**BASH: *Разработка сценариев оболочки***

Ряд встроенных команд, которые мы будем использовать для создания своих скриптов.

|  |  |
| --- | --- |
| **break** | выход из цикла for, while или until |
| **continue** | выполнение следующей итерации цикла for, while или until |
| **echo** | вывод аргументов, разделенных пробелами, на стандартное устройство вывода |
| **exit** | выход из оболочки Bash |
| **export** | отмечает аргументы как переменные для передачи в дочерние процессы в среде |
| **hash** | запоминает полные имена путей команд, указанных в качестве аргументов, чтобы не искать их при следующем обращении |
| **kill** | посылает сигнал завершения процессу |
| **pwd** | выводит текущий рабочий каталог |
| **read** | читает строку из ввода оболочки и использует ее для присвоения значений указанным переменным.\ |
| **return** | заставляет функцию оболочки выйти с указанным значением |
| **shift** | перемещает позиционные параметры налево |
| **test** | вычисляет условное выражение |
| **times** | выводит имя пользователя и системное время, использованное оболочкой и ее потомками |
| **trap** | указывает команды, которые должны выполняться при получении оболочкой сигнала |
| **unset** | вызывает уничтожение переменных оболочки |
| **wait** | ждет выхода из дочернего процесса и сообщает выходное состояние. |

Пример:

**#!/bin/bash #**указываем где у нас хранится bash-интерпретатор

**parametr1=$1 #**присваиваем переменной parametr1 значение первого параметра скрипта

**script\_name=$0 #**присваиваем переменной script\_name значение имени скрипта

**echo** "**Вы запустили скрипт с именем $script\_name и параметром $parametr1"**

**#** команда echo выводит определенную строку, обращение к переменным осуществляется

**#** через $имя\_переменной.

**echo** **'Вы запустили скрипт с именем $script\_name и параметром $parametr1'**

**#** здесь мы видим другие кавычки, разница в том, что в одинарных кавычках не происходит

**#** подстановки переменных.

**exit 0 #**Выход с кодом 0 (удачное завершение работы скрипта)

Результат выполнения скрипта:

***ite@ite-desktop:~$ ./test.sh qwerty***

***Вы запустили скрипт с именем ./test.sh и параметром qwerty***

***Вы запустили скрипт с именем $script\_name и параметром $parametr1***

|  |  |
| --- | --- |
| **Зарезервированные переменные**: | |
| **$DIRSTACK** | содержимое вершины стека каталогов |
| **$EDITOR** | текстовый редактор по умолчанию |
| **$EUID** | Эффективный UID. Если вы использовали программу **su** для выполнения команд от другого пользователя, то эта переменная содержит UID этого пользователя, в то время как... |
| **$UID** | ...содержит реальный идентификатор, который устанавливается только при логине. |
| **$FUNCNAME** | имя текущей функции в скрипте. |
| **$GROUPS** | массив групп, к которым принадлежит текущий пользователь |
| **$HOME** | домашний каталог пользователя |
| **$HOSTNAME** | ваш hostname |
| **$HOSTTYPE** | архитектура машины. |
| **$LC\_CTYPE** | внутренняя переменная, которая определяет кодировку символов |
| **$OLDPWD** | прежний рабочий каталог |
| **$OSTYPE** | тип ОС |
| **$PATH** | путь поиска программ |
| **$PPID** | идентификатор родительского процесса |
| **$SECONDS** | время работы скрипта (в сек.) |
| **$#** | общее количество параметров, переданных скрипту |
| **$\*** | все аргументы, переданные скрипту (выводятся в строку) |
| **$@** | тоже самое, что и предыдущий, но параметры выводятся в столбик |
| **$!** | PID последнего запущенного в фоне процесса |
| **$$** | PID самого скрипта |

**Условия**

Условные операторы, думаю, знакомы практически каждому, кто хоть раз пытался на чем-то писать программы. В bash условия пишутся след. образом (как обычно на примере):

**#!/bin/bash**

**source=$1 #**в переменную **source** засовываем первый параметр скрипта

**dest=$2 #**в переменную **dest** засовываем второй параметр скрипта

**if [[ "$source" -eq "$dest" ]] #** в кавычках указываем имена переменных для сравнения.

# -**eq** - логическое сравнение обозначающие "**равны**"

**then #** если они действительно равны, то

**echo** **"Приёмник $dest и источник $source один и тот же файл!" #**выводим сообщение об ошибке,

# т.к. $source и $dest у нас равны

**exit 1 #** выходим с ошибкой (1 - код ошибки)

**else #** если же они не равны

**cp $source $dest #** то выполняем команду cp: копируем источник в приемник

**echo** **"Удачное копирование!"**

**fi #**обозначаем окончание условия.

*Результат выполнения скрипта:*

***ite@ite-desktop:~$ ./primer2.sh 1 1***

***Применик 1 и источник 1 один и тот же файл!***

***ite@ite-desktop:~$ ./primer2.sh 1 2***

***Удачное копирование!***

Структура **if-then-else** используется следующим образом:

**if <команда или набор команд возвращающих код возврата(0 или 1)>**

**then**

**<если выражение после if истино, то выполняется этот блок>**

**else**

**<если выражение после if ложно, тот этот>**

В качестве команд возвращающих код возврата могут выступать структуры **[[ , [ , test, (( ))** или любая другая(или несколько) linux-команда.

**test** - используется для логического сравнения. после выражения, необходима закрывающая скобка "]"

**[** - синоним команды test

**[**[ - расширенная версия "[" (начиная с версии 2.02)(как в примере), внутри которой могут быть использованы || (или), & (и). Должна иметь закрывающую скобку "]]"

**(( ))** - математическое сравнение.

|  |  |
| --- | --- |
| для построения многоярусных условий вида:  **if ...**  **then ....**  **else**  **if ....**  **then....**  **else ....** | для краткости и читаемости кода, можно использовать структуру:  **if ..**  **then ...**  **elif ...**  **then ...**  **elif ...** |

**Условия. Множественный выбор**

Если необходимо сравнивать какую-то одну переменную с большим количеством параметров, то целесообразней использовать оператор case.

**#!/bin/bash**

**echo "Выберите редактор для запуска:"**

**echo "1 Запуск программы nano"**

**echo "2 Запуск программы vi"**

**echo "3 Запуск программы emacs"**

**echo "4 Выход"**

**read doing** #здесь мы читаем в переменную $doing со стандартного ввода

**case $doing in**

**/usr/bin/nano** # если $doing содержит 1, то запустить nano

Результат работы:

***ite@ite-desktop:~$ ./menu2.sh***

***Выберите редатор для запуска:***

***1 Запуск программы nano***

***2 Запуск программы vi***

***3 Запуск программы emacs***

***4 Выход***

**;;**

**2)**

**/usr/bin/vi** # если $doing содержит 2, то запустить vi

**;;**

**3)**

**/usr/bin/emacs** # если $doing содержит 3, то запустить emacs

**;;**

**4)**

**exit 0**

**;;**

**\*)** #если введено с клавиатуры то, что в case не описывается, выполнять следующее:

**echo "Введено неправильное действие"**

**esac** #окончание оператора case.

После выбор цифры и нажатия Enter запуститься тот редактор, который вы выбрали (если конечно все пути указаны правильно, и у вас установлены эти редакторы :) )

|  |  |
| --- | --- |
| **Список логических операторов**, которые используются для конструкции **if-then-else-fi**: | |
| **-z** | # строка пуста |
| **-n** | # строка не пуста |
| **=, (==)** | # строки равны |
| **!=** | # строки неравны |
| **-eq** | # равно |
| **-ne** | # неравно |
| **-lt,(< )** | # меньше (< –внутри двойных круглых скобок; ниже – аналогично) |
| **-le,(<=)** | # меньше или равно |
| **-gt,(>)** | #больше |
| **-ge,(>=)** | #больше или равно |
| **!** | #отрицание логического выражения |
| **-a,(&&)** | #логическое «И» |
| **-o,(||)** | # логическое «ИЛИ» |

|  |  |  |
| --- | --- | --- |
| **C++** | Bash **for** | bash **while** |
| **for (i = 0; i < 10; i++)**  **{**  **printf("number is %d\n", i);**  **}** | **for (( a = 1; a < 10; a++ ))**  **do**  **echo "number is $i"**  **done** | **while команда проверки условия**  **do**  **другие команды**  **done** |

В **bash** есть много операторов для описания условных выражений и логических операций и их синтаксис не всегда привычен (по крайней мере на первый взгляд), так что не лишним будет иметь шпаргалку.

**Логические операции:**

! – отрицание;

–a или && – логическое И;

–o или || – логическое ИЛИ.

**Элементарные условные выражения для файлов:**

-e — файл существует;

-f — обычный файл (не каталог и не файл устройства);

-s — ненулевой размер файла;

-d — файл является каталогом;

-b — файл является блочным устройством (floppy, cdrom и т.п.);

-c — файл является символьным устройством (клавиатура, модем, звуковая карта и т.п.);

-p — файл является каналом;

-h — файл является символической ссылкой;

-L — файл является символической ссылкой;

-S — файл является сокетом;

-t — файл связан с терминальным устройством;

-r — файл доступен для чтения (пользователю, запустившему сценарий);

-w — файл доступен для записи (пользователю, запустившему сценарий);

-x — файл доступен для исполнения (пользователю, запустившему сценарий);

-g — (sgid) флаг для файла или каталога установлен;

-u — (suid) флаг для файла установлен;

-k — флаг sticky bit установлен;

-O — вы являетесь владельцем файла (пользователь запустивший сценарий);

-G — вы принадлежите к той же группе, что и файл (пользователь запустивший сценарий);

-N — файл был модифицирован с момента последнего чтения;

file1 -nt file2 – file1 более новый, чем file2;

file1 -ot file2 – file1 более старый, чем file2;

file1 -ef file2 – file1 и file2 являются «жесткими» ссылками на один и тот же файл.

**Элементарные условные выражение для сравнения строк:**

-z str – длина строки str равна 0;

-n str – длина строки str не равно 0;

str1 == str2 или str1 = str2 – строки совпадают;

str1 !== str2 или str1 != str2 – строки не совпадают;

str1 < str2 – str1 предшествует str2 в лексикографическом порядке;

str1 > str2 – str1 следует за str2 в лексикографическом порядке.

**Арифметическое условное выражение имеет формат: arg1 операция arg2, где аргументами являются целые числа, и допустимы следующие операции:**

-eq – равно;

-ne – не равно;

-lt – меньше;

-le – меньше или равно;

-gt – больше;

-ge – больше или равно;

< — меньше (внутри двойных круглых скобок);

<= — меньше или равно (внутри двойных круглых скобок);

> — больше (внутри двойных круглых скобок);

>= — больше или равно (внутри двойных круглых скобок);

**Битовые операции:**

<< - сдвигает на 1 бит влево (умножение на 2);

<<= - сдвиг-влево-равно; Например let "x <<= 2" значение переменной x сдвигается влево на 2 бита (умножается на 4);

>> - сдвиг вправо на 1 бит (деление на 2);

>>= - сдвиг-вправо-равно (имеет смысл обратный <<=);

& - по-битовое И (AND);

&= - по-битовое И-равно;

| - по-битовое ИЛИ (OR);

|= - по-битовое ИЛИ-равно;

~ - по-битовая инверсия;

! - По-битовое отрицание;

^ - по-битовое исключающее или (XOR);

^= - по-битовое исключающее-или-равно;

**Команды возвращающие код возврата** *(звучит странно, но команды нужные):*

test - используется для логического сравнения (например test -f /home/kyrych/readme вернет истину если readme - обычный файл);

[ ] - является специальной встроенной командой воспринимающей свои аргументы как выражение сравнения или файловую проверку [.....]. Вызывает ошибку при попытке использования внутри || (логическое или) и & (логическое и);

[[ ]] - расширенная версия "[" (начиная с версии 2.02), внутри которой могут быть использованы || (логическое или), & (логическое и). Должна иметь закрывающую скобку "]]" (является зарезервированным словом, а не командой, его bash выполняет как один элемент с кодом возврата)

(( )) - математическое сравнение.

**Команда break**

Эта команда позволяет прервать выполнение цикла. Её можно использовать и для циклов for, и для циклов while

**Команда continue**

Когда в теле цикла встречается эта команда, текущая итерация завершается досрочно и начинается следующая, при этом выхода из цикла не происходит.

**Чтение параметров командной строки**

Оболочка bash назначает специальным переменным, называемым позиционными параметрами, введённые при вызове скрипта параметры командной строки:

$0 — имя скрипта.

$1 — первый параметр.

$2 — второй параметр — и так далее, вплоть до переменной $9, в которую попадает девятый параметр.

Вот как можно использовать параметры командной строки в скрипте с помощью этих переменных:

|  |  |
| --- | --- |
| **#!/bin/bash**  **echo $0**  **echo $1**  **echo $2**  **echo $3**  Запустим сценарий с параметрами:  **./myscript 5 10 15**  Вот что он выведет в консоль: | https://habrastorage.org/getpro/habr/post_images/fb5/f9f/b80/fb5f9fb80361a4083e31d7225fa3cdbf.png |

**Объявление функций**

|  |  |
| --- | --- |
| Функцию можно объявить так:  **functionName {**  **}** | Или так:  **functionName() {**  **}** |

Функцию можно вызвать без аргументов и с аргументами.

|  |  |
| --- | --- |
| **#!/bin/bash**  **function myfunc {**  **echo "This is an example of using a function"**  **}**  **count=1**  **while [ $count -le 3 ]**  **do**  **myfunc**  **count=$(( $count + 1 ))**  **done**  **echo "This is the end of the loop"**  **myfunc**  **echo "End of the script"** |  |

Команда **return** позволяет задавать возвращаемый функцией целочисленный код завершения.

Максимальное число, которое может вернуть команда **return** — 255. Если функция должна возвращать большее число или строку, понадобится другой подход.

Пример. **Скрипт для мониторинга дискового пространства**

|  |  |
| --- | --- |
| **#!/bin/bash**  **MY\_DIRECTORIES="/home /var/log"**  **echo "Top Ten Disk Space Usage"**  **for DIR in $MY\_DIRECTORIES**  **do**  **echo "The $DIR Directory:"**  **du -S $DIR 2>/dev/null |**  **sort -rn |**  **sed '{11,$D; =}' |**  **sed 'N; s/\n/ /' |**  **awk '{printf $1 ":" "\t" $2 "\t" $3 "\n"}'**  **done**  **exit** |  |

Команда **du**, которая выводит сведения о том, сколько места на диске занимают файлы и папки.

Для наших целей лучше подойдёт ключ -S (заглавная S), так как он позволяет получить сведения как по корневой папке, так и по вложенным в неё директориям: $ **du -S /var/log/**

$ **du -S /var/log/ | sort –rn** # список, который выдаёт **du**, надо отсортировать

Чтобы ограничить полученный список первыми десятью записями, воспользуемся потоковым редактором sed, который позволит удалить из полученного списка все строки, начиная с одиннадцатой.

Следующий шаг — добавить к каждой полученной строке её номер. Тут также поможет **sed**, а именно — его команда N: **sed '{11,$D; =}' |**

**sed 'N; s/\n/ /' |**

Приведём полученные данные в порядок, воспользовавшись **awk**. Передадим **awk** то, что получилось после обработки данных с помощью **sed**, применив, как и в других случаях, конвейер, и выведем полученные данные с помощью команды **printf.**