирования стоит символ новой строки (\n),

служащий для разделения записей. Выводимые данные будут размещаться в двух колонках:

```
$ awk '{printf "%-15s %d\n", $1, $3}' grade.txt
M.Tansley     48311
J.Lulu        48311
P.Bunny       48
J.Troll       4842
L.Tansley     4712
```

9.2.13. Передача переменных утилите awk

Переменные можно создавать не только в сценарии awk, но и непосредственно в командной строке. Формат вызова утилиты awk в этом случае таков:

```
awk 'сценарий' переменная=значение входной_файл
```

В следующем примере переменная AGE создается в командной строке и инициализируется значением 10. Показанная команда находит студентов, возраст которых не превышает 10 лет.

```
$ awk '{if($5 < AGE) print $0}' AGE=10 grade.txt
M.Tansley     05/99     48311     Green     8     40     44
J.Lulu        06/99     48317     green     9     24     26
```

Рассмотрим более сложный пример. Системная команда df отображает информацию о смонтированных файловых системах. Результаты ее работы по умолчанию имеют следующий формат:

	Имя файловой системы	Число блоков	Занято блоков	Свободно блоков	Процент занятых блоков	Точка монтирования
Столбец	1	2	3	4	5	6

Чтобы узнать, в какой из имеющихся файловых систем объем свободного пространства ниже критической отметки, следует передать выходные данные команды df утилите awk и последовательно сравнить значения в четвертом столбце с пороговым значением. В следующей командной строке пороговое число задано в виде переменной TRIGGER, которая равна 56000:

```
$ df -k | awk '$4 ~ /^[0-9]/ {if($4 < TRIGGER) print $1"\t"$4}'
TRIGGER=56000
/dos     55808
/apps    51022
```

Опция -k команды df устанавливает размер блока равным 1 Кб (наиболее привычный для пользователя режим), так как в системе может быть задан другой размер блока. Проверка $4 ~ /^[0-9]/ позволяет отсечь заголовок отчета команды df, содержащий в четвертом столбце строку "Available".

Вот еще один пример. Команда who выводит сведения о пользователях, зарегистрировавшихся в системе. В первом столбце выходных данных этой команды отображаются регистрационные имена пользователей, а во втором — имена терминалов, к которым подключены эти пользователи. С помощью утилиты awk вы можете

отфильтровать результаты работы команды who и узнать, к какому терминалу подключены:

```
$ who | awk '{if($1==user) print $1 " you are connected to " $2}'
   user=$LOGNAME
root you are connected to ttyp1
```

Здесь локальная переменная user инициализируется значением переменой среды $LOGNAME, которая хранит регистрационное имя текущего пользователя.

9.2.14. Файлы сценариев

Если последовательность команд awk слишком велика, чтобы вводить ее в командной строке, или предназначена для многократного использования, целесообразно поместить ее в отдельный файл сценария. Преимуществом сценариев является также возможность добавления комментариев, благодаря которым вы сможете быстро вспомнить назначение той или иной программы.

Давайте возьмем один из рассмотренных ранее примеров и преобразуем его в файл сценария awk. В частности, следующая, уже знакомая нам, команда подсчитывает суммарный рейтинг учеников секции:

```
$ awk 'tot+=$6; END {print "Club student total points: " tot}' grade.txt
```

Создадим на ее основе файл, который назовем *student_tot.awk*. Расширение *awk* является общепринятым соглашением относительно именования файлов сценариев awk. Вот текст этого файла:

```
#! /bin/awk -f
# Все строки комментариев должны начинаться с символа '#'.
# Имя файла: student_tot.awk
# Командная строка: student_tot.awk grade.txt
# Вычисление суммарного и среднего рейтинга учеников секции.

# Сначала выводим заголовок.
BEGIN {
  print "Student      Date     Member   Grade   Age Points Max"
  print "name         joined   number               gained point available"
  print "=============================================================="
}

# Суммируем рейтинг учеников.
 tot+=$6

# В завершение выводим суммарный и средний рейтинг.
END {
  print "Club student total points: " tot
  print "Average club student points: " tot/NR
}
```

В этом сценарии мы формируем заголовок отчета и вычисляем не только суммарный рейтинг, но и средний рейтинг учеников секции, который получается путем деления переменной tot на переменную NR, содержащую число записей во входном файле. Как видите, на языке awk можно писать полноценные программы, содержащие команды, комментарии, а также пустые строки и дополнительные пробелы, предназначенные для повышения удобочитаемости текста сценария.

Ключевым моментом сценария является первая строка, выглядящая как комментарий:

```
#! /bin/awk -f
```

На самом деле это своеобразная системная инструкция, указывающая, какая программа должна выполнять данный сценарий. Подобная инструкция должна быть первой строкой любого сценария. Общий ее формат таков:

```
#! /путь/программа [командная_строка]
```

Выражение #! называется "магической" последовательностью. Подразумевается, что, во-первых, система, в которой запускается сценарий, распознает эту последовательность, а, во-вторых, указанная программа воспринимает символ '#' как признак комментария. Это справедливо в отношении всех современных интерпретаторов shell, а также программ `perl`, `awk` и `sed`, для которых чаще всего создаются автономные сценарии.

Когда происходит запуск исполняемого файла, система проверяет, начинается ли он с "магической" последовательности. Если нет, значит, файл содержит машинные коды и выполняется непосредственно. Если же обнаружено выражение #!, то это файл сценария. В таком случае происходит следующее:

1. Первая строка сценария заменяет собой командную строку, из нее удаляется "магическая" последовательность;
2. Предыдущая командная строка передается новой командной строке в качестве аргумента.

В нашем случае это означает, что при запуске сценария вместо команды

```
$ student_tot.awk grade.txt
```

в действительности выполняется такая команда:

```
$ /bin/awk -f student_tot.awk grade.txt
```

Опция `-f` утилиты `awk` говорит о том, что выполняемые команды находятся в указанном вслед за ней файле.

После создания файл *student_tot.awk* необходимо сделать исполняемым с помощью команды

```
$ chmod u+x student_tot.awk
```

Вот результаты работы сценария:

```
$ student_tot.awk grade.txt
Student      Date     Member    Grade    Age Points Max
name         joined   number                gained point available
=============================================================
M.Tansley    05/99    48311     Green    8    40    44
J.Lulu       06/99    48317     green    9    24    26
P.Bunny      02/99    48        Yellow   12   35    28
J.Troll      07/99    4842      Brown-3  12   26    26
L.Tansley    05/99    4712      Brown-2  12   30    28
Club student total points: 155
Average Club Student points: 31
```

Использование переменной FS в сценариях awk

Если просматривается файл, в котором разделителем полей является символ, отличный от пробела, например '#' или ':', это легко учесть, указав в командной строке опцию -F:

```
$ awk -F: '{print $0}' входной_файл
```

Аналогичную функцию выполняет переменная FS, которую можно установить непосредственно в сценарии. Следующий сценарий обрабатывает файл */etc/passwd*, выводя на экран содержимое первого и пятого полей, включающих соответственно имя пользователя и описание роли пользователя в системе. Поля в этом файле разделены символом двоеточия, поэтому в процедурной части шаблона BEGIN устанавливается переменная FS, значение которой заключается в двойные кавычки:

```
$ cat pasawd.awk
#! /bin/awk -f
# Имя файла: student_tot.awk
# Командная строка: passwd.awk /etc/passwd
# Вывод содержимого первого и пятого полей файла паролей.

BEGIN {
  FS=":"
}

{print $1"\t"$5}
```

Вот возможные результаты работы этого сценария:

```
$ passwd.awk /etc/passwd
root    Special Admin login
xdm     Restart xda Login
sysadm  Regular Admin login
daemon  Daemon Login for daemons needing permissions
```

Передача переменных сценариям awk

При передаче переменной сценарию awk формат командной строки таков:

```
awk файл_сценария переменная=значение входной_файл
```

Следующий небольшой сценарий сравнивает количество полей в каждой записи входного файла с заданным в командной строке значением. Текущее количество полей хранится в переменной NF. Сравниваемое значение передается сценарию в виде переменной MAX.

```
$ cat fieldcheck.awk
#! /bin/awk -f
# Имя файла: fieldcheck.awk
# Командная строка: fieldcheck.awk MAX=n [FS=<разделитель>] имя_файла
# Проверка числа полей в записях файла.

{if(NF != MAX)
   print "line " NR " does not have " MAX " fields"}
```

При запуске этого сценария вместе с файлом */etc/passwd*, содержащим семь полей, необходимо указать следующие параметры:

```
$ fieldcheck.awk MAX=7 FS=":" /etc/passwd
```

Следующий сценарий выводит информацию об учениках, чей возраст ниже значения, заданного в командной строке:

```
$ cat age.awk
#! /bin/awk -f
# Имя файла: age.awk
# Командная строка: age.awk AGE=n grade.txt
# Вывод информации об учениках, чей возраст ниже заданного.

{if($5 < AGE)
   print $0}

$ age.awk AGE=10 grade.txt
M.Tansley     05/99     48311     Green     8    40    44
J.Lulu        06/99     48317     green     9    24    26
```

9.2.15. Массивы

Изучая функцию `split()`, мы говорили о том, каким образом ее можно использовать для преобразования строки символов в массив. Повторим соответствующий пример:

```
$ awk 'BEGIN {print split("123#456#678", myarray, "#")}'
3
```

Функция `split()` возвращает число элементов созданного массива *myarray*. Внутренняя структура массива *myarray* в этом случае такова:

```
myarray[1]="123"
myarray[2]="456"
myarray[3]="678"
```

При работе с массивами в `awk` не требуется заранее их объявлять. Не нужно также указывать количество элементов массива. Для доступа к массиву обычно применяется цикл `for`, синтаксис которого следующий:

```
for (элемент in массив) print массив[элемент]
```

В показанном ниже сценарии функция `split()` разбивает заданную строку на элементы и записывает их в массив `myarray`, а в цикле `for` содержимое массива выводится на экран:

```
$ cat arraytest.awk
#! /bin/awk -f
# Имя файла: arraytest.awk
# Командная строка: arraytest.awk /dev/null
# Вывод элементов массива.

BEGIN {
  record="123#456#789";
```

```
  split(record, myarray, "#")
 }
END {
  for(i in myarray) print myarray[i]
}
```

Для запуска сценария в качестве входного файла следует указать устройство */dev/null*. Если этого не сделать, утилита awk будет ожидать поступления данных с клавиатуры.

```
$ arraytest.awk /dev/null
123
456
789
```

Статические массивы

В предыдущем примере массив формировался динамически. Следующий пример является более сложным и демонстрирует, как создавать в сценарии статические массивы. Ниже показан входной файл *grade_student.txt*, включающий информацию об учениках нашей секции каратистов. Записи файла содержат два поля: первое — название пояса, которым владеет ученик, второе — категория ученика (взрослый, юниор). Разделителем полей служит символ '#'.

```
$ cat grade_student.txt
Yellow#Junior
Orange#Senior
Yellow#Junior
Purple#Junior
Brown-2#Junior
White#Senior
Orange#Senior
Red#Junior
Brown-2#Senior
Yellow#Senior
Red#Junior
Blue#Senior
Green#Senior
Purple#Junior
White#Junior
```

Наша задача заключается в том, чтобы прочитать файл и получить следующие сведения:

1. Сколько учеников имеют желтый, оранжевый и красный пояса?
2. Сколько взрослых и детей посещают секцию?

Рассмотрим такой сценарий:

```
$ cat belts.awk
#! /bin/awk -f
# Имя файла: belts.awk
# Командная строка: belts.awk grade_student.txt
# Подсчет числа учеников, имеющих желтый, оранжевый и красный пояса,
# а также количества взрослых и юных членов секции.
```

```
# Задание разделителя и создание массивов.
BEGIN {
  FS="#"
# Создание массива, индексами которого являются
# названия поясов.
  belt["Yellow"]
  belt["Orange"]
  belt["Red"]
# Создание массива, индексами которого являются
# названия категорий учеников.
  student["Junior"]
  student["Senior"]
}
# Если значение первого поля совпадает с индексом массива belt,
# содержимое соответствующего элемента массива увеличивается на единицу.
  {for(colour in belt)
     {if($1==colour)
        belt[colour]++
     }
  }
# Если значение второго поля совпадает с индексом массива student,
# содержимое соответствующего элемента массива увеличивается на единицу.
  {for(senior_or_junior in student)
     {if($2==senior_or_junior)
        student[senior_or_junior]++
     }
  }

# Вывод полученных результатов
END {
  for(colour in belt)
     print "The club has", belt[colour], colour, "belts"
  for(senior_or_junior in student)
     print "The club has", student[senior_or_junior], \
           senior_or_junior, "students"
}
```

В процедурной части шаблона `BEGIN` в переменную `FS` записывается символ '#' — разделитель полей во входном файле. Затем создаются два массива, `belt` и `student`, индексами которых являются соответственно названия поясов и названия категорий учеников. Оба массива остаются неинициализированными, их тип не определен.

В первом цикле `for` значение первого поля каждой записи входного файла сравнивается с названием индекса массива `belt` (индекс равен "Yellow", "Orange" или "Red"). Если обнаруживается совпадение, выполняется приращение элемента, хранящегося в массиве по этому индексу. Поскольку операция ++ (инкремент) определена для целочисленных значений, утилита `awk` считает, что массив целочисленный, и инициализирует его элементы значением 0.

Во втором цикле `for` значение второго поля сравнивается с названием индекса массива `student` (индекс равен "Junior" или "Senior"). Результат операции вычисляется так же, как и в первом цикле.

В процедурной части шаблона `END` осуществляется вывод итоговых результатов.

```
$ belts.awk grade_student.txt
The club has 2 Red belts
The club has 2 Orange belts
The club has 3 Yellow belts
The club has 7 Senior students
The club has 8 Junior students
```

9.3. Заключение

Язык программирования awk может показаться сложным для изучения, но если осваивать его на примерах употребления отдельных команд и небольших сценариев, то процесс обучения не будет слишком трудным. В данной главе мы изучили основы awk, не углубляясь в детали, но, тем не менее, получили необходимый минимум сведений, относящихся к этому языку. Утилита awk является важным инструментом shell-программирования, и чтобы применять ее, вам не обязательно быть экспертом по языку awk.

ГЛАВА 10

Работа с редактором sed

Редактор sed относится к текстовым редакторам потокового типа и не является интерактивной программой. Он предназначен для обработки текстовых данных, поступающих из файла или стандартного входного потока. Отличительной особенностью редактора sed является то, что он позволяет модифицировать файлы в фоновом режиме. Задача пользователя сводится к тому, чтобы задать последовательность команд редактирования, а всю остальную работу выполнит сам редактор. Кроме того, sed допускает внесение всех изменений за один проход, а это превращает его в достаточно эффективный и, что важнее всего, быстродействующий инструмент редактирования текста.

В этой главе рассматриваются следующие темы:

- синтаксис команд sed;
- поиск строк по номерам и с использованием регулярных выражений;
- изменение входного текста и добавление текста в выходной поток;
- примеры команд и сценариев sed.

Команды sed вводятся в командной строке либо размещаются в файле сценария подобно тому, как это делается в случае с утилитой awk. При использовании sed важно помнить следующее: этот редактор оставляет без изменения исходный файл независимо от того, какая команда выполняется. Копия входного потока помещается в буфер редактирования, а все изменения направляются на экран или переадресуются в выходной файл.

Поскольку редактор sed не предназначен для работы в интерактивном режиме, текстовые строки, подлежащие изменению, отбираются либо по номерам, либо на основании соответствия регулярному выражению.

10.1. Чтение и обработка данных в sed

Общая схема работы редактора sed такова:

1. Редактор последовательно извлекает строки текста из файла или стандартного входного потока и копирует их в буфер редактирования.
2. Затем он считывает первую команду из командной строки или сценария, осуществляет поиск строки с указанным номером или строки, соответствующей шаблону, и применяет к ней эту команду.
3. Второй пункт повторяется до тех пор, пока не будет исчерпан список команд.

10.2. Вызов редактора sed

Вызвать редактор sed можно тремя способами:

1. Ввести набор команд sed в командной строке.
2. Поместить набор команд sed в файл и передать его редактору sed в командной строке.
3. Поместить набор команд sed в файл сценария и сделать его выполняемым.

Если редактор sed вызывается для выполнения одиночных команд, формат командной строки будет таким:

```
sed [опции] 'команды' входной_файл
```

Как и в awk, не забудьте заключить команды в одинарные кавычки, чтобы защитить от интерпретации находящиеся в них специальные символы.

Если команды sed помещены в отдельный файл, командная строка примет следующий вид:

```
sed [опции] -f файл_сценария входной_файл
```

Если файл сценария является исполняемым, запустить его на выполнение следует таким образом:

```
файл_сценария [опции] входной_файл
```

Когда входной файл не указан, sed будет ожидать поступления данных из стандартного входного потока: с клавиатуры или из канала.

Ниже перечислены основные опции редактора sed и описано их назначение:

`-n` Запрет вывода на экран. При наличии этой опции редактор sed не будет записывать обрабатываемые им строки в стандартный выходной поток, тогда как по умолчанию отображается каждая входная строка. Осуществить вывод нужной строки можно будет только с помощью команды p (рассматривается ниже).

`-e` Следующей командой будет команда редактирования. Эта опция используется в том случае, когда команд редактирования несколько. Если же имеется только одна такая команда, то указывать данную опцию не нужно, хотя ее наличие и не является ошибкой.

`-f` Эта опция используется при подключении файла сценария.

10.2.1. Сохранение выходных данных

Если требуется сохранить проделанные изменения, просто перенаправьте результаты работы редактора sed в файл, как показано ниже:

```
$ sed 'команды' входной_файл > выходной_файл
```

10.2.2. Синтаксис команд

Общий синтаксис команд редактора sed таков:

```
[адрес1[,адрес2]][!]команда [аргументы]
```

Команда состоит из одной буквы или одного символа (список основных команд представлен ниже). Аргументы требуются лишь нескольким командам, в частности, команде `s`. Элементы, представленные в квадратных скобках, являются необязательными, а сами скобки набирать не нужно.

Просмотр входного файла по умолчанию начинается с первой строки. Существует два способа адресации строк:

1. По номерам.
2. С помощью регулярных выражений (о них рассказывалось в главе 7).

В команде может быть указано два адреса, один адрес или ни одного адреса. В следующей таблице описаны все возможные правила отбора строк в зависимости от того, сколько компонентов адреса задано (табл. 10.1).

Таблица 10.1. Правила отбора строк в редакторе `sed`

Адрес	Отбираемые строки
нет адреса	Все строки входного файла
x	Строка с номером *x*
x,*y*	Все строки с номерами в диапазоне от *x* до *y*
/шаблон/	Все строки, соответствующие шаблону
/шаблон1/,*/шаблон2/*	Группа строк, начиная от строки, соответствующей первому шаблону, и заканчивая строкой, которая соответствует второму шаблону; подобных групп во входном файле может быть несколько
/шаблон/,*x*	Группа строк, начиная от строки, соответствующей шаблону, и заканчивая строкой с указанным номером
x,*/шаблон/*	Группа строк, начиная от строки с указанным номером и заканчивая строкой, соответствующей шаблону
!	Все строки, не соответствующие заданному адресу
$	Последняя строка входного файла

Некоторые команды, в частности, `a`, `i`, `r`, `q` и `=`, требуют указания только одного адреса.

10.2.3. Основные команды редактирования

Ниже представлен список основных команд, имеющихся в редакторе `sed` (табл. 10.2).

Таблица 10.2. Основные команды `sed`

`p`	Вывод адресуемых строк
`=`	Вывод номеров адресуемых строк
`a`	Добавление заданного текста после каждой адресуемой строки
`i`	Вставка заданного текста перед каждой адресуемой строкой
`c`	Замена адресуемого текстового блока заданным текстом
`d`	Удаление адресуемых строк
`s`	Замена указанного шаблона заданным текстом в каждой адресуемой строке
`w`	Добавление адресуемых строк в указанный файл

r	Чтение текста из указанного файла и добавление его после каждой адресуемой строки
q	Завершение работы после того, как достигнута адресуемая строка
l	Вывод адресуемых строк с отображением непечатаемых символов в виде ASCII-кодов и переносом длинных строк

С помощью фигурных скобок можно объединить несколько команд в группу. Возможны два синтаксиса группировки:

```
[адрес1[,адрес2]]{
   команда1
...
   командаN
}
```

или

```
{адрес1[,адрес2]]{команда1; ...командаN;}
}
```

В первом случае каждая команда записывается в отдельной строке, а разделителем команд является символ новой строки. Во втором случае команды записываются последовательно, отделяясь друг от друга точкой с запятой, которая ставится также после завершающей команды.

Ниже приведен текстовый файл *quote.txt*, который используется во многих примерах данной главы:

```
$ cat quote.txt
The honeysuckle band played all night long for only $90.
It was an evening of splendid music and company.
Too bad the disco floor fell through at 23:10.
The local nurse Miss P.Neave was in attendance.
```

10.3. Регулярные выражения

Редактор sed распознает базовые регулярные выражения, которые мы уже рассматривали в главе 7. Дополнительные особенности появляются только в шаблонах поиска и замены в команде s. С помощью операторов \(и \) можно сохранить до девяти шаблонов поиска во временном буфере, с тем чтобы в шаблоне замены обратиться к ним с помощью оператора *n*, где *n* — номер сохраненного шаблона. Метасимвол & позволяет в шаблоне замены сослаться на фрагмент строки, соответствующий шаблону поиска.

10.4. Вывод строк (команда p)

Рассмотрим, как в редакторе sed осуществляется поиск строк и вывод их на экран.

10.4.1. Отображение строки по номеру

Команда p (print) имеет такой формат:

```
[адрес1[,адрес2]]p
```

Для отображения строки входного файла достаточно указать ее номер, например:

```
$ sed '2p' quote.txt
The honeysuckle band played all night long for only $90.
It was an evening of splendid music and company.
It was an evening of splendid music and company.
Too bad the disco floor fell through at 23:10.
The local nurse Miss P.Neave was in attendance.
```

Что было сделано неправильно? Ведь требовалось отобразить только строку номер 2, однако в результате были выведены на экран все строки файла, причем вторая строка — дважды. Причина подобного поведения заключается в том, что по умолчанию редактор sed отображает каждую просматриваемую строку. Чтобы избежать этого, воспользуемся опцией *-n*:

```
$ sed -n '2p' quote.txt
It was an evening of splendid music and company.
```

10.4.2. Отображение строк из заданного диапазона

Предположим, требуется вывести строки с номерами от 1 до 3. В этом случае следует указать два адреса, разделенные запятой:

```
$ sed -n '1,3p' quote.txt
The honeysuckle band played all night long for only $90.
It was an evening of splendid music and company.
Too bad the disco floor fell through at 23:10.
```

10.4.3. Поиск строк, соответствующих шаблону

В следующем примере показано, как найти строку, содержащую слово "Neave":

```
$ sed -n '/Neave/p' quote.txt
The local nurse Miss P.Neave was in attendance.
```

10.4.4. Поиск по шаблону и номеру строки

Если адрес представлен в виде шаблона, редактор sed находит все строки, соответствующие этому шаблону. Как можно уточнить местонахождение строки? Рассмотрим пример. Предположим, требуется найти слово "The" в последней строке файла *quote.txt*. Если воспользоваться поиском по шаблону, то будет получено две строки:

```
$ sed -n '/The/p' quote.txt
The honeysuckle band played all night long for only $90.
The local nurse Miss P.Neave was in attendance.
```

Чтобы остановить свой выбор на последней строке, следует указать ее номер перед шаблоном:

```
$ sed -n '4,/The/p' quote.txt
The local nurse Miss P.Neave was in attendance.
```

10.4.5. Поиск специальных символов

Если требуется найти строку, содержащую символ '$', который в редакторе sed имеет специальное назначение, следует защитить этот символ от интерпретации с помощью обратной косой черты, как показано ниже:

```
$ sed -n '/\$/p' quote.txt
The honeysuckle band played all night long for only $90.
```

10.4.6. Поиск первой строки

Для вывода первой строки входного файла достаточно указать ее номер:

```
$ sed -n '1p' quote.txt
The honeysuckle band played all night long for only $90.
```

10.4.7. Поиск последней строки

Чтобы сослаться на последнюю строку входного файла, воспользуйтесь метасимволом '$':

```
$ sed -n '$p' quote.txt
The local nurse Miss P.Neave was in attendance.
```

10.4.8. Отображение всего файла

Если требуется отобразить весь файл, задайте диапазон строк от первой до последней:

```
$ sed -n '1,$p' quote.txt
The honeysuckle band played all night long for only $90.
It was an evening of splendid music and company.
Too bad the disco floor fell through at 23:10.
The local nurse Miss P.Neave was in attendance.
```

10.5. Вывод номеров строк (команда =)

Команда = имеет следующий формат:

```
[адрес]=
```

Она предназначена для вывода номера строки, соответствующей заданному адресу. Рассмотрим пример:

```
$ sed '/music/=' quote.txt
The honeysuckle band played all night long for only $90.
2
It was an evening of splendid music and company.
Too bad the disco floor fell through at 23:10.
The local nurse Miss P.Neave was in attendance.
```

В данном случае отображается весь файл, причем перед строкой, содержащей слово "music", выводится ее номер. Из этого можно сделать заключение, что команда = выполняется перед тем, как текущая строка будет выведена на экран.

Если же требуется узнать только номер строки, задайте опцию -n:

```
$ sed -n '/music/=' quote.txt
2
```

Можно также отобразить и строку, и ее номер. Для этого следует воспользоваться опцией -e, позволяющей указать несколько команд подряд. Первая команда выводит строку, в которой найдено совпадение с шаблоном, а вторая — номер этой строки:

```
$ sed -n -e '/music/p' -e '/music/=' quote.txt
It was an evening of splendid music and company.
2
```

10.6. Добавление текста (команда a)

Для добавления текста предназначена команда a (append), которая вставляет одну или несколько строк текста после адресуемой строки. Формат команды таков:

```
[адрес]a\
текст\
текст\
...
текст
```

Адрес может быть представлен в виде номера строки либо регулярного выражения. Во втором случае найденных строк может быть несколько. При добавлении текста отсутствует возможность задать диапазон строк. Допускается указание только одного шаблона адреса. Если адрес, по которому помещается текст, не указан, тогда команда будет применена к каждой строке входного файла.

Обратите внимание на присутствие символа обратной косой черты в конце каждой добавляемой строки, а также после самой команды a. Этот метасимвол защищает от интерпретации символ новой строки. В последней строке указывать обратную косую черту не требуется, поскольку концевой символ новой строки в этом случае является признаком конца команды.

Добавляемый текст записывается в стандартный выходной поток и не дублируется во входном буфере, поэтому не подлежит редактированию, т.е. на него нельзя сослаться в последующих шаблонах поиска. Чтобы иметь возможность отредактировать полученный текст, необходимо сохранить результаты работы редактора sed в новом файле и применить команды редактирования уже к нему.

10.7. Создание файла сценария

Конечно, ввод многострочных команд в режиме командной строки не слишком удобен и чреват ошибками, поэтому лучше всего размещать такие команды в файлах сценариев. Кроме того, в сценариях допускается наличие пустых строк и комментариев, что облегчает восприятие самих сценариев.

Создайте новый файл с именем *append.sed* и добавьте в него показанные ниже команды:

```
$ cat append.sed
#! /bin/sed -f
```

```
/company/a\
Then suddenly it happened.
```

Теперь сделайте этот файл исполняемым:

```
$ chmod u+x append.sed
```

и запустите его на выполнение:

```
$ append.sed quote.txt
The honeysuckle band played all night long for only $90.
It was an evening of splendid music and company.
Then suddenly it happened.
Too bad the disco floor fell through at 23:10.
The local nurse Miss P.Neave was in attendance.
```

Если вместо показанного результата будет выдано сообщение об ошибке "command not found" (команда не найдена), значит, переменная среды `$PATH`, которая содержит список имен каталогов, просматриваемых в поиске исполняемых файлов, не включает имя текущего каталога. В этом случае необходимо явно указать, что исполняемый файл находится в текущем каталоге:

```
$ ./append.sed quote.txt
```

Рассмотрим, что делает сценарий `append.sed`. Первая его строка является системной командой, которая указывает, какая программа выполняет данный сценарий. Формат этой команды мы уже рассматривали при знакомстве с файлами сценариев `awk` в предыдущей главе. Как и утилита `awk`, редактор `sed`, как правило, находится в каталоге */bin*.

Далее в сценарии находится команда `a`, которая ищет во входном файле строку, содержащую слово "company", и вставляет после нее предложение "Then suddenly it happened".

10.8. Вставка текста (команда i)

Команда `i` (insert) аналогична команде `a`, только вставляет текст не после, а перед адресуемой строкой. Как и при добавлении текста, допускается указание только одного шаблона адреса. Ниже приведен общий формат команды:

```
[адрес]i\
текст\
текст\
...
текст
```

В следующем сценарии предложение "Utter confusion followed" вставляется перед строкой, содержащей слово "attendance":

```
$ cat insert.sed
#! /bin/sed -f
/attendance/i\
Utter confusion followed.
```

Результаты работы данного сценария будут такими:

```
$ insert.sed quote.txt
The honeysuckle band played all night long for only $90.
It was an evening of splendid music and company.
Too bad the disco floor fell through at 23:10.
Utter confusion followed.
The local nurse Miss P.Neave was in attendance.
```

Для указания места вставки текста можно было бы воспользоваться номером строки, в данном случае 4:

```
#! /bin/sed -f
4i\
Utter confusion followed.
```

10.9. Изменение текста (команда c)

Команда c (change) заменяет новым текстом каждую адресуемую строку. Если выбрана группа строк, вся группа заменяется одной копией текста. Формат команды c таков:

```
[адрес1[,адрес2]]c\
текст\
текст\
...
текст
```

В следующем примере первая строка файла *quote.txt* заменяется новой строкой:

```
$ cat change.sed
#! /bin/sed -f
1c\
The Office Dibble band played well.
```

Прежде чем выполнять этот сценарий, не забудьте сделать его исполняемым:

```
$ chmod u+x change.sed
$ change.sed quote.txt
The Office Dibble band played well.
It was an evening of splendid music and company.
Too bad the disco floor fell through at 23.10.
The local nurse Miss P.Neave was in attendance.
```

Команды изменения, добавления и вставки текста можно применять к одному и тому же файлу. Ниже приведен пример такого сценария, снабженный необходимыми комментариями:

```
$ cat mix.sed
#! /bin/sed -f

# Изменяем строку номер 1
1c\
The Dibble band were grooving.

# Вставляем строку
```

```
/evening/i\
They played some great tunes.

# Изменяем последнюю строку
$c\
Nurse Neave was too tipsy to help.

# Добавляем строку после строки номер 3
3a\
Where was the nurse to help?
```

Вот что получится в результате выполнения этого сценария:

```
$ mix.sed quote.txt
The Dibble band were grooving.
They played some great tunes.
It was an evening of splendid music and company.
Too bad the disco floor fell through at 23:10.
Where was the nurse to help?
Nurse Neave was too tipsy to help.
```

10.10. Удаление текста (команда d)

Для удаления текста предназначена команда d (deletc), имеющая следующий формат:

```
[адрес1[,адрес2]]d
```

Адрес может быть указан в виде номера строки или регулярного выражения.

Рассмотрим примеры. В первом из них будет удалена первая строка входного файла:

```
$ sed '1d' quote.txt
It was an evening of splendid music and company.
Too bad the disco floor fell through at 23:10.
The local nurse Miss P.Neave was in attendance.
```

В следующем примере удаляются строки 1—3:

```
$ sed '1,3d' quote.txt
The local nurse Miss P.Neave was in attendance.
```

В этом примере удаляется последняя строка:

```
$ sed '$d' quote.txt
The honeysuckle band played all night long for only $90.
It was an evening of splendid music and company.
Too bad the disco floor fell through at 23:10.
```

Можно также удалить строку, в которой найдено совпадение с регулярным выражением. В показанном ниже примере удаляется строка, содержащая слово "Neave":

```
$ sed '/Neave/d' quote.txt
The honeysuckle band played all night long for only $90.
It was an evening of splendid music and company.
Too bad the disco floor fell through at 23:10.
```

10.11. Замена подстроки (команда s)

Команда s (substitute) осуществляет во всех адресуемых строках замену подстроки, соответствующей заданному шаблону, указанной подстрокой. Формат команды таков:

```
[адрес1[,адрес2]]s/шаблон_поиска/шаблон_замены/[флаги]
```

Ниже перечислены возможные флаги:

g	Замена в адресуемой строке каждой подстроки, соответствующей шаблону (по умолчанию заменяется лишь самая первая подстрока каждой адресуемой строки)
n	Замена *n*-й подстроки, соответствующей шаблону (*n* — любое число в диапазоне от 1 до 512)
p	Вывод на экран строки, в которой была произведена замена; если в строке сделано несколько замен, она будет отображена соответствующее число раз
w *имя_файла*	Запись измененной строки в указанный файл

В следующем примере осуществляется замена слова "night" словом "NIGHT":

```
$ sed -n 's/night/NIGHT/p' quote.txt
The honeysuckle band played all NIGHT long for only $90.
```

Если требуется удалить из строки символ '$', оставьте шаблон замены пустым (не забывайте, что в редакторе sed знак доллара является метасимволом, поэтому он должен быть защищен обратной косой чертой).

```
$ sed -n 's/\$//p' quote.txt
The honeysuckle band played all night long for only 90.
```

Флаг g (global) позволяет выполнить глобальную подстановку шаблона замены на место шаблона поиска в пределах каждой адресуемой строки. Предположим, например, что мы хотим заменить все точки в файле *quote.txt* восклицательными знаками. Следующая команда выполнит работу не полностью:

```
$ sed 's/\./!/' quote.txt
The honeysuckle band played all night long for only $90!
It was an evening of splendid music and company!
Too bad the disco floor fell through at 23:10!
The local nurse Miss P!Neave was in attendance.
```

Обратите внимание на последнюю строку: в ней точка встречается дважды, но замене подвергся только первый символ. Для исправления подобной ситуации нужно указать флаг g:

```
$ sed 's/\./!/g' quote.txt
The honeysuckle band played all night long for only $90!
It was an evening of splendid music and company!
Too bad the disco floor fell through at 23:10!
The local nurse Miss P!Neave was in attendance!
```

С помощью флага w (write) можно указать файл, в который будут записаны все модифицируемые строки. В показанном ниже примере осуществляется замена слова

"splendid" словом "SPLENDID", а все строки, где была выполнена эта замена, помещаются в файл *sed.out*:

```
$ sed -n 's/splendid/SPLENDID/w sed.out' quote.txt
```

Вот каким будет содержимое этого файла:

```
$ cat sed.out
It was an evening of SPLENDID music and company.
```

Ссылка на искомую подстроку с помощью метасимвола &

Метасимвол & позволяет сослаться в шаблоне замены на подстроку, соответствующую шаблону поиска. Например, в следующей команде слово "Miss" либо "miss" заменяется фразой "lovely Miss Joan" или "lovely miss Joan" соответственно:

```
$ sed -n 's/[Mm]iss/lovely & Joan/p' quote.txt
The local nurse lovely Miss Joan P.Neave was in attendance
```

Заметьте, что пробелы также являются частью шаблона замены.

10.12. Вывод строк в файл (команда w)

Подобно тому как оператор > применяется для перенаправления результатов работы программы в файл, команда w (write) редактора sed позволяет записать в указанный файл строки, отобранные по заданному шаблону адреса. Формат этой команды таков:

```
[адрес1[,адрес2]]w имя_файла
```

Если файл не существует, он будет создан, если существует — его содержимое будет перезаписано. Если в сценарии встречается несколько команд w, направляющих результаты в один и тот же файл, данные всех команд, кроме первой, будут добавляться в конец файла.

Рассмотрим пример:

```
$ sed '1,2w sed.out' quote.txt
```

Здесь содержимое файла *quote.txt* выводится на экран, а строки с номерами 1 и 2 отправляются в файл с именем *sed.out*.

```
$ cat sed.out
The honeysuckle band played all night long for only $90.
It was an evening of splendid music and company.
```

В следующем примере осуществляется поиск строки, содержащей слово "Neave", и если такая строка найдена, она записывается в файл *sed.out*:

```
$ sed -n '/Neave/w sed.out' quote.txt
$ cat sed.out
The local nurse Miss P.Neave was in attendance.
```

10.13. Чтение строк из файла (команда r)

В процессе обработки входного файла редактор sed позволяет читать текст из другого файла и добавлять его к текущему содержимому буфера, размещая после каждой строки, соответствующей шаблону адреса. Формат предназначенной для этого команды r (read) таков:

```
[адрес]r имя_файла
```

Давайте создадим небольшой файл с именем *sedex.txt*:

```
$ cat sedex.txt
Boom boom went the music.
```

В следующем примере содержимое этого файла выводится на экран после строки файла *quote.txt*, содержащей слово "company":

```
$ sed '/company/r sedex.txt' quote.txt
The honeysuckle band played all night for only $90.
It was an evening of splendid music and company.
Boom boom went the music.
Too bad the disco floor fell through at 23:10.
The local nurse Miss P.Neave was in attendance.
```

10.14. Досрочное завершение работы (команда q)

Иногда требуется завершить работу редактора sed сразу же после нахождения первого совпадения с шаблоном. Эту задачу решает команда q (quit), имеющая следующий формат:

```
[адрес]q
```

Обратимся к примеру. Допустим, требуется осуществить поиск строки, содержащей такой шаблон:

```
/\<.a.\{0,2\}\>/
```

Этому шаблону соответствует любое слово (выражение \< обозначает начало слова, а выражение \> — его конец), в котором вторым символом является буква 'a', а за ней идет не более двух символов. В файле *quote.txt* таких слов четыре:

- строка 1 — band,
- строка 2 — was,
- строка 3 — bad,
- строка 4 — was.

Показанная ниже команда находит строку, в которой шаблон встречается первый раз, после чего завершает работу:

```
$ sed '/\<.a.\{0,2\}\>/q' quote.txt
The honeysuckle band played all night long for only $90.
```

10.15. Отображение управляющих символов (команда l)

Иногда даже в текстовых файлах содержатся различного рода непечатаемые символы. Это может быть следствием неправильного ввода данных в текстовом редакторе или ошибок конвертации при загрузке файлов из других систем. При выводе таких файлов на экране могут быть получены странные результаты, когда вместо непечатаемого символа отображается один или несколько обычных символов непонятного происхождения. Разобраться в таких ситуациях помогает команда `cat -v`, которая помечает начало замещающей последовательности символом '^' (знак крышки).

Предположим, вы обнаружили незнакомый файл *func.txt* и хотите узнать его содержимое:

```
$ cat func.txt
This is the F1 key:P
This is the F2 key:Q
```

Символы 'P' и 'Q' на концах строк кажутся подозрительными. Попробуем применить команду `cat -v`:

```
$ cat -v func.txt
This is the F1 key:^[OP
This is the F2 key:^[OQ
```

Так и есть! Это не буквы 'P' и 'Q', а управляющие символы, хотя и непонятно, с помощью каких клавиш они были сгенерированы.

Аналогичным образом будет вести себя и редактор `sed` при работе с данным файлом. Если вы попытаетесь просмотреть его содержимое, будет выдано следующее:

```
$ sed -n '1,$p' func.txt
This is the F1 key:P
This is the F2 key:Q
```

В редакторе существует команда `l` (list), аналог рассмотренной выше команды `cat -v`. Формат команды `l` таков:

```
[адрес1[,адрес2]]l
```

Ее действие равносильно применению команды `p`, но при этом все непечатаемые символы заменяются восьмеричными ASCII-кодами (кроме того, длинные строки, выходящие за пределы экрана, разбиваются на части, а конец каждой строки помечается символом '$'). Вот что получится, если применить эту команду к файлу *func.txt*:

```
$ sed -n '1,$l' func.txt
This is the F1 key:\033OP$
This is the F2 key:\033OQ$
```

Теперь ситуация немного проясняется. По таблице ASCII-кодов можно узнать, что восьмеричный код 033 соответствует непечатаемому символу `ESC`. Именно он в выводе команды `cat -v` обозначается как `^[`. За ним идут два обычных символа: сначала 'O', затем 'P' либо 'Q'. Таким образом, мы имеем дело с двумя клавишами, каждая из которых сгенерировала трехсимвольную последовательность: одна — `ESC-O-P`, а вторая — `ESC-O-Q`. Когда файл отображается в обычном режиме, каждому символу соответствует одно знакоместо, поэтому первые два символа последовательности отбрасываются и остается последний: 'P' и 'Q' соответственно.

Нет системной команды, которая позволяла бы узнать, какие клавиши генерируют эти коды. Подобные сведения находятся в базах данных termcap и terminfo, хранящих установки терминала, но знакомство с этими базами данных выходит за рамки нашей книги. Проще всего пойти экспериментальным путем: ввести команду cat или cat -v без указания входного файла, с тем чтобы попробовать самостоятельно определить, нажатия каких клавиш приводят к отображению на экране нужных последовательностей символов. В нашем случае последовательность ESC-O-P генерируется клавишей [F1], а последовательность ESC-O-Q — клавишей [F2], хотя в общем это зависит от установок терминала.

Если вам интересно, как можно создать файл *func.txt*, ниже описана возможная процедура:

1. Загрузите редактор vi.
2. Перейдите в режим вставки с помощью команды i и введите первую строку файла вплоть до двоеточия.
3. Нажмите [Ctrl+V], при этом появится символ '^'.
4. Нажмите клавишу [F1], при этом отобразится последовательность [OP.
5. Нажмите клавишу [Enter] и повторите пункты 2—4 для второй строки файла (в конце нажимается клавиша [F2]).
6. Нажмите клавишу [Esc], чтобы выйти из режима вставки.
7. Введите команду w func.txt, чтобы сохранить файл.
8. Введите команду q, чтобы выйти из редактора vi.

10.16. Дополнительные примеры использования редактора sed

Выше были описаны основные команды sed. Далее мы рассмотрим ряд практических примеров применения редактора sed.

10.16.1. Обработка управляющих символов

Одной из задач, для которых редактор sed используется весьма часто, является удаление управляющих, непечатаемых и просто посторонних символов из файлов, которые были загружены из других систем. Ниже приведен фрагмент подобного файла *dos.txt*:

```
$ cat -v dos.txt
12332##DISO##45.12^M
00332##LPSO##23.11^M
01299##USPD##34.46^M
...
```

Вот что необходимо сделать:

1. Заменить все знаки решетки ('#') пробелом.
2. Удалить из первого поля каждой строки все ведущие нули.
3. Удалить в конце каждой строки последовательность ^M, генерируемую непечатаемым символом CR — возврат каретки (ASCII-код 13).

Примечание:

В ряде систем, в которых мне приходилось выполнять подобные преобразования, в конце строки стоит символ перевода строки LF (ASCII-код 10, отображается как ^@). Подобная проблема нередко возникает при передаче файлов между двумя системами, в одной из

которых символ новой строки (\n) заменяется последовательностью CR/LF, а в другой — только одним из этих двух управляющих символов.

Задача 1. Удаление всех символов решетки реализуется без особого труда. Для этого достаточно выполнить команду замены s с флагом глобальной подстановки g, указав при этом, что один или более символов '#', идущих подряд, должны быть заменены пробелом:

```
$ sed 's/##*/ /g' dos.txt | cat -v
12332 DISO 45.12^M
00332 LPSO 23.11^M
01299 USPD 34.46^M
```

Задача 2. Для удаления всех ведущих нулей следует в команде s оставить шаблон замены пустым, а шаблон поиска задать таким: /^0*/. Он означает, что требуется найти любое количество нулей, стоящих в начале строки.

```
$ sed 's/^0*//' dos.txt | cat -v
12332##DISO##45.12^M
332##LPSO##23.11^M
1299##USPD ##34.46^M
```

Задача 3. Чтобы избавиться от управляющих символов ^M в конце строк, необходимо также применить команду s, оставив шаблон замены пустым. А вот при формировании шаблона поиска следует учесть, что мы имеем дело с непечатаемым символом. Нельзя просто ввести символы '^' и 'M', так как полученный шаблон будет означать, что мы ищем букву 'M', стоящую в начале строки. Чтобы действительно создать нужный нам непечатаемый символ, необходимо нажать [Ctrl+V], а затем — клавишу [Enter]. Результат будет выглядеть как регулярное выражение ^M, но на самом деле это экранное представление символа CR.

```
$ sed 's/^M//' dos.txt | cat -v
12332##DISO##45.12
00332##LPSO##23.11
01299##USPD##34.46
```

Теперь можно попробовать объединить три команды в одну с помощью опции -e:

```
$ sed -e 's/^0*//' -e 's/^M//' -e 's/##*/ /g' dos.txt | cat -v
12332 DISO 45.12
332 LPSO 23.11
1299 USPD 34.46
```

Выходные данные редактора sed передаются по каналу команде cat -v, которая позволяет убедиться, что вся работа, включая удаление непечатаемых символов, выполнена правильно.

Приведенные выше команды удобнее поместить в файл сценария. Назовем его *dos.sed*. Вот его текст:

```
$ cat dos.sed
#! /bin/sed -f
# Имя: dos.sed
# Командная строка: dos.sed dos.txt

# Избавляемся от символа решетки
s/##*/ /g
```

```
# Удаляем ведущие нул
s/^0*//

# Удаляем символы возврата каретки
s/^M//
```

На входе данного сценария следует указать файл *dos.txt*. Сценарий "исправит" этот файл и выведет результат на экран. Если же ввести следующую командную строку:

```
$ dos.sed dos.txt | tee dos.txt
```

то результат будет записан обратно в файл *dos.txt*!

10.16.2. Обработка отчетов

Я часто сталкиваюсь с необходимостью форматировать данные, возвращаемые инструкциями SQL. Для этого приходится писать комплексные сценарии, в которых сразу несколько текстовых фильтров выполняют работу совместно. Рассмотрим результаты выполнения некоторой инструкции SQL, выполняющей обращение к одной из таблиц базы данных:

```
Database  Size (MB)  Date created
--------------------------------
GOSOUTH   2244       12/11/97
TRISUD    5632       8/9/99

(2 rows affected)
```

Из этой информации нас, предположим, интересуют только имена баз данных, находящиеся в первой колонке. Чтобы их извлечь, необходимо выполнить следующую последовательность действий:

1. Удалить пунктирную линию с помощью команды `s/--*//g`.
2. Удалить все пустые строки с помощью команды `/^$/d`.
3. Удалить последнюю строку с помощью команды `$d`.
4. Удалить первую строку с помощью команды `1d`.
5. Отобразить первый столбец с помощью команды `awk '{print $1}'`.

Ниже приведена соответствующая цепочка команд:

```
$ sed 's/--*//g' -e '/^$/d' -e '$d' -e '1d' sql.txt | awk '{print $1}'
GOSOUTH
TRISUD
```

10.16.3. Добавление текста

В процессе потоковой обработки файла мне иногда требуется добавить к каждой проверенной строке какой-нибудь текст, сообщающий о том, как прошла обработка. Предположим, имеется такой файл:

```
$ cat ok.txt
AC456
AC492169
AC9967
AC88345
```

Наша задача состоит в добавлении слова "Passed" (обработано) в конец каждой строки. Решить ее несложно. Достаточно в шаблоне поиска указать метасимвол '$', означающий конец строки, а в шаблоне замены — пробел и искомое слово:

```
$ sed 's/$/ Passed/' ok.txt
AC456 Passed
AC492169 Passed
AC9967 Passed
AC88345 Passed
```

10.16.4. Удаление начальной косой черты в путевом имени

Ниже показано, как с помощью редактора sed можно быстро удалить начальную косую черту из имени текущего каталога:

```
$ cd /usr/local
$ echo $PWD | sed 's/^\///g'
usr/local
```

Имя текущего каталога хранится в переменной среды $PWD. Эта переменная обновляется всякий раз, когда выполняется команда cd. Команда echo передает значение переменной по каналу редактору sed, который выполняет несложную обработку: находит в начале строки (метасимвол '^') символ косой черты (защищен от интерпретации обратной косой чертой) и заменяет его пустой подстрокой.

10.17. Заключение

Редактор sed является эффективным инструментом фильтрации текста. Он дает возможность находить во входном потоке требуемые строки и последовательности символов и модифицировать их с наименьшими затратами времени и усилий. Как было показано, для извлечения из файлов требуемой информации вовсе не обязательно прибегать к написанию больших сценариев.

В этой главе мы коснулись лишь основ работы с sed, но даже полученных знаний достаточно, чтобы с помощью sed решать многие задачи, возникающие в процессе обработки текста.

ГЛАВА 11

Дополнительные утилиты работы с текстом

Существует большое количество утилит, предназначенных для сортировки, объединения, разбиения и прочей обработки текстовых файлов. В этой главе мы познакомимся с такими командами:

- `sort`;
- `uniq`;
- `join`;
- `cut`;
- `paste`;
- `split`.

11.1. Сортировка файлов с помощью команды sort

Команда `sort` позволяет выполнять сортировку входного потока по различным полям (ключам сортировки). Это довольно мощная команда, которая весьма полезна при обработке журнальных файлов или реорганизации текстовых столбцов в файлах. В то же время следует быть внимательным при использовании ее многочисленных опций, так как зачастую можно получить неожиданные результаты. Не всегда понятна связь между указанной опцией и результатами, возвращаемыми командой `sort`. Некоторые опции перекрывают друг друга и, таким образом, допускают неоднозначную трактовку.

Мы не станем вдаваться в детали различных алгоритмов сортировки и рассматривать все возможные комбинации опций команды `sort`. Будут описаны лишь важнейшие из опций и представлены многочисленные примеры, позволяющие разобраться в особенностях функционирования этой команды.

11.1.1. Опции команды sort

Команда `sort` имеет следующий формат:

```
sort [опции] [входные_файлы]
```

Команда выполняет конкатенацию указанных входных файлов, сортирует полученный текст и записывает результат в стандартный выходной поток. Если файлы не указаны, ожидается ввод данных с клавиатуры.

Таблица 11.1. Основные опции команды `sort`

`-c`	Проверка того, отсортирован ли файл; сортировка не производится
`-m`	Объединение отсортированных файлов; сортировка не производится
`-u`	Удаление повторяющихся строк
`-o`	Вывод результата не на экран, а в указанный файл
`-b`	Игнорирование начальных пробелов в полях сортировки
`-n`	Включение режима числовой сортировки
`-t`	Задание разделителя полей
`-r`	Сортировка в обратном порядке
+поз1[*-поз2*]	Ключом сортировки становится строка, начинающаяся в позиции *поз1* и заканчивающаяся перед позицией *поз2* (или в конце текущей строки, если второй параметр не указан)*; номера полей и позиции начальных символов отсчитываются от нуля
`-k` *поз1*[*,поз2*]	Ключом сортировки становится строка, начинающаяся в позиции *поз1* и заканчивающаяся в позиции *поз2* (или в конце текущей строки, если второй параметр не указан)*; номера полей и позиции начальных символов отсчитываются от единицы
-n	Поле с номером *n* не должно сортироваться; значение *n* отсчитывается от нуля

* Позиция задается следующим образом: *f*[`.`*c*], где *f* — номер поля, *c* — позиция первого символа ключа *от* начала поля. Если параметр *c* не указан, первым символом ключа считается первый символ поля.

11.1.2. Сохранение результатов сортировки

Чтобы сохранить результаты сортировки, укажите опцию `-o` и выходной файл. Можно также воспользоваться традиционным методом переадресации с помощью оператора >. В следующем примере результаты сортировки сохраняются в файле *results.out*:

```
$ sort video.txt > results.out
```

11.1.3. Тестовый файл

Ниже приведен фрагмент файла *video.txt*, хранящего информацию из базы данных фирмы, которая занимается прокатом видеокассет. В перечень вошли видеокассеты, которые предлагались на протяжении последнего квартала. Поля файла имеют следующее назначение:

1. Название фильма.
2. Код фирмы-дистрибьютора.
3. Количество заказов за последний квартал.
4. Количество заказов за последний год.

```
$ cat video.txt
Boys in Company C:HK:192:2192
Alien:HK:119:1982
```

```
The Hill:KL:63:2972
Aliens:HK:532:4892
Star Wars:HK:301:4102
A Few Good Men:KL:445:5851
Toy Story:HK:239:3972
```

По умолчанию, команда `sort` предполагает, что разделителем полей служит один или несколько пробелов. Если же поля разделены иначе, следует применять опцию `-t`. В нашем файле разделителем является двоеточие. Поэтому в тех примерах, где это необходимо, задается опция `-t:`.

11.1.4. Индексация полей

При работе с командой `sort` не следует забывать, что команда обращается к первому полю как к полю 0, следующее поле имеет номер 1 и т.д. Если номера полей не указаны, вся строка считается единым полем. Обратимся к тестовому файлу и уточним, каким образом команда `sort` разбивает файл на поля:

Поле 0	Поле 1	Поле 2	Поле 3
...			
Star Wars	HK	301	4102
A Few Good Men	KL	445	5851
...			

11.1.5. Проверка факта сортировки файла

Каким образом можно узнать, отсортирован ли данный файл? Если он содержит, например, около 30 строк, то достаточно его просмотреть. А если в нем 400 строк? Примените команду `sort -c`, которая сама определит, отсортирован ли файл:

```
$ sort -c video.txt
sort: disorder on video.txt
```

Команда `sort` считает, что файл не отсортирован. Давайте отсортируем его и посмотрим, что будет:

```
$ sort video.txt | sort -c
$
```

Сообщение не появилось, таким образом, файл является отсортированным.

11.1.6. Простейшая сортировка

В простейшем случае, чтобы отсортировать файл, достаточно передать его имя команде `sort`. Сортировка будет выполнена по строкам:

```
$ sort video.txt
A Few Good Men:KL:445:5851
Alien:HK:119:1982
Aliens:HK:532:4892
Boys in Company C:HK:192:2192
Star Wars:HK:301:4102
The Hill:KL:63:2972
Toy Story:HK:239:3972
```

11.1.7. Сортировка в обратном порядке

Если необходимо отсортировать строки не по возрастанию, а по убыванию, задайте опцию `-r`:

```
$ sort -r video.txt
Toy Story:HK:239:3972
The Hill:KL:63:2972
Star Wars:HK:301:4102
Boys in Company C:HK:192:2192
Aliens:HK:532:4892
Alien:HK:119:1982
A Few Good Men:KL:445:5851
```

11.1.8. Сортировка по заданному полю

В следующем примере файл сортируется по кодам фирм-дистрибьюторов. Поскольку требуемая информация находится во втором поле (ключ сортировки 1), следует указать опцию `+1`. Кроме того, необходимо задать разделитель полей с помощью опции `-t:`, чтобы команда `sort` знала, как найти второе поле.

```
$ sort -t: +1 video.txt
Alien:HK:119:1982
Boys in Company C:HK:192:2192
Toy Story:HK:239:3972
Star Wars:HK:301:4102
Aliens:HK:532:4892
A Few Good Men:KL:445:5851
The Hill:KL:63:2972
```

Обратите внимание на то, что третье и четвертое поля также были отсортированы. Такова стандартная процедура: все последующие поля по умолчанию считаются ключами сортировки, расположенными в порядке убывания приоритета. Причем если вы посмотрите на конечные две строки, то заметите, что к этим полям применялась не числовая, а текстовая сортировка, учитывающая расположение символов в таблице ASCII-кодов. Поэтому поле со значением 445 оказалось расположенным раньше поля со значением 63.

11.1.9. Сортировка по числовому полю

Чтобы корректно отсортировать файл по четвертому, числовому, полю, укажите не только ключ сортировки (`+3`), но и опцию `-n`, включающую режим числовой сортировки. Следующая команда сортирует список фильмов по объемам проката видеокассет за год:

```
$ sort -t: +3n video.txt
Alien:HK:119:1982
Boys in Company C:HK:192:2192
The Hill:KL:63:2972
Toy Story:HK:239:3972
Star Wars:HK:301:4102
Aliens:HK:532:4892
A Few Good Men:KL:445:5851
```

Таким образом, можно заключить, что фильм "A Few Good Men" ("Несколько хороших парней", 1992 г.) является лидером видеопроката в текущем году.

Примечание:

Несмотря на наличие опции `-n`, данный пример работает правильно только потому, что четвертое поле является последним в строке. Причина этого объясняется ниже.

11.1.10. Сортировка с отбрасыванием повторяющихся строк

Иногда приходится иметь дело с файлом, содержащим повторяющиеся строки. Чтобы избавиться от них, достаточно воспользоваться командой `sort` с опцией `-u`. Ниже показан вариант тестового файла, в котором запись о фильме "Alien" ("Чужой", 1977 г.) повторяется дважды:

```
$ cat video.txt
Boys in Company C:HK:192:2192
Alien:HK:119:1982
The Hill:KL:63:2972
Aliens:HK:532:4892
Star Wars:HK:301:4102
A Few Good Men:KL:445:5851
Toy Story:HK:239:3972
Alien:HK:119:1982
```

Вот что получится в результате применения команды `sort -u`:

```
$ sort -u video.txt
A Few Good Men:KL:445:5851
Alien:HK:119:1982
Aliens:HK:532:4892
Boys in Company C:HK:192:2192
Star Wars:HK:301:4102
The Hill:KL:63:2972
Toy Story:HK:239:3972
```

11.1.11. Задание ключа сортировки с помощью опции -k

Команда `sort` позволяет задать ключ сортировки немного по-другому. Если воспользоваться опцией `-k`, то поля (ключи сортировки) можно будет нумеровать, начиная с единицы, а не с нуля, что, в принципе, удобнее. Таким образом, чтобы выполнить сортировку по полю 4, достаточно задать опцию `-k4n`. Это позволит упорядочить список фильмов по объемам видеопроката за год.

```
$ sort -t: -k4n video.txt
Alien:HK:119:1982
Boys in Company C:HK:192:2192
The Hill:KL:63:2972
Toy Story:HK:239:3972
Star Wars:HK:301:4102
Aliens:HK:532:4892
A Few Good Men:KL:445:5851
```

11.1.12. Несколько ключей сортировки

При использовании опций *+позиция* и `-k` следует быть особенно аккуратным. Если вы внимательно прочитали их описание в табл. 11.1, то должны были отметить такой факт: когда не указана конечная позиция, ключ сортировки считается заканчивающимся в конце строки. Подобная тонкость обычно вводит в замешательство новичков, которые пытаются выполнять числовую сортировку или сортировку с несколькими ключами. Если, к примеру, вы ссылаетесь на числовое поле только по номеру, а это поле не является последним в строке, причем за ним идут текстовые поля, данное поле также будет проинтерпретировано как текстовое, вследствие чего будут получены неправильные результаты.

Схожая проблема возникает при работе с несколькими ключами сортировки. Рассмотрим такой пример. Предположим, требуется отсортировать список фильмов по кодам дистрибьюторов (второе поле), а затем по названиям фильмов (первое поле). Если сослаться на поля по номерам, получим следующее:

```
$ sort -t: -k2 -k1 video.txt
Alien:HK:119:1982
Boys in Company C:HK:192:2192
Toy Story:HK:239:3972
Star Wars:HK:301:4102
Aliens:HK:532:4892
A Few Good Men:KL:445:5851
The Hill:KL:63:2972
```

Здесь ссылка на первое поле в действительности означает ссылку на всю строку, т.е. ключ с меньшим приоритетом включает в себя ключ с большим приоритетом, поэтому команда `sort` ведет себя не так, как можно было бы предположить на первый взгляд. Чтобы исправить ситуацию, необходимо четко указать длину каждого из ключей:

```
$ sort -t: -k2,2 -k1,1 video.txt
Alien:HK:119:1982
Aliens:HK:532:4892
Boys in Company C:HK:192:2192
Star Wars:HK:301:4102
Toy Story:HK:239:3972
A Few Good Men:KL:445:5851
The Hill:KL:63:2972
```

Опция `-k2,2` ограничивает ключ сортировки вторым полем, а опция `-k1,1` — первым.

11.1.13. Указание позиции, с которой начинается сортировка

Иногда в качестве ключа сортировки требуется задать не целое поле, а какую-то его часть. В этом случае после номера поля необходимо через точку указать позицию символа, являющегося первым в ключе.

Обратимся к примеру. Допустим, в нашем тестовом файле к каждому коду фирмы-дистрибьютора добавлен код региона дистрибуции:

```
$ cat video.txt
Boys in Company C:HK48:192:2192
Alien:HK57:119:1982
```

```
The Hill:KL23:63:2972
Aliens:HK11:532:4892
Star Wars:HK38:301:4102
A Few Good Men:KL87:445:5851
Toy Story:HK65:239:3972
```

Теперь мы хотим отсортировать файл по кодам регионов. Вот как можно это сделать:

```
$ sort -t: -k2.3,2.4n video.txt
Aliens:HK11:532:4892
The Hill:KL23:63:2972
Star Wars:HK38:301:4102
Boys in Company C:HK48:192:2192
Alien:HK57:119:1982
Toy Story:HK65:239:3972
A Few Good Men:KL87:445:5851
```

Данная команда означает, что ключом сортировки являются третий и четвертый символы второго поля.

11.1.14. Обработка результатов сортировки с помощью команд head и tail

При работе с большими файлами не обязательно выводить на экран весь файл, если требуется просмотреть только его начало и конец. Существуют удобные команды `head` и `tail`, упрощающие подобную задачу. Команда `head` отбирает первые *n* строк файла (по умолчанию 10), а команда `tail` — последние *n* строк (по умолчанию тоже 10).

Предположим, требуется быстро узнать, какой фильм пользуется наименьшим спросом в прокате. Для этого отсортируем файл по четвертому полю и направим результат команде `head`, задав в ней отображение одной строки:

```
$ sort -t: -k4 video.txt | head -1
Alien:HK:119:1982
```

Аналогичным образом можно выяснить, какой фильм чаще всего заказывали в этом году. Формат команды `sort` останется таким же, но результат будет передан команде `tail`:

```
$ sort -t: -k4 video.txt | tail -1
A few Good Men:KL:445:5851
```

11.1.15. Передача результатов сортировки утилите awk

Может возникнуть необходимость по результатам сортировки отобразить небольшое итоговое сообщение. Сделать это легко, если воспользоваться возможностями утилиты `awk`. В следующем примере сведения о фильме с наименьшим объемом проката за год посредством канала направляются утилите `awk`, которая дополняет их поясняющим текстом. В качестве разделителя полей в обеих утилитах применяется двоеточие.

```
$ sort -t: -k4 video.txt | head -1 | \
  awk -F: '{print "Worst rental", $1, "has been rented", $3, "times"}'
Worst rental Alien has been rented 119 times
```

11.1.16. Объединение двух отсортированных файлов

Прежде чем объединять два файла, их необходимо отсортировать, иначе результат будет неотсортированным. Предположим, нам прислали файл *video2.txt*, содержащий дополнения к уже имеющемуся перечню фильмов, причем этот файл отсортирован:

```
$ cat video2.txt
Crimson Tide:134:2031
Die Hard:152:2981
```

Необходимо объединить его с файлом *video.txt*. Для этого нужно предварительно создать отсортированную версию файла *video.txt*, которую назовем *video.sort*, а затем применить команду `sort` с опцией `-m`:

```
$ sort -t: -m -k1 video2.txt video.sort
A Few Good Men:KL:445:5851
Alien:HK:119:1982
Aliens:HK:532:4892
Boys in Company C:HK:192:2192
Crimson Tide:134:2031
Die Hard:152:2981
Star Wars:HK:301:4102
The Hill:KL:63:2972
Toy Story:HK:239:3972
```

11.1.17. Дополнительные примеры команды sort

Команда `sort` может применяться для сортировки имен пользователей в файле */etc/passwd*. Достаточно выполнить сортировку содержимого этого файла по первому полю, которое включает регистрационные имена, а затем по каналу передать полученный результат утилите `awk`. Последняя отобразит содержимое только первого поля данного файла:

```
$ cat /etc/passwd | sort -t: -k1 | awk -F: '{print $1}'
adm
bin
daemon
...
...
```

Команда `sort` может работать совместно с командой `df`, выводя на экран информацию об имеющихся файловых системах в порядке убывания процента используемого ими дискового пространства. Ниже приводится образец работы команды `df`:

```
$ df
Filesystem           1k-blocks      Used Available Use% Mounted on
/dev/hda5               495714    291027    179086  62% /
/dev/hda1               614672    558896     55776  91% /dos
```

Выполним сортировку по полю 5, содержащему процент дискового пространства, занимаемого файловой системой, только предварительно с помощью редактора `sed` удалим первую строку. В процессе сортировки также воспользуемся опцией `-b`, которая позволяет игнорировать любое количество начальных пробелов.

```
$ df | sed '1d' | sort -b -r -k5
/dev/hda1              614672    558896     55776  91% /dos
/dev/hda5              495714    291027    179086  62% /
```

11.2. Удаление повторяющихся строк с помощью команды uniq

Команда `uniq` применяется для удаления идущих подряд повторяющихся строк из текстового файла. Для правильного применения команды `uniq` важно, чтобы рассматриваемый файл был отсортирован. Однако это требование не является обязательным. Можно обращаться к произвольному неупорядоченному тексту и даже сравнивать отдельные фрагменты строк.

Эту команду можно рассматривать как вариант опции `-u` команды `sort`. Следует, однако, учитывать весьма важное отличие. Опция `-u` позволяет избавиться от всех одинаковых строк в файле, тогда как команда `uniq` обнаруживает повторяющиеся строки только в том случае, когда они следуют одна за другой. Если же на вход команды `uniq` подать отсортированный файл, то действие команд `sort -u` и `uniq` будет одинаковым.

Рассмотрим пример. Имеется следующий файл:

```
$ cat myfile.txt
May Day
May Day
May Day
Going Down
May Day
```

В данном случае команда `uniq` будет рассматривать первые три строки как повторяющиеся. Пятая строка таковой не считается, потому что не совпадает с четвертой строкой.

11.2.1. Синтаксис

Общий формат команды `uniq` таков:

```
uniq опции входной_файл выходной_файл
```

Ниже перечислены некоторые из ее опций:

- `-u` Отображение только не повторяющихся строк
- `-d` Отображение одной копии каждой повторяющейся строки
- `-c` Удаление повторяющихся строк с выводом перед каждой из оставшихся строк числа повторений
- `-fn` Игнорирование первых *n* полей; полем считается последовательность непробельных символов, завершающаяся пробелом или табуляцией

В некоторых системах опция `-f` не распознается, в этом случае вместо нее следует использовать опцию *-n*, где *n* — номер поля.

Давайте применим команду `uniq` к показанному выше файлу *myfile.txt*:

```
$ uniq myfile.txt
May Day
```

```
Going Down
May Day
```

Как уже говорилось, последняя строка не считается повторяющейся. Если же выполнить над файлом команду `sort -u`, будут получены только две строки:

```
$ sort -u myfile.txt
Going Down
May Day
```

11.2.2. Определение количества повторений

Указав в команде `uniq` опцию `-c`, можно не только отбросить повторяющиеся строки, но и узнать, сколько раз повторяется каждая строка. В следующем примере команда `uniq` сообщает о том, что первая строка "May Day" встречается три раза подряд:

```
$ uniq -c myfile.txt
3 May Day
1 Going Down
1 May Day
```

11.2.3. Отображение только повторяющихся строк

Опция `-d` позволяет отобразить только те строки, которые встречаются несколько раз подряд:

```
$ uniq -d myfile.txt
May Day
```

11.2.4. Проверка уникальности отдельных полей

Команда `uniq` позволяет разбивать файл на поля, разделенные пробелами, с тем чтобы можно было исключать требуемые поля из процедуры проверки. Ниже показан небольшой файл, содержащий две колонки текста, причем содержимое второго поля в каждой строке одинаковое:

```
$ cat parts.txt
AK123 OP
BK122 OP
CK999 OP
```

Если к этому файлу применить команду `uniq`, будут отображены все строки, поскольку все они разные.

```
$ uniq parts.txt
AK123 OP
BK122 OP
CK999 OP
```

Если же выполнить проверку только по второму полю, получим иной результат. Команда `uniq` сравнит повторяющиеся поля "OP" и отобразит только одну строку:

```
$ uniq -f1 parts.txt
AK123 OP
```

11.3. Объединение файлов с помощью команды join

Команда `join` выполняет соединение строк из двух текстовых файлов на основании совпадения указанных полей. Ее действие напоминает операцию `JOIN` языка SQL. Механизм работы команды таков:

1. Каждый из двух входных файлов разбивается на поля (по умолчанию разделителем полей является пробел).
2. Из первого файла извлекается первая строка, а из нее — первое поле (можно указать другое поле).
3. Во втором файле ищется строка, имеющая такое же первое поле.
4. Найденная строка, из которой удаляется первое поле, присоединяется к исходной строке, и результат записывается в выходной поток.
5. Пункты 3 и 4 повторяются до тех пор, пока во втором файле есть строки с совпадающим первым полем.
6. Пункты 2—6 повторяются для каждой строки первого файла.

Таким образом, в выходной поток попадают только строки, имеющие общие компоненты.

Общий формат команды `join` таков:

```
join [опции] входной_файл1 входной_файл2
```

Рассмотрим некоторые наиболее важные опции этой команды:

`-a` *n*	Задает включение в выходной поток строк из файла *n* (*n* — 1 или 2), для которых не было найдено ни одного совпадения по указанному полю
`-o` *формат*	Задает формат выводимой строки. Параметр *формат* представляет собой разделенный запятыми или пробелами список спецификаций, каждая из которых, в свою очередь, имеет формат *номер_файла.поле*. По умолчанию формат выводимой строки таков: 1 — поле, по которому производится объединение; 2 — оставшаяся часть первой строки; 3 — оставшаяся часть второй строки.
`-1` *поле*	Объединять строки по указанному полю первого файла (по умолчанию таковым является первое поле)
`-2` *поле*	Объединять строки по указанному полю второго файла (по умолчанию таковым является первое поле)
`-t` *символ*	Задает разделитель полей во входном и выходном потоках

11.3.1. Объединение двух файлов

Предположим, имеется два текстовых файла: один называется *names.txt* и содержит имена пользователей с указанием улиц, на которых они проживают, а другой называется *town.txt* и содержит имена пользователей с указанием городов, в которых они живут.

```
$ cat names.txt
M.Golls 12 Hidd Rd
P.Heller The Acre
P.Willey 132 The Grove
T.Norms 84 Connaught Rd
```

```
K.Fletch 12 Woodlea

$ cat town.txt
M.Golls Norwich NRD
P.Willey Galashiels GDD
T.Norms Brandon BSL
K.Fletch Mildenhall MAF
```

Задача состоит в таком соединении двух файлов, чтобы итоговый файл содержал имена пользователей и их полные адреса. Очевидно, что общим полем является первое. Команда `join` по умолчанию выполняет объединение файлов именно по первому полю.

```
$ join names.txt town.txt
M.Golls 12 Hidd Rd Norwich NRD
P.Willey 132 The Grove Galashiels GDD
 T.Norms 84 Connaught Rd Brandon BSL
K.Fletch 12 Woodlea Mildenhall MAF
```

Как видите, пользователь P. Heller, для которого нет строки во втором файле, в результаты работы команды `join` не попал.

11.3.2. Включение несовпадающих строк

Если требуется все-таки включить информацию о пользователе P. Heller в выходные данные, воспользуйтесь опцией `-a`. Поскольку исходная строка находится в первом файле, параметром данной опции будет цифра 1:

```
$ join -a1 names.txt town.txt
M.Golls 12 Hidd Rd Norwich NRD
P.Heller The Acre
P.Willey 132 The Grove Galashiels GDD
T.Norms 84 Connaught Rd Brandon BSL
K.Fletch 12 Woodlea Mildenhall MAF
```

В общем случае, чтобы включать в результаты несовпадающие строки из обоих файлов, применяйте команду `join -a1 -a2`.

11.3.3. Задание формата вывода

Опция `-o` позволяет указать, какие поля и в какой последовательности следует включать в формируемую строку. Допустим, нужно создать файл, который включает только имена пользователей и названия городов, в которых они проживают. Требуемая информация содержится в первом поле первого файла и втором поле второго файла. Соответствующая команда имеет следующий вид:

```
$ join -o 1.1,2.2 names.txt town.txt
M.Golls Norwich
P.Willey Galashiels
T.Norms Brandon
K.Fletch Mildenhall
```

11.3.4. Выбор ключевого поля

Не всегда первое поле является общим для обоих файлов. Рассмотрим пример. Имеются два файла:

```
$ cat pers
P.Jones Office Runner  ID897
S.Round UNIX admin     ID667
L.Clip  Person1 Chief  ID982

$ cat pers2
Dept2C ID897    6 years
Dept3S ID667    2 years
Dept5Z ID982    1 year
```

Файл *pers* содержит имена, названия должностей и личные идентификационные номера служащих фирмы. Файл *pers2* содержит для каждого из служащих код отдела, в котором он работает, идентификационный номер и стаж работы на фирме. В данном случае требуется выполнить соединение строк по номеру служащего. Он хранится в четвертом поле первого файла и во втором поле второго файла. Задать их в команде `join` можно с помощью опции *-n m*, где *n* — номер файла, а *m* — номер поля.

Ниже приведена соответствующая команда и результат ее выполнения:

```
$ join -1 4 -2 2 pers pers2
ID897 P.Jones Office Runner Dept2C 6 years
ID667 S.Round UNIX admin Dept3S 2 years
ID982 L.Clip Person1 Chief Dept5Z  1 year
```

При работе с командой `join` следует быть внимательным, вычисляя номер нужного поля. Можно посчитать, что доступ реализуется к полю 4, а с точки зрения команды `join` это поле 5. В итоге будут получены неправильные результаты. Чтобы проверить, содержит ли поле с указанным номером предполагаемые данные, воспользуйтесь утилитой `awk`:

```
$ awk '{print $4}' имя_файла
```

11.4. Вырезание текста с помощью команды cut

Команда `cut` позволяет вырезать фрагменты строк из текстовых файлов или из стандартного входного потока. Извлеченный подобным образом текст направляется в стандартный выходной поток. Общий формат команды `cut` таков:

```
cut [опции] файлы...
```

Рассмотрим основные опции этой команды:

`-c` *список*	Определяет, какие символы извлекаются из каждого входного файла
`-f` *список*	Определяет, какие поля извлекаются из каждого входного файла
`-d`	Задает разделитель полей

Параметр *список* в опциях `-c` и `-f` представляет собой разделенный запятыми список диапазонов символов или полей соответственно. Диапазон может быть задан в одной из четырех форм:

n	В выходной поток включается каждый *n*-й символ (поле) каждой строки каждого входного файла

n- Диапазон формируется от *n*-го символа (поля) до конца строки

n-m Диапазон формируется от *n*-го символа (поля) до *m*-го символа (поля) включительно

-m Диапазон формируется от начала строки до *m*-го символа (поля)

11.4.1. Задание разделителя полей

В качестве входного файла мы возьмем файл *pers* из предыдущего примера, только на этот раз поля в нем будут разделены двоеточием.

```
$ cat pers
P.Jones:Office Runner:ID897
S.Round:UNIX admin:ID667
L.Clip:Personl Chief:ID982
```

Предположим, необходимо извлечь из файла список идентификационных номеров служащих, находящийся в третьем поле. Вот как можно это сделать:

```
$ cut -d: -f3 pers
ID897
ID667
ID982
```

Опция `-d:` говорит о том, что поля в файле разделяются двоеточием. Опция `-f3` задает выборку третьего поля.

Если требуется вырезать несколько полей, необходимо перечислить их в опции `-f`. Например, показанная ниже команда формирует список служащих с их идентификационными номерами:

```
$ cut -d: -f1,3 pers
P.Jones:ID897
S.Round:ID667
L.Clip:ID982
```

А вот как можно извлечь из каталога */etc/passwd* регистрационные имена пользователей и имена их начальных каталогов, хранящиеся в полях 1 и 6 соответственно:

```
$ cut -d: -f1,6 /etc/passwd
gopher:/usr/lib/gopher-data
ftp:/home/ftp
peter:/home/apps/peter
dave:/home/apps/dave
...
```

11.4.2. Вырезание отдельных символов

Опция `-c` позволяет указывать, какие конкретно символы необходимо извлекать из каждой строки входного потока. Применять эту опцию следует в том случае, если вы имеете дело со строками фиксированной длины

Рассмотрим такой пример. Когда в мою систему поступают файлы сообщений, я просматриваю их имена для определения источника, из которого они были получены. На основании этой информации производится сортировка файлов по каталогам.

Идентификатор источника содержится в последних трех символах имени файла. Вот примерный список имен файлов:

```
2231DG
2232DP
2236DK
```

Извлечение идентификаторов осуществляется с помощью такой команды:

```
$ ls 223* | cut -c4-6
1DG
2DP
6DK
```

Показанная ниже команда возвращает список пользователей, зарегистрированных в данный момент в системе:

```
$ who | cut -c1-8
root
dave
peter
```

11.5. Вставка текста с помощью команды paste

С помощью команды `cut` отдельные символы и целые поля извлекаются из текстовых файлов или стандартного входного потока. Команда `paste` выполняет противоположное действие: она вставляет в выходной поток содержимое входных файлов. Прежде чем вставлять данные из разных источников, следует убедиться, что они содержат равное число строк, иначе будут сформированы неполные строки.

Команда `paste` объединяет строки с одинаковыми номерами: сначала берутся первые строки из каждого файла и объединенная строка записывается в выходной поток, затем берутся вторые строки, третьи и т.д. По умолчанию разделителем полей является символ табуляции, если только не указана опция `-d`, которая позволяет задать иной разделитель.

Формат команды `paste` таков:

```
paste [опции] файл...
```

Рассмотрим опции команды `paste`:

`-d` *список* Сообщает команде `paste` о необходимости применять вместо табуляции другой разделитель полей. Допускается указывать список разделителей. В этом случае разделители используются циклически: между первыми двумя строками вставляется первый разделитель из списка, между следующими двумя — второй и т.д. Когда список заканчивается, осуществляется возврат к началу списка и процедура повторяется.

`-s` Задает режим последовательного слияния строк каждого входного файла по отдельности

`-` Означает выборку строки из стандартного входного потока

11.5.1. Определение порядка вставки столбцов

Для иллюстрации процедуры вставки обратимся к следующим двум файлам, полученным путем применения команды `cut` к рассмотренному выше файлу *pers*:

```
$ cat pas1
ID897
ID667
ID982

$ cat pas2
P.Jones
S.Round
L.Clip
```

По умолчанию команда `paste` вставляет столбцы один за другим:

```
$ paste pas1 pas2
ID897   P.Jones
ID667   S.Round
ID982   L.Clip
```

Порядок задания файлов в командной строке играет роль:

```
$ paste pas2 pas1
P.Jones ID897
S.Round ID667
L.Clip  ID982
```

11.5.2. Выбор разделителя полей

Если требуется создать выходной файл, в котором разделителем полей будет какой-то другой символ вместо табуляции, воспользуйтесь опцией `-d`. В приведенном ниже примере строки объединяемых файлов разделяются двоеточием:

```
$ paste -d: pas2 pas1
P.Jones:ID897
S.Round:ID667
L.Clip:ID982
```

11.5.3. Слияние строк

Наличие опции `-s` заставляет команду `paste` работать немного по-другому: для каждого входного файла она выполняет слияние всех его строк, записывая результат в выходной поток. Представленная ниже команда сначала отображает все имена служащих, а затем — их идентификационные номера.

```
$ paste -s pas2 pas1
P.Jones S.Round L.Clip
ID897   ID667   ID982
```

11.5.4. Чтение данных из стандартного входного потока

Команда `paste` имеет удобную опцию - (дефис), которая позволяет принимать данные из стандартного входного потока. Каждый дефис в командной строке соответствует одной колонке выходных данных. Например, список файлов каталога можно отобразить в четырехколоночном формате, как показано ниже:

```
$ cd /etc
$ ls | paste -d" " - - - -
init.d rc rc.local rc.sysinit
rc0.d rc1.d rc2.d rc3.d
rc4.d rc5.d rc6.d
```

Если же нужно отобразить список в одну колонку, воспользуйтесь такой командой:

```
$ ls | paste -
init.d
rc
rc.local
rc.sysinit
rc0.d
rc1.d
...
```

11.6. Разделение файла на части с помощью команды split

Команда `split` позволяет разделять крупные текстовые файлы на более мелкие. Это может оказаться удобным, например, при передаче файлов по сети. Общий формат команды `split` таков:

```
split [-размер_выходного_файла] входной_файл [префикс]
```

Первый параметр определяет количество строк, на которое нужно разбить файл. По умолчанию файл разбивается на фрагменты по 1000 строк. Если размер файла не кратен 1000, последний фрагмент будет содержать менее 1000 строк. Например, из файла, содержащего 2800 строк, в результате выполнения данной команды образуются три файла, включающих соответственно 1000, 1000 и 800 строк.

Имя каждого созданного файла представляется в формате от *префикс*`[aa]` до *префикс*`[zz]`. По умолчанию префиксом является буква 'x'. Таким образом, команда `split` создает такую последовательность файлов:

```
xaa, xab, ... xzy, xzz
```

Если расположить файлы в алфавитном порядке и выполнить их последовательную конкатенацию, получим исходный файл.

Следующий пример поможет разъяснить сказанное. Допустим, имеется файл *bigone.txt*, содержащий 2800 строк. В результате выполнения команды `split` будут сформированы три выходных файла:

Имя файла	Размер
xaa	1000
xab	1000
xac	800

Теперь рассмотрим, как изменить размер создаваемых файлов. Ниже показан файл *split1*, содержащий шесть строк:

```
$ cat split1
this is line1
this is line2
this is line3
this is line4
this is line5
this is line6
```

Для разделения его на фрагменты по две строки в каждом воспользуемся такой командой:

```
$ split -2 split1
```

Давайте проверим, что было создано (команда `ls -lt` сортирует список файлов по дате создания, а команда `head` отбирает из этого списка первые десять элементов):

```
$ ls -lt | head
total 205
-rw-r--r--   1 dave   admin          28 Apr 30 13:12 xaa
-rw-r--r--   1 dave   admin          28 Apr 30 13:12 xab
-rw-r--r--   1 dave   admin          28 Apr 30 13:12 xac
...
```

Исходный файл состоит из шести строк. В результате применения к нему команды `split` были сформированы три файла, содержащих по две строки каждый. Чтобы убедиться в правильности работы команды, рассмотрим содержимое файла *xac*, который должен включать последние две строки:

```
$ cat xac
this is line5
this is line6
```

11.7. Заключение

В настоящей главе были рассмотрены различные стандартные утилиты (`sort`, `unique`, `join`, `cut`, `paste` и `split`, а также `head` и `tail`), имеющие отношение, главным образом, к сортировке, разделению и объединению текстовых файлов. Применение каждой из них иллюстрировалось многочисленными примерами, которые позволят вам сформировать четкое представление о возможностях этих утилит. Я надеюсь, что благодаря изложенным сведениям вы смогли пополнить свой багаж знаний об инструментах работы с текстом, имеющихся в UNIX и Linux.

ГЛАВА 12

Утилита tr

12.1. Применение утилиты tr

Утилита `tr` выполняет символьное преобразование путем подстановки или удаления символов из стандартного входного потока. Она часто применяется для удаления управляющих символов из файла или преобразования регистра символов. Как правило, утилите `tr` передаются две строки: первая строка содержит искомые символы, а вторая — те, на которые их следует заменить. При запуске команды устанавливается соответствие между символами обеих строк, а затем начинается преобразование.

В этой главе рассматриваются следующие темы:

- преобразование строчных символов в прописные;
- очистка содержимого файлов от управляющих символов;
- удаление пустых строк.

Формат утилиты `tr` с наиболее часто применяемыми параметрами таков:

```
tr -c -d -s ["строка1"] ["строка2"] входной_файл
```

где

`-c` Задает замену набора символов, указанных в `строке1`, их собственным дополнением при условии, что значение этих символов находится в диапазоне значений кодов ASCII

`-d` Задает удаление во входном файле всех символов, указанных в `строке1`

`-s` Задает удаление в последовательности повторяющихся символов всех символов, кроме первого, благодаря чему удаляются повторяющиеся символы

Параметр `входной_файл` определяет имя файла, содержимое которого необходимо преобразовать. Несмотря на то, что входные данные могут иметь и другие формы, широко используется именно указанный выше способ их задания.

12.1.1. Диапазоны символов

При использовании утилиты `tr` можно указать диапазоны или списки символов в виде шаблонов, которые образованы строками. Эти шаблоны подобны регулярным выражениям, однако на самом деле они таковыми не являются. При указании в утилите `tr` содержимого строк `строка1` или `строка2` используются только диапазоны и последовательности символов либо отдельные символы.

`[a-z]` Строка символов, находящихся в диапазоне a-z

`[A-Z]`	Строка символов, находящихся в диапазоне A-Z
`[0-9]`	Строка чисел
`/octal`	Восьмеричное число, состоящее из трех чисел и представляющее любой действительный символ в коде ASCII
`[O*n]`	Означает символ 'O', встречающийся столько раз, сколько указывает значение 'n'. Таким образом, `[O*2]` означает OO, причем в любой строке, включая и 00

В большинстве вариантов утилиты `tr` поддерживаются классы символов и сокращенная запись управляющих символов. В формат класса символов [:class] среди прочего входят следующие обозначения: alnum (буквенно-цифровые символы), alpha (буквы), blank (пропуски), upper (прописные буквы), lower (строчные буквы), cntrl (управляющие символы), space (пробелы), digit (цифры), graph (графические символы) и т.д. В табл. 12.1 представлен способ сокращенного представления некоторых наиболее распространенных управляющих символов, используемый вместо восьмеричного их представления в виде трех чисел, которое также приведено в данной таблице.

Таблица 12.1. Различные способы указания управляющих символов в утилите `tr`

Сокращение	Значение	Восмеричное значение
`\a`	Control-G — звонок	\007
`\b`	Control-H — клавиша возврата на одну позицию	\010
`\f`	Control-L — прокрутка страницы	\014
`\n`	Control-J — новая строка	\012
`\r`	Control-M — клавиша возврата каретки	\015
`\t`	Control-I — клавиша табуляции	\011
`\v`	Control-X	\030

При замене строки или диапазона символов одним символом следует иметь в виду, что этот символ не указывается в квадратных скобках ([]). В некоторых системах допускается применение квадратных скобок, причем в данном случае можно для указания символа новой строки воспользоваться шаблоном `["\012"]` или `"\012"`. Утилита `tr` не предъявляет строгих требований к виду кавычек. Поэтому не следует удивляться, если эта утилита действует даже в том случае, если вместо одинарных кавычек используются двойные кавычки.

Подобно большинству системных инструментальных средств, утилита `tr` восприимчива к специальным символам. Поэтому если требуется выполнить сопоставление с одним из таких символов, следует предварительно отделить этот символ обратной косой чертой. Например, для указания левой фигурной скобки ({) необходимо ввести `\{` для отмены специального значения фигурной скобки.

12.1.2. Сохранение выходного результата

Если нужно сохранить полученные результаты, следует переадресовать их в файл. В приведенном ниже примере выходной результат перенаправляется в файл с именем *results.txt*. В качестве входного используется файл *oops.txt*.

```
$ tr -s "[a-z]" < oops.txt > results.txt
```

Обратимся к некоторым примерам использования рассматриваемой команды.

12.1.3. Устранение повторяющихся символов

Если проанализировать приведенный ниже файл, можно обнаружить некоторые опечатки. Разумеется, допустить опечатки легко — вспомните, сколько раз во время работы в редакторе `vi` вы случайно нажимали не те клавиши.

```
$ pq oops.txt
And the cowwwwws went homeeeeeeee
Or did theyyyy
```

Если нужно избавиться от повторяющихся букв или сократить число подобных букв до одной, можно воспользоваться параметром `'-s'`. Также можно воспользоваться шаблоном [a-z], поскольку в данном случае все символы являются буквенными. При этом входной файл перенаправляется команде `tr`.

```
$ tr -s "[a-z]"< oops.txt
And the cows went home
Or did they
```

Все повторяющиеся символы устраняются. При необходимости файл *oops.txt* можно перенаправить с помощью команды `cat`. Результат будет тот же.

```
$ cat oops.txt | tr -s "[a-z]"
And the cows went home
Or did they
```

12.1.4. Удаление пустых строк

Для удаления пустых строк их следует просто "вытеснить" из файла. Ниже приведен файл с именем *plane.txt*, содержащий ряд пустых строк.

```
$ pq plane.txt
987932 Spitfire


190992 Lancaster


238991 Typhoon
```

В данном случае применяется параметр `'-s'`, который приводит к удалению пустых строк. Также используется восьмеричное значение для символа новой строки `\012`. Ниже приведена соответствующая команда.

```
$ tr -s "[\012]" < plane.txt
987932 Spitfire
190992 Lancaster
238991 Typhoon
```

С другой стороны, можно воспользоваться сокращенной записью символа новой строки '\n'. Можно применять как одинарные, так и двойные кавычки, хотя обычно используются двойные кавычки.

```
$ tr -s "[\n]" < plane.txt
987932 Spitfire
190992 Lancaster
238991 Typhoon
```

12.1.5. Преобразование прописных букв в строчные

Изменение регистра символов является наряду с процедурой удаления управляющих символов одним из наиболее распространенных случаев применения утилиты tr. Чтобы выполнить подобное преобразование, достаточно указать шаблон строчных букв '[a-z]' для входных данных и шаблон прописных букв '[A-Z]' для выходных преобразованных данных.

В первом примере осуществляется передача утилите tr строки, содержащей смешанный набор символов.

```
$ echo "May Day, May Day, Going Down.." | tr "[a-z]" "[A-Z]"
MAY DAY, MAY DAY, GOING DOWN..
```

С другой стороны, можно воспользоваться классами символов [:lower:] и [:upper:].

```
$ echo "May Day, May Day, Going Down.." | tr ":lower:" ":upper:"
MAY DAY, MAY DAY, GOING DOWN..
```

Для преобразования прописных букв из текстового файла в строчные и последующего их размещения в новом файле применяется следующий формат:

```
cat file-to-translate | tr "[A-Z]" "[a-z]" > new-file-name
```

где параметр 'файл-для-преобразования' — преобразуемый файл, а 'имя-нового-файла' — имя, которое нужно присвоить новому файлу. Например:

```
cat myfile | tr "[A-Z]" "[a-z]" > lower_myfile
```

12.1.6. Преобразование строчных букв в прописные

Преобразование строчных букв в прописные выполняется в обратном порядке по сравнению с преобразованием, рассмотренным в предыдущем разделе. Ниже приведены два примера подобного преобразования.

```
$ echo "Look for the route, or make the route" | tr "[a-z] "[A-Z]"
LOOK FOR THE ROUTE, OR MAKE THE ROUTE

$ echo "May Day, May Day, Going Down.." | tr ":lower:" ":upper:"
may day, may day, going down..
```

Для преобразования строчных букв из текстового файла в прописные и последующего их размещения в новом файле применяется формат:

```
cat file-to-translate | tr "[a-z]" "[A-Z]" > new-file-name
```

где 'файл-для-преобразования' — преобразуемый файл, а 'имя-нового-файла' — имя, которое нужно присвоить новому файлу. Например:

```
cat myfile | tr "[a-z]" "[A-Z]" > upper_myfile
```

12.1.7. Удаление определенных символов

Иногда возникает необходимость в удалении некоторых столбцов из загруженного файла, содержащего только буквы и числа. Для этого в рассматриваемой команде необходимо применить оба параметра, '-c' и '-s'.

Приведенный ниже файл содержит часть составленного на неделю личного календаря. Задача заключается в устранении чисел. В личном календаре остаются в наличии только дни недели. Поскольку дни недели используются в формате как прописных, так и строчных букв, в этом случае применяют оба диапазона символов: [a-z] и [A-Z]. При этом команда 'tr -cs "[a-z] [A-Z]" "[\012*]"' выбирает все содержимое файла, которое не находится в пределах [a-z] или [A-Z] (буквенных символов), содержащихся в строка1, и преобразует их в символы новой строки. В приведенной выше команде tr параметр '-s' сообщает о необходимости "сокращения" всех символов новой строки, а параметр '-c' сохраняет без изменения все буквенные символы. Ниже приведен файл, содержащий данные личного календаря, после чего следует строка с утилитой tr и результат ее выполнения.

```
$ pq diary.txt
Monday 10:50
Tuesday 15:30
Wednesday 15:30
Thursday 10:30
Friday 09.20

$ tr -cs "[a-z] [A-Z]" "[\012*]" < diary.txt
Monday
Tuesday
Wednesday
Thursday
Friday
```

12.1.8. Преобразование управляющих символов

Чаще всего утилита tr применяется для преобразования управляющих символов, особенно во время загрузки файлов из DOS в UNIX. Если в команде ftp не задан параметр, выполняющий преобразование символов возврата каретки в символы новой строки, обычно применяют утилиту tr.

Ниже приведен текстовый файл, при пересылке которого не было выполнено преобразование символов возврата каретки. Файл содержит часть требования на выдачу канцелярских принадлежностей. Управляющие символы файла отображены ниже с помощью команды cat -v.

```
$ cat -v stat.txt
Boxes paper^^^^^^12^M
Clips metal^^^^^^^50^M
Pencils-meduim^^^^^^10^M
^Z
```

В этом файле последовательность символов '^^^^^^' кодирует символы табуляции, каждая строка завершается управляющей последовательностью CONTROL-M, а в конце файла находится управляющая последовательность CONTROL-Z. Ниже показано, как можно исправить положение.

В данном случае придется воспользоваться параметром '-s'. Если обратиться к таблице кодов ASCII, то восьмеричный код символа '^' равен 136. Соответствующее значение для управляющей последовательности ^M равно '015', для символа табуляции — '011', а для управляющей последовательности ^Z — '032'. Данная задача выполняется поэтапно.

Для замены в рассматриваемой команде последовательности символов '^^^^^^' символами табуляции используется следующий шаблон: "\136" "[\011*]". Затем полученные результаты перенаправляются во временный рабочий файл с именем *stat.tmp*.

```
$ tr -s '[\136]" "[\011*]" < stat.tr > stat.tmp
Boxes paper 12^M
Clips metal 50^M
Pencils-medium 10^M
^Z
```

Для замены управляющих последовательностей ^M, расположенных в конце каждой строки, символом новой строки и устранения управляющей последовательности ^Z применяется шаблон \n. Не следует забывать, что входные данные поступают из временного файла *stat.tmp*.

```
$ tr -s "[\015\032]@ @\n@ < stat.tmp
Boxes paper    12
Clips metal    50
Pencils-medium 10
```

Таким образом, управляющие символы удаляются и файл готов к применению.

12.1.9. Быстрые преобразования

Если из файла необходимо удалить только управляющие последовательности ^M и заменить их символами новой строки, для этого применяется команда:

```
$ tr -s "[\015]" "\n" < файл_ввода
```

С другой стороны, для получения аналогичного результата можно воспользоваться командой:

```
$ tr -s "[\r]" "[\n]" < файл_ввода
```

То же самое преобразование можно выполнить и с помощью команды:

```
$ tr -s "\r" "\n" < файл_ввода
```

Еще один распространенный вариант преобразования файлов, перенесенных из DOS в UNIX, иллюстрирует команда:

```
$ tr -s "[\015\032]" "[\012*]" < файл_ввода
```

Эта команда удаляет управляющие последовательности ^M и ^Z и заменяет их символами новой строки.

Следующая команда удаляет символы табуляции, заменяя их пробелами:

```
$ tr -s "[\011]" "[\040*]" < файл_ввода
```

Для замены в файле пароля *passwd* всех двоеточий символами табуляции, двоеточие следует заключить в кавычки и указать в строке замены восьмеричное значение символа табуляции, которое равно '011'. Файл станет более удобным для чтения. Сначала приводится файл *passwd*, а затем команда с утилитой `tr`, которая выполняет задачу.

```
$ pq passwd
halt:*:7:0:halt:/sbin:/sbin/halt
mail:*:8:12:mail:/var/spool/mail:
news:*:9:13:news:/var/spool/news:
uucp:*:10:14:uucp:/var/spool/uucp:

$ tr -s "[:]" "[\011]" < passwd
halt    *       7       0       halt    /sbin   /sbin/halt
mail    *       8       12      mail    /var/spool/mail
news    *       9       13      news    /var/spool/news
uucp    *       10      14      uucp    /var/spool/uucp
```

С другой стороны, аналогичного результата можно добиться с помощью следующей команды, где указывается сокращенная запись символа табуляции:

```
& tr "[:]" "[\t]" < passwd
```

12.1.10. Сравнение с несколькими символами

Для выполнения сравнения с несколькими символами применяется формат `[character*n]`. Ниже приводится содержимое файла, описывающего жесткие диски системы. В файле содержатся диски, которые зарегистрированы или распознаны системой. Первый столбец содержит числа. Если этот столбец не состоит из одних нулей, то регистрируется соответствующий диск во втором столбце.

Иногда надоедает наблюдать в подобных списках нули, поэтому заменим их символом, который привлекает к себе внимание. Тогда сразу становится видно, какие диски присутствуют в системе, а какие — отсутствуют. Ниже приведена часть содержимого файла.

```
$ pq hdisk.txt
1293 hdisk3
4512 hdisk12
0000 hdisk5
4993 hdisk12
2994 hdisk7
```

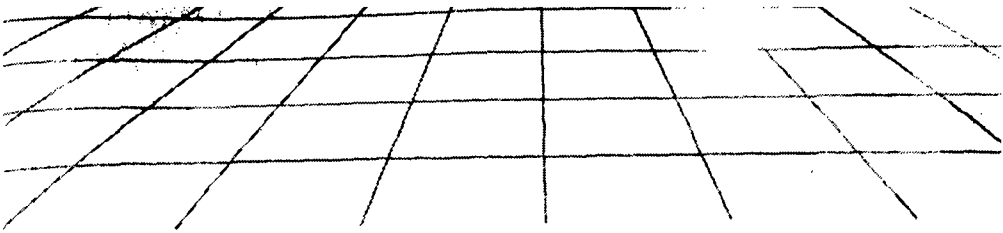
При просмотре файла становится ясно, что имеется один жесткий диск, который не зарегистрирован. Для замены всех нулей, допустим, звездочками можно воспользоваться шаблоном `[0*4]`, который означает поиск соответствия, по крайней мере, четырем нулям. При этом строка замены содержит только звездочки. Ниже приводится соответствующая команда и результат выполнения подобной фильтрации:

```
$ tr "[0*4]" "*" <hdisk.txt
1293 hdisk3
4512 hdisk12
**** hdisk5
4993 hdisk12
2994 hdisk7
```

Теперь, просматривая приведенный выше файл, можно сразу узнать, какой диск не зарегистрирован в системе.

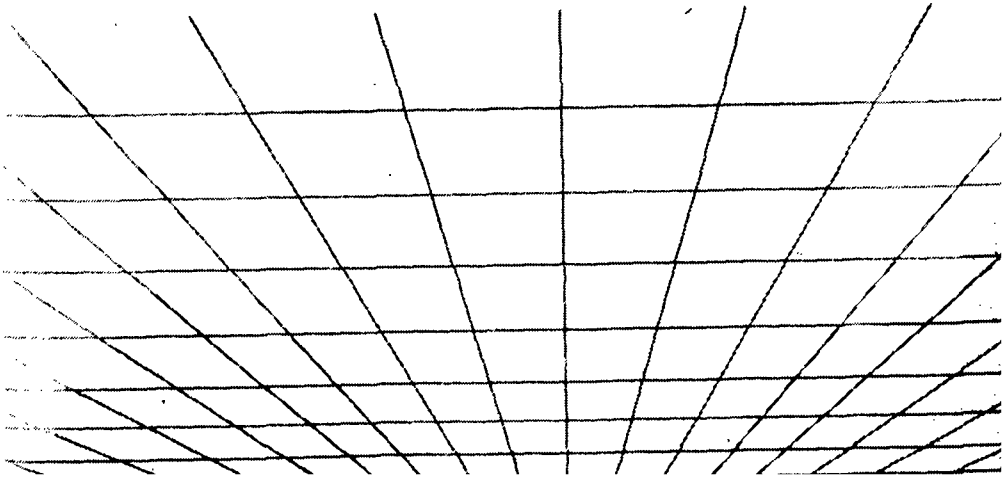
12.2. Заключение

Утилита `tr` применяется, главным образом, для преобразования символов и удаления управляющих символов. Все функции, использованные в этой главе, могут быть реализованы и с помощью команды `sed`. Однако некоторые пользователи предпочитают применять утилиту `tr`, поскольку она выполняет те же задачи и более удобна в применении.



ЧАСТЬ 3

Регистрация в системе



ГЛАВА 13

Регистрация в системе

Во время регистрации в системе, еще до появления командной строки происходит ряд событий, связанных с процессом регистрации. После ввода регистрационного имени и пароля система проверяет, допустима ли регистрация данного пользователя. С этой целью исследуется содержимое файла */etc/passwd*. Если регистрационное имя пользователя окажется правильным, а введенный пароль действительным, начинается процесс, который и реализует регистрацию в системе пользователя.

В этой главе рассматриваются следующие темы:

- процесс регистрации
- файл */etc/profile*
- файл *$HOME/.profile*
- настройка пользовательского файла *$HOME/.profile*

Прежде чем познакомиться с процессом регистрации, давайте рассмотрим текстовый файл */etc/passwd*. В этом файле можно вносить изменения в текстовые поля, включая поля паролей. При обработке файла следует проявлять особое внимание. Файл *passwd* включает семь полей, разделенных двоеточиями. Ниже приведен фрагмент файла *passwd*. В верхней части листинга добавлены номера столбцов, что облегчает идентификацию полей.

```
[1][      2       ][3][4][    5    ][         6         ][ 7 ]
kvp:JFqMmk9.uRioA:405:413:K.V.Pally:/home/sysdev/kvp:bin/sh
dhw:hi/G4UlCUd9aI,B/OJ:407:401:D.Whitley:/home/dept47/dhw:/bin/sh
aec:ILgHtxJ9kXtSc,B/GI:408:401:A.E.Cloody:/b_user/dept47/aec:/bin/sh
gdw:iLFu9BB8RNjpc,B/MK:409:401:G.D.Wilcom:/b_user/dept47/gdw:/bin/sh
```

Рассмотрим некоторые поля файла. Первое поле содержит регистрационное имя, второе поле — зашифрованный пароль, пятое поле — полное имя пользователя, шестое поле — начальный каталог данного пользователя, а седьмое поле описывает интерпретатор shell, применяемый пользователем. В данном случае параметр "/bin/sh" определяет интерпретатор shell, устанавливаемый по умолчанию. Обычно используется интерпретатор Bourne shell.

Существуют и другие типы файлов *passwd*. В одном из вариантов поле `passwd` фактически находится в другом файле. Однако приведенный выше формат является наиболее распространенным.

После успешной регистрации выполняются два файла среды. Первым является файл */etc/profile*, а вторым — файл *.profile*, расположенный в начальном каталоге *$HOME*.

Существуют и другие исполняемые файлы инициализации, однако в данном случае нас интересуют лишь файлы профилей.

13.1. Файл /etc/profile

Информация файла профиля, находящегося в каталоге */etc*, автоматически считывается при регистрации каждого пользователя. Этот файл обычно содержит:

- глобальные или локальные переменные среды;
- информацию о пути к файлам в переменной PATH;
- параметры терминала;

а также

- меры безопасности;
- совет дня или сведения о причинах отказа.

Далее приводится краткое объяснение перечисленных выше пунктов.

Некоторые *глобальные переменные среды* устанавливаются так, что к ним обеспечивается доступ со стороны пользователей, а также shell-процессов и приложений.

В переменной PATH хранится местоположение каталогов, включающих исполняемые файлы, библиотеки и простые файлы, благодаря чему их можно быстро найти.

Параметры терминала содержат наиболее общие характеристики используемого терминала.

Команды, обеспечивающие меры безопасности, описывают режимы создания файлов или удвоенные командные строки регистрации, используемые для доступа к секретным областям системы.

Совет дня реализован в виде текстового файла, содержащего описания любых предстоящих событий. Пользователи получают сообщения об этих событиях при регистрации в системе. Кроме того, этот файл может включать любые сообщения относительно причин отказа в той или иной ситуации.

Ниже приводится содержимое файла */etc/profile*, который рассматривается далее.

```
$ pq /etc/profile
#!/bin/sh
#
trap "" 2 3
# Установка LOGNAME
export LOGNAME

# установка дополнительных путей MAN
MANPATH=/usr/opt/sybase/man
export MANPATH

# Установка TZ
if [ -f /etc/TIMEZONE ]
then
. /etc/TIMEZONE
fi

# Установка TERM
if [ -z "$TERM" ]
then
TERM=vt220 # для стандартного асинхронного терминала/консоли
```

```
export TERM
fi

# Разрешение пользователю на прерывание отображения исключительно совета дня
trap "trap ''2" 2
if [ -f /usr/bin/cat ] ; then
cat -s /etc/motd
fi
trap "" 2
if [ -f /usr/bin/mail ] ; then
if maill -e ; then
echo "Hey guess what? you have mail"
fi
fi
;;
esac

# установка значения umask для более безопасного осуществления операций
umask 022
fi
# установка сред
SYSHOME=/appdvb/menus
ASLBIN=/asl_b/bin
UDTHOME=/dbms_b/ud
UDTBIN=/dbms_b/ud/bin
PAGER=pg
NOCHKLPREQ=1
PATH=$PATH:$UDBIN:$ASLBIN
export PATH UDTHOME UDTBIN PAGER NOCHKLPREQ SYSHOME

trap 2 3

# Установка переменной SAVEDSTTY так, чтобы переменную можно было
# использовать для восстановления
# настроек stty при выходе из системы аудита.
SAVEDSTTY='stty -g'
exporrt SAVEDSTTY

# регистрация всех связей в syslog
logger -p local7. info -t login $LOGNAME 'tty'
trap 'logger -p local7. info -t logout 3LOGNAME 'tty'' 0

# отмена создания файлов дампа ядра
ulimit -c 0
#
# проверка, зарегистрированы ли пользователи более двух раз, не считая...
case $LOGNAME in
idink | psalon | dave)
;;
*)
PID-${$};export PID
Connected='who | awk '{print $1}' | fqrep -xc ${LOGNAME}
if [ "$Connected" -gt 2 ]
then
```

```
  echo
  echo 'You are logged in more than twice.'
  echo
  who -u | grep $LOGNAME
  echo
  echo 'Enter <CR> to exit \c'
  read FRED
  kill -15 $PID
fi
 ;;
esac
# установка приглашения, включающего идентификатор пользователя
PS1="$LOGNAME>"
```

Если вам непонятны некоторые из приведенных выше команд, не стоит огорчаться. Все команды описаны далее в этой книге. А к данному листингу можно вернуться в любой момент.

Рассмотрим последовательно, какие действия выполняются в приведенном листинге.

Первая строка является командой, задающей перехват двух сигналов, которые не дают пользователям возможности прервать выполнение файла с помощью нажатия клавиш [Quit] или [Ctrl+C].

Затем экспортируется значение переменной `LOGNAME`. В данной системе введены дополнительные справочные страницы (страницы `man`). Переменная `MANPATH` вносит их в список поиска по страницам уже имеющегося справочного руководства.

В приведенном листинге проверяется файл часового пояса и соответствующий исходный файл, если таковой существует. Затем устанавливается режим эмуляции терминала, соответствующий значению `vt220`.

Далее в листинге отменяется режим перехвата символов, чтобы пользователи могли отменить отображение совета дня. Однако в следующих строках листинга этот режим восстанавливается.

Затем определяется почтовое сообщение (это сообщение пользователь получает, если для него имеется новая почта).

В следующем фрагменте листинга устанавливается значение `unmask`, разрешающее создание файлов, для которых установлены определенные права доступа, заданные по умолчанию.

Далее инициализируются переменные среды. После установки и экспорта они становятся доступными каждому пользователю.

В следующей строке листинга восстанавливается перехват сигналов, соответствующих клавишам [Quit] или [Ctrl+C].

Затем происходит сохранение заданных по умолчанию параметров стандартного терминала `stty`. В результате этого при выходе пользователей из системы аудита можно повторно инициализировать параметры терминала.

Потом регистрируются все подключения пользователей в файле */var/adm/messages*, который по умолчанию является журнальным файлом системы.

В следующей строке листинга выполняется команда `ulimit`. Она позволяет ограничить количество дампов ядра либо дампов, выполняемых в шестнадцатеричной форме.

После этого следует небольшой фрагмент кода, который дает пользователям возможность регистрироваться дважды, но только одновременно для двух пользователей. Это правило не распространяется на следующих трех пользователей: `idink`,

psalom, dave. Остальным пользователям будет отказано в доступе, если они попытаются зарегистрироваться более двух раз.

И в завершение последняя строка файла задает вид командной строки, содержащей регистрационное имя соответствующего пользователя.

В результате описанных выше действий устанавливается среда для глобального применения. А теперь выполним ее настройку, руководствуясь профильным файлом текущего пользователя.

13.2. Пользовательский файл $HOME/.profile

После выполнения файла */etc/profile* пользователь попадает в собственный начальный каталог *$HOME*. Если вернуться к файлу *passwd*, то видно, что имя этого каталога находится во втором поле от конца файла.

Этот каталог действительно является исходным каталогом пользователя, поскольку именно в нем хранится вся личная информация пользователя.

Если имеется файл *.profile*, система использует его в качестве исходного файла. Значит, данный процесс обращается к существующему интерпретатору shell. В результате этого все существующие среды (*/etc/profile*) не будут изменяться под влиянием информации файла (*.profile*). При создании другого процесса локальные переменные интерпретатора shell фактически переписываются.

Вернемся к файлу *.profile*. Как правило, если учетная запись уже установлена, профильный файл в каком-то виде уже существует. Важно помнить, что установки файла */etc/profile* могут быть переопределены при добавлении в файл *.profile* нового элемента с другим значением либо при выполнении команды `unset`. Специальная настройка данного файла остается в ведении пользователя, поэтому рассмотрим стандартный файл *.profile*:

```
$ pq .profile
#.profile
set -a
MAIL=/usr/mail/${LOGNAME:?}
PATH=$PATH
export PATH
#
```

А теперь приступим к настройке данного файла.

Необходимо установить ряд переменных среды, подобных `EDITOR`, чтобы утилита `cron` и другие приложения были осведомлены о применяемом редакторе. Кроме того, нас не устраивает режим эмуляции терминала vt220. Поэтому, изменим это значение на режим эмуляции терминала vt100. Это обстоятельство связано с переменной `TERM`.

Также можно создать каталог *bin* и указать его в настройках пути к своим файлам. Наличие подобного каталога желательно, поскольку именно в нем можно хранить все свои сценарии. Если же указать каталог в переменной `PATH`, можно вводить не полное имя пути к исполняемому сценарию, а лишь имя сценария.

Нужно ли, чтобы регистрационное имя отображалось в командной строке? Скорее всего, нет. Желательно, чтобы в командной строке отображался текущий каталог, а возможно, даже имя сервера используемой системы. Ниже показано, каким образом устанавливается имя базового компьютера.

```
$ PS1="`имя_сервера`>"
dns-сервер>
```

Или для текущего каталога, в котором вы находитесь:

```
$ PS1="\`pwd\`>"
\home\dave>
```

Если в приведенной выше командной строке возвращается слово "pwd", то вместо него воспользуйтесь командной строкой:

```
PS1='$PWD >';
```

В качестве дополнительной командной строки (которая обычно применяется при выполнении многострочной команды в командной строке) можно применять знак авторского права ©. Значение этого символа в коде ASCII равно восьмеричному числу 251, или десятичному числу 169.

```
$ PS2=" `echo "\251" `:"
/home/dave> while read line
©:do
©:echo $line
©:done
```

В Linux...

Для использования в echo восьмеричного значения применяется следующий синтаксис:

```
$ PS2="`echo -e "\251"`:"
```

Так как пользователи имеют доступ к параметрам администрирования системы, размещенным в каталоге `/usr/admin`, этот каталог следует задать в виде переменной среды. Тогда в данный каталог можно переходить с помощью команды `cd`.

```
ADMIN=/usr/adm
```

Кроме того, сразу же после регистрации желательно знать, сколько пользователей находится в системе. Для этого можно воспользоваться командой `who` или `wc`.

```
$ echo " `who|wc -l` users are on today"
19 users are on today
```

Итак, добавим в файл *.profile* все изменения. Для того чтобы возымели действие любые изменения в файле *.profile* или в файле */etc/profile*, нужно выйти из системы, а затем повторно зарегистрироваться либо обращаться к данному файлу в качестве исходного. При этом используется следующий формат:

```
./имя_пути/имя_файла
```

Чтобы использовать файл *.profile* в качестве исходного, достаточно ввести следующую команду:

```
$. .profile
```

Если приведенный вариант не действует, можно попробовать следующий:

```
$. ./profile
```

Ниже показан измененный файл *.profile*.

```
$ pq .profile
#.profile
MAIL=/usr/mail/${LOGNAME:?}
PATH=$PATH:$HOME:bin
#
EDITOR=vi
TERM vt100
ADMIN=/usr/adm
PS1=" `hostname`>"
PS2=" `echo "\0251" `:"
export EDITOR TERM ADMIN PATH PS1
echo " `who|wc -l` users are on to-day"
```

13.3. Применение команды stty

Команда `stty` позволяет установить характеристики используемого терминала. Чтобы уточнить текущие установки опций `stty`, примените команду `stty -a`.

```
$ stty -a
speed 9600 baud; rows 24; columns 80; line = 0
intr =^C; quit =^\; erase =^?; kill =^U; eof =^D; eol =<undef>;
eol2 =<undef>; start =^Q; stop =^S; susp =^Z; rprnt =^R; werase = ^W;
lnext =^V; flush =^O; min =1; time =0;
-parenb -parodd cs8 -hupcl -cstopb cread -clocal -crtscts
-ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon
-ixoff -iuclc -ixany -imaxbel
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0
vt0 ff0 isig icanon iexten echo echoe echok -echonl
-noflsh -xcase
-tostop -echoprt echoctl echoke
```

Довольно часто при настройке терминала не работает клавиша [Backspace], т.е. при ее нажатии не происходит удаление символа. Данная команда `stty` сообщает, что в качестве клавиши [Backspace] используется комбинация клавиш `^?`. Однако эта комбинация не действует. С помощью комбинации клавиш [Ctrl+H] можно вернуться на одну позицию и удалить предыдущий символ. Поэтому для устранения данной проблемы воспользуемся командой `stty`. Общий формат установки параметров команды `stty` в командной строке:

```
stty символ
```

Ниже показана установка управляющей последовательности `^H`, соответствующей коду клавиши [Backspace]:

```
$ stty erase '\^H'
```

При использовании приведенной выше команды `stty` в файле `.profile` могут возникнуть затруднения. Команда `stty` на самом деле может предполагать наличие управляющей последовательности <CONTROL-H>. Для устранения этого недостатка необходимо во время работы в редакторе `vi` поступить следующим образом: нажмите клавишу [Ctrl] и одновременно с ней клавишу [V], затем отпустите клавишу [V] и нажмите клавишу [H].

Ниже приводятся наиболее распространенные наименования команды `stty`.

Наименование	Клавиша	Значение
intr	^C	Завершение процесса
echo		Подключение режима отображения
-echo		Отключение режима отображения
eof	^D	Конец файла, выход из системы
kill	^Y	Удаление строки
start	^Q	Начало прокрутки текста на экране
Stop	^S	Завершение прокрутки текста на экране

Весьма полезен следующий параметр команды stty:

```
stty -g
```

Этот параметр позволяет сохранить результаты выполнения команды stty в удобочитаемом формате. Затем эти результаты можно назначить команде stty, как показано выше на примере листинга файла */etc/profile*. Для этого достаточно разместить содержимое команды stty -g в соответствующей переменной, ввести необходимые изменения в команду stty, а по завершении передать эти изменения команде stty.

Этим приемом удобно воспользоваться, если при изменении параметров команды stty вы нечаянно нарушили нормальную работу терминала. В этом случае можно легко восстановить исходные параметры терминала. Ниже приведен пример сохранения и последующего восстановления текущих установок команды stty. В данном случае команда stty -echo применяется для отключения режима отображения, а в конце сценария восстанавливаются исходные параметры.

```
$ pq password
#!/bin/sh
# пароль
# отображение процесса восстановления среды stty
SAVEDSTTY= `stty -g`
stty -echo
echo "\nGive me that password :\c"
read PASSWD
echo "\nyour password is $PASSWD
stty $SAVEDSTTY

$ sttypass
Give me that password :
your password is bong
```

В Linux ...

Чтобы сообщить Linux, что в строках предполагается применение управляющих символов, команду echo следует вводить как "echo -e".

```
SAVEDSTTY=`stty -g`
stty -echo
echo "\nGive me that password :\c"
read PASSWD
echo "\nyour password is $PASSWD"
stty $SAVEDSTTY
```

Команда `stty` позволяет настраивать терминал, принтеры или модемы, т.е. она весьма универсальна. Однако, применяя команду `stty`, будьте внимательны. Не следует изменять настройку ранее установленного параметра, поскольку это может привести к нарушению работоспособности системы.

13.4. Создание файла .logout

Одним из недостатков интерпретатора Bourne shell является отсутствие в нем файла *.logout*. Этот файл содержит команды, которые требуется выполнить непосредственно перед завершением сеанса работы в системе, т.е. до выполнения команды `exit`.

Однако собственный файл *.logout* интерпретатора Bourne shell можно создать с помощью команды `trap` (более подробные сведения о команде `trap` и сигналах приведены далее в книге). Рассмотрим, как это делается. Отредактируйте свой файл *.profile* и разместите на последней строке приведенную ниже команду. Затем сохраните этот файл и выйдите из режима редактирования.

```
trap "$HOME/.logout" 0
```

Создайте файл *.logout* и введите в него любые команды, которые требуется выполнить. В этот файл можно включать сведения по своему усмотрению.

```
$ pq .logout
rm -f $HOME/*.log
rm -f $HOME/*.tmp
echo "Bye...bye $LOGNAME"
```

Файл *.logout* вызывается при выходе пользователя из системы. Когда пользователь выходит из среды интерпретатора shell, система посылает сигнал 0, который и означает выход из текущего интерпретатора shell. Прежде чем управление передается обратно интерпретатору shell, для продолжения процесса выхода из системы, упомянутый выше сигнал перехватывается командой `trap`. Эта команда находится в соответствующей строке файла *.profile*. Затем выполняется файл *.logout*.

13.5. Заключение

Каждый пользователь может настроить свой файл *$HOME/.profile* в соответствии с требуемыми задачами. В главе было показано, каким образом можно переопределять общесистемные параметры согласно своим предпочтениям. Существует большое количество способов настройки пользовательской рабочей среды. Можно указывать приветственные сообщения и вносить существенные изменения в параметры терминала.

ГЛАВА 14

Переменные среды и интерпретатора shell

Чтобы продуктивно работать с интерпретатором shell, нужно уметь управлять переменными этого интерпретатора. Переменными интерпретатора shell являются наименования, которым присваиваются значения. В качестве значений может выступать имя пути, имя файла или число. В любом случае интерпретатор shell воспринимает присвоенное значение как текстовую строку.

Существуют переменные двух типов — переменные интерпретатора (shell) и переменные среды. На самом деле имеется четыре типа переменных, но оставшиеся предназначены только для чтения. Эти переменные считаются специальными и применяются при передаче параметров в shell-сценарии.

В этой главе рассматриваются следующие темы:

- переменные интерпретатора shell;
- переменные среды;
- подстановка переменных;
- экспорт переменных;
- специальные переменные;
- передача информации в сценарии;
- применение позиционных параметров в системных командах.

14.1. Понятие о переменных интерпретатора shell

Переменные позволяют выполнить настройку среды. Они содержат информацию, которая применяется определенным пользователем. Благодаря этому система получает более подробные сведения о пользователях. Кроме того, переменные используются для хранения констант. Рассмотрим, например, переменную под именем `EDITOR`. Существует довольно много текстовых редакторов, но как выбрать наиболее подходящий из них? Присвойте имя нужного редактора переменной `EDITOR`, и тогда именно этот редактор будет применяться, когда используется программа `cron` или другие приложения. Система обращается к значению переменной `EDITOR` и применяет указанный редактор как заданный по умолчанию.

Для регистрации в системе *sybase* следует ввести команду:

```
$ isql -Udavet -Pabcd -Smethsys
```

где *-S* — имя сервера, с которым установлена связь. Переменная, содержащая имя сервера, называется DSQUERY. Имя сервера присваивается переменной DSQUERY. При регистрации в системе в случае, если имя сервера не указывается вместе с "-S", приложение обращается к переменной DSQUERY и использует значение этой переменной в качестве имени сервера. Для регистрации нужно ввести

```
$ isql -Udawet -Pabcd
```

Так функционирует большинство приложений.

14.2. Локальные переменные

Переменные интерпретатора shell могут использоваться сценариями в период функционирования интерпретатора shell. После завершения выполнения интерпретатора действие этих переменных прекращается. Например, локальная переменная *имя_файла* может иметь значение *loops.doc*. Это значение важно для пользователя только во время выполнения текущего интерпретатора команд; если запускается на выполнение другой процесс или происходит выход из среды интерпретатора команд, текущее значение локальной переменной не сохраняется. Переменные этого типа недоступны для других интерпретаторов shell или процессов, что имеет свои преимущества.

В табл. 14.1 представлены различные способы присваивания значений переменным.

Некоторые пользователи предпочитают заключать переменные интерпретатора команд в фигурные скобки. Это предупреждает ошибочную интерпретацию значений переменных. Делать это не обязательно, но такой прием удобен в работе.

При присвоении значений локальным переменным используется следующий формат:

```
$ имя_переменной = значение или ${имя_переменной = значение}
```

Обратите внимание, что с обеих сторон знака = имеются пробелы. Если значение содержит пробел, заключите все в двойные кавычки. Для переменных интерпретатора shell можно использовать как строчные, так и прописные буквы.

Таблица 14.1. Различные способы присваивания значений переменным

`Имя_переменной = значение`	Значение переменной присваивается переменной имя_переменной
`Имя_переменной + значение`	Значение переменной присваивается переменной имя_переменной, если оно установлено
`Имя_переменной:?значение`	На экран выводится сообщение об ошибке, если не установлена переменная имя_переменной
`Имя_переменной?значение`	На экран выводится сообщение о системной ошибке, если не установлена переменная имя_переменной
`Имя_переменной: = значение`	Значение переменной присваивается переменной имя_переменной, если она не установлена
`Имя_перменной:-значение`	Как и выше, но значение переменной имя_переменной не присваивается; оно может быть получено подстановкой

14.2.1. Отображение значения переменной

Чтобы отобразить значение отдельной переменной, достаточно применить команду `echo` и предварить имя переменной знаком `$`. Рассмотрим несколько примеров.

```
$ GREAT_PICTURE="die hard"
$ echo ${GREAT_PICTURE}
die hard

$ DOLLAR=99
$ echo ${DOLLAR}
99

$ LAST_FILE=ZLPSO.txt
$ echo ${LAST_FILE}
ZLPSO.txt
```

Переменные можно также комбинировать. Ниже переменной `ERROR_MSG` присваивается сообщение об ошибке, в котором используется значение переменной среды `LOGNAME`.

```
$ ERROR_MSG=" Sorry this file does not exist user $LOGNAME"
$ echo ${ERROR_MSG}
Sorry this file does not exist user dave
```

В приведенном выше примере интерпретатор shell сначала выводит текст, затем рассматривает переменную `$LOGNAME` и отображает ее значение.

14.2.2. Удаление значения переменной

Чтобы удалить значение переменной, достаточно применить команду `unset`:

```
unset имя_переменной

$ PC=enterprise
$ echo ${PC}
enterprise
$ unset PC
$ echo ${PC}
$
```

14.2.3. Отображение значений всех переменных интерпретатора shell

Чтобы просмотреть значения всех переменных интерпретатора shell, достаточно воспользоваться командой `set`.

```
$ set
...
PWD=/root
SHELL=/bin/sh
SHLVL=1
TERM=vt100
UID=7
USER=dave
dollar=99
```

```
great_picture=die hard
last_file=ZLPSO.txt
```

Вывод команды set может быть довольно обширен; при его изучении можно заметить, что интерпретатор shell значительно облегчил работу в среде.

14.2.4. Объединение значений переменных

Чтобы объединить значения переменных, достаточно последовательно расположить переменные:

```
echo ${имя_переменной}${имя_переменной}...
```

```
$ FIRST="Bruce "
$ SURNAME=Willis
$ echo ${FIRST}${SURNAME}
Bruce Willis
```

14.2.5. Проверка на наличие значения переменной (подстановка)

Допустим, что нужно проверить, была ли установлена либо инициализирована переменная. Если это не так, можно тут же воспользоваться другим значением. Формат используемой в этом случае команды будет следующий:

```
${переменная:-значение}
```

Если переменная установлена, она применяется. Если переменная не установлена, вместо нее используется значение, например:

```
$ COLOUR=blue
$ echo "The sky is ${COLOUR:-grey} today"
The sky is blue today
```

Переменная COLOUR имеет значение blue. Когда команда echo выводит на экран значение переменной COLOUR, выполняется проверка, была ли установлена переменная ранее. Если переменная установлена, используется это значение. Выполним сброс значения переменной и рассмотрим результат.

```
$ COLOUR=blue
$ unset COLOUR
$ echo "The sky is ${COLOUR:-grey} today"
The sky is grey today
```

В приведенном примере переменной не присваивается действительное значение; для присваивания значения следует применить формат:

```
${переменная: = значение}
```

Рассмотрим более полезный пример, в котором содержится запрос о времени создания платежной ведомости (payroll), а также относительно ее типа. Если в обоих случаях при указании времени и типа нажать клавишу [Enter], то значения этих переменных не будут установлены пользователем. В этом случае применяются новые значения, заданные по умолчанию (03:00 и Weekly). Затем значения передаются команде at, что свидетельствует о переносе выполнения задания на более поздний срок.

```
$ pg vartest
#!/bin/sh
# vartest
echo "what time do you wish to start the payroll [03:00]:"
read TIME
echo " process to start at ${TIME:=03:00} OK"
echo "Is it a monthly or weekly run [Weekly]:"
read RUN_TYPE
echo "Run type is ${RUN_TYPE:=Weekly}"
at -f $RUN_TYPE $TIME
```

Ниже при выводе данных выбирается клавиша [Enter] в обоих случаях.

```
$ vartest
what time do you wish to start the payroll [03:00]:
 process to start at 03:00 OK
Is it a monthly or weekly run [Weekly]:
 Run type is Weekly

warning: commands will be executed using /bin/sh
job 15 at 1999-05-14 03:00
```

Можно также проверить, установлено ли значение переменной, и затем выйти из системы, получив сообщение об ошибке. В следующем примере проверяется, установлен ли параметр для файла переменной длины.

```
$ echo "The file is ${FILES:?}"
sh: files: parameter null or not set
```

Приведенное выше сообщение не слишком информативно, но пользователь может задать свое собственное сообщение.

```
$ echo "The file is ${FILES:?" sorry cannot locate the variable files")"
sh: files: sorry cannot locate the variable files
```

Чтобы проверить, содержит ли переменная значение (а если не содержит, возвратить пустую строку), примените следующую команду:

```
${переменная: + значение}
```

Для инициализации переменной значением пустой строки примените команду:

```
variable =""
$ DESTINATION=""
```

14.2.6. Применение переменных, содержащих аргументы системных команд

Переменные можно также использовать для хранения информации, применяемой впоследствии для замены аргументов в командах системы. В следующем примере переменные сохраняют информацию об имени файла, которая необходима для копирования файла. С помощью переменной SOURCE можно узнать путь к файлу *passwd*, а с помощью переменной DEST — получателя этого файла. Обе эти переменные применяются в команде cp.

```
$ SOURCE="/etc/passwd"
$ DEST="/tmp/paeswd.bak"
$ cp ${SOURCE} ${DEST}
```

В примере ниже с помощью переменной DEVICE получаем путь к накопителю на магнитной ленте. Эти сведения используются затем в команде mt при перемотке ленты.

```
$ DEVICE="/dev/rmt/0n"
$ mt -f ${DEVICE} rewind
```

14.2.7. Как сделать переменную доступной только для чтения

Если переменной присваивается значение, то может потребоваться, чтобы это значение не изменялось. Для этого достаточно сделать переменную доступной только для чтения. Если пользователь попытается изменить значение, появится сообщение об ошибке, в котором указывается, что данная переменная предназначена только для чтения. В этом случае применяется следующий формат:

```
имя_переменной = значение
readonly имя_переменной
```

В примере ниже установлена переменная TAPE_DEV, которая указывает путь к одному из ленточных устройств системы. Затем она становится доступной только для чтения. При попытке изменить значение возникает сообщение об ошибке.

```
$ TAPE_DEV="/dev/rmt/0n"
$ echo ${TAPE_DEV}
/dev/rmt/0n
$ readonly TAPE_DEV
$ TAPE_DEV="/dev/rmt/1n"
sh: TAPE_DEV: read-only variable
```

Чтобы просмотреть все переменные, которые доступны только для чтения, примените команду readonly.

```
$ readonly
declare -r FILM="Crimson Tide"
declare -ri PPID="1"
declare -r TAPE_DEV="/dev/rmt/0n"
declare -ri UID="0"
```

14.3. Переменные среды

Переменные среды доступны для всех пользовательских процессов (часто их называют дочерними процессами). Пользователь регистрируется в процессе, который именуется родительским. Другие процессы, вызываемые из интерпретатора shell, называются дочерними. Переменные интерпретатора shell доступны только при выполнении текущего интерпретатора команд. Переменные среды доступны для всех дочерних процессов. Этими процессами могут быть редакторы, сценарии, приложения.

Переменные среды можно устанавливать в командной строке. Однако эти значения не сохраняются после выхода из системы. Поэтому переменные среды удобно инициализировать в пользовательском файле *.profile*. Системный администратор

может установить некоторые переменные среды в файле */etc/profile*. Поскольку переменные размещаются в профильных файлах, они инициализируются при регистрации в системе.

Обычно при обозначении переменных среды используются прописные буквы. Чтобы переменная была доступна всем процессам, достаточно применить команду `export`. Присваивание значений переменным среды проходит так же, как и в случае с переменными интерпретатора shell.

14.3.1. Присваивание значений переменным среды

Для присваивания значений переменным среды применяется команда:

```
VARIABLE_NAME= значение; export VARIABLE_NAME
```

Точка с запятой между двумя командами выступает в роли разделителя команд. К аналогичному результату можно прийти следующим образом:

```
VARIABLE_NAME = значение
export VARIABLE_NAME
```

14.3.2. Отображение значений переменных среды

Отображение значений переменных среды выполняется так же, как и в случае с переменными интерпретатора shell. Ниже приведено несколько примеров.

```
$ CONSOLE=tty1; export CONSOLE
$ echo $COMSOLE
tty1

$ MYAPPS=/usr/local/application; export MYAPPS
$ echo $MYAPPS /usr/local/application
```

Чтобы просмотреть глобально определенные переменные среды, достаточно воспользоваться командой `env`.

```
$ env
HISTSIZE=1000
HOSTNAME=localhost.localdomain
LOGNAME=dave
MAIL=/var/spool/mail/root
TERM=vt100
HOSTTYPE=i386
PATH=/sbin: /bin: /usr/sbin: /usr/bin: /usr/X11R6/bin: /root/bin:
CONSOLE=tty1
HOME=/home/dave
ASD=sdf
SHELL=/bin/sh
PS1=$
USER=dave
...
```

14.3.3. Удаление значений переменных среды

Чтобы удалить значение переменной, следует применить команду `unset`.

```
$ unset MYAPPS
$ echo $MYAPPS

$
```

14.3.4. Встроенные переменные интерпретатора shell

Интерпретатор Bourne shell располагает именами, которые зарезервированы для переменных среды. Эти названия переменных нельзя использовать иначе. Обычно они содержатся в файле */etc/profile*. Однако так бывает не всегда, поэтому пользователь может самостоятельно устанавливать значения этих переменных. Ниже приводится перечень переменных, встроенных в интерпретатор shell.

Переменная CDPATH

Переменная определяет выбор альтернативных каталогов, поэтому содержит ряд имен путей, разделенных двоеточием. Эта переменная применяется совместно с командой `cd`. Если значение переменной `CDPATH` установлено на момент выполнения команды `cd`, оно используется для выбора нового рабочего каталога. Если нужный каталог найден, он становится новым рабочим каталогом. Ниже приводится соответствующий пример.

```
$ CDPATH=:/home/dave/bin:/usr/local/apps; export CDPATH
```

Теперь, обратим внимание на следующую команду:

```
$ cd apps
```

Команда `cd` будет выполнять поиск среди каталогов, заданных с помощью переменной `CDPATH`. Если нужный каталог найден, именно он становится текущим рабочим каталогом.

Переменная EXINIT

При работе с редактором `vi` переменная `EXINIT` предоставляет опции для инициализации этого редактора. Например, для установки нумерации всех строк и шага табуляции, равного 10 пробелам, примените следующую команду:

```
$ EXINIT='set nu tab=10'; export EXINIT
```

Переменная HOME

Каталог *HOME* обычно указывается в файле *passwd*, во втором поле от конца файла. Именно в этом каталоге хранятся все персональные файлы пользователя. Если переменная установлена, для перехода в каталог *HOME* можно воспользоваться клавишей быстрого доступа, соответствующей команде `cd`.

```
$ HOME=/home/dave; export HOME
$ pwd
$ /usr/local
$ cd
$ pwd
$ /home/dave
```

Можно также применить команду:

```
$ cd $HOME
```

Переменная IFS

Переменная IFS применяется интерпретатором команд в качестве заданного по умолчанию разделителя полей. Внутренний разделитель полей может устанавливаться произвольным образом. По умолчанию разделителем полей служит пробел, символ новой строки или символ табуляции. Переменную IFS удобно использовать при разделении полей в файлах или переменных. Путь, включающий множество каталогов, более удобен для просмотра, если назначить переменной IFS двоеточие и вывести на экран значение переменной PATH.

```
$ export IFS=:
$ echo $PATH
/sbin /bin /usr/sbin /usr/bin /usr/X11R6/bin /root/bin
```

Чтобы вернуться к исходным установкам, достаточно применить следующую команду:

```
$ IFS=<space><tab>; export IFS
```

где значения <space><tab> задают использование в качестве разделителей полей символов пробела и табуляции.

Переменная LOGNAME

Переменная содержит регистрационное имя пользователя. Значение этой переменной устанавливается по умолчанию. Если этого не случилось, значение можно присвоить с помощью команды

```
$ LOGNAME=`whoami` ; export LOGNAME
$ echo $LOGNAME
dave
```

Переменная MAIL

Переменная MAIL хранит имя пути доступа и имя файла для вашего почтового ящика. По умолчанию ее значение /var/spool/mail/<регистрационное имя>. Интерпретатор shell периодически проверяет почтовый ящик на предмет наличия новых сообщений. Если пользователь получает электронное сообщение, он узнает об этом из командной строки. Если к почтовому ящику имеется другой путь, установите его с помощью переменной MAIL:

```
$ MAIL=/usr/mail/dave; export MAIL
```

Переменная MAILCHECK

Значение переменной MAILCHECK задается по умолчанию таким образом, чтобы наличие новой электронной почты проверялось каждые 60 секунд. Если электронную почту следует запрашивать реже, допустим каждые 2 минуты, примените команду:

```
$ MAILCHECK=120; export MAILCHECK
```

Переменная MAILPATH

Применяйте переменную MAILPATH, если вы располагаете более чем одним почтовым ящиком. При установке значения этой переменной перекрывается значение переменной MAIL:

```
$ MAILPATH=/var/spool/dave:/var/spool/admin; export MAILPATH
```

В данном примере переменная MAIL проверяет два почтовых ящика — *dave* и *admin*.

Переменная PATH

Переменная PATH хранит сведения о каталогах, где находятся выполняемые команды или сценарии. Важно, чтобы эта последовательность указывалась правильно, поскольку при обращении к этой переменной существенно сокращается время выполнения команды или сценария. Например, если в каком-либо каталоге нет заданной команды, то не следует его просматривать. В общем случае лучше сначала задать каталог *HOME*, за которым будет следовать перечень каталогов. Перечень включает каталоги, отсортированные по частоте применения. Сначала указываются наиболее часто применяемые каталоги, а затем те из них, которые применяются реже всего. Если нужно найти текущий рабочий каталог, то независимо от места его нахождения используйте точку. Каждый каталог отделяется двоеточием. Рассмотрим пример.

```
$ PATH=$HOME/bin:.:/bin:/usr/bin; export PATH
```

Согласно приведенному примеру, сначала производится поиск в каталоге *HOME/bin*, затем — в текущем каталоге, каталоге */bin* и в каталоге */usr/bin*.

Значение переменной PATH для системных каталогов задается в файле */etc/profile*. Чтобы воспользоваться переменной PATH, определяя собственный вариант переменной PATH, достаточно в конце указать точку с запятой (;).

```
$ PATH=$PATH:/$HOME/bin:. ; export PATH
```

В данном случае используется переменная PATH, определенная в файле */etc/profile*, а также указан каталог *$HOME/bin* и текущий каталог. В общем случае нежелательно использовать текущий каталог при записи первого пути поиска. Если допустить подобное, то текущий каталог станет доступным и для других пользователей.

Переменная PS1

Основная командная строка включает символ приглашения интерпретатора shell. По умолчанию для основной командной строки используется символ #, а для любой другой командной строки применяется символ $. В командной строке можно использовать любые символы. Ниже приводятся два примера.

```
$ PS1="star trek:"; export PS1
star trek:
$ PS1="->" ; export PS1
->
```

Переменная PS2

Значение этой переменной задает вид вторичной командной строки. По умолчанию этот тип командной строки определяется символом >. Подобная командная

строка используется при выполнении многострочной команды или в случае, когда размер команды превышает длину строки.

```
$ PS2="@:"; export PS2
$ for loop in *
@:do
@:echo $loop
...
```

Переменная SHELL

Переменная SHELL включает сведения об интерпретаторе shell, заданном по умолчанию. Интерпретатор команд обычно указан в файле */etc/passwd*. Это значение нельзя отменить, даже если приходится использовать другой интерпретатор команд.

```
$ echo $SHELL
/bin/sh
```

Переменная TERMINFO

Переменная, определяющая инициализацию терминала, хранит сведения о местонахождении файлов конфигурации терминала. Обычно эти файлы находятся либо в каталоге */usr/lib/terminfo*, либо в каталоге */usr/share/terminfo*.

```
$ TERMINFO=/usr/lib/terminfo; export TERMINFO
```

Переменная TERM

Переменная TERM представляет тип терминала. Эта переменная служит для указания типа управляющих последовательностей, необходимых для правильного функционирования экрана и клавиатуры. Самыми распространенными типами терминалов являются vt100, vt220, vt220-8, wyse и т.д.

```
$ TERM=vt100; export TERM
```

Переменная TZ

Эта переменная позволяет определить часовой пояс. Значение переменной TZ устанавливает администратор. Если для интерпретатора shell потребуется изменить значение этой переменной, обратите внимание на следующие примеры:

```
$ echo $TZ
GMT2EDT
```

В данном случае возвращается значение, которое свидетельствует о том, что отсчет времени происходит по Гринвичу (Greenwich Mean Time) (временное смещение относительно GMT составляет 0 часов) в диапазоне летнего времени (Eastern Daylight Saving).

14.3.5. Другие переменные среды

Для переменных среды резервируются и другие названия; эти имена используются в иных приложениях. Ниже приводятся наиболее распространенные имена. Не забывайте, что значения для этих переменных не устанавливаются; их задает сам пользователь.

Переменная EDITOR

Присвойте этой переменной название предпочитаемого вами текстового редактора.

```
$ EDITOR=vi; export EDITOR
```

Переменная PWD

Эта переменная представляет имя пути доступа для текущего каталога. Обычно значение переменной изменяется всякий раз при выполнении команды cd.

Переменная PAGER

Данная переменная хранит сведения о командах программы постраничной разбивки. К подобным командам можно отнести, например, pg, more. Система проверяет значение этой переменной, когда пользователь выполняет просмотр страниц man.

```
$ PAGER='pg -f -p%d'; export PAGER
```

Переменная MANPATH

Представляет каталоги, которые включают страницы справочной программы man. Имя каждого каталога отделяется двоеточием.

```
$ MANPATH=/usr/apps/man:/usr/local/man; export MANPATH
```

Переменная LPDEST или переменная PRINTER

Эта переменная представляет имя принтера, заданного по умолчанию. При выводе на печать сохраняется имя принтера, заданное пользователем.

```
$ LPDEST=hp3si_systems
```

14.3.6. Применение команды set

Экспорт переменных среды можно задать при установке этих переменных в файле *$HOME .profile*. Воспользуйтесь командой set с параметром "-a" в виде "set -a". Это означает, что выполняется экспорт всех переменных. Не применяйте этот метод при работе с файлом */etc/profile*. Данный метод применим только при работе с файлом *$HOME .profile*.

```
$ pg .profile
#.profile
set -a
MAIL=/usr/mail/${LOGNAME:?}
PATH=$PATH:$HOME:bin
#
EDITOR=vi
TERM vt220
ADMIN=/usr/adm
PS1=" `hostname` >>"
```

14.3.7. Экспорт переменных в дочерние процессы

Для тех, кто только начинает работать с интерпретатором shell, существенное затруднение представляет экспорт переменных в дочерние процессы. Теперь, после рассмотрения основных принципов работы с переменными среды, перейдем к

выполнению практических задач. Ниже приводится пример сценария, который вызывает другой сценарий. Вызываемый сценарий фактически и является дочерним процессом.

Рассматриваемый листинг включает два небольших сценария под именами *father* и *child*. Сценарий *father* задает переменную с именем FILM, значение которой "A Few Good Men". Эта переменная отображается на экране, затем вызывается другой сценарий под названием child. Данный сценарий отображает переменную FILM из первого сценария. Перед отображением на экране значением переменной FILM становится "Die Hard". Затем управление передается назад, сценарию *father*, и значение переменной снова выводится на экран.

```
$ pg father
#!/bin/sh
# сценарий father.
echo "this is the father"
FILM="A Few Good Men"
echo "I like the film :$FILM"
# вызов сценария child
child
echo "back to father"
echo "and the film is :$FILM"

$ pg child
#!/bin/sh
# сценарий child
echo "called from father..i am the child"
echo "film name is :$FILM"
FILM="Die Hard"
echo "changing film to :$FILM"
```

Рассмотрим, что будет отображаться на экране в результате выполнения сценария.

```
$ father
this is the father
I like the film :A Few Good Men
 called from father..i am the child
film name is :
changing film to :Die Hard
back to father
and the film is :A Few Good Men
```

Сценарий *child* не может возвратить переменную FILM, поскольку эта переменная не может быть экспортирована в сценарий *father*.

Теперь, если команду export добавить в сценарий *father*, сценарий *child* получит всю информацию о переменной FILM.

```
pg father
#!/bin/sh
# сценарий father.
echo "this is the father"
FILM="A Few Good Men"
echo "I like the film :$FILM"
# вызов сценария child
# сначала экспортируйте переменную
```

```
export FILM
child
echo "back to father"
echo "and the film is :$FILM"

$ father2
this is the father
I like the film :A Few Good Men
called from father..i am the child
film name is :A Few Good Men
changing film to :Die Hard
back to father
and the film is :A Few Good Men
```

После того как команда `export` включена в состав сценария, можно выполнить произвольное число процессов. И всем этим процессам известно о наличии переменной `FILM`.

Нельзя выполнять экспорт переменных из дочерних процессов в родительский процесс; однако с помощью изменения направления этого можно добиться.

14.4. Позиционные параметры командной строки

Как указывалось в начале главы, существуют переменные четырех типов. Имеются переменные интерпретатора команд, переменные среды, а также переменные двух других типов, которые являются специальными, поскольку они предназначены только для чтения. Существуют позиционные и специальные параметры. Рассмотрим позиционные параметры.

Если информация передается shell-сценарию, с помощью позиционных параметров можно получить доступ к ней. Каждый параметр передает сценарию заданное количество ссылок. Можно передавать произвольное число аргументов, но доступными являются только первые девять из них. Это ограничение преодолевается путем использования команды `shift`. Далее в книге будет показано, как работает команда `shift`. Нумерация параметров начинается с единицы и завершается девятью; каждый параметр, доступ к которому следует получить, предваряется знаком доллара $. Первый параметр, имеющий номер нуль, резервируется для представления действительного названия сценария; это значение может использоваться независимо от того, содержатся ли в сценарии параметры.

В приведенной таблице в качестве примера демонстрируется, как можно получить доступ к каждому параметру, если в сценарий передается строка "Did You See The Full Moon"

$0	$1	$2	$3	$4	$5	$6	$7	$8	$9
Наименование сценария	Did	You	See	The	Full	Moon			

14.4.1. Применение в сценариях позиционных параметров

Воспользуемся приведенным выше примером в следующем сценарии.

```
$ pg param
#!/bin/sh
# параметры
```

```
echo "This is the script name       : $0"
echo "This is the first parameter   : $1"
echo "This is the second parameter  : $2"
echo "This is the third parameter   : $3"
echo "This is the fourth parameter  : $4"
echo "This is the fifth parameter   : $5"
echo "This is the sixth parameter   : $6"
echo "This is the seventh parameter : $7"
echo "This is the eighth parameter  : $8"
echo "This is the ninth parameter   : $9"
```

```
$ param Did You See The Full Moon
This is the script name       : ./param
This is the first parameter   : Did
This is the second parameter  : You
This is the third parameter   : See
This is the fourth parameter  : The
This is the fifth parameter   : Full
This is the sixth parameter   : Moon
This is the seventh parameter :
This is the eighth parameter  :
This is the ninth parameter   :
```

Если передаются шесть параметров, седьмой, восьмой и девятый будут пустыми, чего и следовало ожидать. Обратите внимание, что первый параметр содержит название сценария. Этот параметр удобно использовать, если сценарий выдает сообщение об ошибках. Ниже приводится другой пример, где получается название сценария.

```
$ pg param2
#!/bin/sh
echo "Hello world this is $0 calling"
```

```
$ param2
Hello world this is ./param2 calling
```

Заметьте, что параметр `$0` также позволяет получить путь доступа к текущему каталогу. Для получения названия сценария предварите параметр `$0` командой `basename`.

```
$ pg param2
#!/bin/sh
echo "Hello world this is 'basename $0' calling"
```

```
$ param2
Hello world this is param2 calling
```

14.4.2. Передача параметров в системные команды

Параметры можно передавать внутри сценария в системную команду. В следующем примере используется команда `find`. Параметр `$1` указывает имя искомого файла.

```
$ pg findfile
#!/bin/sh
# findfile
find / -name $1 -print

$ findfile passwd
/etc/passwd
/etc/uucp/passwd
/usr/bin/passwd
```

Ниже приводится другой пример. В виде параметра `$1` файл *user-id* передается команде `grep`; команда `grep` использует эти сведения для поиска файла *passwd*, где содержится полное имя пользователя.

```
$ pg who_is
#!/bin/sh
# who_is
grep $1 passwd | awk -F: {print $4}'

$ who_is seany
Seany Post
```

14.4.3. Специальные параметры

Теперь, когда вы изучили, как получить доступ к параметрам shell-сценариев и применять их, было бы полезным узнать об этих параметрах больше. Рассмотрим, каким образом сценарий применяет специальные параметры. Существует семь специальных параметров, которые представлены в табл. 14.2.

Таблица 14.2. Специальные параметры интерпретатора shell

`$#`	Число аргументов, передаваемых сценарию
`$*`	В отдельной строке отображаются все аргументы, которые передаются сценарию. Здесь может содержаться более девяти параметров, в отличие от позиционных параметров
`$$`	Текущий идентификатор PID для выполняющегося сценария
`$!`	Идентификатор PID для последнего процесса, который выполняется в фоновом режиме
`$@`	Означает то же самое, что и параметр `$#`, но, если параметр заключен в кавычки, то и каждый аргумент отображается в кавычках
`$-`	Отображение текущих опций интерпретатора команд; аналогично применению команды set
`$?`	Показывает код завершения последней команды. Значение 0 свидетельствует об отсутствии ошибок, любое другое значение — о, их наличии

Преобразуем сценарий `param`, применяя некоторые специальные параметры. Затем выполним сценарий.

```
$ pg param
#!/bin/sh
# все параметры
echo "This is the script name          : $0"
echo "This is the first parameter      : $1"
```

```
echo "This is the second parameter     : $2"
echo "This is the third parameter      : $3"
echo "This is the fourth parameter     : $4"
echo "This is the fifth parameter      : $5"
echo "This is the sixth parameter      : $6"
echo "This is the seventh parameter    : $7"
echo "This is the eighth parameter     : $8"
echo "This is the ninth parameter      : $9"
echo "The number of arguments passed   :$@"
echo "Show all arguments               : $*"
echo "Show me my process ID            : $$"
echo "Show me the arguments in quotes  :" "$@"
echo "Did my script go with any errors :$?"
```

```
$ param Merry Christmas Mr Lawrence
This is the script name          : ./param
This is the first parameter      : Merry
This is the second parameter     : Christmas
This is the third parameter      : Mr Lawrence
This is the fourth parameter     :
This is the fifth parameter      :
This is the sixth parameter      :
This is the seventh parameter    :
This is the eighth parameter     :
This is the ninth parameter      :
The number of arguments passed   :3
Show all arguments               : Merry Christmas Mr Lawrence
Show me my process ID            : 630
Show me the arguments in quotes  : "Merry" "Christmas" "Mr Lawrence"
Did my script go with any errors :0
```

При выводе данных с помощью специальных параметров можно получить много полезной информации о сценарии. Существует возможность проверить, какое число аргументов передается, а также идентификатор процесса этого сценария в том случае, если нужно уничтожить сценарий.

14.4.4. Код завершения последней команды

Обратите внимание, что параметр `$?` возвращает значение `0`. Этот прием можно использовать во всех командах или сценариях, если необходимо выяснить, успешно ли выполнена последняя команда. С помощью этой информации можно продолжить работу со сценарием. Если возвращается значение 0, значит все идет успешно; значение 1 означает появление ошибки.

Ниже приводится пример. Сначала файл копируется в каталог */tmp* и с помощью параметра `$?` проверяется результат.

```
$ cp ok.txt /tmp
$ echo $?
0
```

Скопируем файл в несуществующий каталог:

```
$ cp ok.txt /usr/local/apps/dsf
cp: cannot create regular file '/usr/local/apps/dsf': No such file or
```

```
directory

$ echo $?
1
```

При проверке состояния возврата с помощью параметра `$?` отображается 1. Это свидетельствует о наличии ошибок. Появляется сообщение о системной ошибке "cp:cannot...", поэтому не требуется проверять код завершения последней команды. Сценарии функционируют с помощью системных команд. Обычно пользователь не заинтересован в отображении большого количества значений. Поэтому выходной поток перенаправляется в системную корзину */dev/null*. Каким образом пользователь узнает, что последняя команда была успешной? С помощью кода завершения последней команды. Рассмотрим пример, иллюстрирующий изложенный материал.

```
$ cp ok.txt /usr/local/apps/dsf >/dev/null 2>&1
$ echo $?
1
```

Если данные вывода, включая сообщения об ошибках, перенаправить в системную корзину, невозможно установить, выполнялась ли последняя команда. Но благодаря применению параметра `$?`, который возвращает значение 1, становится известно, что при выполнении команды были допущены ошибки.

При проверке в сценариях кода завершения удобно проверяемым операциям присваивать содержательные названия. Желательно, чтобы название отражало функции данной операции; кроме того, сценарии приобретут более структурированный вид.

```
$ cp ok.txt /usr/local/apps/dsf >/dev/null 2>&1
$ cp_status=$?
$ echo $cp_status
1
```

14.5. Заключение

Работа с переменными значительно облегчает функционирование интерпретатора shell. Автоматизируется ввод данных, повышается производительность труда администратора. Переменная интерпретатора shell может принимать произвольные значения. Использование специальных переменных расширяет функциональные возможности сценариев и позволяет получать больше информации о параметрах, передаваемых в сценарии.

ГЛАВА 15

Использование кавычек

В главе 14 обсуждались методы работы с переменными и операции подстановки. Чаще всего ошибки в использовании кавычек возникают при выполнении подстановок переменных в сценариях. Кавычки оказывают существенное влияние на формирование командных строк.

В этой главе рассматриваются следующие темы:

- правила применения кавычек;
- двойные, одинарные и обратные кавычки;
- отмена специального значения символов с помощью символа обратной косой черты.

15.1. Правила применения кавычек

Рассмотрим некоторые основные правила использования кавычек. Излагаемый материал будет сопровождаться большим количеством примеров. Вопросы использования кавычек обсуждаются и в следующих двух частях книги.

Некоторые пользователи не утруждают себя и не применяют при выводе на экран текстовых строк двойные кавычки. Применение кавычек может оказать существенное влияние на работу сценария. Необходимо, чтобы применение двойных кавычек при выводе на экран текстовых строк стало непреложным правилом для каждого пользователя. Рассмотрим следующий пример.

```
$ echo Hit the star button to exit *
Hit the star button to exit DIR_COLORS HOSTNAME Muttrc X11 adjtime aliases
alias
 ...
```

Приведенный текст выводится на экран. Поскольку двойные кавычки не были использованы, интерпретатор команд расценивает знак * как знак подстановки. Заключая выражение в двойные кавычки, получим:

```
$ echo "Hit the star button to exit *"
Hit the star button to exit *
```

Благодаря применению кавычек выражение интерпретируется верно. В табл. 15.1 приведены комментарии к различным типам кавычек.

Таблица 15.1. Использование кавычек интерпретатором shell

""	Двойные кавычки
''	Одинарные кавычки
` `	Символ "тупого ударения" или обратные кавычки
\	Символ обратной косой черты

15.2. Двойные кавычки

Двойные кавычки применяются, когда необходимо буквальное восприятие всех символов, за исключением символов: $, `, \. Символы доллара, обратной кавычки и символ обратной косой черты сохраняют свое специальное значение при функционировании интерпретатора shell. Если при выводе на экран переменной присваивается текстовая строка, заключенная в двойные кавычки, речь идет о выводе на экран самой переменной.

```
$ STRING="MAY DAY, MAY DAY, GOING DOWN"
$ echo "$STRING"
MAY DAY, MAY DAY, GOING DOWN

$ echo $STRING
MAY DAY, MAY DAY, GOING DOWN
```

Предположим, что сведения о системной дате присваиваются переменной с именем *mydate*:

```
$ MYDATE="date"
$ echo $MYDATE
date
```

Интерпретатор команд воспринимает информацию в строке "как есть", т.е. дате не присваивается специальное значение. Поэтому в качестве даты фигурирует слова *data*.

Двойные кавычки следует применять при поиске строки, содержащей пробелы. Воспользуемся командой `grep` для поиска имени Davey Wire. Если не применять кавычки, команда `grep` воспримет Davey как строку для поиска, а Wire — как файл.

```
$ grep Davey Wire /etc/passwd
grep: Wire: No such file or directory
```

Для устранения возможных недоразумений следует заключить строку в двойные кавычки. Тогда интерпретатор команд проигнорирует пробелы. При работе с текстовыми строками следует всегда применять двойные кавычки, независимо от количества слов в строке.

```
$ grep "Davey Wire" /etc/passwd
 davyboy:9sdJUK2s:106:Davey Wire:/home/ap
```

Чтобы включить переменную в строку для вывода на экран, можно применить двойные кавычки. В следующем примере интерпретатор shell выводит на экран строку, считывает знак $, трактует его как переменную и заменяет переменную $BOY значением `boy`.

```
$ BOY="boy"
$ echo " The $BOY did well"
The boy did well

$ echo " The "$BOY" did well"
The boy did well
```

15.3. Одинарные кавычки

Применение одинарных кавычек во многом аналогично применению двойных кавычек. Интерпретатор shell получает указание не выполнять подстановку этих значений; другими словами, символы, заключенные в такие кавычки, лишены специального значения. То, что заключено в одинарные кавычки, воспринимается буквально. Рассмотрим пример, аналогичный примеру с двойными кавычками:

```
$ GIRL='girl'
$ echo "The '$GIRL' did well"
The 'girl' did well
```

15.4. Обратные кавычки

Обратные кавычки позволяют присваивать переменным данные вывода системных команд. Символы, заключенные в обратные кавычки, воспринимаются интерпретатором shell как системная команда, которую следует выполнить. С помощью этого метода можно осуществить подстановку данных вывода в переменную. Можно также комбинировать разные кавычки, как это показано далее.

В следующем примере интерпретатор shell пытается выполнить подстановку слова *hello*. Но, поскольку нет ни команды, ни сценария с таким именем, отображается сообщение об ошибке.

```
$ echo `hello`
sh: hello: command not found
```

Применим снова команду `date`.

```
$ echo `date`
Sun May 16 16:40:19 GMT 1999
```

Теперь команда записана правильно, и интерпретатор shell может выполнить корректную подстановку этой команды.

Присвоим переменной *mydate* данные вывода команды `date`. Ниже приводится формат команды `date`:

```
$ date +%A" the "%e" of "%B" "%Y
Sunday the 16 of May 1999
```

Присвоим значение переменной *mydate* и отобразим его.

```
$ mydate=`date +%A" the "%e" of "%B" "%Y`
$ echo $mydate
Sunday the 16 of May 1999
```

Конечно, переменной *mydate* можно присвоить весь вывод команды `date`:

```
$ mydate='date'
$ echo $mydate
Sun May 16 16:48:16 GMT 1999
```

Ниже приводится другой пример. Обратные кавычки содержатся внутри двойных кавычек:

```
$ echo "The date today is `date` "
The date today is Sun May 16 16:56:53 GMT 1999
```

Выведем на экран небольшое сообщение, включающее сведения о количестве пользователей в системе.

```
$ echo "There are `who | wc -l` users on the system"
There are 13 users on the system
```

В приведенном примере выводится текстовая строка. Интерпретатор shell воспринимает обратные кавычки, рассматривает текст, который заключен в обратные кавычки, интерпретирует его и выполняет.

15.5. Обратная косая черта

Символ обратной косой черты указывает интерпретатору shell, что следующий за ним символ не имеет специального значения. Специальное значение могут иметь такие символы: `& * = ^ $ ' " | ?.`

В результате применения команды `echo` вместе с символом * на экран выводится не символ звездочки, а листинг. В этот листинг включается текущий каталог.

```
$ echo *
conf.linuxconf conf.modules cron. daily cron. houly cron.monthly
cron.weekly crontab csh.cshrc default dosemu.conf dosemu.users exports
fdprm fstab gettydefs gpm-root.c
onf group group- host.conf hosts hosts.allow hosts.deny httpd inetd
...
```

Для отмены специального значения звездочки, примените символ обратной косой черты:

```
$ echo \*
*
```

Аналогичный результат можно получить с помощью команды `$$`. Интерпретатор shell воспринимает ее как текущий `PID` (ID процесса). Чтобы отключить специальное значение и вывести на экран символ *$$*, нужно перед этим символом поместить символ обратной косой черты:

```
$ echo $$
284
$ echo \$$
$$
```

Для вывода на экран символов, заданных восьмеричными ASCII-кодами, перед ними следует размещать символ обратной косой черты. В противном случае интерпретатор shell воспримет их как обычные числа.

```
$ echo "This is a copyright 251 sign"
This is a copyright \251 sign"
$ echo "This is a copyright \251 sign"
This is a copyright © sign"
```

В Linux...

Не забывайте при выводе управляющих символов указывать параметр "-e".

```
$ echo -e "This is a copyright \251 sign"
This is a copyright © sign
```

Если при использовании команды `expr` применить символ * для обозначения операции умножения, то отобразится сообщение об ошибке. Чтобы символ * обозначал операцию умножения, следует поместить перед ним символ обратной косой черты.

```
$ expr 12 * 12
expr: syntax error

$ expr 12 \* 12
144
```

Чтобы оператор `echo` включал метасимволы, воспользуйтесь символом обратной косой черты. В следующем примере отображается цена $19.99. Поскольку не применяется символ обратной косой черты, интерпретатор shell трактует эту цену иначе.

```
$ echo * "That video looks a good price for $19.99"
That video looks a good price for 9.99
```

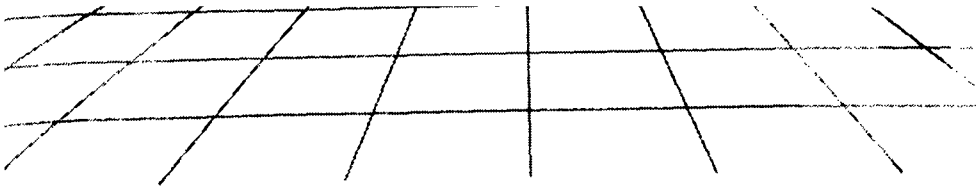
Если же перед знаком доллара поставить знак обратной косой черты, то получим искомый результат.

```
$ echo "That video looks a good price for \$19.99"
That video looks a good price for $19.99
```

15.6. Заключение

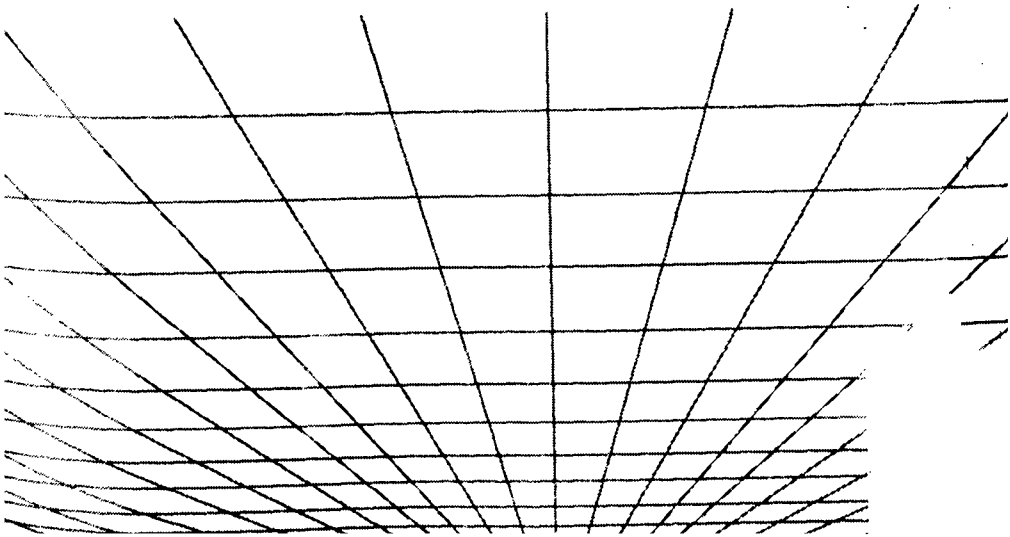
При использовании кавычек желательно следовать двум правилам:

1. Всегда применяйте двойные кавычки при выводе на экран строк командой `echo`.
2. Если при использовании кавычек полученный результат разочаровал вас, поменяйте кавычки и повторите команду. В конце концов, существует всего три вида кавычек.



ЧАСТЬ 4

Основы shell-программирования



ГЛАВА 16

Понятие о shell-сценарии

В shell-сценарий может включаться одна или несколько команд; здесь нет общепринятых правил. Зачем же создавать целый сценарий ради двух-трех команд? Все зависит от предпочтений пользователя.

В этой главе рассматриваются следующие темы:

- цель создания shell-сценариев;
- основные компоненты shell-сценария;
- способы запуска shell-сценария.

16.1. Зачем создаются shell-сценарии

Применение shell-сценариев позволяет экономно расходовать рабочее время при выполнении важных и сложных заданий. В конце концов, почему бы для выполнения определенного задания не использовать листинг команд? Затем можно только просмотреть данные вывода. Если результаты удовлетворительные, можно переходить к выполнению следующего задания. Если же результаты не устраивают вас, следует внимательно изучить листинг. Shell-сценарии производят сортировку файлов, вставку текста в файлы, перемещение файлов, удаление строк из файлов, а также удаление старых файлов из системы. Shell-сценарии также выполняют в системе некоторые административные задания. В этих сценариях используются переменные, условные, арифметические и циклические конструкции, которые можно отнести к системным командам. За счет этих возможностей сценарии создаются довольно быстро. Интерпретатор команд может применять в качестве вводных данных для одной команды информацию, полученную при выполнении другой команды. Чтобы shell-сценарий применялся в различных системах UNIX и Linux, в него нужно внести лишь небольшие изменения. Это связано с тем, что интерпретатор shell обладает высокой степенью универсальности. Определенные трудности возникают лишь вследствие ориентации системных команд на определенные системы.

16.1.1. Не отказывайтесь от новых идей

Не впадайте в уныние, если созданный вами сценарий не оправдывает ваших ожиданий. Просто помните о том, что какой бы ни был сценарий, возникшие в результате его выполнения неприятные последствия являются устранимыми. Однако перед выполнением пробного сценария подумайте о том, созданы ли резервные копии? В любом случае не стоит отказываться от новых идей. Если вы придерживаетесь иного мнения, вряд ли вы искренни перед собой и, что более важно, вы не сможете оценить оригинальность новых решений, если не реализуете их на практике.

16.2. Структура сценария

В книге недостаточно внимания уделяется методике создания совершенных сценариев. Здесь речь идет, в основном, о программном коде, который может послужить прототипом для создания целого семейства новых сценариев. Не следует использовать сложные сценарии, если единственной целью этого является получение абсолютно непонятного программного кода. Если бы автор этой книги придерживался подобной методики, он произвел бы определенное впечатление на читателей, но осознание преимуществ подобного кода отняло бы массу драгоценного времени. Поэтому главной задачей данной книги было создание простого кода, при написании которого применялись базовые технологии создания сценариев. Благодаря этому читатель может довольно быстро освоить эти технологии и поражать своими знаниями друзей и знакомых!

Сценарий не относится к компилируемым программам, поскольку он интерпретируется построчно. Первой строкой сценария всегда должна быть строка вида:

```
#!/bin/sh
```

Система получает указание, где следует искать интерпретатор Bourne shell.

Каждый сценарий может содержать комментарии; если комментарии размещаются в сценарии, первым символом в строке комментария будет символ #. Встретив подобный символ, интерпретатор команд игнорирует строку комментария. Желательно, чтобы имя сценария находилось в строке, расположенной ниже строки комментария.

Сценарий просматривается интерпретатором команд в направлении сверху вниз. Перед выполнением сценария требуется воспользоваться командой `chmod`, устанавливающей право доступа на выполнение. Убедитесь в том, что правильно указан путь к сценариям, тогда для его выполнения достаточно будет указать имя файла сценария.

16.3. Выполнение сценария

Ниже рассматривается пример, который уже обсуждался ранее. В данном случае файл называется *cleanup*.

```
$ pg cleanup
#!/bin/sh
# имя: cleanup
# это общий сценарий, выполняющий очистку
echo "starting cleanup...wait"
rm /usr/local/apps/log/*.log
tail -40 /var/adm/messages /tmp/messages
rm/var/adm/messages
 mv/tmp/messages /var/adm/messages
echo "finished cleanup"
```

Приведенный выше сценарий отменяет отображение сообщений *var/adm/* путем усечения файла сообщений. В задачи этого сценария также входит удаление всех журнальных файлов в каталоге */usr/local/apps/log*.

Для выполнения сценария применим команду `chmod`:

```
$ chmod u+x cleanup
```

Чтобы запустить сценарий на выполнение, введите его название:

```
$ cleanup
```

При отображении сообщения об ошибке, например:

```
$ cleanup
sh: cleanup: command not found
```

воспользуйтесь командой:

```
$ ./cleanup
```

Если перед выполнением сценария нужно указать путь доступа к нему или же сценарий сообщает, что не может обнаружить команду, достаточно в значение переменной PATH из файла *.profile* добавить каталог *bin*. При вводе следующей информации сначала убедитесь, что вы находитесь в каталоге *$HOME/bin*:

```
$ pwd
$ /home/dave/bin
```

Если последняя часть команды pwd включает название подкаталога */bin*, его следует использовать при указании имени пути. Измените файл *.profile* и добавьте в файл *.profile* каталог *$HOME/bin*:

```
PATH = $PATH:$HOME/bin
```

В случае если подкаталог */bin* отсутствует, создайте его; сначала удостоверьтесь, что находитесь в начальном каталоге.

```
$ cd $HOME
$ mkdir bin
```

После этого добавьте каталог *bin* в переменную *PATH* в файле *.profile*, затем заново инициализируйте файл *.profile*

```
$ /.profile
```

Теперь все должно получиться.

Если же проблемы остались, просмотрите главы 2 и 13. В этих главах содержатся сведения о командах find и xargs, а также о настройках переменных среды, которые окажутся полезными при создании и выполнении сценариев.

Все листинги книги являются полными. Чтобы запустить на выполнение сценарии, введите в файл команды, сохраните его на диске и выйдите из текстового редактора. Затем воспользуйтесь командой chmod для установки права выполнения. Теперь можно переходить к делу.

16.4. Заключение

Глава служит кратким введением в методику работы с shell-сценариями. Вероятно, вам пока нет необходимости подробно изучать вопросы, связанные с функционированием сценариев, вводом/выводом данных интерпретатора shell и т.п. Необходимые знания придут после внимательного изучения материала книги. Глава помогает пользователю уяснить, как начать работу с shell-сценарием.

ГЛАВА 17

Проверка условий

При создании сценария уточняется идентичность строк, права доступа к файлу или же выполняется проверка численных значений. На основе результатов проверки предпринимаются дальнейшие действия. Проверка обычно осуществляется с помощью команды `test`. Команда `test` используется для тестирования строк, проверки прав доступа к файлу и численных данных. Как будет показано в следующей главе, эта команда хорошо согласуется с условной конструкцией `if, then, else`.

В главе рассматриваются следующие темы:

- применение команды `test` для обработки файлов, строк и чисел;
- использование команды `expr` при проверке численных значений и строк.

Команда `expr` выполняет проверку и вывод численных данных. Команды `test` и `expr` с помощью кода завершения последней команды `$?` выводят на экран значение 0, если заданное условие выполняется, и 1, если условие не выполняется.

17.1. Проверка прав доступа к файлу

Основные форматы команды `test`:

```
test условие
```

или

```
[ условие ]
```

Обратите внимание, что с обеих сторон от условия обязательно остаются пробелы.

При проверке права на доступ к файлу может применяться довольно много условий. В табл. 17.1 содержится перечень наиболее распространенных условий.

Таблица 17.1. Проверка прав доступа к файлу

`-d`	Каталог
`-f`	Обычный файл
`-L`	Символическая связь
`-r`	Файл для чтения
`-s`	Файл имеет ненулевой размер, он не пуст
`-w`	Файл для записей
`-u`	Файл имеет установленный бит *suid*
`-x`	Исполняемый файл

Чтобы проверить, установлено ли право записи для файла *scores.txt*, применяются оба метода проверки. При осуществлении проверки используется код завершения последней команды. Не забывайте о том, что возвращаемое нулевое значение свидетельствует о том, что условие выполняется, любое другое значение говорит о наличии ошибки.

```
$ ls -l scores.txt
-rw-r--r-- 1 dave admin 0 May 15 11:29 scores, txt
$ [ -w scores.txt ]
$ echo $?
0

$ test -w scores.txt
$ echo $?
0
```

В обоих случаях отображается значение нуль, поэтому для данного файла установлены права записи. Теперь рассмотрим, является ли этот файл исполняемым.

```
$ [ -x scores.txt ]
$ echo $?
1
```

Нет, не является, что и следовало ожидать после изучения списка прав доступа для файла *scores.txt*.

В следующем примере проверяется существование каталога *appsbin*.

```
drwxr-xr-x 2 dave admin 1024 May 15 15:53 appsbin
$ [ -d appsbin ]
$ echo $?
0
```

Ответ утвердительный, каталог *appsbin* присутствует.

Чтобы проверить, установлен ли бит *suid*, примените:

```
-rwsr-x-- 1 root root 28 Apr 30 13:12 xab
$ [ -u xab ]
$ echo $?
0
```

Из этого примера следует, что бит *suid* установлен.

17.2. Применение логических операторов при осуществлении проверки

Итак, проверка прав доступа к файлу была осуществлена, но иногда возникает необходимость в сравнении различных прав доступа. Чтобы реализовать подобную проверку интерпретатор shell предлагает три типа логических операторов:

- `-a` Логическое *AND*, возвращает истину, если обе части оператора принимают истинное значение
- `-o` Логическое *OR*, возвращает истину, если какая-либо из частей оператора может принимать истинное значение
- `!` Логическое *NOT*, возвращает истину, если условие ложно

А теперь выполним сравнение следующих файлов:

```
-rw-r--r-- 1 root root 0 May 15 11:29 scores.txt
-rwxr-xr-- 1 root root 0 May 15 11:49 results.txt
```

В следующем примере проверяется, установлены ли для файлов права чтения.

```
$ [ -w results.txt -a -w scores.txt ]
$ echo $?
0
```

Ответ звучит утвердительно.

Чтобы проверить, установлены ли для какого-либо из файлов права выполнения, воспользуйтесь логическим оператором *OR*.

```
$ [ -x results.txt -o -x scores.txt ]
$ echo $?
0
```

Файл *scores.txt* не является исполняемым, а файл *results.txt* — исполняемый.

Чтобы проверить, установлены ли для файла *results.txt* права записи и выполнения, примените следующую команду:

```
$ [ -w results.txt -a -x results.txt ]
$ echo $?
0
```

В данном случае получается утвердительный ответ.

17.3. Проверка строк

Проверка строк является важным этапом при отслеживании ошибок. Значение этого этапа повышается, если проверяются вводимые пользователями данные либо выполняется сравнение переменных. Чтобы проверить строки, достаточно выбрать один из пяти форматов.

```
test "строка"
test оператор_строки "строка"
test "строка" оператор_строки "строка"
[ оператор_строки строка ]
[ строка оператор_строки строка ]
```

Здесь в качестве выражения `оператор_строки` могут использоваться следующие операторы:

`=`	Две строки равны
`!=`	Две строки не равны
`-z`	Эта строка нулевая
`-n`	Эта строка не является нулевой

Чтобы проверить, присвоено ли переменной среды `EDITOR` нулевое значение, примените команду:

```
$ [ -z $EDITOR ]
$ echo $?
1
```

Ответ отрицательный. Присвоено ли этой переменной значение *vi*?

```
$ [ $EDITOR = "vi" ]
$ echo $?
0
```

Ответ утвердительный. Отобразим это значение на экране:

```
$ echo $EDITOR
vi
```

Проверить, равны ли значения переменных TAPE и TAPE2, можно, воспользовавшись следующей командой:

```
$ TAPE="/dev/rmt0"
$ TAPE2="/dev/rmt1"
$ [ "$TAPE" = "$TAPE2" ]
$ echo $?
1
```

Ответ отрицательный. При присваивании переменных необязательно применять двойные кавычки. Но при сравнении строк использование двойных кавычек является непременным условием.

Чтобы проверить "неравенство" переменных TAPE и TAPE2, примените следующую команду:

```
$ [ "$TAPE" != "$TAPE2" ]
$ echo $?
0
```

Значения этих переменных не равны.

17.4. Проверка чисел

Для сравнения чисел можно воспользоваться операторами другого рода. Общий формат:

```
"число" числовой_оператор "число"
```

или

```
[ "число" числовой_оператор "число" ]
```

где в качестве выражения числовой_оператор могут фигурировать следующие операторы:

-eq	Два числа равны
-ne	Два числа не равны
-gt	Первое число больше второго числа
-lt	Первое число меньше второго числа
-le	Первое число меньше или равно второму числу
-gt	Первое число больше или равно второму числу

Выясним, равно ли одно число другому (в данном случае рассматривается, равно ли число 130 числу 130):

```
$ NUMBER=130
$ [ "$NUMBER" -eq "130" ]
$ echo $?
0
```

Превосходно! Результатом сравнения явилось значение "истина".

Теперь изменим второе число и проверим, отобразится ли сообщение об ошибке. Возвращается значение 1 (130 не равно 100).

```
$ [ "$NUMBER" -eq "100" ]
$ echo $?
1
```

Чтобы проверить, является ли число 130 больше числа 100, воспользуйтесь следующей командой:

```
$ [ "$NUMBER" -gt "100" ]
$ echo $?
0
```

Ответ утвердительный.

Так же можно проверять два целых значения переменных. Ниже уточняется, будет ли значение переменной DEST_COUNT большим, чем значение переменной SOURCE_COUNT.

```
$ SOURCE_COUNT=13
$ DEST_COUNT=15
$ [ "$DEST_COUNT" -gt "$SOURCE_COUNT" ]
$ echo $?
0
```

Необязательно для выполнения проверки обращаться к переменной: можно сравнивать и числа, но в этом случае следует применять кавычки:

```
$ [ "990" -le "995" ]
$ echo $?
0
```

Можно также комбинировать и тестировать выражения с помощью логических операторов. При этом следует пользоваться только одной парой квадратных скобок — не применяйте две пары скобок. Если не учитывать этого замечания, отобразится сообщение об ошибке — "too many arguments" (слишком много аргументов):

```
$ [ "990" -le "995" ] -a [ "123" -gt "33" ]
sh:[: too many arguments
```

В следующем примере проверяются два выражения. Если оба выражения истинны, результат будет истинным. Ниже приводится корректный метод выполнения проверки.

```
$ [ "990" -le "995" -a "123" -gt "33" ]
$ echo $?
0
```

17.5. Применение команды expr

Команда `expr` используется в основном для проверки целочисленных значений, но может применяться также и при обработке строк. Общий формат команды `expr`:

```
expr аргумент оператор аргумент
```

Команду `expr` удобно использовать для подсчета количества строк:

```
$ expr 10 + 10
20

$ expr 900 + 600
1500

$ expr 30/3
10

$ expr 30/3/2
5
```

В случае применения знака умножения, можно с помощью обратной косой черты отменить его значение. Если этого не сделать, интерпретатор команд может неверно определить значение символа `*`:

```
$ expr 30 \* 3
90
```

17.5.1. Приращение переменной цикла

Команда `expr` выполняет приращение переменной цикла. Сначала переменной цикла присваивается начальное значение нуль. Затем добавляется единица. Кавычки применяются для обозначения подстановки команд. Выводимые данные, полученные с помощью команды `expr`, присваиваются переменной цикла.

```
$ LOOP=0
$ LOOP=`expr $LOOP + 1`
```

17.5.2. Проверка численных значений

Команду `expr` можно применять для выполнения сравнений чисел. Если вычисления выполняются с числами, отличными от целых, отображается сообщение об ошибке, например:

```
$ expr rr + 1
expr: нечисловой аргумент
```

Итак, необходимо передать значение переменной (не важно, какой именно), выполнить любую арифметическую операцию и направить выводимые данные в */dev/null*. Затем достаточно проверить код завершения последней команды. Если код равен нулю, тогда мы имеем дело с числом; любое другое значение свидетельствует о том, что данное значение не является числом.

```
$ VALUE=12
$ expr $VALUE + 10 > /dev/null 2>&1
```

```
$ echo $?
0
```

Это — число.

```
$ VALUE=hello
$ expr $VALUE + 10 > /dev/null 2>&1
$ echo $?
2
```

А это — не численное значение.

Команда `expr` также возвращает свой собственный код завершения. К сожалению, значение этого кода противоположно значению кода завершения последней команды. Команда `expr` отображает единицу, если результат проверки истинен, и любое другое значение, если допущена какая-либо неточность или ошибка. Ниже приводится пример проверки двух строк на предмет равенства. В ходе проверки требуется выяснить, является ли строка "hello" равной строке "hello".

```
$ VALUE=hello
$ expr $VALUE = "hello"
1
$ echo $?
0
```

Команда `expr` возвращает единицу. Не беспокойтесь, все идет хорошо. Теперь выполним проверку с помощью кода завершения последней команды: возвращается значение нуль. В ходе проверки выяснилось, что строка "hello" действительно равна строке "hello".

17.5.3. Поиск по шаблону

С помощью команды `expr` можно реализовать поиск по шаблону. Подсчет количества символов строки возможен с помощью команды `expr`. При этом нужно дополнительно указать опцию после двоеточия. Комбинация `'.*'` означает, что в кавычках может указываться любое количество произвольных символов.

```
$ VALUE=accounts.doc
$ expr $VALUE : October 8, ' . * '
12
```

Команду `expr` можно также использовать при поиске совпадающих строк; ниже показано, как применяется шаблон ".doc" для извлечения оставшейся части имени файла.

```
$ expr $VALUE : '\(.*\).doc'
accounts
```

17.6. Заключение

В настоящей главе рассматриваются основные возможности команд `test` и `expr`. Показано, как проверяются права доступа к файлу, как тестируются строки. С помощью других условных операторов, типа "if then else" и "case", можно реализовать всестороннюю проверку. Подобный подход позволяет предпринимать определенные действия по результатам проверки.

ГЛАВА 18

Управляющие конструкции

Все функциональные сценарии должны предлагать возможности по выбору возможных вариантов. При определенных условиях сценарии должны выполнять обработку списков. Этим вопросам посвящена настоящая глава. Кроме того, в ней описывается создание и реализация управляющих конструкций в сценариях. Вот перечень тем, обсуждаемых в данной главе:

- коды завершения;
- циклы `while`, `for` и `until`;
- операторы `if then else`;
- принятие решений в сценариях;
- создание меню.

18.1. Коды завершения

Перед тем как рассмотреть некоторые примеры удачных сценариев, посвятим несколько слов кодам завершения. Каждая выполняемая команда возвращает код завершения. Чтобы просмотреть его, достаточно воспользоваться кодом завершения последней команды:

```
$ echo $?
```

Существует четыре основных типа кодов завершения. Два из них уже упоминались, а именно: код завершения последней команды `$?` и команды, изменяющие ход выполнения сценария (`&&`, `||`). Две оставшиеся разновидности команд имеют отношение к завершению shell-сценария или интерпретатора shell, а также связаны с кодом завершения или кодами возврата функции. Эти коды рассматриваются в главе 19, посвященной изучению функций.

Для завершения текущего процесса в интерпретаторе shell используется команда `exit`. Общий формат команды:

```
exit n
```

где `n` — числовое значение.

Чтобы завершить работу с интерпретатором shell, не создавая во время текущего сеанса другой интерпретатор shell, достаточно в командной строке ввести команду `exit`. Если указать команду `exit` без параметров, интерпретатор shell будет отображать (и отображает) значение последней команды. Существует большое количество

значений кода выхода. Но чаще всего при работе со сценариями и общими системными командами используются два значения кода выхода:

Код завершения 0	Успешное завершение, ошибок нет
Код завершения 1	Неудачное завершение, имеется ошибка

В shell-сценариях можно реализовать пользовательские коды завершения для выхода из сценария. Желательно, чтобы пользователь чаще применял эту возможность. Какой-либо иной shell-сценарий или функция, возвращающая результат, могут использовать код завершения вашего сценария. Кроме того, завершение сценария с генерацией кода завершения свидетельствует о культуре программирования. Выполнение сценария обычно завершается после того, как пользователь неправильно вводит данные, если не удается обнаружить ошибку, а также после окончания обычного процесса обработки.

Примечание:

Начиная с этого места книги, все сценарии включают строки комментария. Строки комментария содержат разъяснения, которые помогают запомнить или ясно представить выполнение сценариев. Поскольку интерпретатор команд игнорирует строки комментария, можно располагать их по своему усмотрению. Строка комментария должна начинаться символом #.

18.2. Управляющие конструкции

Почти все сценарии, за редким исключением, обладают свойством самоуправляемости. В чем состоит управление ходом выполнения сценария? Предположим, что в состав сценария включено несколько команд:

```
#!/bin/sh
# создание каталога
mkdir /home/dave/mydocs
# копирование всех файлов с расширением doc
cp *.docs /home/dave/docs
# удаление всех файлов с расширением doc
rm *.docs
```

Рассматриваемый сценарий выполняет определенные задачи. Каковы же могут быть причины возможных неприятностей? Проблема возникнет, например, в том случае, если нельзя будет создать данный каталог. Как поступить в иной ситуации, если каталог может быть создан, но при копировании файлов появляется сообщение об ошибке? Что произойдет, если применить команду cp к разным файлам из различных каталогов. Продуманные решения нужно принимать до применения команды, а еще лучше, если они реализуются при получении результатов выполнения последней команды. Интерпретатор shell приходит здесь на помощь, поддерживая наборы командных операторов, которые помогают принять верное решение в зависимости от успеха или неудачи при выполнении команды либо при обработке списка. Существует два вида таких командных операторов:

- операторы цикла;
- операторы, изменяющие ход выполнения сценария.

18.2.1. Операторы, изменяющие ход выполнения сценария

Операторы `if`, `then`, `else` позволяют реализовать условное тестирование. Проверить условия можно самыми различными способами. Например, может производиться оценка размера файла, проверка установленных прав доступа к файлу, сравнение каких-либо числовых значений или строк. В результате выполнения сравнений возвращается значение "истина" (0) либо "ложь" (1), и затем предпринимаются действия на основании полученного результата. Перед тем как мы приступим к обсуждению условного тестирования, стоит отметить, что некоторые понятия из этой области были рассмотрены ранее.

Операторы `case` дают возможность выполнять поиск по шаблонам, словам или числовым значениям. После нахождения нужного шаблона или слова можно переходить к выполнению других операторов, которые основываются исключительно на условии, по которому устанавливалось соответствие.

18.2.2. Циклические операторы

Цикл, или итерация, — это процесс повторного выполнения наборов команд. В распоряжении пользователя имеется три вида операторов цикла:

`for loop`	Последовательная обработка значений до тех пор, пока не встретится окончание списка
`until loop`	Используется реже всего. Оператор определяет непрерывное выполнение цикла, пока условие не станет истинным. Проверка условия выполняется в конце цикла
`while loop`	Задает выполнение цикла до тех пор, пока не будет встречено заданное условие. Проверка условия выполняется в начале цикла

Каждый цикл, содержащий операторы управления ходом выполнения сценария, может быть вложен. Например, внутри цикла `for loop` может размещаться другой цикл `for loop`.

После ознакомления с циклами и процессом управления ходом выполнения сценария рассмотрим некоторые сценарии.

С этого момента все операторы `echo` в сценариях имеют отношение к Linux или UNIX BSD. То есть при выводе на экран используется метод "echo -e -n", при котором не выполняется создание новой строки в конце вывода. В главе 19 будет показано, как можно воспользоваться универсальной командой `echo`, которая выполняется для обеих разновидностей системы UNIX (System V и BSD).

18.3. Операторы if then else

Оператор `if` позволяет осуществить проверку условий. Проверка выполняется на основе значений "истина" (0) или "ложь" (1), после чего могут вызываться наборы операторов. Конструкция оператора `if` идеально подходит для проверки ошибок. Этот оператор имеет следующий формат:

```
if условие1
then
команды1
elif условие2
then
```

```
команды2
else
команды3
fi
```

Рассмотрим подробно, какие действия выполняются при вызове оператора `if`.

if условие1	если условие1 истинно
then	тогда
команды1	выполняйте команды1
elif условие2	если условие1 ложно
then	тогда
команды2	выполняйте команды2
else	если условие1 или условие2 не выполняется
команды3	тогда выполняйте команды3
fi	конец

Оператор `if` обязательно завершается ключевым словом `fi`. Довольно распространенной ошибкой является пропуск слова `fi` при закрытии оператора `if`. Следует отметить, что подобную ошибку могут допускать даже опытные программисты.

Ключевые слова `elif` и `else` использовать необязательно. Если оператор не содержит ключевое слово `elif`, то можно не указывать и `else`. Оператор `if` может также включать несколько блоков, начинающихся ключевым словом `elif`. Основной конструкцией оператора `if` является конструкция `if then fi`.

А теперь рассмотрим несколько примеров.

18.3.1. Простые операторы if

Базовая структура оператора `if` выглядит следующим образом:

```
if условие then
    команды
fi
```

При использовании оператора `if` команды ветви `then` следует указывать в новой строке; если это правило нарушается, отображается сообщение об ошибке. По поводу применения разделителя команд нет единого мнения. Ниже указан разделитель, который будет применяться далее в книге. Простой оператор `if` в этом случае приобретает вид:

```
if условие; then
    команды
fi
```

Обратите внимание на отступы. Делать их необязательно, но использование отступов значительно облегчает просмотр сценария и понимание хода проверки условий и выполнения операций.

В следующем примере тестовый оператор используется для проверки того, меньше ли число "10" числа "12". Конечно, это условие истинно, и поэтому выполняются операторы, следующие за частью `then`; в данном случае, на экран просто выводится соответствующее утверждение. Если условие ложно, сценарий завершается, поскольку этот оператор не содержит части `else`.

```
$ pg iftest
#!/bin/sh
# -iftest
# это строка комментария, все строки комментария начинаются символом #
if [ "10" -lt "12" ]
then
  # да, 10 меньше 12
echo "Yes, 10 is less than 12"
fi
```

18.3.2. Проверка значений переменных

Чтобы узнать, задал ли пользователь информацию, можно проверить переменную, которая используется для просмотра вводных данных. Ниже приведены результаты проверки того, присвоены ли какие-либо данные переменной NAME после нажатия пользователем клавиши ввода.

```
$ pg iftest2
#!/bin/sh
# если test2
echo -n "Enter your name :"
read NAME
# правильно ли пользователь ввел данные ????
if [ "$NAME" = " " ] ; then
echo "You did not enter any information"
fi
```

```
$ iftest2
Enter your name :
You did not enter any information
```

18.3.3. Проверка вывода команды grep

Кроме проверки значений переменных или чисел вы можете, к примеру, определить, успешно ли выполнена системная команда.

Чтобы выяснить, была ли выполнена команда `grep`, можно применить оператор `if`. В приведенном ниже примере команда `grep` используется для уточнения, содержится ли в файле *data.file* слово "Dave". Обратите внимание на то, что при поиске соответствия используется шаблон "*Dave*\>".

```
$ pg grepif
#!/bin/sh
# grepif
if grep 'Dave\>' data.file > /dev/null 2>&1
then
echo "Great Dave is in the file" else
echo "No Dave is not in the file"
fi
```

```
$ grepif
No Dave is not in the file
```

В примере вывод команды `grep` направляется в системную корзину. Если поиск значения, совпадающего с шаблоном, завершился удачно, команда `grep` возвращает

значение 0. В этом случае происходит естественная интеграция с оператором `if`; если команда `grep` успешно завершилась, часть `if` принимает значение "истина".

18.3.4. Проверка вывода команды grep с помощью переменной

Как уже упоминалось, команду `grep` можно применять в строке. В следующем сценарии пользователь вводит список имен; затем команда `grep` ищет переменную, которой присвоено имя некого лица (`Peter`).

```
$ pg grepstr
#!/bin/sh
# grepstr
echo -n "Enter a list of names:"
read list
if echo $list | grep "Peter" > /dev/null 2>&1
then
echo "Peter is here"
# можно ли выполнить обработку здесь
else
echo "Peter's not in the list. No comment!"
fi
```

Ниже приводятся результаты вывода, где содержится несколько имен.

```
$ grepstr
Enter a list of names:John Louise Peter James
Peter is here
```

18.3.5. Проверка результата копирования файла

А теперь осуществим проверку того, успешно ли прошло копирование файла. Если команда `cp` не скопировала файл *myfile* в файл *myfile.bak*, отображается сообщение об ошибке. Обратите внимание, что в сообщении об ошибке фигурирует команда `` `basename $0` ``, которая выводит на экран название сценария.

Если все сценарии завершаются ошибкой, желательно, чтобы наряду с указанием на стандартную ошибку отображалось и название сценария. В конце концов, пользователю важно знать название сценария, выполнение которого привело к появлению ошибки.

```
$ pg ifcp
#t/bin/sh
# ifcp
if cp myfile myfile.bak; then
  echo "good copy"
else
   echo "`basename $0`: error could not copy the files" >&2
fi

$ ifcp
cp: myfile: No such file or directory
ifcp: error could not copy the files
```

Обратите внимание на то, что в данном случае файл не может быть найден, и генерируется системное сообщение об ошибке. Ошибки такого типа могут негативно

отразиться на выводимых данных; сценарий уже отображает сообщения об ошибках, поэтому известно, что он функционирует неверно. Зачем же нам повторное уведомление? Чтобы избавиться от ошибок, генерируемых системой, и системных данных вывода, достаточно применить перенаправление стандартного потока ошибок и потока вывода. Для этого немного перепишем сценарий: `> /dev/null 2>&1`. В этом случае получим следующее:

```
$ pg ifcp
#!/bin/sh
# ifcp
if cp myfile myfile.bak >/dev/null 2> then
  echo "good copy"
else
  echo "`basename $0`: error could not copy the files" >&2
fi
```

При выполнении сценария все выводимые данные, включая ошибки, направляются в системную корзину.

```
$ ifcp
ifcp: error could not copy the files
```

18.3.6. Проверка текущего каталога

Некоторые сценарии, реализующие административные задачи, можно выполнять из корневого каталога. Если производится глобальное перемещение файлов или же изменяются права доступа к файлу, несложный тест позволяет уточнить, вовлекается ли в этот процесс корневой каталог. В следующем сценарии для хранения текущего каталога переменная `DIRECTORY` использует подстановку команд. Затем значение этой переменой сравнивается со строкой, содержащей значение "/" (которое и соответствует корневому каталогу). Если значение переменной `DIRECTORY` не идентично этой строке, пользователь завершает работу со сценарием. В этом случае код завершения будет 1, что свидетельствует о наличии ошибки.

```
$ pg ifpwd
#!/bin/sh
# ifpwd DIRECTORY=`pwd`
# захват текущего каталога

if [ "$DIRECTORY" != "/" ]; then
# это корневой каталог ?
# нет, перенаправление вывода в стандартный поток ошибок, который
# отображается на экране по умолчанию.
  echo "You need to be in the root directory not $DIRECTORY to run this
    script" >&2
# выход из сценария со значением 1, ошибка
exit 1
fi
```

18.3.7. Проверка прав доступа к файлу

Вы можете также осуществлять контроль прав доступа к файлу. Ниже приводится несложная проверка на предмет того, можно ли вести записи в файле `test.txt`, который переприсвоен переменной `LOGFILE`.

```
$ pg ifwr
#!/bin/sh
# ifwr
LOGFILE=test.txt
echo $LOGFILE
if [ ! -w "$LOGFILE" ]; then
   echo " You cannot write to $LOGFILE " >&2
fi
```

18.3.8. Проверка параметров, передаваемых сценарию

Оператор `if` может применяться при определении числа параметров, которые передаются сценарию. Чтобы проверить, соответствует ли количество необходимых параметров количеству вызываемых параметров, используется специальная переменная `$#`, содержащая число вызываемых параметров.

В приведенном ниже примере проверяется наличие трех параметров; если они отсутствуют, на экран выводится сообщение из стандартного потока ошибок. Затем сценарий завершается, отображая статус ошибки. Если же количество параметров равно трем, все аргументы выводятся на экран.

```
$ pg ifparam
#!/bin/sh
# ifparam
if [ $# -lt 3 ]; then
# вызывается меньше, чем 3 параметра, на экран выводится сообщение, затем
# прерывается выполнение сценария
  echo "Usage: `basename $0`arg1 arg2 arg3" >&2
  exit 1
fi
# хорошо, получено 3 параметра, отображаются на экране
echo "arg1: $1"
echo "arg2: $2"
echo "arg3: $3"
```

Если передается только два параметра, на экран выводится соответствующее сообщение, и сценарий прекращает выполняться:

```
$ ifparam cup medal
Usage:ifparam arg1 arg2 arg3
```

При передаче трех параметров происходит следующее:

```
$ ifparam cup medal trophy
arg1: cup
arg2: medal
arg3: trophy
```

18.3.9. Определение интерактивного режима выполнения сценария

Иногда требуется выяснить, выполняется сценарий в интерактивном режиме (режим терминала) либо не в интерактивном режиме (команды `cron` или `at`). Такая информация необходима для того, чтобы сценарий мог определить, где можно получить вводимые данные и куда направлять выводимые данные. Чтобы уточнить режим выполнения сценария, достаточно воспользоваться командой `test` с опцией `-t`. Если возвращается значение "истина", сценарий выполняется в интерактивном режиме.

```
$ pg ifinteractive
#!/bin/sh
# ifinteractive
if [ -t ]; then
   echo "We are interactive with a terminal"
else
   echo "We must be running from some background process probably
cron or at "
fi
```

18.3.10. Простые операторы if else

Следующая форма оператора `if` применяется чаще всего:

```
if условие
then
команды1
else
команды2
fi
```

Если условие не удовлетворяет тестированию, часть `else` оператора `if` позволяет перейти к соответствующей операции.

18.3.11. Проверка установок переменных

Ниже проверяется установка переменной среды `EDITOR`. Если переменной `EDITOR` не присвоено значение, пользователь информируется о том, что переменная `EDITOR` не установлена. Если переменная `EDITOR` установлена, тип редактора отображается на экране:

```
$ pg ifeditor
#!/bin/sh
# ifeditor
if [ -z $EDITOR ]; then
# переменная не установлена
echo "Your EDITOR environment is not set"
else
# посмотрим, что же это
echo "Using $EDITOR as the default editor"
fi
```

18.3.12. Проверка пользователя, выполняющего сценарий

В следующем примере для проверки условия используется переменная среды. Здесь проверяется, присвоено ли переменной LOGNAME значение "root". Обычно этот тип оператора добавляется в начале сценариев в качестве дополнительной меры безопасности. Несомненно, переменная LOGNAME может проверяться для каждого действительного пользователя.

Если значение переменной не равно строке "root", на экран выводится сообщение из стандартного потока ошибок. Пользователь информируется о том, что он не является пользователем root, а сценарий завершается со значением ошибки, равным 1.

Если строка "root" равна значению переменной LOGNAME, выполняется оператор, который находится после else.

На практике в этом случае сценарий продолжает обработку задания в обычном режиме. Соответствующие операторы находятся после конструкции fi, поскольку все пользователи, отличные от пользователя root, лишаются доступа к сценарию еще на первом этапе проверки.

```
$ pg ifroot
#!/bin/sh
# ifroot
if [ "$LOGNAME" != "root" ]
# если пользователь не является пользователем root
   echo "You need to be root to run this script" >&2
   exit 1
else
# да, это пользователь root
  echo "Yes indeed you are $LOGNAME proceed"
fi
# выполнение операторов в обычном режиме
```

18.3.13. Передача параметров сценария системной команде

Позиционные параметры можно передать сценарию, а затем проверить значение переменной. Если при этом пользователь указывает после названия сценария наименование каталога, сценарий заново присваивает специальному параметру $1 более содержательное название, в данном случае DIRECTORY. С помощью команды ls -A проверяется, не является ли каталог пустым. Если каталог пуст, эта команда не возвращает данные. Затем отображается соответствующее сообщение.

```
$ pg ifdirec
#!/bin/sh
# ifdirec
# присваивание $1 переменной DIRECTORY
DIRECTORY=$1
if [ "`ls -A $DIRECTORY`" = "" ] ; then
# если строка пуста, каталог пуст
   echo "$DIRECTORY is indeed empty"
else
# в противном случае, нет
   echo "$DIRECTORY is not empty"
fi
```

Вместо указанного метода можно воспользоваться другой возможностью. Получим аналогичный результат.

```
$ pg ifdirec2

#!/bin/sh
# ifdirec2
DIRECTORY=$1
if [ -z "`ls -A $DIRECTORY`" ]
then

   echo "$DIRECTORY is indeed empty"
else
   echo "$DIRECTORY -is not empty"
fi
```

18.3.14. Применение команды null

Обычно при проведении проверки условий выполняются части then и else. Иногда независимо от того, истинно или ложно условие, нет необходимости переходить к действиям.

К сожалению, нельзя оставлять незаполненными части оператора if — здесь должен находиться какой-либо оператор. Чтобы разрешить это затруднение, интерпретатор shell поддерживает команду null ':'. Команда null всегда возвращает значение "истина", что в данном случае нас удовлетворяет. Возвращаясь к предыдущему примеру, заметим, что если каталог пуст, команды можно размещать только в части then.

```
$ pg ifdirectory
#!/bin/sh
# ifdirectory
DIRECTORY=$1
if [ "'ls -A $DIRECTORY'" = "" ]
then
echo "$DIRECTORY is indeed empty"
else : # не выполняет ничего
fi
```

18.3.15. Проверка на предмет создания каталога

В продолжение темы каталогов рассмотрим следующий сценарий. Сценарий получает параметр и пытается создать каталог при помощи значения этого параметра. Параметр передается на командную строку и заново присваивается переменной под названием DIRECTORY. В данном случае проверяется, является ли переменная нулем.

```
if [ "$DIRECTORY" = "" ]
```

Вместо предложенного варианта можно воспользоваться общим случаем проверки параметров:

```
if [ S# -lt 1 ]
```

Если строка нулевая, отображается соответствующее сообщение и сценарий завершается. Если каталог уже имеется, никаких дополнительных действий не предпринимается и сценарий выполняется далее.

Пользователь получает запрос, действительно ли нужно создавать каталог. Если он вводит символ, отличный от Y или y, выполняется команда null, в результате чего не предпринимается никаких действий. Каталог создан.

Чтобы проверить, создан ли каталог, применяется код завершения последней команды. Если каталог не был создан, отображается соответствующее сообщение.

```
$ pg ifmkdir
!/bin/sh
# "ifmkdir
# параметр передается как $1, но заново присваивается переменной DIRECTORY
DIRECTORY=$1
# является ли строка пустой ??
if [ "$DIRECTORY" = "" ]
then
   echo "Usage :`basename $0` directory to create" >&2
   exit 1
fi
if [ -d $DIRECTORY ]
then : # ничего не выполняет
else
   echo "The directory does exist"
   echo -n "Create it now? [y..n] :"
   read ANS
   if [ "$ANS" = "y" ] || [ "$ANS" = "Y" ]
then
   echo "creating now"

   # создайте каталог и перешлите все данные вывода в /dev/null
   mkdir $DIRECTORY >/dev/null 2>&1
   if [ $? != 0 ]; then
     echo "Errors creating the directory $DIRECTORY" >&2
     exit 1
   fi
else : # ничего не выполняет
fi
```

При выполнении указанного сценария получим следующее:

```
$ ifmkdir dt
The directory does exist
Create it now? [y..n]: y
creating now
```

18.3.16. Другие возможности копирования

С помощью команды cp сценарию передается два параметра (они должны содержать имена файлов). Затем системная команда cp копирует значение параметра $1 в параметр $2, а поток вывода перенаправляется в */dev/null*. Если команда выполнилась успешно, никаких действий не предпринимается, т.е. применяется команда null.

С другой стороны, если в процессе выполнения команды произошел сбой, об этом следует узнать до завершения сценария.

```
$ pg ifcp2
# !/bin/sh
# ifcp2
#if cp $1 $2 > /dev/null 2>&1
# успешно, ничего делать не надо
then :
else
# плохо, покажем пользователю, какие файлы здесь были.
   echo "`basename $0`: ERROR failed to copy $1 to $2"
   exit 1
fi
```

Выполнение сценария при отсутствии ошибки, связанной с командой cp:

```
$ cp2 myfile.lex myfile.lex.bak
```

Сценарий выполняется при наличии ошибок в команде cp:

```
$ ifcp2 myfile.lexx myfile.lex.bak
ifcp2: ERROR failed to copy myfile.lexx myfile.lex.bak
```

В следующем примере для сортировки файла под именем *accounts.qtr* применяется команда sort.

Результаты вывода направляются в системную корзину. Но кому интересно видеть на экране 300 отсортированных строк? Если сортировка прошла успешно, не нужно предпринимать никаких действий; если при выполнении команды имелись сбои, следует сообщить об этом пользователю.

```
$ pg ifsort
#!/bin/sh
# ifsort
if sort accounts.qtr > /dev/null
# отсортировано. Прекрасно
then :
else
# лучше сообщим об этом пользователю
  echo "`basename $0`: Oops..errors could not sort accounts.qtr"
fi
```

18.3.17. Применение нескольких операторов if

Операторы if можно вкладывать; при этом нужно следить, чтобы каждому ключевому слову if соответствовало слово fi.

18.3.18. Проверка и установка переменных среды

В предыдущих разделах к переменной среды EDITOR обращались, чтобы узнать, установлена ли она. Сделаем шаг вперед и уточним, не была ли переменная среды установлена ранее. Ниже приводится соответствующий сценарий.

```
$ pg ifseted
#!/bin/sh
```

```
# ifseted
# установлена ли переменная EDITOR ?

if [ -z $EDITOR ] ; then

   echo "Your EDITOR environment is not set"
   echo "I will assume you want to use vi..OK"
   echo -n "Do you wish to change it now? [y..n] :"
   read ANS

# проверка верхнего или нижнего регистра для 'y'
if [ "$ANS" = "y" ] || [ "$ANS" = "Y" ]; then
   echo "enter your editor type :"
   read EDITOR
   if [ -z $EDITOR ] || [ "$EDITOR" = "" ]; then
   # если переменная EDITOR не установлена, ей не присвоено  значение,
   # тогда присвоим ей значение vi
   echo "No, editor entered, using vi as default"
   EDITOR=vi
   export EDITOR
 fi

   # берется значение и присваивается переменной EDITOR
   EDITOR=$EDITOR
   export EDITOR
   echo "setting $EDITOR"
 fi
else
# пользователь
   echo "Using vi as the default editor"
   EDITOR=vi
   export vi
fi
```

Рассмотрим, как работает приведенный сценарий. Сначала проверим, установлена ли эта переменная. Если это так, появляется сообщение, что редактор `vi` применяется как редактор, заданный по умолчанию. Затем `vi` устанавливается в качестве редактора, и сценарий завершается.

Если редактор `vi` не установлен, пользователю поступает запрос, следует ли установить это значение. Проверка выполняется независимо от регистра, в котором находится символ `y`. Если пользователь не вводит значения, отличные от `y` или `Y`, сценарий завершается.

На данном этапе пользователю предлагается ввести тип редактора. Затем выполняется проверка, не установлен ли данный редактор, а также уточняется, не нажимал ли пользователь во время проверки `$EDITOR =""` клавишу [Return]. Действительно эта проверка реализована лучше, чем `-z $EDITOR`, но оба этих метода приводятся лишь в качестве иллюстрации. Если результаты проверки будут отрицательны, на экран выводится сообщение, что применяется редактор `vi`, причем значение `vi` присваивается переменной `EDITOR`.

Если пользователь вводит имя для переменной `EDITOR`, происходит присвоение и экспорт этого имени.

18.3.19. Проверка кода завершения последней команды

До сих пор каталог создавался путем передачи названия каталога в сценарий. Затем сценарий запрашивал пользователя, создан ли каталог. В следующем примере создается каталог, и все файлы *.txt копируются из текущего каталога в новый каталог. В приведенном сценарии с помощью кода завершения последней команды проверяется успешность выполнения каждой из команд. Если результаты выполнения команд неудовлетворительны, пользователю направляется соответствующее сообщение.

```
$ pg ifmkdir2
#!/bin/sh
# ifmkdir2
DIR_NAME=testdirec
# где мы находимся ?
THERE='pwd'
# перенаправление потока вывода в системную корзину
mkdir $DIR_NAME > /dev/null 2>&1
# каталог ли это ?
if [ -d $DIR_NAME ]; then
# можно ли применить к каталогу команду cd
   cd $DIR_NAME
   if [ $? = 0 ]; then
   # да, можно
     HERE='pwd'
     cp $THERE/*.txt $HERE
 else
   echo "Cannot cd to $DIR_NAME" >&2
   exit 1
 fi
else
   echo "$cannot create directory $DIR_NAME" >&2
   exit 1
fi
```

18.3.20. Добавление и проверка целых значений

В следующем примере рассматривается проверка чисел. Сценарий содержит набор значений счетчика, который легко изменяется при вводе пользователем нового числа. Затем сценарий добавляет это новое значение к постоянному значению, равному 100. Ниже показано, как выполняется подобный сценарий.

Пользователь может изменить значение путем ввода нового значения, или же ничего не менять, нажав клавишу [Return]. Затем текущее значение выводится на экран, и сценарий завершается.

Если пользователь вводит `Y` или `y`, поступает приглашение на ввод нового значения, которое добавляется к счетчику. Если пользователь нажимает клавишу [Return], сценарий с помощью команды `echo` сообщает, что счетчик сохраняет старое значение. Когда пользователь вводит значение, проверка чисел позволяет уточнить, является ли это значение числом. Если это так, значение добавляется к значению `COUNTER` и затем отображается на экране.

```
$ pg ifcounter
#!/bin/sh
# ifcounter
```

```
COUNTER=100
echo "Do you wish to change the counter value currently set at $COUNTER ?
      [y..n] :"
read ANS
if [ "$ANS" = "y" ] || [ "$ANS" = "Y" ]; then
# да, пользователь желает изменить значение
   echo "Enter a sensible value "
   read VALUE
# простой тест для уточнения, является ли значение численным, добавим к
VALUE любое число,
# проверим код
# возврата
expr $VALUE + 10 > /dev/null 2>&1
     STATUS=$?
# проверим код возврата для expr
if [ "$VALUE" = "" ] | | [ "$STATUS" != "0" ]; then
# направим ошибки в стандартный поток ошибок
   echo " You either entered nothing or a non-numeric " >&2
   echo " Sorry now exiting..counter stays at $COUNTER" >&2
   exit 1
fi
# если мы здесь, значит, это - число, добавим его к COUNTER
  COUNTER=`expr $COUNTER + $VALUE`
  echo " Counter now set to $COUNTER"
else
# если мы здесь, значит, пользователь вместо того, чтобы ввести число,
# нажал клавишу ввода
# или ответим n для изменения значения приглашения
   echo " Counter stays at $COUNTER"
fi
```

Рассмотрим результаты выполнения приведенного сценария.

```
$ ifcount
Do you wish to change the counter value currently set at 100 ? [y..n]:n
Counter stays at 100

$ ifcount
Do you wish to change the counter value currently set at 100 ? [y..n]:y
Enter a sensible value
fdg
   You either entered nothing or a non-numeric
   Sorry now exiting..counter stays at 100

$ ifcount
Do you wish to change the counter value currently set at 100 ? [y..n]:y
Enter a sensible value 250
Counter now set to 350
```

18.3.21. Простой сценарий, обеспечивающий безопасность при регистрации

А теперь рассмотрим структурное дополнение, которое усиливает меры безопасности при регистрации пользователя перед запуском приложений. Поступает приглашение на ввод имени пользователя и пароля; если имя пользователя и пароль

совпадают со строками, включенными в сценарий, пользователь регистрируется. Если такого совпадения не наблюдается, пользователь завершает работу.

Сценарий устроен так, что вначале переменные получают значение "ложь". Предполагается, что вводные данные пользователя окажутся ошибочными. Текущие установки команды `stty` сохраняются, поэтому можно скрыть символы, которые вводятся в поле пароля. Затем установки `stty` восстанавливаются.

Если вводится корректный ID пользователя и пароль (паролем является mayday), переменные `INVALID_USER` и `INVALID_PASSWD` для недействительного пользователя или пароля имеют значение "no". Затем производится тестирование, и если какая-либо из переменных принимает значение `yes`, сценарий для этого пользователя завершается по умолчанию.

К работе допускаются пользователи с действительными ID и паролями. В регистрационном сценарии удобно применять описанное структурное дополнение. В данном примере действительными ID пользователя служат *dave* или *pauline*.

```
$ pg ifpass
#!/bin/sh
# ifpass

# установим значения переменных в "ложь"
INVALID_USER=yes
INVALID_PASSWD=yes
# сохранение текущих установок команды stty
SAVEDSTTY=`stty -g`
echo "You are logging into a sensitive area"
echo -n "Enter your ID name :"
read NAME
# скройте символы, введенные в
stty -echo
echo "Enter your password :"
 read PASSWORD
# попробуем снова
stty $SAVEDSTTY
if [ "$NAME" = "dave" ] | | [ "$NAME" = "pauline" ]; then
   # если действительно, установите переменную
   INVALID_USER=no
fi

if [ "$PASSWORD" = "mayday" ]; then
   # если пароль действителен, установите переменную
   INVALID_PASSWD=no
fi

if [ "$INVALID_USER" = "yes" -o "$INVALID_PASSWD" = "yes" ]; then
   echo "`basename $0 :` Sorry wrong password or userid"
   exit 1
fi
# если вы здесь, ваш ID и пароль в порядке.
echo "correct user id and password given"
```

Если при выполнении приведенного сценария указывается недействительный пароль, получим:

```
$ ifpass

You are logging into a sensitive area
Enter your ID name : dave
Enter your password :
ifpass :Sorry wrong password or userid
```

Введем верное имя регистрационное имя и пароль.

```
$ ifpass
You are logging into a sensitive area
Enter your ID name : dave
Enter your password :
correct user id and password given
```

18.3.22. Применение elif

Часть `elif` оператора `if then else` применяется для проверки при наличии более чем двух условий.

18.3.23. Несколько проверок, реализуемых с помощью elif

В следующем несложном примере протестируем введенные в сценарий имена пользователей.

Сначала в сценарии проверяется, действительно ли пользователь ввел имя; если имя не введено, то проверка не выполняется. Если имя введено, с помощью части `elif` проверяется, совпадает ли имя с `root`, `louise` или `dave`. В случае несовпадения имени ни с одним из перечисленных имен на экран выводится сообщение, что пользователь не является пользователем `root`, `louise` или `dave`.

```
$ pg ifelif
#!/bin/sh
# ifelif
echo -n "enter your login name :"
read NAME
# имя не введено, рассмотрение прекращается
if [ -z $NAME ] | | [ "$NAME" = "" ]; then
   echo "You did not enter a name"
elif
   # является ли именем root
   [ "$NAME" = "root" ]; then
   echo "Hello root"
elif
   # именем является louise
   [ $NAME = "louise" ]; then
   echo "Hello louise"
elif
   # именем является dave
   [ "$NAME" = "dave" ]; then
   echo "Hello dave"
else
```

```
   # нет, это какое-то другое имя
   echo "You are not root or louise or dave but hi $NAME"
fi
```

При выполнении приведенного сценария с использованием различных регистрационных имен получим следующее:

```
$ ifelif
enter your login name : dave
Hello dave

$ ifelif
enter your login name :
You did not enter a name

$ ifelif2
enter your login name : peter
You are not root or louise or dave but hi peter
```

18.3.24. Проверка нескольких вариантов размещения файла

Предположим, что к файлу проверки регистрации требуется применить команду `cat`. Файл в зависимости от того, кто из пользователей выполнял инсталляцию, находится либо в каталоге */usr/opts/audit/logs*, либо в каталоге */usr/local/audit/logs*. Перед применением к файлу команды `cat` следует убедиться в том, что его можно просматривать; именно это и будет уточняться при проверке. Если файл нельзя найти или же его нельзя просматривать, на экран выводится сообщение об ошибке. Ниже приводится соответствующий сценарий:

```
$ pg ifcataudit
#!/bin/sh
# ifcataudit

# размещение файла регистрации
LOCAT_1=/usr/opts/audit/logs/audit.log
LOCAT_2=/usr/local/audit/audit.logs

if [ -r $LOCAT_1]; then
   # если файл находится в этом каталоге и может просматриваться,
   # применим к нему команду cat
   echo "Using LOCAT_1"
   cat $LOCAT_1
elif
   # иначе, файл должен находиться в этом каталоге и можно его просматривать
   [ -r $LOCAT_2 ]
then
   echo "Using LOCAT_2"
   cat $LOCAT_2
else
   # нет ни в одном каталоге...
   echo `basename $0`: Sorry the audit file is not
   readable or cannot be located." >&2
   exit 1
fi
```

Если при выполнении указанного сценария установлено, что файл находится в каком-либо из двух каталогов и может просматриваться, то к нему можно применить команду `cat`. В противном случае на экран выводится сообщение об ошибке, и сценарий завершается. Этот пример неудачен, поскольку наш воображаемый файл отсутствует.

```
$ ifcataudit
ifcataudit: Sorry the audit file is not readable or cannot be located.
```

18.4. Оператор case

Оператор `case` является многовариантным оператором. С его помощью можно искать значения, используя заданный шаблон. Если совпадение с шаблоном установлено, можно выполнять команды, основываясь исключительно на этом соответствии. Ниже приводится формат оператора `case`:

```
case значение in
шаблон1)
команды1
...
шаблон2)
команды2
...
;;
esac
```

Рассмотрим, как функционирует оператор `case`. После значения должен находиться предлог "in", а каждый шаблон должен завершаться правой скобкой. В качестве значения может применяться переменная или константа. Когда устанавливается соответствие с шаблоном, для этого шаблона выполняются все команды вплоть до символов `;;`.

Поиск значения, совпадающего с шаблоном, выполняется по каждому шаблону. Если устанавливается соответствие с шаблоном, то оставшиеся шаблоны уже не проверяются. Эти шаблоны не принимаются во внимание даже после выполнения команд, относящихся к шаблону, с которым установлено соответствие. Если поиск соответствия по всем шаблонам оказался безрезультатным, можно принять значение с помощью символа звездочки. Этот символ используется для фиксации любых вводимых данных.

Шаблонная часть может содержать метасимволы. Аналогичным образом соответствие с шаблоном устанавливается при обработке расширений имен файлов в командной строке:

`*`	Произвольные символы
`?`	Произвольный отдельный символ
`[..]`	Произвольный символ из класса или диапазона

А теперь рассмотрим несколько примеров.

18.4.1. Простой оператор case

Следующий сценарий отображает приглашение для ввода чисел от 1 до 5. Число передается оператору `case`, переменной `ANS` присваивается значение `ANS` оператора `case`, и значение `ANS` сравнивается с каждым шаблоном.

Если соответствие установлено, команды из шаблонной части выполняются до тех пор, пока не появятся символы `;;`. Тогда на экран выводится команда, которая информирует пользователя о сделанном выборе. Затем выполнение оператора `case` завершается, поскольку совпадение с шаблоном установлено.

Далее выполняются операции, находящиеся после оператора `case`.

Если соответствие не найдено, с помощью шаблона * выполняется прием всей информации. Затем отображается сообщение об ошибке.

```
$ pg caseselect
#!/bin/sh
# caseselect
echo -n "enter a number from 1 to 5 :"
read ANS
case $ANS in
1) echo "you select 1"
;;
2) echo "you select 2"
;;
3) echo "you select 3"
;;
4) echo "you select 4"
;;
5) echo "you select 5"
;;
*) echo "`basename $0`: This is not between 1 and 5" >&2
exit 1
;;
esac
```

Если этот сценарий выполняется с различными вводимыми данными, получим:

```
$ caseselect
enter a number from 1 to 5 : 4
you select 4
```

С помощью шаблона * выполним прием информации, с которой не установлено соответствия:

```
$ caseselect
enter a number from 1 to 5 :pen
caseselect: This is not between 1 and 5
```

18.4.2. Применение символа | при поиске по шаблону

При использовании оператора `case` в качестве команды `or` можно указывать символ |. Например, `vt100|vt102)` соответствует шаблону `vt100` или `vt102`.

В следующем примере у пользователя запрашивают тип терминала. Если пользователь вводит `vt100` или `vt102`, выполняется сравнение с шаблоном "vt100| vt102)". В данном случае переменной `TERM` присваивается значение `vt100`. Если пользователь

указывает тип терминала, который не соответствует шаблону, с помощью шаблона * выполняется прием этой информации и значение типа терминала все равно устанавливается как vt100. Наконец, за пределами действия оператора case производится экспорт переменной TERM. Независимо от тех сведений, которые вводит пользователь, переменная TERM представляет действительный тип терминала, поскольку используется поиск по шаблону *.

```
$ pg caseterm
#!/bin/sh
# caseterm
echo " choices are.. vt100, vt102, vt220"
echo -n "enter your terminal type :"
read TERMINAL
case $TERMINAL in
vt100|vt102) TERM=vt100
;;
vt220) TERM=vt220
;;
*) echo "`basename $0` : Unknown response" >&2
echo "setting it to vt100 anyway, so there"
TERM=vt100
;;
esac
export TERM
echo "Your terminal is set to $TERM"
```

Если при выполнении сценария указывается некорректный тип терминала, получим следующее:

```
$ caseterm
choices are.. vt100, vt102, vt220
enter your terminal type :vt900
caseterm ; Unknown response
setting it to vt100 anyway, so there
Your terminal is set to vt100
```

Если вводится существующий тип терминала, получим:

```
$ case2
choices are.. vt100, vt102, vt220
enter your terminal type :vt220
Your terminal is set to vt220
```

В любом случае пользователь устанавливает тип терминала.

18.4.3. Приглашение для ввода y или n

Оператор case удобно применять при отображении запроса на продолжение обработки. Ниже приводится сценарий, в котором пользователю предлагается указать в качестве ответа либо 'y', что означает продолжение обработки, либо 'n' — выход из сценария. Если пользователь введет y, Y, yes или Yes, обработка будет продолжена в соответствии с частью сценария после оператора case. Если пользователь введет n, N, no или какой-либо другой ответ, работа со сценарием для него завершается.

```
$ pg caseans
#!/bin/sh
# caseans
echo -n "Do you wish to proceed [y..n] :"
read ANS
   case $ANS in
   y|Y|yes|Yes) echo "yes is selected"
    ;;
   n|N) echo "no is selected"
     exit 0    # нет ошибки, поэтому для выхода укажите 0
    ;;
   *) echo "`basename $0` : Unknown response" >&2
   exit 1
    ;;
   esac
# если мы оказались здесь, были выбраны значения y|Y|yes|Yes.
```

Если при выполнении указанного сценария выбрать некорректный вариант ответа, отображаются следующие данные:

```
$ caseans
Do you wish to proceed [y..n] :df
caseans : Unknown response
```

В случае корректного ответа:.

```
$ caseans
Do you wish to proceed [y..n] :y
yes is selected
```

18.4.4. Оператор case и передача командных параметров

Можно также использовать оператор `case` при передаче параметров в сценарии.

В следующем сценарии при осуществлении проверки используется специальный параметр `$#`, который представляет число передаваемых аргументов. Если это число отлично от 1, сценарий завершает работу, отображая соответствующее сообщение.

Затем оператор `case` выполняет прием следующих параметров: `passwd`, `start`, `stop` или `help`. В дальнейшем код реализуется для каждого совпадения с этими шаблонами. При передаче неизвестного значения на экран выводится стандартное сообщение об ошибке.

```
$ pg caseparam
#!/bin/sh

# caseparam
if [ $#!= 1 ]; then
   echo "Usage:`basename $0`[start|stop|help]">&2
   exit 1
fi
# присвойте переменной OPT параметр
OPT=$1
   case $OPT in
   start) echo "starting..`basename $0`"
    # здесь коды для начала процесса
```

```
    ;;
   stop) echo "stopping..`basename $0`"
    # здесь коды для прекращения процесса
    ;;
   help)
    # здесь находится код для отображения справочной страницы
    ;;
   *) echo "Usage:`basename $0`[start|stop|help]"
     ;;
esac
```

Если сценарию передается неверный параметр, получим следующее:

```
$ caseparam what
Usage:caseparam [start|stop|help]
```

В случае передачи действительного параметра:

```
$ caseparam stop
stopping..caseparam
```

18.4.5. Прием потока ввода без применения шаблонных команд

После шаблонной части необязательно указывать команды; если после шаблонной части команды отсутствуют, до перехода к дальнейшей обработке выполняется фильтрация нежелательных откликов.

Если в отдел учета следует направить отчет об учетной записи, сначала желательно удостовериться, что пользователь правильно ввел номер отдела, а затем уже уточнить, какой отчет выполняется. Такой подход реализуется при сравнении с шаблоном всех требуемых значений; любые другие методы неприемлемы.

В следующем сценарии указано, что если пользователь вводит номер отдела, который не совпадает со значением 234, 453, 655 или 454, то пользователь выйдет из сценария. Если номер отдела указан правильно, аналогичный подход применяется при определении типа отчета. По окончании выполнения оператора `case` остаются действительный номер отдела и правильный тип отчета. Ниже приводится соответствующий сценарий.

```
$ pg casevalid
!/bin/sh
# casevalid
TYPE=""
echo -n "enter the account dept No: :"
read ACC

   case $ACC in
    234);;
    453);;
    655);;
    454);;
   *) echo "`basename $0`: Unknown dept No:" >&2
     echo "try..234,453,655,454"
     exit 1
     ;;
   esac
```

```
# если оказались здесь, получен верный номер отдела
echo " 1 . post"
echo " 2 . prior"
echo -n "enter the type of report: "
read ACC_TYPE
   case $ACC_TYPE in
   1)TYPE=post;;
   2)TYPE=prior;;
   *) echo "`basename $0`: Unknown account type." >&2
    exit 1
    ;;
   esac
# если оказались здесь, значит все указано правильно!

echo "now running report for dept $ACC for the type $TYPE"
# выполняем отчет о команде..
```

Если вводимые данные достоверны, получим:

```
$ casevalid
enter the account dept No: :234
1 . post
2 . prior
enter the type of report:2
now running report for dept 234 for the type prior
```

Если номер отдела введен неверно, получим:

```
$ casevalid
enter the account dept No: :432
casevalid: Unknown dept No:
try..234,453,655,454
```

При вводе неправильного типа отчета, получим:

```
$ casevalid
enter the account dept No: :655
1 . post
2 . prior
enter the type of report:4
casevalid: Unknown account type.
```

18.4.6. Значения переменных, заданные по умолчанию

Если при просмотре значения переменной пользователь нажимает клавишу [Return], сценарий завершает работу не всегда. С помощью проверки определяется, установлена ли данная переменная, и если не установлена, то ей может быть присвоено значение.

В следующем сценарии для выполнения отчета пользователю предлагают ввести название дня недели. Если пользователь нажимает клавишу [Return], используется день недели, заданный по умолчанию, а именно "Saturday". Это название и присваивается переменной WHEN.

Если пользователь вводит название другого дня, с помощью оператора case выполняется проверка, совпадает ли введенное название с названием одного из дней недели, предназначенных для выполнения сценария, а именно "Saturday", "Sunday"

и "Monday". Обратите внимание, что всевозможные аббревиатуры названий дней недели составлены так, чтобы выполнялся перехват "всех возможных" комбинаций этих названий.

Ниже приводится соответствующий сценарий.

```
$ pg caserep
#!/bin/sh
# caserep
echo "      Weekly Report"

echo -n "What day do you want to run report [Saturday] :"

# если нажать клавишу ввода, принимается заданное по умолчанию название
# Saturday
read WHEN
echo "validating..${WHEN:="Saturday"}"
   case $WHEN in
   Monday|MONDAY|mon)
    ;;
   Sunday|SUNDAY|sun)
    ;;
Saturday|SATURDAY|sat)
    ;;
   *) echo " Are you nuts!, this report can only be run on " >&2
   echo " on a Saturday, Sunday or Monday" >&2
   exit 1
    ;;
esac
echo "Report to run on $WHEN"
# здесь команда для выполнения действительного отчета
```

При корректных начальных данных получим:

```
$ caserep
      Weekly Report
What day do you want to run report [Saturday] :
validating..Saturday
Report to run on Saturday
```

Если начальные данные были неправильны, получим:

```
$ caserep
      Weekly Report
What day do you want to run report [Saturday] :Tuesday
validating..Tuesday
Are you nuts!, this report can only be run on
on a Saturday, Sunday or Monday
```

Можно заключить, что оператор `case` функционирует так же, как и несколько операторов `if then else`. Такой вывод вполне правомерен.

18.5. Цикл for

Общий формат цикла:

```
for имя_переменной in list
do
команда1
команда...
done
```

Цикл for однократно обрабатывает всю информацию для каждого значения, включенного в список list. Чтобы получить доступ к каждому значению в списке, достаточно задать параметр имя_переменной. Командой служит любая действительная команда или оператор интерпретатора shell. В качестве параметра имя_переменной можно указать любое слово.

Применение опции in list не является обязательным; если не включать эту часть, цикл воспользуется позиционными параметрами командной строки.

Опция in list может содержать подстановки, строки и имена файлов. Рассмотрим несколько примеров.

18.5.1. Простой цикл for

Этот цикл просто выводит на экран список, который состоит из "1 2 3 4 5". Чтобы получить доступ к каждой переменной, в качестве параметра имя_переменной указывается "loop".

```
$ pg for_i
#!/bin/sh
# for_i
for loop in 1 2 3 4 5
do
   echo $loop
done
```

Приведенный выше сценарий выводит следующие данные:

```
$ for_i
1
2
3
4
5
```

18.5.2. Вывод на экран строки списка

Ниже приводится цикл for, список которого содержит строку значений "orange red blue grey". Для каждой переменной указана команда echo, в качестве параметра имя_переменной указывается 'loop'. Команда echo с помощью части $loop выводит на экран каждое значение списка до тех пор, пока список не окажется пустым.

```
$ pg forlist
#!/bin/sh
# forlist
for loop in "orange red blue grey"
```

```
do
echo $loop
done
```

```
$ forlist
orange red blue grey
```

Также с помощью цикла имя_переменной можно комбинировать строки (в данном случае речь идет о цикле loop).

```
echo "this is the fruit $loop"
```

Результат:

```
This is the fruit orange red blue grey
```

18.5.3. Использование команды ls совместно с циклом for

Этот цикл оценивает команду `ls` интерпретатора shell и отображает сведения о файлах текущего каталога.

```
$ pg forls
#!/bin/sh
# forls
for loop in `ls`
do
   echo $loop
done
```

```
$ forls
array
arrows
center
center1
center2
centerb
```

18.5.4. Применение параметров вместе с циклом for

Если в цикле `for` опустить часть `in list`, позиционные параметры командной строки становятся аргументами. Действительно, этот подход аналогичен следующему:

```
for params in "$@"
```

или

```
for params in "$*"
```

Ниже приводится пример, который показывает, как можно избежать применения конструкции `in list`. Цикл `for` обращается к специальному параметру `$@` или `$*` для получения аргументов из командной строки.

```
$ pg forparam2
#/bin/sh
# forparam2
for params
```

```
do
   echo "You supplied $params as a command line option"
done
   echo $params
done
```

```
$ forparam2 myfile1 myfile2 myfile3
 You supplied myfile1 as a command line option
 You supplied myfile2 as a command line option
 You supplied myfile3 as a command line option
```

Следующий сценарий содержит часть `in "$@"` и образует тот же самый поток вывода, что и предыдущий сценарий.

```
$ pg forparam3
#!/bin/sh
# forparam3
for params "in "$@"
do
   echo "You supplied $params as a command line option"
done
   echo $params
done
```

Если развить этот подход далее и осуществлять поиск набора файлов, то совместно с циклом `for` можно применять команду `find`. При передаче всех файлов используют преимущество параметра командной строки.

```
$ pg forfind
#!/bin/sh
# forfind
for loop
do
   find / -name $loop -print
done
```

Значения передаются с помощью параметра командной строки и образуют часть `-name` команды `find`.

```
$ forfind passwd LPSO.AKSOP
/etc/passwd
/etc/pam.d/passwd
/etc/uucp/passwd
/usr/bin/passwd
/usr/local/accounts/LPSO.AKSOP
```

18.5.5. Посылка сигналов серверам с помощью цикла for

Поскольку цикл `for` может обработать каждое слово списка, установим переменную для отображения названий некоторых серверов сети. Воспользуемся циклом `for` для посылки сигналов каждому из этих серверов.

```
$ pg forping
#!/bin/sh
# forping
```

```
HOSTS="itserv dnssevr acctsmain ladpd ladware"
for loop in $HOSTS
do
   ping -c 2 $loop
done
```

18.5.6. Создание резервных копий файлов с помощью цикла for

Цикл `for` можно использовать для создания резервных копий файлов. При этом переменная просто добавляется к целевому аргументу команды `cp`. Ниже применяется переменная под названием `BAK`. Эта переменная добавляется к каждому имени целевого файла при использовании цикла с помощью команды `cp`. Список включает shell-команду `ls`.

```
$ pg forcp
#!/bin/sh
# forcp
BAK=".bak"
for loop in `ls`
do
   echo "copying $loop to $loop$BAK"
   cp $loop $loop$BAK
done
```

```
$ forcp
copying array to array.bak
copying arrows to arrows.bak
copying center to center.bak
copying center1 to center1.bak
...
```

18.5.7. Массовое преобразование

Чтобы найти все файлы, которые начинаются символами "LPSO", и преобразовать их содержимое в символы верхнего регистра, используются команды `ls` и `cat`. Команда `ls` отображает список файлов, а команда `cat` применяется для передачи списка команде `tr`. Передаваемые файлы получают расширение *.UC*. Обратите внимание, что при использовании в цикле `for` команды `ls` применяются обратные кавычки.

```
$ pg forUC
#!/bin/sh
# forUC
for files in `ls LPSO*`
do
   cat $files |tr "[a-z]" "[A-Z]" >$files.UC
done
```

18.5.8. Удаления, выполняемые с помощью редактора sed

В следующем примере для удаления всех пустых строк применяется потоковый редактор `sed`. Выходной поток данных направляется в новые файлы с расширением *.HOLD*. Затем команда `mv` возвращает файлам их исходные имена.

```
$ pg forsed
#!/bin/sh
# forsed

for files in `ls LPSO*`
do
   sed -e "/^$/d" $files>$files.HOLD
   mv $flies.HOLD $files
done
```

18.5.9. Подсчет с помощью циклов

При обсуждении команды `expr` отмечалось, что эта команда применяется, если в циклы необходимо ввести счетчики. Ниже рассматривается пример, в котором цикл `for` обрабатывает файлы, а вывод и подсчет количества файлов осуществляется с помощью команды `ls`.

```
$ pg forcount
# !/bin/sh
# forcount
counter=0
for files in *
do
   # increment
   counter=`expr $counter + 1`
done
   echo "There are $counter files in 'pwd' we need to process"

$ forcount
There are 45 files in /apps/local we need to process
```

Аналогичный результат можно получить с помощью команды `wc`.

```
$ ls |wc -l
45
```

18.5.10. Циклы for для обработки документов

С циклом `for` можно комбинировать любые команды. В приведенном примере переменная содержит имена всех зарегистрированных пользователей. Для реализации этой конструкции обращаются к команде `who` и утилите `awk`. Затем цикл `for` обрабатывает имена этих пользователей и каждому высылает электронное сообщение. При отправке сообщения используется конструкция "документ здесь".

```
$ pg formailit
#!/bin/sh
# formailit
WHOS_ON=`who -u | awk '{print $1}'`
for user -in $WHOS_ON
do
mail $user << MAYDAY
Dear Colleagues,
It's my birthday today, see you down the
club at 17:30 for a drink.
See ya.
```

```
$LOGNAME
MAYDAY
Done
```

Ниже приводится электронное сообщение для данного сценария.

```
$ pg mbox
Dear Colleagues,
It's my birthday today, see you down the
club at 17:30 for a drink.

See ya.
dave
```

18.5.11. Вложенные циклы for

Чтобы вложить циклы, достаточно цикл `for` поместить в другой цикл:

```
for имя_переменной in list
do
for имя_переменной2 in list2
do
команда1

done
done
```

В следующем сценарии представлен вложенный цикл `for`. Имеются два списка, `APPS` и `SCRIPTS`, первый из которых содержит путь к приложениям на сервере, а второй включает административные сценарии, которые выполняются для каждого приложения. Для каждого приложения из списка `APPS` имеется соответствующее название сценария в списке `SCRIPTS`. Обычно сценарии выполняются в фоновом режиме (с помощью указания префикса `&`). Данный сценарий посредством команды `tee` размещает также запись в журнальном файле, поэтому наряду с файлом на экране отображается поток выходных данных. Обратите внимание на выходные данные, чтобы понять, как цикл `for` использует список `SCRIPTS` при обработке элементов списка `APPS`.

```
$ pg audit_run
#!/bin/sh
# audit_run
APPS="/apps/accts /apps/claims /apps/stock /apps/serv"
SCRIPTS="audit.check report.run cleanup"
LOGFILE=audit.log
MY_DATE='date +%H:%M" on "%d/%m%Y'

# внешний цикл
for loop in $APPS
do
   # внутренний цикл
   for loop2 in $SCRIPTS
   do
     echo "system $loop now running $loop2 at $MY_DATE" | tee -a $LOCFILE
     $loop $loop2 &
   done
done
```

```
$ audit_run

system /apps/accts now running audit.check at 20:33 on 23/051999
system /apps/accts now running report.run at 20:33 on 23/051999
system /apps/accts now running cleanup at 20:33 on 23/051999
system /apps/claims now running audit.check at 20:33 on 23/051999
system /apps/claims now running report.run at 20:33 on 23/051999
system /apps/claims now running cleanup at 20:34 on 23/051999
system /apps/stock now running audit.check at 20:34 on 23/051999
system /apps/stock now running report.run at 20:34 on 23/051999
system /apps/stock now running cleanup at 20:34 on 23/051999
system /apps/serv now running audit.check at 20:34 on 23/051999
system /apps/serv now running report.run at 20:34 on 23/051999
system /apps/serv now running cleanup at 20:34 on 23/051999
```

18.6. Цикл until

Цикл `until` позволяет выполнять ряд команд, пока условие остается истинным. Практически цикл `until` противоположен по смыслу циклу `while`. Цикл `while` является более предпочтительным, но в определенных случаях цикл `until` лучше справляется с работой. Формат цикла `until`:

```
until условие
команда1
...
done
```

В качестве условия выступает любое действительное проверочное условие. Проверка производится в конце цикла, поэтому в любом случае цикл выполняется однократно — не забывайте об этом обстоятельстве. А теперь рассмотрим несколько примеров.

18.6.1. Простой цикл until

Этот сценарий непрерывно ищет пользователя "root" и выполняет команду `who`. Результат выполнения команды `grep` содержится в переменной `IS_ROOT`.

Если пользователь "root" обнаружен, цикл завершает обработку. Затем пользователю `simon` направляется электронное сообщение, в котором отмечается, что пользователь "root" зарегистрирован. Обратите внимание на применение команды `sleep`, которая довольно часто используется в циклах `until`. Эта команда дает возможность приостановить выполнение цикла на несколько секунд, что позволяет указать дополнительные команды.

```
$ pg until_who
#!/bin/sh
# until_who
IS_ROOT=`who | grep root`
until [ "$IS_ROOT" ]
do
  sleep 5
done
echo "Watch it. roots in " | mail simon
```

18.6.2. Контроль наличия файла

В этом примере в цикле `until` выполняется команда "sleep 1" до тех пор, пока не будет удален файл под именем */tmp/monitor.LCK*. После удаления файла сценарий продолжает выполняться в обычном режиме.

```
$ pg until_lck
!/bin/sh
# until_lck
LOCK_FILE=/tmp/process.LCK
until [ ! -f $LOCK_FILE ]

do
   sleep 1
done
echo "file deleted "
# обычная обработка, файл имеется
```

В приведенном примере показан один из методов, обеспечивающий совместную работу сценариев при выполнении.

В данном случае речь идет об одном из методов взаимодействия сценариев. Предположим, что другой сценарий, например `process_main`, используется для сбора информации со всех компьютеров локальной сети и размещения этих данных в файле отчета.

Когда выполняется сценарий `process_main`, создается файл `LCK` (файл блокировки). Приведенный выше сценарий получает информацию, собранную сценарием `process_main`. Однако он не обрабатывает файл, если сценарий `process_main` продолжает обновление файла отчета.

Чтобы устранить эти затруднения, сценарий `process_main` создает файл `LCK` при запуске и удаляет его при завершении работы.

Рассматриваемый сценарий ожидает удаления файла `LCK`. После того как файл `LCK` удален, сценарий может обрабатывать содержимое файла отчета.

18.6.3. Мониторинг дисковой памяти

Цикл `until` также полезно применять для контроля состояния системы. Предположим, что необходимо уточнить пространство, занимаемое файлами на диске, и направить пользователю "root" электронное сообщение по достижении определенного уровня.

В следующем сценарии просматривается система файлов */logs*. С помощью утилиты `awk` и команды `grep` информация постоянно извлекается из переменной `$LOOK_OUT`. Значения этой переменной характеризуют объем, занимаемый системой файлов `/logs`.

Если занятый объем превышает 90%, опция `command` запускает триггер, пользователю "root" направляется сообщение и сценарий завершает работу. Сценарий следует завершить, поскольку в противном случае, если условие истинно (например, занятый объем по-прежнему превышает 90%), электронное сообщение пользователю "root" будет высылаться непрерывно.

```
$ pg until_mon
# !/bin/sh
# until_mon
```

```
# получите столбец со значениями процентов и удалите заголовок строки из df
LOOK_OUT=`df |grep /logs | awk '{print $5}'| sed 's/%//g'`
echo $LOOK_OUT
until [ "$LOOK_OUT" -gt "90" ]
do
   echo "Filesystem..logs is nearly full" | mail root
   exit 0
done
```

18.7. Цикл while

Цикл `while` выполняет ряд команд до тех пор, пока истинно условие. Этот цикл используется также для просмотра данных из файла ввода. Формат цикла `while`:

```
while команда
do
команды1
команды2
...
done
```

Между конструкциями `while` и `do` находится несколько команд, хотя в общем случае применяется только одна команда. Обычно команда выполняет проверку условия.

Команды, размещенные между ключевыми словами `do` и `done`, выполняются только в том случае, если код завершения `command` равен нулю; если код завершения принимает какое-либо другое значение, цикл заканчивается.

Когда команды выполнены, контроль передается обратно, в верхнюю часть цикла. И все начинается снова до тех пор, пока проверяемое условие отлично от нуля.

18.7.1. Простой цикл while

Ниже приводится основная форма цикла `while`. Условие тестирования состоит в том, что если "COUNTER is less than 5", условие останется истинным. Переменная `COUNTER` имеет начальное значение нуль, и ее значение увеличивается на постоянную величину при выполнении цикла.

```
$ pg whilecount
#!/bin/sh
# whilecount
COUNTER=0
# счетчик равен 5 ?
while [ $COUNTER -lt 5 ]
do
# прибавление к счетчику единицы
COUNTER= `expr $COUNTER + 1`
echo $COUNTER
done
```

Указанный сценарий выводит на экран числа от 1 до 5, затем завершает работу.

```
$ whilecount
1
2
3
4
5
```

18.7.2. Применение цикла while при вводе с клавиатуры

Цикл while может применяться для ввода информации с клавиатуры. В следующем примере введенная информация присваивается переменной FILM. Если нажать клавиши [Ctrl+D], цикл завершает выполнение.

```
$ pg whileread
#!/bin/sh
# whileread
echo " type <CTRL-D> to terminate"
echo -n "enter your most liked film :"
while read FILM
do
   echo "Yeah, great film the $FILM"
done
```

Когда сценарий выполняется, вводимыми данными могут быть:

```
$ whileread
enter your most liked film: Sound of Music
Yeah, great film the Sound of Music
<CTRL-D>
```

18.7.3. Применения цикла while для считывания данных из файлов

Обычно цикл while используется для считывания данных из файла, что позволяет сценарию обрабатывать информацию.

Предположим, что следует просмотреть информацию из следующего персонального файла, содержащего имена служащих, их ID и названия отделов.

```
$ pg names.txt
Louise Conrad:Accounts:ACC8987
Peter James:Payroll:PR489
Fred Terms:Customer:CUS012
James Lenod:Accounts:ACC887
Frank Pavely:Payroll:PR489
```

Для хранения строк данных можно использовать переменные. Условие истинно до тех пор, пока не считываются новые данные. Для просмотра содержимого файла цикл while использует перенаправление потока данных ввода. Обратите внимание, что отдельной переменной $LINE присваивается целая строка.

```
$ pg whileread
#!/bin/sh
# whileread
while read LINE
do
   echo $LINE
```

```
done < names.txt

$ whileread
Louise Conrad:Accounts:ACC8987
Peter James:Payroll:PR489
Fred Terms:Customer:CUS012
James Lenod:Accounts:ACC887
Frank Pavely:Payroll:PR489
```

18.7.4. Считывание данных из файлов с помощью IFS

Чтобы при выводе данных устранить разделитель полей в виде двоеточия, примените переменную IFS, предварительно сохранив ее установки. После того как сценарий завершит работу с этими установками, восстановите установки переменной IFS. С помощью переменной IFS можно изменить разделитель полей на двоеточие вместо пробела или символа табуляции, которые заданы по умолчанию. Как известно, отдельной переменной можно присвоить значения трех полей: NAME, DEPT и ID.

Чтобы улучшить внешний вид записей, немного увеличивая поля, можно с помощью команды echo применить символы табуляции. Рассмотрим сценарий:

```
$ pg   whilereadifs
#!/bin/sh
# whilereadifs
# сохраните установку IFS
SAVEDIFS=$IFS
# присвоим переменной IFS новый разделитель
IFS=:
while read NAME DEPT ID
do
   echo -e "$NAME\t $DEPT\t $ID"
done < names.txt
# восстановим установки переменной IFS
IFS=$SAVEDIFS
```

При выполнении сценария получим более привлекательный поток вывода:

```
$ whilereadifs
Louise Conrad       Accounts    ACC8987
Peter James         Payroll     PR489
Fred Terms          Customer    CUS012
James Lenod         Accounts    ACC887
Frank Pavely        Payroll     PR489
```

18.7.5. Обработка файла с помощью проверок условий

Большинство циклов while включает некоторый оператор проверки, который уточняет последовательность действий.

Ниже рассматривается файл с именами служащих, и на экран выводятся подробности. После обнаружения имени служащего "James Lenod" сценарий завершает работу. Вывод на экран подробностей до осуществления проверки позволяет удостовериться, что "James Lenod" добавляется в содержимое файла.

Обратите внимание, что все переменные задаются в начале сценария. При внесении небольших поправок в переменные можно сэкономить рабочее время и

сократить количество типов. Все редакторские правки находятся в начале, а не рассеяны по всему сценарию.

```
$ pg whileread_file
#!/bin/sh
# whileread_file
# инициализация переменных
SAVEDIFS=$IFS
IFS=:
HOLD_FILE=hold_file
NAME_MATCH="James Lenod"
INPUT_FILE=names.txt

# создавайте каждый раз новый HOLD_FILE, в случае, когда сценарий
непрерывно выполняется
>$HOLD FILE
while read NAME DEPT ID
do
  # выводит на экран всю информацию в holdfile с помощью перенаправления
  echo $NAME $DEPT $ID >>$HOLD_FILE
  # имеется ли соответствие ???
  if [ "$NAME" = "$NAME_MATCH" ]; then
    # да, тогда удобно завершить работу
    echo "all entries up to and including $NAME_MATCH are in $HOLD_FILE"
    exit 0
  fi
done < $INPUT_FILE
# восстановление IFS
IFS=$SAVEDIFS
```

Выполним следующий шаг и уточним количество служащих в каждом из отделов. Сохраним прежний формат просмотра, в котором каждому полю присваивается название переменной. Затем для добавления каждого совпадения с помощью оператора `case` просто применим команду `expr`. Если обнаруживается неизвестный отдел, его название выводится на экран в виде стандартного потока ошибок; поэтому, если отдел не существует, нет необходимости прерывать выполнение сценария.

```
$ pg whileread_cond
!/bin/sh
# whileread_cond
# инициализация переменных
ACC_LOOP=0; CUS_LOOP=0; PAY_LOOP=0;

SAVEDIFS=$IFS
IFS=:
while read NAME DEPT ID
do
  # счетчик увеличивается на единицу для каждого совпадающего названия отдела.
  case $DEPT in
  Accounts) ACC_LOOP=`expr $ACC_LOOP + 1`
    ACC="Accounts"
    ;;
  Customer) CUS_LOOP=`expr $CUS_LOOP + 1`
    CUS="Customer"
    ;;
```

```
  Payroll) PAY_LOOP=`expr $PAY_LOOP + 1`
    PAY="Pay roll"
    ;;
  *) echo "`basename $0`: Unknown department $DEPT" >&2
  ;;
  esac
done < names.txt
IFS=$SAVEDIFS
echo "there are $ACC_LOOP employees assigned to $ACC dept"
echo "there are $CUS_LOOP employees assigned to $CUS dept"
echo "there are $PAY_LOOP employees assigned to $PAY dept"
```

При выполнении сценария получим следующий вывод:

```
$ whileread_cond
there are 2 employees assigned to Accounts dept
there are 1 employees assigned to Customer dept
there are 2 employees assigned to Payroll dept
```

18.7.6. Выполнение суммирования

Довольно часто приходится сталкиваться с задачей считывания информации из файла и выполнения суммирования по определенным столбцам, содержащим числа. Предположим, в файле `total.txt` находятся данные о продажах отделами `STAT` и `GIFT`.

```
$ pg total.txt
STAT    3444
GIFT     233
GIFT     252
GIFT     932
STAT     212
STAT     923
GIFT     129
```

Задача состоит в подсчете общей суммы всех записей отдела `GIFT`. Чтобы сохранить общие значения сумм, применим оператор `expr`. Как показано в следующем операторе `expr`, переменным `LOOP` и `TOTAL` первоначально вне цикла присваивается значение ноль. Когда сценарий выполняет цикл, значение переменной `ITEMS` добавляется к значению переменной `TOTAL`. В первую итерацию цикла входит только первый пункт, но в дальнейшем к накапливающимся значениям переменной `TOTAL` добавляются значения переменной `ITEMS`.

Следующий оператор `expr` увеличивает значение счетчика.

```
LOOP=0
TOTAL=0
...
while...
TOTAL=`expr $TOTAL + $ITEMS`
ITEMS=`ex pr $ITEMS + 1`
done
```

Очень распространенной является такая ошибка: при работе с оператором `expr` забывают сначала инициализировать переменную.

```
LOOP=0
TOTAL=0
```

Если переменная не инициализирована, на экране появится сообщение об ошибочном применении оператора `expr`. При необходимости можно инициализировать переменную в цикле:

```
TOTAL=`expr ${TOTAL:=0} + ${ITEMS}`
```

В вышеприведенном примере переменной `TOTAL` присваивается значение нуль, если эта переменная не имеет значения. Чаще распространен первый вариант инициализации переменных с помощью оператора `expr`. Не забывайте выводить на экран конечное общее значение, полученное в результате выполнения цикла.

Рассмотрим следующий сценарий.

```
$ pg $total
#!/bin/sh
# общая сумма
# инициализация переменных
LOOP=0
TOTAL=0
COUNT=0

echo "Items Dept"
echo "_______"
while read DEPT ITEMS
do
  # сохраните результаты подсчета при просмотре общих записей
  COUNT=`expr $COUNT + 1`
  if [ "$DEPT" = "GIFT" ]; then
     # сохраните выполнение суммирования
     TOTAL=`expr $TOTAL + $ITEMS`
     ITEMS=`expr $ITEMS + 1`
     echo -e "$ITEMS\t$DEPT"
  fi
  #echo $DEPT $ITEMS
done < total.txt
echo "============"
echo $TOTAL
echo "There were $COUNT entries altogether in the file"
```

При выполнении сценария получим:

```
$ total
Items Dept
-----------
234     GIFT
253     GIFT
933     GIFT
130     GIFT
===========
1546
There were 7 entries altogether in the file
```

18.7.7. Одновременный просмотр двух записей

В некоторых случаях возникает необходимость в одновременной обработке двух записей, например, если нужно сравнить различные поля двух записей. Чтобы просматривать одновременно по две записи, достаточно после первой конструкции "while read" поместить оператор "do" с другим оператором "read". Применяя эту методику, следует помнить, что количество записей для просмотра составляет четное число. Не забывайте проверять число просматриваемых записей!

Ниже приводится файл, содержащий шесть записей: `record1`, `record2` и т.д.

```
$ pg record.txt
record 1
record 2
record 3
record 4
record 5
record 6
```

В этом примере одновременно просматривается по две записи. Никакой проверки записей в примере не производится.

Ниже приводится соответствующий сценарий.

```
$ pg readpair
#!/bin/sh
# readpair
# первая запись
while read rec1
do
  # вторая запись
  read rec2
  # выполняется дальнейшая обработка/тестирование для тестирования или
  # сравнения обеих записей
  echo "This is record one of a pair :$rec1"
  echo "This is record two of a pair :$rec2"
  echo "------------------------------"
done < record.txt
```

С помощью команды `wc` сначала проверим, что было выполнено четное число записей.

```
$ cat record.txt | wc -l
6
```

Имеется шесть записей, поэтому приступим к их обработке.

```
$ readpair
This is record one of a pair :record 1
This is record two of a pair :record 2
------------------------------
This is record one of a pair :record 3
This is record two of a pair :record 4
------------------------------
This is record one of a pair :record 5
This is record two of a pair :record 6
------------------------------
```

18.7.8. Игнорирование символа

При просмотре текстовых файлов возникает необходимость игнорировать или пропускать строки комментария. Ниже приводится типичный пример.

Предположим, что с помощью обычного цикла `while` просматривается файл конфигурации. Обычно выполняется построчный просмотр.

Однако с помощью оператора `case` можно игнорировать некоторые строки, начинающиеся с определенных символов. Поскольку символ # является специальным, для его отключения используется символ "\"; затем после символа хэша указывается звездочка, что позволит после хэша размещать любые символы.

Ниже приводится типичный файл конфигурации.

```
$ pg config
# ЭТО КОНФИГУРАЦИОННЫЙ ФАЙЛ ПОДСИСТЕМЫ АУДИТА
# НЕ РЕДАКТИРУЙТЕ ЕГО!! ОН РАБОТАЕТ
#
# задание административного доступа
AUDITSCM=full
# местонахождение подсистем
AUDITSUB=/usr/opt/audit/sub
# серийный номер хэша для продукта
HASHSER=12890AB3
# КОНЕЦ ФАЙЛА КОНФИГУРАЦИИ!!!
```

Ниже приводится сценарий, где игнорируются символы хэша:

```
$ pg ignore_hash
#!/bin/sh
# игнорируйте_хэш
INPUT_FILE=config
if [-s $INPUT_FILE ]; then
   while read LINE
   do
    case $LINE in
    \#*) ;; # игнорировать все символы хэша
    *) echo $LINE
       ;;
    esac
   done <$INPUT_FILE
else
   echo "`basename $0` : Sorry $INPUT_FILE does not exist or is empty"
   exit 1
fi
```

При выполнении получим:

```
$ ignore_hash
AUDITSCM=full
AUDITSUB=/usr/opt/audit/sub
HASHSER=12890AB3
```

18.7.9. Работа с форматированными отчетами

Обычно удаление нежелательных строк целесообразно при просмотре файла отчета. Ниже приводится часть отчета, перечисляющая заказ на канцелярские товары. В данном случае представляют интерес столбцы, содержащие уровень переупорядочения пункта и текущий уровень. Ниже приводится сам отчет.

```
$ pg order
######### RE-ORDER REPORT #########
ITEM              ORDERLEVEL     LEVEL
####################################
Pens                  14             12
Pencils               15             15
Pads                   7              3
Disks                  3              2
Sharpeners             5              1
####################################
```

Наша задача заключается в просмотре значения и уточнении, какие пункты следует заказать повторно. Если требуется повторный заказ, нужно дважды его выполнить для данного пункта. Выводной поток данных также должен сообщить, какое количество пунктов нужно заказать повторно, и, кроме того, уточнить общее количество повторных заказов.

Ранее уже рассматривалось, каким образом можно игнорировать строки, начинающиеся определенными символами. Сначала производится чтение из файла. При этом игнорируются все строки, которые начинаются символами # и строками вида "ITEM". После прочтения осуществляется перенаправление данных во временный файл. Затем команда sed применяется для удаления пустых строк. На самом деле все, что делается в этом случае, напоминает фильтрацию текстового файла. Описанные действия реализованы с помощью следующего кода.

```
$ pg whileorder
#!/bin/sh
# whileorder
INPUT_FILE=order
HOLD=order.tmp
if [ -s $INPUT_FILE ]; then
  # пустой файл вывода, добавление не производится!
  >$HOLD
  while read LINE
  do
    case $LINE in
    \#* ITEM*) ;; # -игнорирование строк, содержащих символы # или ITEM
    *)
    # перенаправление вывода во временный файл
    echo $LINE >$HOLD
    ;;
  esac
done <$INPUT_FILE
  # применение команды sed для удаления пустых строк
  sed -e '/^$/d' order.tmp > order.$$
  mv order,$$ order.tmp
else
  echo `"basename $0` : Sorry $INPUT_FILE does not exist or is empty"
fi
```

В результате выполнения сценария получаем следующее:

```
$ pg order.tmp
Pens        14 12
Pencils     15 15
Pads         7 3
Disks        3 2
Sharpeners   5 1
```

Теперь остается только выполнить считывание временного файла в другом цикле `while`, а затем реализовать некоторые сравнения с помощью команды `expr`. Вот соответствующий сценарий:

```
$ pg whileorder2
#!/bin/sh
# whileorder2
# инициализация переменных
HOLD=order.tmp
RE_ORDER=0
ORDERS=0
STATIONERY_TOT=0
if [ -s $HOLD ]; then
  echo "=========== STOCK RE_ORDER REPORT ==============="
  while read ITEM REORD LEVEL
  do
    # находимся ли мы ниже уровня переупорядочивания для данного пункта??
    if [ "$LEVEL" -lt "$REORD" ]; then
    # да, выполняется заказ другого количества товаров
    NEW_ORDER=`expr $REORD + $REORD`
    # подсчет итогов по заказам
    ORDERS=`expr $ORDERS + 1`
    # подсчет итогов по уровням запасов
    STATIONERY_TOT=`expr $STATIONERY_TOT + SLEVEL`
    echo "$ITEM need reordering to the amount $NEW_ORDER"
  fi
  done <$HOLD
  echo "$ORDERS new items need to be ordered"
  echo "Our reorder total is $STATIONERY_TOT"
else
else
echo "`basename $0` : Sorry $HOLD does not exist or is empty"
fi
```

Результат выполнения сценария при обработке файла заказов.

```
$ whileorder

=========== STOCK REORDER REPORT ==============
Pens need reordering to the amount 28
Pads need reordering to the amount 14
Disks need reordering to the amount 6
Sharpeners need reordering to the amount 10
4 new items need to be ordered
Our reorder total is 18
```

Не составляет особого труда скомбинировать два описанных сценария в один; в действительности изначально использовался один сценарий, который был разбит на два в учебных целях.

18.7.10. Цикл while и дескрипторы файлов

При изучении дескрипторов файлов в главе 5 уже упоминалось о том, что для считывания данных в файл применяется цикл `while`. С помощью дескрипторов файлов 3 и 4 следующий сценарий создает резервную копию файла *myfile.txt* под именем *myfile.bak*. Обратите внимание, что в начале сценария осуществляется проверка, которая позволяет убедиться в наличии файла. Если файл отсутствует или не содержит данные, выполнение сценария немедленно прекращается. Также обратите внимание на то, что в цикле while имеется команда `null` (:). Из-за наличия этой команды цикл может выполняться бесконечно, поскольку `null` всегда возвращает значение "истина". При осуществлении попытки считывания по достижении конца файла отображается сообщение об ошибке. При этом выполнение сценария прекращается.

```
$ pg copyfile
#!/bin/sh
# copyfile
FILENAME=myfile.txt
FILENAME_BAK=myfile.bak
if [ -s $FILENAME ]; then
  # открыть FILENAME для записи
  # открыть FILENAME для считывания
  exec 4>$FILENAME_BAK
  exec 3<$FILENAME
# бесконечный цикл до тех пор, пока имеются данные или пока не возникает
# ошибка, связанная с достижением конца файла
  while :
  do
    read LINE <&3
    if [ "$?" -ne 0 ]; then
    # ошибки при закрытии
    exec 3<&-
    exec 4<&-
    exit
  fi
  # запись в файл FILENAME_BAK
  echo $LINE>&4
done else
echo `"basename $0` : Sorry, $FILENAME is notpresent or is empty" >&2
fi
```

18.8. Управление ходом выполнения циклов с помощью команд break и continue

Иногда в процессе работы возникает необходимость в прерывании или пропуске отдельных итераций цикла. При этом применяются определенные критерии. Для обеспечения подобных возможностей интерпретатор shell предлагает две команды:

- `break`;
- `continue`.

18.8.1. Команда break

Команда `break` позволяет прервать выполнение цикла. Эта команда обычно используются для выхода из цикла или прекращения выполнения оператора `case` после осуществления некоторой обработки. Если вы окажетесь внутри вложенного цикла, можно указать количество прерываемых циклов: например, если существуют два вложенных цикла, для их прерывания используется команда `break 2`.

18.8.2. Прекращение выполнения оператора case

Рассмотрим следующий пример. В сценарии выполняется бесконечный цикл до тех пор, пока пользователь не введет число, большее 5. Для прерывания цикла и возврата в командную строку интерпретатора используется команда `break`.

```
$ pg breakout
#!/bin/sh
# breakout
# while : бесконечный цикл
while :
do
  echo -n "Enter any number [1..5] :"
  read ANS
  case $ANS in
    1|2|3|4|5) echo "great you entered a number between 1 and 5"
    ;;
  *) echo "Wrong number..bye"
    break
    ;;
  esac
done
```

18.8.3. Команда continue

Команда `continue` по своему действию напоминает команду `break`, за исключением одной существенной детали: она не прерывает цикл, а лишь приводит к пропуску текущей итерации цикла.

18.8.4. Пропуск строк в файлах

Рассмотрим файл, содержащий перечень сотрудников. Этот файл уже использовался ранее, но теперь он будет включать некоторую заголовочную информацию.

```
$ pg names2.txt
------LISTING OF PERSONNEL FILE-----
--- TAKEN AS AT 06/1999 -----
Louise Conrad:Accounts:ACC8987
Peter James:Payroll:PR489
Fred Terms:Customer:CUS012
James Lenod:Accounts:ACC887
Frank Pavely:Payroll:PR489
```

При просмотре этого файла нетрудно заметить, что его первые две строки не содержат сведений о сотрудниках. Поэтому эти строки желательно исключить.

Также не следует принимать во внимание сведения о сотруднике с именем Peter James. Этот человек уволился из компании, но запись о нем осталась в файле.

Давайте все же сохраним заголовочную информацию, выполнив простой подсчет считываемых строк. При этом обработка файла будет выполняться в том случае, когда номер строки будет больше двух. Если строка содержит имя Peter James, она будет исключена из процесса обработки.

Вот сценарий, выполняющий описанные задачи.

```
$ pg whilecontinue
#!/bin/sh
# whilecontinue
SAVEDIFS=$IFS
IFS=:
INPUT_FILE=names2.txt
NAME_HOLD="Peter James"
LINE_NO=0
if [ -s $INPUT_FILE ]; then
  while read NAME DEPT ID
  do
    LINE_NO=`expr $LINE_NO + 1`
    if [ "$LINE_NO" -le 2 ]; then
      # пропуск, если номер строки меньше 2
      continue
  fi
  if [ "$NAME" = "$NAME_HOLD" ]; then
    # пропуск, если переменной NAME_HOLD присвоено имя Peter James
    continue
  else
    echo " Now processing...SNAME SDEPT $ID"
    # обработка файла
  fi
  done < $INPUT_FILE
IFS=$SAVEDIFS
else
  echo "`basename $0` : Sorry file not found or there is no data in the
file" >&2
  exit 1
fi
```

При выполнении сценария получим следующие результаты:

```
$ whilecontinue
Louise Conrad Accounts ACC8987
Fred Terms Customer CUS012
James Lenod Accounts ACC887
Frank Pavely Payroll PR489
```

18.9. Меню

При создании меню представляется весьма удобным использование команды null совместно с циклом while. Объединение этих конструкций приводит к бесконечному циклу, что и требуется в меню. Цикл должен выполняться до тех пор, пока пользователь не осуществит выход или не выберет нужную опцию.

Для создания меню потребуется цикл while и оператор case, задающий шаблоны, с которыми сравниваются результаты ввода пользователя. При некорректном вводе обычно раздается звуковой сигнал и отображается сообщение об ошибке. Затем цикл продолжает выполняться до тех пор, пока пользователь не выберет опцию выхода из меню.

Меню должно быть дружественным по отношению к пользователю; все его элементы должны быть интуитивно понятными. Главный экран меню обычно отображает имя хоста и дату, а также имя пользователя, работающего с данным меню. В целях тестирования все опции меню будут использовать системные команды.

Ниже приводится примерный вид меню.

```
-------------------------------------------------------
User: dave          Host:Bumper              Date:31/05/1999
-------------------------------------------------------
            1 :  List files in current directory
            2 :  Use the vi editor
            3 :  See who is on the system
            H :  Help screen
            Q :  Exit Menu
-------------------------------------------------------
         Your Choice [1,2,3,H,Q] >
```

Сначала воспользуемся подстановкой команд для назначения даты, имени хоста и пользователя. Для указания даты используется формат DD/MM/YYYY. Этот формат задается с помощью следующего параметра:

```
$ date +%d/%B/%Y
32/05/1999
```

Для указания имени хоста может применяться опция -s, которая просто отображает часть имени хоста. Иногда имя хоста задается путем присвоения соответствующей переменной полностью определенного доменного имени. Существует возможность отображения на экране меню подобного длинного имени хоста.

Присвоим переменным осмысленные имена:

```
MYDATE = `date +%d/%m/%Y`
THIS_HOST= `hostname -s`
USER= `whoami`
```

В цикле while команда null помещена непосредственно после слова "while". Вот формат бесконечного цикла:

```
while :
do
   команды..
done
```

При отображении экрана меню не тратьте зря время, указывая множество одиночных конструкций echo, — потом будет очень трудно настроить вывод. Здесь мы встречаем конструкцию, которая обеспечивает ввод данных после слова-ограничителя до того момента, пока вновь не встретится это слово. В этом случае применяется следующий формат:

```
команда << СЛОВО
любые вводимые данные
СЛОВО
```

Все это используется для экрана меню. Можно также воспользоваться этой технологией для экрана помощи. Созданный экран помощи выглядит не слишком информативным.

Для обработки результатов выбора пользователя используется конструкция case. Варианты выбора меню будут следующими:

```
1: List files in current directory
2: Use the vi editor
3: See who is on the system
H: Help screen
Q: Exit Menu
```

Оператор case должен обрабатывать все шаблоны, переводя все строчные символы шаблона в прописные. При этом устраняется риск случайного нажатия пользователем клавиши [Caps Lock], в результате чего могут искажаться вводимые данные. В случае, когда сценарий меню выполняет бесконечный цикл, пользователь нуждается в реализации элегантного выхода. В связи с этим, если пользователь выбирает клавишу [Q] или [q], сценарий должен осуществить выход с нулевым значением.

Если пользователь выполняет некорректный ввод, раздается звуковой сигнал и отображается предостерегающее сообщение. Хотя в начале этой главы упоминалось о том, что будут применяться операторы echo Linux BSD, для генерации звукового сигнала будет применена команда из версии System V:

```
echo "\007 the bell rang"
```

Конструкции echo и read применяются для задержки отображения экрана до тех пор, пока пользователь не нажмет клавишу [Enter]. При этом могут просматриваться любые сообщения или вывод команд.

В конце концов, потребуется очистить экран. Эта задача выполняется с помощью команды tput (о применении этой команды мы поговорим позже) в том случае, когда не используется clear. А теперь посмотрите на сам сценарий.

```
$ pg menu
#!/bin/sh
# меню
# установка даты, имени пользователя и хоста
MYDATE=`date +%d/%m/%Y`
THIS_HOST=`hostnaiDe -s`
USER=`whoami`
# бесконечный цикл!
while :
do
  # очистка экрана с помощью команды tput
  # здесь начинается конструкция "документ здесь"
  cat <<MAYDAY
  ---------------------------------------------------------------
  User; $USER              Host:$THIS_HOST            Date:$MYDATE
  ---------------------------------------------------------------
                1 : List files in current directory
                2 : Use the vi editor
                3 : See who is on the system
                H : Help screen
                Q : Exit Menu
  ---------------------------------------------------------------
  MAYDAY
```

```
  # завершение конструкции "документ здесь"
  echo -e -n "\tYour Choice [1,2,3,H,Q] >"
  read CHOICE
    case $CHOICE in
    1) ls
      ;;
    2) vi
      ;;
    3) who
      ;;
    H|h
    # использование конструкции "документ здесь" для экрана помощи
    cat <<MAYDAY
    This "is the help screen, nothing here yet to help you!
    MAYDAY
      ;;
    Q|q) exit 0
      ;;
    *) echo -e "\t\007unknown user response"
      ;;
    esac
  echo -e -n "\tHit the return key to continue"
  read DUMMY
done
```

18.10. Заключение

Ядро любого более или менее сложного сценария образуют различные управляющие конструкции. Если вы ожидаете от сценария определенного "интеллекта", он должен быть способным принимать решения.

В этой главе было изучено, каким образом управляющие конструкции обеспечивают компромисс между хорошими и надежными сценариями. Здесь также изучается обработка списков и рассматривается работа с циклами, условиями для которых являются значения "истина" или "ложь".

ГЛАВА 19

Функции интерпретатора shell

До сих пор весь программный код сценариев данной книги выполнялся последовательно от начала до конца программы. Подобный подход неплох, но при этом некоторые фрагменты кода, рассмотренного в наших примерах, дублируются в пределах одного сценария.

Интерпретатор команд shell позволяет группировать наборы команд или конструкций, создавая повторно используемые блоки. Подобные блоки называются shell-функциями.

В данной главе будут рассмотрены следующие темы:

- определение функций;
- работа с функциями в сценарии;
- использование функций, определенных в файле функций;
- примеры функций.

Функция состоит из двух частей:

```
Метка функции
Тело функции
```

В качестве метки выступает имя функции; тело функции образует набор команд, составляющих саму функцию. Имя функции должно быть уникальным: если это не так, результаты будут плачевны. Это связано с тем, что в случае наличия двух различных функций с одинаковыми именами сценарий просто не сможет вызвать нужную функцию.

Формат, применяемый для определения функций:

```
имя_функции()
{
команда1
...
}
```

или

```
имя_функции() {
команда 1          \
...
}
```

Приемлемы оба способа определения функции. Можно также использовать ключевое слово `function` перед именем функции `имя_функции`.

```
function имя_функции()
{
...
}
```

Функцию можно представлять себе как некоторый вид сценария, находящегося внутри другого сценария, но в этом случае следует учитывать одну особенность. При вызове функции она остается в текущем интерпретаторе shell, а ее копия хранится в памяти. С другой стороны, если вызывается или выполняется сценарий из другого сценария, создается отдельный интерпретатор shell. В таком случае становятся недействительными все существующие переменные, определенные в предыдущем сценарии.

Функции могут быть размещены в том же самом файле, что и сценарии, либо в отдельном файле, содержащем функции. При этом функции не всегда включают множество конструкций или команд; может просто содержаться единственная конструкция `echo`, которая выполняется при вызове функции.

19.1. Объявление функций в сценарии

Вот пример простой функции:

```
hello ()
{
echo "Hello there today's date is `date`"
}
```

Перед использованием функций их необходимо объявить. Суть объявления заключается в том, что все функции должны быть размещены в начале кода сценария. Невозможно сослаться на функцию до тех пор, пока она не попадет в "поле зрения" интерпретатора команд. Для вызова функции требуется просто ввести ее имя. В предыдущем примере функция называлась "hello"; тело функции включало конструкцию `echo`, которая, в свою очередь, отображала текущую дату.

19.2. Использование функций в сценарии

После создания функции рассмотрим методы ее применения в сценарии.

```
$ pg func1
#!/bin/sh
# func1
hello ()
{
echo "Hello there today's date is `date`"
}

echo "now going to the function hello"
hello

echo "back from the function"
```

При выполнении этого сценария получаются следующие результаты:

```
$ func1
now going to the function hello
Hello there today's date is Sun Jun 6 10:46:59 GMT 2000
back from the function
```

В предыдущем примере функция была объявлена в начале сценария. Для обращения к функции просто вводится ее имя, которое в данном случае звучит как "hello". После завершения выполнения функции управление возвращается следующей конструкции, которая размещена после вызова функции. В приведенном примере речь идет о конструкции `echo "back from the function"`.

19.3. Передача параметров функции

Порядок передачи параметров функции аналогичен передаче параметров обычному сценарию. При этом используются специальные переменные `$1`, `$2`, ... `$9`. При получении функцией переданных ей аргументов происходит замена аргументов, изначально переданных сценарию интерпретатора shell. В связи с этим неплохо было бы повторно присвоить значения переменным, получаемым функцией. В любом случае это стоит сделать, поскольку при наличии ошибок в функциях их можно будет легко обнаружить, воспользовавшись именами локальных переменных. Для вызывающих аргументов (переменных), находящихся внутри функции, имя каждой переменной начинается с символа подчеркивания, например: `_FILENAME` или `_filename`.

19.4. Возврат значения функции

После естественного завершения выполнения функции либо в том случае, когда она завершается в результате выполнения какого-либо условия, можно выбрать один из двух возможных вариантов:

1. Дождаться, пока функция естественным образом не завершится сама с последующей передачей управления той части сценария, которая вызвала данную функцию.
2. Воспользоваться ключевым словом `return`, в результате чего будет осуществлена передача управления конструкции, которая расположена за оператором вызова функции. При этом может также указываться необязательный числовой параметр. Этот параметр принимает значение 0 в случае отсутствия ошибок и значение 1 — при наличии ошибок. Действие этого параметра аналогично действию кода завершения последней команды. При использовании ключевого слова `return` применяется следующий формат:

return	возвращает результат из функции, использует код завершения последней команды для проверки состояния
return 0	применяется при отсутствии ошибок
return 1	применяется при наличии ошибок

19.5. Проверка значений, возвращаемых функцией

Для проверки значения, возвращаемого вызванной функцией, можно воспользоваться кодом завершения последней команды, размещенной непосредственно после функции, которая вызывается из сценария. Например:

```
check_it_is_a_directory $FILENAME # вызов функции и проверка
if [ $? = 0 ] # применение кода завершения последней команды для
тестирования
then
  echo "All is OK"
else
  echo " Something went wrong!"
fi
```

Лучшим методом является использование оператора `if`, с помощью которого осуществляется проверка возвращаемого значения (0 или 1). Встраивание вызова функции в структуру оператора `if` значительно улучшает читабельность программного кода. Например:

```
if check_it_is_a_directory $FILENAME; then
  echo "All is OK"
  # действия
else
  echo "Something went wrong!"
  # действия
fi
```

Если функцию планируется использовать для отображения результатов некоторой проверки, для фиксации результата применяется подстановка. Формат, используемый для выполнения подстановки при вызове функции, будет следующим:

```
имя_переменной = `имя_функции`
```

Выводимый результат функции `имя_функции` присваивается переменной `имя_переменной`.

В последующих разделах будет продемонстрировано большое количество разнообразных функций, а также рассмотрены различные способы применения возвращаемых и выводимых значений для функции.

19.6. Файл функций

После того, как будет создано несколько регулярно используемых функций, их можно поместить в файл функций, а затем загружать этой файл в среду интерпретатора shell.

В начале файла функции должна находиться конструкция `#!/bin/sh`. Этому файлу можно присвоить любое имя, но все же лучше использовать какое-либо осмысленное наименование, например *functions.main*.

Как только файл будет загружен интерпретатором shell, появляется возможность вызова функций из командной строки либо из сценария. Чтобы просмотреть все определенные функции, воспользуйтесь командой `set`. Поток вывода будет содержать все функции, которые были загружены интерпретатором shell.

Для изменения любой из ранее определенных функций сначала примените команду `unset`, после чего функция будет выгружена из интерпретатора shell. В результате этого данную функцию будет невозможно вызвать из пользовательских сценариев или интерпретатора команд, хотя физического удаления функции при этом не происходит. После внесения изменений в файл функций выполните его перезагрузку. Некоторые интерпретаторы команд могут распознавать изменения, и в этом случае применение команды `unset` вовсе не обязательно. Однако все же в целях безопасности всегда лучше использовать данную команду при модификации функций интерпретатора shell.

19.7. Создание файла функций

А теперь создадим файл функций, включающий одну функцию. Эта функция будет загружена интерпретатором команд, протестирована, изменена, а затем повторно загружена.

Создаваемый файл функций `functions.main` будет содержать следующий код:

```
$ pg functions.main
#!/bin/sh
# functions.main
#
# findit: интерфейс для базовой команды find
findit() {
# findit
if [ $# -lt 1 ]; then
  echo "usage :findit file"
  return 1
fi
find / -name $1 -print
```

Код, приведенный выше, ранее уже упоминался в книге, но теперь он включен в состав функции. Этот код лежит в основе интерфейса для базовой команды `find`. Если команде не передаются аргументы, то возвращается значение 1 (что свидетельствует о возникновении ошибки). Обратите внимание, что ошибочная конструкция фактически является отображенным именем функции (если же была использована команда `$0`, интерпретатор команд просто возвращает сообщение `sh-`). Причина отображения подобного сообщения заключается в том, что файл не является файлом сценария. В любом случае это сообщение не несет много информации для пользователя.

19.8. Подключение файла функций

Команда подключения файла функций имеет следующий формат:

```
. /путь/имя_файла
```

Теперь, когда файл создан, настало время загрузить его содержимое в интерпретатор команд (подключить его). Введите команду:

```
$. functions.main
```

Если в результате выполнения этой команды возвращается сообщение 'file not found' (файл не найден), попытайтесь воспользоваться следующей командой:

```
$. /functions.main
```

В этом случае применяется нотация <точка><пробел><косая черта><имя файла>. Теперь файл должен быть загружен интерпретатором команд. Если по-прежнему возникают ошибки, убедитесь в том, что было указано полное наименование пути к файлу.

19.9. Проверка загруженных функций

Чтобы удостовериться в том, что функции были загружены интерпретатором команд, воспользуйтесь командой `set`, которая отображает все загруженные функции, доступные для сценария.

```
$ set
USER=dave
findit=()
{
if [ $# -lt 1 ]; then
  echo "usage :findit file";
  return 1;
fi;
find / -name $1 -print
}
...
```

19.10. Вызов функций интерпретатора shell

Для вызова функции просто введите ее имя (в данном случае `findit`) и укажите аргумент, в роли которого может выступать файл, размещенный в системе.

```
$ findit groups
/usr/bin/groups
/usr/local/backups/groups.bak
```

19.10.1. Удаление shell-функций

Теперь настало время немного изменить функцию. Сначала функция будет удалена, в результате чего она не будет доступна интерпретатору shell. Для выполнения этой операции применяется команда `unset`. При вызове данной команды используется следующий формат:

```
unset имя_функции
```

```
$ unset findit
```

Если вы сейчас введете команду `set`, функция не будет найдена.

19.10.2. Редактирование shell-функций

А теперь изменим файл *functions.main*. Добавим в функцию цикл `for`, в результате чего сценарий сможет считывать более одного параметра из командной строки. Функция приобретет следующий вид:

```
$ pg functions.main
#!/bin/sh
findit()
```

```
{
# findit
#if [ $# -lt 1 ]; then
  echo "usage :findit file"
  return 1
fi
for loop
do
  find / -name $loop -print
done
}
```

Снова загрузим исходный файл:

```
$. /functions.main
```

Используйте команду `set`, чтобы убедиться, что файл действительно загружен. Если интерпретатор shell корректно интерпретирует цикл `for` и воспринимает все требуемые параметры, на экране отобразится соответствующее сообщение.

```
$ set
findit=()
{
if [ $# -lt 1 ]; then
  echo "usage :`basename $0` file";
  return 1;
fi;
for loop in "$@";
do
  find / -name $loop -print;
done
}
...
```

Далее вызывается измененная функция `findit`. При этом поддерживаются два файла с целью осуществления поиска:

```
$ findit LPSO.doc passwd
/usr/local/accounts/LPSO.doc
/etc/passwd
...
```

19.10.3. Примеры функций

Теперь, когда вы получили начальные сведения о функциях, рассмотрим их практическое применение.

Сразу же стоит заметить, что использование функций позволяет сэкономить массу времени и усилий программистов. Это возможно благодаря тому, что функция является повторно применяемой.

Подтверждение ввода

Рассмотрим небольшой сценарий, который запрашивает имя и фамилию пользователя:

```
$ pg func2
#!/bin/sh
```

```
# func2
echo -n "What is your first name :"
read F_NAME
echo -n "What is your surname :"
read S_NAME
```

Данный сценарий предназначен для выполнения проверки значений переменных (переменным присваиваются исключительно одни символы). Выполнение этой задачи без применения функций может привести к дублированию программного кода. Функции позволяют обойтись без дублирования. Для проверки наличия одних лишь символов можно воспользоваться утилитой `awk`. Ниже приводится функция, которая осуществляет проверку ввода либо только прописных, либо строчных символов.

```
char_name()
{
# char_name
# вызов: char_name string
# назначение аргумента новой переменной
_LETTERS_ONLY=$1
# использование awk для проверки на наличие символов !
_LETTERS_ONLY=`echo $1|awk '{if($0~/[^a-z A-Z]/)print "1"}'`
if [ "$_LETTERS_ONLY" != "" ] then
  # ошибки
  return 1
else
  # содержит только символы
  return 0
fi
}
```

Сначала переменной `$1` будет присвоено более осмысленное имя. Затем применяется утилита `awk`, осуществляющая проверку, состоит ли переданная строка из одних литер. В результате выполнения возвращается код, включающий 1 (для символов, не являющихся литерами) и 0 — для символов-литер. Этот код присваивается переменной `_LETTERS_ONLY`.

Затем выполняется проверка значения переменной. Если переменной присвоено какое-либо значение, то это свидетельствует о наличии ошибки; в случае отсутствия присвоенного значения ошибки нет. На основании результатов этой проверки формируется код возврата. Использование кода возврата позволяет сценарию фиксировать момент завершения проверки, выполняемой функцией в вызывающей части сценария.

Для проверки вывода функции можно использовать формат, применяемый для оператора `if`:

```
if char_name $F_NAME; then
  echo "OK"
else
  echo "ERRORS"
fi
```

Если происходит ошибка, можно создать другую функцию, отображающую сообщение об ошибке:

```
name_error()
# name_error
```

```
# отображение сообщения об ошибке
{
echo " $@ contains errors, it must contain only letters"
}
```

Функция name_error будет отображать сообщения об ошибках, игнорируя при этом все некорректные записи. Применение специальной переменной $@ позволяет отображать на экране значения всех аргументов. В рассматриваемом случае будет отображено либо значение F_NAME, либо значение S_NAME. А теперь приведем завершенный сценарий, созданный с применением функций:

```
$ pg func2
!/bin/sh
char_name()
# наименование_символа
# вызов: char_name строка
# проверка на предмет того, действительно ли $1 содержит только символы
a-z,A-Z
{
# присвоение аргумента новой переменной
_LETTERS_ONLY=$1
_LETTERS_ONLY=`echo $1|awk '{if($0~/[^a-zA-Z]/) print "1"}'`
if [ "$_LETTERS_ONLY" != "" ]
then
  # присутствуют ошибки
  return 1
else
  # содержит только символы
  return 0
fi
}

name_error()
# отображение сообщения об ошибке
{
echo " $@ contains errors, it must contain only letters"
}

while :
do
  echo -n "What is your first name :"
  read F_NAME
  if char_name $F_NAME
  then
    # все OK, завершение выполнения
    break
  else
    name_error $F_NAME
  fi
done

while :
do
  echo -n "What is your surname :"
```

```
  read S_NAME
  if char_name $S_NAME
  then
    # все OK, завершение выполнения
    break
  else
    name_error $S_NAME
  fi
done
```

Обратите внимание на то, что для обработки результатов ввода применяется цикл while. Благодаря этому на экране отображается запрос до тех пор, пока пользователь не укажет верное значение. После ввода нужного значения выполнение цикла прерывается. Конечно, при реализации работающего сценария пользователю предоставляется возможность прервать выполнение цикла. При этом также применяется соответствующее управление курсором, например при проверке наличия полей с нулевой длиной.

Ниже приведены результаты выполнения описанного сценария:

```
$ func2
What is your first name :Davi2d
Davi2d contains errors, it must contain only letters
What is your first name :David
What is your surname :Tansley1
Tansley1 contains errors, it must contain only letters
What is your surname :Tansley
```

Проблемы с конструкцией echo

В системах Linux, BSD или System V конструкция echo по-разному интерпретирует служебные символы. Создадим функцию, определяющую систему, в которой используется конструкция echo.

После того как была применена конструкция echo, командная строка может и далее отображаться на экране, ожидая ввода данных со стороны команды read.

Для реализации описанного поведения в системах Linux и BSD совместно с командой echo применяется опция -n. Ниже приводится пример конструкции echo LINUX (BSD), когда командная строка продолжает отображаться на экране до момента завершения сценария:

```
$ echo -n "Your name :"
Your name : {{?}}
```

В System V в этом случае применяется параметр \c:

```
$ echo "Your name :\c"
Your name : □
```

Для отображения на экране управляющих символов в Linux также потребуется указывать опцию -e в начале оператора echo. В других системах достаточно просто воспользоваться обратной косой чертой, в результате чего интерпретатор shell будет "уведомлен" о наличии управляющего символа.

Существует два метода проверки типа используемой конструкции echo. Мы рассмотрим оба этих метода, и вы сможете выбрать один из них для дальнейшего применения.

При использовании первого метода проверка управляющего символа происходит внутри конструкции echo. Если после ввода команды echo \007 прозвучал звуковой сигнал, значит, перед нами System V. Если отображается строка "\007", значит, текущей операционной системой является Linux.

Вот первая функция, выполняющая проверку на наличие управляющих символов.

```
uni_prompt ()
# uni_prompt
# универсальная конструкция echo
{
if [ `echo "\007"` = "\007" ] >/dev/null 2>&1
# слышен звуковой сигнал либо отображаются символы?
then
  # отображаются символы, это LINUX/BSD
  echo -e -n "$@"
else
  # это System V
  echo "$@\c"
fi
}
```

И снова обратите внимание на применение специальной переменной $@ для отображения строки. Для вызова функции в сценарии можно использовать следующую команду:

```
uni_prompt "\007There goes the bell, What is your name :"
```

В результате выполнения этой команды выдается звуковой сигнал и отображается строка "What is your name", которая остается на экране.

Для проверки на наличие новой строки можно отобразить любой символ, используя версию команды echo \c системы System V. Если символ "зависает" в конце строки, мы имеем дело с System V; если же нет, значит у нас система Linux/BSD.

Второй метод заключается в проверке того, будет ли литера Z "зависать" в конце строки. При этом используется опция \c системы System V.

```
uni_prompt ()
# uni_prompt
# универсальная командная строка
{
if [ `echo "Z\c"` = "Z" ] >/dev/null 2>&1
then
  # System V
  echo "$@\c"
else
  # LINUX/BSD
  echo -e -n "$@"
fi
}
```

Для вызова данной функции из сценария применяется команда:

```
uni_prompt "\007 There goes the bell, What is your name :"
```

Вызов любой из описанных выше функций возможен с помощью следующего кода:

```
uni_prompt "\007 There goes the bell, What is your name :" read NAME
```

В результате мы получим:

```
There goes the bell, What is your name :
```

Чтение одиночного символа

При создании меню одной из самых неприятных задач является необходимость нажимать клавишу [Return] после выбора каждого пункта меню либо в ответ на сообщение "нажмите любую клавишу для продолжения". В этом случае приходит на помощь команда dd, избавляющая пользователя от необходимости нажимать клавишу [Return] для отсылки ключевой последовательности.

Команда `dd` обычно применяется для выполнения различных преобразований и устранения проблем, связанных с хранением данных на лентах, либо при выполнении обычных задач резервирования. Данная команда также может применяться для создания файлов фиксированного размера. В следующем примере с ее помощью создается файл `myfile`, размер которого составляет 1 мегабайт.

```
dd if=/dev/zero of=myfile count=512 bs=2048
```

Команда `dd` также может интерпретировать результаты ввода с клавиатуры и использоваться для чтения символов. В данном случае ожидается появление лишь одного символа. Команда `dd` должна завершать выполнение после нахождения символа новой строки; этот управляющий символ появляется после нажатия пользователем клавиши [Return]. Команда `dd` в данном случае также посылает один символ. Перед тем как произойдет одна из описанных ситуаций, необходимо установить терминал в исходный режим. Для этого применяется команда `stty`. Настройки в приведенном ниже коде сохраняются перед вызовом команды `dd` и затем восстанавливаются после завершения выполнения команды `dd`.

```
read_a_char()
# read_a_char
{
# сохранение настроек
SAVEDSTTY=`stty -g`
# задание параметра терминала
stty cbreak
# чтение и вывод лишь одного символа
dd if=/dev/tty bs=1 count=1 2> /dev/null
# восстановление параметра терминала и настроек
stty -cbreak
stty $SAVEDSTTY
}
```

Для вызова функции и возврата введенного символа используется подстановка команд. Вот соответствующий пример.

```
echo -n "Hit Any Key To Continue"
character=`read_a_char`
echo " In case you are wondering you pressed $character"
```

Проверка наличия каталога

Проверка наличия каталогов является весьма распространенной задачей, возникающей при копировании файлов. Приведенная ниже функция проверяет имя файла, переданное функции, что позволяет установить наличие каталога. Поскольку в этом случае используется команда `return` с параметром, изменяющимся в зависимости от успеха или неудачи при выполнении команды, для проверки результатов наиболее удачным является выбор конструкции `if`.

```
is_it_a_directory()
(
# is_it_a_directory
# вызов: is_it_a_directory имя_каталога
if [ $# -lt 1 ]; then
  echo "is_it_a_directory: I need an argument"
  return 1
fi
# это каталог ?
_DIRECTORY_NAME=$1
if [ ! -d $_DIRECTORY_NAME ]; then
  # нет
  return 1
else
  # да
  return 0
fi
}
```

Для вызова функции и проверки результата можно воспользоваться кодом:

```
echo -n "enter destination directory :"
read DIREC
if is_it_a_directory $direc;
then :
else
  echo "SDIREC does not exist, create it now ? [y..n]"
  # здесь должны находится команды для создания каталога или для выхода
  ...
  ...
fi
```

Запрос на ввод Y или N

Многие сценарии выдают запрос на ввод подтверждения перед выполнением дальнейшей обработки. Запрос может выглядеть следующим образом:

```
Create a directory
Do you wish to delete this file
Run the backup now~
Confirm to save a record
```

Этот перечень может быть достаточно длинным.

Следующая функция реализует обобщенный запрос. Пользователь указывает отображаемое сообщение, а также ответ, выбираемый по умолчанию. Ответ, выбираемый по умолчанию, используется в том случае, если пользователь просто нажал клавишу [Return]. Конструкция `case` применяется для перехвата ответов.

```
continue_prompt()
# continue_prompt
# вызов: continue_prompt "отображаемая_строка" ответ_по_умолчанию
{
_STR=$1
_DEFAULT=$2
# проверка на предмет указания правильных параметров
if [ $# -lt 1 ]; then
  echo "continue_prompt: I need a string to display"
  return 1
fi
# бесконечный цикл
while :
do
  echo -n "$_STR [Y..N] [$_DEFAULT]:"
  read _ANS
  # если пользователь нажал [Return], устанавливаются настройки
  # по умолчанию и определяется возвращаемое значение,
  # ниже находится пробел, а затем символ $
  : ${_ANS:=$_DEFAULT}
  if [ "$_ANS" = "" ]; then
    case $_ANS in
    Y) return 0 ;;
    N) return 1 ;;
    esac
  fi
  # пользователь что-то выбрал
  case $_ANS in
  y|Y|Yes|YES)
    return 0
    ;;
  n|N|No|NO)
    return 1
    ;;
  *) echo "Answer either Y or N, default is $_DEFAULT"
    ;;
  esac
  echo $_ANS
done
}
```

Для вызова функции можно указать отображаемое сообщение в двойных кавычках либо вызвать ее вместе с аргументом `$1`, либо, в крайнем случае, использовать переменную, содержащую строку. Также может передаваться ответ, заданный по умолчанию, в виде 'Y' или 'N'.

Ниже демонстрируется несколько методов, с помощью которых можно вызвать функцию `continue_prompt`.

```
if continue_prompt "Do you want to delete the var filesytem" "N"; then
  echo "Are you nuts!!"
else
  echo "Phew, what a good answer!"
fi
```

При использовании этого кода можно указывать следующий ввод:

```
Do you really want to delete the var filesystem [Y..N] [N]:
Phew, what a good answer!
Do you really want to delete the var filesystem [Y..N] [N]:y
Are you nuts!!
```

Теперь вам понятно, почему функция имеет ответ, заданный по умолчанию. Причем этот ответ может задавать сам пользователь!

Ниже приводится другой способ вызова функции:

```
#if continue_prompt "Do you really want to print this report" "Y"; then
  lpr report
else:
fi
```

Функцию можно также вызвать с использованием переменной `$1`, содержащей строку:

```
#if continue_prompt $1 "Y"; then
  lpr report
else:
fi
```

Получение сведений об идентификаторе регистрации

При работе в составе большой системы может возникнуть необходимость связаться с одним из зарегистрированных пользователей. Что делать в том случае, если вы не помните имя пользователя? Часто системные администраторы наблюдают идентификаторы пользователей, блокирующих какие-либо процессы. Для того чтобы увидеть полное имя пользователя, они должны просмотреть файл `passwd` с помощью команды `grep`. И только после этого можно связаться с пользователем, чтобы сделать ему справедливый выговор.

А сейчас мы рассмотрим функцию, которая позволит избежать просмотра файла `/etc/passwd` с помощью команды `grep`.

В тестовой системе полное имя пользователя хранится в пятом поле файла `passwd`; в вашей системе все может быть по-другому, поэтому, возможно, придется изменить номер поля, чтобы обеспечить соответствие с файлом `passwd`.

Функции передается один либо множество идентификаторов пользователей, а функция имитирует действие команды `grep` по отношению к файлу `passwd`.

Программный код функции:

```
whois()
# whois
# вызов: whois идентификатор_пользователя
{
# проверка на наличие корректных параметров
if [ $# -lt 1 ]; then
  echo "whois : need user id's please"
  return 1
fi

for loop
do
```

```
  _USER_NAME=`grep $loop /etc/passwd | awk -F: '{print $4}'`
  if [ "$_USER_NAME" = "" ]; then
    echo "whois: Sorry cannot find $loop"
  else
    echo "$loop is $_USER_NAME"
  fi
done
}
```

Функция whois может быть вызвана следующим образом:

```
$ whois dave peters superman
dave is David Tansley - admin accts
peter is Peter Stromer - customer services
whois: Sorry cannot find superman
```

Использование нумерации в текстовом файле

При использовании редактора vi появляется возможность нумерации строк. Это полезно в целях отладки, но при выводе на печать некоторых файлов с номерами строк потребуется команда nl. Ниже приведена функция, имитирующая действие команды nl (выполняет нумерацию строк файла). Исходный файл при этом не перезаписывается. Вот программный код функции.

```
number_file()
# number_file
# вызов: number_file имя_файла
{
_FILENAME=$1
# проверка наличия корректных параметров
if [ $# -ne 1 ]; then
  echo "number_file: I need a filename to number"
  return 1
fi

loop=1
  while read LINE
  do
    echo "$loop: $LINE"
    loop=`expr $loop + 1`
done < $_FILENAME
}
```

Для вызова функции number_file просто укажите имя файла, подлежащего нумерации, в качестве аргумента функции. Вызов функции может также осуществляться из среды интерпретатора shell путем указания имени файла. Например:

```
$ number_file my file
```

Кроме того, функцию можно вызвать из сценария, воспользовавшись с этой целью предыдущим примером. Или вы можете задать оператор:

```
number_file $1
```

Результат выполнения функции может выглядеть следующим образом:

```
$ number_file /home/dave/file_listing
1: total 105
2: -rw-r--r-- 1 dave admin 0 Jun 6 20:03:DT
3: -rw-r--r-- 1 dave admin 306 May 23 16:00 LPSO.AKS
4: -rw-r--r-- 1 dave admin 306 May 23 16:00 LPSO.AKS.UC
5: -rw-r--r-- 1 dave admin 324 May 23 16:00 LPSO.MBB
6: -rw-r--r-- 1 dave admin 324 May 23 16:00 LPSO.MBB.UC
7: -rw-r--r-- 1 dave admin 315 May 23 16:00 LPSO.MKQ
...
...
```

Преобразование символов в прописные

Иногда возникает потребность в преобразовании строчных символов в прописные. Например, для создания каталогов в файловой системе используются прописные символы. Кроме того, при вводе данных в поля может выполняться проверка, являются ли все символы прописными.

Ниже приводится соответствующая функция. Эта функция имитирует действие команды `tr`.

```
str_to_upper ()
# str_to_upper
# вызов: str_to_upper $1
{
 STR=$1
# проверка на наличие корректных параметров
if [ $# -ne 1 ]; then
  echo "number_file: I need a string to convert please"
  return 1
fi
echo $@ |tr '[a-z]' '[A-Z]'
}
```

Переменной `UPPER` присваивается строка, символы которой преобразованы в прописные символы. Обратите внимание, что снова применяется специальный символ `$@` для передачи всех аргументов. Функция `str_to_upper` может вызываться двумя способами. Можно указать строку в сценарии следующим образом:

```
UPPER=`str_to_upper "documents.live"`
echo $upper
```

либо указать аргумент функции вместо строки:

```
UPPER=`str_to_upper $1`
echo $UPPER
```

В обоих примерах используется подстановка для получения результатов, возвращаемых функцией.

```
is_upper
```

Функция `str_to_upper` осуществляет преобразование регистра символов. Но иногда перед выполнением дальнейшей обработки необходимо только знать, содержатся ли в строке прописные символы. Это может потребоваться для записи информации в текстовое поле файла. Функция `is_upper` выполняет именно эту задачу. Применение

оператора `if` в сценарии позволит определить, будет ли передаваемая строка включать прописные символы.

Программный код функции:

```
is_upper()
# is_upper
# вызов: is_upper $1
{
# проверка на наличие корректных параметров
if [ $# -ne 1 ]; then
  echo "is_upper: I need a string to test OK"
  return 1
fi
# применение awk для проверки на наличие прописных символов
_IS_UPPER=`echo $1|awk '{if($0~/[^A-Z]/) print "1"}'`
if [ "$_IS_UPPER" != "" ]
then
  # нет, не все символы являются прописными
  return 1
else
  # да, все символы являются прописными
  return 0
fi
}
```

При вызове функции `is_upper` укажите строковый аргумент. На примере показано, как вызывается функция.

```
echo -n "Enter the filename :"
read FILENAME
if is_upper $FILENAME; then
  echo "Great it's upper case"
else
  echo "Sorry it's not upper case"
fi
```

Для проверки наличия в строке строчных символов просто замените существующую конструкцию `awk` в функции `is_upper` и измените имя функции на `is_lower`.

```
_IS_LOWER=`echo $1|awk '{if($0~/[^a-z]/) print "1"}`
```

Преобразование символов строки в строчные символы

В предыдущем разделе мы рассмотрели функцию `str_to_upper`, а теперь речь пойдет о функции `str_to_lower`. Вот код самой функции:

```
str_to_lower ()
# str_to_lower
# вызов: str_to_lower $1
{
# проверка на наличие корректных параметров
if [ $# -ne 1 ]; then
  echo "str_to_lower: I need a string to convert please"
  return 1
fi
echo $@ |tr '[A-Z]' '[a-z]'
}
```

Переменная LOWER хранит возвращенное значение строки, содержащей строчные символы. Обратите внимание на повторное использование специального параметра $@ для передачи всех аргументов. Функция str_to_lower может быть вызвана двумя способами. Во-первых, можно указать строку в сценарии:

```
LOWER=`str_to_lower "documents.live"`
echo $LOWER
```

Альтернативный вариант — указать аргумент для функции вместо задания строки:

```
LOWER=`str_to_upper $1`
echo $LOWER
```

Определение длины строки

Проверка результата ввода в поле является общей задачей, выполняемой сценариями. Процесс проверки может включать множество частных задач, например определение того, является ли ввод числовым или только символьным. При этом также может проверяться формат либо длина поля.

Предположим, что имеется сценарий, в рамках которого пользователь вводит данные в поле имени с помощью интерактивного экрана. В этом случае часто возникает необходимость проверить, может ли поле включать лишь определенное количество символов, например 20 символов для ввода имени персоны. Ведь пользователю не так уж трудно ввести и 50 символов в это поле. Ниже приведен код функции, осуществляющей подобную проверку. Этой функции могут быть переданы два параметра: фактическая строка и максимально возможная длина строки.

Вот сама функция:

```
check_length()
# check_length
# вызов: check_length строка максимальная_длина_строки
{
_STR=$1
_MAX=$2
# проверка на наличие корректных параметров
if [ $# -ne 2 ]; then
  echo "check_length: I need a string and max length the string should be"
  return 1
fi
# проверка длины строки
_LENGTH=`echo '$_STR |awk '{print length ($0)}'
if [ "$_LENGTH" -gt "$_MAX" ]; then
  # длина строки слишком велика
  return 1
else
  # строка имеет обычную длину
  return 0
fi
}
```

Функция check_length может быть вызвана следующим образом:

```
$ pg test_name
# !/bin/sh
# test_name
```

```
while :
do
  echo -n "Enter your FIRST name :"
  read NAME
  if check_length $NAME 10
  then
    break
    # ничего не происходит, если все условия выполнены
  else
    echo "The name field is too long 10 characters max"
  fi
done
```

Цикл продолжает выполняться до тех пор, пока данные, вводимые для переменной NAME, меньше, чем значение переменной MAX (эта переменная содержит количество разрешенных символов; в данном случае речь идет о 10 символах). Команда `break` позволяет завершить выполнение цикла.

При использовании приведенного выше фрагмента кода, вывод будет следующим:

```
$ val_max
Enter your FIRST name :Porterrrrrrrrrrrrrrr
The name field is too long 10 characters max
Enter your FIRST name :Peter
```

Команда `wc` также может применяться для определения длины строки, но имейте в виду следующий факт. При использовании команды `wc` для обработки результатов ввода с клавиатуры могут появиться проблемы. Если после ввода имени несколько раз нажать клавишу пробела, то, как правило, некоторые пробелы будут учитываться в качестве части строки. По этой причине будет определяться некорректная длина строки. Утилита `awk` "обрезает" конечные пробелы в строке при осуществлении ввода с клавиатуры (эта функция задана по умолчанию).

Ниже приведен пример, иллюстрирующий сказанное:

```
echo -n "name :"
read NAME
echo SNAME | wc -c
```

Результат выполнения описанного фрагмента сценария (здесь □ является пробелом):

```
name :Peter□□
6
```

Функция chop

Функция `chop` удаляет символы в начале строки. Этой функции передается строка; пользователь указывает, сколько символов необходимо "обрезать", начиная с первого символа. Предположим, что имеется строка `MYDOCUMENT.DOC` и требуется "обрезать" часть `MYDOCUMENT`, в результате чего функция будет возвращать только часть `.DOC`. При этом функции `chop` могут быть переданы следующие параметры:

```
MYDOCUMENT.DOC 10
```

Код функции chop:

```
chop()
# chop
# вызов:chop строка количество_обрезаемых_символов
{
_STR=$1
_CHOP=$2
# подстрока awk начинается с 0, нам потребуется прирастить ее на единицу
# для отображения того, что если пользователь задал обрезание 2 символов, 2
символа будут удалены
# а не 1
CHOP=`expr $_CHOP + 1`

# проверка на корректность параметров
if [ $# -ne 2 ]; then
  echo "check_length: I need a string and how many characters to chop"
  return 1
fi
# первоначальная длина строки
# мы не можем обрезать больше символов, чем содержится в строке!!
_LENGTH=`echo $_STR |awk '{print length ($0)}'`
if [ "$_LENGTH" -lt "$_CHOP" ]; then
  echo "Sorry you have asked to chop more characters than there are in the
string"
  return 1
fi
echo $_STR |awk '{print substr($1,'$_CHOP')}'
}
```

Возвращаемая строка, которая была "обрезана", присваивается переменной CHOPPED. Для вызова функции chop используется следующая последовательность:

```
CHOPPED=`chop "Honeysuckle" 5`
echo $CHOPPED
suckle
```

Вызов также можно осуществить другим способом:

```
echo -n "Enter the Filename :"
read FILENAME
CHOPPED=`chop $FILENAME 1`
# первый символ будет обрезан !
```

Функция months

При создании отчетов либо при отображении информации на экране часто удобным для программистов является использовании быстрого метода отображения полных наименований месяцев. Функция months в качестве аргумента использует номер месяца либо его аббревиатуру и затем возвращает полное наименование месяца.

Например, использование в качестве аргумента значения 3 либо 03 приведет к возврату значения "March". Вот описание самой функции:

```
months()
{
```

```
# months
_MONTH=$1
# проверка на наличие корректных параметров
if [ $# -ne 1 ]; then
  echo "months: I need a number 1 to 12 "
  return 1
fi

case $_MONTH in
1|01|Jan)_FULL="January";;
2|02|Feb)_FULL="February";;
3|03|Mar)_FULL="March";;
4|04|Apr)_FULL="April";;
5|05|M ay)_FULL="May";;
6|06|Jun)_FULL="June";;
7|07|Jul)_FULL="July";;R 8|08|Aug)_FULL="August;;
9|10|Sep|Sept)_FULL="September";;
10|Oct)_FULL="October";;
11|Nov)_FULL="November;;
12|Dec)_FULL="December";;
*) echo "months: Unknown month"
  return 1
  ;;
esac
echo $_FULL
}
```

Для вызова функции `months` можно применить одии из следующих методов.

```
months 04
```

В результате отобразится наименование месяца "April". Можно также вызвать функцию из сценария:

```
MY_MONTH=`months 06`
echo "Generating the Report for Month End $MY_MONTH"
...
```

В результате отобразится название месяца "June".

19.10.4. Подведение итогов

При рассмотрении примеров функций в этой главе автор не придерживался какого-либо порядка. Основная цель заключалась в том, чтобы продемонстрировать на примерах, что программный код функции не обязательно должен быть слишком сложным либо громоздким.

Многие из функций выполняют обычные задачи. Они не предназначены для осуществления "прорыва" в деле создания сценариев, а просто экономят ваше время, позволяя не вводить одинаковые фрагменты кода. Только благодаря этому использование функций, несомненно, полезно.

В начале главы было рассмотрено, каким образом функции могут применяться в интерпретаторе shell. При первом знакомстве с применением функций говорилось о значениях, возвращаемых функциями.

В приведенных примерах функций использовались различные методы для вызова и проверки результатов, возвращаемых функциями. Поэтому, если у вас возникают какие-либо проблемы в этой области, просмотрите еще раз примеры и вспомните, каким образом функции возвращают и проверяют значения.

Вам может пригодиться следующий совет. Выполняя тестирование функции, проверяйте ее действия в виде сценария, а, лишь получив удовлетворительные результаты, оформляйте сценарий в виде функции. При этом вы сэкономите время, затрачиваемое на проверку.

19.11. Вызов функций

В завершение этой главы рассмотрим два различных способа работы с функциями: вызов функций из исходного файла и применение функций, размещенных в сценариях.

19.11.1. Вызов функций, размещенных в сценариях

Чтобы использовать функцию в сценарии, ее нужно создать, затем убедиться в том, что конструкция, вызывающая эту функцию, находится после программного кода самой функции. Ниже приводится сценарий, из которого вызываются две функции. Сценарий уже рассматривался ранее; здесь осуществляется проверка существования каталога.

```
$ pg direc_check
!/bin/sh
# файл функций
is_it_a_directory()
{
# is_it_a_directory(
# вызов: is_it_a_directory имя_каталога
_DIRECTORY_NAME=$1
#if [ $# -lt 1 ]; then
  echo "is_it_a_directory: I need a directory name to check"
  return 1
fi
# это каталог?
if [ ! -d $_DIRECTORY_NAME ]; then
  return 1
else
  return 0
fi
}
#---------------------------------------
error_msg()
{
# error_msg
# сигнал; сообщение; повторный сигнал
echo -e "\007"
echo $@
echo -e "\007"
  return 0
}

### END OF FUNCTIONS
```

```
echo -n "enter destination directory :"
read DIREC
if is_it_a_directory $DIREC
then :
else
  error_msg "$DIREC does not exist...creating it now"
  mkdir $DIREC > /dev/null 2>&1
  if [ $? != 0 ]
  then
    error_msg "Could not create directory:: check it out!"
    exit 1
  else :
  fi
fi # не каталог
echo "extracting files..."
```

В верхней части данного сценария определены две функции. Вызов этих функций осуществляется из основной части сценария. Все функции должны быть помещены в верхней части сценария перед началом основных блоков. Обратите внимание, что в качестве конструкции, отображающей сообщение об ошибке, используется функция `error_msg`. Все аргументы, переданные функции `error_msg`, просто отображаются на экране. При этом раздаются два звуковых сигнала.

19.11.2. Вызов функций из файла функций

Мы уже рассматривали, каким образом функции вызываются из командной строки. Эти типы функций обычно используются утилитами, создающими системные сообщения.

А теперь воспользуемся снова описанной выше функцией, но в этом случае поместим ее в файле функций. Назовем этот файл `functions.sh`, где "sh" означает "shell scripts" (сценарии интерпретатора shell).

```
$ pg functions.sh
#!/bin/sh
# functions.sh
# основные функции
is_it_a_directory()
{
# is_it_a_directory
# вызов: is_it_a_directory имя_каталога
#
#if [ $# -lt 1 ]; then
  echo "is_it_a_directory: I need a directory name to check"
  return 1
fi
# это каталог ?
DIRECTORY_NAME=$1
if [ ! -d $DIRECTORY_NAME ]; then
  return 1
else
  return 0
fi
}

#-----------------------------------------------
```

```
error_msg()
{
echo -e "\007"
echo $@
echo -e "\007"
return 0
}
```

Создадим сценарий, вызывающий функции из файла `functions.sh`. Затем эти функции могут использоваться для выполнения каких-либо задач. Обратите внимание, что файл функций загружается с помощью следующего формата команд:

```
. /<путь к файлу>
```

При использовании этого метода не создается порожденный интерпретатор shell; все функции остаются в текущем интерпретаторе shell.

```
$ pg direc_check
!/bin/sh
# direc_check
# загрузка файла функций functions.sh
# ниже точка, пробел и косая черта
. /home/dave/bin/functions.sh

# теперь могут использоваться функции

echo -n "enter destination directory :"
read DIREC
if is_it_a_directory $DIREC
then :
else
  error_msg "$DIREC does not exist...creating it now"
  mkdir $DIREC > /dev/null 2>&1
  if [ $? != 0 ]
  then
    error_msg "Could not create directory:: check it out!"
    exit 1
  else :
  fi
fi # не является каталогом
echo "extracting files..."
```

При выполнении сценария получается тот же вывод, что и при встраивании функции в сценарий:

```
$ direc_check
enter destination directory :AUDIT
AUDIT does not exist...creating it now
extracting files...
```

19.12. Загрузка файлов, которые состоят не только из функций

Загружаемый файл не обязательно должен содержать только функции; он может включать глобальные переменные, образующие файл конфигурации.

Предположим, что существует пара сценариев резервирования, которые архивируют различные части системы. Неплохой идеей в этом случае является разделение единого файла конфигурации. Все, что нужно для этого, — создать переменные внутри файла. Затем, когда начинает выполняться один из сценариев резервирования, можно загрузить эти переменные, если потребуется изменить любые настройки, заданные по умолчанию, перед началом выполнения сценария резервирования. Это может быть связано с тем, что пользователь захочет выполнять архивирование на различных носителях.

Подобный подход может применяться сценариями, разделяющими общие параметры конфигурации, с целью выполнения процесса. Ниже приведен пример. В следующем файле конфигурации содержатся переменные среды, заданные по умолчанию. Эти переменные используются несколькими сценариями резервирования. Вот содержимое этого файла:

```
$ pg backfunc
# !/bin/sh
# name: backfunc
# конфигурационный файл содержит настройки по умолчанию для систем архивации
_CODE="comet"
_FULLBACKUP="yes"
_LOGFILE="/logs /backup/"
_DEVICE="/dev/rmt/0n"
_INFORM="yes"
_PRINT_STATS="yes"
```

Комментарии разъясняют суть программы. Первое поле, _CODE, содержит кодовое слово. Для просмотра его содержимого и изменения значений пользователь должен ввести код, соответствующий значению _CODE. В данном случае указывается слово "comet".

Ниже приведен сценарий, который в ответ на ввод пароля отображает конфигурацию, заданную по умолчанию:

```
$ pg readfunc
#!/bin/sh
# readfunc

if [ -r backfunc ]; then
  # указание файла с параметрами
  . /backfunc
else
  echo "S`basename $0` cannot locate backfunc file"
fi

echo -n "Enter the code name :"
# соответствует ли указанный код коду из файла backfunc?
if [ "${CODE}" != "${_CODE}" ]; then
  echo "Wrong code...exiting..will use defaults"
  exit 1
fi
```

```
echo " The environment config file reports"
echo "Full Backup Required            : $_FULLBACKUP"
echo "The Logfile Is                  : $_LOGFILE"
echo "The Device To Backup To is      : $_DEVICE"
echo "You Are To Be Informed by Mail  : $_INFORM"
echo "A Statistic Report To Be Printed: $_PRINT_STATS"
```

После запуска сценария на выполнение отобразится запрос на ввод кода. Если введенный код соответствует заданному кодовому слову, можно будет просмотреть настройки, заданные по умолчанию. Рабочий вариант сценария предоставит возможности для изменения настроек, заданных по умолчанию.

```
$ readfunc
Enter the code name :comet

 The environment config file reports
Full Backup Required            : yes
The Logfile Is                  : /logs/backup/
The Device To Backup To is      : /dev/rmt/0n
You Are To Be Informed by Mail  : yes
A Statistic Report To Be Printed: yes
```

19.13. Заключение

Применение функций позволяет сэкономить массу времени, затрачиваемого на разработку сценариев. Благодаря созданию универсальных и многократно используемых функций отпадает необходимость в технической поддержке разработанных сценариев, использующих эти функции.

После разработки набора требуемых функций необходимо разместить их в файле функций. В результате этого функциями смогут воспользоваться другие сценарии.

ГЛАВА 20

Передача параметров сценарию

В предыдущих главах рассматривались способы передачи параметров сценариям с помощью специальных переменных $1...$9. Специальная переменная $# указывает количество передаваемых параметров. Также обсуждалась конструкция usage. Эта конструкция применяется для информирования пользователя о том, как вызвать сценарий или функцию с помощью соответствующих параметров вызова. В этой главе будут рассмотрены следующие темы:

- применение команды shift;
- работа с командой getopts;
- примеры использования команд shift и getopts.

Для проверки степени усвоения материала рассмотрим схему сценария, в котором используются параметры start и stop. При запуске сценария необходимо указать два параметра. Если они не заданы, отображается предупреждающее сообщение. Обратите внимание, что для обработки различных параметров, передаваемых сценарию, применяется конструкция case.

```
$ pg opt
#!/bin/sh
# opt

usage()
{
echo "usage:'basename $0' start|stop process name"
}
OPT=$1
PROCESSID=$1
#if [ $# -ne 2 ]
then
  usage
  exit 1
fi
case $OPT in
start|Start) echo "Starting..$PROCESSID"
  # выполняется некоторая обработка
  ;;
stop|Stop) echo "Stopping..$PROCESSID"
  # выполняется некоторая обработка
  ;;
*) usage
  ;;
  esac
```

Приведенный сценарий при вводе данных выдает такие результаты:

```
$ opt start named
Starting..named

$ opt start
usage:opt start|stop process name
```

Общий формат произвольной команды UNIX или Linux:

```
команда опции файлы
```

Часть опции может принимать до 12 различных значений. Как показано в примере со сценарием opt, для работы с командными опциями следует создавать большой объем программного кода. В данном случае мы имеем дело лишь с двумя опциями, start и stop.

К счастью, интерпретатор команд поддерживает команду shift, с помощью которой можно выбирать различные опции. Команда shift позволяет устранить ограничение, состоящее в том, что при передаче параметров применяются только специальные переменные $1...$9.

20.1. Команда shift

При передаче сценарию параметров потребуется соответствующий метод обработки, использующий возможности команды shift. В результате выполнения этой команды позиционные аргументы смещаются влево на один элемент. Чтобы уточнить принцип действия команды shift, рассмотрим простой сценарий. Здесь применяется цикл while, обеспечивающий отображение на экране всех аргументов, переданных сценарию.

```
$ pg opt2
#!/bin/sh
# opt2
loop=0
while [ $# -ne 0 ] # цикл выполняется до тех пор, пока остаются аргументы
do
  echo $1
done
```

Создается впечатление, что указанный сценарий будет выполняться до тех пор, пока в командной строке не останется аргументов. К сожалению, это не так. При запуске сценария на экран выводится только первый аргумент, поскольку в сценарии не предусмотрен переход к следующему параметру. Вот результат выполнения вышеприведенного сценария:

```
$ opt2 файл1 файл2 файл3
файл1
файл1
файл1
...
```

20.1.1. Простой способ использования команды shift

Для обработки каждого передаваемого аргумента достаточно воспользоваться командой `shift`. Ниже приводится соответствующий сценарий:

```
$ pg opt2
#!/bin/sh
# opt2
loop=0
while [ $# -ne 0 ] # цикл выполняется до тех пор, пока остаются аргументы
do
  echo $1
  shift
done
```

Теперь, при выполнении этого сценария можно получить более приемлемый результат:

```
$ opt2 файл1 файл2 файл3
файл1
файл2
файл3
```

20.1.2. Последний параметр командной строки

Несмотря на то что команда `eval` еще не обсуждалась, можно воспользоваться ею, если требуется уточнить последний параметр командной строки (обычно в качестве этого параметра используется имя файла). Получить последний параметр командной строки вы можете двумя способами. Во-первых, с помощью команды `eval echo \$$#`, а во-вторых, путем применения команды `shift `expr$# - 2``.

20.1.3. Преобразования файла с помощью команды shift

Использование возможностей команды `shift` значительно облегчает работу с опциями командной строки. Рассмотрим сценарий, выполняющий преобразование регистра символов. При этом будет применяться команда `tr`.

При выполнении сценария используются две опции:

`-l` для нижнего регистра
`-u` для верхнего регистра

С помощью команды `shift` можно запустить на выполнение сценарий, работающий с опциями `-l` и `-u`. Ниже приводится первый вариант этого сценария.

```
$ pg tr_case
!/bin/sh
# tr_case
# преобразование регистра
usage()
{
# сообщение usage
echo "usage:`basename $0` -[l|u] file [files]" >&2
exit 1
}
if [ $# -eq 0 ]; then
```

```
  # параметры не переданы!
  usage
fi

while [ $# -gt 0 ]
do
  case $1 in
  -u|-U) echo "-u option specified"
    # здесь укажите все установки переменных для нижнего регистра, затем
    # примените команду shift
    shift
    ;;
  -l|-L) echo "-l option specified"
    # здесь укажите все установки переменных для верхнего регистра, затем
    # примените команду shift
    shift
    ;;
  *) usage
    ;;
  esac
done
```

Вначале выполняется проверка, имеются ли аргументы в сценарии. Если они отсутствуют, выводится сообщение `usage`. При обработке аргументов с помощью конструкции `case` отображается значение каждой передаваемой опции. После этого применяется команда `shift`, выполняющая вывод в командной строке очередной опции. Если все опции обработаны, отображается сообщение `usage`.

А теперь посмотрим, как выглядят выводимые результаты при передаче сценарию двух некорректных аргументов.

```
$ tr_case -u -l -k
-u option specified
-l option specified
usage:tr_case -[l|u] file [files]
```

На следующем этапе обрабатываются файлы, передающиеся после обработки опций с помощью конструкции `case`. Для реализации этой задачи достаточно выполнить небольшие изменения. В конструкции `case` шаблон `*` заменяется шаблоном `-*`, что позволяет передавать некорректные опции, например `-p` или `-q`.

Чтобы с помощью цикла `for` обрабатывать каждый файл, следует для всех передаваемых имен файлов указывать шаблон `*`. Кроме того, с помощью опции `-f` проверяется наличие всех требуемых имен файлов.

Итак, улучшенный вариант конструкции `case` имеет следующий вид:

```
case
...
-*) usage
  ;;
*)  if [ -f $1 ]; then
      FILES=$FILES" " $1 # присвоить переменной имена файлов
    else
      echo "`basename $0` cannot find the file $1"
    fi
  shift # получите следующий параметр !
  ;;
esac
```

Следует также задать значения некоторых переменных в зависимости от указанной опции (-l, -u). При этом используются следующие переменные:

TRCASE	Указывает тип преобразования регистра (верхний или нижний регистр)
EXT	Все преобразованные файлы имеют либо расширение *.UC* (для верхнего регистра), либо расширение *.LC* (для нижнего регистра). В исходный файл изменения не вносятся
OPT	При передаче опций принимает значение yes, в противном случае — no. Только тогда, когда опции не передаются, можно перехватить их значение и вывести на экран сообщение

Для выполнения реального преобразования достаточно добавить еще один фрагмент кода, включающий команду tr. Этот фрагмент кода добавляется в конец сценария. При этом для считывания имен файлов используется цикл for и конструкция case. Ниже приводится код созданного сценария:

```
$, pg tr_case
!/bin/sh
# tr_case
# преобразование символов файлов либо в верхний, либо в нижний регистр
FILES=""
TRCASE=""
EXT=""
OPT=no

# вызывается при неудачном преобразовании
error_msg()
(
_FILENAME=$1
echo "`basename $0`: Error the conversion failed on $_FILENAME"
)

if [ $# -eq 0 ] then
  echo "For more info try `basename $0` --help"
  exit 1
fi
while [ $# -gt 0 ]
do
  case $1 in
  # установите переменные на базе применяемой опции
  -u) TRCASE=upper
    EXT=".UC"
    OPT=yes
    shift
    ;;
  -l) TRCASE=lower
    EXT=".LC"
    OPT=yes
    shift
    ;;
  -help) echo "convert a file(s) to uppercase from lowercase"
         echo "convert a file(s) from lowercase to uppercase"
         echo "will convert all characters according to the"
         echo " specified command option."
         echo " Where option is"
```

```
        echo " -l Convert to lowercase"
        echo " -u Convert to uppercase"
        echo " The original file(s) is not touched. A new file(s)"
        echo "will be created with either a .DC or .LC extension"
        echo "usage: $0 -[l|u] file [file..]"
      exit 0
      ;;
    -*) echo "usage: `basename $0` -[l|u] file [file..]"
      exit 1
      ;;

    *) # сбор файлов для обработки
      if [ -f $1 ]
      then
        # добавьте имена файлов в список переменных
        FILES=$FILES" "$1
      else
        echo "`basename $0`: Error cannot find the file $1"
      fi
      shift
      ;;
    esac

done
# опции не заданы ... помогите пользователю
if [ "$OPT" = "no" ]
then
  echo "`basename $0`: Error you need to specify an option. No action taken"
  echo " try `basename $0` --help"
  exit 1
fi
# просмотр всех файлов
# используется переменная LOOP, такое красивое слово LOOP
for LOOP in $FILES
do
  case $TRCASE in
  lower) cat $LOOP|tr "[a-z]" "[A-Z]" >$LOOP$EXT
    if [ $? != 0 ]
    then
      error_msg $LOOP
    else
      echo "Converted file called $LOOP$EXT"
    fi
    ;;
  upper) cat $LOOP|tr "[A-Z]" "[a-z]" >$LOOP$EXT
    if [ $? != 0 ]
    then
      error_msg $LOOP
    else
      echo "Converted file called $LOOP$EXT"
    fi
    ;;
  esac
done
```

Если выполнять указанный сценарий с различными опциями в качестве входных данных, получим следующие результаты.

При передаче имени несуществующего файла:

```
$ tr_case -k cursor
usage: shift1 -[l|u] file [file..]
```

При передаче некорректных опций:

```
$ tr_case cursor
tr_case:Error you need to specify an option. No action taken
try tr_case -help
```

Если просто указать имя файла в надежде на ответную помощь сценария, результатом будет:

```
$ tr_case
For more info try tr_case -help
```

Если указать два имени существующих файлов, а третье — имя несуществующего файла, получим следующие результаты:

```
$ tr_case -l cursor sd ascii
tr_case: Error cannot find the file sd
Converted file called cursor.LC
Converted file called ascii.LC
```

С помощью указанного сценария можно осуществить преобразование к одному и тому же регистру символов из нескольких файлов. Сценарий, который сможет обрабатывать различные опции командной строки, будет иметь большой размер.

Предположим, что существует сценарий, выполняющий обработку различных опций командной строки:

```
command -l -c 23 -v файл1 файл2
```

Здесь нельзя применить команду `shift`; вместо нее следует воспользоваться командой `getopts`.

20.2. Команда getopts

Применение команды `getopts` обеспечивает создание программного кода, который без труда справляется с несколькими аргументами командной строки. Благодаря использованию этой команды процесс обработки командной строки приводится в соответствие с некоторым стандартом. Ведь сценарии должны соответствовать стандартному формату файлов командных опций.

20.2.1. Пример сценария, использующего команду getopts

С командой `getopts` лучше ознакомиться на основе примеров. Ниже приводится простой сценарий `getopts`, использующий следующие параметры, или аргументы:

- `-a` Переменной `ALL` присваивается значение "истина"
- `-h` Переменной `HELP` присваивается значение "истина"
- `-f` Переменной `FILE` присваивается значение "истина"
- `-v` Переменной `VERBOSE` присваивается значение "истина"

Как обычно, при всех установках переменных всегда предполагается худший вариант, поэтому изначально переменным присваивается значение "ложь":

```
$ pg getopt1
!/bin/sh
#getopt1

# присвоение значений переменным
ALL=false
HELP=false
FILE=false
VERBOSE=false

while getopts ahfgv OPTION
do
  case $OPTION in
  a)ALL=true
    echo "ALL is $ALL"
    ;;
  h)HELP=true
    echo "HELP is $HELP"
    ;;
  f)FILE=true
    echo "FILE is $FILE"
    ;;
  v)VERBOSE=true
    echo "VERBOSE is SVERBOSE"
    ;;
  esac
done
```

Общий формат команды `getopts`:

```
getopts строка_параметров переменная
```

А теперь используем код из нашего примера:

```
while getopts ahfgv OPTION
```

Нетрудно заметить, что цикл `while` применяется для считывания в командной строке. Параметр `строка_параметров` включает пять указанных опций (`-a`, `-h`, `-f`, `-g`, `-v`), а также переменную, которая в данном примере именуется `OPTION`. Заметьте, что не требуется при определении каждой одиночной опции указывать дефис.

При выполнении сценария с корректными и некорректными опциями получаются следующие результаты:

```
$ getopt1 -a -h
ALL is true
HELP is true

$ getopt1 -ah
ALL is true
HELP is true

$ getopt1 -a -h -p
```

```
ALL is true
HELP is true
./getopt1: illegal option -- p
```

Обратите внимание, что возможно комбинирование различных опций.

20.2.2. Принцип работы команды getopts

Команда `getopts` считывает строку `строка_параметров`. При этом она выбирает корректные опции, которые могут быть применены в сценарии.

Команда `getopts` разыскивает все аргументы, начинающиеся дефисом, и определяет значения всех опций. Затем значение опции сравнивается со строкой `строка_параметров`. Если соответствие установлено, переменной присваивается значение `OPTION`. В противном случае переменной присваивается значение `?`. Этот процесс продолжается до тех пор, пока не будут обработаны все опции.

Завершив обработку всех аргументов, команда `getopts` возвращает ненулевое состояние. Это означает, что все аргументы были переданы. Переменная `OPTIND` содержит значение последнего обработанного аргумента. В следующем разделе мы рассмотрим, какую пользу эта переменная может принести при обработке аргументов.

20.2.3. Указание значений опций с помощью команды getopts

Иногда для сценариев требуется включение фактического значения одной из опций командной строки. При этом используется команда `getopts`. Все, что требуется для этого сделать, — вставить двоеточие после буквы опции параметра `строка_параметров`. Например:

```
getopts ahfvc: OPTION
```

Эта команда определяет передачу опций `a`, `h`, `f`, `v` без указания значений, но опция `c` должна иметь значение. После указании значения оно будет присвоено переменной `OPTARG`. Если попытаться передать данную опцию без этого значения, отобразится сообщение об ошибке. Стандартное сообщение об ошибке не является особо информативным, поэтому "подавите" его отображение и выполните следующее:

Укажите двоеточие перед параметром `строка_параметров`.

```
while getopts :ahfgvc: OPTION
```

Используйте оператор `usage` внутри конструкции `case`. При этом применяется символ `?`, выполняющий функции перехвата ошибок.

```
case
...
...
\?) # оператор usage
  echo "`basename $0` -[a h f v] -[c value] file"
;;
esac
```

Ниже представлен измененный сценарий `getopt1`:

```
$ pg getopt1
#!/bin/sh
#getopt1
```

```
# установка значений переменных
ALL=false
HELP=false
FILE=false
VERBOSE=false
COPIES=0 # значение опции -c равно нулю

while getopts :ahfgvc: OPTION
do
  case $OPTION in
  a)ALL=true
    echo "ALL is $AIL"
    ;;
  h)HELP=true
    echo "HELP is $HELP"
    ;;
  f)FILE=true
    echo "FILE is $FILE"
    ;;
  v)VERBOSE=true
    echo "VERBOSE is $VERBOSE"
    ;;
  c)COPIES=$OPTARG
    echo "COPIES is $COPIES"
  \?) # оператор usage
    echo "`basename $0` -[a h f v] -[c value] file" >&2
    ;;
  esac
done
```

При выполнении указанного выше сценария с опцией `-c`, не содержащей значения, возникает ошибка. В этом случае отображается сообщение `usage`:

```
$ getopt1 -ah -c
ALL is true
HELP is true
getopt1 -[a h f v] -[c value] file
```

Теперь указываются все допустимые опции:

```
$ getopt1 -ah -c 3
ALL is true
HELP is true
COPIES is 3
```

20.2.4. Доступ к значениям

Команда `getopts` часто применяется для выполнения сценариев резервирования. Благодаря этому пользователь может указывать различные ленточные накопители, используемые для резервирования данных. Ниже приводится образец сценария, использующего команду `getopts`.

```
$ pg backups
#!/bin/sh
# backups
```

```
QUITE=n
DEVICE=awa
LOGFILE=/tmp/logbackup
usage()
{
echo "Usage: `basename $0` -d [device] -l [logfile] -q"
exit 1
}
if [ $# = 0 ]
then
  usage
fi

while getopts :qd:l: OPTION
do
  case $OPTION in
  q) QUIET=y
    LOGFILE="/tmp/backup.log"
    ;;
  d) DEVICE=$OPTARG
    ;;
  l) LOGFILE=$OPTARG
    ;;
  \?) usage
    ;;
  esac
done
echo "you chose the following options..I can now process these"
echo "Quite= $QUITE $DEVICE $LOGFILE"
```

В данном сценарии при указании опции d нужно присваивать значение. Это значение представляет собой наименование пути для ленточного накопителя. Пользователь может также определить, создавать ли резервную копию, если весь вывод направляется в журнальный файл. Выполнение сценария, использующего указанные ниже входные данные, приводит к следующим результатам:

```
$ backups -d/dev/rmt0 -q
you chose the following options..I can now process these
Quite= y /dev/rmt0 /tmp/backup.log
```

После того как команда getopts завершит выполнение проверки, значения, присвоенные OPTARG, могут быть использованы в процессе обычной обработки. Конечно, если в сценарии имеются опции, они должны быть установлены для дальнейшей обработки и проверки значений.

Вот и все, что требуется знать о том, каким образом команда getopts обеспечивает передачу параметров командной строки.

Для фактической обработки файлов используется цикл for, подобно тому, как это было в сценарии tr_case, где применялась команда shift для работы с опциями.

Применение команды getopts позволяет радикально сократить объем создаваемого кода по сравнению с использованием метода shift.

20.2.5. Использование команды getopts для преобразования файлов

А теперь воспользуемся сценарием tr_case, который преобразуем с помощью только что изученной команды getopts. Существует единственное отличие между методами getopts и shift, применяемыми для обработки опций командной строки. Это отличие заключается в том, что в первом случае используется опция VERBOSE.

Переменная VERBOSE имеет значение "no", заданное по умолчанию; но при перехвате значения опции командной строки с помощью конструкции case переменной VERBOSE присваивается значение "yes". Отображение команд на экране осуществляется с помощью простой конструкции if.

```
if [ "VERBOSE" = "on" ]; then
  echo "doing upper on $LOOP..newflie called $LOOP$EXT"
fi
```

Если применяется оболочка для системных команд, которые всегда отображают результаты своих действий, то вывод, включающий произвольные ошибки, просто перенаправляется в */dev/null*.

```
command > /dev/null 2 > &1
```

По умолчанию переменная VERBOSE не установлена (нет отображения). Активизировать эту переменную можно посредством опции -v. Например, для преобразования серии файлов *myfiles* в символы нижнего регистра с помощью VERBOSE применяется следующий формат:

```
tr_case -l -v myfilel myfile2...
```

либо

```
tr_case -v -l myfilel myfile2...
```

Сразу же бросается в глаза заметное сокращение объема программного кода при использовании команды getopts. Код, применяемый для обработки файлов, аналогичен коду с командой shift.

Пример сценария:

```
$ pg tr_case2
#!/bin/sh
#tr_case2
# преобразование регистра, используется команда getopts
EXT=""
TRCASE=""
FLAG=""
OPT="no"
VERBOSE="off"

while getopts :luv OPTION
do
  case $OPTION in
  l) TRCASE="lower"
    EXT=".LC"
    OPT=yes
    ;;
  u) TRCASE="upper"
    EXT=".UC"
    OPT=yes
```

```
    ;;
  v) VERBOSE=on
    ;;
  \?) echo "usage: `basename $0`: -[l|u] --v file[s]"
    exit 1 ;;
  esac
done
# следующий аргумент, пожалуйста
shift `expr $OPTIND - 1`
# есть аргументы ???
if [ "$#" = "0" ] || [ "$OPT" = "no" ]
then
  echo "usage: `basename $0`: -[l|u] -v file[s]" >&2
  exit 1
fi
for LOOP in "$@"
do
  if [ ! -f $LOOP ]
  then
    echo "`basename $0`: Error cannot find file $LOOP" >&2
    exit 1
  fi
  echo $TRCASE $LOOP
  case $TRCASE in
  lower) if [ "VERBOSE" = "on" ]; then
           echo "doing..lower on $LOOP..newfile called $LOOPSEXT"
         fi
    cat $LOOP | tr "[a-z]" "[A-Z]" >$LOOP$EXT
    ;;
  upper) if [ "VERBOSE" = "on" ]; then
           echo "doing upper on SLOOP..newflie called $LOOP$EXT"
         fi
    cat $LOOP | tr "[A-Z]" "[a-z]" >$LOOP$EXT
    ;;
  esac
done
```

При указании опций командной строки в сценариях было бы неплохо придерживаться соглашений о наименовании, принятых в UNIX либо Linux. В следующей таблице описываются некоторые общие опции и их значения.

Опция	Значение
-a	добавление
-c	счетчик, копирование
-d	каталог, устройство
-e	выполнение
-f	имя файла, форсировать
-h	справка
-i	игнорировать регистр
-l	журнальный файл
-o	полный вывод
-q	полностью
-p	путь
-v	многословный

20.3. Заключение

Способность корректно обрабатывать опции командной строки придает сценариям профессиональный вид. Для пользователя эти опции выглядят точно так же, как любые другие системные команды. В этой главе были рассмотрены два метода, предназначенные для обработки опций командной строки, — `shift` и `getopts`. Размер программного кода, необходимого для внедрения метода `getopts`, является намного меньшим, чем в случае использования метода `shift`.

Метод `shift` также позволяет преодолеть ограничения параметров `$1...$9`, проявляющиеся при их передаче сценариям. При использовании метода `shift` сценарий просто выполняет "смещение" среди всех вызываемых аргументов, благодаря чему можно выполнять дальнейшую обработку.

ГЛАВА 21

Создание экранного вывода

С помощью shell-сценариев можно создавать профессионального вида экраны, позволяющие реализовать интерактивное взаимодействие пользователя с системой. Для этого достаточно располагать цветным монитором и использовать команду `tput`.

В главе рассматриваются следующие темы:

- применение команды `tput`;
- использование escape-последовательностей и генерирование управляющих кодов;
- работа с цветом.

Известно, что существует три различных варианта команды `tput`. Наилучшим из них является команда `tput GNU`. Если в системе отсутствует эта версия команды, загрузите и установите ее. Команда `tput` использует файл */etc/terminfo* или файл */etc/termcap*. В shell-сценариях можно применять большинство команд, поддерживаемых терминалом.

Команда `tput` не распознает настройки цвета. Для работы с цветом используются управляющие символы.

21.1. Применение команды tput

Чтобы применить команду `tput`, следует инициализировать установки терминала, обращаясь к команде `tput` с помощью сценариев или командной строки.

```
$ tput init
```

Команда `tput` генерирует три различных потока вывода: строки, числовые и булевы значения (истина/ложь). Далее будут рассмотрены некоторые наиболее часто используемые свойства каждого потока вывода.

21.1.1. Строчный поток вывода данных

Ниже приводятся общие строки из потока вывода:

Название	Значение
`bel`	Звуковой сигнал
`blink`	Режим мерцания
`bold`	Двойная интенсивность
`civis`	Скрыть курсор
`clear`	Очистка экрана

cnorm	Отобразить курсор
cup	Перемещение курсора на экране в позицию x, y
el	Очистка до конца строки
el1	Очистка к началу строки
smso	Переход в режим отступа
rmso	Выход из режима отступа
smul	Переход в режим подчеркивания
rmul	Выход из режима подчеркивания
sc	Сохранение текущего положения курсора
rc	Восстановление последней позиции курсора
sgr0	Обычный экран
rev	Обратное видео

21.1.2. Числовой вывод

Наиболее распространенный числовой вывод:

Название	Значение
cols	Количество столбцов
it	Настройка табуляции
lines	Количество строк на экране

21.1.3. Поток вывода булевых данных

Команда tput включает незначительное количество булевых операторов.

Название	Значение
chts	Курсор трудно заметить
hs	Имеет строку статуса

21.2. Работа с командой tput

Рассмотрим наиболее распространенные разновидности команды tput, а также методы ее использования в сценариях.

21.2.1. Присвоение имен командам tput

Можно использовать поток вывода всех имен команды tput, присваивая их переменным с более осмысленными наименованиями. При этом применяется следующий формат:

```
имя_переменной = 'tput name'
```

21.2.2. Применение булевого потока вывода

Чтобы применить булев поток вывода для команды `tput`, воспользуйтесь конструкцией `if`:

```
STATUS_LINE='tput hs'
if $STATUS_LINE; then
  echo "your terminal has a status line"
else
  echo "your terminal has NO status line"
fi
```

21.2.3. Использование команды tput в сценариях

В приведенном сценарии командам `tput bel` и `cl` присваиваются более значимые имена.

```
$ pg tput1
#!/bin/sh
BELL='tput bel'
CLEAR='tput cl'

echo $BELL
echo $CLEAR
```

В следующем сценарии изменяются несколько видеоатрибутов и происходит отображение и сокрытие курсора:

```
$ pg tput2
#!/bin/sh
BOLD='tput bold'
REV='tput rev'
NORMAL='tput sgr0'
CURSOR_OFF='tput civis'
CURSOR_ON='tput cnorm'
tput init

# сокрытие курсора, выделение текста, перестановка текста, отображение курсора
echo $CURSOR_OFF
echo "${BOLD} WELCOME TO THE PIZZA PLACE${NORMAL}"
echo -e "\n${REV} WE ARE OPEN 7 DAYS A
WEEK${NORMAL}"
echo $CURSOR_ON
```

21.2.4. Генерирование escape-последовательностей

Обратите внимание, что при использовании эмулятора довольно затруднительно скрыть курсор. Это обусловлено несколькими причинами:

1. Некоторые эмуляторы не воспринимают управляющий символ, скрывающий курсор. Желательно, чтобы разработчики программного обеспечения для эмуляторов учли это замечание. В исходный код потребуется внести изменения, реализующие сокрытие курсора.
2. Существует мнение, что некоторые более ранние версии команды `tput civis` не функционируют должным образом.

Управляющим символом, используемым для сокрытия курсора, является ?251 (буква l). Для возврата в исходное положение применяется символ ?25h.

Все управляющие символы выполняются с помощью escape-последовательности. Обычно за кодом клавиши [Esc] следует символ [. Затем управляющая последовательность подключает или отключает определенный атрибут терминала.

Для генерирования escape-последовательностей можно воспользоваться двумя различными методами. В таблице ниже приводятся оба метода, которые зависят от имеющейся системы. Третий метод можно применить независимо от того, используется система UNIX или Linux, поскольку управляющая последовательность реализована в составе конструкции echo. Именно третий метод и применяется в книге.

Чтобы переслать escape-последовательность и отключить курсор, используется следующий код:

Linux/BSD	`echo-e "\033[?251"`
System V	`echo "\033[?251"`
Обобщенный метод	`echo "<CTRL-V><ESCAPE>[?251"`

Клавише [Esc] соответствует код \033. Символ \ указывает команде echo, что далее следует восьмеричное значение. Например, для вывода на экран символа @ можно применить команду:

```
echo "@"
```

Или же для вывода на экран этого символа можно воспользоваться восьмеричным значением символа, которое равно 100.

```
echo -e "\100"
```

Для System V примените команду:

```
echo "\100"
```

Результат будет аналогичным.

Команда clear очищает экран и устанавливает курсор в верхнем левом углу экрана. Это положение курсора обычно называется home. При работе с терминалами, относящимися к семейству VT, эту процедуру выполняет последовательность ESC [2J. Данную последовательность можно отправить с помощью конструкции echo.

System V	`echo "\033[2J"`
LINUX/BSD	`echo-e "\033[2J"`

При этом следует придерживаться тех же правил, которыми руководствуются при работе с управляющими символами, помещенными в текстовые файлы. Не следует применять методику "вырезать и вставить", поскольку в этом случае будет утеряно специальное значение символов. Чтобы вставить управляющие символы, отображающие курсор, воспользуйтесь следующим кодом:

```
echo '<CTRL-V> hit the <ESCAPE> key then [?25h'
```

Здесь даны указания о том, что следует воспользоваться комбинацией клавиш [Ctrl+V], затем нажать клавишу [Esc] и после этого ввести символы [?25h.

Если при использовании команды tput civis курсор не исчезает и при этом не запущен эмулятор, воспользуйтесь следующим небольшим сценарием. С его помощью можно скрыть либо отобразить курсор. Вы можете подробно изучить приведенную

функцию либо перевернуть пару страниц и сразу ознакомиться с результатами выполнения сценария.

```
$ pg cursor
#!/bin/sh

# отображение|сокрытие курсора
# отображает или скрывает курсор при работе с терминалами vt100, 200, 220, meth220
# замечание: функционирует при нормальном tty-соединении при использовании
# некоторых win-эмуляций
# проверьте TERM env для вашего типа !
_OPT=$1
if [ $# -ne 1 ]; then
  echo "Usage:`basename $0` cursor [on|off]"
  exit 1
fi

case "$_OPT" in
on|ON|On)
  # отображение курсора
  ON=`echo ^[[?25h`
  echo $ON
  ;;
off[OFF|Off)
  # сокрытие курсора
  OFF=`echo ^[[?25l`
  echo $OFF
  ;;
;;
*)echo "Usage: cursor on|off"
  exit 1
  ;;
esac
```

21.2.5. Изменение положения курсора

Команду `tput` также можно применять для отображения курсора в произвольном месте экрана. При этом используется следующий формат:

```
cup r c
```

где `r` — это номер ряда (строки) в нижней части экрана, а `c` — номер столбца на экране.

Лучше применять эту команду в форме функции, тогда можно указывать значения для строки и столбца.

```
xy()
{
#_R= row, _C=column
_R=$1
_C=$2
tput cup $_R $_C
}
```

```
clear
xy 1 5
echo -n "Enter your name :"
read NAME
xy 2 5
echo -n "Enter your age :"
read ACE
```

Конечно, желательно передавать строку для отображения; ниже приводится небольшая модификация этого сценария.

```
xy()
{
#_R= row, _C=column
_R=$1
_C=$2
_TEXT=$3
tput cup $_R $_C
echo -n $_TEXT
}
```

Сценарий можно вызвать следующим образом:

```
xy 5 10 "Enter your password :"
read CODE
```

21.2.6. Центрирование отображаемого текста

При центрировании текста на экране нужно быть предельно внимательным. С помощью команды `tput` необходимо получить столбцы, затем получить значение для длины строки, исключить это значение из столбцов `tput` и разделить ответ на две части. В дальнейшем нужно только указать номер строки, где отобразится данная строка.

Ниже приводится часть кода, который выполняет эту задачу. Чтобы просмотреть строки файла и центрировать на экране весь текст, достаточно выполнить небольшое изменение.

Введите символы, нажмите клавишу [Return], и текст отобразится в середине экрана начиная со строки 10.

```
echo -n "input string :"
read STR
# быстрый способ вычисления длины строки
LEN=`echo $STR | we -c`
COLS=`tput cols`
NEW_COL=`expr \($COLS - $LEN \) / 2`
xy 10 $NEW_COL
echo $STR
```

Чтобы указанная функция отличалась большей гибкостью, при ее вызове можно использовать текст и номер строки. Функция будет иметь следующий вид:

```
centertxt()
{
_ROW=$1
_STR=$2
```

```
# быстрый способ получения длины строки
LEN=`echo $_STR | wc -c`
COLS=`tput cols`
_NEW_COL=`expr \($COLS -- $LEN \) / 2`
xy $_ROW $_NEW_COL
echo $_STR
}
```

Чтобы вызвать функцию, следует применить команду:

```
centertxt 15 "THE MAIN EVENT"
```

или аналогичную команду, но с использованием строки в качестве аргумента:

```
centertxt 15 $1
```

21.2.7. Определение атрибутов терминала

Рассмотрим сценарий, в котором с помощью команды `tput` производится обращение к базе данных `terminfo`. С помощью некоторых команд `tput` отображаются управляющие коды терминала.

```
$ pg termput
#!/bin/sh
#termput

#инициируйте tput для терминала
tput init

clear

echo " tput <> terminfo"
infocmp -1 $TERM | while read LINE
do
  case $LINE in
  bel*) echo "$LINE: sound the bell" ;;
  blink*) echo "$LINE: begin blinking mode" ;;
  bold*) echo "$LINE: make it bold" ;;
  el*) echo "$LINE: clear to end of line" ;;
  civis*) echo "$LINE: turn cursor off";;
  cnorm*) echo "$LINE: turn cursor on ";;
  clear*) echo "$LINE: clear the screen ';;
  kcuu1*) echo "$LINE: up arrow ";;
  kcub1*) echo "SLINE: left arrow ";;
  kcuf1*) echo "$LINE: right arrow ";;
  kcud1*) echo "SLINE: down arrow ";;
  esac
done
```

Команда `infocmp` извлекает из файла базы данных `terminfo` информацию о терминале. Если нужно просмотреть файл, содержащий сведения о настройках терминала, выполните команду:

```
$ infocmp $TERM
```

Ниже с помощью сценария termput отображается поток вывода для некоторого терминала:

```
$ termput
tput <> terminfo
bel=^G,: sound the bell
blink=E[5m,: begin blinking mode
bold=E[1m,: make it bold
civis=E[?25l,: turn cursor off
clear=E[HE[J,: clear the screen
cnorm=E[?25h,: turn cursor on
el=E[K,: clear to end of line
el1=E[1K,: clear to end of line
kcub1=E[D,: left arrow
kcud1=E[B,: down arrow
kcuf1=E[C,: right arrow
kcuu1=E[A,: up arrow
```

21.2.8. Применение функциональных клавиш при работе со сценариями

С помощью команды cat можно обращаться к специальным клавишам ([F1], [↑] и т.д.). Введите команду cat -v, затем нажмите любую управляющую клавишу и просмотрите, что отобразится в нижней строке. Когда просмотр завершится, нажмите комбинацию клавиш [Ctrl+C].

В следующем примере вызывается команда cat и используются клавиши [F1] (^[OP), [F2] (^[OQ) и [↑] (^[[A).

```
$ cat -v
^[OP^[OQ^[[A
<CTRL-C>
```

Эта информация позволяет применять рассмотренные символы в сценариях в качестве дополнительных методов, обеспечивающих интерактивный режим работы пользователя.

В приведенном ниже сценарии распознаются клавиши [F1], [F2], а также клавиши стрелок. Пользовательские значения могут быть иными, поэтому выполните команду cat, чтобы уточнить, какие значения соответствуют управляющим клавишам терминала.

```
$ pg control_keys
#!/bin/sh
# управляющие_клавиши
# для вставки примените последовательность '<CTRL-V><ESCAPE>sequence'
uparrowkey='^[[A'
downarrowkey='^[[B'
leftarrowkey='^[[D'
rightarrowkey='^[[C'
f1key='^[OP'
f2key='A[OQ'

echo -n " Press a control key then hit return"
read KEY

case $KEY in
```

```
$uparrowkey) echo "UP Arrow"
$downarrowkey) echo "DOWN arrow"
  ;;
$leftarrowkey) echo "LEFT arrow"
  ;;
$rightarrowkey) echo "RIGHT arrow"
  ;;
$f1key) echo "F1 key"
  ;;
$f2key) echo "F2 key"
  ;;
*) echo "unknown key $key"
  ;;
esac
```

21.2.9. Применение различных цветов

Благодаря применению цвета можно придать экрану, где отображается поток ввода, более привлекательный вид. При работе с цветами используется стандарт ANSI. Однако не все цвета можно применить во всех системах. Предлагаем перечень наиболее часто употребляемых цветов.

Цвета переднего плана

Значение	Цвет
30	черный
31	красный
32	зеленый
33	желтый (или коричневый)
34	голубой
35	пурпурный
36	синий (циан)
37	белый (или серый)

Фоновые цвета

Значение	Цвет
40	черный
41	красный
42	зеленый
43	желтый (или коричневый)
44	голубой
45	пурпурный
46	синий (циан)
47	белый (или серый)

Для отображения цветов переднего плана и фоновых цветов применяется следующий формат:

```
<ESCAPE>[значение_фона; значение_переднего_плана m
```

21.2.10. Генерирование цветов

Для генерирования цвета управляющие символы встраиваются в конструкцию `echo`. Данный метод применяется при работе с цветным терминалом и произвольной системой. Как и в случае с управляющими символами, цвета можно генерировать с помощью escape-последовательностей, встраиваемых в конструкцию `echo`. Для создания черного фона с передним планом зеленого цвета воспользуйтесь командой:

Linux/BSD	`echo -e "\033 [40; 32m"`
System V	`echo "\033 [40; 32m"`
Обобщенный метод	`echo "<CTRL-V><ESCAPE>[40;32m"`

При использовании обобщенного метода, т.е. комбинации клавиш [Ctrl+V], нажмите клавишу [Esc], затем введите символы `[40; 32m`. Обобщенный метод и применяется далее в книге.

Возможно, лучше поместить конструкции `echo`, отвечающие за воспроизведение цвета, в конструкцию `case`, а затем оформить все это в виде функции. Ниже приводится функция цвета `case`.

```
colour()
{
# формат цвет_фона;цвет_переднего_планам
case $1 in
black_green)
  echo '^[[40;32m'
  ;;
black_yellow)
  echo '^[[40;33m'
  ;;
black_white)
  echo '^[[40;37m'
  ;;
black_cyan)
  echo '^[[40;36m'
  ;;
red_yellow)
  echo '^[[41;33m'
  ;;
black_blue)
  echo '^[[40;34m'
  ;;
esac
}
```

Чтобы задать красный цвет фона и желтый цвет переднего плана, примените команду:

```
colour red_yellow
```

Для использования цветов в сценариях выполните следующие действия:

```
colour what_ever
echo something
# измените на другой цвет
colour what_ever
echo something
```

Обычно по умолчанию в качестве цветов экрана используются черный и белый. Рассмотрим, как изменить эту установку, чтобы фоновым цветом был черный, а цветом переднего плана — зеленый. С этой целью в файл *.profile* достаточно добавить конструкцию `echo`, которая создает требуемую комбинацию.

Ниже приводится пример кода для управления экраном.

```
$ pg colour_scr
#!/bin/sh
# colour_scr
tput init
MYDATE='date +%D'
colour()
{
# формат цвет_фона;цвет_переднего_планам
case $1 in
black_green)
  echo '^[[40;32m'
  ;;
black_yellow)
  echo '^[[40;33m'
  ;;
black_white)
  echo '^[[40;37m'
  ;;
black_cyan)
  echo '^[[40;36m'
  ;;
black_red)
  echo '^[[40;31m'
  ;;
esac
}

xy()
#xy
# для вызова: xy строка,столбец,"text"
# переход к координатам xy на экране
{
#_R= row, _C=column
_R=$1
_C=$2
_TEXT=$3
tput cup $_R $_C
echo -n $_TEXT
}
```

```
center(
{
# center
# центрирование строки текста на экране
# для вызова: center "строка" номер_строки
_STR=$1
_ROW=$2
# неудачный способ получения длины строки
LEN=`echo $_STR | wc -c`

COLS=`tput cols`
HOLD_COL=`expr $COLS - $LEN`
NEW_COL=`expr $HOLD_COL / 2`
tput cup $_ROW $NEW_COL
echo -n $_STR
}

tput clear
colour red_yellow
xy 2 3 "USER: $LOGNAME"
colour black_cyan
center "ADD A NEW WARP DRIVE TO A STAR SHIP" 3
echo -e "\f\f"
center "_____" 4

colour black_yellow
xy 5 1  "_____"
xy 7 1  "_____"
xy 21 1 "_____"
center "Star Date $MYDATE " 22
xy 23 1 "_____"

colour black_green
xy 6 6 "Initials :"
read INIT
xy 8 14
echo -n "Security Code No:      :"
read CODE
xy 10 14
echo -n "Ship's Serial No:      :"
read SERIAL
xy 12 14
echo -n "Is it on the Port Side :"
read PORT

colour red_yellow
center " Save This Record [Y..N]:" 18
read ans

# восстановление обычных цветов экрана
colour black_white
```

Нетрудно заметить, что этот сценарий не включает методов проверки. В данном случае все нормально. Сценарий просто демонстрирует, как можно раскрасить экран.

21.2.11. Улучшение внешнего вида меню

Помните ли вы меню, которое создавалось при обсуждении циклов `while`? Внесем в сценарий этого меню некоторые усовершенствования. В результате оно будет включать следующие варианты выбора:

```
1 : ADD A RECORD
2 : VIEW A RECORD
3 : PAGE ALL RECORDS
4 : CHANGE A RECORD
5 : DELETE A RECORD
P : PRINT ALL RECORDS
H : Help screen
Q : Exit Menu
```

В сценарии обработки этого меню применяется функция `read_char`, поэтому пользователь не должен при выборе опций меню нажимать клавишу [Return]. Для игнорирования сигналов 2, 3 и 15 применяется команда `trap` (более подробно это команда обсуждается далее), поэтому пользователь может не прерывать работу с меню.

Меню также имеет возможности по управлению доступом. Пользователи, обладающие определенными привилегиями, могут вносить изменения в записи или удалять их. Остальным пользователям разрешается только добавлять записи, просматривать их и выводить на экран. Список действительных пользователей с указанием уровней доступа находится в файле *priv.user*.

Если к меню попытаются обратиться пользователи, имена которых не содержатся в указанном файле, отобразится сообщение о том, что эти пользователи не имеют права выполнять данное приложение. После этого выполнение приложения завершается.

Опции меню скрывают системные команды. Если нужно ознакомиться с подтверждением данных при обновлении файла, обратитесь к главе 22.

Ниже приводится файл *priv.user*, содержащий имена пользователей, которые могут или не могут удалять записи и вносить в них изменения. Из приведенного текста видно, что пользователи `root`, `dave` и `matty` не имеют права вносить изменения в файлы баз данных, а пользователи `peter` и `louise` располагают этим правом.

```
$ pg priv.user
# файл доступа prov.user для меню apps
# его изменение является рискованным !!!!
# формат реализуют записи USER AMEND/DELETE
# например, запись "root yes" означает, что пользователь root может
# обновлять или удалять записи
# запись "dave no" означает, что пользователь dave не может обновлять или
удалять записи
root no
dave no
peter yes
louise yes
matty no
```

Чтобы проверить права доступа пользователей, сначала просмотрим файл. При этом игнорируются строки комментария, а все другие строки перенаправляются в файл *temp*.

```
user_level()
{
while read LINE
do
  case $LINE in
  \#*);;
  *) echo $LINE >>$HOLD1
    ;;
  esac
done < $USER_LEVELS

FOUND=false
while read MENU_USER PRIV
do
  if [ "$MENU_USER" = "$USER" ];
  then
    FOUND=true
    case $PRIV in
    yes|YES)
      return 0
      ;;
    no|NO)
      return 1
      ;;
    esac
  else
    continue
  fi
done <$HOLD1
  if [ "$FOUND" = "false" ]; then
  echo "Sorry $USER you have not been authorised to use this menu"
  exit 1
  fi
}
```

На следующем этапе просматривается заново отформатированный файл. Переменной FOUND присваивается значение "ложь". Теперь файл *temp* включает только описание имен и прав доступа; именам пользователей и правам доступа назначаются переменные. Чтобы уточнить, соответствует ли имя в файле значению USER, производится проверка; значение USER берется из команды whoami, расположенной в начале сценария. Если совпадение не найдено, выполняется проверка с помощью конструкции else, и с помощью команды continue обработка продолжается на следующей итерации.

Этот процесс длится до тех пор, пока все имена пользователей не будут просмотрены. При этом имя пользователя сравнивается со значением переменной USER. Если при просмотре всего файла совпадение не установлено, конструкция test в конце программного кода проверяет значение переменной FOUND. Если значением переменной является "ложь", пользователю отказывается в дальнейшей работе.

Если в процессе выполнения цикла while устанавливается искомое соответствие, переменной FOUND присваивается значение "истина". Затем с помощью конструкции case выбираются права доступа. При этом возвращается 1 для обычных прав доступа либо 0 — для расширенных прав доступа.

Когда пользователь приступает к внесению изменений в запись или к удалению записи, выполняется проверка. Этот процесс основывается на коде возврата функции, описанном выше. В приведенном сценарии либо выполняется сортировка файла *passwd*, либо отображается каталог:

```
if user_level; then
  sort /etc/passwd
else
  restrict
fi
```

Функция `restrict` просто выводит на экран сообщение о нарушении прав доступа.

Вышеприведенная проверка может быть реализована за один цикл. Однако кодирование выглядит более привлекательно, если применяется метод работы с двумя файлами. В результате значительно облегчается отладка кода.

Чтобы выйти из меню, пользователь выбирает пункт `q` или `Q`, в результате чего вызывается функция для очистки экрана. Когда пользователь завершает работу с любым обширным сценарием, очень удобно в этом случае воспользоваться функцией. Это позволяет увеличить количество команд, выполняемых пользователем при завершении работы. Кроме того, значительно улучшается читабельность кода.

Ниже приводится соответствующий сценарий.

```
$ pg menu2
#!/bin/sh
# menu2
# СЦЕНАРИЙ ГЛАВНОГО МЕНЮ
# игнорирование CTRL-C и прерывания QUIT
trap "" 2 3 15
MYDATE=`date +%d/%m/%Y`
THIS_HOST=`hostname -s`

USER=`whoami`

# файл, описывающий права доступа пользователя
USER_LEVELS=priv.user

# файл для хранения
HOLD1=hold1.$$

# функция задания цвета
colour()
(
# формат цвет_фона;цвет_переднего_планам
case $1 in
black_green)
  echo '^[[40;32m'
  ;;
black__yellow)
  echo '^[[40;33m'
  ;;
black_white)
```

```
  echo '^[[40;37m'
  ;;
black_cyan)
  echo '^[[40;36m'
  ;;
red_yellow)
  echo '^[[41;33m'
  ;;
esac
}

# чтение значения клавиши
get_char()
{
# get_char
# сохранение текущих установок stty
SAVEDSTTY=`stty -g`
  stty cbreak
  dd if=/dev/tty bs=1 count=1 2> /dev/null
  stty -cbreak
# восстановление установок stty
stty $SAVEDSTTY
}

# отображение или сокрытие курсора
cursor()
{
# cursor
# отображение/сокрытие курсора

_OPT=$1
  case $_OPT in
  on) echo '^[[?25h'
    ;;
  off) echo '^[[?25l
    ;;
  *) return 1
    ;;
  esac
}

# проверка прав доступа пользователя
restrict()
{
colour red_yellow
echo -e -n "\n\n\007Sorry you are not authorised to use this function"
colour black_green
}

user_level()
{
# user_level
```

```
# просмотр файла priv.user
while read LINE
do
  case $LINE in
  # игнорирование комментариев
  \#*);;
  *) echo $LINE >>$HOLD1
    ;;
  esac
done < $USER_LEVELS

FOUND=false
while read MENU_USER PRIV
do
  if [ "$MENU_USER" = "$USER" ];
  then
    FOUND=true
    case $PRIV in
    yes|YES)
      return 0
      ;;
    no|NO)
      return 1
      ;;
    esac
  else
  # соответствие не найдено, чтение следующей записи
    continue
  fi
done <$HOLD1
if [ "$FOUND" = "false" ]; then
  echo "Sorry $USER you have not been authorised to use this menu"
  exit 1
fi
}

# вызывается, если пользователь выполняет выход из программы
my_exit()
{
# my_exit
# вызывается, если пользователь выбирает выход из сценария!
colour black_white
  cursor on
  rm *.$$
  exit 0
}
tput init
# отображение на экране уровня доступа пользователя
if user_level; then
  ACCESS="Access Mode is High"
else
  ACCESS="Access Mode is Normal"
fi
```

```
tput init
while :
do
tput clear
colour black_green
cat <MAYDAY
$ACCESS
```

```
 User: $USER     Host:$THIS_HOST              Date:$MYDATE

                 1 : ADD A RECORD
                 2 : VIEW A RECORD
                 3 : PAGE ALL RECORDS
                 4 : CHANCE A RECORD
                 5 : DELETE A RECORD
                 P : PRINT ALL RECORDS
                 H : Help screen
                 Q : Exit Menu
```

```
MAYDAY
colour black_cyan
echo -e -n "\tYour Choice [1,2,3,4,5,P,H,Q] >"
@ read CHOICE
CHOICE=`get_char`
  case $CHOICE in
  1) ls
    ;;
  2)vi
    ;;
  3) who
    ;;
  4) if userlevel; then
       ls -l |wc
     else
       restrict
     fi
     ;;
  5) if userlevel; then
       sort /etc/passwd
     else
       restrict
     fi
     ;;
  P|p)
    echo -e "\n\nPrinting records......"
    ;;
  H|h)
  tput clear
  cat <MAYDAY
  This is the help screen,nothing here yet to help you!
  MAYDAY
    ;;
  Q|q) my_exit
    ;;
  *) echo -e "\t\007unknown user response"
```

```
    ;;
  esac
echo -e -n "\tHit the return key to continue"
read DUMMY
done
```

Подобное меню можно вызвать с помощью команды `exec` (из файла `profile`). Пользователи не могут изменить эту последовательность действий. Такой подход распространен в случае с пользователями, работающими только с приложениями UNIX или Linux и не использующими возможности интерпретатора shell.

21.3. Заключение

Команда `tput` позволяет значительно улучшить контроль со стороны пользователей за выполнением сценариев. Применение цветов улучшает внешний вид меню. Будьте осторожны при использовании гаммы цветов; цвета могут быть привлекательны для вас, но не отвечать вкусам пользователя, вынужденного применять сценарий. Работа со сценариями значительно облегчается при использовании управляющих символов. Это особенно проявляется в случае, когда пользователи осуществляют ввод информации путем выбора назначенных клавиш.

ГЛАВА 22

Создание экранного ввода

Когда речь идет об экранном вводе, или вводе данных, подразумевают ввод информации (в нашем случае с помощью клавиатуры), а затем — проверку достоверности введенных данных. Если данные удовлетворяют неким критериям, они сохраняются.

Существуют функции для проверки некоторых условий, например длины строки. Можно также проверить, состоит ли строка из чисел или символов. Эти функции применяются наравне с другими функциями, рассматриваемыми в данной книге.

В главе обсуждаются следующие темы:

- проверка достоверности вводимых данных;
- добавление, удаление, обновление записей и их просмотр;
- сценарии, выполняющие обновление файлов.

Материал главы может с первого взгляда показаться сложным. В этом случае советуем вначале просмотреть главу, а затем вернуться к ней позже. Реализация проверки вводимых данных требует довольно большого программного кода. Однако для выявления всех возможных ошибок код должен выполнять проверку на наличие наиболее вероятных ошибок.

Рассмотрим отдельные задачи, из которых складывается общая система по обновлению файлов, включающая такие функции, как добавление, удаление, обновление и просмотр записей. В результате создается система хранения актуальной информации о сотрудниках. Записи в файле *DBFILE* содержат следующие сведения:

Поле	Длина	Разрешен ввод следующей информации	Описание
Staff number	10	Числовая	Номер служащего по штатному расписанию
First name	20	Символьная	Имя служащего
Second name	20	Символьная	Фамилия служащего
Department	—	Accounts IT Services Sales Claims	Отдел, где работает служащий

Поля разделяются двоеточием (:). Например:

```
<Staff number>:<First name>:<Second name>:<Department>
```

Каждое задание представляет собой завершенный сценарий. Небольшие части кода дублируются в некоторых представленных ниже сценариях. И это не случайно, поскольку настоящая глава посвящена описанию системы обновления данных. Когда я впервые приступил к написанию сценариев, то обнаружил явный дефицит документации, написанной простым и понятным языком. Особенно недостает описаний совместного использования базы данных и системы по обновлению файлов.

Для связи различных заданий реальный сценарий должен иметь соответствующее меню или модуль. Чаще всего для этой цели используется оболочка функций, содержащихся в файле, совместно с каким-либо меню сценария. Каждый сценарий включает команду `trap`; благодаря ее использованию игнорируются сигналы 2, 3 и 15.

22.1. Добавление записей

При добавлении записи в файл выполняются следующие задачи:

1. Подтверждение вводимых данных.
2. Внесение записи в файл.

Сначала необходимо связать вместе некоторые функции. Тогда можно узнать, являются ли поля числовыми или символьными, а также уточнить их размеры. В этом и состоит подтверждение вводимых данных. Подтверждение данных осуществляется при добавлении записей, а также при их обновлении. К счастью, некоторые из требуемых функций уже имеются в вашем арсенале.

Функция для проверки длины строки:

```
length_check()
{
# length_check
# $1=строка, $2= длина строки не превышает этого значения
_STR=$1
_MAX=$2
_LENGTH=`echo $_STR |awk '{print length($0)}'`
if [ "$_LENGTH" -gt "$_MAX" ]; then
  return 1
else
  return 0
fi
}
```

Функция, выполняющая проверку наличия в строке исключительно числовых данных:

```
a_number()
# a_number
# $1= string
{
_NUM=$1
_NUM=`echo $1 awk '{if($0~/[^0-9]/) print "1"}'`
if [ "$_NUM" != "" ]
then
  return 1
else
  return 0
fi
}
```

Функция, позволяющая определить, состоит ли строка исключительно из одних символов:

```
characters()
# characters
# $1 = string
(
_LETTERS_ONLY=$1
_LETTERS_ONLY=`echo $1|awk '{if($0~/[^a-zA-Z]/) print "1"}'`

if [ "$_LETTERS_ONLY" != "" ]
then
  return 1
else
  return 0
fi
```

При просмотре полей можно просто вызвать необходимую функцию и затем проверить коды возврата.

Чтобы сообщения находились на экране до тех пор, пока пользователь не нажмет на клавишу для их удаления, нужно воспользоваться командным приглашением. Следующая функция реализует командное приглашение `read_a_char`.

```
continue_promptYN()
{
# continue_prompt
echo -n "Hit any key to continue.."
DUMMY=`read_a_char`
}
```

Когда вводятся данные пользователя, содержащие номер служащего, нужно убедиться, что ранее подобная информация не вводилась. Поле должно быть уникальным. Существует несколько способов для выполнения этой задачи; в данном случае применяется команда `grep`. С помощью команды `grep` выполняется поиск номера служащего, который содержится в строке `_CODE`. Если утилита `awk` не возвращает какого-либо значения, то дублирующиеся значения отсутствуют и функция завершает выполнение с кодом возврата 0. Ниже приводится код этой функции. (Обратите внимание, что для нахождения точного соответствия в команде `grep` используется выражение "$_CODE\>". Двойные кавычки служат для сохранения значения переменной; при использовании одинарных кавычек ничего возвращаться не будет.)

```
check_duplicate()
{
# check_duplicate
# проверка дубликата номера служащего
_CODE=$1
MATCH=`grep "$_CODE\>" $DBFILE`
echo $_CODE
if [ "$MATCH" = "" ]; then
  return 0 # нет дублирования
else
  return 1 # дубликат найден
fi
}
```

Ниже приводится часть программного кода, выполняющая проверку номера служащего. Функционирование этого программного кода объясняется дальше.

```
while :
do
echo -n "Employee Staff Number :"
read NUM
# проверка вводимых данных
if [ "$NUM" != "" ]; then
  if a_number $NUM; then
  # номер OK
    NUM_PASS=0
  else
    NUM_PASS=1
  fi
  if length_check $NUM 10; then
  # длина OK
    LEN_PASS=0
  else
    LEN_PASS=1
  fi
  # проверка наличия дубликатов...

  if check_duplicate $NUM; then
  # нет дубликатов
    DUPLICATE=0
  else
    DUPLICATE=1
    echo "Staff Number: There is already an employee with this number"
    continue_prompt
  fi
  # проверка значений всех трех переменных; все они должны выполняться
  if [ "$LEN.PASS" = "0" -a "$NUM_PASS" = "0" -a "$DUPLICATE" = "0" ]
  then
    break
  else
    echo "Staff Number: Non-Numeric or Too Many Numbers In Field"
    continue_prompt
  fi
else
  echo "Staff Number: No Input Detected, This Field Requires a Number"
  continue_prompt
fi

done
```

Вся информация обрабатывается в цикле `while` (фактически, каждое поле, содержащее записи данных, обрабатывается в отдельном цикле `while`). Поэтому, если имеется недействительная запись, последует запрос о просмотре ее исходного значения.

После просмотра номера служащего проверяется, содержатся ли в поле некоторые данные:

```
if [ "$NUM" ! = "" ]
```

Если поле не содержит данные для ввода, не выполняется часть then конструкции if. В части else, которая завершает код подтверждения поля, отображается следующее сообщение:

```
Staff Number: No Input Detected, This Field Requires a Number
```

Часть then производит все необходимые проверки правильности вводимых данных. При условии, что имеются вводные данные, вызывается функция a_number. Эта функция проверяет, содержит ли передаваемая строка числовые данные; если это так, функция возвращает значение 0, в противном случае возвращается значение 1. На базе возвращаемых значений флагу NUM_PASS присваивается либо значение 0 (успешный возврат — наличие строки из чисел), либо значение 1 (неудачный возврат — строка не является числовой).

Затем вызывается функция length_check. Эта функция передает не только строку, но и максимальное число символов, содержащихся в строке. В данном случае передается десять символов. Если количество символов меньше или равно максимальной величине, отображается значение 0; иначе возвращается значение 1. Флаг LEN_PASS принимает либо значение 0 (в случае успешного возврата, когда длина строки не превышает максимального значения), либо значение 1 (при неудачном возврате, если получена строка, длина которой равна максимальному значению).

Проверим, имеются ли дубликаты для номеров служащих. Эту задачу выполняет функция check_duplicate. Если дубликаты не обнаружены, флагу DUPLICATE присваивается значение 1. Теперь нужно убедиться, что все три флага имеют значение 0 (ошибки отсутствуют). Для этого воспользуемся логической функцией AND. При выполнении части then обе части уравнения должны принять истинное значение.

Если проверка завершилась успешно, значение поля подтверждается. Поскольку цикл while выполняется непрерывно, следует с помощью команды break прервать его выполнение.

```
if [ "$LEN_PASS" = "0" -a "$NUM_PASS" = "0" -a "$DUPLICATE" = "0"
]; then
break
```

Если при выполнении подтверждения какой-либо части (например, при проверке длины строки или наличия только числовых значений) возникает ошибка, то в нижней части экрана отображается соответствующее сообщение.

```
Staff Number: Non-Numeric or Too Many Numbers In Field
```

Действительно, значение одного поля подтверждается.

Чтобы подтвердить второе и третье поля, осуществляется аналогичный процесс. Часть, отвечающая за подтверждение, содержится в другом непрерывно выполняющемся цикле while. При этом вызывается функция characters. Эта функция проверяет, содержатся ли в поле только символы. Ниже приводится фрагмент кода, в котором выполняется подтверждение имени:

```
while :
do
echo -n "Employee's First Name :"
read F_NAME
if [ "$F_NAME" != "" ]; then
  if characters $F_NAME; then
    F_NAME_PASS=0
```

```
  else
    F_NAME_PASS=1
  fi
  if length_check $F_NAME 20; then
    LEN_PASS=0
  else
    LEN_PASS=1
  fi
  if [ "$LEN_PASS" = "0" -a "$F_NAME_PASS" = "0" ]; then
    break
  else
    echo "Staff First Name: Non-Character or Too Many Characters In Field"
    continue_prompt
  fi
else
  echo "Staff First Name: No Input Detected, This Field Requires Characters"
  continue_prompt
fi
done
```

Для подтверждения названия отдела (обратите внимание на приведенный ниже листинг) применяется конструкция `case`. Компания имеет пять отделов, а поле содержит только одно название. Обратите внимание, что названию каждого из отделов соответствуют три различных шаблона. Благодаря этому можно точно установить название отдела, если пользователь не помнит его наименование. Если совпадение с шаблоном найдено, пользователь прерывает выполнение конструкции `case`. Другие вводные данные перехватываются, в результате чего отображается список имеющихся отделов.

```
while :
do
echo -n "Company Department :"
read DEPART
  case $DEPART in
  ACCOUNTS|Accounts|accounts) break;;
  SALES|Sales|sales) break;;
  IT|It|it) break;;
  CLAIMS|Claims|claims) break;;
  SERVICES|Services|services) break;;
  *) echo "Department: Accounts,Sales,IT,Claims,Services";;
  esac
done
```

Когда все поля подтверждены, отображается приглашение с вопросом, следует ли сохранять эту запись. С этой целью применяется функция `continue_promptYN`, к которой мы уже обращались ранее, используя `Y` или `N` в качестве ответа. Если пользователь нажимает клавишу [Return], можно также передать ответ, заданный по умолчанию.

Если пользователь выбирает `N`, производится просмотр блоков кода в конструкции `if` и выполнение сценария завершается. При вводе пользователем большого числа записей следует применить функцию, которая добавляет записи в теле цикла `while`. Тогда после добавления записи возврат в меню не происходит, а выполнение сценария не завершается.

Если пользователь выбирает Y, запись сохраняется. Для добавления записи в файл применяется следующий код:

```
echo "$NUM:$F_NAME:$S_NAME:$DEPART" >>$DBFILE
```

Пользователь получает сообщение о том, что запись сохраняется в файле; при этом команда sleep приостанавливает выполнение сценария на одну секунду. Именно за это время пользователь сможет просмотреть сообщение.

Разделителем поля служит двоеточие. Затем файл сортируется по полю, содержащему фамилию, а поток вывода направляется в файл *temp*. После этого файл перемещается назад, в исходный файл *DEFILE*. Во время перемещений файла выполняется проверка кода завершения. При появлении каких-либо затруднений пользователь получает соответствующее сообщение.

Вот как выглядит поток данных вывода в случае добавления записи:

```
ADD A RECORD
    Employee Staff Number : 23233
    Employee's First Name : Peter
    Employee's Surname    : Wills
    Company Department    :Accounts
Do You wish To Save This Record [Y..N] [Y]:
saved
```

А теперь обратите внимание, какой вид имеет файл *DBFILE* после добавления нескольких записей:

```
$ pg DBFILE
32123:Liam:Croad:Claims
2399:Piers:Cross:Accounts
239192:John:Long:Accounts
98211:Simon:Penny:Services
99202:Julie:Sittle:IT
23736:Peter:Wills:Accounts
89232:Louise:Wilson:Accounts
91811:Andy:Wools:IT
```

Ниже приводится полный сценарий, выполняющий добавление записи:

```
$ pg dbase_add
#!/bin/sh
# dbase_add
# добавление записи
# игнорирование сигналов
trap "" 2 3 15
# файл temp содержит файлы
DBFILE=DBFILE
HOLD1=HOLD1.$$

read_a_char()
{
# read_a_char
# сохранение установок
SAVEDSTTY=`stty -g`
  stty cbreak
  dd if=/dev/tty bs=1 count=1 2> /dev/null
```

```
  stty -cbreak
stty $SAVEDSTTY
}

continue_promptYN()
#вызов: continue_prompt "string to display" default_answer
{
# continue_prompt
_STR=$1
_DEFAULT=$2
# проверка наличия корректных параметров
if [ $# -lt 1 ]; then
  echo "continue_prompt: I need a string to display"
  return 1
fi
while :
do
  echo -n "$_STR [Y..N] [$_DEFAULT]:"
  read _ANS
  # если пользователь нажимает клавишу ввода, устанавливается значение
  # по умолчанию, затем определяется возвращаемое значение

  : ${_ANS:=$_DEFAULT}
  if [ "$_ANS" = "" ]; then
    case $_ANS in
    Y) return 0 ;;
    N) return 1 ;;
    esac
  fi
  # пользователь выбирает что-либо
    case $_ANS in
    y|Y|Yes|YES)
      return 0
      ;;
    n|N|No|NO)
      return 1
      ;;
    *) echo "Answer either Y or N, default is $_DEFAULT"
    ;;
    esac
  echo $_ANS
done
}

continue_prompt()
{
# continue_prompt
echo -n "Hit any key to continue.."
DUMMY=`read_a_char`
}

length_check()
{
# length_check
```

```
# $1=строка для проверки длины $2=максимальная длина
_STR=$1
_MAX=$2
_LENGTH=`echo $_STR |awk '{print length($0)}'`
if [ "$_LENGTH" -gt "$_MAX" ]; then
  return 1
else
  return 0
fi
}

a_number()
{
# a_number
# вызов: a_number $1=number
_NUM=$1
_NUM=`echo $1|awk '{if($0~/[^0-9]/) print "1"}'`
if [ "$_NUM" != "" ]
then
  # ошибки
  return 1
else
  return 0
fi
}

characters()
# characters
# вызов: char_name string
{
_LETTERS_ONLY=$1
_LETTERS_ONLY=`echo $1|awk '{if($0~/[^a-zA-Z]/) print "1"}'`
if [ "$_LETTERS_ONLY" != "" ]
then
  # ошибки
  return 1
else
  # содержит только символы
  return 0
fi
}

check_duplicate()
{
# check_duplicate
# проверка дублирования номера служащего
# для вызова: check_duplicate строка
_CODE=$1
MATCH=`grep "$_CODE\>" $DBFILE`
echo $_CODE
if [ "$MATCH" = "" ]; then
  return 0 # нет дубликата
else
  return 1 # дубликат обнаружен
```

```
fi
}

add_rec()
{
# add_rec
# == STAFF NUMBER
while :
do
  echo -n "Employee Staff Number :"
  read NUM
  if [ "$NUM" != "" ]; then
    if a_number $NUM; then
      NUM_PASS=0
    else
      NUM_PASS=1
    fi
    if length_check $NUM 10; then
      LEN_PASS=0
    else
      LEN_PASS=1
    fi
    # проверка наличия дубликатов...
    if check_duplicate $NUM; then
      DUPLICATE=0
    else
      DUPLICATE=1
      echo "Staff Number: There is already a employee with this number"
      continue_prompt
    fi
    if [ "$LEN_PASS" = "0" -a "$NUM_PASS" = "0" -a "$DUPLICATE" = "0" ]
    then
      break
    else
      echo "Staff Number: Non-Numeric or Too Many Numbers In Field"
      continue_prompt
    fi
  else
    echo "Staff Number: No Input Detected, This Field Requires a Number"
    continue_prompt
  fi
done

# == Имя
while :
do
  echo -n "Employee's First Name :"
  read F_NAME
  if [ "$F_NAME" != "" ]; then
    if characters $F_NAME; then
      F_NAME_PASS=0
    else
      F_NAME_PASS=1
    fi
```

```
    if length_check $F_NAME 20; then
      LEN_PASS=0
    else
      LEN_PASS=1
    fi
    # оба условия должны быть истинными для выхода из этого цикла
    if [ "$LEN_PASS" = "0" -a "$F_NAME_PASS" = "0" ]; then
      break
    else
      echo "Staff First Name: Non-Character or Too Many Characters In Field"
      continue_prompt
    fi
  else
    echo "Staff First Name: No Input Detected, This Field Requires
Characters"
    continue_prompt
  fi
done

# == Фамилия
while :
do
  echo -n "Employee's Surname :"
  read $_NAME
  if [ "$S_NAME" != "" ]; then
    if characters $S_NAME; then
      S_NAME_PASS=0
    else
      S_NAME_PASS=1
    fi
    if length_check $S_NAME 20; then
      LEN_PASS=0
    else
      LEN_PASS=1
    fi
    if [ "$LEN_PASS" = "0" -a "$S_NAME_PASS" = "0" ]; then
      break
    else
      echo "Staff Surname: Non-Character or Too Many Characters In Field"
      continue_prompt
    fi
  else
    echo "Staff Surname: No Input Detected, This Field Requires Characters"
    continue_prompt
  fi
done

# == Отдел
while :
do
  echo -n  "Company Department  :"
  read DEPART
  case $DEPART in
    ACCOUNTS|Accounts|accounts) break;;
```

```
    SALES|Sales|sales) break;;
    IT|It|it) break;;
    CLAIMS|Claims|claims) break;;
    Services|SERVICES|services) break;;
    *) echo "Department: Accounts,Sales,IT,Claims,Services";;
  esac
done
}

# основная программа
clear
echo -e "\t\t\tADD A EMPLOYEE RECORD"
if [ -s $DBFILE ]; then :
else
  echo "Information: Creating new file to add employee records"
  >$DBFILE
fi
add_rec
if continue_promptYN "Do You wish To Save This Record " "Y"; then
  echo "$NUM:$F_NAME:$S_NAME:$DEPART" >>$DBFILE
  echo "record saved"

  sleep 1
  sort +2 -t: $DBFILE >$HOLD1 2> /dev/null
  if [ $? -ne 0 ]; then
    echo "problems trying to sort the file..check it out"
    exit 1
  fi
  mv $HOLD1 $DBFILE
  if [ $? -ne 0 ]; then
    echo "problems moving the temp sort file..check it out"
    exit 1
  fi


else
  echo " record not saved"
  sleep 1
fi
```

22.2. Удаление записей

Прежде чем удалить запись из файла, ее нужно сначала отобразить для пользователя, чтобы он мог убедиться, что именно эта запись подлежит удалению. После получения подтверждения можно приступать к удалению записи.

При удалении записи выполняются следующие операции:

1. Поиск записи.
2. Отображение записи.
3. Подтверждение процедуры удаления.
4. Обновление файла.

Чтобы найти запись, воспользуемся полем, где содержатся фамилии. После того как пользователь введет фамилию, по которой ведется поиск, можно приступать к

обработке. Можно применить команду grep или утилиту awk. Но поскольку данный файл, скорее всего, содержит не более 100 записей, просмотрим его и определим наличие совпадений.

Если же файл содержит более двух сотен записей, желательно воспользоваться утилитой awk. Это значительно быстрее, чем просмотр файла; к тому же утилита awk более удобна при разделении полей, принадлежащих переменным.

Чтобы применить команду grep или утилиту awk, можно выполнить поиск в файле *DBFILE*:

```
echo "enter the surname to search "
read STR

# при работе с awk используйте команду
awk -F: '/$STR/' DBFILE

# при работе с grep используйте команду
grep "$STR" DBFILE

# либо команду
grep "$STR\ > "DBFILE
```

Обратите внимание, что при использовании утилиты awk переменную заключают в одинарные кавычки. Если не придерживаться этого условия, не выполняется возврат данных.

Чтобы разделить поля, можно каждому полю назначить переменные (не забывайте, что разделителем полей служит двоеточие). Переменной IFS нужно присвоить значение двоеточия. Если не сделать этого, запись нельзя будет просматривать. При изменении значения переменной IFS желательно сначала сохранить установки. Благодаря этому их можно будет восстановить по завершении работы сценария.

Чтобы сохранить значения переменной IFS, примените следующую команду:

```
SAVEDIFS=$IFS
```

Заменить значение переменной IFS двоеточием можно, выполнив команду

```
IFS=:
```

По завершении работы с переменной IFS вы можете легко восстановить ее значение:

```
IFS=$SAVEDIFS
```

С помощью функции get_rec можно выполнить полномасштабный поиск; этой функции не передаются параметры.

```
get_rec()
{
# get_rec
clear
echo -n "Enter the employee surname :"
read STR
if [ "$STR" = "q" ]; then
  return 1
fi
```

```
REC=0
MATCH=no
if [ "$STR" != "" ]; then
  while read CODE F_NAME_S_NAME DEPART
  do
    REC=`expr $REC + 1`
    tput cup 3 4
    echo -n " searching record.. $REC"
    if [ "$S_NAME" = "$STR" ]; then
      MATCH=yes
      display_rec
      break
    else
      continue
    fi
  done <DBFILE
else
  echo "Enter a surname to search for or q to quit"
fi
if [ "$MATCH" = "no" ]; then
  no_recs
fi
}
```

Пользователь может ввести фамилию или же указать q для завершения выполнения задания. Если указывается фамилия, производится проверка для того, чтобы удостовериться, что фамилия представляет собой вводные данные. Имейте в виду, что удобнее воспользоваться следующей проверкой:

```
if [ "$STR" ! = "" ]; then
```

Но не такой:

```
[ -z $STR ]
```

При выборе первой проверки пользователю достаточно нажать на клавишу [Return], чтобы выполнить команду trap. Во втором случае устанавливается лишь наличие строки нулевой длины.

При считывании значения каждого поля файла используются значащие наименования переменных. Затем при считывании записей применяется счетчик. Действительно, такой прием отражается только на внешней форме сценария и позволяет контролировать процесс поиска записей. Если совпадение найдено, вызывается другая процедура, которая приводит к отображению значений полей. Команда break позволяет прекратить выполнение цикла. Если совпадение не обнаружено, сценарий продолжает выполняться до следующей итерации цикла.

Когда устанавливается соответствие, пользователь получает запрос, действительно ли нужно удалить запись. По умолчанию ответ будет no.

```
if continue_promptYN "Do You Wish To DELETE This Record" "N"; then
echo "DEL"
grep -v $STR DBFILE >$HOLD1 2> /dev/null
if [ $? -ne 0 ]; then
  echo "Problems creating temp file $HOLD1.. check it out"
  exit 1
fi
...
```

Удаление записи выполняется выполнением команды `grep` с опцией `-v`. В этом случае с помощью строки `STR` отображаются все несовпадающие поля. (Эта строка содержит фамилию, которая запрашивается пользователем при удалении записи.)

Поток данных вывода для команды `grep` перенаправляется во временный файл, где выполняется сортировка. Затем временный файл заменяет исходный файл *DBFILE*.

При реализации всех перемещений данных выполняется проверка с помощью кода завершения последней команды. Ниже показан поток вывода при удалении записи:

```
Enter the employee surname :Wilson

     searching record.. 6

EMPLOYEE NO: 89232
FIRST NAME : Louise
SURNAME    : Wilson
DEPARTMENT : Accounts

Do You Wish To DELETE This Record [Y..N] [N]:
```

А теперь приведем полный сценарий, выполняющий удаление записи:

```
$ pg dbase_del
#!/bin/sh
# dbase_del
# удаление записи

# перехват сигналов
trap "" 2 3 15

# Файл данных
DBFILE=DBFILE

# временные файлы
HOLD1=HOLD1.$$
HOLD2=HOLD2.$$

continue_promptYN()
{
# continue_prompt
_STR=$1
_DEFAULT=$2
# проверим наличие правильных параметров
if [ $# -lt 1 ]; then
  echo "continue_prompt: I need a string to display"
  return 1
fi
while :
do
  echo -n "$_STR [Y..N] [$_DEFAULT]:"
  read _ANS
  : ${_ANS:=$_DEFAULT}
  if [ "$_ANS" = "" ]; then
```

```
    case $_ANS "in
    Y) return 0 ;;
    N) return 1 ;;
    esac
  fi
  case $_ANS in
  y|Y|Yes|YES)
    return 0
    ;;
  n|N|No|NO)
    return 1
    ;;
  *) echo "Answer either Y or N, default is $_DEFAULT"
    ;;
  esac
done
}

display_rec()
{
# display_rec
# можно воспользоваться командой cat << документ
tput cup 5 3
echo "EMPLOYEE NO: $CODE"
echo "FIRST NAME : $F_NAME"
echo "SURNAME    : $S_NAME"
echo "DEPARTMENT : $DEPART"
echo -e "\n\n"
}

no_recs()
{
# no_recs
echo -e "\n\nSorry could not find a record with the name $STR"
}

get_rec()
{
# get_rec
clear
echo -n "Enter the employee surname :"
read STR
if [ "$STR" = "q" ]; then
  return 1
fi

REC=0
MATCH=no
if [ "$STR" != "" ]; then
  while read CODE F_NAME S_NAME DEPART
  do
    REC=`expr $REC + 1`
```

```
    tput cup 3 6
    echo -n " searching record.. $REC"
    if [ "$S_NAME" = "$STR" ]; then
      MATCH=yes
      display_rec
      break
    else
      continue
    fi
  done $DBFILE
else
  echo "Enter a surname to search for or q to quit"
fi
if [ "$MATCH" = "no" ]; then
  no_recs
fi
}


SAVEDIFS=$IFS
IFS=:
get_rec
if [ "$MATCH" = "yes" ]; then
  if continue_promptYN "Do You Wish To DELETE This Record" "N"; then
    echo "DEL"
    grep -v $STR DBFILE >$HOLD1 2> /dev/null
    if [ $? -ne 0 ]; then
      echo "Problems creating temp file $HOLD1..check it out"
      exit 1
    fi
    mv $HOLD1 DBFILE
    if [ $? -ne 0 ]; then
      echo "Problems moving temp file..check it out"
      exit 1
    fi
    # сортировка файла после изменений
    sort +2 -t: $DBFILE >$HOLD2 2> /dev/null
    if [ $? -ne 0 ]; then
      echo "problems trying to sort the file..check it out"
      exit 1
    fi
    mv $HOLD2 $DBFILE
    if [ $? -ne 0 ]; then
      echo "problems moving the temp sort file..check it out"
      exit 1
    fi
  else
    echo "no deletion"
    # удаление отсутствует
  fi # если нужно удалить
fi # если совпадает

# восстановление установок IFS
IFS=$SAVEDIFS
```

22.3. Обновление записей

При рассмотрении процесса удаления записи уже обсуждался код, приводящий к обновлению записи.

Когда обнаруживается корректная запись, все переменные поля записи с помощью переменной, выполняющей присваивание по умолчанию, назначаются переменной `temp`:

```
: [переменная, заданная по умолчанию: = переменная]
```

Пользователь может просто нажать клавишу [Return] для тех полей, значения которых изменять нежелательно. Затем переменная `temp` заменяется значением, заданным по умолчанию. Для выполнения произвольных обновлений можно просто вводить в соседние поля новые значения.

```
echo -n -e "EMPLOYEE NO: $CODE\n"

echo -n "FIRST NAME : [$F_NAME] >"
read _F_NAME
: ${_FNAME:=$P_NAME}
...
```

Для реального обновления файла достаточно снова воспользоваться командой `grep` с опцией `-v`. Все записи обновляются по отдельности и перенаправляются во временный файл. Номер служащего задается в виде строки команды `grep`:

```
grep -v $CODE $DBFILE >$HOLD1
```

Пользователь получает запрос, нужно ли сохранять данную запись. Если ответ положителен, обновленная запись также добавляется во временный файл. Затем временный файл перемещается в исходный файл *DEFILE*.

```
echo "$CODE:$_F_NAME:$_S_NAME:$_DEPART" >> $HOLD1
mv $HOLD1 $DBFILE
```

После этого выполняется сортировка файла с помощью потока вывода, который сначала перенаправлялся во временный файл, затем перемещается обратно — в исходный файл *DBFILE*. Проверки кода завершения выполняются во время упомянутых действий с файлами, и пользователю выдаются сообщения об имеющихся затруднениях. Ниже показан поток вывода при обновлении записи.

```
Enter the employee surname :Penny

     searching record.. 7

EMPLOYEE NO: 98211
FIRST NAME : Simon
SURNAME    : Penny
DEPARTMENT : Services

Is this the record you wish to amend [Y..N] [Y]:
amending
EMPLOYEE NO: 98211
FIRST NAME : [Simon] >
SURNAME    : [Penny] >
```

```
DEPARTMENT : [Services] >Accounts
Ready to save this record [Y..N] [Y] :
```

Полный сценарий, выполняющий обновление записей:

```
$ pg dbasechange
# !/bin/sh
# dbasechange
# обновление записи

# игнорирование сигналов
trap "" 2 3 15

# временные файлы
DBFILE=DBFILE
HOLD1=HOLD1.$$
HOLD2=HOLD2.$$

continue_promptYN()
{
# continue_prompt
_STR=$1
_DEFAULT=$2
# проверим, что параметры верны
if [ $# -lt 1 ]; then
  echo "continue_prompt: I need a string to display"
  return 1
fi
while :
do
  echo -n "$_STR [Y..N] [$_DEFAULT]:"
  read _ANS
  : ${_ANS:=$_DEFAULT}
  if [ "$_ANS" = "" ]; then
    case $_ANS in
    Y) return 0 ;;
    N) return 1 ;;
    esac
  fi
  case $_ANS in
  y|Y|Yes|YES)
    return 0
    ;;
  n|N|No|NO)
    return 1
    ;;
  *) echo "Answer either Y or N, default is $_DEFAULT"
    ;;
  esac
done
}

display_rec()
{
```

```
# отображение_записи
# можно применить команду cat << документ, но нежелательно
tput cup 5 3
echo "EMPLOYEE NO: $CODE"
echo "FIRST NAME : $F_NAME"
echo "SURNAME    : $S_NAME"
echo "DEPARTMENT : $DEPART"
echo -e "\n\n"
}

no_recs()
{
# no_recs
echo -e "\n\nSorry could not find a record with the name $STR"
}

get_rec()
{
# get_rec
clear
echo -n "Enter the employee surname :"
read STR
if [ "$STR" = "q" ]; then
  return 1
fi

REC=0
MATCH=no
if [ "$STR" != "" ]; then
  while read CODE F_NAME S_NAME DEPART
  do
    REC=`expr $REC + 1`
    tput cup 3 4
    echo -n " searching record.. $REC"
    if [ "$S_NAME" = "$STR" ]; then
      MATCH=yes
      display_rec
      break
    else
      continue
    fi
  done $DBFILE
else
  echo "Enter a surname to search for or q to quit"
fi
if [ "$MATCH" = "no" ]; then
  no_recs
fi
}

# основная программа
SAVEDIFS=$IFS
IFS=:
get_rec
```

```
if [ "$MATCH" = "yes" ]; then
  if continue_promptYN "Is this the record you wish to amend" "Y"
  then
    echo "amending"
    # нельзя изменить код служащего
    echo  -n -e "EMPLOYEE NO: $CODE\n"
    echo  -n "FIRST NAME : [$F_NAME] >"
    read _F_NAME
    : ${_FNAME:=$F_NAME}

    echo  -n "SURNAME    : [$S_NAME] >"
    read _S_NAME
    : ${_S_NAME:=$S_NAME}

    echo  -n "DEPARTMENT : [$DEPART] >"
    read _DEPART
    : ${_DEPART:=$DEPART}

    grep -v $CODE $DBFILE >$HOLD1
    if [ $? -ne 0 ]; then
      echo "Problems creating temporary file..check it out"
      exit 1
    fi
    if continue_promptYN "Ready to save this record""Y";then
      echo "$CODE:$_F_NAME:$_S_NAME:$_DEPART" >> $HOLD1
      mv $HOLD1 $DBFILE
      if [ $? -ne 0 ]; then
        echo "Problems moving temporary file...check it out"
      fi
      echo " Record Amended"
      # сортировка файла после изменений
      sort +2 -t: $DBFILE >$HOLD2 2> /dev/null
      if [ $? -ne 0 ]; then
        echo "problems trying to sort the file..check it out"
        exit 1
      fi
      mv $HOLD2 $DBFILE
      if [ $? -ne 0 ]; then
        echo "problems moving the temp sort file..check it out"
        exit 1
      fi

    else #если обновление прерывается
      echo "Amend aborted"
      exit 0
    fi
  else # если не выполняется обновление
    echo "no amending"
    # нет удаления
  fi # если желательно удалить
fi # если имеется совпадение

IFS=$SAVEDIFS
```

22.4. Просмотр записей

Пользователь может просматривать как все записи, так и отдельную запись. Если пользователь желает просмотреть все записи, достаточно воспользоваться конструкцией cat и awk. Если в запись входит большое количество полей, требуется более дружественный для пользователя вывод данных. Однако такой подход может негативно сказаться на других свойствах сценария.

```
if [ "$STR" = "all" ]; then
  echo "Surname    Name    Employee Code"
  echo "_____________________________________"
  cat $DBFILE |awk -F:  '{print $2"\t"$3"\t\t"$1)' | more
  return 0
fi
```

При рассмотрении процесса удаления и обновления записей рассматривался код, приводящий к отображению отдельной записи. Небольшое изменение кода связано с тем, что при выборе пользователем этой опции, запись выводится на экран. Ниже приводится часть кода, который позволяет пересылать запись на принтер:

```
pr <<- MAYDAY

RECORD No             : $REC
EMPLOYEE NUMBER       : $CODE
EMPLOYEE NAME         : $F_NAME
EMPLOYEE SURNAME      : $S_NAME
EMPLOYEE DEPARTMENT   : $DEPART
MAYDAY
```

Вот как выглядит поток вывода при просмотре записи:

```
Enter the employee surname to view or all for all records:Wilson

     searching record.. 8

EMPLOYEE NO   : 89232
FIRST NAME    : Peter
SURNAME       : Wilson
DEPARTMENT    : IT

Do You Wish To Print This Record [Y..N]   [N]:
```

Полный сценарий, позволяющий просматривать записи, имеет следующий вид:

```
$ pg dbaseview
#!/bin/sh
# dbaseview
# просмотр записей

# игнорирование сигналов
trap "" 2 3 15

# временные файлы
HOLD1=HOLD1.$$
DBFILE=DBFILE
```

```
continue_promptYN()
{
# continue_prompt
_STR=$1
_DEFAULT=$2
# проверим, что параметры верны
if [ $# -lt 1 ]; then
  echo "continue_prompt: I need a string to display"
  return 1
fi
while :
do
  echo -n "$_STR [Y..N] [$_DEFAULT]:"
  read _ANS
  : ${_ANS:=$_DEFAULT}
  if [ "$_ANS" = "" ]; then
    case $_ANS in
    Y) return 0 ;;
    N) return 1 ;;
    esac
  fi
  case $_ANS in
  y|Y|Yes|YES)
    return 0
    ;;
  n|N|No|NO)
    return 1
    ;;
  *) echo "Answer either Y or N, default is $_DEFAULT"
  esac
done
)

display_rec()
{
# display_rec
# можно применить команду cat <<.
tput cup 5 3
echo "EMPLOYEE NO: $CODE"
echo "FIRST NAME : $F_NAME"
echo "SURNAME    : $S_NAME"
echo "DEPARTMENT : $DEPART"
echo -e "\n\n"
}

no_recs()
{
# no_recs
echo -e "\n\nSorry could not find a record with the name $STR"
}

get_rec()
{
# get_rec
```

```
clear
echo -n "Enter the employee surname to view or all for all records:"
read STR
if [ "$STR" = "q" ]; then
  return 1
fi
if [ "$STR" = "all" ]; then
  # просмотр всех записей
  echo "Surname    Name    Employee Code"
  echo "_____________________________"
  cat $DBFILE |awk -F: '{print $2"\t"$3"\t\t"$1}' | more
  return 0
fi
REC=0
MATCH=no
if [ "$STR" != "" ]; then
  while read CODE F_NAME S_NAME DEPART
  do
    REC=`expr $REC + 1`
    tput cup 3 4
    echo -n " searching record.. $REC"
    if [ "$S_NAME" = "$STR" ]; then
      # обнаружено имя
      MATCH=yes
      display_rec
      break
    else
      continue
    fi
  done <$DBFILE
else
  echo "Enter a surname to search for or q to quit"
fi
if [ "$MATCH" = "no" ]; then
  no_recs
fi
}

# главная программа
SAVEDIFS=$IFS
IFS=:
get_rec
if [ "$MATCH" = "yes" ]; then
  if continue_promptYN "Do You Wish To Print This Record" "N"; then
  lpr <<- MAYDAY

  RECORD No: $REC

  EMPLOYEE NUMBER       : $CODE
  EMPLOYEE NAME         : $F_NAME
  EMPLOYEE SURNAME      : $S_NAME
  EMPLOYEE DEPARTMENT   : $DEPART

  MAYDAY
```

```
  else
    echo "No print of $S_NAME"
    # не отображается
  fi # если желательно отображение
fi # если имеется совпадение

IFS=$SAVEDIFS
```

22.5. Заключение

Проверка достоверности данных, вводимых пользователем, играет важную роль. Однако для ее организации требуются некоторые дополнительные навыки. Вы имеете представление, какая вводимая информация помещается в запись, но в общем случае пользователи не располагают такими сведениями.

В среде программистов известно довольно старое изречение. Звучит оно примерно так: "Мусор на входе, мусор на выходе... кого это заботит, только б не было слишком поздно". Это означает, что если не проверять входные данные сценариев, в поток вывода может попасть ненужная информация.

ГЛАВА 23

Отладка сценариев

Одной из самых сложных задач при создании shell-сценариев является их отладка. Желательно, чтобы пользователь, выполняющий эту задачу, получил консультации на данном этапе. Чтобы избежать распространенных ошибок, достаточно следовать указанному ниже правилу.

Разбейте предлагаемый сценарий на задачи или процедуры, затем запрограммируйте и проверьте каждую процедуру и лишь потом переходите к следующему этапу.

В этой главе рассматриваются следующие темы:

- распространенные ошибки;
- применение команды `set`.

Действительно, ничто так не раздражает, как поиск ошибки, "спрятанной" глубоко в сценарии. Однако некоторый опыт написания сценариев поможет локализовать ошибку.

Чаще всего при написании сценариев пропускаются кавычки либо ключевое слово `fi` в конце конструкции `if`.

Следует учитывать, что если интерпретатор команд выдает сообщение о наличии ошибки в сценарии, нужно проанализировать не только строку, где может находиться ошибка, но также и блок кода, включающий эту строку. Интерпретатор shell не всегда точно указывает на местонахождение ошибки: сообщение об ошибке обычно появляется после выполнения строки с ошибочным оператором.

23.1. Наиболее распространенные ошибки

23.1.1. Ошибки, связанные с циклом

Если сообщение об ошибке появляется при выполнении конструкций `for`, `while`, `until` или `case`, то это может означать, что фактический блок инструкций некорректно определен. Возможно, было пропущено зарезервированное слово, требуемое в данной ситуации.

Ниже в сообщении об ошибке содержится слово "done", которое помогает разобраться в сути проблемы. Теперь пользователь знает, что нужно внести изменения в конструкцию `while`. При внимательном просмотре кода следует проверить наличие всех необходимых зарезервированных слов для конструкции `while`, например "do", или ключевого слова для применяемой условной конструкции.

```
syntax error near unexpected token 'done'
line 31: ' done'
```

23.1.2. Как обычно пропускают кавычки

Второй распространенной ошибкой является элементарный пропуск кавычек. Обратите внимание на приведенный пример. Обычно приходится сталкиваться с большим количеством подобных примеров. Можно посоветовать еще раз изучить сценарий и проконтролировать наличие всех необходимых открывающих и закрывающих кавычек.

```
unexpected EOF while looking for '"'
line 36: syntax error
```

Если требуется настроить отображение сообщения об ошибке, содержащего номер ошибочной строки, то в этом случае обычно применяется опция `set nu` текстового редактора `vi`. Это удобно, если просмотр файлов осуществляется с помощью редактора `vi`. Для настройки отображаемых сообщений откройте окно редактора `vi`, затем нажмите клавишу [Esc] и введите двоеточие. После этого выполните команду `set nu` и нажмите клавишу [Return]. В результате этого происходит нумерация строк и можно перейти к той строке, где, по сообщению интерпретатора shell, содержится ошибка.

23.1.3. Проверка на наличие ошибки

Другой распространенной ошибкой является неправильное применение конструкции `-eq`. Обычно забывают указать число с какой-либо стороны уравнения.

Если поступает сообщение об ошибке, которое приводится ниже, можно сделать вывод, что произошло одно из двух событий: либо между переменной и квадратной скобкой не указан пробел, либо в квадратных скобках пропущен оператор. Вероятнее всего, в данном случае ошибка связана с первым обстоятельством.

```
[: missing '] '
```

23.1.4. Регистр символов

Чаще всего причиной ошибки является неверное использование регистра при работе с переменными. Например, при присваивании переменной применяется верхний регистр, а при ссылке на нее — нижний. Тогда не следует удивляться тому, что присваивания значения не происходит.

23.1.5. Циклы for

При работе с циклом `for` пользователи иногда забывают в части списка указать знак доллара. В результате список воспринимается как строка.

23.1.6. Команда echo

При отладке сценариев чрезвычайно удобно применять команду `echo`. Добавьте команду `echo` в наиболее существенных частях сценария, где могут возникнуть какие-либо затруднения. Например, воспользуйтесь командой `echo` до и после считывания или изменения значения переменной.

Примените код завершения последней команды для уточнения того, успешно ли была выполнена данная команда. Следует иметь в виду, что команду `echo` желательно не применять перед кодом завершения последней команды, поскольку в этом случае команда всегда возвратит истинное значение.

23.2. Команда set

При отладке сценария можно использовать команду `set`. Ниже приведены наиболее часто применяемые отладочные опции команды `set`.

`set -n`	Считывание, но не выполнение команд
`set -v`	Отображение всех строк при считывании
`set -x`	Отображение всех команд и их аргументов

Чтобы отключить опцию `set`, просто замените знак `-` на знак `+`. Конечно, привычнее было бы, наоборот, знак `+` применять для подключения, а знак `-` использовать для отключения. Но здесь все зависит от привычки.

Команду `set` можно запустить, начиная с верхней части сценария, и завершить ее выполнение по завершении сценария. Или же можно активизировать эту команду при выполнении определенной блочной конструкции, которая содержит ошибки.

Рассмотрим, как функционирует команда `set`. Ниже приводится небольшой сценарий, который включает в список переменных определенные имена. Пользователь вводит имя, затем с помощью цикла `for` просматривается список в поисках соответствия. Обратите внимание, что опция `set -x` применяется в верхней части сценария, а в нижней части сценария ее действие завершается.

```
$ pg error
#!/bin/sh
# error
# установка set -x
set -x
LIST="Peter Susan John Barry Lucy Norman Bill Leslie"
echo -n "Enter your Name :"
read NAME

for LOOP in $LIST
do
  if [ "$LOOP" = "$NAME" ]; then
    echo "you're on the list, you're in"
    break
  fi
done
# отмена установки set -x
set +x
```

Выполним сценарий, используя имя, которое отсутствует в списке. При этом поток вывода выглядит следующим образом.

```
$ error
error
+ error
+ LIST=Peter Susan John Barry Lucy Norman Bill Leslie
+ echo -n Enter your Name :
Enter your Name :+ read NAME
Harry
+ [ Peter = Harry ]
+ [ Susan = Harry ]
+ [ John = Harry ]
```

```
+ [ Barry = Harry ]
+ [ Lucy = Harry ]
+ [ Norman = Harry ]
+ [ Bill = Harry ]
+ [ Leslie = Harry ]
```

По мере того как цикл `for` обрабатывает список, полностью отображаются результаты сравнения. Нежелательно применять команду `set` аналогичным образом, если возникают затруднения при просмотре файлов или на этапе сравнения строк и значений.

23.3. Заключение

При поиске ошибок нужно самостоятельно просматривать сценарии и применять команду `set` наряду с набором конструкций `echo`.

ГЛАВА 24

Встроенные команды интерпретатора shell

В предыдущих главах нам уже встречались конструкции, встроенные в интерпретатор shell. Напомним, что речь идет о командах, которые не находятся в каталоге */bin* или *usr/bin*, а встроены в интерпретатор Bourne shell. Скорость выполнения встроенных команд выше по сравнению с их эквивалентами в системе (если таковые имеются). В этой главе рассматривается единственная тема: список стандартных встроенных команд интерпретатора Bourne shell.

Например, для выполнения одних и тех же действий можно воспользоваться как системными командами `cd` и `pwd`, так и аналогичными командами, встроенными в интерпретатор shell. Чтобы выполнять системную версию команды, кроме ее имени укажите и каталог, в котором она находится:

```
/bin/pwd
```

24.1. Полный список команд, встроенных в интерпретатор shell

В табл. 24.1 содержится полный перечень стандартных встроенных команд.

Таблица 24.1. Стандартные встроенные команды

`:`	Нуль, всегда возвращает истинное значение
`.`	Считывание файлов из текущего интерпретатора shell
`break`	Применяется в конструкциях `for`, `while`, `until`, `case`
`cd`	Изменяет текущий каталог
`continue`	Продолжает цикл, начиная следующую итерацию
`echo`	Записывает вывод в стандартный поток вывода
`eval`	Считывает аргумент и выполняет результирующую команду
`exec`	Выполняет команду, но не в этом интерпретаторе shell
`exit`	Выход из интерпретатора shell
`export`	Экспортирует переменные, вследствие чего они доступны для текущего интерпретатора shell
`pwd`	Отображает текущий каталог
`read`	Просматривает строку текста из стандартного потока
`readonly`	Превращает данную переменную в переменную "только для чтения"

return	Выход из функции с отображением кода возврата
set	Управляет отображением различных параметров для стандартного потока вводных данных
shift	Смещает влево командную строку аргументов
test	Оценивает условное выражение
times	Отображает имя пользователя и системные промежутки времени для процессов, которые выполняются с помощью интерпретатора shell
trap	При получении сигнала выполняет определенную команду
type	Интерпретирует, каким образом интерпретатор shell применяет имя в качестве команды
ulimit	Отображает или устанавливает ресурсы интерпретатора shell
umask	Отображает или устанавливает режимы создания файлов, заданные по умолчанию
unset	Удаляет из памяти интерпретатора shell переменную или функцию
wait	Ожидает окончания дочернего процесса и сообщает о его завершении

Изучим более подробно некоторые команды, которые до сих пор не рассматривались либо рассматривались поверхностно.

24.1.1. Команда pwd

Эта команда отображает текущий каталог:

```
$ pwd
/tmp
```

24.1.2. Команда set

Команда `set` рассматривалась при изучении процесса отладки для подключения и отключения опций. Эту команду можно также использовать для передачи аргументов в пределах сценария. Опишем, как происходит этот процесс. Предположим, что сценарий должен обработать два параметра. Вместо передачи параметров в сценарий выполняют их задание в сценарии. Для этого применяется команда `set`.

Формат команды:

```
set параметр1 параметр2..
```

В следующем примере параметрам присваиваются значения `accounts.doc` и `accounts.bak`. Затем в сценарии по этим параметрам выполняется цикл.

```
$ pg set_ex
#!/bin/sh
set accounts.doc accounts.bak

while [ $#!= 0 ]
do
  echo $1
  shift
done

$ set_ex
```

```
accounts.doc
accounts.bak
```

Команду set удобно использовать, если при проверке сценария необходимы параметры. Дело в том, что вновь введенные аргументы сохраняются всякий раз, когда выполняется сценарий.

24.1.3. Команда times

Команда times информирует пользователя о том, сколько времени требуется для выполнения пользовательских и любых системных команд. В первой строке указывается время, которое необходимо интерпретатору shell, а во второй — время, которое нужно всем исполняемым командам. Ниже приводится пример потока вывода, который получен с помощью команды times.

```
$ times
0m0.10s 0m0.13s
0m0.49s 0m0.36s
```

Эта команда применяется довольно часто!

24.1.4. Команда type

Команда type позволяет выяснить, содержится ли некоторая команда в системе, и определить тип данной команды. Команда type также сообщает, является ли название команды действительным и где именно в системе находится эта команда. Ниже приводятся примеры использования команды type:

```
$ type mayday
type: mayday: not found
$ type pwd
pwd is a shell builtin
$ type times
times is a shell builtin
$ type cp
cp is /bin/cp
```

24.1.5. Команда ulimit

Команда ulimit используется для задания и отображения предельных значений, применяемых при выполнении сценария. Обычно эта команда находится в файле */etc/profile*, но вы можете использовать ее для своих нужд из текущего интерпретатора shell либо из вашего файла *.profile*. Общий формат команды:

```
ulimit опции
```

Ниже приводится несколько опций команды ulimit; здесь рассматриваются наиболее часто применяемые опции:

Опция	Значение
-a	Отображает текущие ограничения
-c	Ограничивает размер дампов ядра
-f	Ограничивает N блоками размер выходного файла, который создается исполняемым процессом

Ниже указаны значения, полученные при выполнении команды `ulimit`:

```
$ ulimit -a
core file size (blocks)     1000000
data seg size (kbytes)    unlimited
file size (blocks)        unlimited
max memory size (kbytes)  unlimited
stack size (kbytes)          8192
cpu time (seconds)        unlimited
max user processes            256
pipe size (512 bytes)           8
open files                    256
virtual memory (kbytes)    2105343
```

Чтобы отключить выгрузку файлов ядра, установите нулевое значение для команды.

```
$ ulimit -c 0
$
$ ulimit -a
core file size (blocks)           0
data seg size (kbytes)    unlimited
file size (blocks)        unlimited
max memory size (kbytes)  unlimited
stack size (kbytes)          8192
cpu time (seconds)        unlimited
max user processes            256
pipe size (512 bytes)           8
open files                    256
virtual memory (Kbytes)    2105343
```

24.1.6. Команда wait

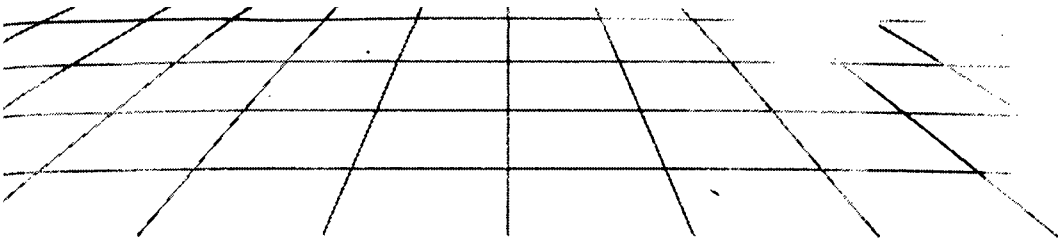
Команда `wait` применяется для ожидания завершения одного из дочерних процессов. Команду `wait` можно определить с помощью процесса ID. Если этого не сделать, ожидание будет длиться до завершения всех дочерних процессов.

Формат команды ожидания завершения всех дочерних процессов:

```
$ wait
```

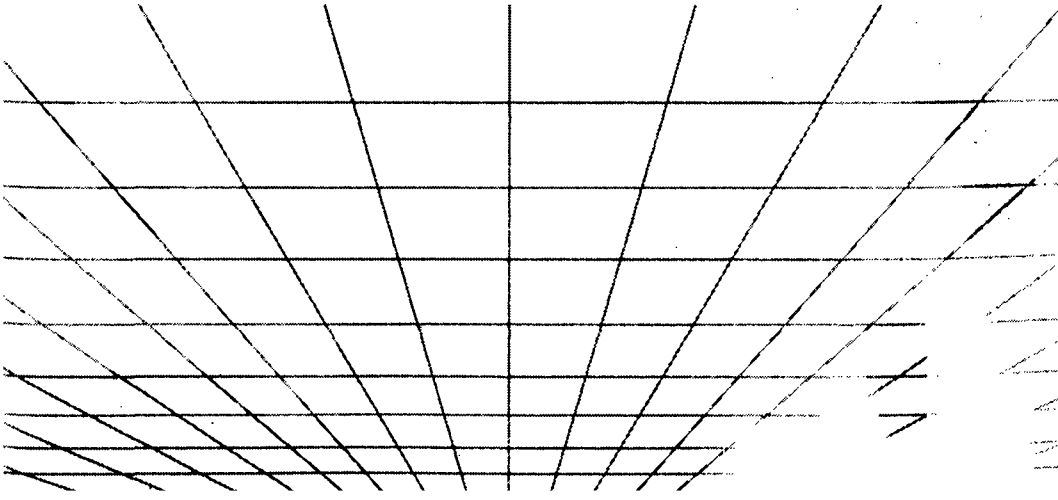
24.2. Заключение

В главе содержится обзор всех команд, встроенных в интерпретатор shell. Некоторые из них рассматривались ранее; здесь же подробно обсуждается применение этих команд



ЧАСТЬ 5

Совершенствование навыков по написанию сценариев



ГЛАВА 25

Дальнейшее изучение конструкции "документ здесь"

При рассмотрении стандартного потока ввода и вывода, а также циклов `while` уже обсуждалась конструкция "документ здесь". Описывались методика пересылки электронной почты и способы формирования экранов меню, но существуют и другие способы применения конструкции "документ здесь".

В этой главе рассматриваются следующие темы:

- скоростной метод создания файла;
- меню с автоматизированными возможностями поиска;
- передача файлов с помощью протокола `ftp`;
- подключение к другим системам приложений.

Формат конструкции "документ здесь":

```
команда << слово
текст
слово
```

Чтобы вспомнить методы применения конструкции "документ здесь", рассмотрим принципы ее работы. Когда интерпретатор shell встречает символ <<, он ожидает встретить за ним разделитель (слово). Все, что следует за этим словом до следующего слова в строке, воспринимается как поток ввода. В качестве разделителя может служить любое слово.

Конструкция "документ здесь" может применяться при создании файла, выводе на экран списка файлов, сортировке списка файлов и при создании экранов.

25.1 Быстрый метод формирования файла

Ниже рассматривается быстрый метод формирования файла, содержащего некоторый текст:

```
$ cat >> myfile <<NEWFILE
```

Введите какой-нибудь текст, а по завершении ввода в новой строке просто укажите слово `NEWFILE`. В результате образуется файл *myfile*, содержащий этот текст.

Если файл с указанным именем существует, то вводимое содержимое добавляется в исходный файл.

При использовании клавиши [Tab] следует учитывать, что более ранние версии командного интерпретатора не сохраняют значения символов табуляции. Для устранения этого недостатка введите дефис после двух угловых левых скобок, например:

```
cat >> myfile << - NEWFILE
...
```

25.2. Скоростной способ вывода документа на печать

Предположим, что нужно создать и вывести на экран небольшой документ, содержащий сообщение. Необязательно использовать редактор vi; вместо этого можно применить метод, показанный в следующем примере. Если после ввода символов QUICKDOC нажать клавишу [Return], документ пересылается на принтер.

```
$ lpr <<QUICKDOC
**** INVITATION**** The Star Trek convention is in town
next week. Be there.

Ticket prices: (please phone)
-----------------------------
QUICKDOC
```

25.3. Автоматизация меню

Несмотря на то что конструкция "документ здесь" успешно применяется для создания экранов меню, ее можно также использовать при автоматизации навигации по пунктам меню. В этом случае пользователю не придется вручную выбирать определенные опции.

Существует сценарий по работе с меню базы данных, который отвечает за резервное копирование баз данных и выполнение административных задач. Меню базы данных применяется в течение дня для выполнения резервирования базы данных и выполнения других административных задач. Если же было принято решение о том, что каждую ночь создаются резервные копии всех баз данных с помощью утилиты cron, отпадает необходимость в создании другого сценария.

В данном случае используется конструкция "документ здесь", причем для навигации по меню применяется сценарий syb_backup, использующий поток ввода. Ниже показан поток вывода для сценария меню syb_backup.

Главный экран меню, в котором выбран пункт 2, имеет следующий вид.

```
1: Admin Tasks
2: Sybase Backups
3: Maintenance Tasks
Selection > 2
```

Второй экран меню с выбранным пунктом 3 выглядит так:

```
1: Backup A Single Database
2: Backup Selected Databases
3: Backup All Databases
Selection > 3
```

Третий экран меню при нажатии клавиши [Y]:

```
1. dw_levels
2. dw_based
3. dw_aggs
...
...
Are You Sure You Wish To Backup [Y..N] : Y
```

Итак, как следует из экрана меню, для резервирования всех баз данных нужно ввести следующую информацию:

1. Название меню сценария, syb_backup.
2. Цифру 2.
3. Цифру 3.
4. Символ Y.

Ниже приводится сценарий, с помощью которого можно автоматически выделить резервные копии баз данных. При этом используется сценарий меню `syb_backup`:

```
$ pg auto.sybackup
#!/bin/sh
# задание пути
PATH=/usr/bin:/usr/sbin:/sybase/bin:$LOCALBIN
export PATH
# присваивается значение переменной
DSQUERY=COMET; export DSQUERY
# устанавливается значение TERM с последующей инициализацией
TERM=vt220; export TERM
tput -T vt220 init
# журнальный файл для всего потока вывода
log_f=/logs/sql.backup.log
#
>$log_f

# ниже приводится код, выполняющий всю эту работу!
/usr/local/sybin/syb_backup >> $log_f 2>&1 << MAYDAY
2
3
Y
MAYDAY

chown sybase $log_f
```

Далее приводится конструкция фактического перенаправления, которая выполняет все необходимые действия:

```
usr/local/sybin/syb_backup >> $log_f 2>&1 << MAYDAY
2
3
Y
MAYDAY
```

Изучая часть кода, связанного с перенаправлением, можно заметить, что в сценарии `syb_backup` содержится полностью заданный путь; `>>$log_f>&1` означает, что весь поток вывода направляется в файл *$log_f*, где находится переменная,

содержащая значение `/logs/sql.backup.log`. Такой подход довольно удобен, так как впоследствии можно захватить поток вывода, включая подробности резервного копирования и возможные ошибки приложения.

Конструкция "документ здесь" начинается со слова `<<MAYDAY`. Затем передаются коды, возникшие в результате выбора пунктов меню, выполняемого вручную для реализации резервного копирования. Конструкция "документ здесь" завершается также словом `MAYDAY`.

Вот и все. Необязательно переписывать сценарий, если его можно встроить в меню. Для автоматизации сценария примените конструкцию "документ здесь".

25.4. Автоматизация передачи файлов по протоколу ftp

Автоматизация передачи файлов по протоколу `ftp` является одним из широко распространенных методов применения конструкции "документ здесь". При использовании протокола `ftp` желательно предоставить пользователю несложный интерфейс для удобства работы. В следующем сценарии для создания подключения `ftp` применяется анонимное имя пользователя. Это специальное имя позволяет системе создавать защищенные учетные записи `ftp`, содержащие общедоступные каталоги (public). В общем случае каждый пользователь при установке подключения с использованием анонимного имени сможет только загружать файлы из данного общедоступного каталога. Но можно будет также выполнять выгрузку файлов.

Пароль можно выбрать произвольным образом, однако удобно воспользоваться названием хоста и локальным идентификатором пользователя или электронным адресом.

Приведенный ниже сценарий запрашивает следующую информацию:

1. Адрес удаленного компьютера, с которым требуется установить связь.
2. Тип передаваемых файлов: двоичный или ASCII.
3. Имя получаемого файла.
4. Локальный каталог, в котором размещается выбранный файл.

Когда пользователь вводит наименование удаленного компьютера, с которым устанавливается соединение, для удаленного хоста выполняется команда `traceroute`. Таким образом, можно быть уверенным в том, что соединение действительно установлено. Если же результат выполнения команды `traceroute` был неудачным, сценарий отображает повторный запрос.

При нажатии клавиши [Return] по умолчанию принимается двоичный режим передачи файлов. После ввода имени загружаемого файла пользователя запрашивают о каталоге назначения для загружаемого файла. По умолчанию этим каталогом служит каталог */tmp*. Если пользователь указывает другой каталог, который не может быть найден, используется каталог */tmp*.

Именем загружаемого файла будет имя файла с присоединенным к нему расширением *.ftp*. И, наконец, после того как все эти варианты отображаются и подтверждаются, начинается процесс передачи файлов.

Ниже показано, как выглядят результаты выполнения сценария.

```
$ ftpauto
User: dave              05/06/1999      This host: bumper
                  FTP RETRIEVAL / POSTING SCRIPT
                  ==============================
                     Using the ID of anonymous
```

```
Enter the host you wish to access :uniware
Wait..seeing if uniware is out there..
bumper can see uniware
What type of transfer / receive mode ?
1 : Binary
2 : ASCII
Your choice [1..2] [1] :
Enter the name of the file to retrieve :gnutar.Z
Enter the directory where the file is to be placed[/tmp] :
   Host to connect is: uniware
   File to get is    : gnutar.Z
   Mode to use is    : binary
   File to be put in : /tmp/gnutar.Z.ftp
   Ready to get file 'y' or 'q' to quit? [y..q] :
```

Далее приводится соответствующий сценарий.

```
$ pg ftpauto
#!/bin/sh
# сценарий ftp
# ftpauto
USER=`whoami`
MYDATE=`date +%d/%m/%Y`
THIS_HOST=`hostname -s`
tracelog=/tmp/tracelog.$$

while :
do
  # бесконечный цикл
  tput clear
  cat <<MAYDAY
  User:          $USER $MYDATE This host:          $THIS_HOST
              FTP RETRIEVAL SCRIPT
              ====================
            Using the ID of anonymous
  MAYDAY
  echo -n "Enter the host you wish to access :"
  read DEST_HOST
  # введено ли имя хоста ???
  if [ "$DEST_HOST" = "" ]
  then
    echo "No destination host entered" >&2
    exit 1
  fi
  # можно ли увидеть хост ???
  echo "wait..seeing if $DEST_HOST is out there.."
  # traceroute проверяет соединение
  traceroute $DEST_HOST > $tracelog 2>&1

  if grep "unknown host" $tracelog >/dev/null 2> then
    echo "Could not locate $DEST_HOST"
    echo -n "Try another host? [y..n] :"
    read ANS
    case $ANS in
```

```
    y|Y) ;;
    *) break;;   # выход из бесконечного цикла
    esac
  else
    echo "$THIS_HOST can see $DEST_HOST"
    break   # выход из бесконечного цикла
  fi
done
# по умолчанию двоичный режим
echo  "What type of transfer /receive mode ?"
echo  " 1 : Binary"
echo  " 2 : ASCII"
echo -n -e "\fYour choice [1..2] [1]:"
read $TYPE
case $TYPE in
1) MODE=binary
  ;;
2) MODE=ascii
  ;;
*) MODE=binary
  ;;
esac

echo -n " Enter the name of the file to retrieve :"
read FILENAME
if [ "$FILENAME" = "" ]; then
  echo "No filename entered" >&2
  exit 1
fi

# по умолчанию, это tmp
echo -n -e "\f Enter the directory where the file is to be placed[/tmp] :"
read DIREC
cd $DIREC >/dev/null 2>&1
# если нельзя перейти в нужный каталог, используйте tmp
if [ "$DIREC" = "" ]; then
  DIREC=/tmp
fi

if [ $? != 0 ]
then
  echo "$DIREC does not exist placing the file in /tmp anyway"
  DIREC=/tmp
fi

echo -e "\t\tHost to connect is: $DEST_HOST"
echo -e "\t\tFile to get is :    $FILENAME"
echo -e "\t\tMode to use is :    $MODE"
echo -e "\t\tFile to be put in : $DIREC/$FILENAME.ftp"
echo -e -n "\t\tReady to get file 'y' or 'q' to quit? [y..q] :"
read ANS
case $ANS in
Y|y);;
q|Q) exit 0;;
```

```
*) exit 0 ;;
esac
echo "ftp.."
ftp -i -n $DEST_HOST<<FTPIT
user anonymous $USER@$THISHOST
$MODE
get $FILENAME $DIREC/$FILENAME.ftp
quit
FTPIT
if [ -s $DIREC/$FILENAME.ftp ]
then
  echo "File is down"
else
  echo "Unable to locate $FILENAME.ftp"
fi
```

Фактическая конструкция "документ здесь", применяемая при передаче файла с помощью `ftp`, использует опции `ftp -i -n`. Эти опции означают "не выводить какие-либо автоматические запросы регистрации" и "отключение режима интерактивного запроса". Тогда сценарий выполняет регистрацию с помощью команды "user". Паролем является конструкция вида `$USER@THISHOST`, которая имеет фактическое значение `dave@bumper`.

Предположим, что пользователь ежедневно загружает один и тот же файл с помощью одного и того же хоста и, возможно, при этом требуется сохранить определенные данные, полученные за предыдущий день. Тогда пользователю нет необходимости каждый раз вводить имя файла или данные для одного и того же удаленного хоста. Для полей `DEST_HOST` и `FILENAME` можно установить значения, заданные по умолчанию. Применение заданных по умолчанию значений позволит не вводить в поля однотипную информацию.

Ниже приводится часть сценария `ftpauto`, который запрашивает название удаленного хоста. Однако теперь для поля `DEST_HOST` устанавливается значение, заданное по умолчанию, а именно `my_favourite_-host`. Теперь пользователь может в ответ на запрос ввести другое название для удаленного хоста или нажать клавишу [Return]. Тогда значение, заданное по умолчанию, загружается в переменную `DEST_HOST`.

Обратите внимание, что нет необходимости проверять, ввел ли пользователь значение. Значение, заданное по умолчанию, присваивается переменной `DEST_HOST`.

```
echo -n "Enter the host you wish to access :"
read DEST_HOST
: ${DEST_HOST:="my_favourite_host"}
echo "Wait.. see-ing if $DEST_HOST is out there.."
traceroute $DEST_HOST >$tracelog 2>&1
...
```

25.5. Организация доступа к базам данных

Общей задачей для всех сценариев является организация доступа к базам данных системы с целью осуществления выборки информации. В этом случае незаменимой является конструкция "документ здесь". Она позволяет выполнять практически все необходимые действия по запросу базы данных. Рассмотрим, как конструкция "документ здесь" применяется для установки соединения с другими приложениями, а также для выполнения других задач.

Одна из систем баз данных 'select into' отключается в том случае, если база данных становится доступной для определенных программных продуктов сторонних производителей. Значит, некоторые базы данных нельзя использовать для ввода произвольных данных или для создания временных таблиц.

Для решения этой проблемы применяется конструкция "документ здесь". С помощью этой конструкции поддерживается связь с системной базой данных, а цикл for применяется для изменения "документа здесь" вместе с названиями баз данных. После установки подключения конструкция "документ здесь" применяется для поддержки команд sql, используемых для задания параметров.

Ниже приводится сценарий, реализующий установку параметров для каждой базы данных:

```
$ pg set.select
#!/bin/sh
# set.select
# устраняется известная ошибка. Устанавливается выделение в опции db
PATH=$PATH:/sybase/bin:/sybase/install
export PATH
SYBASE="/sybase"; export SYBASE
DSQUERY=ACESRV; export DSQUERY
PASSWORD="prilog"
DATABASES="dwbased tempdb aggs levels reps accounts"

for loop in $DATABASES
do
  su Sybase -c '/sybase/bin/isql -Usa -P$PASSWORD' << MAYDAY
  use master
  go
  sp_dboption $loop,"select into/bulkcopy/pllsort",true
  go
  use $loop
  go
  checkpoint
  go
  MAYDAY
done
```

Рассматривая конструкцию "документ здесь", при выполнении вышеуказанной команды интерпретатор shell оценивает приведенный выше код.

```
use master
go
sp_dboption dwbased,"select into/bulkcopy/pllsort",true
  go
  use dw_based
  go
  checkpoint
  go
```

Когда интерпретатор shell просматривает завершающее слово MAYDAY, сценарий начинает следующую итерацию для цикла for. При этом из списка выбирается следующая база данных. При выполнении сценария получаются следующие результаты:

```
$ set.select
Database option 'select into/bulkcopy/pllsort' turned ON for database
'dwbased '.
```

```
Run the CHECKPOINT command in the database that was changed.
(return status = 0)
Database option 'select into/bulkcopy/pllsort' turned ON for database
'tempdb'.
Run the CHECKPOINT command in the database that was changed.
(return status = 0)
Database option 'select into/bulkcopy/pllsort' turned ON for database
'aggs'.
Run the CHECKPOINT command in the database that was changed.
(return status = 0)
```

25.6. Заключение

В главе содержатся примеры по автоматизации задач с использованием конструкции "документ здесь". Конструкцию "документ здесь" можно применять при решении большого количества задач, особенно при установке подключения к приложениям или выполнении команды `ftping`. Приведенные сценарии можно выполнять и модифицировать для решения конкретных задач конечного пользователя.

ГЛАВА 26

Утилиты интерпретатора shell

В этой главе рассматриваются следующие темы:

- создание датируемых имен файлов и временных файлов;
- сигналы;
- команда `trap` и способы перехвата сигналов;
- команда `eval`;
- команда `logger`.

26.1. Создание регистрационных файлов

Используя любой сценарий, нужно создавать временные файлы или файлы регистрации (журнальные файлы). При выполнении сценариев, создающих резервные копии, удобно сохранять журналы фактических резервных копий. Обычно журналы хранятся в файловой системе несколько недель, затем происходит их очистка.

В процессе разработки сценариев непрерывно создаются временные файлы. Временные файлы также необходимы при функционировании обычных сценариев, содержащих информацию, которая ранее использовалась при вводе данных для другого процесса. Чтобы отобразить временный файл на экране или вывести его на печать, можно применить команду `cat`.

26.1.1. Применение команды date для создания журнальных файлов

Если возникла необходимость создать журнальный файл, желательно сделать его уникальным. Для этого достаточно воспользоваться командой `date`. Командой `date` можно манипулировать, а также добавлять ее к имени файла, который станет журнальным файлом.

Для изменения формата отображения данных применяется следующий формат:

```
date option + %format
```

С помощью знака плюс (+) можно в различных форматах отображать текущую дату. Ниже дата отображается в формате `день,месяц,год`.

```
$ date +%d%m%y
090699
```

Приведем некоторые наиболее часто применяемые форматы данных.

```
$ date +%d-%m-%y
09-06-99

$ date +%A%e" "%B" "%Y
Wednesday 9 June 1999
```

Отображение времени в формате чч:мм:

```
$ date +%R
10:07

$ date +%A" "W "%p
Wednesday 10:09 AM
```

Отображение времени в расширенном формате:

```
$ date +%T
10:29:41

$ date +%A" "%T
Wednesday 10:31:19
```

Обратите внимание на применение двойных кавычек, которые позволяют в потоке данных вывода указывать пробелы.

Для использования даты в качестве части имени файла проще всего воспользоваться подстановкой. Введите переменную, значением которой является заново отформатированная дата, и присоедините эту переменную к имени файла. Этим именем и будет назван журнальный файл.

В следующем примере создаются два журнальных файла. Для одного файла дата указывается в формате дд,мм,гг, а для другого — в формате дд,чч,мм.

Сценарий имеет вид:

```
$ pg log
#!/bin/sh
# регистрация
#
MYDATE=`date +%d%m%y`
# присоединение MYDATE к переменной LOGFILE, которая содержит
действительное имя файла регистрации.
LOGFILE=/logs/backup_log.$MYDATE
# создание файла
>$LOGFILE

MYTIME=`date +%d%R`
LOGFILE2=/logs/admin_log.$MYTIME
# создание файла
>$LOGFILE2
```

При выполнении этого сценария создаются два журнальных файла.

```
backup_log.090699
admin_log.0910:18
```

26.1.2. Создание уникальных временных файлов

При рассмотрении специальных переменных уже обсуждалась переменная *$$*. Она содержит ID или номер процесса, выполняющегося в текущий момент. Эти сведения применяются при создании временных файлов в текущем сценарии, поскольку ID процесса уникален в рамках сценария. Достаточно лишь создать файл и присоединить к нему символ *$$*. После завершения можно удалить все файлы, имеющие символ *$$* в качестве расширения. Интерпретатор shell оценивает переменную *$$* как текущий ID процесса и удаляет эти файлы, но оставляет файлы, к которым присоединен ID процесса.

В командной строке введите следующую команду:

```
$ echo $$
281
```

Это значение ID процесса приведено для примера; в вашем случае, скорее всего, будет получено другое значение. Если создать новый сеанс и ввести эту же команду, получится другой номер, поскольку будет запущен иной процесс.

```
$ echo $$
382
```

Ниже приводится сценарий, который создает два временных файла, обрабатывает и затем удаляет их.

```
$ pg tempfiles
#!/bin/sh
# tempfiles
# именование временных файлов
HOLD1=/tmp/hold1.$$
HOLD2=/tmp/hold2.$$

# выполнение определенной обработки с помощью этих файлов
df -tk >$HOLD1
cat $HOLD1 >$HOLD2
# удаление файлов
rm /tmp/*.$$
```

При выполнении этого сценария создаются следующие два файла.

```
hold1.408
hold2.408
```

Когда указывается команда `rm /tmp/*.$$`, интерпретатор shell в действительности выполняет команду `rm /tmp/*.408`.

Важно помнить, что ID процесса является уникальным только в конкретный момент времени. Например, если приведенный выше сценарий выполнить снова, получим новый ID процесса, поскольку речь идет о другом процессе.

Благодаря использованию даты можно отслеживать файлы, созданные для специальных целей. Помимо этого, значительно облегчается очистка файлов на базе определенных дат, поскольку с первого взгляда видно, какие файлы создавались раньше, а какие позже.

Временные файлы создаются легко и быстро; кроме того, они являются уникальными для данного процесса. После того как сценарий завершает обработку, их несложно удалить без искажения остальной информации.

26.2. Сигналы

Сигнал относится к типу сообщений, которые пересылаются из системы для информирования команды или сценария о совершении какого-либо события. Обычно речь идет об ошибках, связанных с функционированием памяти, о проблемах с доступом к информации или об определенных пользовательских попытках прекратить процесс. Сигналы представлены числами. Ниже приводится список наиболее распространенных сигналов и их значений.

Номер сигнала	Название сигнала	Значение
1	SIGHUP	"Зависание" или прекращение выполнения родительского процесса
2	SIGINT	Прерывание с помощью клавиатуры; обычно используется комбинация клавиш [Ctrl+C]
3	SIGQUIT	Завершение выполнения с помощью клавиатуры
9	SIGKILL	Прекращение выполнения определенного процесса
11	SIGSEGV	Нарушение сегментации (память)
15	SIGTERM	Завершение выполнения программы (завершение выполнения программы, заданное по умолчанию)

Существует сигнал 0, который ранее уже рассматривался (при создании файла *.logout*). Этот сигнал является сигналом "выхода из интерпретатора shell". Чтобы переслать сигнал 0, введите в командную строку команду `exit` либо примените к процессу или укажите в командной строке комбинацию клавиш [Ctrl+D].

Для пересылки сигнала используется формат:

```
kill [- номер сигнала:| имя сигнала] ID процесса
```

Если команда `kill` вводится без указания номера или названия сигнала, то она по умолчанию относится к сигналу с номером 15. Для просмотра списка всех сигналов примените следующую команду:

```
$ kill -1
 1) SIGHUP     2) SIGINT    3) SIGQUIT    4) SIGILL
 5) SIGTRAP    6) SIGIOT    7) SIGBUS     8) SIGFPE
 9) SIGKILL   10) SIGUSR1 11) SIGSEGV  12) SIGUSR2
13) SIGPIPE   14) SIGALRM 15) SIGTERM  17) SIGCHLD
18) SIGCONT   19) SIGSTOP 20) SIGTSTP  21) SIGTTIN
22) SIGTTOU   23) SIGURG  24) SIGXCPU  25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO
30) SIGPWR
```

26.2.1. Уничтожение процесса

При пересылке сигнала 1 процесс заново считывает файл конфигурации. Например, если при выполнении процесса под названием демон DNS файлы базы данных изменяются, не следует уничтожать демон и заново запускать его. Выполните всего лишь команду `kill -1`. В этом случае файлы конфигурации просматриваются заново.

Ниже приводится пример пересылки сигнала 9 (гарантированное уничтожение) для уничтожения процесса `mon_web`, который выполняется в системе. Сначала примените команду `ps` для создания процесса.

```
$ ps -ef | grep mon_web |grep -v root
157  ?  S    0:00 mon_web
```

Если в системе не поддерживается команда `ps -ef`, то воспользуйтесь командой `ps xa`. Для уничтожения процесса можно применить команду:

```
kill -9 157
```

или

```
kill -s SIGKILL 157
```

В некоторых системах можно не указывать опцию `-s`. Например, введите команду

```
kill SIGKILL 157
```

В приведенном ниже сценарии уничтожение процесса основано на использовании имени процесса. Имя устраняемого процесса указывается в виде параметра. Выполняется проверка, что данный процесс был действительно уничтожен. Утилита `grep` используется для обнаружения всех совпадающих имен процесса. Если соответствующие имена обнаруживаются, поступает запрос пользователю, следует ли уничтожить найденные процессы. Для уничтожения процесса применяется команда `kill -9`.

Соответствующий сценарий выглядит следующим образом:

```
$ pg pskill
#!/bin/sh
# pskill
HOLD1=/tmp/hold1.$$
PROCESS=$1
usage()
{

# usage
echo "Usage :'`basename $0` process_name"
exit 1
}
if [ $# -ne 1 ]; then
  usage
fi

case $1 in
*)
# применение программы grep для исключения нашего сценария из вывода команды ps
# извлечение полей 1 и 6, перенаправление во временный файл
  ps x | grep $PROCESS | grep -v $0 | awk '{print $1"\t" $6}'>$HOLD1
```

```
  # ps -ef |.. если команда ps x не срабатывает
  ;;
esac

# есть ли файл??
if [ ! -s $HOLD1 ]; then
  echo "No processes found..sorry"
  exit 1
fi

# просмотр содержимого временного файла и отображение значений полей
while read LOOP1 LOOP2
do
  echo $LOOP1 $LOOP2
done <$HOLD1
echo -n "Are these the processes to be killed ? [y..n] >"
read ANS

case $ANS in
Y|y) while read LOOP1 LOOP2
     do
       echo $LOOP1
       kill -9 $LOOP1
     done <$HOLD1
  rm /tmp/*.$$
  ;;
N|n);;
esac
```

При выполнении сценария поток вывода имеет вид:

```
$ pskill web
1760 ./webmon
1761 /usr/apps/web_col
Are these the processes to be killed ? [y..n] >y

1760
1761
[1]+ Killed webmon
```

Чтобы убедиться в том, что процесс уничтожен, введите команду повторно:

```
$ pskill web
No processes found..sorry
```

26.2.2. Обнаружение сигнала

Некоторые сигналы можно захватить и выполнить соответствующие действия. Другие сигналы нельзя уловить. Например, если команда получает сигнал 9, пользователю не нужно предпринимать какие-либо действия.

Если ограничиться написанием сценариев, следует обращать внимание только на сигналы 1, 2, 3 и 15. Если сценарий получает сигнал, возможен один из трех вариантов дальнейших действий:

1. Ничего не предпринимать, система самостоятельно отреагирует на полученный сигнал.

2. Захватить сигнал, но игнорировать его.
3. Захватить сигнал и предпринять определенные действия.

Большинство сценариев используют сигнал с номером 1. Этот метод применяется далее в книге во всех сценариях.

Чтобы применить два других метода, нужно воспользоваться командой `trap`.

26.3. Команда trap

Команда `trap` позволяет перехватывать сигналы. Формат команды `trap`:

```
trap "имя" сигнал(ы)
```

`Имя` представляет собой функцию, содержащую инструкции, которые выполняются при перехвате сигнала. В действительности функцию следует называть так, чтобы ее имя было связано именно с перехваченными сигналами. Имя следует заключать в двойные кавычки (" "). Сигналы являются входящими.

Когда сигнал перехватывается, сценарий обычно находится в стадии обработки. Самые распространенные действия следующие:

1. Очистить временные файлы.
2. Игнорировать сигналы.
3. Запросить пользователя, следует ли завершить сценарий.

Ниже приводится таблица, где описаны наиболее распространенные варианты применения команды `trap`:

`trap "" 2 3`	Игнорирование сигналов 2 и 3; пользователь не может завершить сценарий
`trap "команды" 2 3`	Если захвачены сигналы 2 и 3, выполняются команды
`trap 2 3`	Восстановление сигналов 2 и 3; пользователь может завершить сценарий

Вместо двойных кавычек можно использовать одинарные; результат будет аналогичен.

26.3.1. Перехват сигналов и выполнение действий

А теперь создадим сценарий, выполняющий подсчет итераций до тех пор, пока пользователь не нажмет комбинацию клавиш [Ctrl+C] (сигнал 2). После этого сценарий отобразит сообщение, содержащее номер текущей итерации цикла.

В этом случае применяется следующий формат:

```
trap "какие-либо действия" номер сигнала:(s)
```

Соответствующий сценарий имеет вид:

```
$ pg trap1
#!/bin/sh
#trap1
trap "my_exit" 2
LOOP=0
my_exit()
{
```

```
echo "You just hit <CTRL-C>, at number $LOOP"
echo " I will now exit "
exit 1
}

while :
do
  LOOP=`expr $LOOP + 1`
  echo $LOOP
done
```

Рассмотрим сценарий более подробно.

```
Trap "my_exit" 2
```

В результате выполнения команды `trap` после получения сигнала 2 выполняется команда, заключенная в двойные кавычки; в данном случае вызывается функция `my_exit`.

```
my_exit()
{
echo "You just hit <CTRL-C>, at number $LOOP"
echo " I will now exit "
exit 1
}
```

Функция `my_exit` вызывается при получении сигнала 2; при этом отображается значение переменной `$LOOP`, информирующее пользователя о том, какая итерация цикла выполнялась при нажатии комбинации клавиш [Ctrl+C]. Функции подобного типа применяются на практике для удаления временных файлов.

При выполнении сценария получим следующие результаты:

```
$ trap1
1
...
...
211
212
You just hit <CTRL-C>, at number 213
I will now exit
```

26.3.2. Захват сигнала и выполнение действий

Наиболее часто выполняемым действием является удаление временных файлов.

В следующем сценарии с помощью команд `df` и `ps` непрерывно добавляется информация во временные файлы *HOLD1.$$* и *HOLD2.$$*. Не забывайте, что символы *$$* заменяют ID процесса. Когда пользователь нажимает комбинацию клавиш [Ctrl+C], эти файлы удаляются.

```
$ pg trap2
#!/bin/sh
# trap2
# перехват только сигнала 2....<CTRL-C>
trap "my_exit" 2
HOLD1=/tmp/HOLD1.$$
```

```
HOLD2=/tmp/HOLD2.$$

my_exit()
{
# my_exit
echo "<CTRL-C> detected..Now cleaning up..wait"
# удаление временных файлов
rm /tmp/*.$$ 2> /dev/null
exit 1
}

echo "processing...."
# основной цикл
while :
do
  df >>$HOLD1
  ps xa >>$HOLD2
done
```

Результаты выполнении этого сценария будут следующими:

```
$ trap2
processing....
<CTRL-C> detected..Now cleaning up..wait
```

При получении сигнала можно предоставлять пользователю определенный выбор. Однако если получены сигналы 2 или 3, следует убедиться, что они не появились случайно. Необходимо предусмотреть поток ошибок, благодаря чему выявляется ошибочное нажатие клавиш [Ctrl+C].

В следующем примере пользователю предоставляется возможность решить, следует ли выйти из сценария после получения сигнала 2. Отображается запрос о том, действительно ли пользователь желает завершить работу. Для уточнения предпринимаемого действия применяется конструкция `case`.

Если пользователь желает выйти из сценария, он выбирает 1, в результате чего после отображения статуса "exit 1" запускается процесс очистки. Если пользователь не желает выходить из сценария, никакие действия не производятся; зададим, что при выполнении конструкции `case` результаты выбора будут неудачными и произойдет возврат к исходному коду. Конечно, при подтверждении должны захватываться значения всех пустых полей.

Далее приводится функция, которая при получении сигнала предоставляет пользователю возможность выбора.

```
my_exit ()
{
# my_exit
echo -e "\nReceived interrupt ..."
echo "Do you really wish to exit ???"
echo " 1: Yes"
echo " 2: No"
echo -n " Your choice [1..2] >"
read ANS
case $ANS in
1) # удаление временных файлов.. и т.д...
  exit 1
```

```
  ;;
2) # не выполняет ничего
3) ;;
esac
}
```

Соответствующий сценарий выглядит так:

```
$ pg trap4
#!/bin/sh
# trap4
# перехват сигналов 1 2 3 и 15
trap "my_exit" 1 2 3 15

LOOP=0

# временные файлы
HOLD1=/tmp/HOLD1.$$
HOLD2=/tmp/HOLD2.$$

my_exi t()
{
# функция my_exit
echo -e "\nRecieved interrupt..."
echo "Do you wish to really exit ???"
echo " Y: Yes"
echo " N: No"
echo -n " Your choice [Y..N] >"
read ANS
case $ANS in
Y|y) exit 1;; # выход из сценария
N|n) ;; # возврат к обычной обработке
esac
}

# цикл while применяется здесь, например, для просмотра полей
echo -n "Enter your name :"
read NAME
echo -n "Enter your age :"
read AGE
```

Если при выполнении этого сценария происходит нажатие клавиш [Ctrl+C] в середине поля ввода (сразу после начала ввода имени), то пользователю предоставляется выбор: возвратиться к обычной обработке или выйти из сценария.

```
$ trap4
Enter your name :David Ta
Received interrupt...
Do you really wish to exit ???
1: Yes
2: No
Your choice [1..2] >2
Enter your age :
```

26.3.3. Блокировка терминала

Ниже приводится сценарий, в котором предлагается другой путь для перехвата сигналов в функционирующем сценарии. Сценарий `lockit` блокирует терминал пользователя с помощью командной строки. При этом командная строка помещается в непрерывный цикл `while`. Команда `trap` захватывает сигналы 2, 3 и 15. Если пользователь пытается прервать выполнение сценария, отображается сообщение о том, что действия пользователя не были успешными.

При первом обращении к сценарию запрашивается пароль. Для отмены блокировки терминала сведения поступают из устройства */dev/tty*, следовательно, отсутствует запрос на разблокировку терминала; нужно просто ввести пароль и нажать клавишу ввода.

Если пользователь забыл пароль, нужно зарегистрироваться на другом терминале и устранить сценарий. Проверка длины пароля не производится, но при желании ее можно установить.

Если сценарий уничтожен с помощью другого терминала, можно вернуться на ваш терминал и разобраться с проблемами, связанными с настройками терминала: например, выяснить, почему не функционирует клавиша ввода. По командному запросу воспользуйтесь приведенной подсказкой. Это позволит устранить большую часть затруднений, связанных с терминалом.

```
$ stty sane
```

Сценарий имеет вид:

```
$ pg lockit
#!/bin/sh

# lockit
# перехват сигналов 2 3 и 15
trap "nice_try" 2 3 15

# устройство, на котором выполняется сценарий
TTY=`tty`

nice_try()
{
# nice_try
echo "Nice try, the terminal stays locked"
}

# сохраните настройки stty, скрытие символов при вводе пароля
SAVEDSTTY=`stty -g`
stty -echo
echo -n "Enter your password to lock $TTY :"
read PASSWORD
clear

while :
do
  # чтение только из tty !!
  read RESPONSE < $TTY
  if [ "$RESPONSE" = "$PASSWORD" ]; then
```

```
    # пароль соответствует...разблокировка
    echo "unlocking..."
    break
  fi

  # отображение сообщения, если пользователь введет неверный пароль
  # или нажмет клавишу ввода

  echo "wrong password and terminal is locked.."
done

# восстановление настроек stty
stty $SAVEDSTTY
```

Вывод сценария `lockit`:

```
$ lockit
Enter your password to lock /dev/tty1 :
```

Затем экран очищается. При нажатии клавиши ввода или наборе неверного пароля сценарий выводит следующие данные:

```
wrong password and terminal is locked..
Nice try, the terminal stays locked
wrong password and terminal is locked..
Nice try, the terminal stays locked
wrong password and terminal is locked..
```

Введите правильный пароль

```
unlocking...
$
```

Теперь возвращаемся обратно, в командную строку.

26.3.4. Игнорирование сигналов

Когда пользователь регистрируется в системе, просматривается файл */etc/profile*; нежелательно, чтобы пользователь прерывал этот процесс. Обычно задается перехват, или игнорирование, сигналов 1, 2, 3 и 15, но потом при просмотре сообщения *motd* (ежедневного сообщения) их подключают вновь (восстанавливают). Затем для игнорирования сигналов 1, 2, 3 и 15 снова устанавливается перехват.

Аналогичный подход можно реализовать при работе со сценариями. Можно ввести понятие критического момента, который наступает при открытии большого количества файлов. Начиная с этого момента, нельзя прерывать выполнение сценария, поскольку это может повредить файлы. Для решения этой проблемы следует установить команду `trap`, что позволит игнорировать некоторые сигналы. Когда завершится критический момент в функционировании сценария, примените команду `trap`, чтобы снова можно было захватывать сигналы.

Для игнорирования входящих сигналов (кроме сигнала 9) применяется следующая команда:

```
trap "" номер сигнала:(s)
```

Обратите внимание, что двойные кавычки ничего не содержат. Чтобы восстановить перехват и заново захватывать сигналы, выполните команду:

```
trap "любые действия" номер сигнала:(s)
```

Суммируем сведения о процессах игнорирования и выявления сигналов.

```
trap "" 1 2 3 15 # игнорирование сигналов
code that does really critical stuff
trap "my_exit" 1 2 3 15 # выполните снова захват сигналов с помощью функции
                        # my_exit
```

Обратите внимание на сценарий, выполняющий критическую обработку. Цикл `while` аккуратно передвигает имеющийся критический момент. Для игнорирования сигналов 2, 3 и 15 применяется команда `trap`. После завершения цикла `while`, завершается еще один цикл `while`, но перехват уже восстановлен и прерывания разрешены.

Оба цикла `while` выполняются до завершения шести итераций, затем в цикле активизируется команда `sleep`. Благодаря этому имеется достаточно времени для того, чтобы прервать выполнение сценария.

Рассмотрим следующий сценарий:

```
$ pg trap_ignore
#!/bin/sh
# trap_ignore
# игнорирование сигналов
trap "" 1 2 3 15

LOOP=0
my_exit()
# my_exit
{
echo "Received interrupt on count $LOOP"
echo "Now exiting..."
exit 1
}

# критическая обработка, нельзя прерывать....
LOOP=0
while :
do
  LOOP=`expr $LOOP + 1`
  echo "critical processing..$LOOP..you cannot interrupt me"
  sleep 1
  if [ "$LOOP" -eq 6 ]; then
    break
  fi
done

LOOP=0
# критическая обработка завершена, перехват задан снова, но разрешены прерывания
trap "my_exit" 1 2 3 15

while :
```

```
do
  LOOP=`expr $LOOP + 1`

  echo "Non-critical processing..$LOOP..interrupt me now if you want"
  sleep 1
  if [ "$LOOP" -eq 6 ]; then
    break
  fi
done
```

Если в процессе работы этого сценария попытаться нажать клавиши [Ctrl+C] во время выполнения первого цикла "критической обработки", ничего не произойдет. Это связано с тем, что была введена команда `trap` для игнорирования сигналов.

После того как начинается второй цикл "некритической обработки", статус команды `trap` восстанавливается, в результате чего разрешаются прерывания.

```
$ trap_ignore
critical processing..1..you cannot interrupt me
critical processing..2..you cannot interrupt me
critical processing..3..you cannot interrupt me
critical processing..4..you cannot interrupt me
critical processing..5..you cannot interrupt me
critical processing..6..you cannot interrupt me
Non-critical processing..1..interrupt me now if you want
Non-critical processing..2..interrupt me now if you want
Received interrupt on count 2
Now exiting...
```

Благодаря применению команды `trap` можно обрести большую степень контроля над "поведением" сценария при получении сигнала. Перехват и последующая обработка сигналов обеспечивают устойчивую работу сценариев.

26.4. Команда eval

Команда `eval` служит для оценки командной строки с целью завершения каких-либо подстановок интерпретатора shell с последующим их вызовом. Команда `eval` используется для расширения значений переменных (если в результате одного прохода расширения не происходит, выполняется второй проход). Переменные, для оценки которых требуется два прохода, иногда называют сложными переменными. Хотя, по моему мнению, переменные не могут быть сложными.

Команда `eval` также может применяться для отображения значений простых переменных; эти переменные не должны быть сложными.

```
$ NAME=Honeysuckle
$ eval echo $NAME
Honeysuckle
$ echo $NAME
Honeysuckle
```

Наилучшим методом для понимания работы команды `eval` является изучение оценки командной строки, выполняемой этой командой.

26.4.1. Выполнение команд, находящихся в строке

Вначале создадим небольшой файл, *testf*, содержащий некоторый текст. Затем переменной MYFILE будет присвоена команда `cat testf` с последующим отображением значения переменной. Благодаря этому можно проверить возможность отображения на экране содержимого файла *testf*.

```
$ pg testf
May Day, May Day
Going Down
```

Присвоим переменной MYFILE строку "cat testf":

```
$ MYFILE="cat testf"
```

Если требуется вывести содержимое файла *testf* на экран, нужно выполнить команду `cat testf`.

```
$ echo $MYPILE cat testf
```

Теперь применим команду `eval` для оценки значения переменной; при этом следует помнить, что `eval` выполняет два прохода при оценке переменной MYFILE.

```
$ eval $MYFILE
May Day, May Day
Going Down
```

В результате этого успешно оценивается значение переменной и применяется команда `cat` к файлу `testf`. При первом проходе отображается фактическая строка "cat testf"; при втором проходе выполняется содержимое строки, в данном случае `cat testf`.

Рассмотрим другой пример. Переменной CAT_PASSWD будет присвоена строка `"cat /etc/passwd | more"`. Команда `eval` выполняет оценку содержимого данной строки.

```
$ CAT_PASSWD="cat /etc/passwd | more"
$ echo $CAT_PASSWD
cat /etc/passwd|more
$ eval $CAT_PASSWD
root:HccPbzT5tbOOg:0:0:root:/root:/bin/sh
bin:*:1:1:bin:/bin:
daemo n:*:2:2:daemon:/sbin:
adm:*:3:4:ada:/var/adm:
...
...
```

Команда `eval` также хорошо подходит для отображения значения последнего параметра, передаваемого сценарию. Значение последнего параметра уже отображалось, но в этом случае оно будет отображено снова.

```
$ pg evalit
#!/bin/sh
# сценарий evalit
echo " Total number of arguments passed is $#"
echo " The process ID is $$"
echo " Last argument is " $(eval echo \$$#)
```

При выполнении этого сценария получим следующий результат (ID процесса может отличаться от случая к случаю):

```
$ evalit alpha bravo charlie
Total number of arguments passed is 3
The process ID is 780
Last argument is charlie
```

В этом сценарии команда eval сначала оценивает значение переменной $$# в сравнении с ID процесса, а при выполнении второго прохода производится оценка последнего параметра, передаваемого переменной.

26.4.2. Присвоение значения имени переменной

Можно также поставить в соответствие полю данных имя переменной. Рассмотрим, что это означает на практике. Предположим, что в нашем распоряжении имеется следующий файл:

```
$ pg data
PC       486
MONITOR svga
NETWORK  yes
```

Нам необходимо, чтобы первому столбцу текста соответствовали имена переменных, а второму столбцу текста — значения соответствующих переменных. При этом должно отображаться следующее:

```
echo $PC
486
```

Как же можно достичь желаемого результата? Ниже приведен соответствующий сценарий, использующий команду eval.

```
$ pg eval_it
#!/bin/sh
#сценарий eval_it
while read NAME TYPE
do
  eval `echo "${NAME}=${TYPE}"`
done < data
echo "You have a $PC pc, with a $MONITOR monitor"
echo "and have you network ? $NETWORK"
```

Рассмотрим, как функционирует сценарий. Сначала берутся значения PC и 486, которые присваиваются переменным NAME и TYPE соответственно. При первом проходе команда eval отображает на экране два значения переменных, PC и 486; во время выполнения второго прохода вместо переменной NAME подставляется значение PC, а вместо переменной TYPE — значение 486. При выполнении сценария получаются следующие результаты:

```
$ eval_it
You have a 486 pc, with a svga monitor
and have you network ? yes
```

Команда eval не слишком часто применяется в сценариях, однако ее удобно использовать при оценке значения переменной, выполняемой более одного раза.

26.5. Команда logger

В системе поддерживается достаточно много журнальных файлов. Некоторые из них, именуемые `messages`, обычно размещены в каталоге */var/adm* или */var/log*. Сообщения, регистрируемые в этом файле, передаются с помощью файла конфигурации *syslog* и имеют строго заданный формат. Чтобы убедиться, что система сконфигурирована для генерирования сообщений из программ, проверьте файл */etc/syslog.conf*. Этот файл содержит описания приоритетов и свойств, которые программа может использовать для отсылки различных типов сообщений.

Здесь мы не будем подобно рассматривать, каким образом UNIX либо Linux регистрирует сообщения в файле. Все, что вам требуется пока знать, — это номера различных уровней сообщений (от информационных до критических).

Сообщения могут отсылаться в файл с помощью команды `logger`. Перед тем как использовать эту команду, нужно обратиться к справочной странице `man`, так как различные версии этой команды могут иметь отличающийся синтаксис. Поскольку в этой главе рассматриваются только информационные сообщения, нужно обратить внимание на соответствующие команды, рассматриваемые далее.

Необходимость отправки сообщений в файл диктуется одной из следующих причин:

- количество попыток доступа/регистраций за определенный период;
- критическая обработка одного из сбойных сценариев;
- мониторинг отчетов сценариев.

Ниже показано, как выглядит файл */var/adm/messages*. Формат этого файла может немного отличаться от данного образца:

```
$ tail /var/adm.messages
Jun 16 20:59:03 localhost login[281]: DIALUP AT ttyS1 BY root
Jun 16 20:59:03 localhost login[281]: ROOT LOGIN ON ttyS1
Jun 16 20:59:04 localhost PAM_pwdb[281]: (login) session closed for user root
Jun 16 21:58:38 localhost named[211: Cleaned cache of 0 RRs]
Jun 16 21:58:39 localhost named[211]: USAGE 929570318 929566719
Jun 16 21:58:39 localhost named[211]: NSTATS 929570318 929566719
```

Общий формат команды `logger` выглядит так:

```
logger -p -i message
```

Параметры этой команды выполняют следующие функции:

`-p` Определяет приоритет; в данном случае затрагивается только файл *user.notice*, который всегда является используемым по умолчанию

`-i` Регистрирует ID процесса для каждого сообщения

26.5.1. Использование команды logger

В командной строке интерпретатора shell введите следующую команду:

```
$ logger -p notice "This is a test message.Please Ignore $LOGNAME"
```

Возможно, вам придется подождать пару минут, пока не отобразится информация о регистрации сообщения.

```
$ tail /var/adm/messages
...
...
Jun 17 10:36:49 acers6 dave: This is a test message.Please Ignore dave
```

Из приведенного примера видно, что регистрируется пользователь, выполняющий регистрацию сообщения.

А теперь создадим небольшой сценарий, регистрирующий сообщение, в котором говорится о количестве пользователей в системе. Данный сценарий может использоваться для оценки интенсивности ежедневной загрузки системы. Для этого нужно просто запускать его из файла *crontab* примерно каждые 30 минут.

```
$ pg test_logger
#i/bin/sh
# test_logger
logger -p notice "`basename $0`:there are currently `who |wc -l` users on
the system"
```

Запустим сценарий.

```
$ test_logger
```

Теперь отобразим содержимое файла сообщений.

```
$ tail /var/adm/messages
Jun 17 11:02:53 acers6 dave: test_script:there are currently 15 users on
the system
```

26.5.2. Использование команды logger в сценариях

Регистрацию сообщений лучше использовать в том случае, когда осуществляется безусловное прерывание выполнения одного из сценариев. Для регистрации этих типов сообщений просто включите команду `logger` в функции, выполняющие перехват сигналов при выходе из сценария.

В следующем сценарии очистки при получении любого из сигналов с номерами 2, 3 или 15 производится регистрация сообщения.

```
$ pg cleanup
# !/bin/sh
# cleanup
# очистка журнальных файлов системы
trap "my_exit" 2 3 15

my_exit()
{
# my_exit
logger -p notice " `basename $0`: Was killed whilst cleaning up system
logs..CHECK OUT ANY DAMAGE"
exit 1
}

tail -3200c /var/adm/utmp > /tmp/utmp
mv /tmp/utmp /var/adm/utmp
>/var/adm/wtmp
#
tail -10 /var/adm/sulog > /tmp/o_sulog
mv /tmp/o_sulog /var/adm/sulog
...
```

При просмотре файла сообщений можно заметить, что возникла проблема, связанная с выполнением сценария очистки.

```
$ tail /var/adm/messages
...
Jun 17 11:34:28 acers6 dave: cleanup:Was killed whilst cleaning up
systemlogs.. CHECK OUT ANY DAMAGE
```

Помимо использования при работе с различными критическими сценариями, команду `logger` можно также применять для регистрации любых подключений удаленных пользователей к системе. Ниже приведен сегмент кода, регистрирующий пользователей, которые подключаются к системе с помощью последовательных линий `tty0` и `tty02`. Этот фрагмент кода берет свое начало от одного из файлов /etc/profile.

```
TTY_LINE=`tty`
case $TTY_LINE in
"/dev/tty0")
  TERM=ibm3151
  ;;
"/dev/tty2")
  TERM=vt220
  # проверка пользователей, которым разрешен доступ к модемной линии
  #
  echo "This is a modem connection"
  # modemf содержит регистрационные имена для допустимых пользователей
  modemf=/usr/local/etc/modem.users
  if [-s $modemf ]
  then
    user=`cat $modemf| awk '(print $1}' | grep SLOGNAME`
    # если имя не содержится в файле, пользователь не допускается в систему
    if [ "$user" != "$LOGNAME" ]
    then
      echo "INVALID USER FOR MODEM CONNECTION"
      echo " DISCONNECTING,,,,,,"
      sleep 1
      exit 1
    else
      echo "modem connection allowed"
    fi
  fi
  logger -p notice "modem line connect $TTY_LINE..$LOGNAME"
  ;;
*) TERM=vt220
  stty erase '^h'
  ;;
esac
```

Команда `logger` является превосходным инструментальным средством, применяемым для регистрации информации в глобальных файлах сообщений системы.

26.6. Заключение

Благодаря использованию функции перехвата и сигналов реализуется изящное завершение выполнения сценариев. Возможность регистрации сообщений в системном журнальном файле обеспечивает пользователей и администраторов полезной информацией, облегчающей распознавание и устранение любых потенциальных проблем.

ГЛАВА 27

Небольшая коллекция сценариев

В настоящей главе содержатся примеры некоторых наиболее распространенных сценариев. Изучая их, можно заметить, что все они невелики по размеру и довольно просты. В этом и состоит преимущество использования сценариев; они не должны быть сложными и объемными, поскольку сценарии создаются с целью экономии времени пользователя.

Конечно, в состав данной главы неплохо было бы включить сценарий `comet`, выполняющий общую проверку баз данных. Но поскольку этот сценарий содержит более 500 строк, нецелесообразно включать его в эту небольшую книгу. Разработка сценария `comet` началась еще пару лет назад. Тогда этот сценарий состоял не более чем из пяти строк. Но в ходе естественного процесса эволюции величина сценария существенно выросла. Приведем перечень сценариев, рассматриваемых в данной главе:

`pingall`	Сценарий, использующий записи из файла */etc/hosts* для выполнения опроса всех хостов
`backup_gen`	Общий сценарий резервного копирования, который загружает заданные по умолчанию настройки
`del.lines`	Оболочка потокового редактора `sed`, выполняющая удаление строк из файлов
`access_deny`	Утилита, реализующая запрет доступа для определенных пользователей при выполнении регистрации
`logroll`	Утилита, реализующая прокрутку журнального файла в случае, если он достигает определенного размера
`nfsdown`	Утилита, реализующая быстрый метод демонтирования всех каталогов `nfs`

27.1. Сценарий pingall

Еще несколько лет назад сценарий `pingall` представлял собой часть общего сценария отчета, который выполнялся по ночам. Этот сценарий опрашивает все хосты, записи о которых находятся в файле *hosts*.

Сценарий реализует просмотр файла */etc/hosts* и разыскивает все строки, которые не начинаются с символа `#`. Затем цикл `while` считывает строки отфильтрованного текста. Для присваивания переменной `ADDR` значения первого поля отфильтрованного текста используется утилита `awk`. Затем с помощью цикла `for` по каждому найденному адресу отправляется запрос.

Ниже приводится сам сценарий.

```
$ pg pingall
#!/bin/sh
# pingall

# просмотр файла /etc/hosts и отправка запроса по каждому адресу
cat /etc/hosts| grep -v '^#' | while read LINE
do
  ADDR=`awk '{print $1}'`
  for MACHINE in $ADDR
  do
    ping -s -c1 $MACHINE
  done
done
```

Сценарий `pingall` можно легко расширить и включить в него функции отчетов, связанные с другими сетевыми утилитами.

27.2. Сценарий backup_gen

Сценарий `backup_gen` приводится здесь вовсе не для иллюстрации методики резервирования каталогов. Этот сценарий является удачным примером совместного использования настроек, общих для нескольких сценариев.

Сценарий `backup_gen` предназначен для создания резервных копий. При выполнении сценария просматривается заданный по умолчанию файл конфигурации, который затем используется для резервирования системы. При желании пользователь может изменять настройки, заданные по умолчанию. Сценарий является отличным примером того, как различные сценарии могут применять одинаковые настройки или изменять их во время выполнения сценария. После запуска сценария выполняется проверка на наличие исходного файла (*backup.defaults*). Если этот файл не найден, сценарий завершается.

При выполнении сценария отображается заголовок экрана и настройки, заданные по умолчанию. Пользователю направляется запрос о том, требуется ли изменять какие-либо настройки, заданные по умолчанию. Если ответ положителен, поступает запрос на ввод кода, применяемого для изменения необходимых настроек. Для ввода правильного кода пользователю предоставляются три попытки; если введен неверный код, используются настройки, заданные по умолчанию. При вводе корректного кода пользователь может изменить приведенные ниже настройки (значения, заданные по умолчанию, содержатся в квадратных скобках []):

`tape device [rmt0]`	Можно выбрать `rmt1` и `rmt3`
`mail admin when the backup has finished [yes]`	Нет вариантов выбора
`type of backup [full]`	Можно выбрать опцию `normal` или `sybase`

Изменения настроек выполняются с помощью временных переменных. Для получения доступа к заданным по умолчанию настройкам установите курсор мыши в любом поле и нажмите клавишу [Return]. Однако следующие настройки изменять нельзя:

```
backup log filename
code name.
```

Все внесенные изменения затем подтверждаются. После завершения процесса подтверждения значения временных переменных снова присваиваются исходным переменным. До завершения резервного копирования выполняется тестирование магнитной ленты. В процессе резервного копирования применяются команды `find` и `cpio`. С помощью этих команд используются переменные из файла настроек или значения новых переменных, указанные пользователем.

Далее приводится соответствующий сценарий.

```
$ pg backup_run
#!/bin/sh
# backup_run

# сценарий выполнения резервного копирования
# загрузка файла с конфигурационными параметрами

SOURCE=/appdva/bin/backup.defaults
check_source()
(
# check_source
# файл backup.defaults содержит параметры конфигурации/функции
# проверка того, что путь содержит нужный каталог
if [ -r $SOURCE ]; then
  . /$SOURCE
else
  echo "`basename $0`: cannot locate defaults file"
  exit 1
fi
}

header()
{
# header
 USER=`whoami`
MYDATE=`date +%A" "%e" of "%B-%Y`
clear
cat << MAYDAY
User : $USER                                $MYDATE
                    NETWORK SYSTEM BACKUP
                    =====================
MAYDAY
}

change_settings()
{
# change_settings
# отображение параметров, заданных по умолчанию
header
echo "Valid Entries Are..."
echo "Tape Device: rmt0, rmt1, rmt3"
echo "Mail Admin: yes, no"
echo "Backup Type: full, normal, sybase "
while :
do
  echo -n -c "\n\n Tape Device To Be Used For This Backup [$_DEVICE] :"
```

```
  read T_DEVICE
  : ${T_DEVICE:=$_DEVICE}
  case $T_DEVICE in
  rmt0|rmt1|rmt3) break;;
  *) echo "The devices are either ... rmt0, rmt1, rmt3"
    ;;
  esac
done

# если пользователь нажимает клавишу ввода при установке курсора в любом
# из полей, применяются настройки, заданные по умолчанию
while :
do
  echo -n " Mail Admin When Done        [$_INFORM] :"
  read T_INFORM
  : ${T_INFORM:=$_INFORM}
  case $T_INFORM in
  yes|Yes) break;;
  no|No) break;;
  *) echo "The choices are yes, no"
    ;;
  esac
done

while :
do
  echo.-n " Backup Type                  [$_TYPE] :"
  read T_TYPE
  : ${T_TYPE:=$_TYPE}
  case $T_TYPE in,
  Full|full) break;;
  Normal|normal)break;;
  Sybase|sybase)break;;
  *) echo "The choices are either... full, normal, sybase"
  esac
done
# повторное присваивание значений временных переменных исходным переменным,
# которые были загружены
_DEVICE=$T_DEVICE; _INFORM=$T_INFORM; _INFORM=$T_INFORM
}

 show_settings()
# отображение текущих настроек
{
cat << MAYDAY
                    Default Settings Are...
            Tape Device To Be Used              : $_DEVICE
            Mail Admin When Done                : $_INFORM
            Type Of Backup                      : $_TYPE
            Log file of backup                  : $_LOGFILE
MAYDAY
}
```

```
get_code()
{
# пользователи имеют 3 попытки для ввода правильного кода
# _CODE загружается из исходного файла
clear
header
_COUNTER=0
echo " YOU MUST ENTER THE CORRECT CODE TO BE ABLE TO CHANGE DEFAULT
SETTINGS"
while :
do
  _COUNTER=`expr $_COUNTER + 1`
  echo -n "Enter the code to change the settings:"
  read T_CODE
  # echo $_COUNTER
  if [ "$T_CODE" = "$_CODE" ]; then
    return 0
  else
    if [ "$_COUNTER" -gt 3 ]; then
      echo "Sorry incorrect code entered, you cannot change the settings.."
      return 1
    fi
  fi
done
}

 check_drive()
{
# перемотка ленты
mt -f /dev/$_DEVICE rewind > /dev/null 2>&1
if [ $? -ne 0 ]; then
  return 1
else
  return 0
fi
}
#========== main==============

# чтение файла с параметрами
check_source
header
# отображение содержимого переменных
show_settings
# уточнение у пользователя, желает ли он изменить настройки
if continue_prompt "Do you wish To Change Some Of The System Defaults" "Y";
then
  # да, тогда введите имя
  if get_code; then
    # изменение параметров
    change_settings
  fi
fi

#------- параметры получены, резервное копирование
```

```
if check_drive; then
  echo "tape OK...."
else
  echo "Cannot rewind the tape..Is it in the tape drive ???"
  echo "Check it out"
  exit 1
fi

# что копировать
case $_TYPE in
 Full|full)
  BACKUP_PATH="sybase syb/support etc var bin apps use/local"
  ;;
Normal|normal)
  BACKUP_PATH="etc var bin apps usr/local"
  ;;
Sybase|sybase)
  BACKUP_PATH="sybase syb/support"
  ;;
esac
# резервное копирование
cd /
echo "Now starting backup......"
find $BACKUP_PATH -print | cpio -ovB -0 /dev/$_DEVICE >> $_LOGFILE 2>&1

# если приведенная выше команда cpio не выполняется в системе,
# воспользуйтесь командой cpio, приведенной ниже
# find $BACKUP_PATH -print | cpio -ovB > /dev/$_DEVICE >> $_LOGFILE 2>&1

# для получения дополнительной информации измените -ovB на -ovcC66536

if [ "$_INFORM" = "yes" ]; then
  echo "Backup finished check the log file" | mail admin
fi
```

Файл *backup.defaults* содержит заданные по умолчанию настройки наряду с функцией `continue_prompt`. Ниже приводится содержимое файла.

```
$ pg backup.defaults
#!/bin/sh
# backup.defaults
# файл конфигурации, заданный по умолчанию, для сетевых резервных копий
# отредактируйте этот на свой страх и риск!!
#----------------------
_CODE="comet"
_LOGFILE="/appdva/backup/log.`date +%y%m%d`"
_DEVICE="rmt0"
_INFORM="yes"
_TYPE="Full"
 #-----------------
continue_prompt()
# continue_prompt
# для вызова: continue_prompt "отображаемая строка" default_answer
{
```

```
_STR=$1
_DEFAULT=$2
# проверка ввода корректных параметров
if [ $# -lt 1 ]; then
  echo "continue_prompt: I need a string to display"
  return 1
fi
while :
do
  echo -n "$_STR [Y..N] [$_DEFAULT]:"
  read _ANS
  : ${_ANS:=$_DEFAULT}
    if [ "$_ANS" = "" ]; then
      case $_ANS in
      Y) return 0 ;;
      N) return 1 ;;
      esac
    fi
  # пользователь сделал выбор
  case $_ANS in
  y|Y|Yes|YES)
    return 0
    ;;
  n|N|No|NO)
    return 1
    ;;
  *) echo "Answer either Y or N, default is $_DEFAULT"
    ;;
  esac
  echo $_ANS
done
}
```

Ниже приводится поток вывода при отображении настроек, заданных по умолчанию, причем пользователя запрашивают, желает ли он изменить эти настройки:

```
User : dave                    Tuesday 15 of June-2000
               NETWORK SYSTEM BACKUP
               ======================
               Default Settings Are...
Tape Device To Be Used       :rmt0
Mail Admin When Done         :yes
Type Of Backup               :Full
Log file of backup           :appdva/backup/log.000615
Do you wish To Change Some Of The System Defaults [Y..N] [Y]:
```

Следующий поток вывода иллюстрирует процесс изменения значения для настроек, заданных по умолчанию. Здесь изменился тип резервного копирования, и когда сценарий проверяет ленточный накопитель, он обнаруживает определенные проблемы. Сценарий завершает работу с помощью кода завершения последней команды.

```
User : dave                    Tuesday 15 of June-2000
               NETWORK SYSTEM BACKUP
               ======================
Valid Entries Are...
```

```
Tape Device: rmt0, rmt1, rmt3
Mail Admin: yes, no
Backup Type: full, normal, sybase

Tape Device To Be Used For This Backup  [rmt0]:
Mail Admin When Done                    [yes]:
Backup Type                             [Full: Normal
Cannot rewind the tape..Is it in the tape drive ???

Check it out
```

27.3. Сценарий del.lines

О данном сценарии часто заходит речь, когда разработчики вопрошают: "Где же команда sed, выполняющая повторное удаление пустых строк?" Именно для этой цели и создан этот небольшой сценарий.

Действительно, сценарий представляет собой оболочку для потокового редактора sed. Поскольку этот процесс применяется довольно часто, разработчики заинтересованы в подобном сценарии.

Сценарии интерпретатора shell не должны быть большими. Создание сценариев целесообразно, если при автоматизации задач экономится время пользователя.

Сценарий del.lines может обрабатывать один или несколько файлов. До того, как команда sed приступит к удалению всех пустых строк, проверяется наличие каждого файла. Поток вывода команды sed с помощью символов *$$* направляется во временный файл. Затем файл перемещается обратно, заменяя исходный файл.

Чтобы просмотреть все имена файлов, применяется команда shift. Цикл while выполняется до тех пор, пока имеются обрабатываемые файлы.

Введите команду del.lines -help, в результате чего отобразится немного разреженная справочная строка. Желательно создать более удобную справочную конструкцию. Сценарий имеет следующий вид:

```
$ pg del.lines
#!/bin/sh

# del.lines
# сценарий получает имена файлов и удаляет из них все пустые строки

TEMP_F=/tmp/del.lines.$$

usage()
{
# usage
echo "Usage :`basename $0` file [file..]"
echo "try `basename $0` -help for more info"
exit 1
}

if [ $# -eq 0 ]; then
  usage
fi

FILES=$1
```

```
while [ $# -gt 0 ]
do
  echo "..$1"
  case $1 in
  -help) cat << MAYDAY
    Use this script to delete all blank lines from a text file(s)
    MAYDAY
    exit 0
    ;;
  *)FILE_NAME=$1

    if [ -f $1 ]; then
      sed '/^$/d' $FILE_NAME >$TEMP_F
      mv $TEMP_F $FILE_NAME

    else
      echo "`basename $0` cannot find this file : $1"
    fi
    shift
    ;;
  esac
done
```

27.4. Сценарий access.deny

Чтобы пользователи не регистрировались в системе при введении необходимых обновлений, можно воспользоваться методом */etc/nologin*, который доступен для большинства систем. Когда в каталоге */etc* создается файл *nologin*, обычно применяется команда `touch`. И никто из пользователей, кроме пользователя root, не может зарегистрироваться.

Если данная система не поддерживает метод *nologin*, не все потеряно — можно создать подобный метод самостоятельно. Ниже показано, как это осуществить на практике.

В файл */etc/profile* помещается следующий код:

```
if [ -f /etc/nologin ]; then
  if [ $LOGNAME != "root" ]; then
    echo "Sorry $LOGNAME the system is unavailable at the moment"
    exit 1
  fi
fi
```

Теперь, если требуется запретить регистрацию для всех пользователей, за исключением пользователя root, примените команду `touch` для создания в каталоге */etc* файла `nologin` и удостоверьтесь, что все пользователи имеют право читать этот файл.

```
touch /etc/nologin
chmod 644 /etc/nologin
```

Если необходимо вернуться к старому порядку, удалите файл `nologin` следующим образом:

```
rm /etc/nologin
```

Описанная методика используется для отключения всех пользователей, кроме пользователя root. Если нужно временно отключить некоторые учетные записи, можно обратиться к файлу */etc/passwd*, а в качестве первого символа в поле пароля следует указать символ *. Однако такой подход применяется редко, и если пользователь недостаточно четко выполняет все действия, можно столкнуться с проблемами при работе всей системы.

Linux располагает утилитой, с помощью которой в файл *login.access* вводятся имена пользователей и групп. Этот файл предназначен для предоставления доступа к системе.

Рассмотрим версию утилиты под названием *deny.access*. Сценарий, который выполняется из файла */etc/profile*, просматривает файл *lockout.user*. Этот файл включает имена пользователей, в регистрации которых вы не заинтересованы. Если в файле присутствует слово "all", доступ запрещается всем пользователям, кроме пользователя root.

Ниже приводится пример файла *lockout.user*. Этот файл может включать строки комментария.

```
$ pg lockout.users
# lockout.users
# поместите в этот файл имена пользователей по вашему усмотрению
# снятие системной блокировки
# Удалите из этого файла имена пользователей, чтобы пользователи могли
# вернуться назад.
# peter находится в долговременном отпуске и вернется в следующем месяце
peter
# lulu отсутствует две недели, вернется в конце месяца
lulu
#dave
#pauline
```

Рассмотрим, как функционирует сценарий. Сначала выполняется команда `trap` для игнорирования сигналов. Вследствие этого пользователь не может прервать выполнение сценария. Если имеется файл *lock.users*, сценарий продолжает выполняться. Первым делом проверяется наличие слова "all". Если это слово присутствует, тогда не принимаются во внимание имена всех пользователей из данного файла. Не следует применять строку комментария для устранения влияния слова "all"; этот путь не приведет к успеху. Однако можно устранить из комментария имена пользователей.

Если обнаружена запись "all", блокируются имена всех пользователей, кроме пользователя root. Чтобы удостовериться в том, что найдено точное соответствие шаблону, применяют шаблон `all\>` команды `grep`. На экран для пользователей системы выводится сообщение о том, что система в данный момент недоступна.

Основной функцией является функция `get_users`. Она реализует просмотр файла *lockout.users*, причем игнорируются все строки, начинающиеся с символа хэша. Выполняется сравнение имен и проверяется, что имя пользователя root не содержится в файле. В результате этого имя пользователя root блокируется.

Регистрационное имя пользователя, находящегося в текущий момент в системе, извлекается из переменной `LOGNAME` и сравнивается со значением переменной `NAMES`. Переменная `NAMES` сохраняет имя текущего пользователя из просматриваемого файла *lockout.users*. Если совпадение найдено, значение переменной `LOGNAME` отображается вместе с сообщением. Затем пользователь завершает работу сценария.

Этот сценарий выполнялся на нескольких машинах, которые охватывали до 40 пользователей. Во время процесса регистрации скорость выполнения сценария была довольно высока. Сценарий использовался с целью временной блокировки доступа для тех пользователей, которые отсутствовали более недели. Кроме того, доступ для отдельных пользователей блокировался ежедневно на определенное время, в течение которого обновлялись жизненно важные системы.

В файл */etc/profile* следует поместить следующую строку. Эта строка может быть размещена в конце файла, тогда пользователи получат возможность просматривать это сообщение первым среди дополнительных сведений.

```
. /apps/bin/deny.access
```

Каталог */apps/bin* является областью, где можно хранить все глобальные сценарии. Возможно применение другой области, однако при этом следует убедиться в том, что каждый пользователь может выполнить этот сценарий и воспользоваться каталогом, в котором он находится.

Если получено сообщение об ошибке "permission denied" ("разрешения нет"), значит, сценарий или каталог не имеют достаточного уровня разрешения.

В данном случае файл *lockout.users* находится в каталоге */apps/etc*. Этот каталог можно изменить, поскольку ваша структура наверняка отличается от рассматриваемой. Поскольку файл является исходным, с помощью команды `set` можно просматривать код функции (но не фактический файл *lockout.users*). Если это затруднительно, примените команду `unset` для удаления функции после ее выполнения. Поместите команду `unset` непосредственно после обращения к сценарию в файл */etc/profile*. Например:

```
unset getusers
```

Сценарий имеет следующий вид:

```
$ pg deny.access
#!/bin/sh
# deny.access

trap "" 2 3
# ОТКОРРЕКТИРУЙТЕ СЛЕДУЮЩУЮ СТРОКУ, ЕСЛИ МЕСТОПОЛОЖЕНИЕ ФАЙЛА LOCKOUT.USERS
# ИЗМЕНЕНО
LOCKOUT=/apps/etc/lockout.users
MSG="Sorry $LOGNAME, your account has been disabled, ring the administrator"
MSG2="Sorry $LOGNAME, the system is unavailable at the moment"

check_lockout()
# check_lockout
# проверка наличия файла, содержащего имена для блокировки
{
if [ -r $LOCKOUT ] ; then
  return 0
else
  return 1
fi
}

get_users()
```

```
# get_users
# чтение файла, если содержимое LOGNAME совпадает с именем в lockout.users
# отбросьте его!
{
while read NAMES
do
  case $NAMES in
  \#*);; #игнорируйте комментарии
  *}
  # если кто-либо попытается блокировать root,
  # в этом сценарии ему это сделать не удастся
    if [ "$NAMES" = "root" ]; then
      break
    fi
    if [ "$NAMES" = "$LOGNAME" ]; then
      # сообщение об отказе в регистрации
      echo $MSG
      sleep 2
      exit 1
    else
      # нет совпадения, следующая итерация
      continue
    fi
    ;;
  esac
done < $LOCKOUT
}
if check_lockout; then
  if grep 'all\>' $LOCKOUT >/dev/null 2>&1
  then
    # сначала проверьте, имеется ли слово "all". Если это слово
    # присутствует, все, кроме root, должны держаться подальше
    if [ "$LOGNAME" != "root" ]; then
      echo $MSG2
      sleep 2
      exit 2
    fi
  fi
  # обработка информации об обычных пользователях
  get_users
fi
```

27.5. Сценарий logroll

Некоторые системные журнальные файлы увеличиваются довольно быстро. Становится затруднительным вручную уточнять размеры журнальных файлов и выполнять прокрутку определенного журнала (обычно, с помощью отметки даты). Поэтому назрела необходимость создания сценария для автоматизации этой процедуры. Сценарий выполняется с помощью утилиты `cron`, и если какой-либо журнальный файл достигает определенного размера, осуществляется прокрутка данного журнала и создание нового журнального файла.

Этот сценарий может быть легко обновлен для работы с иными журнальными файлами. Например, при обработке системных журнальных файлов можно применить другой сценарий, который выполняется еженедельно и усекает журнальные файлы. Если необходимо просмотреть более ранние сообщения, нужно проверить резервную копию; при работе в 16-недельном цикле это нетрудно.

Ограничение размера устанавливается с помощью переменной BLOCK_LIMIT. Эта переменная указывает размер блока, который в данном случае равен восьми и соответствует 4 Кб. При необходимости можно установить большее значение. Информация обо всех журнальных файлах, подлежащих проверке, хранится с помощью переменной LOGS.

Затем с помощью этой переменной и цикла for выполняется проверка каждого журнального файла. Применяя команду du, можно оценить размер журнального файла. Если размер файла превышает значение BLOCK_LIMIT, журнальный файл копируется и с добавлением временной метки присоединяется к этому файлу. Затем исходный файл обнуляется, и изменяются права владения на группу файлов.

Сценарий выполняется с помощью утилиты cron два раза в неделю; при этом создается резервная копия файла с указанием временной метки. Поэтому при возникновении проблем можно быстро отследить выполненные действия.

```
$ pg logroll
#!/bin/sh
# logroll
# усечение журнальных файлов, размеры которых более MARK
# может использовать и для почтовых ящиков?

# максимальный размер журнального файла
# 4096 k
BLOCK_LIMIT=8

MYDATE=`date +%d%m`
# список журнальных файлов для проверки...ваш список может быть другим!
LOGS="/var/spool/audlog /var/spool/networks/netlog /etc/dns/named_log"
for LOG_FILE in $LOGS
do
  if [ -f $LOG_FILE ] ; then
    # определение размера блока
    F_SIZE=`du -a $LOG_FILE | cut -f1`
  else
    echo "`basename $0` cannot find $LOC_FILE" >&2
    # можно выйти здесь, но следует убедиться, что проверены все
    # журнальные файлы
    continue
  fi

  if [ "$F_SIZE" -gt "$BLOCK_LIMIT" ]; then
    # копирование журнального файла и присоединение к нему даты в формате ddmm
    cp $LOG_FILE $LOG_FILE$MYDATE
    # создание нового пустого журнального файла
    >$LOG_FILE
    chgrp admin $LOG_FILE$MYDATE
  fi
done
```

27.6. Сценарий nfsdown

Если в вашей системе имеется файловая система nfs, вы наверняка оцените преимущества приведенного ниже сценария. Иногда приходится следить за работой нескольких компьютеров и перезагружать их во время работы. Эту задачу следует выполнять как можно скорее.

Если везде смонтированы удаленные каталоги, не следует рассчитывать на то, что процесс демонтирования nfs будет выполнен в ходе перезагрузки. Желательно выполнять все вручную; кроме того, такой подход экономит время.

При выполнении сценария (на всех компьютерах) демонтируются все каталоги nfs, что позволяет довольно быстро выполнить перезагрузку.

Сценарий содержит список компьютеров, на которых находятся монтировки nfs. Этот список обрабатывается с помощью цикла for. Во время обработки с помощью команды df для каждого хоста запускается утилита grep. Смонтированные каталоги nfs имеют вид:

```
machine:remote_directory
```

Данная строка присваивается переменной NFS_MACHINE. Затем эта переменная применяется при выполнении команды unmount.

Соответствующий сценарий имеет вид:

```
$ pg nfsdown
#!/bin/sh
# nfsdown
LIST="methalpha accounts warehouse dwaggs"
for LOOP in $LIST
do
  NFS_MACHINE=`df -k | grep $LOOP | awk '{print $1}'`
  if [ "$NFS_MACHINE" != "" ]; then
    umount $LOOP
  fi
done
```

27.7. Заключение

Рассмотренные в этой главе сценарии в течение длительного времени используются автором книги. Как уже упоминалось, сценарии не должны быть объемными и сложными, поскольку они создаются с целью экономии рабочего времени пользователя.

ГЛАВА 28

Сценарии уровня выполнения

Если при загрузке системы вам нужно автоматически запустить приложение, службу или сценарий либо корректно завершить их работу при перезапуске системы, то необходимо создать сценарий уровня выполнения. Почти все варианты системы Linux, а также некоторые системы UNIX в настоящее время имеют каталоги конфигурации уровня выполнения, которые основаны на System V.

Поскольку большая часть систем включает конфигурацию этого типа, именно он и рассматривается далее. Если вы не располагаете каталогами уровня выполнения, не беспокойтесь. Приложения можно запускать в автоматическом режиме; этот метод также обсуждается в данной главе.

В главе рассматриваются следующие темы:

- уровни выполнения;
- способы создания сценариев `rc.scripts`;
- методы внедрения сценариев `rc.scripts` на различных уровнях выполнения;
- запуск приложений с помощью файла *inittab*.

Благодаря созданию сценариев уровня выполнения обеспечивается повышенная степень гибкости при управлении системой. Для запуска или останова приложения на определенном уровне выполнения нужно инсталлировать сценарий уровня выполнения (его еще называют `rc.script`).

Все сценарии, которые запускают и прекращают выполнение приложения, и в названии которых имеются ключевые слова "start" или "stop", обычно относятся к сценариям класса `rc.script`. Обратите внимание на то, что именно пользователь определяет, является ли реализуемый сценарий сценарием типа `rc.script`. В задачу этого сценария входит успешный запуск и прекращение функционирования какой-либо службы.

Методика создания каталогов конфигурации уровня выполнения позволяет автоматизировать работу сценариев `rc.scripts` только при изменении уровня выполнения. Однако нельзя определить, запущены или остановлены все необходимые службы на уровне выполнения. Эта часть работы должна выполняться shell-программистом.

Уровни выполнения можно настраивать в соответствии с действительно выполняемыми службами, но данная тема в книге не рассматривается.

28.1. Определение наличия каталогов уровня выполнения

Каталоги, где хранятся сценарии `rc.scripts` (здесь фактически речь идет о ссылках, которые мы рассмотрим далее), имеют следующий вид:

```
/etc/rcN.d
```

или

```
/etc/rc.d/rcN.d
```

где N — число. Обычно это число равно семи, поскольку каталоги `rcN.d` нумеруются от 0 до 6. Однако в системе можно иметь несколько дополнительных каталогов типа `rcS.d`. Количество каталогов не столь важно; все рассматриваемые каталоги перечислены ниже.

```
$ pwd
/etc
drwxr-xr-x 2 root sys 1024 Dec 22 1996 rc0.d
drwxr-xr-x 2 root sys 1024 Dec 22 1996 rc1.d
drwxr-xr-x 2 root sys 1024 Dec 22 1996 rc2.d
drwxr-xr-x 2 root sys 1024 Dec 22 1996 rc3.d
drwxr-xr-x 2 root sys 1024 Dec 22 1996 rc4.d
drwxr-xr-x 2 root sys 1024 Dec 22 1996 rc5.d
drwxr-xr-x 2 root sys 1024 Dec 22 1996 rc6.d
drwxr-xr-x 2 root sys 1024 Dec 22 1996 rcS.d
```

В Linux...

```
$ pwd
/etc/rc.d
$ ls
init.d  rc.local    rc0.d  rc2.d  rc4.d  rc6.d
rc      rc.sysinit  rc1.d  rc3.d  rc5.d
```

Если команда `cd` применяется в одном из каталогов `rcN.d`, можно просмотреть и другие сценарии `rc.scripts`, связанные с этими каталогами.

```
$ pwd
/etc/rc.d/rc2.d
$ ls -l
lrwxrwxrwx 1 root root 16 Dec 3 15:16 K87ypbind -> ../init.d/yd
lrwxrwxrwx 1 root root 17 Dec 3 15:10 K89portmap -> ../init.d/p
lrwxrwxrwx 1 root root 17 Dec 3 15:07 S01kerneld -> ../init.d/d
...
...
```

28.2. Уточнение текущего уровня выполнения

В этой главе не рассматриваются вопросы системного администрирования, однако shell-программист должен знать не только принципы функционирования сценариев `rc.scripts`, но также принципы их совмещения с каталогами конфигурации уровня выполнения. Для уточнения уровня выполнения примените команду:

```
$ who -r
   .       run-level 4 Apr 22 13:26  4    0    3
```

Число, расположенное после слов "run-level" является текущим уровнем выполнения. Следующие за ним данные определяют время выполнения последней перезагрузки системы.

В Linux...

```
$ runlevel
2 3
```

В первом столбце указан уровень, на котором система находилась на предварительном этапе, а во втором столбце — текущий уровень, который в данном случае равен 3.

28.3. Ускорение работы с помощью файла inittab

Каталог уровня выполнения состоит из набора сценариев, более совершенных, чем службы. Слово "services" в этом контексте означает и демон, и приложение, и серверы, и подсистемы или процессы сценария. Во время загрузки системы вызывается процесс `init` (этот процесс является родоначальником всех остальных процессов). Одной из задач упомянутого процесса является определение запускаемых служб, а также определение уровня выполнения, заданного по умолчанию. Эти сведения можно получить, просматривая текстовый файл конфигурации под названием *inittab*, размещенный в каталоге */etc*. Процесс `init` также использует этот файл для получения указаний по поводу загрузки определенных процессов. Если необходимо изменить этот файл, сначала создайте резервную копию. В случае повреждения файла или возникновения ошибок, приводящих к "деградации" системы, система не будет загружаться обычным образом; вам придется загружаться в однопользовательском режиме и устранять повреждения в файле.

Файл *inittab* включает поля, имеющие весьма лимитированный формат. Формат файла будет следующий:

```
id:rstart:action:process
```

Поле `id` имеет уникальное название, которое идентифицирует запись процесса.

Поле `rstart` содержит число, которое указывает, на каком уровне выполнения запускается процесс.

Поле `action` указывает процессу `init`, как рассматривать текущий процесс. Существует большое количество названий для поля `action`, но наиболее распространенными являются `wait` и `respawn`. Название `wait` означает, что начавшийся процесс ожидает завершения. Название `respawn` означает, что процесс начинается даже в том случае, если он еще не существует. Если же существует, то запускается заново в тот момент, когда он уже завершается.

Поле `process` содержит действительную команду для выполнения. Ниже приводится фрагмент файла *inittab*.

```
$ pg /etc/inittab
id:3:initdefault:

# Инициализация системы.
si::sysinit:/etc/rc.d/rc.sysinit
# уровень выполнения 0
10:0:wait:/etc/rc.d/rc 0
# уровень выполнения 1
11:1:wait:/etc/rc.d/rc 1
# уровень выполнения 2
12:2:wait:/etc/rc.d/rc 2
```

```
# уровень выполнения 3
13:3:wait:/etc/rc.d/rc 3
# уровень выполнения 4
14:4:wait:/etc/rc.d/rc 4
# уровень выполнения 5
15:5:wait:/etc/rc.d/rc 5
# уровень выполнения 6
16:6:wait:/etc/rc.d/rc 6
# Выполнение gettys на стандартных уровнях выполнения
1:12345:respawn:/sbin/mingetty tty1
2:2345:respawn:/sbin/mingetty tty2
3:2345:respawn:/sbin/mingetty tty3
4:2345:respawn:/sbin/mingetty tty4
5:2345:respawn:/sbin/mingetty tty5
6:2345:respawn:/sbin/mingetty ttyS1 vt100
```

Первая строка файла описывает уровень выполнения системы, заданный по умолчанию; ниже приводится уровень выполнения 3, который не является чем-либо необычным.

Строки, которые начинаются числами 10—16, определяют запуск или прекращают выполнение сценариев уровней выполнения для определенных уровней выполнения. Например, рассмотрим следующую строку:

```
15:5:wait:/etc/rc.d/rc 5
```

В строке содержится следующая информация: если пользователь находится на уровне выполнения 5, сценарий `/etc/rc.d/rc` запускается с параметром 5. Это означает, что сценарий `/etc/rc.d/rc` выполняет все сценарии в каталоге */etc/rc.d/rc/rc5.d*.

Последняя строка файла — уровни выполнения 2, 3, 4 и 5 — свидетельствует о том, что процесс заново возрождается. То есть, процесс никогда не уничтожится (ну, по крайней мере, в течение одной секунды). Непрерывно отвергается процесс `mingetty` для последовательного порта `ttyS1`. В данном случае в роли параметра используется ID терминала, который имеет значение vt100.

28.4. Переходим к уровням выполнения

Одной из последних задач процесса `init`, которая реализуется перед тем, как система "полностью запустится", является выполнение всех сценариев для уровня выполнения, заданного по умолчанию. Файл, осуществляющий эту задачу, называется либо */etc/rc.d/rc*, либо */etc/rc.init*. Роль этого сценария заключается в первоначальном уничтожении процессов для этого уровня, а затем — в установке процессов данного уровня.

Как процесс определяет, какие службы запускаются или прекращают выполнение? Файл *rc* или *rc.init* выполняет функции цикла `for` при обработке каждого сценария `rc.script`. Каждый сценарий `rc.script` запускается в каталоге *rc3.d* с помощью опции `K`, и ему передается параметр "stop". Затем аналогичный процесс поддерживается для всех сценариев `rc.scripts`, которые запускаются с помощью опции `S`, и им передаются параметры "start". Конечно, подобный процесс поддерживается при обращении к измененному уровню выполнения. Но, в отличие от каталога *rc3.d*, в данном случае обрабатывается каталог *rcN.d*, благодаря чему изменяется уровень выполнения `N`.

Сценарии, находящиеся в каталоге *rcN.d*, представляют собой только ссылки — фактические сценарии вызываются в другом месте. Эти сценарии располагаются в каталоге под названием */usr/sbin/init.d* или */etc/init.d*.

В Linux...

```
/etc/rc.d/init.d
```

В этом каталоге хранятся несколько сценариев, которые могут запускать или прекращать функционирование служб. Имена этих сценариев формируются по системе *rc.<что он делает>*, где *rc* означает *run command* или *run control*. Некоторые системные администраторы называют эти сценарии "реально критическими" (really crucial). Ниже приводится листинг подобного файла.

```
$ ls
rc.localfs rc.halt rc.reboot rc.syslogd rc.daemon
...
```

Общий формат вызова сценариев *rc.scripts* будет следующим:

```
rc.name stop - останов службы
rc.name start - запуск службы
```

Необязательными вызовами являются вызовы перезапуска и состояния. Любой другой вызов появляется в ответ на сообщение о применении, в котором содержится методика вызова сценария *rc.script*. Следует отметить, что эти сценарии можно вызывать вручную.

Итак, вы уже изучили, какие функции выполняет сценарий при вызове. Следующий этап — помещение сценариев в соответствующие каталоги *rcN.d*. Но сначала рассмотрим систему уровней выполнения.

28.4.1. Различные уровни выполнения

Существует семь уровней выполнения (табл. 28.1). Различные системы имеют на некоторых уровнях небольшие отличия.

Прежде чем размещать сценарий на различных уровнях выполнения, уточните, на каких уровнях эта служба должна запускаться или уничтожаться (если сценарий запускает и прекращает выполнение службы). Если решение принято, можно приступать к делу.

Таблица 28.1. Фуикции различных уровней выполнения

Уровень выполнения 0	Прекращает и останавливает целую систему
Уровень выполнения 1	Отдельный пользователь или режим администрирования
Уровень выполнения 2	Многопользовательский режим; запускаются некоторые сетевые службы. Ряд систем использует этот уровень как уровень выполнения в обычном режиме функционирования вместо уровня выполнения 3
Уровень выполнения 3	Обычный режим функционирования, применяется для всех сетевых служб
Уровень выполнения 4	Уровень определенного пользователя; применяйте этот уровень для настройки при выполнении

Уровень выполнения 5	Этот уровень имеет некоторые вариации в виде заданного по умолчанию режима X-windows; в других случаях этот уровень применяется для перевода системы в режим поддержки
Уровень выполнения 6	Перезагрузка

28.4.2. Формат сценария уровня выполнения

Сценарии в каталогах *rcN.d* представляют собой все символические ссылки, которые сохраняют дублирование сценариев на нулевом уровне. Формат этих ссылок:

```
Snn.имя_сценария
```

или

```
Knn.имя_сценари
```

где

`S`	Означает запуск процесса
`K`	Означает уничтожение процесса
`nn`	Является двузначным числом от 00 до 99, хотя некоторые системы характеризуются трехзначными числами от 000 до 999. При установлении ссылок на различные каталоги сохраняйте то же самое число. Например, если служба запускается в каталоге *rc3.d* и сценарий называется `S45.myscript`, то при запуске этой службы в каталоге *rc2.d* нужно убедиться, что сценарий также называется `S45.myscript`.

`имя_сценария`	Является названием сценария, зависящим от типа системы. Может находиться в одном из файлов:
	`/usr/sbin/init.d`
	`/etc/rc.d`
	`/etc/init.d`

Когда процесс `init` требует вызова сценариев `rc.scripts`, выполняется процесс уничтожения, начиная от самого большего и завершая самым меньшим числом K, т.е. `K23.myscript K12.named`. Запуск выполняется в диапазоне от самого меньшего до самого большего значения. Если вы работаете в системе Linux, числа K вызываются от самого большего до самого меньшего числа.

28.4.3. Инсталляция сценария уровня выполнения

Чтобы инсталлировать собственный сценарий *rc.script*, следует выполнить следующее:

- написать сценарий, который действительно удовлетворяет стандартам вызова;
- удостовериться, что сценарий действительно запускает или останавливает необходимую службу;
- разместить сценарий (в зависимости от системы) в каталоге */etc/init.d*, */usr/sbin/init.d* или в каталоге */etc/rc.d*;

- создать ссылки во всех подходящих каталогах *rcN.d*, используя соответствующее соглашение о наименовании.

Ниже приводится сценарий, который запускает и прекращает выполнение приложения под названием `rc.audit`. Эта служба запускается на уровнях выполнения 3, 5 и 4 и уничтожается на уровнях выполнения 6, 2 и 1. При просмотре некоторых записей в каталогах *rcN.d* число 35 является зарезервированным, поэтому оно применяется в данном случае. Действительно, нет причин прекращать функционирование сценария, поэтому применяется число, которое уже использовалось.

Рассмотрим этот сценарий. Как можно заметить, простая конструкция `case` выполняет перехват параметров `stop` и `start`.

```
$ pg rc.audit
#!/bin/sh
# rc.audit start | stop
# сценарий запускает или прекращает выполнение контролирующего приложения
zeega
#
case "$1" in
start)
  echo -n "Starting the audit system...."
  /apps/audit/audlcp -a -p 12
  echo
  touch /var/lock/subsys/rc.audit
  ;;
stop)
  echo -n "Stopping the audit system...."
  /apps/audit/auddown -k0
  echo
  rm -f /var/lock/subsys/rc.audit
  ;;
 restart)
  $0 stop
  $0 start
  ;;
*)
  echo "To call properly..Usage: $0 {start|stop|restart}"
  exit 1
  ;;
esac
 exit 0
```

В Linux...

В некоторых вариантах Linux предполагается, что файл блокировки создается при запуске службы. Если файл блокировки отсутствует, при уничтожении сценариев могут возникнуть трудности.

Опция start вызывает контрольный процесс, который запускает действительную систему контроля, а опция stop вызывает сценарий, останавливающий систему контроля. Конечно, перед помещением сценария в каталог *init.d* его следует проверить.

```
$ rc.audit
To call properly..Usage:./rc.audit {start|stop|restart}
```

```
$ rc.audit start
Starting the audit system....
```

Предположим, что сценарий проверен. Запуск и прекращение функционирования службы контроля реализуется без затруднений. Установим связь сценария с нужными каталогами выполнения.

В данной системе каталоги *rcN.d* помещены в файл */etc/rc.d*, а сценарии `rc.scripts` помещены в файл */etc/rc.d/init.d*. Измените пути, если это необходимо.

Внимательно приступайте к первоначальному запуску сценария; не забывайте, что запуск сценариев начинается с указания опции `S`.

```
$ pwd
/etc/rc.d/rc3.d
$ ln -s../init.d/rc.audit S35rc.audit

$ ls -l
...
lrwxrwxrwx   1 root   root 27 May 8 14:37 S35rc.audit -> ../init.d/
rc.audit
...
```

Теперь создается ссылка. Поток вывода команды `ls -l`, который показывает ссылку, направляется в файл */etc/init.d/rc.audit*. Как часть команды ссылки, поддерживается путь ко всему каталогу, но это не обязательно. Теперь необходимо применить команду `cd` для последовательного перехода во все каталоги, где нужно запустить службу (в данном случае *rc4.d* и *rc5.d*), и выполнить в них аналогичные действия. Для уничтожения сценариев примените следующие команды:

```
$ pwd
/etc/rc.d/rc6.d
$ ln -s../init.d/rc.audit K35rc.audit
$ ls -l
...
lrwxrwxrwx   1 root   root  27 May  8 14:43 K35rc.audit -> ../init.d/
rc.audit
...
```

Аналогичную процедуру можно реализовать для других каталогов, где нужно остановить выполнение службы контроля. Теперь при перезагрузке системы служба контроля прекращает выполняться. Это происходит и в том случае, когда значения уровней выполнения изменяются на 2 или 1. Служба контроля запускается, если значение уровня выполнения изменяется на 4 или 5.

28.5. Использование файла inittab для запуска приложений

Существуют другие возможности для запуска приложений; например, можно запустить приложение путем размещения записи в файле *inittab*. Это является удобным вовсе не потому, что такие системы не имеют каталогов уровней выполнения. Использование записей в файле *inittab* связано с тем, что существует несколько сценариев системной проверки, которые необходимо выполнять, когда система завершает загрузку. Файл *inittab* является идеальным местом для помещения этих сценариев.

В приведенном примере применяется один из сценариев проверки зеркального образа диска, который выполняется, когда номер уровня выполнения равен 3. Вначале следует удостовериться, что сценарий выполняется надлежащим образом, затем осуществляется резервное копирование файла *inittab*.

```
$ cp /etc/inittab /etc/inittab.bak
```

Отредактируем файл *inittab*. В конец файла добавим следующую запись.

```
# сценарий проверки диска, рассмотрим, не повреждены ли какие-либо
# зеркальные образы.
rc.diskchecker:3:once:/usr/local/etc/rc.diskchecker > /dev/console 2>&1
```

Теперь следует сохранить файл и выйти из редактора.

Вышеприведенная запись означает следующее: *Rc.diskchecker* является уникальным ID на уровне выполнения 3. Выполните этот процесс один раз. Сценарий находится в файле */usr/local/etc/rc.diskchecker*, весь поток вывода направляется на консоль.

28.6. Другие методы, применяемые для запуска и останова служб

Если вы не желаете применять файл */etc/inittab*, существует другая возможность запустить службу. Большая часть систем включает файл *rc.local*, который помещается в каталоге */etc* либо рядом с ним. Этот файл сценария закрывается после запуска файла *inittab* и сценария `rc.scripts`. В файл *rc.local* можно ввести все необходимые команды или добавить запись для вызова удобного вам сценария запуска.

Некоторые системы также поддерживают сценарный файл под названием `shutdown`, который находится в каталоге */bin* (хотя довольно часто этот файл может располагаться в каталоге */usr/sbin*). Воспользуйтесь этим файлом для завершения выполнения служб, используя команду завершения работы системы.

28.7. Заключение

Уровни выполнения на самом деле относятся к области системного администрирования. Задача данной главы состоит в том, чтобы продемонстрировать наилучшие способы контроля и гибкого использования различных сценариев и служб, которые запускаются при загрузке системы.

Также нужно учесть, что при перезагрузке системы не следует беспокоиться о запуске или прекращении выполнения службы в ручном режиме.

ГЛАВА 29

Сценарии cgi

В настоящее время, когда практически на каждом ПК установлен Web-сервер, глава, посвященная сценариям cgi, органически вписывается в книгу по shell-программированию.

В главе будут рассмотрены следующие темы:

- базовые сценарии cgi;
- использование SSI;
- метод `get`;
- метод `post`;
- создание интерактивного сценария;
- сценарий cgi, автоматически обновляющий Web-страницу.

Для установки Web-сервера вовсе необязательно организовывать сеть; этот сервер может быть запущен на локальном компьютере. Изначально предполагается, что у вас установлен Web-сервер (Apache, Cern и т.п.) и броузер, используемый для просмотра Web-страниц (Netscape, Internet Explorer и т.п.). Помимо этого, сервер должен поддерживать cgi. По умолчанию поддержка cgi отключена путем добавления знаков комментариев в определенные строки сценария. Дополнительные сведения по этому вопросу можно найти в последующих разделах этой главы.

Вопросы установки и настройки Web-сервера выходят за рамки данной книги, хотя установка и запуск Web-сервера занимают не более 20 минут. Примеры, приведенные в этой главе, были выполнены с помощью Web-сервера Apache. В качестве Web-броузера использовался броузер Internet Explorer.

В главе не рассматриваются подробно особенности HTML или Web, поскольку в настоящее время существует множество книг, посвященных этим вопросам. Обсуждение HTML потребовало бы написания нескольких дополнительных глав.

29.1. Определение Web-страницы

Web-страница (или Web-документ) состоит из тегов HTML. Когда броузер загружает Web-страницу, теги определяют способ отображения этой страницы на экране. Web-страница может состоять из многих элементов, включая ссылки (которые позволяют связывать между собой различные страницы), цвета, выделения, шрифты разных размеров, линии и таблицы. В состав Web-страницы могут также включаться картинки и звуковые фрагменты.

Существует два типа Web-страниц: динамические и статические. Статические страницы могут применяться только для отображения информации либо, возможно, для выполнения загрузки файлов. Динамические страницы являются интерактивными: они

могут создавать отчеты на основе предоставляемых пользователем сведений. Помимо этого, динамические страницы применяются для отображения изменяющейся информации в режиме реального времени, например цен на акции, либо осуществляют задачи мониторинга. Для отображения подобных динамических процессов также необходимы сценарии.

Для поддержки информационного обмена между сервером и сценариями требуется протокол Common Gateway Interface (Интерфейс общего шлюза), который обычно именуется cgi.

29.2. Протокол cgi

Аббревиатура `cgi` обозначает спецификацию, которая задает для получающих информацию сценариев способ обмена данными с сервером. Подобные сценарии (или сценарии cgi) могут быть созданы с помощью любого языка написания сценариев. Наиболее популярным является язык Perl, хотя с этой целью могут применяться и обычные shell-сценарии, как вы убедитесь далее.

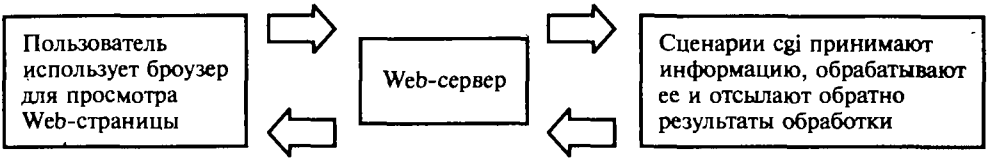


Рис. 29.1. Броузер и сервер, использующие интерфейс cgi для обмена информацией

29.3. Подключение к Web-серверу

Для подключения к Web-серверу используется URL (Uniform Resource Locator — унифицированный указатель ресурсов). Указатель URL содержит два типа информации:

```
протокол
адрес и данные
```

Протоколом может быть протокол http, ftp, mailto, file, telnet и news. В этой главе будет рассматриваться только http, протокол передачи гипертекста (hypertext transfer protocol).

В качестве адреса обычно выступает имя DNS или хост-имя сервера, хотя может применяться IP-адрес. Можно использовать и другую информацию, например фактическое имя пути к файлу, к которому осуществляется доступ.

Все подключения реализуются с помощью протокола TCP. При этом по умолчанию используется порт 80.

Если на локальном компьютере установлен Web-сервер, а основная HTML-страница называется *index.html*, можно воспользоваться следующим URL:

```
http://localhost/index.html
```

Вообще говоря, файл *index.html* — это файл, загружаемый по умолчанию. (Имя файла, загружаемого по умолчанию, можно изменить с помощью файлов конфигурации сервера.) Следовательно, в этом случае можно ввести такой URL:

```
http://localhost/
```

29.4. Сценарии cgi и HTM

Когда броузер генерирует запрос на загрузку страницы, Web-сервер задается в виде входящего URL. Если в качестве части URL-пути указывается `cgi-bin`, сервер открывает соединение, которое обычно реализует перенаправление к запрашиваемому cgi-сценарию. Входной и выходной поток сценария cgi отсылаются с помощью этого перенаправления. Если сценарий cgi используется для отображения форматированной Web-страницы, он должен включать теги HTML. Благодаря этому отображаемая страница может распознаваться Web-сервером, хотя при этом от пользователя потребуются некоторые познания в области HTML. Этот документ может отсылаться Web-сервером броузеру с целью отображения для пользователя. В табл. 29.1 представлены некоторые полезные теги HTML.

Таблица 29.1. Основные теги HTML, применяемые для создания страниц

`<HTML></HTML>`	Теги открытия и закрытия документа
`<HEAD></HEAD>`	Открытие и закрытие информационной области
`<TITLE></TITLE>`	Открытие и закрытие заголовка
`<BODY></BODY>`	Открытие и закрытие отображаемой страницы
`<Hn></Hn>`	Заголовочный шрифт, увеличение размера шрифта
`<P></P>`	Начало и конец абзаца
` `	Разбиение строки
`<HR>`	Горизонтальная линия
`<PRE></PRE>`	Открытие и закрытие предварительно отформатированного текста, всех символов табуляции, всех сохраненных строк
`<B></B>`	Полужирный стиль символов
`<I></I>`	Курсив
`<OL></OL>`	Сортированные списки
`<A HREF=url>link</A>`	Гипертекстовая или горячая ссылка на страницу или URL
`<FORM></FORM>`	Определение формы
`METHOD`	Метод post или get
`ACTION`	Адрес
`<INPUT...>`	Запись данных
`NAME`	Имя переменной
`SIZE`	Ширина текстового поля, заданная в символах
`TYPE`	Флажок, переключатель, кнопка восстановления или фиксации
`<SELECT...>`	Разворачивающееся меню
`NAME`	Имя переменной
`SIZE`	Количество отображаемых элементов списка
`<OPTION VALUE>`	Возврат выбранной опции переменной NAME
`</SELECT>`	Закрытие выбранного списка

29.4.1. Базовый сценарий cgi

Все сценарии обычно находятся в каталоге *cgi-bin* Web-сервера, хотя подобное размещение может быть изменено. Для изменения размещения сценариев и подключения сервера cgi следует обратиться к файлам конфигурации `srm.conf` и разделу `ScriptAlias`. Все сценарии должны иметь расширение *.cgi*. Все документы обычно размещаются в каталоге *html* либо *htdocs* и имеют расширение *.html*. Для всех сценариев требуется установить следующие права доступа:

```
chmod 755 script.cgi
```

По умолчанию любые подключения к Web-странице обычно осуществляются от имени пользователя `nobody`, хотя это можно изменить с помощью файла конфигурации *httpd.conf*. Несмотря на то, что в этой главе не рассматриваются вопросы настройки Web, некоторые моменты все же стоит отметить. В частности, неплохо было бы проверить, отключено ли поле пароля "nobody". Если это так, запрещается подключение для произвольных пользователей, в то время как пользователь `nobody` физически подключен к терминалу. Для отключения пароля пользователя `nobody` в соответствующее поле пароля просто вставьте звездочку (файл пароля */etc/passwd*).

Если какой-либо из сценариев не функционирует, первым делом нужно просмотреть журнальные файлы ошибок. В этих файлах содержатся четкие описания всех возникших ошибок. Если применяется сервер `apache`, журнальные файлы обычно находятся в каталоге */etc/httpd/logs* либо */usr/local/apache/logs*, в зависимости от того, в каком месте системы устанавливается Web-сервер. Сценарии могут быть также протестированы путем выполнения их запуска из командной строки. Конечно, в этом случае вы получите только текстовый вывод, но он окажет вам помощь при дальнейшей отладке.

А теперь приступим к созданию сценария cgi. Введите указанный ниже текст в файл, назовите его *test.cgi* и сохраните в каталоге *cgi-bin*. Не забудьте установить для сценария права доступа `755`.

```
$ pg firstpage.cgi
#!/bin/sh
# firstpage.cgi
# отображение текстовой страницы
echo "Content-type: text/html"
echo ""
echo "<HTML>"
echo "<H1><CENTER> THIS IS MY FIRST CGI PAGE</CENTER></H1>"
echo "<HR>"
echo "<H2><CENTER>STAND-BY TO STAND-TO! </CENTER></H2>"
echo "</HTML>"
```

В первой строке (как вы уже, наверное, знаете) указывается местоположение интерпретатора shell. Первая строка, содержащая команду `echo`, сообщает серверу о том, что это заголовок MIME; вторая команда `echo` сообщает о новой строке. Вывод сценариев cgi не будет осуществляться, если не указана новая строка после заголовка MIME.

На этом этапе отображается начальный тег `<HTML>`, информирующий броузер о том, что весь документ представлен в формате HTML. При этом могут отображаться различные символьные шрифты, размеры которых варьируются от наибольшего, `<H1>`, до наименьшего — `<Hn>`. Обычно шрифт наименьшего размера, который хорошо различим, задается тегом `<H6>`. Для придания красивого внешнего вида

выполняется центрирование текста на странице. Затем отображается горизонтальная линия. В дальнейшем снова используется тег <H2> для определения размера шрифта, и тег <CENTER> — для центрирования текста "Stand-By To Stand-To". Последняя строка завершается тегом <HTML>.

Если вы забудете указать какие-либо закрывающие теги, не беспокойтесь — вы их вскоре обнаружите, поскольку при попытке загрузки в окно броузера подобного документа открывающие теги, для которых отсутствуют закрывающие, будут отображены на экране.

Теперь для отображения документа введите URL:

```
http://ваш_сервер/cgi-bin/firstpage.cgi
```

Вместо параметра ваш_сервер подставляется фактическое имя сервера.

Если вы работаете в сети, и при этом на экране отображается сообщение "DNS lookup failure" ("Сбой при поиске DNS"), это означает, что броузер, возможно, пытается подключиться к Internet для осуществления поиска заданной страницы. Измените параметры настройки броузера, позволяющие обойти proxy-сервер при обращении к локальным компьютерам и перезагрузите броузер.

На рис. 29.2 демонстрируется внешний вид Web-страницы.

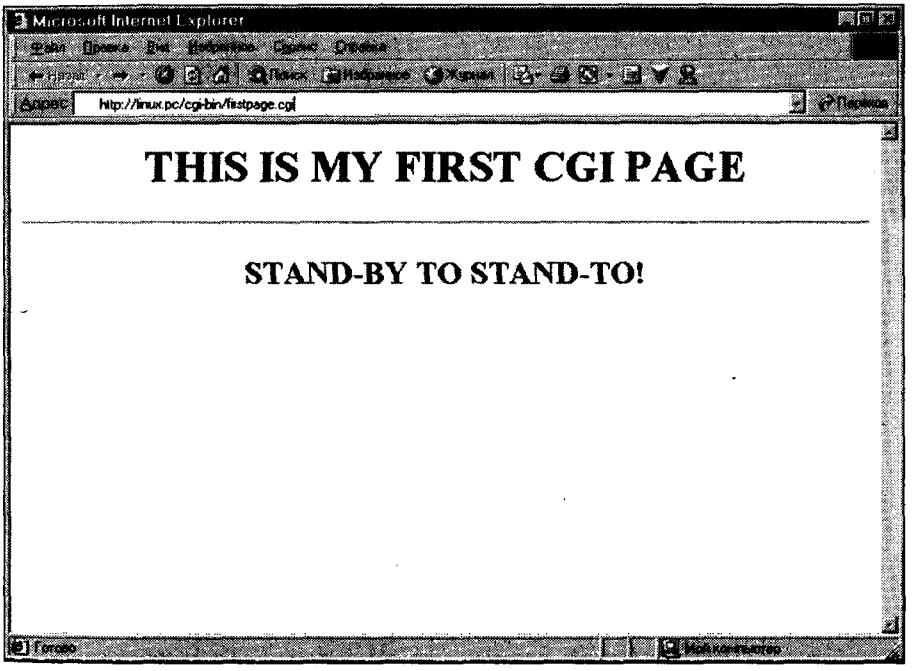


Рис. 29.2. Результат выполнения сценария firstpage.cgi

29.4.2. Отображение вывода команды интерпретатора shell

Теперь поместим команду интерпретатора shell в сценарий, при этом вывод команды будет отображен в документе HTML.

Вы можете увидеть, сколько пользователей зарегистрировано на данный момент времени. Выполните команду who и отфильтруйте ее вывод с помощью команды wc.

В результате отобразится количество зарегистрированных пользователей. Также выводится значение даты. Соответствующий сценарий имеет вид:

```
$ pg pagetwo.cgi
#!/bin/sh
# pagetwo.cgi
# отображение страницы с помощью вывода команды Unix
MYDATE=`date +%A" "%d" "%B" "%Y`
USERS=`who |wc -l`
echo "Content-type: text/html"
echo ""
echo "<HTML>"
echo "<H1><CENTER> THIS IS MY SECOND CGI PAGE</CENTER></H1>"
echo "<HR>"
echo "<H2><CENTER>$MYDATE</CENTER></H2>"
echo " Total amount of users on to-day is :$USERS"
echo "<PRE>"
if [ "$USERS" -lt 10 ]; then
  echo " It must be early or it is dinner time"
  echo " because there ain't many users logged on"
fi
echo "</PRE>"
echo "</HTML>"
```

В начале сценария считывается информация о дате и текущих пользователях. Дата отображается в центральной части страницы. Также отображается значение переменной USERS. Конструкция `if` используется для определения, является ли число зарегистрированных пользователей меньшим десяти; если это условие выполняется, отображается сообщение "It must be early or it's dinner time".

Тег <PRE> применяется для сохранения опций форматирования, состоящих из служебных символов и символов табуляции. Обычно тег <PRE> используется для отображения вывода системных команд, таких как `df`, либо списка файлов, либо нескольких конструкций `echo`.

В данном случае вовсе не обязательно применять тег <PRE>, однако автор специально упомянул этот тег на ранней стадии для того, чтобы пользователь уже сейчас сознательно использовал его для разработки собственных Web-страниц. Для отображения документа введите следующий адрес:

```
http://ваш_сервер/cgi-bin/pagetwo.cgi
```

Здесь вместо параметра `ваш_сервер` подставляется фактическое имя сервера. На рис. 29.3 показан внешний вид этой Web-страницы.

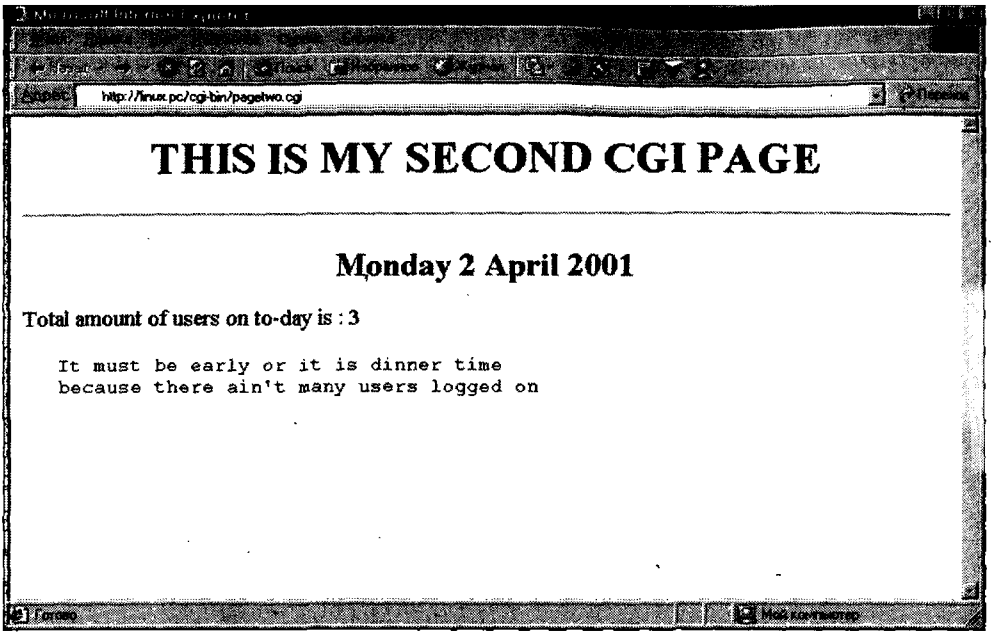


Рис. 29.3. Результат выполнения сценария pagetwo.cgi

29.4.3. Использование SSI

Использование сценариев cgi, открывающих Web-страницы с целью отображения небольшого количества сведений, не всегда является оправданным. Например, была отображена дата, но также был создан сценарий cgi, который тоже отображает дату. Не лучше ли было бы внедрить сценарий cgi в документ HTML таким образом, чтобы вывод сценария отображался на обычной странице? Это вполне возможно, и именно такую методику мы рассмотрим в дальнейшем.

Для внедрения сценариев cgi в документы можно воспользоваться технологией SSI (Server Side Includes — Включения со стороны сервера). При отображении документа происходит замена команды SSI результатом выполнения данной команды или сценария. При этом также экспортируются дополнительные переменные среды, содержащие сведения об установленном сервере и командах.

Для активизации возможностей SSI, обеспечивающих просмотр сервером команд SSI внутри документов, следует убрать комментарии в соответствующих строках файлов конфигурации. В случае с сервером Apache используются следующие строки:

```
Addhandler server-passed.shtml
Addtype text/htm shtml
```

Для перезапуска сервера введите команду `kill -1`, в результате чего сервер повторно считает конфигурационные файлы. Документы, для которых применяется SSI, используют расширение файла *shtml* вместо расширения *html*.

29.4.4. Счетчик количества посещений

Создадим документ, в котором отображается счетчик количества посещений. Счетчик будет выдавать сообщение типа "you are the nth visitor to this site" ("вы являетесь n-м посетителем этого сайта"). Можно также отображать дату последнего изменения страницы.

Не забудьте поместить сценарий в каталог *cgi-bin*; вызовите его путем ввода `hitcount.cgi`.

```
$ pg hitcount.cgi
#!/bin/sh
# hitcount.cgi
# счетчик попыток доступа к страницам для html <cgi>
# файл счетчика должен иметь атрибуты chmod 666
counter=../cgi-bin/counter
echo "Content-Type: text/html"
echo ""
read access < $counter
access=`expr $access + 1`
echo $access
echo $access >$counter
```

Как видно из приведенного кода, сценарий считывает файл *../cgi-bin/counter*, присваивает его переменной `access`, добавляет к нему единицу, затем записывает результат обратно в файл */cgi-bin/counter*.

Теперь создадим файл *counter*. Все, что требуется в данном случае, — поместить в этот файл начальный номер; в качестве начального номера будет использована единица. Итак, создайте файл *counter*, введите в него 1, затем сохраните файл и выйдите из него.

Поскольку этот файл будет использоваться любым пользователем, необходимо присвоить ему права владельца, группы и других пользователей.

```
$ chmod 666 counter
```

Теперь осталось создать файл с расширением *.shml* и поместить его в корневой каталог Web, где обычно находятся другие документы HTML. Файл также может находиться в каталоге *htdocs* или *html*. Ниже приводится образец этого файла; не забывайте присваивать ему расширение *.shtml*:

```
$ pg main.shtml
<!- main.shtml>
<!- строка комментария>
<HTML>
<H4> Last modified: <!--#echo var="LAST_MODIFIED" -->
</H4>
<HR>
<H1><CENTER> THE MAY DAY OPERATIONS CENTER </H1>
<H2>Stand-by to Stand-to
<HR>
This page has been visited <!--#exec cgi="/cgi-bin/hitcount.cgi"--> times
</CENTER>
```

```
</H2>
<HR>
</HTML>
```

Последняя изменяемая переменная, также как и другие переменные, экспортируются с помощью SSI. Обратитесь к Web-узлу apache (*www.apache.org*) для получения полного описания всех дополнительных переменных, которые были экспортированы с помощью SSI.

Посмотрите на команду SSI:

```
This page has been visited <!--#exec cgi = "/cgi-bin/hitcount.cgi"-- > times
```

Общий формат команды:

```
< !--# команда аргумент = "значение"-- >
```

В нашем случае для запуска cgi-сценария `hitcount` применяются следующие значения параметров:

- команда — exec,
- аргумент — cgi,
- "значение" — имя вызываемого сценария.

В рассматриваемом случае файл конфигурации был изменен таким образом, что данная страница будет отображаться по умолчанию вместо страницы `index.html`. Но остается также возможность вызова файла с помощью указания полного пути.

Если требуется изменить страницу, заданную по умолчанию, отредактируйте файл *srm.conf*. При этом обеспечивается доступ к следующей записи:

```
DirectoryIndex
```

В данной строке находится имя файла *index.html*. Измените это имя для новой страницы, заданной по умолчанию. Не забудьте закрыть и перезапустить Web-сервер, чтобы изменения возымели эффект.

Для вызова сценария введите URL:

```
http://<имя_сервера>/main.shtml
```

или

```
http://<имя_сервера>
```

если это страница, заданная по умолчанию.

На рис. 29.4 показан пример страницы, содержащей счетчик посещений; для просмотра приращения счетчика достаточно просто обновить страницу. Обратите внимание, каким образом отображается значение переменной `LAST_MODIFIED`.

Конечно, можно каждый день сбрасывать значение счетчика. Для этого нужно воспользоваться записью одиночной команды `cron`, которая отправляет в файл число 1.

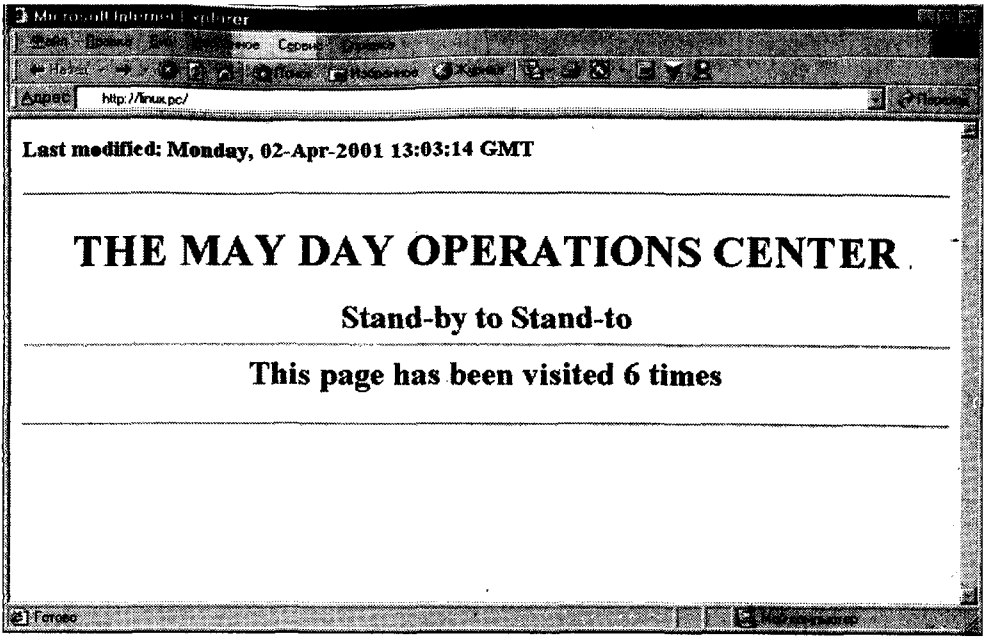


Рис. 29.4. Страница HTML с простым счетчиком посещений

29.4.5. Вывод на печать текущих настроек Web-среды с помощью ссылки

При выполнении сценария cgi какое-то количество переменных среды становится незадействованным. Для просмотра значений большинства переменных используется команда `env` или `set`. Давайте создадим ссылку на основе файла *main.shtml* для вызова сценария, отображающего значения этих переменных. Ниже приведен тег HTML, задающий такую ссылку:

```
<A HREF = "/cgi-bin/printenv.cgi">Environment</A>
```

Набор символов `A HREF` обозначает начало тега ссылки. За этим набором символов следует адрес (или назначение), заключенный в двойные кавычки. Слово Environment отображается на экране; этим определяется область, в результате щелчка на которой выполняется сценарий *printenv.cgi*. Тег `</A>` обозначает конец описания ссылки.

Пример файла *main.shtml*:

```
$ pg main.shtml

<HTML>
<! строка комментария>
<! main.shtml>
<H4> Last modified: <!--#echo var="LAST_MODIFIED" -->
</H4>
<HR>
<CENTER>
<H1> THE MAY DAY OPERATIONS CENTER </H1>
```

```
:H2> Stand-by to Stand-to
:HR>
'his page has been visited <!--#exec cgi="/cgi-bin/hitcount.cgi"--> times
:HR>
'o see your environment settings just click
:A HREF="/cgi-bin/printenv.cgi" >here</A>
:/CENTER>
:/H2>
:HR>
'/HTML>
```

Ниже приведен сценарий *printenv.cgi*, выводящий на печать значения параметров среды. В сценарии используется команда `env`. Тег `<PRE>` применяется для сохранения форматирования (вывода табуляции и пробелов).

```
; pg printenv.cgi
:!/bin/sh
: printenv.cgi
: вывод на печать настроек Web-сервера с помощью команды env
:cho "Content-type: text/html"
:cho ""
:cho "<HTML><PRE>"
:nv
:cho "</PRE></HTML>"
```

На рис. 29.5 показано, как выглядит страница с добавленной ссылкой.

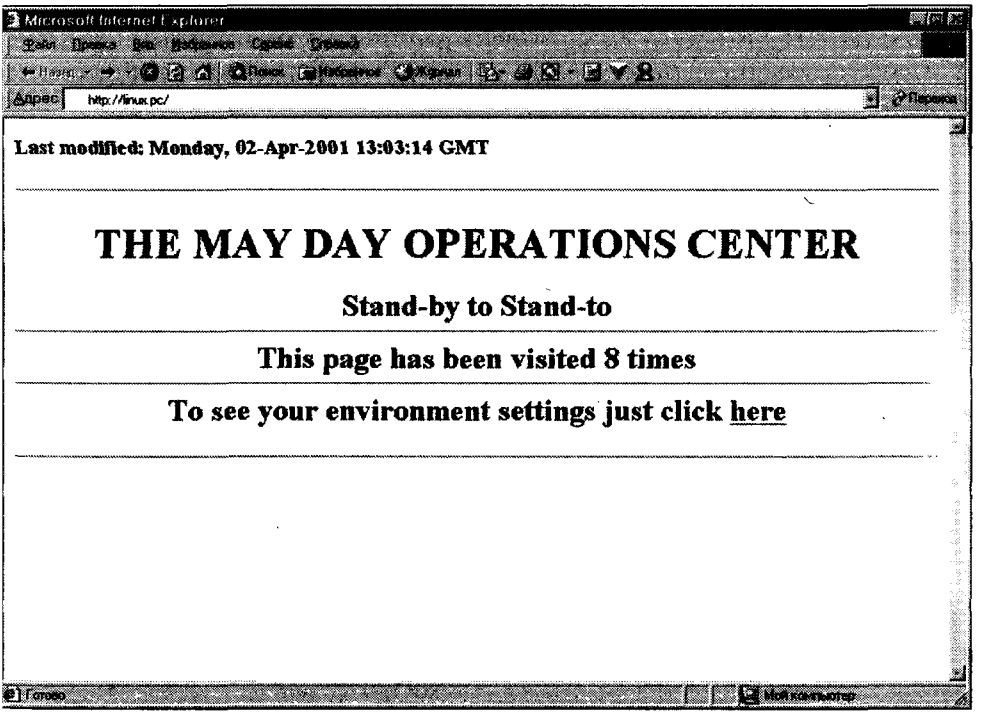


Рис. 29.5. Страница, включающая ссылку для просмотра переменных среды

После щелчка на ссылке отображаются настройки среды (рис. 29.6). Эти настройки могут слегка отличаться в каждом конкретном случае. При выполнении различных сценариев возможно изменение настроек с целью адаптации к новой среде.

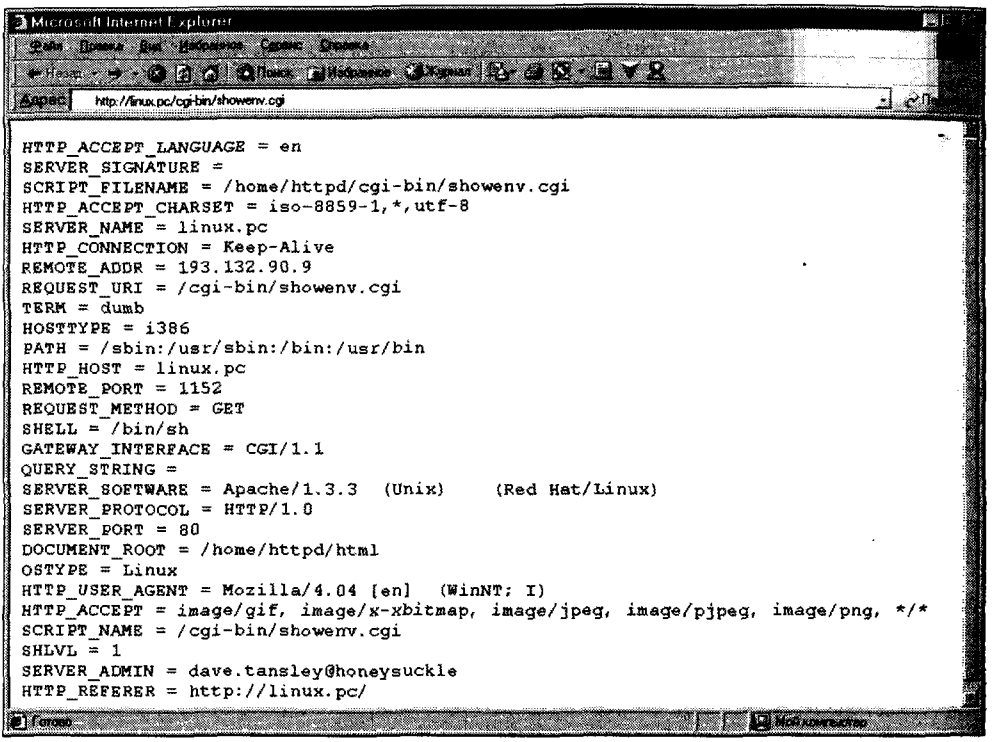


Рис. 29.6. Страница, отображающая значения текущих переменных

29.4.6. Другие общие переменные среды

В табл. 29.2 представлены наиболее часто применяемые переменные среды cgi. Значения некоторых из этих переменных могут быть просмотрены с помощью команды `env` либо `set`.

Таблица 29.2. Общие переменные cgi Web-сервера

DOCUMENT_ROOT	Основной каталог Web-сервера, куда загружаются документы
GATEWAY_INTERFACE	Редакция cgi
HTTP_ACCEPT	Другие подтвержденные типы MIME
HTTP_CONNECTION	Предпочитаемое подключение HTTP
HTTP_HOST	Имя локального хост-компьютера
HTTP_USER_AGENT	Клиентский броузер
REMOTE_HOST	Удаленный хост-компьютер
REMOTE_ADDR*	IP-адрес удаленного хост-компьютера
REQUEST_METHOD	Метод, используемый для передачи информации

SCRIPT_FILENAME	Абсолютное имя пути сценария cgi
SCRIPT_NAME	Относительное имя пути сценария cgi
SERVER_ADMIN	Адрес электронной почты Web-администратора
SERVER_NAME	Хост-имя, DNS либо IP-адрес сервера
SERVER_PROTOCOL	Протокол, используемый для реализации соединения
SERVER_SOFTWARE	Наименование программного обеспечения Web-сервера
QUERY_STRING	Передаваемые данные из метода GET
CONTENT_TYPE	Тип MIME
CONTENT_LENGTH	Количество байтов, передаваемых с помощью метода post

* Эта переменная определяет адрес шлюза, используемого для подключения к Internet

Для отображения значения переменных можно заключить эти переменные в небольшой сценарий cgi, а затем вызывать данный сценарий в случае, если нужно проверить значение переменной.

```
$ pg evncgi.cgi
#!/bin/sh
# envcgi.cgi
# вывод на печать настроек web-сервера с помощью команды env
echo "Content-type: text/html"
echo ""
echo "<HTML><PRE>"
echo "CGI Test ENVIRONMENTS"
echo "SERVER_SOFTWARE = $SERVER_SOFTWARE"
echo "SERVER_NAME = $SERVER_NAME"
echo "GATEWAY_INTERFACE = $GATEWAY_INTERFACE"
echo "SERVER_PROTOCOL = $SERVER_PROTOCOL"
echo "SERVER_PORT = $SERVER_PORT"
echo "REQUEST_METHOD = $REQUEST_METHOD"
echo "HTTP_ACCEPT = $HTTP_ACCEPT"
echo "PATH_INFO = $PATH_INFO"
echo "PATH_TRANSLATED = $PATH_TRANSLATED"
echo "QUERY_STRING = $QUERY_STRING"
echo "SCRIPT_NAME = $SCRIPT_NAME"
echo "REMOTE_HOST = $REMOTE_HOST"
echo "REMOTE_ADDR = $REMOTE_ADDR"
echo "REMOTE_USER = $REMOTE_USER"
echo "AUTH_TYPE = $AUTH_TYPE"
echo "CONTENT_TYPE = $CONTENT_TYPE"
echo "CONTENT_LENGTH = $CONTENT_LENGTH"
echo "</PRE></HTML>"
```

29.5. Введение в методы get и post

До сих пор мы рассматривали только вывод данных на экран. Для получения информации, введенной пользователем, следует использовать формы, при создании которых применяются сценарии cgi. В любом случае требуется средство, реализующее обработку результатов пользовательского ввода. Благодаря формам можно отображать текстовые поля, раскрывающиеся меню и переключатели.

После того как пользователь выполнил ввод или выбрал некоторые данные в форме, он может щелкнуть на кнопке send для передачи введенной информации сценарию, в данном случае — сценарию cgi. Как только информация будет введена, "в игру вступают" методы get и post.

29.5.1. Метод get

Для любой формы по умолчанию используется метод get. Это один из методов, применяемых для выборки файлов из статических HTML-страниц.

Как только пользователь щелкнет на кнопке submit, информация, которая была выбрана или выбирается пользователем, добавляется к URL сервера в виде закодированной строки. Затем эта закодированная строка присваивается переменной среды сервера, QUERY_STRING. Переменная REQUEST_METHOD также используется для хранения метода формы.

Создание простой формы

Создадим простую форму, реализующую ссылку из документа *main.shtml* на сценарий *booka.cgi*.

Вставьте следующие две строки после последней записи ссылки, которая была создана в файле *main.shtml*:

```
<BR> Basic form using GET method <A HREF="/cgi-bin/booka.cgi" >Form1</A>
```

Теперь введите следующий код и сохраните его в файле *booka.cgi*; не забудьте поместить этот файл в каталог *cgi-bin*.

```
$ pg booka.cgi
#!/bin/sh
# сценарий booka.cgi
echo "Content-type: text/html"
echo ""
echo "<HTML>"
echo "<BODY>"
# вызов booka_result.cgi, затем пользователь щелкает на кнопке отправки!
echo "<FORM action="/cgi-bin/booka_result.cgi" METHOD=GET>"

echo "<H4> CGI FORM</H4>"
# текстовое поле, результаты ввода присвоены переменной с именем 'contact'
echo "Your Name: <INPUT NAME=contact SIZE=30><BR><BR>"
# раскрытие выбранного пункта меню, присвоенного переменной 'film'
echo "<SELECT NAME=film>"
echo "<OPTION>"-- Pick a Film --"
echo "<OPTION>A Few Good Men"
echo "<OPTION>Die Hard"
echo "<OPTION>Red October"
echo "<OPTION>The Sound Of Music"
echo "<OPTION>Boys In Company C"
echo "<OPTION>Star Wars"
echo "<OPTION>Star Trek"
echo "</SELECT>"
# раскрытие выбранного пункта меню, присвоенного переменной 'actor'
echo "<SELECT NAME=actor>"
echo  "<OPTION>-- Pick Your Favourite Actor --"
```

```
echo "<OPTION>Bruce Will is"
echo "<OPTION>Basil Rathbone"
echo "<OPTION>Demi Moore"
echo "<OPTION>Lauren Bacall"
echo "<OPTION>Sean Connery"
echo "</SELECT>"
echo "<BR><BR>"
# имена переменных флажков 'view_cine' и 'view_vid'
echo "Do you watch films at the..<BR>"
echo "<INPUT TYPE="Checkbox" NAME=view_cine> Cinema"
echo "<INPUT TYPE="Checkbox" NAME=view_vid> On video"
echo "<BR><BR>"
# результаты ввода, присвоенного переменной 'textarea'
echo "Tell what is your best film, or just enter some comments<BR>"
echo "<TEXTAREA COLS="30" ROWS="4" NAME="textarea"></TEXTAREA>"

echo "<BR><INPUT TYPE=Submit VALUE="Send it">"
echo "<INPUT TYPE="reset" VALUE="Clear Form">"

echo "</FORM>"
echo "</BODY>"
echo "</HTML>"
```

Действие `form action` выбирается, как только пользователь щелкнет на кнопке 'Send it', в результате чего вызывается сценарий `booka_result.cgi`. В этом случае будет использоваться метод `get`.

В форме, код которой приведен выше, отображаются два текстовых поля, два раскрывающихся поля и два флажка.

Текстовое поле, предназначенное для ввода пользовательского имени, имеет длину 30 символов; результаты ввода присваиваются переменной `contact`.

Первое раскрывающееся меню обеспечивает выбор любимого пользовательского фильма; выбранная опция присваивается переменной `film`.

Второе раскрывающееся меню обеспечивает выбор любимого актера; выбранная опция присваивается переменной `actor`.

Можно установить один или оба флажка, выполнив щелчок мышью на требуемом варианте. Выбранные значения хранятся в переменных `view_cine` и `view_vid`. Если пользователь указывает один из флажков, переменные должны иметь значение "on".

Область текстового поля обеспечивает ввод большего количества строк текста, чем стандартное текстовое поле (текст, разбитый на 30 столбцов и 4 строки, в нашем случае), а вся введенная информация присваивается переменной `textarea`.

Для отсылки данных в качестве типа ввода указывается слово `submit`. Чтобы очистить форму, щелкните на кнопке `clear`.

Введите следующий сценарий cgi, назовите его *booka_result.cgi* и сохраните его в каталоге *cgi-bin*.

```
$ pg booka_result.cgi
#!/bin/sh
# сценарий booka_result.cgi
# вывод на печать настроек web-сервера env для метода get
echo "Content-type: text/html"
echo ""
echo "<HTML><PRE>"
echo "<PRE>"
```

```
echo " Results from a GET form"
echo "REQUEST_METHOD : $REQUEST_METHOD"
echo "QUERY STRING : $QUERY_STRING"
ech o "</PRE></HTML>"
```

Сценарий отображает значения пары переменных cgi, QUERY_STRING и REQUEST_METHOD. Переменная QUERY_STRING будет хранить все данные в виде кодированной строки, которая отправлена формой, созданной с помощью сценария *booka.cgi*. Переменная REQUEST_METHOD сохраняет тип используемого метода; в данном случае будет выбран метод get. На рис. 29.7 показано, как выглядит форма, созданная с помощью сценария *booka.cgi*.

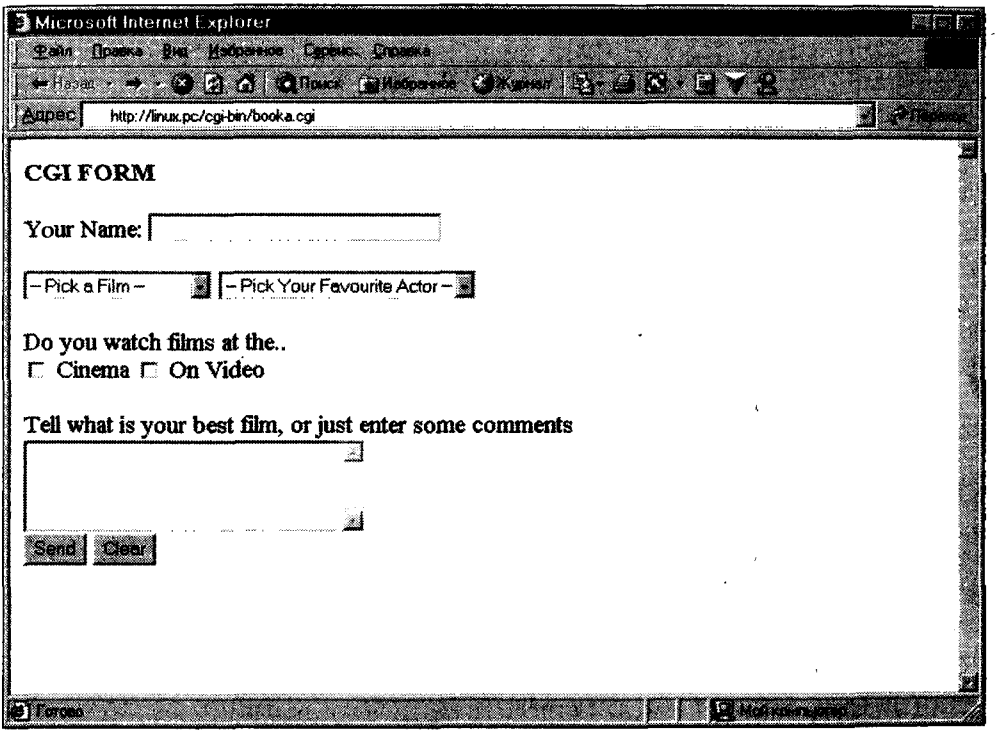


Рис. 29.7. Форма cgi, использующая метод get

Теперь выполним ввод и отсылку некоторой информации (рис. 29.8). После щелчка на кнопке 'Send it' отображается страница, показанная на рис. 29.9. Значение переменной QUERY_STRING отображается только частично по причине большой длины строки. Ниже приведена строка, имеющая полную длину:

```
contact=David+Tansley&film=The+Sound+Of+Music&actor=Bruce+
Willis&view_cine=on&view_vid=on&textarea=%21%22%A3%A3%24%25
%24%25%5E*%5E%26*%28%29*%28%29%28*%OD%OA
How%27s+that+%21%21
```

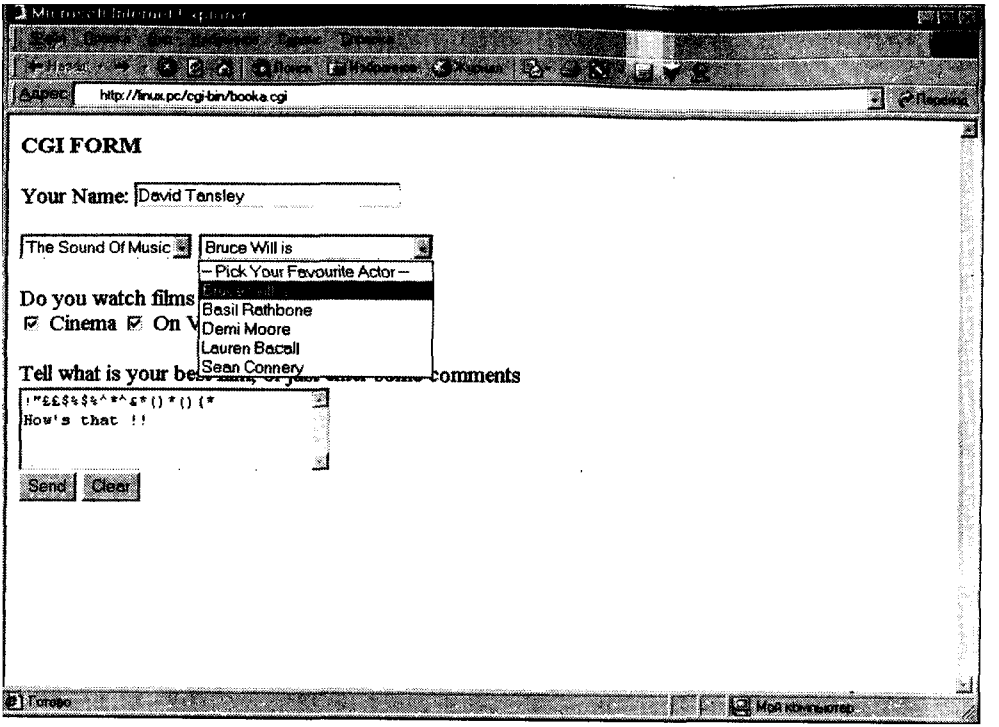


Рис. 29.8. Выбор и ввод информации в форму

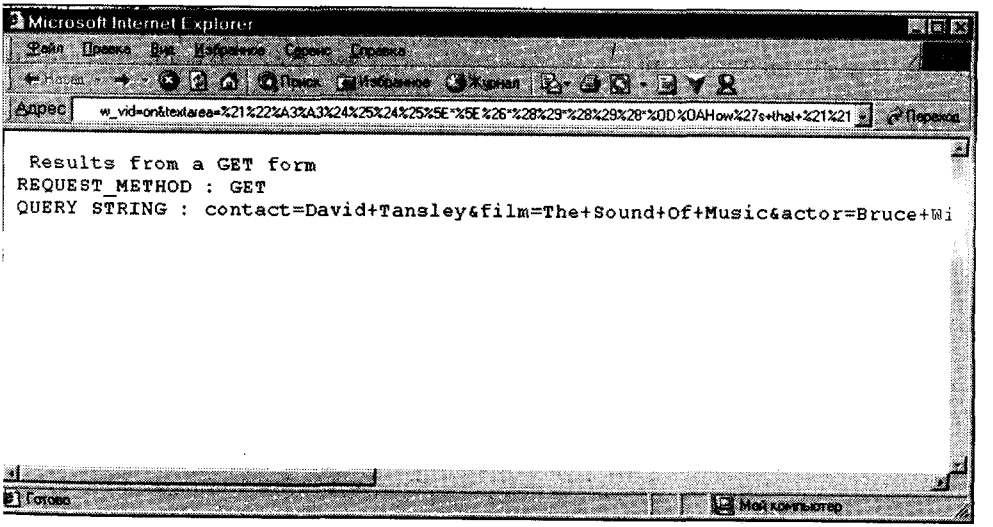


Рис. 29.9. Информация, отправленная формой, закодирована

Для пересланной информации необходимо иметь следующие поля, присвоенные переменной QUERY_STRING, показанной выше:

Переменная	Значение переменной
contact	David Tansley
film	The Sound of Music
actor	Bruce Willis
view_cine	Если установлено значение on, то флажок выбран
view_vid	Если установлено значение on, то флажок выбран
textarea	!"ff$%$%^*^&*()*()(* How's that !!

Декодирование закодированной строки

После того как пользователь щелкнет на кнопке "submit", информация присваивается переменной QUERY_STRING, а строка кодируется следующим образом:

- Все пробелы заменяются знаками +.
- Все поля значений разделяются символами &.
- Все значения и соответствующие поля разделяются знаками =.
- Все символы и некоторые специальные символы представляются кодами %xy, где xy является шестнадцатеричным эквивалентом данного символа. При просмотре переменной QUERY_STRING можно заметить, что многие из этих символов представлены переменной textarea.

Протокол cgi определяет, что любые символы в форме %xy (где xy является шестнадцатеричным числом) могут быть преобразованы в эквивалентные символы ASCII. Эти шестнадцатеричные символы состоят из специальных символов &, %, +, =, (,) и всех других символов, выходящих за рамки десятичного диапазона ASCII с границей 127. Например, символу (соответствует эквивалент в виде %29.

Шестнадцатеричные символы создаются в том случае, когда пользователь вводит значения в свободные текстовые поля. Однако эти символы могут также являться частью выбранного текста меню.

Для выполнения декодирования закодированной строки нужно выполнить следующее:

- заменить все символы & символами перевода строки;
- заменить все символы + пробелами;
- заменить все символы = пробелами;
- преобразовать все значения %xy в эквивалентные символы ASCII.

После завершения описанной выше последовательности действий должна быть возможность осуществить запрос или реализовать доступ к каждой переменной. Благодаря этому можно обрабатывать отсылаемую информацию. В ходе декодирования выполняется только половина работы, хотя и наиболее трудоемкая. Для обеспечения доступа к значениям переменных можно воспользоваться командой eval.

Следующий сценарий выполняет все необходимые преобразования, а также обеспечивает доступ к переменным. Сценарий снабжен множеством комментариев, поэтому разобраться в его работе не составит особого труда.

```
$ pg conv.cgi
#!/bin/sh
```

```
# сценарий conv.cgi
# декодирование строки URL
echo "Content-type: text/html"
echo ""
echo "<HTML><PRE>"
# отображение метода кодированной строки
echo "Method       : $REQUEST_METHOD"
echo "Query String : $QUERY_STRING"
echo "<HR>"
# применение редактора sed для замены символов & символами табуляции
LINE=`echo $QUERY_STRING | sed 's/&/    /g'`

for LOOP in $LINE
do
  # разбивка на поля NAME и TYPE
  NAME=`echo $LOOP | sed 's/=/ /g' | awk '{print $1}'`
  # получение TYPE, замена всех символов = пробелами, а %hex_num — \xhex_num
  # замена всех символов + пробелами
  TYPE=`echo $LOOP | sed 's/=/ /g' | awk '{print $2}' | \
  sed -e 's/%\(\)/\\\x/g' | sed 's/+/ /g'`
  # используется функция printf, которая отображает значения переменных
  # после завершения преобразований шестнадцатеричных значений
    printf "${NAME}=${TYPE}\n"
    # в переменную VARS записываются значения отдельных полей, которые
    # затем передаются команде eval, благодаря чему отдельные поля
    # можно адресовать; при этом, если поля содержат пробелы, требуется
    # удвоенная обратная косая черта
    VARS=`printf "${NAME}=\\${TYPE}\n"`
  eval `printf $VARS`
done
echo "<HR>"
# используется printf для отображения специальных символов в случае их наличия
printf "Your name is                  : $contact\n"
printf "Your choice of film is        : $film\n"
printf "Your choice of actor is       : $actor\n"
printf "You watch films at the  cinema : $view_cine\n"
printf "You watch films on video      : $view_vid\n"
printf "And here are your comments    : $textarea\n"
echo "</PRE>"
echo "</HTML>"
```

Нетрудно заметить, что в данном случае используется функция `printf` для вывода данных на экран. Причина этому весьма проста. Функция `printf` выполняет те же действия, что и обычная команда `echo`, но дополнительно выполняет шестнадцатеричные преобразования. В связи с этим следует сделать небольшое замечание. При использовании функции `printf` не происходит вставка символа новой строки; для устранения этого недостатка необходимо после каждой функции `printf` указать символы "\n". Шестнадцатеричные числа, хранящиеся в переменной `QUERY_STRING`, имеют формат `%hex_num`. Этот формат будет просто преобразован в формат `\xhex_num` с помощью потокового редактора `sed`, а также функции `printf`, выполняющей все необходимые преобразования. Зачем создавать себе дополнительные трудности, если для решения задачи существует простой способ?

Сохраните указанный выше сценарий под именем *conv.cgi* в каталоге *cgi-bin*. Теперь осталось выполнить небольшое изменение в сценарии `booka.cgi`, в результате чего форма будет вызывать сценарий `conv.cgi` вместо сценария `booka_result.cgi`. Для этого следует воспользоваться следующей строкой:

```
<FORM action = "/cgi-bin/conv.cgi" METHOD = GET>
```

Если теперь повторно передать форму (содержащую одну и ту же информацию), получим результаты, приведенные на рис. 29.10.

Теперь, когда строка имеет более удобочитаемый формат, можно выполнить некоторую обработку информации.

Метод `get` является стандартным методом, применяемым для работы с формами. В зависимости от имеющегося окружения, при использовании метода `get` существуют две потенциальные проблемы. Вся закодированная строка добавляется к адресу URL при отсылке информации, поэтому отсылаемая информация может быть просмотрена в окне URL. Хотя многие пользователи не видят в этом особой опасности, все же не стоит посылать информацию частного характера с помощью Web или сети.

Если пользовательская форма включает множество полей ввода, длина переменной `QUERY_STRING` может чрезмерно возрасти. В этом случае многие пользователи для работы с такими формами используют метод `post`. Этот метод подробно рассматривается в следующем разделе.

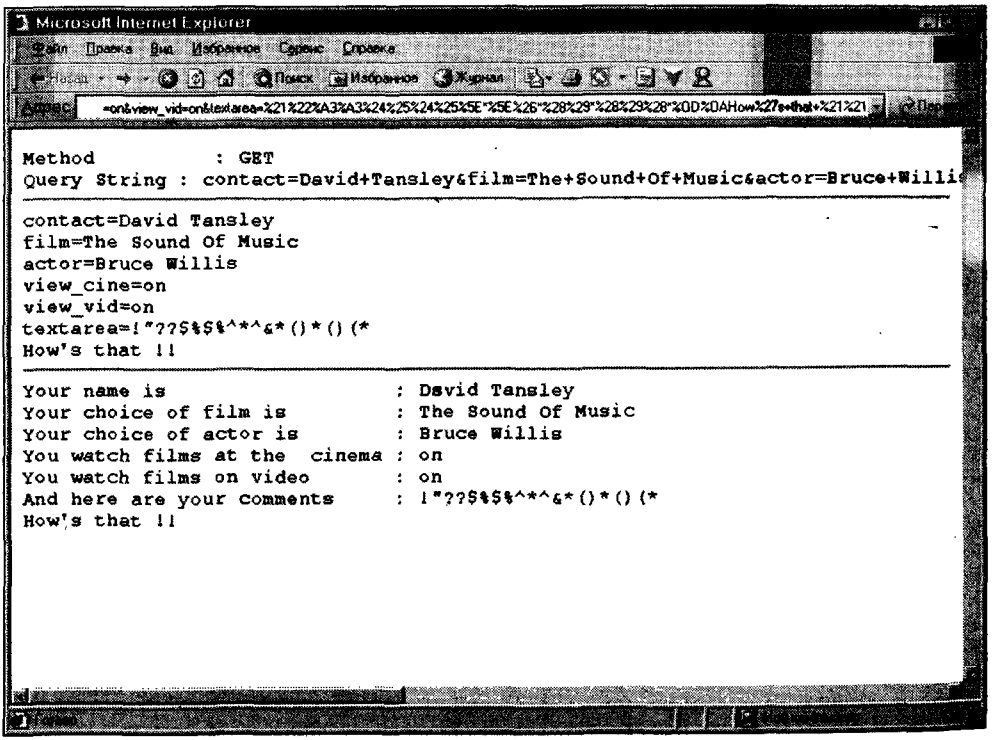


Рис. 29.10. Полностью декодированные данные формы

29.5.2. Метод post

Метод post, также как и метод get, предназначен для работы с закодированными строками. Разница заключается в способе получения данных: метод post считывает данные из стандартного потока. Для отсылки данных с помощью метода post просто замените ключевое слово get словом post в конструкции FORM action сценария.

```
<FORM action="/cgi-bin/conv.cgi" METHOD = POST>
```

Переменная CONTENT_LENGTH будет хранить общее количество байтов, отосланных с применением метода post. Производится считывание строки из потока стандартного ввода, а затем выполняется то же самое преобразование, что и при использовании метода get. Процесс считывания завершается после того, как считанное количество байтов становится равным количеству байтов, хранящихся в переменной CONTENT_LENGTH.

После выполнения небольшого изменения в конструкции form action получится обобщенный декодер форм. Для осуществления считывания из стандартного потока ввода можно использовать команду cat. Ниже показана конструкция, которую следует добавить в сценарий *conv.cgi*, в результате чего появится возможность использования методов get и post.

```
if ["$REQUEST_METHOD" = "POST"]; then
QUERY_STRING= `cat -`
fi
```

Обратите внимание на то, что команда cat содержит дефис, благодаря чему эту команду можно применять для считывания данных из стандартного потока ввода.

При использовании метода post осуществляется обычная проверка значения переменной QUERY_STRING. Затем все символы, поступающие из стандартного потока ввода, присваиваются переменной QUERY_STRING. В этой ситуации возможно использование метода get, поскольку в любом случае требуется получить информацию из переменной QUERY_STRING.

Замените строку FORM action в cgi-сценарии *booka.cgi*:

```
<FORM action ="/cgi-bin/conv.cgi" METHOD = GET>
```

строкой

```
<FORM action ="/cgi-bin/conv.cgi" METHOD = POST>
```

Кроме того, будут выполнены небольшие изменения в сценарии *conv.cgi*, благодаря чему можно будет проверять значения, введенные в текстовые поля, а также определять установленные флажки. Обновленный сценарий будет иметь следующий вид:

```
$ pg conv.cgi
#!/bin/sh
# conv.cgi
# декодирование строки URL
echo "Content-type: text/html"
echo ""
echo "<HTML><PRE>"
# это post ???
if [ "$REQUEST_METHOD" = "POST" ]; then
  QUERY_STRING=`cat -`
fi
```

```
# отображение имени метода и кодированной строки
echo "Method       : $REQUEST_METHOD"
echo "Query String : $QUERY_STRING"
echo "<HR>"
# используется sed для замены & символом табуляции
LINE=`echo $QUERY_STRINC | sed 's/&/    /g'`

for LOOP in $LINE
do
  NAME=`echo $LOOP | sed 's/=/ /g' | awk '{print $1}'`
  TYPE=`echo $LOOP | sed 's/=/ /g' | awk '{print $2}' | \
  sed -e 's/%\(\)/\\\x/g' | sed 's/+/ /g'`
  # используется printf для преобразования шестнадцатеричных символов
    printf "${NAME}=${TYPE}\n"
    VARS=`printf "${NAME}=\\${TYPE}\n"`
  eval `printf $VARS`
done
echo "<HR>"
if [ "$contact" != "" ]; then
  printf "Hello $contact, it's great to meet you\n"
else
  printf "You did not give me your name ... no comment !\n"
fi

if [ "$film" != "-- Pick a Film --" ]; then
  printf "Hey I agree, $film is great film\n"
else
  printf "You didn't pick a film\n"
fi

if [ "$actor" != "-- Pick Your Favourite Actor --" ]; then
  printf "So you like the actor $actor, good call\n"
else
  printf "You didn't pick a actor from the menu\n"
fi

if [ "$view_cine" = "on" ]; then
  printf "Yes, I agree the cinema is still the best place to watch a
film\n"
else
  printf "So you don't go to the cinema, do you know what you're missing\n"
fi

if [ "$view_vid" = "on" ]; then
  printf "I like watching videos at home as well\n"
else
  printf "No video!!, you're missing out on all the classics to rent or buy\n"
fi

if [ "$textarea" != "" ]; then
  printf " And here are your comments...OK $textarea\n"
else
  printf "No comments entered, so no comment !\n"
fi
echo "</PRE>"
echo "</HTML>"
```

Обратите внимание, что в этом сценарии везде используется функция `printf`; хотя в некоторых случаях могут быть задействованы конструкции `echo` (когда не требуется доступ к переменным). Применение функций `printf` улучшает восприятие сценариев.

Теперь загрузим форму и осуществим тестирование путем отсылки некоторых данных с помощью метода `post`:

```
http://<имя_сервера>/cgi-bin/booka.cgi
```

На рис. 29.11 иллюстрируются данные, введенные на Web-страницу. После завершения ввода некоторых данных щелкните на кнопке "Send". Результаты выполнения этой операции показаны на рис. 29.12.

Сценарий опрашивает различные переменные для того, чтобы установить факт ввода информации. Затем будет выполнена дальнейшая обработка, позволяющая убедиться в том, что все поля имеют значения. Если значения имеются не для всех полей, форма возвращается пользователю и отображается запрос на повторный ввод информации. Как только форма будет корректно заполнена, ее можно добавить в файл. Таким образом можно создавать небольшие базы данных.

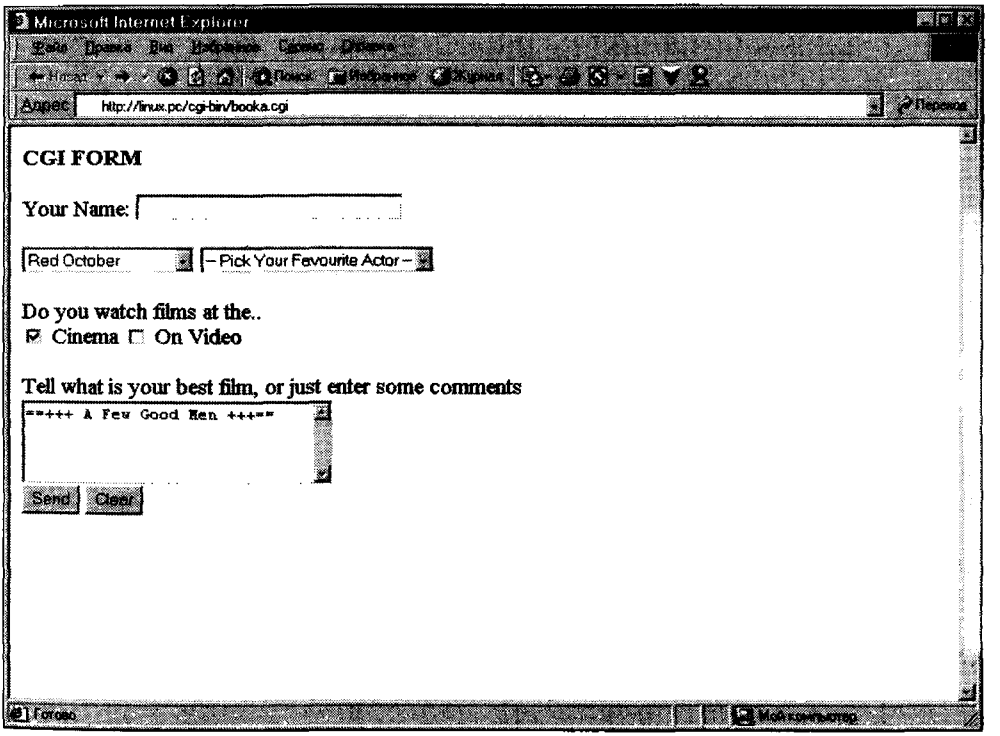


Рис. 29.11. Форма cgi, в которой используется метод post

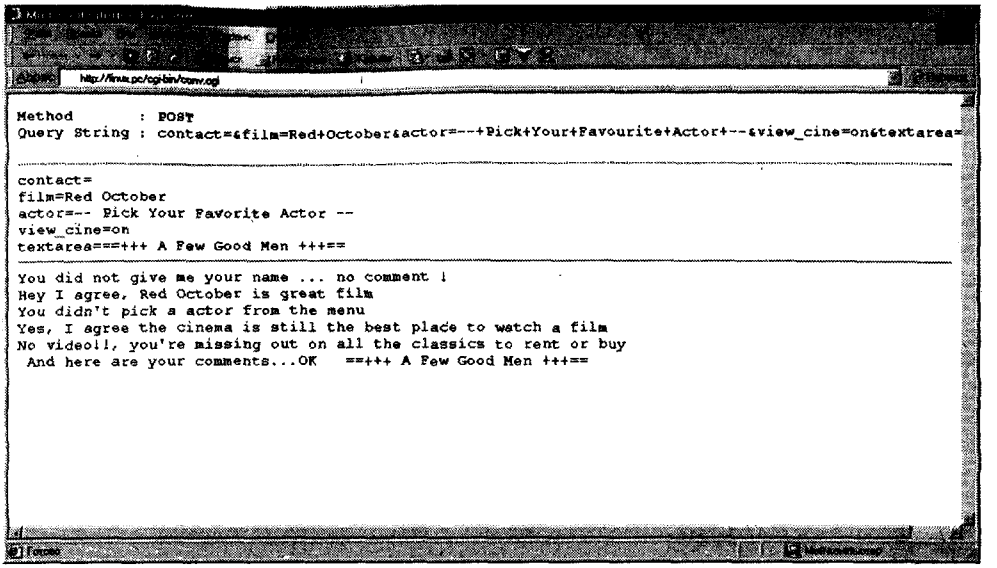


Рис. 29.12. Данные формы были полностью декодированы с помощью метода post

Практическое применение сценария cgi

Создадим сценарий, который будет выполнять некоторую полезную обработку. Пусть это будет отчет какой-то фиктивной компании, именуемой Wonder Gifts.

Файл отчета содержит номера различных товаров, проданных в каждом квартале 1998 г. Номера товаров соответствуют отделам Stationery, Books и Gifts.

Наша задача — создать отчет, который основан на запросе пользователя. Пользователь может делать выборку по номеру квартала либо по названию отдела. Необходимо выполнить дополнительную обработку, которая заключается в суммировании месячных продаж товаров по кварталам. Результаты могут выводиться на экран, принтер либо на экран и принтер одновременно.

В данном случае форма будет содержать два раскрывающихся меню и переключатель. Одно меню предназначено для выборки по кварталам, второе — для выборки по отделам. Переключатель служит для выбора устройства вывода. В нашем примере отчет выводится только на экран; переключатели выполняют демонстрационную роль.

Ниже приводится файл данных, содержащий сведения о продаже по кварталам. В нем имеются следующие поля: отдел, год, квартал, и количество различных товаров.

```
$ pg qtr_1998.txt
STAT    1998    1st    7998    4000    2344    2344
BOOKS   1998    1st    3590    1589    2435     989
GIFTS   1998    1st    2332    1489    2344     846
STAT    1998    2nd    8790    4399    4345     679
BOOKS   1998    2nd     889     430    2452     785
GIFTS   1998    2nd    9822    4822    3555     578
STAT    1998    3rd    8911    4589    2344    8690
BOOKS   1998    3rd     333    1489    6322     889
GIFTS   1998    3rd    2310    1483    3443     778
STAT    1998    4th    9883    5199    2344    6456
```

```
BOOKS  1998   4th   7333   3892   5223    887
GIFTS  1998   4th   8323   4193   2342    980
```

Сценарий формы.

```
$ pg gifts.cgi
#!/bin/sh
# сценарий gifts.cgi .... используется POST
echo "Content-type: text/html"
echo ""
echo "<HTML>"
echo "<BODY>"
# gifts_result.cgi используется для обработки вывода этой формы
echo "<FORM action="/cgi-bin/gifts_result.cgi" METHOD=POST>"
echo "<P>"
echo "<HR>"
echo "<H1><CENTER>GIFTS Inc <BR>"
echo "QUARTERLY REPORT</H1></CENTER>"
echo "</P><HR>"
echo "Department: <SELECT NAME=dept>"
echo "<OPTION>GIFTS"
echo "<OPTION>STATIONERY"
echo "<OPTION>BOOKS"
echo "</SELECT>"
echo "Quarter End:<SELECT NAME=qtr>"
echo "<OPTION>1st"
echo "<OPTION>2nd"
echo "<OPTION>3rd"
echo "<OPTION>4th"
echo "</SELECT>"
echo "<BR><BR>"
echo "Report To Co To:<BR>"
echo "<INPUT TYPE="radio" NAME= stdout VALUE=Printer >Printer"
echo "<INPUT TYPE="radio" NAME= stdout VALUE=Screen CHECKED>Screen"
echo "<INPUT TYPE="radio" NAME= stdout VALUE=Both >Both"
echo "<BR><BR><HR>"
echo "<INPUT TYPE=Submit VALUE="Send it">"
echo "<INPUT TYPE=Reset VALUE="Clear">"
echo "</FORM>"
echo "</BODY>"
echo "</HTML>"
```

Переменной `dept` присваивается выбранное значение для отдела; переменной `qtr` присваивается номер выбранного квартала. Переменной `stdout` присваивается значение "printer", "screen" или "both"; в качестве значения по умолчанию выбирается `screen` (это значение указывается с помощью слова "CHECKED"). Ниже приведен сценарий, обрабатывающий полученную информацию.

```
$ pg gifts_result.cgi
#!/bin/sh
# сценарий gifts_result.cgi
# декодирование строки URL
ec ho "Content-type: text/html"
echo ""
echo "<HTML><PRE>"
```

```
# это post ???
if [ "$REQUEST_METHOD" = "POST" ]; then
  QUERY_STRING='cat -'
fi
# декодирование
# используется sed для замены & символом табуляции
LINE=`echo $QUERY_STRING | sed 's/&/    /g'`
for LOOP in $LINE
do
  NAME=`echo $LOOP | sed 's/=/ /g' | awk '{print $1}'`
  TYPE=`echo $LOOP | sed 's/=/ /g' | awk '{print $2}' | \
  sed -e 's/%\(\)/\\\x/g' [ sed 's/+/ /g'`
  # используется printf при выполнении шестнадцатеричных преобразований
    VARS=`printf "${NAME}=\\${TYPE}\n"`
  eval `printf $VARS`
done
echo "<HR>"
echo "<H1><CENTER> GIFTS Inc</CENTER></H1>"
echo "<H2><CENTER> Quarter End Results </CENTER></H2>"
echo "<HR>"
# нужно изменить имена полей со STATIONERY на STAT
# для осуществления корректного поиска
if [ "$dept" = "STATIONERY" ];then
  dept=STAT
fi

# считывание из файла qtr_1995.txt
TOTAL=0
while read DEPT YEAR Q P1 P2 P3 P4
do
  if [ "$DEPT" = "$dept" -a "$Q" = "$qtr" ]; then
    TOTAL='expr $P1 + $P2 + $P3 + $P4'
  fi
  continue
done </home/httpd/cgi-bin/qtr_1995.txt
echo "<H2>"
echo " TOTAL ITEMS SOLD IN THE $dept DEPARTMENT"
echo " IS $TOTAL IN THE $qtr QUARTER"
echo "</H2><HR>"
# куда будет выведен отчет
if [ "$stdout" = "Both" ]; then
  echo "This report is going to the printer and the screen"
else
  echo " This report is going to the $stdout"
fi
echo "</PRE>"
echo "</HTML>"
```

Первая часть сценария является общей для любой формы, обработка которой осуществляется с помощью метода `post`. Поскольку отсутствуют шестнадцатеричные значения для преобразования (так как поля ввода являются предопределенными опциями меню), нет нужды в использовании функции `printf`, но особого смысла в том, чтобы отказаться от использования этой команды, нет. Содержательная часть сценария выполняет считывание из файла *qtr_1995.txt*.

Цикл `while` осуществляет считывание и присваивание значений полей переменным `DEPT`, `YEAR`, `Q`, `P1`, `P2`, `P3`, `P4` соответственно. Затем выполняется проверка значения переменной `$dept` (значение, отправленное пользователем) и переменной `DEPT`; результат конкатенируется с результатом другой проверки с помощью оператора `AND`. Если значение переменной `$qtr` (значение, отосланное пользователем) равно значению переменной `Q`, имеет место соответствие. Все числа, содержащиеся в сравниваемой строке, добавляются вместе.

В настоящее время в нашем распоряжении имеется сценарий формы и сценарий, предназначенный для обработки информации, пересылаемой формой (запустите этот сценарий). Введите URL (либо создайте соответствующую ссылку на главной странице):

```
http://<имя_сервера>/cgi-bin/gifts.cgi
```

Результаты показаны на рис 29.13.

Сценарий обрабатывает информацию, выбранную пользователем, и генерирует вывод, показанный на рис. 29.14.

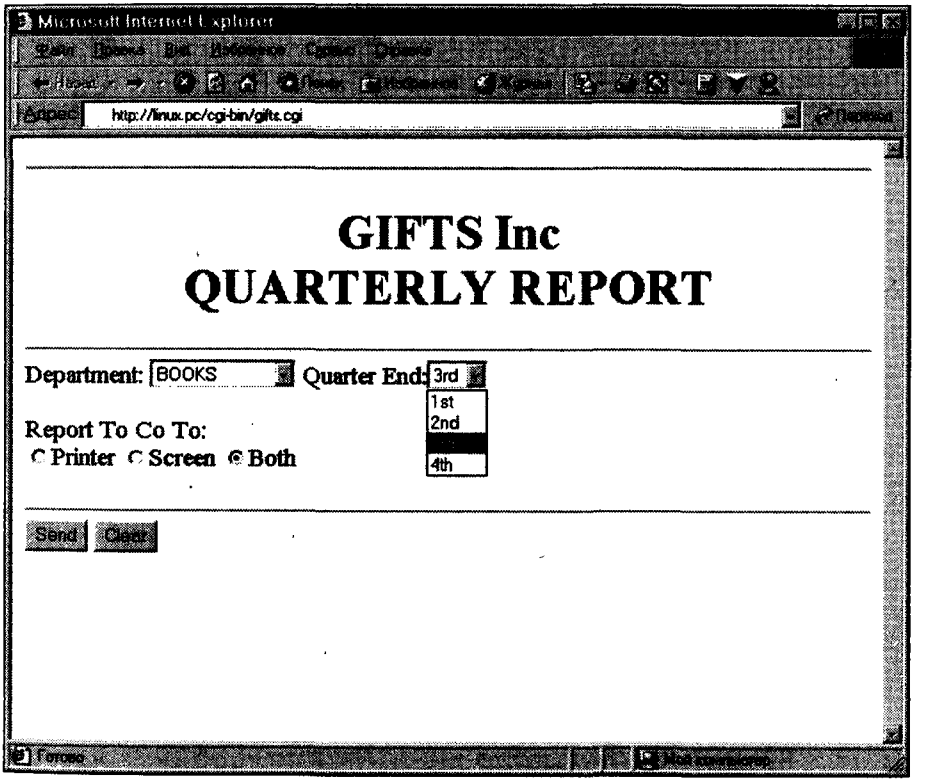


Рис. 29.13. Выборка квартальной информации для дальнейшей обработки

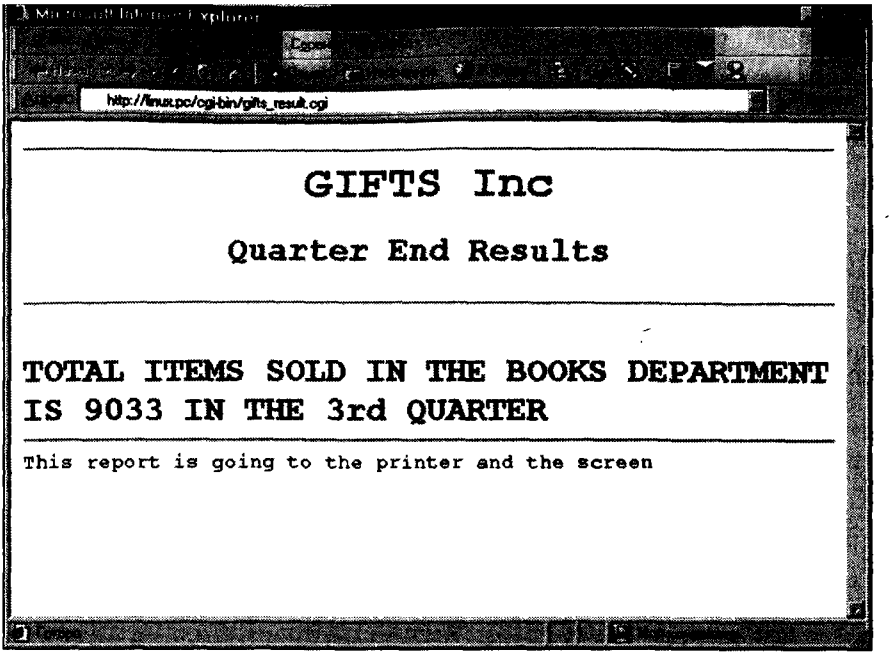


Рис. 29.14. Обработка завершена, вывод результатов

29.5.3. Заполнение списка

Если HTML-страницы являются действительно динамическими, следует предусмотреть возможность заполнения списков или таблицы текущими данными, выбранными из существующего файла, вместо того, чтобы жестко кодировать эти данные в сценариях cgi.

Следующий сценарий реализует заполнение раскрывающегося списка данными, содержащимися в текстовом файле *list*. Этот файл находится во временном каталоге, но не в корневом каталоге web-сервера. Цикл `while` используется для считывания содержимого (построчно) из файла. Для заполнения списка используется следующий кода:

```
echo "<OPTION>$LINE"
```

Выбранный элемент присваивается переменной `menu_selection`.

Ниже приводится сценарий, заполняющий элементы меню; действие формы не указывается.

```
$ pg populat.cgi
#!/bin/sh
# сценарий populat.cgi
# заполнение раскрывающегося списка значениями из текстового файла
echo "Content-type: text/html"
echo ""
echo "<HTML>"
echo "<BODY>"
echo "<H4> CGI FORM....populat.cgi..populate pull-down list from a text
```

```
file</H4>"
echo "<SELECT NAME=menu_selection>"
echo "<OPTION>-- PICK AN OPTION --"
# считывание значений из файла для заполнения опций списка
while read LINE
do
  echo "<OPTION>$LINE"
done < ../temp/list
echo "</SELECT>"
echo "</FORM>"
echo "</BODY>"
echo "</HTML>"
```

29.5.4. Автоматическое обновление Web-страницы

При использовании cgi для программирования заданий, выполняющих функции мониторинга либо контроля, часто бывает удобно выполнять обновление страниц в непрерывном режиме. Для этого вызывается пользовательский сценарий или страница. Ниже приводится тег, вызывающий выполнение сценария *dfspace.cgi* каждые 60 секунд.

```
<meta http-equiv = "Refresh" content = "60;URL =
http:/linux.pc/cgi-bin/dfspace.cgi">"
```

Здесь ключевым словом является слово Refresh. Благодаря его использованию Web-сервер получает сведения о загрузке данной страницы, а строка "content = 60" задает время (в секундах) между повторными загрузками. Для обновления сценария просто добавьте имя сценария в качестве части адреса URL.

В распоряжении автора имеются несколько контролирующих сценариев, выполняющих опрос всех основных хостов в сети. Благодаря их применению можно сразу же определить, какие хосты выполняются, а какие — отключены. Более симпатичный вид сценарию придает использование в тексте вместо опций `on` и `off` зеленых и красных шариков.

Ниже приводится сценарий, использующий часть вывода `df` и отображающий файловую систему и количество полей в таблице.

Следующий сегмент кода реализует заголовок таблицы, в который помещаются наименования колонок. При использовании таблиц с неотформатированным выводом может происходить потеря данных.

```
echo "<TABLE align="center" cellspacing="20" border=9 width="40%"
cols="2">"
echo "<TH align="center">- Capacity % -</TH>"
echo "<TH align="center">- File System -</TH>"
```

Параметр `cellspacing` устанавливает расстояние между внутренними и внешними границами таблицы. Параметр `border` хранит число, определяющее толщину табличной рамки. Параметр `cols` определяет количество столбцов в таблице. Ниже приведена основная часть описываемого сценария.

```
df |sed 1d| awk '{print $5"\t"$6}' | while read percent mount
do
  echo "<TR><TD align="center"><B>$percent</B></TD><TD
align="center">$mount</TD>
```

```
  </TR>"
done
```

С помощью команды df выполняется перенаправление посредством редактора sed для удаления заголовка, затем выполняется перенаправление к awk и считывание в пятом и шестом столбцах. Результаты присваиваются переменным percent и mount.

Аббревиатура TR обозначает строку таблицы, а TD — табличные данные. Тем самым определяется место, куда направляется информация.

Ниже приведен текст соответствующего сценария. Конечно, 60 секунд — это слишком много при мониторинге небольших файловых систем, но при перемещении большого количества файлов в файловых системах полезно отображать информацию об этом на протяжении целой минуты!

```
$ pg dfspace.cgi
#!/bin/sh
# сценарий dfspace.cgi
echo "Content-type: text/html"
echo ""
# автоматическое обновление каждые 60 секунд
echo "<meta http-equiv="Refresh" content="60;URL=http:/"linux.pc/cgi-bin/
dfspace.cgi">"
echo "<HTML>"
echo "<HR>"
echo "<A NAME="LINUX.PC Filesystems">LINUX.PC Filesystems</A>"
echo "<TABLE align="center" cellspacing="20" border=9 width="40%"
cols="2">"
echo "<TH align="center">- Capacity % -</TH>"
echo "<TH align="center">- File System -</TH>"
# получение вывода из df, но сначала фильтруется нужная информация!
df |sed 1d| awk '{print $5"\t"$6}' | while read percent mount
do
  echo "<TR><TD align="center"><B>$percent</B></TD><TD
align="center">$mount</TD>
  </TR>"
done
echo "</TABLE>"
echo "</HTML>"
```

При вводе URL

```
http://<имя_сервера>/cgi-bin/dfspace.cgi
```

в окне броузера отображается вывод, показанный на рис. 29.15. В вашем случае могут наблюдаться отличия.

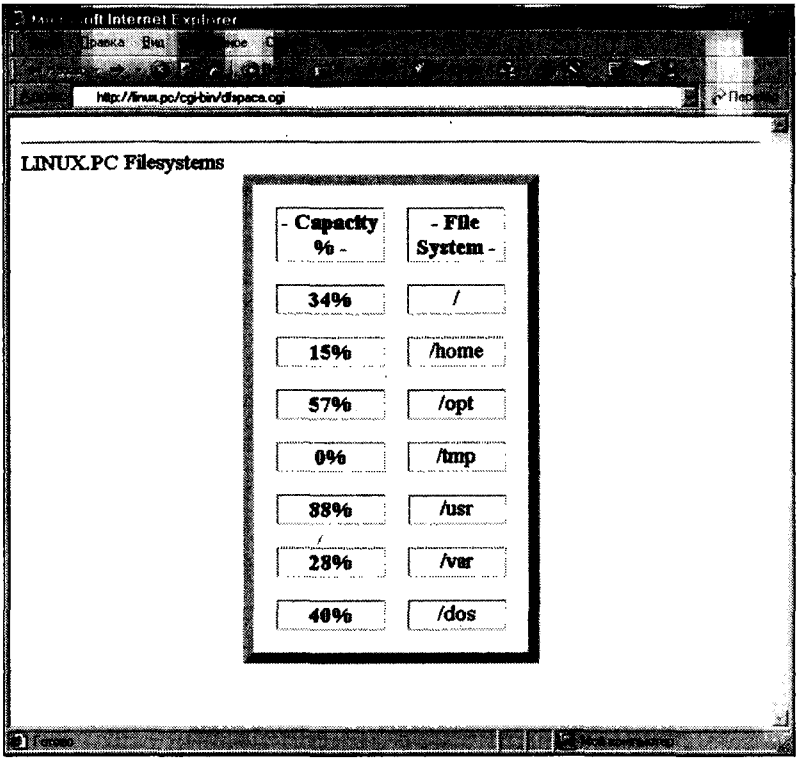


Рис. 29.15. Использование вывода команды df для генерирования таблиц

29.6. Заключение

Сценарии cgi обеспечивают создание весьма привлекательного пользовательского интерфейса. Страницы HTML могут использоваться в качестве интерфейса для всех распространенных задач, выполняющих обработку информации.

Создаваемые сценарии могут применяться для мониторинга, создания интерфейса, запросов к базам данных и решения других задач. Язык HTML является стандартным форматом для создания документации, поставляемой вместе с программным обеспечением.

ПРИЛОЖЕНИЕ А

Коды ASCII

Символ	Десятичное значение	Шестнадцатеричное значение	Восьмеричное значение
Ctrl-@ (NUL)	0	00	000
Ctrl-A	1	01	001
Ctrl-B	2	02	002
Ctrl-C	3	03	003
Ctrl-D (EOT)	4	04	004
Ctrl-E	5	05	005
Ctrl-F	6	06	006
Ctrl-G (BEL)	7	07	007
Ctrl-H (BS)	8	08	010
Ctrl-I (TAB)	9	09	011
Ctrl-J (NL)	10	0A	012
Ctrl-K	11	0B	013
Ctrl-L (FF)	12	0C	014
Ctrl-M (CR)	13	0D	015
Ctrl-N	14	0E	016
Ctrl-O	15	0F	017
Ctrl-P	16	10	020
Ctrl-Q	17	11	021
Ctrl-R	18	12	022
Ctrl-S	19	13	023
Ctrl-T	20	14	024
Ctrl-U	21	15	025
Ctrl-V	22	16	026
Ctrl-W	23	17	027
Ctrl-X	24	18	030
Ctrl-Y	25	19	031

Символ	Десятичное значение	Шестнадцатеричное значение	Восьмеричное значение
Ctrl-Z	26	1A	032
Ctrl-[(ESC)	27	1B	033
Ctrl-\	28	1C	034
Ctrl-]	29	1D	035
Ctrl-^	30	1E	036
Ctrl-_	31	1F	037
Пробел	32	20	040
!	33	21	041
"	34	22	042
#	35	23	043
$	36	24	044
%	37	25	045
&	38	26	046
'	39	27	047
(	40	28	050
)	41	29	051
*	42	2A	052
+	43	2B	053
,	44	2C	054
-	45	2D	055
.	46	2E	056
/	47	2F	057
0	48	30	060
1	49	31	061
2	50	32	062
3	51	33	063
4	52	34	064
5	53	35	065
6	54	36	066
7	55	37	067
8	56	38	070
9	57	39	071
:	58	3A	072
;	59	3B	073
<	60	3C	074

Символ	Десятичное значение	Шестнадцатеричное значение	Восьмеричное значение
=	61	3D	075
>	62	3E	076
?	63	3F	077
@	64	40	100
A	65	41	101
B	66	42	102
C	67	43	103
D	68	44	104
E	69	45	105
F	70	46	106
G	71	47	107
H	72	48	110
I	73	49	111
J	74	4A	112
K	75	4B	113
L	76	4C	114
M	77	4D	115
N	78	4E	116
O	79	4F	117
P	80	50	120
Q	81	51	121
R	82	52	122
S	83	53	123
T	84	54	124
U	85	55	125
V	86	56	126
W	87	57	127
X	88	58	130
Y	89	59	131
Z	90	5A	132
[	91	5B	133
\	92	5C	134
]	93	5D	135
^	94	5E	136
_	95	5F	137

Символ	Десятичное значение	Шестнадцатеричное значение	Восьмеричное значение
`	96	60	140
a	97	61	141
b	98	62	142
c	99	63	143
d	100	64	144
e	101	65	145
f	102	66	146
g	103	67	147
h	104	68	150
i	105	69	151
j	106	6A	152
k	107	6B	153
l	108	6C	154
m	109	6D	155
n	110	6E	156
o	111	6F	157
p	112	70	160
q	113	71	161
r	114	72	162
s	115	73	163
t	116	74	164
u	117	75	165
v	118	76	166
w	119	77	167
x	120	78	170
y	121	79	171
z	122	7A	172
{	123	7B	173
|	124	7C	174
}	125	7D	175
~	126	7E	176
Ctrl-? (DEL)	127	7F	177

ПРИЛОЖЕНИЕ Б

Полезные команды интерпретатора shell

В этом приложении перечисляются и описываются некоторые полезные команды интерпретатора shell. Здесь не приводится исчерпывающий список параметров для каждой команды. Однако имеющихся параметров вполне достаточно для понимания работы команд.

Дополнительные примеры использования можно найти в других главах книги.

basename

Формат:

```
basename путь
```

Команда `basename` выделяет путь из имеющегося имени пути и просто возвращает имя файла. Эта команда обычно применяется в конструкциях `usage` в сценариях. В этом случае подстановка используется при отображении на экране имен файлов.

```
$ basename /home/dave/myscript
myscript

echo "Usage: `basename $0` give me a file"
exit 1
...
```

Если приведенный выше код именуется *myscript*, вывод может быть следующим:

```
myscript: give me a file
```

Здесь `$0` является специальной переменной, которой присвоено текущее полное имя пути для сценария.

cat

Формат:

```
cat параметры файлы
```

Параметры:

`-v` Задается отображение управляющих символов

Команда `cat` является одной из наиболее часто применяемых команд постраничной разбивки текстовых файлов.

```
$ cat myfile
```

Отображение содержимого файла *myfile*:

```
$ cat myfile myfile2 >>hold_file
```

Приведенная выше команда осуществляет объединение двух файлов (*myfile* и *myfile2*) в один файл, именуемый *hold_file*.

```
cat dtl | while read line
do
  echo $line
done
```

Команда `cat` также используется для считывания файлов, осуществляемого с помощью сценариев.

compress

Формат:

```
compress параметры файлы
```

Параметры:

`-v` Вывод на экран результатов сжатия

Команда `compress` используется для уменьшения размера файлов. После завершения сжатия файл получает расширение *.Z*. Для возврата файла в исходное состояние используется команда `uncompress`.

```
$ compress myfile
$ ls myfile*
myfile.Z
```

cp

Формат:

```
cp параметры файл1 файл2
```

Параметры:

`-i` Запрос на подтверждение перезаписи файлов
`-p` Сохранение набора прав доступа и времени изменения
`-r` Рекурсивное копирование каталога

Для копирования файла *myfile* в *myfile1.bak* применяется команда:

```
$ cp myfile 1 myfile1.bak
```

Копирование файла *get.prd* из каталога */usr/local/sybin* в каталог */usr/local/bin* осуществляется командой

```
$ pwd
usr/local/sybin
$ cp get.prd../bin
```

Для рекурсивного копирования всех файлов и подкаталогов из каталога */logs* (вниз) в каталог */hold/logs* применяется команда:

```
$ cp -r /logs/ /hold/logs
```

diff

Формат:

```
diff параметры файл1 файл2
```

Параметры:

`-c` Генерирует различный вывод на основе стандартного формата (см. ниже)

`-I` Игнорирование регистров символов

При использовании файлов *файл1* и *файл2* из нашего примера `comm` команда `diff` будет выводить строки, которые не совпадают в заданных файлах.

```
$ diff файл1 файл2
2,3c2,3
<The game
<Boys in company C
---
>The games
>The boys in company C
```

Команда `diff` сообщает о том, что строки 2 и 3 различны, но второй столбец в строке 3 не совпадает.

dircmp

Формат:

```
dircmp параметры каталог1 каталог2
```

Параметры:

`-s` Не отображать различные файлы

Действие команды `dircmp` напоминает действие команды `diff`: она также производит сравнение с последующим выводом на экран найденных различий.

dirname

Формат:

```
dirname имя_пути
```

В противоположность команде `basename`, команда `dirname` выводит на экран только имя пути:

```
$ dirname /home/dave/myfile/home/dave
```

du

Формат:

```
du параметры каталог
```

Параметры:

`-a` Отображается размер каждого файла, а не только размер каталогов

`-s` Отображается только итоговая сумма

Команда du выводит информацию об использовании диска в виде блоков размером по 512 байтов. Эта команда применяется, главным образом, для вывода размеров каталогов.

```
$ pwd
/var
$ du -s
14929
```

Размер структуры каталога */var* составляет 14929 блоков.

file

Формат:

```
file имя_файла
```

Благодаря этой команде интерпретатор shell определяет тип файла.

```
$ file core
core: ELF 32-bit LSB core file of 'awk' (signal 6), Intel 80386,
version 1

$ file data.f
data.f: ASCII text

$ file month_end.sh
month_end.sh: Bourne shell script text
```

fuser

Формат:

```
fuser параметры файл
```

Параметры:

`-k` Уничтожает все процессы для файла или файловой системы

`-u` Отображает все процессы для файла или файловой системы

Команда `fuser` применяется для отображения процессов, выполняющихся в файловой системе или для файлов, к которым осуществляется доступ. В некоторых системах возможен выбор при использовании параметров `-u` и `-m`. Совместно с командой `fuser` возможно применение конструкции `if`.

Для отображения списка активных процессов, соответствующих устройству */dev/hda5*, применяется команда:

```
$ fuser -m /dev/hda5
/dev/hda5: 1 1r 1c 1e 37 37r 37c 37e 144
144r 144c 144e 158 158r 158c 158e 167r 167c
167e 178 17 8r 178c 178e 189 189r 189c
```

Уничтожение всех процессов, связанных с устройством */dev/hda5*, можно осуществить посредством команды:

```
$ fuser -k /dev/hda5
```

Для проверки, что файл *doc_part* открыт, и для определения выполняющихся процессов используется следующая команда:

```
$ fuser -m /root/doc_part
/root/dt: 1 1r 1c 1e 37 37r 37c 37e 144
144r 144c 144e 158 158r 158c 158e 167r 167c
167e 178 178r 178c 178e 189 189r 189c 189e 201
201r 201c 201e 212 212r 212c 212e 223 223r
```

Некоторые варианты команды `fuser` отображают идентификаторы регистрации в качестве части вывода. Если же в вашем случае этого не происходит, используйте номера, оканчивающиеся на 'e', а затем примените команду `grep` посредством `ps xa` или `ps -ef`.

head

Формат:

```
head -number файлы
```

Команда `head` используется для отображения первых десяти строк файла. Для отображения меньшего или большего количества строк используется параметр `-number`. Например, команда

```
$ head -1 myfile
```

задает отображение первой строки файла, а команда

```
$ head -30 logfile | more
```

приводит к отображению первых 30 строк файла *logfile*.

logname

Формат:

```
logname
```

При этом отображается регистрационное имя текущего пользователя:

```
$ logname
dave
```

mkdir

Формат:

```
mkdir параметры каталог
```

Параметры:

`-m` Устанавливает уровень доступа при создании каталога

Команда

```
$ mkdir HOLD_AREA
$ ls -1 HOLD*
-rw-rw-r-- 1 dave admin 3463 Dec 3 1998 HOLD_AREA
```

приведет к созданию каталога *HOLD_AREA*.

more

Формат:

```
more параметры файлы
```

Эта команда эквивалентна командам page и pg, т.е. поочередно отображает на экране содержимое страниц.

Параметры:

-c Не прокручивать текст, но отображать полностью страницу
-d Отображение запроса на ввод при постраничной разбивке файла
-n Отображение n строк вместо отображения полного экрана

Команда

```
$ more /etc/passwd
```

отображает содержимое файла *passwd*.

Команда

```
$ cat logfile |more
```

отображает файл *logfile*.

nl

Формат:

```
nl параметры файл
```

Параметры:

-I Задает приращение каждой строки на n; по умолчанию задается 1
-p Не восстанавливать нумерацию при появлении новой страницы

Команда nl используется для добавления нумерации в файл. Она является полезной при печати исходного кода или листингов журнальных файлов. Команда

```
$ nl myscript
```

добавляет нумерацию в файл *myscript*.

Команда

```
$ nl myscript >hold_file
```

направляет вывод команды nl в файл *hold_file*, а команда

```
$ nl myscript |lpr
```

направляет вывод команды nl на принтер.

printf

Формат:

```
printf формат аргументы
```

Эта команда выводит форматированный текст в стандартный поток вывода, а ее действие подобно действию функции printf утилиты awk.

Параметр формат может включать три различных типа элементов; здесь будут рассмотрены элементы форматирования. Форматирующая последовательность выглядит следующим образом:

```
%[- +]m.nx
```

Знак дефиса задает выравнивание текста в поле по левому краю. Вообще говоря, параметр m используется для представления длины поля, а n задает максимальную длину поля.

Символ % предшествует любому из следующих символов форматирования:

s	строка
c	символ
d	десятичное число
x	шестнадцатеричное число
o	восьмеричное число

Команда printf не может создавать новые строки; с этой целью применяются управляющие последовательности. Ниже приводится перечень наиболее часто применяемых управляющих последовательностей:

\a	звуковой сигнал
\b	удаление предшествующего символа
\r	возврат каретки
\f	прокрутка страницы
\n	создание новой строки
\t	символ табуляции

Команда

```
$ printf "Howzat!\n"
Howzat!
```

выводит строку в поток стандартного вывода; используйте символ '\n' для создания новой строки. Команда

```
$ printf "\x2B\n"
+
```

преобразует шестнадцатеричное число 2B в соответствующее десятичное значение ASCII, '+'.

Команда

```
$ printf "%-10sStand-by\n"
          Stand-by
```

выводит на печать строку, выровненную по левому краю, начиная с 10-го символа от левого края.

pwd

Формат:

```
pwd
```

Эта команда применяется для отображения текущего рабочего каталога. Введите следующую команду:

```
$ pwd
/var/spool
```

```
$ WHERE_AM_I=`pwd`
$ echo $WHERE_AM_I
/var/spool
```

Здесь используется подстановка для передачи сценарию сведений о текущем рабочем каталоге.

rm

Формат:

```
rm параметры файлы
```

Параметры:

`-I` Запрос перед удалением каждого файла
`-r` Удаление существующего каталога

Команда `rm` удаляет файлы и/или каталоги. Команда

```
$ rm myfile
$ rm -r /var/spool/tmp
```

удаляет все файлы, включая подкаталоги из каталога */var/spool/tmp* и ниже.

rmdir

Формат:

```
rmdir параметры каталоги
```

Параметры:

`-p` Удаление всех пустых каталогов, найденных в процессе удаления

Команда

```
$ rmdir /var/spool/tmp/lp_HP
```

удаляет каталог *lp_HP*, находящийся в каталоге */var/spool/tmp*.

script

Формат:

```
script параметр файл
```

Параметры:

`-a` Добавление вывода в файл

С помощью команды `script` можно создать полную хронологию сеанса. Для этого нужно просто вызвать эту команду из командной строки. Выполнение команды `script` завершается после выхода из сеанса. Эта команда копирует введенные пользователем команды и данные и добавляет их в файл. Команда

```
$ script mylogin
```

задает регистрацию информации о сеансе в файле *mylogin*.

shutdown

Формат:

```
shutdown
```

Выполнение этой команды приводит к завершению работы системы. Многие поставщики программного обеспечения поддерживают свои специфические версии этой команды. Команда

```
$ shutdown now
```

вызовет немедленное завершение работы системы, а команда

```
$ shutdown -g60 -16 -y
```

приведет к завершению работы системы через 60 секунд, после чего последует перезагрузка системы.

sleep

Формат:

```
sleep число
```

Применение этой команды вызовет приостановку работы системы на указанное количество секунд. Например, команда

```
$ sleep 10
```

приостановит систему на 10 секунд.

strings

Формат:

```
strings имя_файла
```

Команда `strings` может быть использована для просмотра текста, содержащегося в двоичных файлах.

touch

Формат:

```
touch параметры имя_файла
```

Параметры:

`-t ММДДччмм`	Создание файла, содержащего штамп даты (месяц, день, час, минута).

Эта команда создает файл с текущим или новым штампом даты.

```
$ touch myfile
$ ls -l myfile
-rw-r--r-- 1 dave admin 0 Jun 30 09:59 myfile
```

Данный код создает новый пустой файл *myfile*, содержащий текущую дату/время.

```
$ touch -t 06100930 myfile2
$ ls -l myfile2
-rw-r--r-- 1 dave admin 0 Jun 10 09:30 myfile2
```

Указанная выше команда `touch` создает новый пустой файл *myfile2* со штампом даты `June 10, 09:30am`.

tty

Формат:

```
tty
```

Используйте команду `tty` для получения сведений о том, на каком устройстве или терминале вы работаете.

```
$ tty
/dev/tty08
```

Команда `tty -s` определяет, является ли сценарий интерактивным. Коды возврата в этом случае будут следующими:

0	Для терминала
1	Для устройства, не являющегося терминалом

uname

Формат:

```
uname параметры
```

Параметры:

-a	Отображение всей информации
-s	Системное имя
-v	Отображение только номера версии либо даты выпуска версии

Команда используется для отображения имени текущей системы и другой связанной информации:

```
$ uname a
Linux bumper.honeysuckle.com 2.0.36 #1 Tue Oct...
```

uncompress

Формат:

```
uncompress файлы
```

Команда `uncompress` используется для разархивирования любых сжатых файлов.

```
$ uncompress myfile
```

Приведенная выше команда разархивирует файл *myfile*, который был ранее сжат. Обратите внимание, что после разархивирования файла для него не может использоваться расширение `.Z`.

wait

Формат:

```
wait ID процесса
```

Эта команда устанавливает длительность ожидания для ID процесса перед возобновлением его выполнения либо устанавливает длительность ожидания всех фоновых процессов перед возобновлением их выполнения.

Для задания интервала ожидания ID процесса перед возобновлением его выполнения используется команда:

```
$ wait 1299
```

Следующая команда определяет ожидание до тех пор, пока не завершатся все фоновые процессы:

```
$ wait
```

wc

Формат:

```
wc параметры файлы
```

Параметры:

`-c` Вывод количества символов

`-l` Вывод количества строк

`-w` Вывод количества слов

Эта команда осуществляет подсчет количества символов или слов.

```
$ who | wc
        1 6 46
$ who | wc -l
        1
```

В первом примере вывод команды `who` направляется команде `wc`; при этом отображаются следующие столбцы:

```
количество строк, количество слов, количество символов
```

Во втором примере команда `wc` просто выводит на экран количество строк.

```
$ VAR="tapedrive"
echo $VAR | wc -c
10
```

В результате применения этой команды выводится количество символов в строке VAR.

whereis

Формат:

```
whereis имя_команды
```

Команда `whereis` используется для поиска двоичных или текстовых страниц справки для команды.

```
$ whereis fuser
fuser: /usr/sbin/fuser /usr/man/man1/fuser.1

$ whereis sort
sort: /bin/sort /usr/man/man1/sort.1
```

Обратите внимание, что двоичные файлы не отображаются в следующих двух примерах, поскольку они встроены в интерпретатор shell, но в этом случае для команд имеются справочные страницы.

```
$ whereis times
times: /usr/man/man2/times.2

$ whereis set
set: /usr/man/mann/set.n
```

who

Формат:

```
who параметры
```

Параметры:

- `-a` Отображение всего вывода
- `-r` Отчет о текущем уровне выполнения (в Linux применяется команда `runlevel`)
- `-s` Отображение полей имен и дат

Команда

```
whoami
```

Отображает имя пользователя, выполняющего команду. Эта команда не является параметром команды `who` и может быть вызвана отдельно.

Команда `who` выводит отчет о пользователях, зарегистрированных в системе. Для отображения информации об этих пользователях введите команду:

```
$ who
root   console Apr 22 13:27
pgd    pts/3   Jun 14 15:29
peter  pts/4   Jun 14 12:08
dave   pts/5   Jun 14 16:10
```

Сведения о самом себе можно получить с помощью следующей команды:

```
$ whoami
dave
```

Предметный указатель

Л

М

О

П

Р

С

Содержание

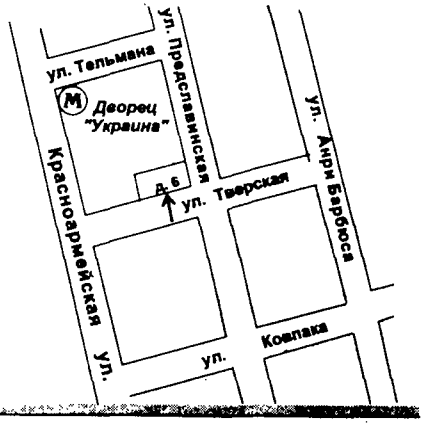
Издательская группа BHV рекомендует

серию "Руководство разработчика"

Официальные представители Издательской группы BHV:

Почтовый адрес:
03150, г. Киев,
а/я 469.
Тел./факс:
(044) 269-25-78, 269-04-23
E-mail: market@irina-press.kiev.ua

www.bhv.kiev.ua,
www.bookshop.kiev.ua



Почтовый адрес:
123098, г. Москва,
ул. Маршала Новикова,
д. 4. корп. 1;7

Тел./факс офиса:
(095) 196-04-54,
196-04-27

Тел./факс склада:
(095) 196-07-19,
196-08-28

E-mail: sales@sparrk.pikenet.ru

Почтовый адрес:
191011, г. С-Петербург,
а/я 44.
Тел./факс:
(812) 162-57-84,

E-mail: an@elbi.spb.su
www.bhv.kiev.ua

Учебное пособие

Дэвид Тейнсли

Linux и UNIX: программирование в shell. Руководство разработчика

Редакторы Н.Е. Курбатова, Е.А. Курбатова,
Технический редактор З.В. Лобач

Продукция соответствует требованиям Министерства здравоохранения Российской Федерации. Гигиеническое заключение 77.ФЦ.8.953.П.197.3.99 от 12.03.1999

Лицензия на издательскую деятельность № 071405
от 28 февраля 1997 г.
ООО «Спаррк».
123364, г. Москва, ул. Свободы, д. 28, корп. 2.

ООО "Издательская группа BHV"
Свидетельство о занесении в Государственный реестр
серия ДК №175 от 13.09.2000

Подписано в печать 05.10.2001. Формат 70×100$^{1}/_{16}$.
Печать офсетная. Усл. печ. л. 29. Тираж 5000 экз.
Заказ № 1885.

Отпечатано с готовых диапозитивов в ОАО «Типография "Новости"»
107005, Москва, ул. Фр. Энгельса, 46.

Linux и UNIX: программирование в shell

Руководство разработчика

Дэвид Тейнсли имеет многолетний опыт работы системным администратором UNIX и Linux на различных платформах. В настоящее время он занимает должность системного администратора и администратора баз данных в фирме Ace Global Markets, принадлежащей Lloyds of London Underwriting Agency.

- Команды интерпретатора shell и их синтаксис
- Утилиты фильтрации текста
- Среда, создаваемая при регистрации пользователя в системе, и ее настройка
- Базовые принципы программирования в среде интерпретатора shell
- Приемы создания сложных сценариев в среде интерпретатора shell

Данная книга представляет собой практическое руководство по программированию в интерпретаторе Bourne shell — стандартном командном интерпретаторе UNIX, полностью совместимом с интерпретатором BASH в Linux.

Издание предназначено как для начинающих, так и для опытных программистов и содержит множество полезных примеров, советов и подсказок. С его помощью читатель быстро научится создавать shell-сценарии для решения задач, возникающих при повседневной работе в большинстве систем UNIX и Linux.

ISBN 5-7315-0114-9

ISBN 966-552-085-7

http://www.bhv.kiev.ua
http://www.bookshop.kiev.ua

bhv®