Расширенное администрирование Linux.

Блок 1

v 1.03

Оглавление

Основы shell script

Две разновидности shell:
- Bourne shell
- C shell

Почему администратор должен знать shell script?

Дополнительные возможности, встроенные в shell script:
- переменные
- массивы
- условные операторы
- циклы

В shell встроен язык программирования, называемый shell script. Поскольку, существуют две основные разновидности интерпретаторов shell: Bourne shell и C shell, имеются две разновидности языков программирования, встроенных в эти оболочки.

Язык C shell, по своему синтаксису напоминает язык программирования C. Именно по этому оболочка и получила своё название. Язык Bourne shell — это самостоятельный язык со своим синтаксисом и особенностями.

Для чего администратору Linux необходимо знать shell script? Дело в том, что этот язык разрабатывался как вспомогательный инструмент администратора.

Одна из замечательных особенностей UNIX — это большой набор небольших утилит, каждая из которых выполняет свою задачу. При помощи конвеерной обработки команд эти утилиты можно объединять для выполнения более сложных задач. К сожалению, не все можно решить путем составления конвейеров, иногда требуется нечто большее. Можно написать программу на каком-либо языке программирования, но для этого мало того, что надо изучить этот язык, в системе должен присутствовать компилятор. Программы на shell script не надо компилировать, так как они выполняются интерпретатором, встроенным в оболочку. То есть shell script — это интерпретируемый язык.

Кроме того, вся система инициализации Linux (загрузка системы) построена с использованием набора файлов на языке shell script и администратор Linux может по своему усмотрению изменять эти файлы, но для этого необходимы хотя бы базовые знания языка.

Файл с программой на shell script — это набор команд, которые пользователь может вводить в командной строке. И если файл запустить на выполнение, будут выполнены все команды, описанные в этом файле. Но кроме простого выполнения команд в shell имеются дополнительные возможности, присущие многим языкам программирования: переменные, массивы, условные операторы, циклы и другое.

В данном курсе будут изучаться основы языка, встроенного в Bourne shell, т.к. именно эта разновидность используется в файлах системы инициализации Linux.

Запуск приложений

При запуске программы проверяются условия:

- право на исполнение
- является ли файл бинарным исполняемым файлом
- если это текстовый файл, есть ли в первой строке путь к интерпретатору

При использовании bash первая строка shell script должна быть такой:

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

Для того, чтобы файлы shell script можно было запускать, их следует сделать исполняемыми. В первой строке указать путь к интерпретатору, который будет вызван для выполнения программы. Поскольку в Linux наибольшее распространение получила разновидность Bourn shell — bash, в начале файла необходимо писать:

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

Существует еще один вариант запуска программ, написанных на shell scrit, причем файл с программой может быть не исполняемым. В этом случае необходимо быть уверенным в том, что в файле используются команды оболочки, в которой Вы сейчас работаете. Для запуска на выполнение программы в командной строке необходимо набрать:

```
.  programm
```

В этом случае, указанная программа будет выполнена экземпляром оболочки, в которой Вы сейчас работаете. То есть, для выполнения программы не будет порожден новый процесс. Правда, если в коде программы будет выполнен оператор exit — Вы выйдете из системы. Так же возможен явный вызов интерпретатора в командной строке, например:

```
bash program.sh
```

Переменные в shell script

Переменные в shell script:
- не типизированы
- область видимости переменных – весь код программы
- при обращении к неопределенной переменной не выдаются ошибки

Пример использования переменной:

```
PERM=value
echo $PERM
```

В директории /root/bin находятся файлы с примерами, на основе которых будет происходить изучение основ программирования в shell script. В заголовке раздела обычно будет указываться имя файла с примером, рассматриваемом в этом разделе.

В shell script, как и в любом языке программирования можно использовать переменные. Переменный shell script — это переменные среды окружения программы интерпретатора (bash). В первую очередь следует отметить, что переменные не типизированы. Все значения в переменных считаются строками. И только если переменная будет использоваться в математическом выражении, будет происходить проверка типа переменной.

В shell script нет такого понятия как «область видимости» переменных. Переменные доступны в любом месте кода, в том числе и в функциях. Если переменная будет определена в функции, она все равно будет глобальной переменной и к ней можно будет обращаться из любого места программы. В bash версии 2 появилась возможность определять локальные переменные при помощи оператора local. Но эта возможность не является стандартной для других разновидностей Bourne shell.

Если обратиться к неопределенной переменной, интерпретатор не будет выдавать сообщение об ошибке. Просто будет подставлена пустая строка.

Ниже приводится содержимое файла sample01.

```
#!/bin/bash
clear
CAR="porsh the best"
echo "CAR:"
echo CAR
echo '$CAR: '
echo $CAR
```

В строке 1 указывается тип интерпретатора, который будет запущен для выполнения этой программы.

Для определения переменной необходимо написать имя переменной (большие и маленькие буквы отличаются), затем символ «равно» и значение переменной (строка 3). Если в значении переменной встречаются пробелы, значение необходимо поместить в двойные или одинарные кавычки, например так:

```
CAR="porsh the best"
```

Для получения значения переменной, перед ее именем следует писать символ $(строка 7), например:

```
echo $CAR
```

В строке 4, на экран будет выведено — CAR:

В строке 5, на экран будет выведено — CAR

В строке 6 применяются одинарные кавычки. В shell script они имеют особое значение — все специальные символы, находящиеся внутри одинарных кавычек экранируются. То есть, символ $ не будет интерпретироваться как специальный символ и в результате на экран будет выведено $CAR:

Массивы переменных

Три способа определения массивов:
MASS[0]=value
MASS=([0]=value1 [5]=value2)
MASS=(value1 value2)
Получение значения элемента массива:
${MASS[0]}
Получение значений всех элементов массива:
${MASS[*]} или ${MASS[@]}

В shell script можно использовать массивы переменных. В классическом bourne shell существует одно ограничение интерпретатора — в массиве не может быть больше, чем 1024 элемента. В bash это ограничение снято.

Ниже приводится содержимое файла sample02 с пронумерованными строками:

```
1  #!/bin/bash
2  #Массивы
3  #CAR[0]=porsh
4  #CAR[1]=bmw
5  #CAR[2]=mers
6  #CAR[3]=zaporozets
7  #CAR[10]=LADA
8  #CAR=([0]=porsh [1]=bmw [2]=mers [5]=zaporozets [10]=LADA)
9  CAR=(porsh bmw mers zaporozets LADA)
10  echo "********************************"
11  echo "CAR[0]=${CAR[0]}"
12  echo "CAR[1]=${CAR[1]}"
13  echo "CAR[2]=${CAR[2]}"
14  echo "CAR[3]=${CAR[3]}"
15  echo "CAR[4]=${CAR[4]}"
16  echo "********************************"
17  echo "ALL - ${CAR[*]}"
18  echo "Alternative All - ${CAR[@]}"
19  echo "********************************"
```

В bash существует несколько способов определения массива. В файле примера первые два способа закомментированы (символ # в начале строки).
В первом примере (строки с 3 по 7) определение элементов происходит путем указания имени массива. Затем в квадратных скобках указывается номер элемента в массиве и дальше ему присваивается значение как обычной переменной.
Например:

```
CAR[1]=bmw
```

Элементы массива определяются не по порядку: 0, 1, 2, 3, и 10. Если обратиться к несуществующему элементу массива, интерпретатор не будет выдавать ошибку. Другой способ определения массива описан в строке 8:

```
CAR=([0]=porsh [1]=bmw [2]=mers [5]=zaporozets [10]=LADA)
```

В этом примере сначала пишется имя массива, а затем внутри круглых скобок элементам массива присваиваются значения. Третий пример определения массива (строка 9) похож на предыдущий, но в нем не указываются номера элементов массива. Это значит, что значения будут присваиваться по порядку: сначала нулевому элементу, затем первому и т.д.

Например:
```
CAR=(porsh bmw mers zaporozets LADA)
```

Для получения значения элемента массива следует использовать следующую конструкцию:
`${CAR[index]}` , где index >=0.

Обратите внимание на фигурные кавычки. В случае массивов их использование обязательно. В примере есть две строки: 17 и 18, в которых выводится все содержимое массива. Для этого, вместо номера элемента массива необходимо указать либо символ @, либо *.
Примеры:
```
${CAR[*]}
${CAR[@]}
```

Переменные окружения

Экспортирование переменных.
    PERM=value
    expot PERM
Область видимости переменных окружения.
Просмотр переменных окружения.
    set, env и export
Удаление переменных.
Встроенные переменные shell.

Программа, написанная на языке shell script, имеет доступ к переменным окружения, может изменять их значение, а так же определять новые переменные окружения. Ниже приводится содержимое файла sample03.

```
#! /bin/bash
clear
#PATH=$PATH:~/bin; export PATH
#export PATH=$PATH:~/bin
echo "Текущая директория = $PWD"
echo "Это UID = $UID"
echo "Текущий уровень исполнения = $SHLVL"
echo "Случайное число = $RANDOM"
echo "Уникальный идентификатор = $(cat
/proc/sys/kernel/random/uuid)"
echo "Текущая дата = `date`"
echo "Домашний каталог = $HOME"
echo "Регистрационное имя пользователя = $LOGNAME"
echo "Периодичность проверки почтового ящика = $MAILCHECK"
echo "Пути для поиска программ = $PATH"
echo "Внешний вид приглашения командной строки = $PS1"
echo "Внешний вид доп/ приглашения командной строки = $PS2"
echo "Текущий интерпретатор = $SHELL"
echo "Тип терминала = $TERM"
```

В классическом варианте Bourne shell для создания новой переменной окружения сначала создается переменная оболочки, а затем она экспортируется. Для экспорта переменной используется оператор export.

```
PATH=$PATH:~/bin
export PATH
```

В bash операции определения и экспортирования переменной могут происходить одновременно:

```
export PATH=$PATH:~/bin
```

Переменные окружения будут доступны в текущем процессе, а также во всех порожденных этой программой процессах. В других процессах системы эти переменные не будут видны.
Для просмотра всех переменных окружения и функций можно воспользоваться командой set. Программа env покажет только переменные, а export — только переменные помеченные как экспортированные. Удаление переменных происходит при помощи оператора unset. Например: `unset CAR`
В shell имеется большое количество встроенных переменных. В таблице перечислены только некоторые из них. Полное описание всех встроенных переменных можно найти в

справочном руководстве интерпретатора.

Переменная	Описание
HOME	Домашняя директория пользователя.
LOGNAME	Регистрационное имя пользователя.
MAILCHECK	Количество секунд, через которое будет происходить проверка наличия новых писем в почтовом ящике пользователя.
PATH	Содержит список директорий, разделенных двоеточием, в которых интерпретатор будет искать программу, если пользователь при запуске последней явно не указал путь к ней.
PS1	Внешний вид приглашения командной строки.
PS2	Внешний вид дополнительного приглашения командной строки.
SHELL	Содержит интерпретатор по умолчанию текущего пользователя.
TERM	Определяет тип терминала.
PWD	Содержит текущую директорию.
UID	UID пользователя, выполняющего программу.
SHLVL	Эта переменная увеличивается на 1, при следующем запуске shell. Учитываются только порожденные процессы.
RANDOM	При каждом чтении из переменной пользователь получает псевдослучайное число.

Взаимодействие с пользователем

Оператор read.
read [переменная ...]

Для того, чтобы программа на shell script могла получить данные, вводимые пользователем с клавиатуры, используется специальный оператор read.
Например: `read` [переменная ...]

При выполнении оператора read, на экране терминала появится курсор и пользователю дается возможность ввести данные. Ввод завершается нажатием на кнопку Enter. Ниже приводится содержимое файла sample04 с пронумерованными строками.

```
1   #!/bin/bash
2   #REPLY test
3   #echo "Write a car name and press \"Enter\" :"
4   echo -n 'Write a car name and press "Enter" :'
5   read
6   echo "Вы выбрали - $REPLY"
```

Обычно перед применением оператора read на экран выводится вопрос. Делается это при помощи программы echo. Если программе echo не указать опцию -n, она автоматически добавляет символ перевода строки после вывода данных. Поэтому строка 3 в примере закомментирована.
В 4-й строке приглашение выводиться на экран. Выводимая строка взята в одинарные кавычки, которые используются для того, чтобы отменить специальное значение символов двойные кавычки, окружающие слово Enter.
Строка 5. Если оператор read вызывать без указания переменной, он все данные, введенные пользователем, поместит в переменную по умолчанию — REPLAY, а в 6-й строке выводится содержимое этой переменной.

Подстановочные переменные

В подстановке используется то, что программа будет выводить на стандартный вывод. Два варианта записи подстановки: `program` $(program)

В shell script встроено очень мощное средство — подстановка данных, выводимых программой на стандартный вывод.
Для того, чтобы воспользоваться подстановкой необходимо взять программу в обратные одинарные кавычки. Например так:

```
`date`
```

или в круглые скобки со знаком $:

```
$(date)
```

В том месте кода, где используется подстановка, будет подставляться то, что программа вывела бы на стандартный вывод. Это значение динамическое, то есть, подставляются данные на момент выполнения скрипта. Ниже приводится содержимое файла sample05 пронумерованными строками.

```
1     #!/bin/bash
2     # Примеры подстановки
3     clear
4     echo "********************************************************************"
5     DATE=`date`
6     echo "Текущая дата = $DATE"
7     echo "********************************************************************"
8     USERS=`who | wc -l`
9     echo "Пользователей в системе = $USERS"
10    echo "********************************************************************"
11    UP=$(date; uptime)
12    echo "Текущие дата и uptime = $UP"
13    echo "********************************************************************"
14    MAXTHREADS=`cat /proc/sys/kernel/threads-max`
15    echo "Максмально возможное число потоков в системе = $MAXTHREADS"
16    echo "********************************************************************"
```

В строке 5 переменной DATE присваивается то, что программа date вывела бы на стандартный вывод. В 6-й строке выводится содержимое этой переменной.
В подстановке можно использовать конвейер команд (строка 8). В результате будет использоваться то, что программа wc вывела бы на стандартный вывод.
В строке 11 в подстановке выполняются сразу две программы. В результате переменной UP будет присвоено то, что программы date и uptime вывели бы на
стандартный вывод.

Арифметические выражения

В shell script используется ограниченный набор арифметических операций:
+ - * / и круглые скобки.
Для подстановки значения арифметических выражений используется:
$((выражение))
Если значение переменной, используемой в арифметическом выражении, не является целым числом, то её значение считается равным 0.

Язык, встроенный в оболочку, в основном предназначен для операций с объектами файловой системы. Поэтому, хотя в нем и есть возможность использования арифметических выражений, можно пользоваться только целочисленной арифметикой и минимальным набором арифметических операций.
В арифметических выражениях можно использовать операторы: +, -, *, / и круглые скобки. Так же можно использовать унарные операторы: ++ и --.
Для подстановки значения арифметического выражения его необходимо поместить в две круглые скобки, начинающиеся символом $.
Например: `$(( 2*2 ))`

В арифметическом выражении можно использовать переменные оболочки и окружения. При этом происходит проверка наличия в них целого числа. Если переменная не содержит целое число, ее значение в выражении принимается равным нулю. Ниже приводится содержимое файла sample06 с пронумерованными строками.

```
1   #!/bin/bash
2   # Подстановка арифметических выражений
3   PERM=2
4   echo "2*2=$(( 2*$PERM ))"
5   echo "((2*3+5)-4)/2=$(( ((2*3+5)-4)/2 ))"
```

Если в примере переменной PERM присвоить строку, например, test, то в результате программа echo (строка 4) выведет на экран 0.

Условный оператор if

```
if условие
then
    список операторов
[else
    список операторов ]
[elif условие
    список операторов ]
fi
```

В операторе if в качестве условия проверяется число — код возврата программы. Если программа была выполнена успешно — она возвращает ноль.
Если во время выполнения программы возникли ошибки, она возвращает число, отличное от нуля. Таким образом, в shell script истиной считается ноль. Все что не ноль — это ложь.
Оператор if всегда должен завершаться оператором fi. Если проверяемое условие истина, тогда выполняется набор операторов, находящихся между ключевыми словами then и fi.
Используя оператор else, можно определить набор команд, которые будут выполняться в случае ложного значения. Ниже приводится содержимое файла sample07 с пронумерованными строками.

```
1   #!/bin/bash
2   # Пример if then else
3   if rm test 2> /dev/null
4   then
5        echo "Deleted"
6   else
7        echo "Not deleted"
8   fi
```

В строке 3 проверяется код возврата программы rm. Тут указываются все опции, которые будут переданы программе rm при ее запуске. Если программа смогла удалить файл test, ее код возврата будет равен нулю. Если по какой-то причине программа не сможет удалить файл, она вернет код возврата отличный от нуля.
При этом сообщение об ошибке выводиться не будет, т.к. стандартный вывод ошибки был перенаправлен в /dev/null. При коде возврата 0 будет выполнена строка 5. При коде возврата отличном от нуля — 7.

Если в качестве условия оператора if используется выражение, помещенное в квадратные кавычки, для разрешения этого условия будет вызвана программа test. If будет проверять код возврата программы test.
Программа в файле sample08 выполняет те же действия, что и в предыдущем примере, но для проверки условия существования файла вызывается программа test (строка 3).

```
1   #!/bin/bash
2   # Пример if then else с использованием test
3   if [ -w $HOME/bin -a -f $HOME/bin/test ]
4   then
5        rm $HOME/bin/test
6        echo "test deleted"
7   else
8        echo "test not deleted"
9   fi
```

Проверка условий при помощи программы test

test [опции] условия ...
Программа test предназначена для проверки следующих типов условий:
- сравнение различных значений,
- проверка типов и наличия файлов,
- проверка логических условий.

Программа может проверить два типа логических условий И (AND) и ИЛИ (OR).

- Выражение1 -a Выражение2 — возвращает истину, если истинно и Выражение1 и Выражение2.
- Выражение1 -o Выражение2 — возвращает истину, если истинно или Выражение1 или Выражение2.
- Оператор ! инвертирует значение логического выражения.
- Сравнение чисел происходит при помощи следующих операторов:
  - число1 -eq число2 — истина, если числа равны.
  - число1 -ne число2 — истина, если числа не равны.
  - число1 -gt число2 — истина, если первое число больше второго.
  - число1 -ge число2 — истина, если первое число больше или равно второму.
  - число1 -lt число2 — истина, если первое число меньше второго.
  - число1 -le число2 — истина, если первое число меньше или равно второму.
- Сравнение строк:
  - -n строка — истина, если строка имеет не нулевую длину.
  - -z строка — истина, если строка имеет нулевую длину
  - строка1 = строка2 — истина, если строка1 идентична строке2.
- Проверка существования и типов файлов:
  - -e /путь/к/файлу — истина, если файл существует.
  - -f /путь/к/файлу — истина, если файл существует и является обыкновенным файлом.
  - -d /путь/к/файлу — истина, если файл существует и является директорией.
  - -L /путь/к/файлу — истина, если файл существует и является символьной ссылкой.
  - -r /путь/к/файлу — истина, если файл существует и доступен для чтения.
  - -w /путь/к/файлу — истина, если файл существует и доступен на запись.
  - -x /путь/к/файлу — истина, если файл существует и доступен на выполнение.
  - -s /путь/к/файлу — истина, если файл существует и имеет не нулевую длину.

Использование встроенных операторов && и ||

&& — логическое И.
|| — логическое ИЛИ.

В shell script есть два оператора, предназначенные для проверки условий логическое И — &&, и логическое ИЛИ — ||.
Наиболее часто их применяют, когда необходимо проверить условие и, если оно истинно, выполнить одну команду или наоборот — не выполнять.
Например:

```
[ -f file ] && rm file
```

Поскольку проверяется логическое И, необходимо чтобы оба условия были истиной. Поэтому, если первое условие истина, то будет проверяться второе условие, то есть будет вызвана программа rm.
Например:

```
[ -f file ] || touch file
```

В этом примере проверяется наличие файла file. Если его не существует (первое условие ЛОЖЬ), вызывается программа touch, которая его создает. Для того что бы получилась ИСТИНА, хотя бы одно из условий должно возвратить значение ИСТИНА. Поэтому, если файл существует (ИСТИНА), программа touch не будет вызываться, так как нет необходимости в проверки второго условия. Если файл не существует (ЛОЖЬ), необходимо проверить второе условие — будет выполнена программа touch.

Оператор case

```
case строка in
     шаблон)
          список операторов
               ;;
     [ шаблон)
          список операторов
               [;;] ]
esac
```

Оператор case поочередно сравнивает строку с шаблонами. Если шаблон совпадает, то выполняется группа операторов, находящихся между шаблоном и специальными символами ;;. После выполнения всех строк управление передается операторам, находящимся за ключевым словом esac.

Оператор case всегда завершается ключевым словом esac. Ниже приводится содержимое файла sample09 с пронумерованными строками.

```
1    #!/bin/bash
2    # Пример case esac
3    case $TERM in
4         *term)
5              echo "May be xterm?!"
6              ;;
7         unknown|vt[0-9]*)
8              echo "May be vt100 ?"
9              ;;
10         linux)
11              echo " This is a LINUX terminal!!!"
12              ;;
13         *)
14              echo "I don't know a this terminal :("
15   esac
16   exit 0
```

В строке 3 оператору case для проверки передается строка, находящаяся в переменной TERM. В строке 4 происходит сравнение с шаблоном *term). При написании шаблона можно использовать символы подстановки, такие же, как и в именах файлов.
В данном шаблоне имеется в виду любая строка, заканчивающаяся на term. Если шаблон сработает, то будет выполнена команда в строке 5. Если шаблон не сработает, то произойдет проверка следующего шаблона.
В 7-й строке показано применение условного оператора ИЛИ — символ |. Таким образом проверяется, соответствует ли строка слову unknown или начи нается ли она на vt и цифру. Если шаблон срабатывает, то выполняется команда в строке 8. Если не срабатывает, то проверяется следующий шаблон.
В строке 10 проверяется, соответствует ли строка слову linux. Если соответствует, то выполняется команда в строке 11. Если нет, то проверяется следующий шаблон.
Если ни один из перечисленных выше шаблонов не сработал (строка 13), можно в качестве шаблона использовать символ *, что аналогично применению ключевого слова default языка программирования C.

Ниже приводится код из файла sample10, в котором демонстрируется еще одна возможность применения оператора case для разбора ответа пользователя.

```
1   #!/bin/bash
2   # Пример case esac
3   echo -n "Please enter [Y|yes] : "
4   read YN
5   case $YN in
6        [yY]|[yY][eE][sS])
7            echo "You enter $YN"
8            ;;
9        *)
10            echo "You don't enter [Y|yes]"
11   esac
12   exit 0
```

В этом примере пользователя просят ввести Y или yes (строка 3). Ответ пользователя помещается в переменную YN (строка 4). При помощи оператора case проверяется, что он ввел. Поскольку регистр букв нас не интересует, шаблон на строке 6 проверяет различные варианты написания слова. Если было введено то значение, которое просили, выполняется команда на строке 7. Если было введено любое другое значение, срабатывает шаблон по умолчанию (строка 9) и выводится сообщение об ошибке (строка 10).

Оператор for

```
for переменная [ in список ]
do
   список операторов
done
```

При каждой итерации в операторе for, переменной присваивается следующее значение из списка и выполняются все операторы, находящиеся между do и done.
Оператор работает до тех пор, пока не будет обработан весь список или в теле цикла не встретится оператор break. Ниже приведен код из файла sample11, в котором показан пример простейшего цикла for.

```
1   #!/bin/bash
2   # Пример for
3   for I in 1 2 3 4 5 6 7 8 9 10
4   do
5         echo "--> $I <--"
6   done
7   exit 0
```

В этом примере переменной I подставляются значения из списка: 1, 2, 3, 4, 5, 6, 7, 8, 9 и 10. В результате на экран будут выведены 10 строк. Элементы в списке могут разделяться символами пробела или табуляции. Если список не помещается на одну строку, его можно продолжить на следующей, но перед тем как нажать на Enter, поставьте символ \ для экранирования значения символа перевода строки.
Например:

```
    for I in list1 list2 list3 \
             list5 list6
```

В цикле for в качестве списка можно указывать шаблон файловой системы. В этом случае в каждой итерации, переменной будет присваиваться путь к файлу, удовлетворяющий шаблону. Использование этого механизма показано в sample12.

```
 1   #!/bin/bash
 2   # Пример for
 3   # Создание html файлов
 4   #for FILES in `ls ~/.bash_*`
 5   for FILES in ~/.bash_*
 6   do
 7     echo "<HTML>" > ${FILES}.html
 8     echo "<HEAD><TITLE>${FILES}</TITLE></HEAD>" >> ${FILES}.html
 9     echo "<BODY><PRE>" >> ${FILES}.html
10     cat ${FILES} >> ${FILES}.html
11     echo "</PRE></BODY></HTML>" >> ${FILES}.html
12     chmod a+r ${FILES}.html
13   done
14   exit 0
```

В строке 5 в качестве списка используется шаблон ~/.bash_*. Это означает, что переменной FILES будет присваиваться файл, находящийся в домашней директории пользователя, имя которого начинается с .bash_. Обычно этому условию соответствуют файлы: .bash_profile, .bash_logout и .bash_history. То есть, будет три итерации, в каждой из которых переменная FILES будет иметь одно из перечисленных значений.

В строках с 9-й по 11-ю создается файл с таким же именем как и у исходного, но с расширением .html. В создаваемый файл помещается содержимое исходного файла.
После выполнения этого скрипта в домашней директории пользователя должны появиться файлы с расширением .html.

Получение данных из внешних файлов

```
for I in `cat ./sample13-data`
```

Иногда возникает необходимость помещать конфигурационные данные программы shell script во внешние файлы. Существует несколько способов получения данных из внешних файлов. Один из них показан в sample13. Другие будут рассматриваться в следующих примерах.
Предположим, что в скрипте необходимо выполнить много одинаковых команд, в которых изменяется содержимое только одного параметра. Например, необходимо в правилах firewall описать несколько IP адресов машин, с которых можно получать доступ к определенным ресурсам. Строка, описывающая эти разрешения, может выглядеть следующим образом:
`iptables -A FORWARD -s IP_ADDRESS -j ACCEPT`

Наша задача вместо IP_ADDRESS подставить значения IP адресов, находящихся во внешнем файле. В качестве такого файла возьмем sample13-data:
1 192.168.0.1
2 192.168.0.2
3 #192.168.0.3
4 192.168.0.4 # not pay
5 192.168.0.5

В этом файле приходится комментировать строку или часть строки (строки 3 и 4). Как это принято в Linux, для комментариев используется символ #. В sample13 показано, как можно получить данные из файла:

```
1  #!/bin/bash
2  # Пример получения данных из внешнего файла
3  for I in `cat ./sample13-data`
4  do
5       echo "--> $I"
6   done
7   exit 0
```

Для этого применяется цикл for. В качестве списка берутся данные, которые будут выданы в результате подстановки: cat ./sample13-data. В итоге, на экран будут выведены не только IP адреса, но и все остальные значения, которые были в этом файле. Дело в том, что shell script — язык вспомогательный и в нем нет операторов работы со строками.
Для получения правильного результата необходимо сначала осуществить фильтрацию данных при помощи других программ. Один из возможных примеров показан в sample13-2:

```
1   #!/bin/bash
2   # Пример получения данных из внешнего файла
3   for I in `cat ./sample13-data | cut -f1 -d ' ' | sed -e '/#/ d'`
4   do
5        echo "--> $I"
6   done
7   exit 0
```

В нем сначала отбираются первые поля файла, а затем удаляются строки, содержащие символ #.

Оператор while

```
while условие
do
    набор операторов
done
```

В цикле while выполняются строки, расположенные между do и done, до тех пор, пока условие истинно или пока не встретится оператор break или exit. Самый простой пример использования оператора while показан в файле sample14:

```
1  #!/bin/bash
2  # Primer while
3  X=1
4  while [ $X -lt 10 ]
5  do
6      echo "--> $X <--"
7      X=$(( $X+1 ))
8  done
9  exit 0
```

Сначала переменной X присваивается значение 1 (строка 3). Затем проверяется: X меньше 10? Если условие истинно, то выполняются строки 6 и 7. В 7-й строке значение X увеличивается на 1 и снова проверяется условие. В результате работы скрипта на экран будет выведено девять строк.

Оператор select

```
select переменная in список
do
     набор операторов
done
```

Оператор select выводит пронумерованный список на стандартный вывод и строку приглашения, в которой пользователь должен ввести номер элемента списка и нажать Enter. Затем значение выбранного элемента присваивается переменной и выполняются строки, расположенные между do и done. После этого опять выводится список, либо выводится приглашение на ввод номера элемента (зависит от версии shell).

Выход из цикла возможен путем явного вызова операторов break или exit. Пример использования оператора select находится в файле sample15:

```
 1   #!/bin/bash
 2   # Пример оператора select
 3   select FILE in ~/.b* QUIT
 4   do
 5        if [ -e $FILE ]
 6        then
 7             ls -l $FILE
 8        else
 9             break
10        fi
11   done
12   exit 0
```

Оператор select выводит пронумерованные значения списка. В список попадут все файлы, находящиеся в домашней директории пользователя, начинающиеся на .b и слово QUIT. Когда на экране появится приглашение ввода, следует ввести номер элемента и нажать Enter. После этого переменной FILE будет присвоено значение, соответствующее номеру и будут выполнены строки, находящиеся между do и done.

В строке 5 проверяется условие: «а существует ли такой файл?». Причем тип файла не имеет значения. Если он существует, то выполняется программа ls (строка 7). Затем снова появляется либо список, либо приглашение ввода.

Если файл не существует, а это условие может быть выполнено только если был выбран номер соответствующий слову QUIT, будет выполнен оператор break (строка 9) и программа выйдет из цикла select.

Оператор точка и функции

Оператор точка позволяет включать код,
находящийся в другом файле.
function sample
{
    Тело функции.
}
sample()
{
    Тело функции.
}

Оператор точка позволяет в текущем интерпретаторе выполнить код shell script, находящийся в другом файле.
По своей сути оператор точка похож на инструкцию include языка программирования C. Но при включении другого shell script файла ему можно передавать параметры командной строки, как обычной программе при ее выполнении.
Оператор точка очень часто используют для включения конфигурационных параметров, находящихся во внешних конфигурационных файлах. Например, существует файл со следующим содержанием:

```
PARAM=value
PARAM2=value2
```

Для того, чтобы воспользоваться этими параметрами в другом файле, необходимо подключить текст первого файла. Подключаемый файл может быть не исполняемым, например:

```
. file
echo $PARAM
echo $PARAM2
```

В примере sample16 показано использование оператора точка.

```
1   #!/bin/bash
2   # Пример использования функций и оператора "."
3   if [ ! -x $HOME/bin/sample16-2 ]; then
4        exit 1
5   fi
6   . $HOME/bin/sample16-2
7 select FILE in ~/.* QUIT
8 do
9        if [ -f $FILE ]
10         then
11              any
12         else
13              break
14         fi
15 done
16 exit 0
```

По своей функциональности данный пример выполняет те же действия, что и sample15, но в нем используются две интересные особенности. В строке 3 проверяется наличие дополнительного файла sample16-2, того файла, который будет подключаться в строке 6.

Если его нет, то работа скрипта завершается оператором exit, с кодом возврата 1, свидетельствующем об ошибке, возникшей при выполнении программы.
В sample16-2 показан пример определения функции:

```
1   #!/bin/bash
2   # Определение функции
3   # function
4   any()
5   {
6        ls -l $FILE
7   }
```

Функцию в shell script можно определить двумя способами: при помощи оператора function или после имени функции написать открывающую и закрывающую круглые скобки. Тело функции располагается между фигурными скобками.
Поскольку все переменные в shell script являются глобальными, внутри тела функции можно пользоваться переменными, определенными в любом месте shell script. А также определять новые переменные, которые можно использовать в любом месте программы. Единственным исключением являются позиционные переменные, о которых будет рассказано в следующем подразделе.
Если в функцию необходимо передать какие-либо параметры, они пишутся после имени функции так же, как аргументы командной строки. Получить эти параметры внутри функции можно при помощи позиционных переменных.

Специальные переменные

$0 $1 ... $9 — позиционные переменные.
$* и $@ — все параметры командной строки.
$# — количество параметров командной строки.
$? — код возврата программы.
$! — PID программы, запущенной в background режиме.
$$ — PID процесса shell.

В shell script встроены специальные переменные. В первую очередь — это позиционные переменные. При их помощи можно получить значение параметров, преданных при вызове программы в командной строке.
$0 $1 ... $9 — позиционные переменные.
$0 — имя программы.
$1 — первый параметр командной строки.
$2 — второй параметр командной строки и т.д.
$# — количество параметров командной строки, переданных программе[1].
$* и $@ — все параметры командной строки.

Есть существенное различие между переменными $* и $@. Предположим,
что программа была запущена следующим образом:

```
program -v -f «The file»
```

Если внутри этой программы попытаться получить список всех переменных, например, в цикле for, можно написать одну из перечисленных ниже строк:

```
for I in $* # четыре итерации.
for I in $@ # четыре итерации.
for I in «$@» # три итерации.
```

Как видно из примера, удовлетворительный результат можно получить только используя «$@», обязательно поместив переменную в двойные кавычки.

1Обратите внимание, что имя самой программы (значение $0) не учитывается.

Другие специальные переменные

$? — код возврата последней выполненной программы.
$! — PID последней программы, запущенной в background режиме.
$$ — PID процесса shell, исполняющего данный shell script.

Пример использования позиционных переменных показан в sample17:

```
1   #!/bin/bash
2   # Primer ispolzovanija $0
3   case $0 in
4        *listtar)
5             echo "List archive $1 ..."
6             TARGS="-tvf $1"
7             ;;
8        *maketar)
9             echo "Create archive $1.tar ..."
10             TARGS="-cvf $1.tar $1"
11             ;;
12        *) echo "Usage: listtar file | maketar dir"
13            exit 88
14   esac
15   tar $TARGS
16   exit 0
```

В Linux довольно часто программам необходимо передавать большое количество аргументов командной строки. Причем, все время программ передаются одни и теже аргументы командной строки. Чтобы для каждого конкретного случая не писать свой скрипт, запускающий программу, используют следующий способ запуска программ. На нее делают несколько символьных ссылок. И запускают программу, пользуясь этими ссылками.
Программа видит с каким именем ее запускают (по имени ссылки) и в зависимости от имени автоматически включается необходимые функции.
В shell script тоже можно проследить, с каким именем его запускают. Для этого используют позиционную переменную $0. В приведенном примере на строке 3 оператор case контролирует имя, с которым вызвана программа.
При помощи шаблонов отслеживается два возможных имени: listtar (строка 4) и maketar (строка 8). Поскольку мы не знаем какой путь будет указан при запуске программы (а в $0 попадает и имя и путь), в шаблоне перед именем стоит символ *.
Скрипт предназначен для просмотра содержимого и создания tar архивов.
Поэтому, в случае listtar, программе необходимо передать имя просматриваемого архива. Это имя попадает в переменную $1. На строке 5, на экран выводится сообщение «List archive $1 ...», где вместо $1 подставляется имя архива. А на строке 6 переменной TARGS присваиваются параметры, которые будут переданы программе tar. После этого управление переходит на строку 15.
В случае maketar, программа будет создавать архив. В качестве аргумента следует передать имя директории. Это имя попадет в переменную $1. На строке 9 на экран выводится сообщение об имени создаваемого архива. А на 10-й строке формируются параметры, которые будут переданы программе tar. После этого управление передается на 15-ю строку.
Если программу вызвать под каким-либо другим именем, то сработает ша-
блон по умолчанию (строка 12). При этом будет выведено сообщение об ошибке и мы выйдем из программы с кодом ошибки 88 (строка 13). Какой код ошибки возвращает программа решает программист.

Для того, чтобы скрипт заработал, необходимо создать две символьных
ссылки: listtar и maketar. И вызывать его только по этим именам. Если скрипт вызвать по его имени — sample17, будет выдано сообщение об ошибке.

Использование программы getopts

getopts параметры переменная

getopts — это встроенная в shell команда, позволяющая разобрать командную строку, передаваемую программе. Она понимает только параметры, написанные в стиле POSIX, то есть, параметр должен состоять из одной буквы, перед которой необходимо написать тире. Например: -v, -t, -f file и т.п. При определении параметра символ : означает, что параметр должен иметь дополнительное значение. Пример использования программы показан в sample18.

```
1 #!/bin/bash
2 while getopts f:o:v OPTION
3 do
4        case $OPTION in
5           f) echo "Option f - argument $OPTARG" ;;
6           o) echo "Option o - argument $OPTARG" ;;
7           v) echo "Option v - no argument" ;;
8           \?) echo "Usage: `basename $0` -f infile [-o outfile] [-v]"
9        esac
10 done
11 exit 0
```

Программу getopts вызывают как условие цикла while. Она пытается найти аргумент командной строки и, если такой аргумент есть, программа помещает его в переменную OPTION и возвращает истину.

После завершения итерации цикла while, снова вызывается программа getopts. Она ищет следующий аргумент командной строки и если находит, то все повторяется как и в предыдущем случае. Если аргумент не находиться, то программа возвращает ложь и мы выходим из цикла while.

Если у аргумента командной строки присутствует дополнительный параметр, getopts помещает этот параметр в специальную переменную OPTARG. Во время выполнения программы getopts могут возникать следующие ошибки:

•Указана не определенная в параметрах программы опция.

•У аргумента командной строки не указан обязательный дополнительный параметр.

Если при работе программы возникает ошибка, она выводит сообщение об ошибке на stderr, а в переменную OPTION помещает символ ?.

Оператор trap

trap команда список сигналов

Оператор trap позволяет переопределить стандартную реакцию программы на получаемые сигналы. В качестве первой опции необходимо указать команду, которая будет выполнена при получении сигнала. В качестве команды можно использовать функцию. Затем указать список сигналов, разделенных пробелами. Пример использования оператора trap находится в файле sample19:

```
1  #!/bin/bash
2  trap clean 2
3  clean() {
4       X=1
5       echo "Start formatting /dev/hda3:"
6       while [ $X -lt 10 ]
7       do
8            echo -n ".."
9            sleep 2
10            X=$(( $X+1 ))
11       done
12       echo "Done"
13       exit 0
14  }
15  while [ 0 ]
16  do
17  :
18  done
19  exit 0
```

На строке 2 вызывается оператор tarp, определяющий, что при получении программой сигнала 1 будет выполнена функция clean.
Основное тело программы представляет из себя бесконечный цикл while (строки 15-18), где условие всегда будет истина. И программа никогда не завершиться. Поскольку в цикле while между do и done необходимо написать какие-либо операторы, используется пустой оператор.
Функция clean выведет на экран сообщение «Start formatting /dev/hda3:», а затем с задержкой в две секунды (оператор sleep, строка 9) будет выводить две точки. В конце функции выполняется оператор exit (строка 13), который завершает работу программы.

Контрольные вопросы

1.Что произойдет, если в программе обратиться к переменной, которая не была определена на момент ее вызова?

2.Что рассматривается в качестве условия в операторах if и while?

3.При помощи какого оператора можно завершить выполнение скрипта?

4.Какой оператор позволяет включить код, находящийся в другом файле?

5.Какие символы позволяют экранировать значение специальных символов?

6.Какой код возврата должна возвращать программа при возникновении ошибки?