Сегодня поговорим о bash-скриптах. Это — сценарии командной строки, написанные для оболочки bash. Существуют и другие оболочки, например — zsh, tcsh, ksh, но мы сосредоточимся на bash. Этот материал предназначен для всех желающих, единственное условие — умение работать в командной строке Linux.

Сценарии командной строки — это наборы тех же самых команд, которые можно вводить с клавиатуры, собранные в файлы и объединённые некоей общей целью. При этом результаты работы команд могут представлять либо самостоятельную ценность, либо служить входными данными для других команд. Сценарии — это мощный способ автоматизации часто выполняемых действий.

Итак, если говорить о командной строке, она позволяет выполнить несколько команд за один раз, введя их через точку с запятой:

```
pwd ; whoami
```

На самом деле, если вы опробовали это в своём терминале, ваш первый bash-скрипт, в котором задействованы две команды, уже написан. Работает он так. Сначала команда pwd выводит на экран сведения о текущей рабочей директории, потом команда whoamiпоказывает данные о пользователе, под которым вы вошли в систему.

Используя подобный подход, вы можете совмещать сколько угодно команд в одной строке, ограничение — лишь в максимальном количестве аргументов, которое можно передать программе. Определить это ограничение можно с помощью такой команды:

```
getconf ARG_MAX
```

Командная строка — отличный инструмент, но команды в неё приходится вводить каждый раз, когда в них возникает необходимость. Что если записать набор команд в файл и просто вызывать этот файл для их выполнения? Собственно говоря, тот файл, о котором мы говорим, и называется сценарием командной строки.

Как устроены bash-скрипты

Создайте пустой файл с использованием команды `touch`. В его первой строке нужно указать, какую именно оболочку мы собираемся использовать. Нас интересует bash, поэтому первая строка файла будет такой:

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

В других строках этого файла символ решётки используется для обозначения комментариев, которые оболочка не обрабатывает. Однако, первая строка — это особый случай, здесь решётка, за которой следует восклицательный знак (эту последовательность называют шебанг) и путь к bash, указывают системе на то, что сценарий создан именно для bash.

Команды оболочки отделяются знаком перевода строки, комментарии выделяют знаком решётки. Вот как это выглядит:

```
#!/bin/bash
# This is a comment
pwd
whoami
```

Тут, так же, как и в командной строке, можно записывать команды в одной строке, разделяя точкой с запятой. Однако, если писать команды на разных строках, файл легче читать. В любом случае оболочка их обработает.

Установка разрешений для файла сценария

Сохраните файл, дав ему имя myscript, и работа по созданию bash-скрипта почти закончена. Сейчас осталось лишь сделать этот файл исполняемым, иначе, попытавшись его запустить, вы столкнётесь с ошибкой Permission denied.

Сделаем файл исполняемым:

```
chmod +x ./myscript
```

Теперь попытаемся его выполнить:

```
./myscript
```

После настройки разрешений всё работает как надо.

Вывод сообщений

Для вывода текста в консоль **Linux** применяется команда echo. Воспользуемся знанием этого факта и отредактируем наш скрипт, добавив пояснения к данным, которые выводят уже имеющиеся в нём команды:

```
#!/bin/bash
# наш комментарий здесь
echo Текущим каталогом является:"
pwd
echo "Пользователь, вошедший в систему:"
```

```
whoami
```

Теперь мы можем выводить поясняющие надписи, используя команду echo. Если вы не знаете, как отредактировать файл, пользуясь средствами **Linux**, или раньше не встречались с командой echo, взгляните на этот материал.

Использование переменных

Переменные позволяют хранить в файле сценария информацию, например — результаты работы команд для использования их другими командами.

Нет ничего плохого в исполнении отдельных команд без хранения результатов их работы, но возможности такого подхода весьма ограничены.

Существуют два типа переменных, которые можно использовать в bash-скриптах:

- Переменные среды
- Пользовательские переменные

Переменные среды

Иногда в командах оболочки нужно работать с некими системными данными. Вот, например, как вывести домашнюю директорию текущего пользователя:

```
#!/bin/bash
# отображение дома пользователя
echo "Домашняя страница для текущего пользователя -
это: $HOME"
```

Обратите внимание на то, что мы можем использовать системную переменную $HOME в двойных кавычках, это не помешает системе её распознать.

А что если надо вывести на экран значок доллара? Попробуем так:

```
echo "У меня в кармане $1"
```

Система обнаружит знак доллара в строке, ограниченной кавычками, и решит, что мы сослались на переменную. Скрипт попытается вывести на экран значение неопределённой переменной $1. Это не то, что нам нужно. Что делать?

В подобной ситуации поможет использование управляющего символа, обратной косой черты, перед знаком доллара:

```
echo "У меня в кармане \$1"
```

Теперь сценарий выведет именно то, что ожидается.

Пользовательские переменные

В дополнение к переменным среды, bash-скрипты позволяют задавать и использовать в сценарии собственные переменные. Подобные переменные хранят значение до тех пор, пока не завершится выполнение сценария.

Как и в случае с системными переменными, к пользовательским переменным можно обращаться, используя знак доллара:

```
#!/bin/bash
# testing variables
grade=5
person="Ваня"
echo "$person is a good boy, he is in grade $grade"
```

Подстановка команд

Одна из самых полезных возможностей bash-скриптов — это возможность извлекать информацию из вывода команд и назначать её переменным, что позволяет использовать эту информацию где угодно в файле сценария.

Сделать это можно двумя способами.

- С помощью значка обратного апострофа «`»
- С помощью конструкции $()

Используя первый подход, *проследите за тем, чтобы вместо обратного апострофа не ввести одиночную кавычку*. Команду нужно заключить в два таких значка:

```
mydir=`pwd`
```

При втором подходе то же самое записывают так:

```
mydir=$(pwd)
```

А скрипт, в итоге, может выглядеть так:

```
#!/bin/bash
mydir=$(pwd)
echo $mydir
```

В ходе его работы вывод команды pwdбудет сохранён в переменной mydir, содержимое которой, с помощью команды echo, попадёт в консоль.

Математические операции

Для выполнения математических операций в файле скрипта можно использовать конструкцию вида $((a+b)):

```
#!/bin/bash
var1=$(( 5 + 5 ))
echo $var1
var2=$(( $var1 * 2 ))
echo $var2
```

Управляющая конструкция if-then

В некоторых сценариях требуется управлять потоком исполнения команд. Например, если некое значение больше пяти, нужно выполнить одно действие, в противном случае — другое. Подобное применимо в очень многих ситуациях, и здесь нам поможет управляющая конструкция if-then. В наиболее простом виде она выглядит так:

if команда
then
команды
fi

А вот рабочий пример:

```
#!/bin/bash
if pwd
then
echo "It works"
fi
```

В данном случае, если выполнение команды pwdзавершится успешно, в консоль будет выведен текст «it works».

Воспользуемся имеющимися у нас знаниями и напишем более сложный сценарий. Скажем, надо найти некоего пользователя в /etc/passwd, и если найти его удалось, сообщить о том, что он существует.

```
#!/bin/bash
user=likegeeks
if grep $user /etc/passwd
then
```

```
echo "The user $user Exists"
fi
```

Здесь мы воспользовались командой grep для поиска пользователя в файле /etc/passwd. Если команда grep вам незнакома, её описание можно найти здесь.

В этом примере, если пользователь найден, скрипт выведет соответствующее сообщение. А если найти пользователя не удалось? В данном случае скрипт просто завершит выполнение, ничего нам не сообщив. Хотелось бы, чтобы он сказал нам и об этом, поэтому усовершенствуем код.

Управляющая конструкция if-then-else

Для того, чтобы программа смогла сообщить и о результатах успешного поиска, и о неудаче, воспользуемся конструкцией if-then-else. Вот как она устроена:

if команда
then
команды
else
команды//1?//1
fi

Если первая команда возвратит ноль, что означает её успешное выполнение, условие окажется истинным и выполнение не пойдёт по ветке else. В противном случае, если будет возвращено что-то, отличающееся от нуля, что будет означать неудачу, или ложный результат, будут выполнены команды, расположенные после else.

Напишем такой скрипт:

```
#!/bin/bash
user=anotherUser
if grep $user /etc/passwd
then
echo "Пользователь $user Существует"
else
echo "Пользователь $user не существует"
fi
```

Его исполнение пошло по ветке else.

Ну что же, продолжаем двигаться дальше и зададимся вопросом о более сложных условиях. Что если надо проверить не одно условие, а несколько?

Например, если нужный пользователь найден, надо вывести одно сообщение, если выполняется ещё какое-то условие — ещё одно сообщение, и так далее. В подобной ситуации нам помогут вложенные условия. Выглядит это так:

if команда1
then
команды
elif команда2
then
команды
fi

Если первая команда вернёт ноль, что говорит о её успешном выполнении, выполнятся команды в первом блоке then, иначе, если первое условие окажется ложным, и если вторая команда вернёт ноль, выполнится второй блок кода.

```
#!/bin/bash
user=anotherUser
if grep $user /etc/passwd
then
echo "The user $user Exists"
elif ls /home
then
echo "The user doesn’t exist but anyway there is a
directory under /home"
fi
```

В подобном скрипте можно, например, создавать нового пользователя с помощью команды useradd, если поиск не дал результатов, или делать ещё что-нибудь полезное.

Сравнение чисел

В скриптах можно сравнивать числовые значения. Ниже приведён список соответствующих команд.

Оператор	**Значение**
n1 -eq n2	Возвращает True, если n1 > n2.
n1 -ge n2	Возвращает True, если n1> или == n2.

Оператор	Значение
n1 -gt n2	Возвращает истинное значение, если n1 больше n2.
n1 -le n2	Возвращает истинное значение, если n1 меньше или равно n2.
n1 -lt n2	Возвращает истинное значение, если n1 меньше n2.
n1 -ne n2	Возвращает истинное значение, если n1не равно n2.

В качестве примера опробуем один из операторов сравнения. Обратите внимание на то, что выражение заключено в квадратные скобки.

```
#!/bin/bash
val1=6
if [ $val1 -gt 5 ]
then
echo "Тестовое значение $val1 больше 5"
else
echo "Тестовое значение $val1 не превышает 5"
fi
```

Значение переменной val1больше чем 5, в итоге выполняется ветвь thenоператора сравнения и в консоль выводится соответствующее сообщение.

Сравнение строк

В сценариях можно сравнивать и строковые значения. Операторы сравнения выглядят довольно просто, однако у операций сравнения строк есть определённые особенности, которых мы коснёмся ниже.

Список операторов.

Оператор	Значение

Оператор	Значение
str1 = str2	Возвращает True, если строки идентичны.
tr1 != str2	Возвращает True, если строки не идентичны.
str1 < str2	Возвращает True, если str1меньше, чем str2
str1 > str2	Возвращает True, если str1больше, чем str2
-n str1	Возвращает True, если длина str1больше нуля.
-z str1	Возвращает True, если длина str1равна нулю.

Вот пример сравнения строк в сценарии:

```
#!/bin/bash
user="zzz"
if [ $user = $USER ]
then
echo "Пользователь $user - это текущий вошедший в
систему пользователь"
fi
```

Вот одна особенность сравнения строк, о которой стоит упомянуть. А именно, операторы **'>'** и **'<'** необходимо экранировать с помощью обратной косой черты, иначе скрипт будет работать неправильно, хотя сообщений об ошибках и не появится. Скрипт интерпретирует знак **'>'** как команду перенаправления вывода.

Вот как работа с этими операторами выглядит в коде:

```
#!/bin/bash

val1=text
val2="another text"
```

```
if [ $val1 \> $val2 ]
then
echo "$val1 больше, чем $val2"
else
echo "$val1 меньше, чем $val2"
fi
```

Обратите внимание на то, что скрипт, хотя и выполняется, выдаёт предупреждение:

./myscript: line 5: [: too many arguments

Для того, чтобы избавиться от этого предупреждения, заключим $val2 в двойные кавычки:

```
#!/bin/bash

val1=text
val2="another text"

if [ $val1 \> "$val2" ]
then
echo "$val1 больше, чем $val2"
else
echo "$val1 меньше, чем $val2"
fi
```

Ещё одна особенность операторов **'>'** и **'<'** заключается в том, как они работают с символами в верхнем и нижнем регистрах. Для того, чтобы понять эту особенность, подготовим текстовый файл с таким содержимым:

Likegeeks
likegeeks

Сохраним его, дав имя myfile, после чего выполним в терминале такую команду:

```
sort myfile
```

Она отсортирует строки из файла так:

likegeeks
Likegeeks

Команда sort, по умолчанию, сортирует строки по возрастанию, то есть строчная буква в нашем примере меньше прописной. Теперь подготовим скрипт, который будет сравнивать те же строки:

#!/bin/bash

```
val1=Likegeeks
val2=likegeeks

if [ $val1 \> $val2 ]
then
echo "$val1 больше, чем $val2"
else
echo "$val1 меньше, чем $val2"
fi
```

Если его запустить, окажется, что всё наоборот — строчная буква теперь больше прописной.

В командах сравнения прописные буквы меньше строчных. Сравнение строк здесь выполняется путём сравнения ASCII-кодов символов, порядок сортировки, таким образом, зависит от кодов символов.

Команда sort, в свою очередь, использует порядок сортировки, заданный в настройках системного языка.

Пример Операции сравнения

```
#!/bin/bash

a=4
b=5

#  Здесь переменные "a" и "b" могут быть как целыми
числами, так и строками.
#  Здесь наблюдается некоторое размывание границ
#+ между целочисленными и строковыми переменными,
#+ поскольку переменные в Bash не имеют типов.

#  Bash выполняет целочисленные операции над теми
переменными,
#+ которые содержат только цифры
#  Будьте внимательны!

echo
```

```
if [ "$a" -ne "$b" ]
then
  echo "$a не равно $b"
  echo "(целочисленное сравнение)"
fi

echo

if [ "$a" != "$b" ]
then
  echo "$a не равно $b."
  echo "(сравнение строк)"
  #     "4"  != "5"
  # ASCII 52 != ASCII 53
fi

# Оба варианта, "-ne" и "!=", работают правильно.

echo

exit 0
```

Пример Проверка -- является ли строка *пустой*

```
#!/bin/bash
# str-test.sh: Проверка пустых строк и строк, не
заключенных в кавычки,

# Используется конструкция   if [ ... ]


# Если строка не инициализирована, то она не имеет
никакого определенного значения.
# Такое состояние называется "null" (пустая) (это не то
же самое, что ноль).

if [ -n $string1 ]    # $string1 не была объявлена или
инициализирована.
then
  echo "Строка \"string1\" не пустая."
else
  echo "Строка \"string1\" пустая."
fi
# Неверный результат.
# Выводится сообщение о том, что $string1 не пустая,
#+не смотря на то, что она не была инициализирована.
```

```
echo


# Попробуем еще раз.

if [ -n "$string1" ]  # На этот раз, переменная
$string1 заключена в кавычки.
then
  echo "Строка \"string1\" не пустая."
else
  echo "Строка \"string1\" пустая."
fi      # Внутри квадратных скобок заключайте строки в
кавычки!


echo


if [ $string1 ]       # Опустим оператор -n.
then
  echo "Строка \"string1\" не пустая."
else
  echo "Строка \"string1\" пустая."
fi
# Все работает прекрасно.
# Квадратные скобки -- [ ], без посторонней помощи
определяют, что строка пустая.
# Тем не менее, хорошим тоном считается заключать
строки в кавычки ("$string1").
#
# Как указывает Stephane Chazelas,
#    if [ $string 1 ]   один аргумент "]"
#    if [ "$string 1" ]  два аргумента, пустая
"$string1" и "]"



echo



string1=initialized
```

```
if [ $string1 ]       # Опять, попробуем строку без
ничего.
then
  echo "Строка \"string1\" не пустая."
else
  echo "Строка \"string1\" пустая."
fi
# И снова получим верный результат.
# И опять-таки, лучше поместить строку в кавычки
("$string1"), поскольку...


string1="a = b"

if [ $string1 ]       # И снова, попробуем строку без
ничего..
then
  echo "Строка \"string1\" не пустая."
else
  echo "Строка \"string1\" пустая."
fi
# Строка без кавычек дает неверный результат!

exit 0
# Спасибо Florian Wisser, за предупреждение.
```

Пример most

```
#!/bin/bash

#Просмотр gz-файлов с помощью утилиты 'most'

NOARGS=65
NOTFOUND=66
NOTGZIP=67

if [ $# -eq 0 ] # то же, что и:  if [ -z "$1" ]
# $1 должен существовать, но может быть пустым:  zmost
"" arg2 arg3
then
  echo "Порядок использования: `basename $0` filename"
>&2
  # Сообщение об ошибке на stderr.
  exit $NOARGS
  # Код возврата 65 (код ошибки).
fi
```

```
filename=$1

if [ ! -f "$filename" ]   # Кавычки необходимы на тот
случай, если имя файла содержит пробелы.
then
  echo "Файл $filename не найден!" >&2
  # Сообщение об ошибке на stderr.
  exit $NOTFOUND
fi

if [ ${filename##*.} != "gz" ]
# Квадратные скобки нужны для выполнения подстановки
значения переменной
then
  echo "Файл $1 не является gz-файлом!"
  exit $NOTGZIP
fi

zcat $1 | most

# Используется утилита 'most' (очень похожа на 'less').
# Последние версии 'most' могут просматривать сжатые
файлы.
# Можно вставить 'more' или 'less', если пожелаете.


exit $?   # Сценарий возвращает код возврата,
полученный по конвейеру.
# На самом деле команда "exit $?" не является
обязательной,
# так как работа скрипта завершится здесь в любом
случае
```

Проверки файлов

Пожалуй, нижеприведённые команды используются в bash-скриптах чаще всего. Они позволяют проверять различные условия, касающиеся файлов. Вот список этих команд.

Оператор	**Значение**
-d file	Проверяет, существует ли файл, и является ли он

Оператор	Значение
	директорией.
-e file	Проверяет, существует ли файл.
-f file	Проверяет, существует ли файл, и является ли он файлом.
-r file	Проверяет, существует ли файл, и доступен ли он для чтения.
-s file	Проверяет, существует ли файл, и не является ли он пустым.
-w file	Проверяет, существует ли файл, и доступен ли он для записи.
-x file	Проверяет, существует ли файл, и является ли он исполняемым.
file1 -nt file2	Проверяет, новее ли file1, чем file2.
file1 -ot file2	Проверяет, старше ли file1, чем file2.
-O file	Проверяет, существует ли файл, и является ли его владельцем текущий пользователь.
-G file	Проверяет, существует ли файл, и соответствует ли его идентификатор группы идентификатору группы текущего пользователя.

Эти команды, как впрочем, и многие другие рассмотренные, несложно запомнить. Их имена, являясь сокращениями от различных слов, прямо указывают на выполняемые ими проверки.

Опробуем одну из команд на практике:

```
#!/bin/bash

mydir=/home/zzz

if [ -d $mydir ]
then
    echo "Каталог $mydir существует"
    ls $mydir
else
    echo "Каталог $mydir не существует"
fi
```

Этот скрипт, для существующей директории, выведет её содержимое.