Introducción a Bash

PID_00270628

Àngel Ollé Blázquez

Tiempo mínimo de dedicación recomendado: 4 horas

Universitat Oberta de Catalunya

Àngel Ollé Blázquez

Ingeniero técnico en Informática de gestión por la Universidad Rovira i Virgili (URV). Ingeniero en Informática y máster en Software libre por la Universitat Oberta de Catalunya (UOC). Actualmente trabaja como ingeniero de software y participa en proyectos *open source*. Anteriormente desarrolló su actividad profesional en el sector de servicios y consultoría IT por EMEA.

El encargo y la creación de este recurso de aprendizaje UOC han sido coordinados por el profesor: Julià Minguillón Alfonso (2020)

Primera edición: febrero 2020

Avda. Tibidabo, 39-43, 08035 Barcelona
Realización editorial: FUOC

Índice

Introducción

Bash (*Bourne-again Shell*) es una de las *shells* o línea de comandos más populares que hay en la mayoría de distribuciones GNU/Linux.

Shell

Una *shell* es una aplicación que permite ejecutar comandos del sistema operativo.

Bash incluye mejoras propias, pero también tiene características de otras *shells* populares, como la **Korn Shell** y la **C Shell**, y, a su vez, es compatible con la de su predecesora, **Bourne Shell**.

En este módulo se presentan los conceptos básicos de las partes más importantes de Bash, empezando por conocer la línea de comandos y qué es un *script*. Veremos los caracteres especiales del lenguaje, los operadores y las expansiones, detallando su uso y sus cualidades, y los diferentes tipos de variables para almacenar los datos. Además, el módulo hace una introducción a las sentencias fundamentales, como, por ejemplo, las sentencias condicionales y las sentencias iterativas. Finalmente, se muestra cómo depurar el código desarrollado.

1. Línea de comandos con Bash

En el módulo «Introducción a GNU/Linux» se ha visto la línea de comandos con las instrucciones más típicas que se acostumbran a utilizar de manera interactiva. En este apartado, dedicado a la línea de comandos con Bash, se muestra esta interacción desde el punto de vista de Bash, revisando los comandos más utilizados que Bash proporciona como `built-in` (incorporados).

Para obtener una lista de los comandos incorporados en Bash, ejecutamos `help` o `compgen -b` (alternativamente, `compgen -A builtin`):

```
$ compgen -b
.
:
[
alias
bg
bind
break
builtin
caller
cd
command
compgen
complete
compopt
continue
declare
dirs
disown
echo
enable
eval
exec
exit
export
false
fc
fg
getopts
hash
help
history
jobs
kill
let
local
logout
mapfile
popd
printf
pushd
pwd
read
readarray
readonly
return
set
shift
shopt
source
suspend
```

```
test
times
trap
true
type
typeset
ulimit
umask
unalias
unset
wait
```

El mismo `compgen` es un `built-in` de Bash que nos proporciona una lista de todos los comandos, alias y funciones disponibles. El argumento `-b` filtra la lista para mostrar solo los nombres de los comandos incorporados a Bash.

Consultaremos la descripción meticulosa de cada comando con `help [comando]`.

```
$ help pwd
pwd: pwd [-LP]
    Print the name of the current working directory.

    Options:
    -L    print the value of $PWD if it names the current working
          directory
    -P    print the physical directory, without any symbolic links

    By default, 'pwd' behaves as if '-L' were specified.

    Exit Status:
    Returns 0 unless an invalid option is given or the current directory
    cannot be read.
```

Por ejemplo, `pwd`[1] es el comando que nos dice cuál es el directorio de trabajo (directorio donde estamos situados).

(1)Proviene del acrónimo *Print Working Directory*.

```
$ pwd
/home/user
```

No debemos confundir los comandos incorporados a Bash con los comandos que también están disponibles a partir de los binarios del sistema operativo.

Por ejemplo, el comando `pwd` está disponible como `built-in`, como ya hemos visto, pero también está el comando `pwd` a nivel de sistema operativo.

```
$ which pwd
/usr/bin/pwd

$ file /usr/bin/pwd
/usr/bin/pwd: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0
```

Podemos ver que `pwd` también es un *binary* (ELF) en el que su intérprete es el *dynamic linker* en vez de Bash. El `ld-linux` se encarga de cargar las dependencias y ejecutar el binario, al contrario que los comandos incorporados, en los que la funcionalidad y la ejecución están dentro del mismo Bash.

Para más información, ved las páginas del man:

```
man ld-linux
man elf
```

Dado que algunos comandos están disponibles tanto vía binario del sistema operativo como vía `built-in` de Bash, tenemos la opción de habilitar o deshabilitar la versión `built-in` del comando deseado. En este ejemplo, mostraremos lo que sucede con el comando `kill`, en el que se producen claras diferencias en el formato de salida de los datos entre la versión del sistema operativo y la versión incorporada a Bash.

`kill` es un `built-in` de Bash:

```
$ type kill
kill is a shell builtin

$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
<...>
```

También disponemos del comando/programa `kill` del sistema operativo:

```
$ which kill
/usr/bin/kill

$ /usr/bin/kill -l
HUP INT QUIT ILL TRAP ABRT BUS FPE KILL USR1 SEGV USR2 PIPE ALRM TERM STKFLT
CHLD CONT STOP TSTP TTIN TTOU URG XCPU XFSZ VTALRM PROF WINCH POLL PWR SYS
```

Podemos observar que la salida del comando `kill` del sistema operativo es diferente a la de la versión provista por Bash. Si queremos ejecutar el comando `kill` proporcionado por el sistema operativo en lugar del `built-in` de Bash sin tener que especificar la ruta absoluta, podemos deshabilitar el `built-in`:

```
# Deshabilitamos el built-in del comando kill.
$ enable -n kill

# Ejecutamos kill y vemos que la salida es la de /usr/bin/kill.
$ kill -l
HUP INT QUIT ILL TRAP ABRT BUS FPE KILL USR1 SEGV USR2 PIPE ALRM TERM STKFLT
CHLD CONT STOP TSTP TTIN TTOU URG XCPU XFSZ VTALRM PROF WINCH POLL PWR SYS

# Habilitamos de nuevo el built-in.
$ enable kill

# Ejecutamos kill, ahora es el built-in.
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
<...>
```

2. *Shell Scripting* con Bash

Los ***shell scripts*** nos permiten ejecutar los comandos de manera no interactiva, es decir, no requieren la interacción del usuario.

Shell script

Un *shell script* es un fichero que contiene comandos y es interpretado por Bash.

Opcionalmente, un *script* puede contener en su inicio lo que se denomina *shebang*.

El *shebang* es el par de caracteres `#!`, que indican el intérprete que se va a utilizar para ejecutar el *script*.

El *shebang* más habitual de Bash es `#!/bin/bash`, aunque hay varias maneras de definirlo, por ejemplo, usando `env` si desconocemos la ruta absoluta del intérprete. Podemos ver que, dado que hay sistemas que no tienen el `bash` en la ruta `/bin`, muchos *scripts* definen el *shebang* como `#!/usr/bin/env bash` para ser más portables.

El *shebang* no es obligatorio para ejecutar el *script*, pero sí que facilita la identificación del tipo de intérprete esperado para ejecutarlo.

```
# Ejecución de un script sencillo con Bash sin shebang, pasándolo como argumento al intérprete (bash).

$ cat hola.sh
echo "¡hola, mundo!"

$ bash hola.sh
¡hola, mundo!
```

```
# Ejecución con shebang.
$ cat hola.sh
#!/bin/bash

echo "¡hola, mundo!"

$ chmod +x hola.sh
$ ./hola.sh
¡hola, mundo!
```

```
# Ejecución de un script sencillo con Bash sin shebang.

$ cat hola.sh
echo "¡hola, mundo!"

$ chmod +x hola.sh

$ ./hola.sh
¡hola, mundo!
```

En el último ejemplo, vemos que ejecutando directamente el *script* sin *shebang* funciona. El motivo es porque en la distribución GNU/Linux en la que se ha hecho la prueba, la *shell* por defecto es `bash`.

```
$ echo $0
```

```
bash
```

`$0` es una variable especial de Bash y, en este caso, su valor es el nombre de la *shell*.

Ved también

Ved el apartado «Variables especiales» del presente módulo.

Esta última manera de ejecutar el *script*, sin *shebang* y sin indicarlo, no es recomendable, ya que podemos tener efectos no deseados si ejecutamos una distribución que no tenga `bash` como *shell* por defecto y el *script* está programado con Bash. Una buena práctica es incluir el *shebang*.

Para ejecutar un *script* desde otro *script* se puede hacer de diferentes maneras, las más básicas son:

1) Con el comando (`builtin`) `source` (también se puede utilizar el comando punto «.»):

```
# Script b.sh que será llamado desde el script a.sh
$ cat b.sh
echo "script B. Argumentos: ${@}"

# Script a.sh, utilizando el comando source ejecutando el script b.sh
$ cat a.sh
#!/bin/bash
echo "script A"
source b.sh a b c

# Ejecución
$ ./a.sh
script A
script B. Argumentos: a b c
```

2) Ejecutándolo igual que se hace con la línea de comandos:

```
# Script b.sh que será llamado desde el script a.sh
$ cat b.sh
echo "script B. Argumentos: ${@}"

# Script a.sh, ejecuta el script b.sh
$ cat a.sh
#!/bin/bash
echo "script A"
bash b.sh a b c

# Ejecución
$ ./a.sh
script A
script B. Argumentos: a b c
```

3. Caracteres especiales, operadores básicos y palabras reservadas

Bash dispone de **caracteres especiales** que tienen un significado concreto. Su uso es reservado y el lenguaje reconoce estos caracteres y les aplica un comportamiento definido en lugar de su sentido literal.

Los caracteres especiales no tienen que ir entre comillas para poder ser interpretados con esa finalidad. Si aparecen entre comillas, se interpretan como valor literal.

este carácter sirve para comentar líneas. Bash ignorará cualquier texto posterior que esté en la misma línea.

```
$ # Este contenido será ignorado.
$ echo Hola # Este contenido será ignorado.
Hola
$    # Este contenido será ignorado.
$
```

El carácter «#» tiene un comportamiento diferente cuando está dentro de expresiones numéricas o de sustitución de cadenas de caracteres. Estos casos singulares los veremos más adelante.

\ (contrabarra). Carácter de escape. Conserva el valor literal del carácter que lo sucede. Si el carácter que lo sucede es un salto de línea, significa que la línea continúa.

```
$ echo "escape de comillas dobles \"\""
escape de comillas dobles ""
```

' (comilla simple). Los caracteres entre comillas simples ' ' se interpretan como literales.

```
$ echo 'abc\'
abc\
```

" (comillas dobles). Los caracteres entre comillas dobles " " se interpretan como literales a excepción de la barra inversa o contrabarra «\», el dólar «$» y las *backticks* «` `».

```
$ echo "abc\"
>
$ echo "abc\\"
abc\
```

3.1. Operadores de control

Realizan tareas de control de flujo.

`&` ejecuta el comando en segundo plano.

```
$ (sleep 10; echo "hola") &
[1] 29000
$ hola
[1]+  Done                    ( sleep 10; echo "hola" )
```

`;` separa diferentes comandos y se ejecutan en orden secuencial.

```
$ echo "1"; echo "2"
1
2
```

`;;` indica el final de la sentencia de control *case*.

`;&` continúa con la ejecución de los comandos de la disposición siguiente de la sentencia de control *case*.

`;;&` continúa con la disposición siguiente de la sentencia de control *case* y ejecuta los comandos si se cumple la condición.

Ved también

Ved el apartado «El condicional *case*» del presente módulo.

`(...)` agrupa comandos para ser ejecutados en una *subshell*. Su comportamiento es similar a utilizar `{...}`, pero en este último caso, no se ejecutan en ninguna *subshell*.

```
$ echo "Es subshell: $BASH_SUBSHELL"
Es subshell: 0
$ (echo "Es subshell: $BASH_SUBSHELL")
Es subshell: 1
```

`&&` operador lógico *AND*. Permite ejecutar un comando, pero solo si el anterior se ha ejecutado con éxito (código de regreso igual a cero).

```
$ true && echo "abc"
abc
$ false && echo "abc"
(sin salida)
```

`||` operador lógico *OR*. Ejecuta un comando, pero solo si el anterior se ha ejecutado sin éxito (código de regreso diferente de cero).

```
$ true || echo "abc"
(sin salida)
$ false || echo "abc"
abc
```

`|` pipa (*pipe - pipeline*). Envía la salida de un comando a la entrada del comando siguiente.

```
$ echo -e "2\n1\n3" | sort
```

```
1
2
3
```

3.2. Operadores de redirección

Redirigen la entrada y salida de los comandos.

< el comando obtiene la entrada del fichero proporcionado a la derecha del operador.

```
$ cat fichero.txt
abc
$ cat < fichero.txt
abc
```

> redirige la salida del comando a un fichero. Si el fichero existe, lo sobrescribe.

```
$ echo "abc" > fichero.txt
$ cat fichero.txt
abc
```

>| redirige la salida del comando a un fichero. A diferencia del anterior, sobrescribirá el fichero si existe, aunque la *shell* esté configurada para no sobrescribir ficheros existentes (*set* o *noclobber*).

Ved página del `man`:

```
man set
```

<< la entrada del comando es un *here document*.

```
$ cat <<EOF
> abc
> def
> EOF
abc
def
```

<<< la entrada es un *here string*. Similar al *here document*, pero de una línea.

```
$ cat <<< "abc"
abc
```

>> redirige la salida a un fichero, pero si este existe, agrega el contenido al final del fichero.

```
$ cat fichero.txt
abc
$ echo "def" >> fichero.txt
$ cat fichero.txt
abc
def
```

`>&` redirige la salida estándar y la salida de error a un fichero. Si el fichero existe, lo sobrescribe.

`&>` redirige la salida estándar y la salida de error a un fichero. Si el fichero existe, lo sobrescribe.

`>>&` redirige la salida estándar y la salida de error a un fichero. Si el fichero existe, agrega el contenido al final del fichero.

`&>>` redirige la salida estándar y la salida de error a un fichero. Si el fichero existe, agrega el contenido al final del fichero.

`|&` pipa (*pipe - pipeline*). Envía la salida estándar y la salida de error de un comando a la entrada del comando siguiente.

3.3. Operadores aritméticos

Realizan operaciones aritméticas (con números **enteros**).

Estos operadores se utilizan habitualmente con las expansiones aritméticas de Bash con `$((...))`. También se pueden utilizar con `backticks`, `expr` o con el `builtin let`, pero se recomienda usar las expansiones de Bash debido a su sencillez. El comando `expr` se puede utilizar por la compatibilidad con *shells* antiguas.

Ved también

Ved el apartado «Expansiones» del presente módulo.

`+` suma dos operandos.

```
$ echo $((1 + 2))
3
```

`-` resta dos operandos.

```
$ echo $((5 - -10))
15
```

`*` multiplica dos operandos.

```
$ echo $((-2 * -5))
10
```

`/` divide dos operandos.

```
$ echo $((4 / 2))
2
```

`%` módulo de dos operandos. Residuo de la división de los dos operandos.

```
$ echo $((3 % 2))
1
```

++ incrementa el operando en una unidad.

```
$ a=0
$ echo $((a++))
0
$ echo $a
1
$ echo $((++a))
2
$ echo $a
2
```

-- decrementa el operando en una unidad.

```
$ a=0
$ echo $((a--))
0
$ echo $a
-1
$ echo $((--a))
-2
$ echo $a
-2
```

Consultar los `man`:

```
man let
man expr
```

3.4. Operadores relacionales

Los **operadores relacionales** comparan dos valores.

== Compara si los dos operandos son iguales. El resultado es cierto si son iguales y falso en caso contrario.

!= Compara si los dos operandos son diferentes. El resultado es cierto si son diferentes y falso en caso contrario.

< Compara si el primer operando (el de la izquierda) es más pequeño que el segundo (el de la derecha). El resultado es cierto si el primer operando es más pequeño que el segundo. El resultado es falso en otro caso.

> Compara si el primer operando es más grande que el segundo. El resultado es cierto si el primer operando es más grande que el segundo. El resultado es falso si no lo es.

<= Compara si el primer operando es más pequeño o igual que el segundo. El resultado es cierto si el primer operando es más pequeño o igual que el segundo. El resultado es falso si pasa lo contrario.

>= Compara si el primer operando es más grande o igual que el segundo. El resultado es cierto si el primer operando es más grande o igual que el segundo. El resultado es falso si pasa lo contrario.

3.5. Operadores lógicos o booleanos

Realizan operaciones lógicas.

&&**Intersección o *AND* lógica**. El resultado es cierto si los dos operandos son ciertos. El resultado es falso si pasa lo contrario.

```
$ true && echo "abc"
abc
$ false && echo "abc"
$ echo $?
1
```

|| **Unión u *OR* lógica**. El resultado es cierto si uno de los dos operandos es cierto o los dos son ciertos. El resultado es falso si pasa lo contrario.

```
$ true || echo "abc"
$ false || echo "abc"
abc
```

! ∂**Negación**. Este operador es unario. El resultado es cierto si el operando es falso. El resultado es falso si pasa lo contrario.

3.6. Operadores bit a bit

Realizan operaciones bit a bit.

& Intersección o *AND* bit a bit.

```
$ echo $((2#0101 & 2#0011))
1
```

| Unión u *OR* bit a bit.

```
$ echo $((2#0101 | 2#0011))
7
```

^*OR* exclusiva o *XOR* bit a bit.

```
$ echo $((2#0101 ^ 2#0011))
6
```

~*NOT* bit a bit.

```
$ echo $((~2#0011))
-4
```

<< Desplazamiento aritmético hacia la izquierda.

```
$ echo $((2#0011<<2))
12
```

>> Desplazamiento aritmético hacia la derecha.

```
$ echo $((2#1100>>2))
3
```

Adicionalmente, Bash tiene un compendio de **palabras reservadas** del lenguaje que forman parte de su léxico y solo pueden ser utilizadas para la construcción de comandos. Estas palabras reservadas son:

```
$ compgen -k
if
then
else
elif
fi
case
esac
for
select
while
until
do
done
in
function
time
{
}
!
[[
]]
```

Veremos el uso de estas palabras reservadas en apartados posteriores.

4. Expansiones

Durante la fase de análisis del léxico, el intérprete de Bash divide los comandos con palabras reconocidas. En este proceso se detectan patrones y caracteres internos de expresiones que se sustituyen por valores.

Referencia

Ved el enlace «3.5 Shell Expansions».

4.1. Expansión de llaves

La **expansión de llaves** es la que se produce en primer lugar. Las expansiones de llaves "`{...}`" sustituyen la expresión por las cadenas de caracteres generadas.

Para no interpretarlo como literal, la expresión no tiene que estar entre comillas.

```
$ echo "{a,b,c}{1,2,3}"
{a,b,c}{1,2,3}

$ echo {a,b,c}{1,2,3}
a1 a2 a3 b1 b2 b3 c1 c2 c3
```

4.2. Expansión de las virgulillas

> Expande el carácter «~» y los sucesivos hasta la primera barra oblicua «/». Si esta no aparece, se tratan todos los caracteres en la expansión.

El resultado de la expansión vendrá determinado en función de los caracteres que sucedan a la virgulilla. Los ejemplos más habituales son:

- ~ valor de `$HOME` si está definida en el directorio *home* del usuario.
- ~+ valor de `$PWD`.
- – valor de `$OLDPWD`.

```
# Expansión de ~ por $HOME.
$ HOME=/tmp
$ echo ~
/tmp

# Expansión de ~ por la home del usuario cuando $HOME no está definido.
$ unset HOME
$ echo ~
/home/user

# Expansión de ~+ por $PWD.
$ pwd
/tmp
$ echo ~+/abc
/tmp/abc

# Expansión de ~- por $OLDPWD.
```

```
$ pwd
/home/user
$ cd /tmp/
$ echo ~-
/home/user
```

4.3. Expansión de parámetros

La expansión de parámetros es del tipo `${parametro}` o `$parametro`.

Las llaves son obligatorias cuando «parametro» es un parámetro posicional de más de un dígito (por ejemplo `${10}`) o cuando queremos indicar que hay caracteres después del «parametro» que no forman parte del mismo nombre. Cuando se hace la expansión, «parametro» es sustituido por su valor.

Ved también

Ved el apartado «Variables especiales» del presente módulo.

```
$ echo $USER
user
$ a=(1 2 3)
$ echo ${a[@]}
1 2 3
```

4.4. Expansión de comandos

Se sustituye el comando de la expresión por su salida después de ejecutarlo en una *subshell*.

```
$ echo "$(whoami)"
user
```

4.5. Expansión aritmética

Se sustituye por el resultado de la operación aritmética.

```
$ echo $((1 + 2))
3
```

4.6. Sustitución de procesos

Sustituye la entrada o la salida de un comando vía ficheros, de forma que la salida o la entrada del proceso es la entrada o salida del otro.

```
$ echo <(:)
/dev/fd/63
$ echo >(:)
/dev/fd/63
```

4.7. División de palabras

La división de palabras utiliza un delimitador para poderlas separar.

Este delimitador está marcado por un carácter y su valor es el de la variable `$IFS`.[2] El valor por defecto del `$IFS` es espacio, tabulador y nueva línea (`\t \n`).

[2] Proviene del acrónimo *Internal Field Separator*.

```
$ printf "%q" "$IFS"
$' \t\
```

4.8. Expansión de nombre de ficheros

También conocido como *globbing*, la expansión del nombre de los ficheros busca los caracteres asterisco (`*`), corchetes (`[...]`) o el signo de interrogación (`?`).

Si contiene alguno de estos caracteres, la palabra se trata como patrón y se utiliza para obtener una lista de los ficheros que coincidan con este patrón.

```
$ ls
fichero1  fichero11  fichero2  fichero3  LÉEME  script.sh
$ ls fichero?
fichero1  fichero2  fichero3
$ ls *[1-2]
fichero1  fichero11  fichero2
$ ls *[^1-2]
fichero3  LEEME  script.sh
$ ls s*
script.sh
```

Para más información, ved la página del `man`:

```
man 7 glob
```

4.9. Eliminación de comillas

Es la última expansión, elimina las comillas ('), las comillas dobles (") y la barra inversa (\ ) que no sean resultado de las expansiones.

5. Variables

Las **variables** son una de las partes más importantes de los *scripts*. Durante la ejecución de un *script*, podemos necesitar almacenar datos temporalmente para que se puedan tratar con posterioridad. Estos datos se pueden guardar en variables.

Con el uso de variables, podemos tomar decisiones en nuestro *script* en función del valor que tienen, es decir, podemos dotar a nuestro *script* de comportamientos en función de las variables.

Cuando se define una variable con Bash no es obligatorio definir el tipo. El tipo básico es el *string*. A pesar de que la variable contenga solo dígitos, podemos realizar operaciones aritméticas básicas con enteros.

```
# Entero
$ a=1234
$ echo $((a + 1))
1235

# String
$ a=$a"ABC"
$ echo $a
1234ABC
```

También se puede declarar el tipo de variable con el `built-in declare`.

Para más información, ved la ayuda con el `built-in help`:

```
help declare
```

5.1. Variables de entorno

Además de las variables que podemos definir dentro de un *script*, con Bash también están las llamadas *variables de entorno*.

Las **variables de entorno** tienen la finalidad de configurar el entorno en el que estamos ejecutando Bash.

Las variables de entorno pueden ser globales (exportadas) o locales. La diferencia principal entre ellas es el ámbito. Las **variables globales** son visibles para todos los subprocesos que cree la *shell*, mientras que las **variables locales** solo son visibles desde la *shell* que las ha definido.

Para ver el valor de las variables de entorno globales, podemos hacerlo con el comando `printenv`:

```
$ printenv
SHELL=/bin/bash
<...>
LANGUAGE=_OS:en
PWD=/home/user
LOGNAME=user
HOME=/home/user
LANG=_OS.UTF-8
<...>
PATH=/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
```

El comando `env` también nos mostrará las variables de entorno:

```
$ env
SHELL=/bin/bash
<...>
LANGUAGE=_OS:en
PWD=/home/user
LOGNAME=user
HOME=/home/user
LANG=_OS.UTF-8
<...>
PATH=/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
```

Tanto `printenv` como `env`, cuando los ejecutamos sin argumentos, muestran las variables de entorno. Adicionalmente, el comando `env`, cuando le indicamos argumentos, permite asignar variables de entorno antes de ejecutar cualquier *script*.

Si queremos mostrar el valor de una variable en concreto, podemos usar `echo` o `printenv`:

```
$ echo $HOME
/home/user

$ printenv HOME
/home/user
```

`echo` requiere que le indiquemos el símbolo del dólar (`$`) con el nombre de la variable para que Bash internamente le indique el valor de la variable como argumento para poderlo mostrar. `echo` tiene el propósito general de mostrar líneas de texto por pantalla y no de resolver valores de variables.

El comando `printenv` sí que está diseñado especialmente para resolver nombres de variables y es capaz de resolver su valor internamente vía la función `getenv` de la `stdlib` de `glibc`.

Las variables (y métodos) locales las podemos ver con el `built-in set`:

```
$ set
BASH=/usr/bin/bash
BASH_ALIASES=()
BASH_ARGC=([0]="0")
<...>
```

5.2. Variables de usuario

Podemos crear nuestras variables con el valor que deseemos realizando una asignación de la manera siguiente:

variable=valor

Donde *variable* es el nombre de la variable y *valor* es su valor.

Por ejemplo:

```
$ var="mundo"
$ echo "Hola, $var!"
¡Hola, mundo!
```

La variable `var` declarada es local y su ámbito es la *shell* actual y sus *subshells*:

```
$ var="valor"
$ echo "Valor de var shell ($BASH_SUBSHELL) actual: $var"; (echo "Valor de var subshell
($BASH_SUBSHELL): $var")
Valor de var shell (0) actual: valor
Valor de var subshell (1): valor
```

Vemos que en este caso tanto la *shell* como la *subshell* (creada con los paréntesis «( )») tienen definida la variable `var`. `BASH_SUBSHELL` es una variable interna de Bash que nos dice si la *shell* es una *subshell* o no. Retorna el valor «0» para indicarnos que no es una *subshell* o el valor «1» para indicar que sí que lo es. El punto y coma («;») lo usamos para separar diferentes comandos.

En cambio, si abrimos una nueva *shell* (subproceso de la *shell* actual), veremos que no existe:

```
$ var="valor"
$ bash
$ echo $var

(salida en blanco)
```

Para que esté disponible en una nueva *shell*, tenemos que declararla como variable global. Para hacerlo, utilizamos el comando `export`:

```
$ var="valor"
$ export var
$ bash
$ echo $var
valor
```

Cuando la exportamos como variable global, todos los subprocesos creados por la *shell* actual tendrán definida la variable.

También podemos asignar el valor de una variable a partir de la salida de un comando. Bash soporta las *backticks* (`` ` ` ``) o `$()`.

```
$ var=$(echo "valor1")
$ echo $var
```

```
valor1
$ var=`echo "valor2"`
$ echo $var
valor2
```

Para definir una variable de solo lectura:

```
$ readonly var="valor"
$ echo $var
valor
$ var="nuevo valor"
bash: var: readonly variable
```

Para eliminar una variable, utilizamos el `built-in unset`:

```
$ var="valor"
$ echo $var
valor
$ unset var
$ echo $var

(salida en blanco)
```

Las variables declaradas como solo lectura no pueden ser eliminadas.

5.3. Variables especiales

Bash proporciona una serie de variables especiales relacionadas con los parámetros para ser utilizadas en los *scripts*. Estas variables son solo de lectura para el usuario. Internamente están las *built-ins*, que sí que modifican el valor de algunas de estas, por ejemplo, el comando `dshift` (que modifica un *array* interno llamado `dollar_vars`).

- `$@`: *array* con todos los argumentos.
- `$*`: cadena de caracteres con todos los argumentos.
- `$#`: número de argumentos.
- `$_`: nombre del *script* y último argumento del comando ejecutado anteriormente.
- `$-`: opciones habilitadas vía *set* de la *shell*.
- `$?`: valor del código de estado.
- `$$`: PID de la *shell*.
- `$!`: PID del último proceso lanzado al *background*.
- `$n`: argumentos posicionales. `$0` corresponde al nombre del comando. `$1` a `$9` son los argumentos. Para acceder a partir del décimo argumento (incluido), tenemos que hacerlo con `${n}`, es decir, `${10}`, `${11}`, etc.

Ved también

Ved los apartados «*Arrays*» y «Funciones» del presente módulo.

6. *Arrays*

Un ***array*** es una estructura de datos que permite almacenar varios valores a los que se accede a partir de un índice.

Con Bash, podemos acceder a los elementos del *array* de manera global o individual.

```
# Inicialización.
$ array=(a b c d e f)

# Acceso global, muestra todos los elementos del array.
$ echo ${array[@]}
a b c d e f
$ echo ${array[*]}
a b c d e f

# Acceso individual, con el índice se accede a la posición del elemento.
$ echo ${array[5]}
f

# Medida del array.
$ echo ${#array[@]}
6
```

Como se puede observar, un *array* es una variable que contiene seis elementos (`a`, `b`, `c`, `d`, `e`, `f`). Para acceder a todos los elementos del *array*, lo hacemos con el índice `@` o `*`. Para acceder a cada elemento individual del *array*, lo hacemos con los índices numéricos. Con Bash, el primer elemento del *array* empieza por el índice cero (`0`) y no el uno (`1`). Obtenemos la medida del *array* con el operador `#`.

También podemos inicializar un *array* posición por posición (o con `declare`), eliminar cualquier elemento individual del *array*, añadir elementos nuevos o acceder a determinadas posiciones vía un índice y un desplazamiento:

```
# Inicialización.
$ array[0]='a'
$ array[1]='b'
$ array[2]='c'

# Mostramos todos los elementos.
$ echo ${array[@]}
a b c

# Eliminamos el elemento de la posición 1.
$ unset array[1]
$ echo ${array[@]}
a c

# Debemos tener presente que cuando eliminamos un elemento de una posición del array, el resto del
array se mantiene intacto. Los elementos mantienen su índice.
$ echo ${array[0]} ${array[2]}
a c
$ array[1]='b'
```

```
$ echo ${array[@]}
a b c

# Añadimos un elemento haciendo uso del índice.
$ array[3]='de
$ echo ${array[@]}
a b c d

# Añadimos varios elementos a la vez.
$ array=(${array[@]} e f)
$ echo ${array[@]}
a b c d e f

# Elementos del tercero al quinto.
$ echo ${array[@]:2:3}
c d e

# Último elemento. Notad el espacio en blanco después de ':'.
$ echo ${array[@]: -1}
f

# Sustituimos todos los elementos "b" por "c".
$ echo ${array[@]/b/c}
a c c d e f

# Borramos el array.
$ unset array
$ echo ${array[@]}

(salida en blanco)
```

7. Funciones

Las **funciones** son bloques de código que realizan unas determinadas acciones y pueden ser llamadas desde otras partes del código y las veces que se quiera.

Hay dos maneras para declarar una función:

1) Especificar la palabra reservada `function` en su firma.
2) Utilizar paréntesis en su firma.

```
# Definimos la función "hola" empleando function en la firma.
$ function hola {
 echo "¡Hola, mundo!"
 }

# Definimos la función "adios" sin emplear function en la firma. Usamos paréntesis
para indicar que es una función.
$ adios() {
 echo "Adiós!"
 }

# Ejecutamos las funciones.
$ hola
¡Hola, mundo!
$ adios
Adiós!
```

El paso de parámetros a una función es posicional y se hace durante su llamamiento:

```
$ h="Hola"
$ hola() {
 echo "$1, $2!"
 }
$ hola $h "mundo"
¡Hola, mundo!
```

Dentro de las funciones también se pueden definir variables de ámbito local:

```
$ f() {
local var=2
echo "valor de la variable \$var desde la función ${FUNCNAME[0]}(): $var"
}
$ var=1
$ f
valor de la variable $var desde la función f(): 2
$ echo "valor de la variable \$var: $var"
valor de la variable $var: 1
```

Además de recibir argumentos, las funciones con Bash pueden devolver códigos de estado.

Para más información, ved:

```
man exit
```

Los **códigos de estado** son números enteros [0-255] que sirven para indicar, generalmente, si el resultado de un comando (o función) se ha ejecutado con éxito o no.

Cuando un comando retorna un código de estado diferente de cero, significa que ha habido un error en la ejecución del comando. Si el código de estado retornado es cero, significa que el comando se ha ejecutado con éxito.

El código de estado retornado se puede consultar con la variable especial `$?`, que almacena el valor de retorno del último comando ejecutado.

No es obligatorio devolver explícitamente un código de estado en una función. Si no lo especificamos, Bash ejecutará la función, y si se ejecuta con éxito o no, le asignará el código de retorno correspondiente.

```
# Creamos dos funciones, "f()" y "ff()". La primera se ejecutará con éxito y la segunda con error.
$ f() {
 echo "f"
 }
$ ff() {
 comando_no_existente
 }

# Llamamos a las funciones y consultamos el código de estado de salida.
$ f
f
$ echo $?
0
$ ff
bash: comando_no_existente: command not found
$ echo $?
127
```

Los diferentes códigos de estado de varios comandos con *pipes* se pueden comprobar con la variable interna `PIPESTATUS`.

```
$ true | false | false | true
$ echo ${PIPESTATUS[@]}
0 1 1 0
```

Para devolver un código de estado, se utiliza la palabra reservada `return`.

```
$ f() {
 # Cuerpo de la función.
 return 1
 }
$ f
$ echo $?
1
```

Con Bash no podemos devolver valores que no sean numéricos desde las funciones para que sean tratados como códigos de estado. Si queremos devolver valores desde las funciones, podemos utilizar variables de diferentes maneras:

```
# En este ejemplo, se utiliza una variable desde el código que llama a la función para guardar
el valor de retorno.

$ f() {
 echo "valor"
 }

$ v=$(f)
$ echo "$v"
valor

# También directamente.
echo "$(f)"
valor

# En este ejemplo, se utiliza una variable definida en la función para guardar el valor de retorno
y es leída por el código que llama a la función.

$ f() {
 v="valor"
 }

$ echo "$v"
valor

# En este ejemplo, se aprovecha la variable especial $? de código de estado para retornar un
resultado entero. Este método es ilustrativo y no es recomendable. Para utilizar este
método, se tienen que establecer unas pautas sobre los códigos de estado, puesto que se está
aprovechando este método para retornar un resultado que está fuera del propósito de los
códigos de estado.

$ f() {
 return 22
 }

$ f
$ echo $?
22

# Con el último ejemplo, podemos tener efectos no deseados en función del escenario.

$ f() {
 # Código que calcula varias operaciones matemáticas.
 # Retornamos un valor que supondremos dinámico.
 return 350
 }

$ f
$ echo $?
94

# Se ha obtenido un valor no esperado y diferente al retornado por la función. Dado que se están
aprovechando los códigos de estado para retornar valores, estos están dentro del rango [0, 255].
Cualquier valor fuera del rango (como el 350), implica que se retornará al módulo del código
de estado para 256 que es el rango definido, es decir, 350 mod 256 = 94.
```

8. Sentencias condicionales

8.1. La sentencia `if`

La sentencia `if` decide cuándo se ejecutará o no una acción o grupos de acciones determinadas.

La decisión de ejecutarse o no viene determinada por una expresión lógica. Si el resultado de la expresión es cierta, las acciones se ejecutarán. En caso contrario, no se ejecutarán y se pasará a evaluar la expresión lógica siguiente o a ejecutar la alternativa si existe.

Las diferentes sintaxis del condicional `if` son las siguientes:

1) Sentencia `if`

Evalúa la condición y, si se cumple, ejecuta el bloque de código.

```
if condición; then
  # Código que se va a ejecutar si se cumple la condición.
fin
```

2) Sentencia `if/else`

Evalúa la condición. Si se cumple, ejecuta el bloque de código de dentro de la `if`; si no se cumple, ejecuta el bloque de código de dentro de la `else`.

```
if condición; then
  # Código que se va a ejecutar si se cumple la condición.
else
  # Código que se va a ejecutar si no se cumple la condición.
fin
```

3) Sentencia `if/elif/else`

Evalúa la condición de la `if` y si se cumple, ejecuta su bloque de código. Si la primera condición no se cumple, se evalúa la condición `elif` siguiente. Si se cumple, se ejecuta el bloque de código de la `elif`. Si no se cumple ninguna de las condiciones anteriores, se ejecuta el código de dentro de la `else` (si existe).

```
if condición; then
  # Código que se va a ejecutar si se cumple la condición if.
elif condición; then
  # Código que se va a ejecutar si se cumple la condición elif.
else
  # Código que se va a ejecutar si no se cumple ninguna condición anterior.
fin
```

8.1.1. Formato de las condiciones

Las **condiciones** tienen que tener un formato específico. Actualmente, hay tres formatos:

1) Condiciones con corchetes simples

Tienen la forma siguiente:

```
if [ condición ]; then
<...>
fin
```

Esta es la versión portable a otras *shells* POSIX.

Soporta tres tipos de condiciones:

- Condiciones basadas en cadenas de caracteres.
- Condiciones basadas en números (aritméticas).
- Condiciones basadas en ficheros.

Ved también

Cada uno de los diferentes tipos de condiciones se detallan en el apartado «Tipo de condiciones».

2) Condiciones con corchetes dobles

```
if [[ condición ]]; then
<...>
fin
```

Esta es la versión mejorada del condicional y es soportada por Bash. De entre las mejoras, se pueden destacar:

- Soporte de expansiones en cadenas de caracteres y ficheros.
- Opción de poner las cadenas de caracteres entre comillas.
- Permite combinar expresiones dentro de una misma condición.

3) Condiciones con paréntesis

```
if ( comando ); then
```

```
<...>
fin
```

Estos condicionales comprueban el código de estado retornado después de ejecutar el comando en una *subshell*.

4) Condiciones con doble paréntesis

```
if (( condición )); then
<...>
fin
```

Estos condicionales comprueban los resultados aritméticos.

5) Condiciones sin signos de puntuación

```
if comando; then
<...>
fin
```

6) Unión e intersección de condiciones

```
if [[ "abc" != "cde" && "abc" == "abc" ]]; then
<...>
fin

if [[ "abc" == "cde" || "abc" == "abc" ]]; then
<...>
fin
```

8.1.2. Tipos de condiciones

Las condiciones pueden estar compuestas por diferentes tipos de expresiones a evaluar.

1) Condiciones de cadenas de caracteres

Las **condiciones basadas en cadenas** de caracteres evalúan expresiones sobre sus propiedades, por ejemplo, la longitud, el orden alfabético o la igualdad entre ellas.

```
if [[ "abc" == "cde" ]]; then
<...>
fin
```

Las expresiones utilizadas en cadenas de caracteres son:

- <: la cadena de caracteres de la izquierda tiene precedencia alfabética sobre la cadena de caracteres de la derecha.

- >: la cadena de caracteres de la derecha tiene precedencia alfabética sobre la cadena de caracteres de la izquierda.

- =: las cadenas de caracteres son iguales. También es compatible con patrones.

- ==: igual que la anterior.

- !=: las cadenas de caracteres son diferentes.

- =~: la cadena de caracteres coincide con la expresión regular.

```
# Compara si "abc" es igual a "abc". El resultado es cierto.
if [[ "abc" == "abc" ]]; then
  echo "iguales"
fin

# Compara si "abc" es diferente a "cde". El resultado es cierto.
if [[ "abc" != "cde" ]]; then
  echo "diferentes"
fin

# La cadena "ab" precede alfabéticamente a "ac".
if [[ "ac" < "ab" ]]; then
  echo "\"ac\" tiene precedencia"
else
  echo "\"ab\" tiene precedencia"
fin
```

2) Condiciones de números

Las **condiciones basadas en números enteros** evalúan expresiones sobre las propiedades de estos números, como, por ejemplo, la igualdad o cuál de los dos números es menor o mayor.

```
if [[ 1 -ne 2 ]]; then
<...>
fin
```

Las expresiones utilizadas en números son:

- -eq compara si los dos números son iguales.

- -ne compara si los dos números no son iguales.

- -gt compara si el número de la izquierda es mayor que el número de la derecha.

- -lt compara si el número de la izquierda es menor que el número de la derecha.

- `-ge` compara si el número de la izquierda es mayor o igual que el número de la derecha.

- `-le` compara si el número de la izquierda es menor o igual que el número de la derecha.

```
# Números iguales.
if [[ 1 -eq 001 ]]; then
  echo "iguales"
fin

# 4 es más grande que 1.
if [[ 4 -gt 1 ]]; then
  echo "4 es mayor"
fin
```

3) Condiciones de ficheros

Las **condiciones basadas en ficheros** evalúan expresiones sobre las propiedades de los ficheros. Estas propiedades pueden ser la existencia y el tipo de fichero, entre otras.

```
if [[ -e fichero ]]; then
<...>
fin
```

Hay muchas expresiones sobre ficheros. Se recomienda consultar la página del `man` para poder tener una lista completa de estas. A continuación se enumeran algunas:

- `-e` comprueba si el fichero existe.
- `-d` comprueba si el fichero existe y si es un directorio.
- `-r` comprueba si el fichero existe y si tiene permisos de lectura.
- `-w` comprueba si el fichero existe y si tiene permisos de escritura.
- `-x` comprueba si el fichero existe y si tiene permisos de ejecución.

```
if [[ -e fichero1.txt ]]; then
  echo "existe"
fin
```

Para más información, ved la página del `man`:

```
man test
```

8.2. El condicional `case`

El **condicional `case`** se utiliza habitualmente para simplificar cuando se tienen muchos condicionales posibles y esto comporta tener muchas sentencias `if`/`else` anidadas.

La sintaxis del condicional `case` es la siguiente:

```
case expresión in
condición 1)
  # Código a ejecutar para el caso1.
  ;;
condición N)
  # Código a ejecutar para el casoN.
  ;;
esac
```

- `;;` indica el final de la sentencia de control `case`.

- `;&` continúa con la ejecución de los comandos de la disposición siguiente de la sentencia de control `case`.

- `;;&` continúa con la disposición siguiente de la sentencia de control `case` y ejecuta los comandos si se cumple la condición.

```
$ case "abc" in
abc)
 echo "abc"
 ;;
cde)
 echo "cde"
 ;;
esac
```

```
case "abc" in
abc)
 echo "abc";
 ;;&
cde)
 echo "cde";
 ;;
ab*)
 echo "ab*";
 ;;
esac
```

9. Sentencias iterativas

9.1. La sentencia `for`

La sentencia **`for`** la usamos para definir bucles que ejecutarán una secuencia de comandos un número determinado de veces.

Su sintaxis es la siguiente:

```
for expresión; do
  # Código que se va a ejecutar.
done
```

La expresión estándar es:

```
for element in [LISTA]; do
  # Código que se va a ejecutar.
done
```

Por ejemplo:

```
# Iterar sobre una lista de elementos.
for elem in a b c; do
  echo $elem
done

# Iterar sobre un rango de números.
for num in {0..10}; do
  echo $num
done

# Iterar sobre un rango de números con incremento
for num in {0..10..2}; do
  echo $num
done
```

La expresión también puede ser del estilo de C:

`for` (inicialización; condición de finalización; incremento)

Por ejemplo:

```
for ((i=0; i<=10; i++)); do
  echo $i
done
```

9.2. La sentencia `while`

Usamos la sentencia `while` para definir bucles que ejecutarán una secuencia de comandos mientras se cumpla una condición.

Su sintaxis es la siguiente:

```
while [[ condición ]]; do
  # código que se va a ejecutar
done
```

Por ejemplo:

```
i=0
while [[ i -lt 10 ]]; do
 echo $((i++))
done
```

9.3. La sentencia `until`

La sentencia **`until`** es igual que la sentencia `while` pero con la única diferencia de que su condición es negada, es decir, el bucle se ejecuta mientras la condición no se cumpla.

Su sintaxis es la siguiente:

```
until [[ condición ]]; do
  # Código que se va a ejecutar.
done
```

Ejemplo:

```
i=10
until [[ i -lt 1 ]]; do
 echo $((--i))
done
```

9.4. Las sentencias `break` y `continue`

Las **sentencias `break` y `continue`** pueden ser utilizadas en los bucles `for`, `while` y `until` para salir del bucle en el caso de `break` o para saltar la parte de código restante del bucle en el caso de `continue`.

Ejemplo de uso de `break`:

```
# Bucle infinito del cual se sale con break cuando el valor de i es 5.
i=0
while true; do
  echo $i
  if [[ $((++i)) -eq 5 ]]; then
```

```
    break
  fin
done
```

Ejemplo de uso de `continue`:

```
# Bucle en el que en la segunda iteración no imprime la segunda cadena debido al continue.
for i in {1..3}; do
  echo "[$i] primera"
  if [[ $i -eq 2 ]]; then
    continue
  fin
  echo "[$i] segunda"
done
```

10. Depuración de *scripts*

Para poder **depurar (o *debuggar*) *scripts*** con Bash, se habilita el modo de depuración ejecutando el *script* con la opción `-x`.

Por ejemplo, el *script* sencillo siguiente:

```
#!/bin/bash

echo "prueba de depuración"
echo "ejecutamos date: $(date -d "01/01/2019")"
echo "fin"
```

Y su ejecución:

```
$ ./debug.sh
prueba de depuración
ejecutamos date: Tue 01 Jan 2019 12:00:00 AM CET
fin
```

Con el modo de depuración se muestran trazos adicionales para cada comando ejecutado por el *script*:

```
$ bash -x debug.sh
+ echo 'prueba de depuración'
prueba de depuración
++ date -d 01/01/2019
+ echo 'ejecutamos date: Tue 01 Jan 2019 12:00:00 AM CET'
ejecutamos date: Tue 01 Jan 2019 12:00:00 AM CET
+ echo fin
fin
```

Para depurar partes concretas del *script*, se habilita con *set* el modo entre las líneas que se quiera depurar:

```
#!/bin/bash

echo "prueba de depuración"
set -x
echo "ejecutamos date: $(date -d "01/01/2019")"
set +x
echo "fin"
```

Durante la ejecución se imprimen únicamente los trazos de las líneas entre el *set*:

```
$ ./debug.sh
prueba de depuración
++ date -d 01/01/2019
+ echo 'ejecutamos date: Tue 01 Jan 2019 12:00:00 AM CET'
ejecutamos date: Tue 01 Jan 2019 12:00:00 AM CET
+ set +x
fin
```

Bibliografía

Cooper, M. (2011). *Advanced Bash-Scripting Guide*. Autoedición.

Free Software Foundation (2019). *Bash Reference Manual*. Boston, Massachusetts: Free Software Foundation.

GoalKicker (2018). «Bash Notes for Professionals». Goalkicker.com.

Shotts, W. (2017). «The Linux Command Line» (4.ª ed.). Linuxcommand.org.