Лекция 8. Скрипты Bash

Цель лекции

- Понять основы технологии контейнеризации

Дополнительные материалы (ссылки, файлы)

1. Advanced Bash-Scripting Guide
2. Костромин В. Linux для пользователя
3. Регулярные выражения в Linux
4. Регулярные выражения

Термины

Bash — это командный интерпретатор, работающий, как правило, в интерактивном режиме в текстовом окне. Он также может читать команды из текстового файла, который называется скриптом. Как и большинство Unix-оболочек, он поддерживает автодополнение имён файлов и директорий, подстановку вывода результата команд, переменные, контроль за порядком выполнения, операторы ветвления и цикла.
Переменные окружения — это специальные переменные, определённые оболочкой и используемые программами во время выполнения. Они могут определяться системой и пользователем.
POSIX — набор стандартов, описывающих интерфейсы между операционной системой и программным обеспечением. Стандарт создан для обеспечения совместимости различных Unix-подобных операционных систем.
Регулярные выражения — инструмент для поиска текста по шаблону, обработки и изменения строк, который можно применять для следующих задач: поиск и замена текста в файле, проверка строки на соответствие шаблону и т. д.

План

1. Оболочка bash, язык программирования.
2. Команды в bash. Типы команд.
3. Код возврата. Связки команд && и ||.
4. Потоки ввода-вывода, перенаправление (>, <, >>, 2>).
5. Конвейер (pipeline).
6. Переменные и их классификация.
7. Простейший скрипт.

8. Методы запуска скрипта.
9. Однострочные скрипты.
10. Условия if, test.
11. Цикл for.
12. Цикл while.

Текст лекции

Оболочка bash, язык программирования

С самого начала курса мы используем оболочку bash для взаимодействия с системой. Мы разбирали приглашение оболочки, вводили команды и получали требуемый результат.
У каждого пользователя есть несколько файлов конфигурации, которые определяют поведение оболочки. Это файлы ~/.bashrc и ~/.profile.
Мы можем назвать такой способ работы с bash интерактивным. То есть вводим команду – получаем результат. Есть также второй способ использования bash – в качестве интерпретатора программного кода (скрипта) на bash. **Скриптом** обычно называется программный код, который находится в текстовом файле и исполняется интерпретатором. В нашем случае мы будем создавать bash-скрипты и исполнять их интерпретатором bash.
Самый простой вариант создания скрипта — написать несколько обычных команд (одна на каждой строке) в текстовый файл. Например:

```
mkdir test
cd test
touch testfile
```

Мы можем ввести эти команды последовательно в оболочке и они будут обработаны по одной. В случае использования скрипта, мы можем выполнить эти команды вместе. Кроме того, набор этих команд (скрипт) можно повторять сколько угодно раз. Таким образом, мы сокращаем рутинные операции. Например, мы можем написать скрипт, который добавляет пользователя в систему и проводит начальные настройки его профиля (копирует настройки, файлы, создаёт каталоги и т.д.)
Кроме того, мы посмотрим более сложные примеры применения bash как языка программирования. В bash есть переменные, циклы, условия и другие конструкции, позволяющие создавать сложные программы.
Для начала разберём, какие типы команд мы используем в bash и как их различать.

Команды в bash. Типы команд

Когда мы запускаем на исполнение команду, bash должен определить её тип и отправить на исполнение. За командой может стоять псевдоним, исполняемый файл, встроенная команда bash или функция.
Для разделения типов можно использовать команду **type**. Например:

type -a ls
type -a grep
type cd
type passwd
Для определения пути к исполняемому файлу можно воспользоваться which:
which rm
В нашей оболочке по умолчанию настроены псевдонимы. Найти их можно в конфиге ~/.bashrc. Вот некоторые из них:
alias ls='ls --color=auto'
alias ll='ls -al'
alias la='ls -A'
С помощью псевдонимов можно сокращать часто используемые команды с параметрами до нескольких символов. Если добавить новый псевдоним, он будет работать после запуска новой оболочки.
Некоторые команды являются встроенными в bash, например cd.
Теперь давайте познакомимся с интересной возможностью связки команд в зависимости от результата (кода возврата).

Код возврата. Связки команд && и ||

При исполнении любой команды мы получаем **код возврата (exit code)**. Вывести его можно командой echo $?.
Код возврата также можно назвать кодом ошибки. Если код равен 0, то ошибки не было и программа отработала штатно. Если код ответа больше нуля, то была ошибка. Интерпретация кодов ошибок может быть в документации к программе.
Используя коды возврата, мы можем создавать условные связки команд. Например, мы хотим создать директорию и файл в ней, если она создалась успешно. Для этого мы используем символы &&, получив логическую связку:
mkdir testdir && touch testdir/testfile
Вторая команда (touch) выполнится только в случае успешного выполнения первой команды.
Иногда необходимо обратное: выполнить команду, если первая завершилась с ошибкой, например можно попробовать просмотреть директорию, а в случае ошибки создать её, используем || в качестве связки:
ls -al /home/db/testdir || mkdir -p /home/db/testdir
В этом варианты вторая команда выполнится, если первая вернёт ошибку (код ответа больше нуля).
Кстати, вывод команд (также и ошибки) мы можем направить в файл вместо терминала. Давайте узнаем, как это работает.

Потоки ввода-вывода, перенаправление (>, <, >>, 2>)

Запуск любой программы порождает процесс в операционной системе. У этого процесса открывается как минимум 3 файловых дескриптора:

- STDIN (0) – стандартный поток ввода;
- STDOUT (1) – стандартный поток вывода;
- STDERR (2) – стандартный поток ошибок.

Номера этих потоков (указаны в скобках) мы будем использовать при перенаправлении в файлы.
По умолчанию эти потоки направляются в консоль (на вывод), или на клавиатуру (на ввод). Чтобы перенаправить их в файл мы используем символы угловых скобок, рассмотрим следующие примеры:

- program < file – перенаправление ввода из файла file;
- program > file – перенаправление вывода (STDOUT) в файл file (запись с начала файла);
- program >> file – перенаправление вывода (STDOUT) в файл file в режиме дополнения файла;
- program 2> file – перенаправление ошибок (STDERR) в файл file (запись с начала файла);
- program 2>> file – перенаправление ошибок (STDERR) в файл file в режиме дополнения файла;
- program > file 2>&1 – перенаправление вывода (STDOUT) и ошибок (STDERR) в файл file (запись с начала файла).

С помощью перенаправления ввода-вывода мы можем запускать скрипты или другие утилиты в фоновом режиме (например, через cron) и получать информацию об ошибках или сохранять вывод в файл. Если нам не нужен какой-то поток, мы можем перенаправить его в специальное устройство /dev/null (работает как черная дыра):
program 2> /dev/null
Потоки ввода-вывода можно перенаправлять не только в файлы, но и в другие процессы. Такой механизм называется конвейером (pipeline).

Конвейер (pipeline)

Как мы уже знаем, каждый процесс в системе имеет стандартные потоки ввода-вывода. С использованием этих потоков процессы могут обмениваться данными, образуя **конвейеры**. Первая команда в конве йере выводит исходные данные, они подаются на ввод второй команде, её вывод на ввод третьей и так далее.
Давайте рассмотрим простой конвейер:
ls -al | grep file
В этом примере вывод команды ls -al (который мы обычно видим в терминале) отправляется на ввод команды grep file. Команда **grep** является универсальным инструментом поиска и фильтрации данных. Она позволяет сопоставлять строчки

текста с шаблоном. Шаблон может быть простой строкой, например ‘text’, а может быть регулярным выражением, например:
grep '^test' testfile
Эта команда найдёт и покажет строчку, где есть test в начале строки.
ls -al | grep -P '\.[cs]+'
А эта команда покажет элементы каталога, в которых есть точка и далее буквы c или s. Тема регулярных выражений обширна и заслуживает отдельного изучения, здесь можно воспользоваться ссылкам из дополнительных материалов.
Количество команд в конвейере практически не ограничено, поэтому можно создавать сложные комбинации команд.
Например:
cat /var/log/syslog | grep 'mysql' | grep -v 'file' | wc -l
В этом конвейере мы фильтруем вывод файла syslog, ищем там строчки с mysql и убираем строки, в которых есть file. Далее считаем количество строк через wc.
На этом мы готовы переходить к использованию переменных в bash.

Переменные и их классификация. Ограничения на имена

Переменные необходимы для хранения информации. С ними можно выполнить два действия:

- установить значение переменной;
- прочитать значение переменной.

Переменные могут быть определены и вызваны в любом месте скрипта. Для вызова используется символ $ перед именем переменной. Например:

- a=123 — присвоение переменной a значения 123;
- echo $a — вызов значения переменной $a (на экран будет выведено число 123).

В качестве имени переменной могут быть использованы буквы или слова, написанные латиницей, а также сочетания букв и цифр. Значением переменной может быть любое слово или группа слов, цифра, а также команда. В bash нет строгих различий между типами переменных. С точки зрения командного интерпретатора любая переменная является строкой.

Переменные в bash можно разбить на три группы:

- переменные окружения;
- пользовательские переменные;
- специальные переменные.

Переменные окружения

Это специальные переменные, которые определяются оболочкой и используются программами в момент исполнения. Значения эти переменные получают в момент входа пользователя в систему. Пользовательские переменные окружения

прописываются в файлах .bashrc, .profile, которые расположены в домашнем каталоге пользователя.

Примеры переменных окружения:

- $PWD — текущий каталог.
- $ID — покажет имя текущего пользователя и группы, в которых он состоит.
- $PATH — покажет путь до исполняемых файлов.

Переменные можно переопределить в рамках текущей сессии. Например, PATH=$PATH:/opt/my_progs/bin — для сохранения всех путей до исполняемых файлов мы при переопределении переменной PATH присваиваем её предыдущее значение $PATH и добавляем, используя разделитель «:», новый путь: /opt/my_progs/bin. Без сохранения в файлы bashrc или .profile это переопределение будет действовать до конца сеанса пользователя или же всё время работы терминала.

Пользовательские переменные

Это переменные, которые определяет пользователь в момент написания скрипта. Например, присвоение строки hello world: a="hello world". Если переменная состоит из нескольких слов, строка берётся в кавычки. Если значение переменной — команда, то либо она берётся в обратные апострофы: a=`ls`, либо используется $(command), например, a=$(ls).

Специальные переменные

Это ряд переменных bash, отличных от переменных окружения, которые предопределены операционной системой. Например:

- позиционные параметры: $0 $1..$9, где $0 — это имя скрипта, $1..$9 — аргументы, которые мы можем передать скрипту;
- $? — статус выполнения предыдущей команды или скрипта (код возврата), в своём роде логическая переменная. Если возвращает 0 — скрипт завершился успешно, любое другое значение — ошибка выполнения.

Чтобы выполнять арифметические операции с переменными, в bash существует специальная запись: $(( )). В скобках указываются числа и операции над ними. Например:

```
a=$((5+5))
echo $a
```

Запись выведет на экран 10.

Итак, мы изучили важные компоненты для создания скриптов, поэтому переходим к их созданию.

Простейший скрипт

Для создания простого скрипта стоит разобрать понятие **shebang** (шебанг). Это первая строчка файла, в которой мы указываем путь к интерпретатору. Дело в том, что любой скрипт это текстовый файл. У скрипта может не быть никакого расширения. В таком случае обязательно указать интерпретатор, чтобы система могла выбрать корректный способ запуска скрипта. Например в шебанге может быть прописаны такие интерпретаторы:
#!/usr/bin/perl
#!/usr/bin/python
#!/usr/bin/php
#!/bin/bash
#!/bin/sh
Важно помнить, что шебанг должен располагаться на первой строке скрипта без пробелов.
Итак, напишем первый полноценный скрипт:
cat > testscript
#!/bin/bash

directory=$1
hidden_count=$(ls -A $directory | grep '^\.' | wc -l)

echo “Hidden files in $directory found: $hidden_count”

Разберём, что здесь происходит. Сначала мы указали шебанг, для определения интерпретатора.
Далее мы определили переменную $directory, в которой будет храниться первый параметр скрипта. Следующим действием мы записали в переменную $hidden_count количество скрытых файлов в указанной директории. Для этого мы использовали присвоение результата команды, в которой есть конвейер.
Наконец, мы вывели количество найденных скрытых файлов с помощью команды echo.
Теперь нужно разобраться с тем, как можно запустить этот скрипт на исполнение.

Методы запуска скрипта

Для начала попробуем запустить скрипт, находясь в директории с файлом простой командой:
testscript
В результате получим ошибку: command not found. То есть, такая команда не найдена. Действительно, в переменной $PATH нет нашего текущего каталога. Система не будет запускать скрипт, потому что ищет его в других местах.

Обойти эту проблему можно либо дополнив переменную $PATH текущим каталогом, либо указав на текущую директорию явно:
./testscript
/home/db/test/testscript
Теперь мы получим другую ошибку: Permission denied. Исправить это несложно:
chmod +x testscript
Теперь наш скрипт запускается и работает, как положено.
Если мы не можем или не хотим ставить бит исполнения на файл, то можно исполнить скрипт с помощью команды:
bash testscript
Кстати, необязательно писать классический скрипт для того, чтобы автоматизировать задачи. Можно использовать однострочные скрипты.

Однострочные скрипты

Как мы говорили ранее, в обычном скрипте мы разделяем команды символом новой строки.
Часто нам нужно выполнить всего несколько команд подряд и создавать файл со скриптом нецелесообразно. В этом случае мы можем использовать разделитель “;” и получать однострочный скрипт:
apt update; apt upgrade; echo “Upgrade complete!”
Таким образом мы можем перевести любой скрипт в однострочный и выполнить его как одну команду.
Теперь давайте усложним наш скрипт и посмотрим на условия.

Условия if, test

Условные операторы предоставляют возможность решить, продолжать дальнейшие действия или нет. Решение принимается на основе вычисления выражения. В bash, как и в других языках программирования, основной оператор выбора — конструкция if/then/else/fi (если/тогда/иначе/конец_блока).
Конструкция выглядит следующим образом:

```
if [ выражение ]
  then
    Действия, если выражение истинно
  else
    Действия в противоположном случае
fi
```

Оператор fi обязателен, им закрывается проверка условия. Оператор else не обязателен, поскольку действий в противном условию случае может не быть.
[ ] — аналог команды test. Это команда, которая проверяет типы файлов и сравнивает значения. Подробности можно прочитать на странице справочного руководства man test.

[[ ]] — аналог оператора [ ], но с более широкими возможностями. К примеру, у него лучшая поддержка регулярных выражений.
(( )) — используется для арифметических операций.
Используя условный оператор if, мы можем осуществить операции проверки и сравнений.

Операции проверки файлов (наиболее используемые):

- -e возвращает true (истина), если файл существует (exists);
- -d возвращает true (истина), если каталог существует (directory).

Например:

```
if [ -e file_name ]
  then
     echo "true"
 else
    echo "false"
fi
```

Вернётся true, если файл существует, и false, если такого файла нет. Аналогично выполняется проверка существования каталога.

Операции сравнения строк (наиболее используемые):

- = или == возвращает true (истина), если строки равны;
- != возвращает true (истина), если строки не равны;
- -z возвращает true (истина), если строка пуста;
- -n возвращает true (истина), если строка не пуста.

Простой пример проверки передачи параметра скрипту:

```
#!/bin/bash
a=$1 #присваиваем переменной a значение переменной подстановки $1, используем как параметр
#проверяем, что параметр задан
if [ -z $a ]
  then
     echo "Error" #если строка пустая, сообщаем об ошибке
     exit # завершаем скрипт
 else
    echo $a # в противном случае выводим на экран значение параметра
fi
```

Операции сравнения целых чисел (наиболее используемые)

- -eq возвращает true (истина), если числа равны (equals);
- -ne возвращает true (истина), если числа не равны (not equal).

Например:

```
#!/bin/bash
a=$1
b=$2
if [ $a -eq $b ]
```

```
 then
     echo "true"
else
     echo "false"
fi
```

Скрипт сравнивает два числа, переданные в качестве параметров. В случае равенства вернёт true (истина), в противном случае вернёт false (ложь).
Для выполнения действий над список элементов нам потребуются циклы.

Цикл for

Цикл — последовательность, которая позволяет выполнить определённый участок кода заданное количество раз. Существует несколько типов циклов. Мы рассмотрим два наиболее часто используемых.
Цикл for позволяет организовать перебор последовательности значений. Структура цикла:

```
for имя_переменной in значения
do
  тело_цикла
done
```

Здесь имя_переменной — переменная, которая будет получать значения из массива «значения». В качестве такого массива может быть задана последовательность чисел, какой-то набор слов, результат работы команды. Тело_цикла — команды, которые будут обрабатывать переменную.
Пример: for i in $(ls); do echo $i;done — здесь переменная i получает значения из работы команды ls, и команда echo выводит это значение на терминал.

Цикл while

Цикл **while** выполняется до тех пор, пока условие истинно. Структура цикла:

```
while [ условие ]
do
  Тело_цикла
done
```

Здесь в качестве [условие] осуществляются операции сравнения и проверки, аналогичные условному оператору if. Тело_цикла — команды, которые будут выполняться до тех пор, пока условие возвращает true (истина).
Пример бесконечного цикла: while [true];do echo "true"; done — так как условие всегда «истина», то на экран терминала всегда будет выводиться слово true.

Итоги занятия

- Познакомились с оболочкой bash.

- Узнали, какие бывают команды.
- Научились работать с потоками ввода-вывода.
- Узнали, как использовать конвейер.
- Познакомились с переменными в bash.
- Создали простой скрипт.
- Научились работать с условиями и циклами.