

Программирование на Bash

Краткое введение. Часть II

Всвоей второй статье Гарольд продолжает первоклассное введение в программирование на bash. На этот раз он объясняет, как выполнять арифметические операции в скриптах bash и как определить функции в своих программах. Завершается статья введением в такие продвинутые вещи как чтение пользовательского ввода, обработка скриптом аргументов, перехватывание сигналов и обработка кодов завершения программ.

Безусловно, результаты прочтения превзойдут все ожидания! После этой статьи вас уже нельзя будет назвать новичком. Ведь вы на пути к тому, чтобы называться мастером программирования на bash!

АРИФМЕТИКА И BASH

bash позволяет выполнять арифметические операции. Как вы уже видели в предыдущей статье, арифметика выполняется с помощью команды expr. Однако, подобно команде true, этот вариант считается медленным. Причина кроется в том, что для использования true и expr оболочки должна предварительно запустить их. Лучше всего использовать встроенную в bash функцию, которая работает быстрее. Аналогично тому, что альтернативой true является команда «»:, альтернатива expr – заключение арифметического выражения в конструкцию вида \$(...). Будьте внимательны, она отличается от \$(...). Отличие тут в количестве скобок. Так давайте же испробуем это:

```
#!/bin/bash
x=8 # присваиваем x значение 8
y=4 # присваиваем y значение 4
# результат сложения x и y сохраняем в z:
z=$((x + $y))
echo «Сумма $x и $y равна $z»
```

Как обычно, выбор используемого метода вычислений за вами. Если ис-

пользование expr для вас более комфортно и привычнее, чем \${(...)}, используйте его.

bash умеет выполнять сложение, вычитание, умножение, целочисленное деление и получение остатка от деления. Каждое арифметическое действие имеет соответствующий ему оператор:

Действие	Оператор
Сложение	+
Вычитание	-
Умножение	*
Целочисленное деление	/
Остаток от деления	%

Большинство из вас должно быть знакомо с первыми четырьмя операциями. Если вы не знаете, что такое деление по модулю, то это просто число равное остатку от деления одного целого числа на другое. Ниже приведен пример выполнения арифметических операций в bash:

```
#!/bin/bash
x=5 # устанавливаем x равным 5
y=3 # устанавливаем y равным 3
# сохраняем сумму x и y в переменную add
add=$((x + $y))
# сохраняем разность x и y в переменную sub
sub=$((x - $y))
# умножаем x на y и сохраняем результат в переменную mul
mul=$((x * $y))
# в переменную div сохраняем результат деления x на y
div=$((x / $y))
# получаем остаток от деления x на y и сохраняем его в переменную mod
mod=$((x % $y))
# печатаем ответы
```

```
echo «Сумма равна: $add»
echo «Разность равна $sub»
echo «Произведение равно $mul»
echo «Результат деления $div»
echo «Остаток от деления $mod»
```

Код, приведенный выше, можно было бы написать с использованием expr. Например, вместо add=\$((x + \$y)) мы могли бы использовать add=\$(expr \$x + \$y) или add=`expr \$x + \$y`.

ЧТЕНИЕ ВВОДА ПОЛЬЗОВАТЕЛЯ

А теперь – самое интересное. Мы напишем свой скрипт так, что он будет взаимодействовать с пользователем, а пользователь с ним. Команда для получения данных от пользователя – read. Это встроенная в bash команда, сохраняющая ввод пользователя в указанной переменной:

```
#!/bin/bash
# спросить у пользователя его имя
и поздороваться с ним
echo -n "Введите свое
имя: "
read user_name
echo "Привет $user_name!"
```

Переменная здесь – это user_name. Конечно, мы могли бы назвать ее как угодно. read прервет выполнение скрипта и будет ждать, пока пользователь введет что-нибудь и нажмет клавишу ENTER. Если клавиша ENTER была нажата без ввода чего-либо, read запустит следующую строку кода. Попробуйте это сделать.

Ниже приведен тот же пример, только на этот раз мы проверяем, вводит ли что-то пользователь:

```
#!/bin/bash
# спрашиваем имя пользователя и
выводим приветствие
echo -n «Введите имя: »
```

```
read user_name
# проверка ввода пользователя
if [ -z «$user_name» ]; then
    echo «Вы не сказали мне свое имя!»
    exit
fi
echo «Привет $user_name!»
```

В приведенном примере, если пользователь нажал ENTER и не ввел ничего при этом, наша программа напишет об этом и завершит свою работу. В противном случае она напечатает приветствие. Получение пользовательского ввода полезно для интерактивных программ, которые требуют от пользователя ввести какие-то данные.

ФУНКЦИИ

Использование функций делает проще поддержку своих скриптов. Проще говоря, это хороший способ разделить программу на более мелкие куски. Функция выполняет определенное действие и может возвращать то значение, какое вы пожелаете. Прежде чем продолжать, я приведу пример скрипта, написанного с использованием функции:

```
#!/bin/bash
# функция hello() печатает сообщение
hello()
{
    echo «Вы находитесь в функции hello()»
}
echo «Вызываем функцию hello()...»
hello
```

Попробуйте запустить код из примера выше. Функция hello() в нем имеет только одно предназначение – просто напечатать сообщение. Но, конечно же, они могут решать и более сложные задачи. Выше мы вызвали функцию hello(), используя строку:

hello

Когда запускается эта строка, bash ищет скрипт для строки hello(). Он находит его в начале файла и выполняет его содержимое. Функции всегда вызываются по своему имени, что мы и видели выше. При написа-

нии функции вы можете объявить ее, просто указав имя_функции(), как это сделано выше, или если вы хотите сделать ее объявление более явным, можете объявить ее так: function имя_функции(). Ниже представлен альтернативный способ написания функции **hello()**:

```
function hello()
{
    echo «Вы находитесь в функции hello()»
}
```

Функции имеют в имени пустые открывающую и закрывающую скобки: «()», за ними следует пара фигурных скобок: «{...}», содержащих тело функции. Другими словами, весь код функции заключен в эти фигурные скобки. Функции всегда должны быть предварительно объявлены до своего вызова. Давайте попробуем в приведенном выше примере вызвать функцию до ее объявления:

```
#!/bin/bash
echo «Вызов функции hello() ...»
hello
# функция hello() просто выводит сообщение
hello()
{
    echo «Вы находитесь в функции привет ()»
}
```

Вот что мы получим, когда попытаемся запустить этот скрипт:

```
$ ./hello.sh
Вызов функции привет ()
./hello.sh: hello:
command not found
```

Как видите, мы получили сообщение об ошибке. Поэтому стоит всегда размещать ваши функции в начале кода или, по крайней мере, непосредственно перед вызовом функции. Еще один пример использования функции:

```
#!/bin/bash
# admin.sh – инструмент для администратора
# функция new_user () создает новую учетную запись пользователя
new_user()
{
    echo «Подготовка к соз-
```

данию новых пользователей ...»

```
sleep 2
# запускаем программу adduser
adduser
}
echo «1. добавить пользователя»
echo «2. Выход»
echo «Укажите, что вы хотите сделать:»
read choice
case $choice in
1) new_user # вызов функции new_user()
;;
*) exit
;;
esac
```

Для того чтобы приведенный скрипт работал правильно, вам необходимо запустить его из-под пользователя root, т. к. иначе программа adduser не сможет создать новых пользователей. Надеюсь, этот пример (хоть он и краток) показывает положительный эффект от использования функций.

ПЕРЕХВАТ СИГНАЛОВ

Вы можете использовать встроенную в bash программу trap для перехвата сигналов в своих программах. Это хороший способ изящно завершать работу программы. Например, если пользователь, когда ваша программа работает, нажмет CTRL-C – программе будет отправлен сигнал interrupt (SIGINT, signal (7)), который завершит ее. Trap позволит вам перехватить этот сигнал, что даст возможность либо продолжить выполнение программы, либо сообщить пользователю, что программа завершает работу. Синтаксис этой команды такой:

trap action signal

Здесь:
action – то, что вы хотите делать, когда сигнал получен;
signal – сигнал, на который стоит реагировать.

Список сигналов можно посмотреть с помощью команды trap -l. При указании сигналов в своих скриптах можно опустить первые три буквы названия сигнала, т. е. SIG. Например, сигнал прерывания это – SIGINT. В ва-

шем скрипте, в качестве его имени, можно указать просто INT. Вы также можете использовать номер сигнала, указанный рядом с его именем. Например, числовое значение сигнала SIGINT – 2. Попробуйте написать и запустить приведенный ниже пример:

```
#!/bin/bash
# использование команды
trap
# перехватываем нажатие
CTRL-C и запускаем функцию
sorry()
trap sorry INT

# function sorry() prints
a message
sorry()
{
echo «Извини меня, дэйв.
я не могу этого сделать»
sleep 3
}
# обратный отсчет от 10
до 1:
echo «Подготовка к уничтожению системы»
for i in 10 9 8 7 6 5 4 3
2 1; do
echo «Осталось $i секунд
до уничтожения...»
sleep 1
done
echo «Запуск программы
уничтожения!»
```

Наберите и запустите приведенный пример. Когда программа будет работать и вести обратный отсчет, нажмите CTRL-C. Это действие отправит программе сигнал прерывания – SIGINT. Тем не менее сигнал будет перехвачен командой trap, которая, в свою очередь, выполнит функцию sorry(). Вы можете заставить trap игнорировать сигнал, указав символ кавычек вместо указания действия. Также вы можете отключить ловушку с помощью тире: «-». Например:

```
# запускать функцию
sorry(), если получен сиг-
нал SIGINT
trap sorry INT

# отключение ловушки
trap - INT
# ничего не делать при
получении сигнала SIGINT
trap " INT
```

Если вы отключаете ловушку, программа работает как обычно – при получении сигнала прерывается ее

исполнение и она завершает работу. Когда вы говорите trap ничего не делать при получении сигнала – она делает именно это. Ничего. Программа будет продолжать работать, игнорируя сигнал.

ЛОГИЧЕСКИЕ И И ИЛИ

Вы уже видели, что такое управляющие структуры и как их использовать. Для решения тех же задач есть еще два способа. Это логическое И – «&&» и логическое «ИЛИ» – «||». Логическое И используется следующим образом:

```
выражение_1 && выраже-
ние_2
```

Сначала выполняется выражение, стоящее слева, если оно истинно, выполняется выражение, стоящее справа. Если выражение_1 возвращает ЛОЖЬ, то выражение_2 не будет выполнено. Если оба выражения возвращают ИСТИНУ, выполняется следующий набор команд. Если какое-то из выражений не истинно, приведенное выражение считает ложным в целом. Другими словами, все работает так:

```
если выражение_1 истинно
и выражение_2 истинно, тог-
да выполнять...
```

Примечание переводчика: при работе с булевыми переменными ИСТИНА и ЛОЖЬ (True и False), bash ведет себя отлично от других языков программирования. В других языках 0 соответствует False (Ложь), а 1 – True (Истина). В bash все наоборот. Это связано с кодами завершения программ (см. ниже). Об этом следует всегда помнить при написании своих скриптов!

Пример использования:

```
#!/bin/bash
x=5
y=10
if [ «$x» -eq 5 ] && [
«$y» -eq 10 ]; then
echo «Оба условия верны»
else
echo «Условия не верны»
fi
```

Здесь мы обнаруживаем, что переменные x и y содержат именно те значения, которые мы проверям, поэтому проверяемые условия верны. Если вы

измените значение с x = 5 на x = 12, а затем снова запустите программу, она выдаст фразу «Условия не верны».

Логическое ИЛИ используется аналогичным образом. Разница лишь в том, что оно начинает проверять ложность первого условия, если оно ложно, начинает выполняться следующий оператор:

```
выражение_1 || выраже-
ние_2
```

Данное выражение в псевдокоде выглядит так:

```
если выражение_1 истин-
но ИЛИ выражение_2 истинно,
выполняем ...
```

Таким образом, любой последующий код будет выполняться, если хотя бы одно из выражений истинно:

```
#!/bin/bash
x=3
y=2
if [ «$x» -eq 5 ] || [
«$y» -eq 2 ]; then
echo «Одно из условий ис-
тинно»
else
echo «ни одно из условий
не является истинным»
fi
```

Здесь вы видите, что только одно из выражений истинно. Попробуйте изменить значение y и повторно запустите программу. Вы увидите сообщение, что ни одно из выражений не является истинным.

Аналогичная реализация условия с помощью оператора if будет большего размера, чем вариант с использованием логического И и ИЛИ, поскольку потребует дополнительного вложенного if. Ниже приведен код, реализующий тот же функционал, но с использованием оператора if:

```
#!/bin/bash
x=5
y=10
if [ «$x» -eq 5 ]; then
if [ «$y» -eq 10 ]; then
echo «Оба условия верны»
else
echo «Оба условия невер-
ны»
fi
```

Приведенный код менее нагляден для чтения и требует больших усилий

для написания. Но у вас остается шанс избавить себя от этих трудностей путем использования логических операторов И и ИЛИ.

ИСПОЛЬЗОВАНИЕ АРГУМЕНТОВ

Возможно, вы уже заметили, что большинство программ в Linux не интерактивны. Вы должны указать им какие-то параметры, в противном случае вы получите сообщение со списком возможных аргументов. Возьмем, к примеру, команду `more`. Если вы не укажете имя файла, она выдаст краткую справку по использованию программы. Также можно сделать так, чтобы ваши скрипты также могли принимать аргументы. Для этого вам нужно знать, что такое переменная вида `$#`. В ней содержится общее количество аргументов, переданных программе. Например, если вы запустите программу следующим образом:

```
$ что-то параметр
```

то значение переменной `$#` будет равно единице, потому что программе передан только один аргумент. Для двух аргументов ее значение будет равно двум и так далее. Также стоит знать о том, что каждый параметр командной строки (включая даже имя скрипта!!!) сохраняется в соответствующей переменной.

Так, имя нашей программы «что-то» будет сохранено в переменной `$0`. Аргумент программы – параметр сохраняется в переменной `$1`. Вы можете использовать до 9 переменных, начиная с `$0` (обозначающего имя скрипта), а затем `$1-$9`, обозначающие аргументы программы. Давайте посмотрим, как это работает:

```
#!/bin/bash
# скрипт, печатающий свои аргументы
# проверяем, переданы ли скрипту аргументы:
```

```
if [ «$#» -ne 1 ]; then
    echo «корректный запуск программы: $0 <параметр>»
fi
echo «Переданный параметр - $1»
```

Приведенный скрипт ожидает один и только один аргумент для

своего запуска. Если вы не укажете ему аргументов – будет выводиться справочная информация. В противном случае, если при запуске указан какой-то аргумент – он будет передан в наш скрипт, который выведет его на экран. Напоминаю, что `$0` – это имя скрипта. Именно поэтому эта переменная используется в справочном сообщении. Последняя строка выводит переданный программе параметр – `$1`.

РАБОТА С ВРЕМЕННЫМИ ФАЙЛАМИ

Довольно часто вам будет необходимо создавать временные файлы. Обычно это файл, в котором хранятся какие-то используемые скриптом данные, либо что-то еще. Как только работа скрипта будет завершена, этот файл нужно удалить. При создании такого файла вы должны задать его имя. Проблема тут кроется в том, что файл, создаваемый вами, не должен случайно переписать уже существующий в той же директории, если их имена совпадут.

Для того, чтобы создать временный файл с гарантированно уникальным именем, вам нужно использовать символ `«$$»`, либо как префикс, либо как суффикс к имени создаваемого файла. Предположим, вы хотите создать временный файл с именем `hello`. Возможно, что у пользователя, который работает с нашим скриптом, уже есть файл с таким именем. Создавая файл с именем `hello.$$` или `$$hello`, вы создадите файл с уникальным именем. Например:

```
$ touch hello
$ ls
hello

$ touch hello.$$
$ ls
hello hello.689
```

Примерно так и будет выглядеть имя вашего временного файла.

Примечание переводчика: В переменной `$$` обычно хранится следующий свободный PID. Именно поэтому использование такой переменной гарантирует уникальные имена для вновь создаваемых файлов.

КОДЫ ЗАВЕРШЕНИЯ ПРОГРАММ

Большинство программ возвращают в операционную систему какое-то число, показывающее, насколько удачно программа завершила свою работу. Например, страница `grep` говорит, что `grep` вернет 0, если заданный шаблон найден, и 1, если совпадений не найдено. Почему нас так волнуют эти коды завершения? По разным причинам. Допустим, мы хотим проверить – есть ли пользователь с данным именем в системе? Один из способов сделать – использовать команду вида: `grep имя_пользователя /etc/passwd`. Допустим, имя пользователя — `vasya`:

```
$ grep vasya /etc/passwd
$
```

Ничего не вывелоось. Это означает, что `grep` не обнаружила заданного пользователя. Но для нас было бы значительно лучше получить сообщение об этом. Это как раз тот случай, когда нужно использовать код завершения программы. Он сохраняется в переменной с именем `$?`. Посмотрим на следующий фрагмент кода:

```
#!/bin/bash
# ищем пользователя vasya в /etc/passwd,
# весь вывод перенаправляем в /dev/null
grep vasya /etc/passwd > /dev/null 2>&1
```

смотрим код завершения и действуем по обстоятельствам:

```
if [ «$?» -eq 0 ]; then
    echo «Пользователь vasya найден»
    exit
else
    echo «Пользователь vasya не найден»
fi
```

Теперь, когда вы запустите скрипт, он будет перехватывать и анализировать код завершения `grep`. Если он равен 0, значит пользователь найден и мы выводим соответствующее сообщение об ошибке. В противном случае скрипт напечатает, что пользователя найти не получилось. Это очень простой способ использования получаемого кода завершения программы. По мере практики вы сами будете

понимать, для решения какой задачи вам нужно использовать эти коды завершения.

Возможно вас озадачивает конструкция вида `2>&1`, но все довольно просто. В Linux этими числами обозначаются дескрипторы файлов. 0 – стандартный ввод (по умолчанию, клавиатура), 1 стандартный вывод (по умолчанию, монитор) и 2 – вывод стандартных ошибок (по умолчанию, монитор).

Весь вывод команды идет в файл с дескриптором 1, любые ошибки отправляются в файл с дескриптором 2. Если вы не хотите, чтобы сообщения об ошибках появлялись на экране, просто перенаправьте его в `/dev/null`. Но это не прекратит вывод на экран обычной информации. Например, если у вас нет разрешения на чтение домашнего каталога другого пользователя, вы не сможете просмотреть список его содержимого:

```
$ ls /root
ls: /root: Permission
denied
```

```
$ ls /root 2> /dev/null
```

Как видите, во второй раз информация об ошибке не была напечатана. Все то же самое относится к другим программам и дескриптору 1. Если вы не хотите видеть нормальный выход из программы, то есть хотите, чтобы она работала молча, вы можете перенаправить в `/dev/null` и его. Теперь, если вы не хотите видеть вообще никакого вывода программы – добавьте в нее следующее:

```
$ ls /root > /dev/null
2>&1
```

Это означает, что программа будет отправлять свой вывод и ошибки, которые возникают в `/dev/null`, т. е. будет работать молча, что нам и нужно.

Примечание переводчика: на самом деле все работает так:

Конструкция вида `2>&1` перенаправляет вывод ошибок (дескриптор 2) на стандартный вывод (дескриптор 1). Знак «загогулины» – & – тут нужен для того, чтобы пояснить bash, что вы имеете в виду не файл с именем 1, а

именно файл с дескриптором 1, т. е. стандартный вывод. Если вы укажете что-то вроде:

`$ команда 2>1`

то стандартный вывод ошибок пойдет в файл с именем 1. Конструкцией `2>&1` мы «цепляем» вывод команды и вывод ошибок вместе. А первым перенаправлением (первым символом `>` в комментируемой команде) мы отправляем весь вывод команды в `/dev/null`. Чтобы дополнительно понять, как все работает, можете поэкспериментировать, убрав `2>&1` из команды и перезапустив ее.

А что если вы хотите, чтобы ваш скрипт тоже возвращал какой-нибудь код завершения при выходе? Команда `exit` может принимать один аргумент – тот самый код завершения. Обычно число 0 используется для обозначения успешного завершения работы. Число, отличное от нуля означает, что произошла какая-то ошибка. Какое число возвращать – решает сам программист. Посмотрим приведенный пример:

```
#!/bin/bash
if [ -f «/etc/passwd» ]; then
    echo «файл passwd существует»
    exit 0
else
    echo «нет такого файла»
    exit 1
fi
```

Задавая значение кода завершения, вы делаете возможным для других скриптов, использующих ваш скрипт, анализировать результаты его работы.

ПЕРЕНОСИМОСТЬ ВАШИХ СКРИПТОВ НА BASH

При написании ваших собственных скриптов важно делать это так, чтобы они оставались переносимыми. Термин «переносимость» означает, что если ваш скрипт работает под Linux, то он должен работать в другой Unix-системе с малыми изменениями или вообще без них. Чтобы добиться этого, вы должны быть осторожны при вызове внешних программ. Разработчик должен при этом ответить на вопрос: «Будет ли эта программа

доступна на другом варианте Unix?» (и что более важно – будет ли она там работать также, как на Linux – прим. перев.). Допустим, вы используете программу `foo`, которая на Linux работает аналогично `echo`, поэтому вместо `echo` вы используете ее. Но если ваш скрипт будет работать на других системах, где нет программы `foo`, он начнет выдавать сообщения об ошибках. Кроме того, имейте в виду, что разные версии `bash` могут иметь разные методы для одних и тех же операций. Например, конструкция `VAR = $(ps)` делает то же самое, что и `VAR = `ps``, но на самом деле старые версии оболочек, например Bourne shell (`sh`), признают только последний синтаксис. Если вы собираетесь распространять свои скрипты, обязательно включайте текстовый файл `README`, который будет предупреждать пользователя о любых сюрпризах, в том числе и о том, что скрипт проверялся на такой-то версии `bash`. Желательно также указать, какие программы и библиотеки (и каких версий) будут нужны скрипту.

Примечание переводчика: Для проверки наличия в скрипте команд и функций специфичных для `bash` в ALT Linux есть пакет `checkbashisms`, который взят из пакета `devscripts` Debian.

ЗАКЛЮЧЕНИЕ

Пришла пора завершить это краткое введение в написание скриптов на `bash`. Однако ваше обучение этому умению еще не завершено. В тоже время, написанного вполне достаточно, чтобы вы могли модифицировать имеющиеся скрипты и писать собственные. Если вы действительно хотите стать мастером написания скриптов на `bash`, я рекомендую приобрести книгу «*Learning the bash shell*» (Изучение оболочки `bash`), 2-е издание издательства O'Reilly & Associates, Inc.

Скрипты на `bash` идеально подходят для повседневной работы по администрированию. Но если вы планируете что-то более серьезное, следует использовать гораздо более мощный язык, такой как C или Perl. Удачи!

Антон Чернышов
Linux-преподаватель R-Style
www.tux-the-penguin.blogspot.com