Unix: написание shell скриптов

Эридан Доморацкий

26 октября 2023 г.

Скрипт?

Скрипт (script, сценарий) – последовательность команд, записанных в одном файле. Частный случай программы. (неформально)

Мы будем говорить о Unix shell-скриптах, которые состоят из команд shell.

Команды shell

Пример команды:

```
ls a.txt b .. 2>/dev/null
```

1. `ls` — команда
2. `a.txt`, `b`, `..` — аргументы
3. `2>/dev/null` — перенаправление вывода из потока 2 (`stderr`, поток ошибок) в файл `/dev/null`

Команды shell

Поговорим о непосредственно командах.
Командой может быть:

- **Исполняемый файл**, находящийся в *правильном месте* в системе
  - Доступный через **переменную окружения** `PATH`: `ls`, `cat`, `man`
  - Используемый по относительному или абсолютному пути: `./a.out`, `~/test.sh`, `/opt/do_smth`
- **Функция**
- **Псевдоним** (или **alias**)
- **Встроенная команда**: `cd`, `alias`, `[[`

Команды shell: встроенные команды

Любой shell имеет некоторый набор **встроенных команд**, которые описаны в его документации. Например:

- `cd` — смена текущей директории
- `pwd` — вывод текущей директории
- `[[` — условное выражение
- `alias` — создание псевдонима
- `which` — определение источника команды (путь до файла, является ли алиасом или функцией)
- `exit` — завершение текущего скрипта
- `read` — чтение из `stdin`
- `bg`/`fg`/`jobs` — работа с задачами (jobs)
- и многие другие...

Команды shell: псевдонимы

Псевдонимы объявляются в Unix shell командой `alias`:

```
# определили новую команду
alias hello='echo Hello,'

# добавили аргументы "по-умолчанию"
alias ls='ls -A --color=always'

# переопределили переменную окружения
alias ghci='TERM=xterm ghci'
```

Алиасы вставляются на месте использования как есть, то есть команда `FOO=bar hello World` будет выполнена как `FOO=bar echo Hello, World`

Команды shell: функции

Функции определяются с помощью специального синтаксиса:

```
function do_quiet() {
    # выполнить все аргументы как команду
    # и подавить весь ввод и вывод
    "$@" >/dev/null 2>&1 </dev/null
}
```

В отличие от алиасов, функции можно считать полноценными скриптами. В том смысле, что также как и скрипты, функции могут оперировать своими аргументами, имеют свои потоки ввода и вывода, но определяются и запускаются в том же Unix shell, в котором их используют.

Команды shell: исполняемые файлы

- Если команда содержит в себе символ / (слеш), то она воспринимается как путь до конкретного файла, и shell пытается выполнить файл по указанному абсолютному или относительному (от текущей директории) пути
- Иначе — shell ищет файл с таким названием в известных ему адресах: а именно, в директориях, перечисленных через знак : (двоеточие) в **переменной окружения** `PATH`

А где же shell скрипты?

А где же shell скрипты?

В исполняемых файлах!

Unix: исполняемые файлы

Краткий экскурс в исполнение файлов в Unix-подобных системах.

Для запуска исполняемого файла по стандарту POSIX используется системный вызов:

```
int execve(
    const char * path,
    char * const argv[],
    char * const envp[]
);
```

Нам здесь важно:

- Процесс получает на вход массив **аргументов** (`argv[]`) — это просто массив строк, без какой-либо классификации
- Процесс получает на вход массив **переменных окружения** (`envp[]`) — это массив пар ключ-значение, где ключ и значение — это строки

Unix: исполняемые файлы

Прежде чем запустить файл, система проверяет наличие прав на запуск файла, как исполняемого (eXec).

Для запуска файла ядро операционной системы анализирует его содержимое, чтобы понять, как его запускать. В большинстве современных систем поддерживается формат запускаемых файлов ELF (Executable and Linkable Format). *Windows не в счёт...*

Кроме того POSIX-совместимые системы позволяют запускать текстовые скрипты как исполняемые файлы, используя **shebang**[1].

[1] В ядре Linux механизм запуска расширяем, то есть можно добавлять свои форматы запускаемых файлов

Unix: #!

shebang-ом называют специальную инструкцию по запуску в самом начале скрипта, из которой операционная система узнаёт, с помощью какой программы его запустить. Синтаксис:

```
#!путь аргумент
```

- `путь` — это абсолютный путь до файла, с помощью которого нужно запустить скрипт
- `аргумент` — это не более, чем один аргумент, который будет передан помимо пути до скрипта

Unix: #!

Примеры **shebang**:

```
#!/bin/sh
```

Здесь мы говорим, что наш скрипт стоит запускать командой `/bin/sh <путь до файла>` (путь до файла всегда передаётся одним аргументом, даже если он с пробелами).

Unix: #!

Примеры **shebang**:

```
#!/usr/bin/env sh
```

Здесь мы говорим, что наш скрипт стоит запускать командой `/usr/bin/env sh <путь до файла>`.

Это более предпочтительный способ, чем в предыдущем случае, потому что shell может находиться не в `/bin/sh`, а `/usr/bin/env` — это де-факто стандартный способ указания кросс-платформенных shebang.

Unix: без #!

Ясно, что скрипты можно запускать и вручную, полностью указывая команду запуска:

```
sh путь/к/файлу.sh аргументы
```

Pros:

+ не нужны права на выполнение для файла
+ явно указываем, какой shell использовать

Cons:

— неудобно

— громоздко

Скрипт — почти программа?

Пока что мы умеем в скриптах... Запускать команды? Также, как руками, но автоматически?

Что делать, если нам нужно также автоматически принимать решение о том, какую команду выполнять далее?

Команды shell: разделение команд

По-умолчанию shell считает, что каждая строка — это отдельная команда:

```
cmd1 args...
cmd2 args...
```

Чтобы перенести команду на новую строку необходимо написать в конце строки обратный слеш:

```
cmd args... \
    args...
```

Чтобы записать несколько команд в одну строку их необходимо разделить точкой с запятой:

```
cmd1 args... ; cmd2 args...
```

Команды shell: управление потоком выполнения

Да, в shell есть условия и даже циклы.

Простой пример управления потоком выполнения мы уже видели на первой лекции:

```
do_work && echo Ok || echo Fail
```

Если нам необходимо записать большее число команд в ветку условия, можно воспользоваться конструкцией

```
if do_work ; then echo Ok
           else echo Fail
fi
```

Все переводы строк могут быть заменены на ;.

Команды shell: условные выражения

Можно писать несколько команд:

```
if do_work ; then
    git commit -m Done
    echo Ok
else
    git reset --hard HEAD
    echo Fail
fi
```

Если нужно проверить что-то кроме успешности выполнения конкретной команды, существуют специальные команды для условий: `[`, `[[` и `test`.
В чем отличие?

Команды shell: условные выражения

Можно писать несколько команд:

```
if do_work ; then
    git commit -m Done
    echo Ok
else
    git reset --hard HEAD
    echo Fail
fi
```

Если нужно проверить что-то кроме успешности выполнения конкретной команды, существуют специальные команды для условий: `[`, `[[` и `test`.
В чем отличие? Уточните у `which` в `sh` и `man test`

Команды shell: условные выражения

`[` и `test` — это существующие программы, предназначенные для проверки условий. Они включены в POSIX, поэтому наверняка будут в целевой системе.

`[[` — это встроенная команда shell, которая не гарантируется POSIX (говорят, что она *может* быть реализована). Она будет работать быстрее, чем честный запуск программы.

`[` и `test` — это синонимы.
`[[`, если определяется, обычно делает то же самое.

Некоторые shell предоставляют встроенную команду `[` наравне с `[[`.

Команды shell: условные выражения

Синтаксис команд и условных выражений можно найти в `man test`.

Синтаксис команд:

- `test EXPRESSION` или `[ EXPRESSION ]` — вычисляет выражение
- `test` или `[]` — всегда ложь

Синтаксис условных выражений (`EXPRESSION`):

- `( EXPRESSION )` — подвыражение в скобках
- `! EXPRESSION` — логическое отрицание
- `EXPRESSION1 -a EXPRESSION2` — логическое И
- `EXPRESSION1 -o EXPRESSION2` — логическое ИЛИ
- `-n STRING` или просто `STRING` — строка не пустая
- `-z STRING` — строка пустая
- `STRING1 = STRING2` — **строки** равны
- `STRING1 != STRING2` — **строки** не равны

Команды shell: условные выражения

Синтаксис команд и условных выражений можно найти в `man test`.

Синтаксис условных выражений (`EXPRESSION`):

- `INTEGER1` **`op`** `INTEGER2` — сравнение **чисел**
  1. **`op`** может быть одним из: `-eq -ge -gt -le -lt -ne` для $=$, $\geq$, $>$, $\leq$, $<$, $\neq$ соответственно
- `-t FD` — файловый дескриптор FD открыт в текущем терминале
- `FILE1 -nt FILE2` — `FILE1` новее (modification date), чем `FILE2`
- `FILE1 -ot FILE2` — `FILE1` старше, чем `FILE2`

Команды shell: условные выражения

Синтаксис команд и условных выражений можно найти в `man test`.

Синтаксис условных выражений (`EXPRESSION`):

- **op** `FILE` — проверка чего-либо для файла. **op** может быть:
  - `-e` — файл существует
  - `-d -f -L` и др. — файл существует и конкретного типа (директория, обычный файл, символьная ссылка и др.)
  - `-N` — файл существует и был изменён с момента прошлого чтения
  - `-r -w -e` — файл существует и у нас есть права на чтение/запись/выполнение
  - `-s` — файл существует и его размер больше нуля
  - и другие...

Команды shell: условные выражения

Все отдельные элементы условных выражений должны быть разными цельными аргументами команды!
То есть, не забывайте писать пробелы между ними:

```
[[ 123=abc ]] && echo True || echo False
```

и не забывайте оборачивать в кавычки сложные выражения:

```
var='0 -o 1'
[[ 123 -eq $var ]] && echo True || echo False
```

Команды shell: условные выражения

Все отдельные элементы условных выражений должны быть разными цельными аргументами команды!
То есть, не забывайте писать пробелы между ними:

```
[[ 123=abc ]] && echo True || echo False
True
```

и не забывайте оборачивать в кавычки сложные выражения:

```
var='0 -o 1'
[[ 123 -eq $var ]] && echo True || echo False
True
```

Команды shell: условные выражения

Все отдельные элементы условных выражений должны быть разными цельными аргументами команды!
То есть, не забывайте писать пробелы между ними:

```
[[ 123 = abc ]] && echo True || echo False
False
```

и не забывайте оборачивать в кавычки сложные выражения:

```
var='0 -o 1'
[[ 123 -eq "$var" ]] && echo True \
  || echo False
False
```

Команды shell: сопоставление с образцом

В shell есть конструкция, схожая по смыслу с `case of` из Haskell, позволяющая сопоставлять значения с образцом:

```
case word in
    pattern | ... | pattern) command ;;
    pattern | ... | pattern) command ;;
    pattern | ... | pattern) command
esac
```

Более конкретный синтаксис смотреть в `man bash` или в стандарте POSIX (например, здесь).

Команды shell: циклы

В Unix shell есть 3 вида циклов:

```
while command1 ; do
    command2
done
```

Выполняется, пока `command1` возвращает ИСТИНА.

```
until command1 ; do
    command2
done
```

Выполняется, пока `command1` возвращает ЛОЖЬ.

Команды shell: циклы

```
for name in word ... word ; do
    command
done
```

Итерируется по каждому из `word`, помещая его в переменную `name`.

Также многие shell предоставляют синтаксис бесконечного цикла:

```
while : ; do command ; done
```

И C-style цикла `for`:

```
for (( e1 ; e2 ; e3 )) ; do command ; done
```

Команды shell: обработка ошибок

По-умолчанию shell не проверяет коды возврата команд, поэтому если какая-то команда из скрипта завершилась неуспешно, это не прерывает весь скрипт, а команды продолжают выполняться дальше:

```
make      # если сборка упадёт
./a.out   # здесь будет проблема
```

Что делать?

Команды shell: обработка ошибок

По-умолчанию shell не проверяет коды возврата команд, поэтому если какая-то команда из скрипта завершилась неуспешно, это не прерывает весь скрипт, а команды продолжают выполняться дальше:

```
make      # если сборка упадёт
./a.out   # здесь будет проблема
```

Чтобы досрочно завершить работу скрипта существует команда `exit`, она может принимать необязательный параметр — код возврата, по-умолчанию 0:

```
make || exit $?  # если сборка упадёт
./a.out          # мы сюда не дойдём
```

Команды shell: конвейеры и перенаправление ввода-вывода

Мы знаем, что мы можем добавлять для команд перенаправления ввода-вывода и объединять команды в конвейеры.

И то, что мы рассмотрели выше — не исключение:

```
produce_data | while read line ; do
    process_line "$line"
done
```