Introducció a Bash

PID_00270621

Àngel Ollé Blázquez

Temps mínim de dedicació recomanat: 4 hores

Universitat Oberta de Catalunya

Àngel Ollé Blázquez

Enginyer tècnic en Informàtica de gestió per la Universitat Rovira i Virgili (URV). Enginyer en Informàtica i màster en Programari Lliure per la Universitat Oberta de Catalunya (UOC). Actualment treballa com a enginyer de programari i participa en projectes *open source*. Anteriorment ha desenvolupat la seva activitat professional en el sector de serveis i consultoria IT per EMEA.

L'encàrrec i la creació d'aquest recurs d'aprenentatge UOC han estat coordinats pel professor: Julià Minguillón Alfonso (2020)

Primera edició: febrer 2020

Av. Tibidabo, 39-43, 08035 Barcelona
Realització editorial: FUOC

Índex

Introducció

Bash (*Bourne-again Shell*) és una de les *shells* o línia de comandes més populars que hi ha en la majoria de distribucions GNU/Linux.

Shell

Una *shell* és una aplicació que permet executar comandes del sistema operatiu.

Bash inclou millores pròpies, però també té característiques d'altres *shells* populars com la **Korn Shell** i la **C Shell**, a la vegada que és compatible amb la de la seva predecessora, la **Bourne Shell**.

En aquest mòdul es presenten els conceptes bàsics de les parts més importants de Bash, començant per conèixer la línia de comandes i què és un *script*. Veurem els caràcters especials del llenguatge, els operadors i les expansions, detallant el seu ús i les seves qualitats, i els diferents tipus de variables per a emmagatzemar les dades. A més, el mòdul fa una introducció a les sentències fonamentals, com ara les sentències condicionals i les sentències iteratives. Finalment, es mostra com depurar el codi desenvolupat.

1. Línia de comandes amb Bash

En el mòdul «Introducció a GNU/Linux» s'ha vist la línia de comandes amb les instruccions més típiques que s'acostumen a utilitzar de manera interactiva. En aquest apartat, dedicat a la línia de comandes amb Bash, es mostra aquesta interacció des del punt de vista de Bash, revisant les comandes més utilitzades que Bash proporciona com a `built-in` (incorporades).

Per a obtenir una llista de les comandes incorporades a Bash, executem `help` o `compgen -b` (alternativament, `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 mateix `compgen` és un `built-in` de Bash que ens proporciona una llista de totes les comandes, àlies i funcions disponibles. L'argument `-b` filtra la llista per a mostrar només els noms de les comandes incorporades a Bash.

Consultarem la descripció acurada de cada comanda amb `help [comanda]`.

```
$ 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.
```

Per exemple, `pwd`[1] és la comanda que ens diu quin és el directori de treball (directori on estem situats).

(1)Prové de l'acrònim *Print Working Directory*.

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

No hem de confondre les comandes incorporades a Bash amb les comandes que també estan disponibles a partir dels binaris del sistema operatiu.

Per exemple, la comanda `pwd` està disponible com a `built-in`, com ja hem vist, però també hi ha la comanda `pwd` a nivell de sistema operatiu.

```
$ 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
```

Podem veure com `pwd` també és un *binary* (ELF) on el seu intèrpret és el *dynamic linker* en lloc de Bash. L'`ld-linux` s'encarrega de carregar les dependències i executar el binari, al contrari de les comandes incorporades, en què la funcionalitat i l'execució estan dintre del mateix Bash.

Per a més informació, vegeu les pàgines del man:

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

Com que algunes comandes estan disponibles tant via binari del sistema operatiu com a `built-in` de Bash, tenim l'opció d'habilitar o deshabilitar la versió `built-in` de la comanda desitjada. En aquest exemple, mostrarem les diferències amb la comanda `kill`, en què hi ha clares diferències amb el format de sortida de les dades entre la versió del sistema operatiu i la versió incorporada a Bash.

`kill` és 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
<...>
```

També tenim disponible la comanda/programa `kill` del sistema operatiu:

```
$ 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
```

Podem observar que la sortida de la comanda `kill` del sistema operatiu, té una sortida diferent de la versió proveïda per Bash. Si volem executar la comanda `kill` proporcionada pel sistema operatiu en lloc del `built-in` de Bash sense tenir que especificar la ruta absoluta, podem deshabilitar el `built-in` amb:

```
# Deshabilitem el built-in de la comanda kill.
$ enable -n kill

# Executem kill i veiem que la sortida és 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

# Habilitem de nou el built-in.
$ enable kill

# Executem kill, ara és 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* amb Bash

Els ***shell scripts*** ens permeten executar les comandes de manera no interactiva, és a dir, no requereixen la interacció de l'usuari.

Shell script

Un *shell script* és un fitxer que conté comandes i és interpretat per Bash.

Opcionalment, un *script* pot contenir al principi d'aquest el que s'anomena *shebang*.

El *shebang* és el parell de caràcters `#!` que indiquen l'intèrpret a utilitzar per a executar l'*script*.

El *shebang* més habitual de Bash és `#!/bin/bash`, tot i que hi ha diverses maneres de definir-lo, per exemple, fent servir `env` si desconeixem la ruta absoluta de l'intèrpret. Podem veure que, a causa que hi ha sistemes que no tenen el `bash` a la ruta `/bin`, molts *scripts* defineixen el *shebang* com a `#!/usr/bin/env bash` per ser més portables.

El *shebang* no és obligatori per a executar l'*script*, però sí que facilita la identificació del tipus d'intèrpret esperat per a executar-lo.

```
# Execució d'un script senzill amb Bash sense shebang, passant-lo com a argument a l'intèrpret (bash).

$ cat hola.sh
echo "hola, món!"

$ bash hola.sh
hola, món!
```

```
# Execució amb shebang.
$ cat hola.sh
#!/bin/bash

echo "hola, món!"

$ chmod +x hola.sh
$ ./hola.sh
hola, món!
```

```
# Execució d'un script senzill amb Bash sense shebang.

$ cat hola.sh
echo "hola, món!"

$ chmod +x hola.sh

$ ./hola.sh
hola, món!
```

En el darrer exemple, veiem que executant directament l'*script* sense *shebang* funciona. El motiu és perquè en la distribució GNU/Linux en què s'ha fet la prova, la *shell* per defecte és `bash`.

```
$ echo $0
```

```
bash
```

`$0` és una variable especial de Bash i, en aquest cas, el seu valor és el nom de la *shell*.

Vegeu també

Vegeu l'apartat «Variables especials» del present mòdul.

Aquesta darrera manera d'executar l'*script*, sense *shebang* i sense indicar-lo no és recomanada, ja que podem tenir efectes no desitjats si executem una distribució que no tingui `bash` com a *shell* per defecte i l'*script* està programat amb Bash. Una bona pràctica és incloure el *shebang*.

Per a executar un *script* des d'un altre *script* es pot fer de diferents maneres, les més bàsiques són:

1) Amb la comanda (`builtin`) `source` (també es pot utilitzar la comanda punt «.»):

```
# Script b.sh que serà cridat des de l'script a.sh
$ cat b.sh
echo "script B. Arguments: ${@}"

# Script a.sh, utilitzant la comanda source executant l'script b.sh
$ cat a.sh
#!/bin/bash
echo "script A"
source b.sh a b c

# Execució
$ ./a.sh
script A
script B. Arguments: a b c
```

2) Executant-lo igual que es fa amb la línia de comandes:

```
# Script b.sh que serà cridat des de l'script a.sh
$ cat b.sh
echo "script B. Arguments: ${@}"

# Script a.sh, executa l'script b.sh
$ cat a.sh
#!/bin/bash
echo "script A"
bash b.sh a b c

# Execució
$ ./a.sh
script A
script B. Arguments: a b c
```

3. Caràcters especials, operadors bàsics i paraules reservades

Bash disposa de **caràcters especials** que tenen un significat concret, el seu ús és reservat i el llenguatge reconeix aquests caràcters i hi aplica un comportament definit en lloc del seu sentit literal.

Els caràcters especials no han d'anar entre cometes per a poder ser interpretats amb aquesta finalitat. Si apareixen entre cometes, s'interpreten com a valor literal.

`#` aquest caràcter serveix per a comentar línies. Bash ignorarà qualsevol text posterior que estigui en la mateixa línia.

```
$ # Aquest contingut serà ignorat.
$ echo Hola # Aquest contingut serà ignorat.
Hola
$    # Aquest contingut serà ignorat.
$
```

El caràcter «#» té un comportament diferent quan està dins d'expressions numèriques o de substitució de cadenes de caràcters. Aquest casos singulars els veurem més endavant.

`\` (contrabarra). Escapa caràcters. Conserva el valor literal del caràcter que el succeeix. Si el caràcter que el succeeix és un salt de línia, significa que la línia continua.

```
$ echo "escapem cometes dobles \"\""
escapem cometes dobles ""
```

`'` (cometa simple). Els caràcters entre cometes simples `' '` s'interpreten com a literals.

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

`"` (cometes dobles). Els caràcters entre cometes dobles `" "` s'interpreten com a literals a excepció de la barra obliqua inversa «\», el dòlar «$» i les backticks «` `».

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

3.1. Operadors de control

Realitzen tasques de control de flux.

`&` executa la comanda en segon pla.

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

`;` separa diferents comandes i s'executen en ordre seqüencial.

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

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

`;&` continua amb l'execució de les comandes de la disposició següent de la sentència de control *case*.

`;;&` continua amb la disposició següent de la sentència de control *case* i executa les comandes si es compleix la condició.

Vegeu també

Vegeu l'apartat «El condicional *case*» del present mòdul.

`(...)` agrupa comandes per a ser executades en una *subshell*. El seu comportament és similar a utilitzar `{...}` però en aquest darrer cas, no s'executen en cap *subshell*.

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

`&&` operador lògic *AND*. Permet executar una comanda, però només si l'anterior s'ha executat amb èxit (codi de retorn igual a zero).

```
$ true && echo "abc"
abc
$ false && echo "abc"
(sense sortida)
```

`||` operador lògic *OR*. Executa una comanda, però només si l'anterior s'ha executat sense èxit (codi de retorn diferent de zero).

```
$ true || echo "abc"
(sense sortida)
$ false || echo "abc"
abc
```

`|` pipa (*pipe* - *pipeline*). Envia la sortida d'una comanda a l'entrada de la comanda següent.

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

```
1
2
3
```

3.2. Operadors de redirecció

Redirigeixen l'entrada i sortida de les comandes.

< la comanda obté l'entrada del fitxer proporcionat a la dreta de l'operador.

```
$ cat fitxer.txt
abc
$ cat < fitxer.txt
abc
```

> redirigeix la sortida de la comanda a un fitxer. Si el fitxer existeix, el sobreescriu.

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

>| redirigeix la sortida de la comanda a un fitxer. A diferència de l'anterior, sobreescriurà el fitxer si existeix, encara que la *shell* estigui configurada per a no sobreescriure fitxers existents (*set* o *noclobber*).

Vegeu pàgina del `man`:

```
man set
```

<< l'entrada de la comanda és un *here document*.

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

<<< l'entrada és un *here string*. Similar al *here document* però d'una línia.

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

>> redirigeix la sortida a un fitxer però si aquest existeix, agrega el contingut al final del fitxer.

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

`>&` redirigeix la sortida estàndard i la sortida d'error a un fitxer. Si el fitxer existeix, el sobreescriu.

`&>` redirigeix la sortida estàndard i la sortida d'error a un fitxer. Si el fitxer existeix, el sobreescriu.

`>>&` redirigeix la sortida estàndard i la sortida d'error a un fitxer. Si el fitxer existeix, agrega el contingut al final del fitxer.

`&>>` redirigeix la sortida estàndard i la sortida d'error a un fitxer. Si el fitxer existeix, agrega el contingut al final del fitxer.

`|&` pipa (*pipe* - *pipeline*). Envia la sortida estàndard i la sortida d'error d'una comanda a l'entrada de la comanda següent.

3.3. Operadors aritmètics

Realitzen operacions aritmètiques (amb nombres **enters**).

Aquests operadors s'utilitzen habitualment amb les expansions aritmètiques de Bash amb `$((...))`. També es poden utilitzar amb `backticks`, `expr` o amb el `builtin let`, però es recomana fer servir les expansions de Bash a causa de la seva senzillesa. La comanda `expr` es pot utilitzar per la compatibilitat amb *shells* antigues.

Vegeu també

Vegeu l'apartat «Expansions» del present mòdul.

`+` suma dos operands.

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

`-` resta dos operands.

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

`*` multiplica dos operands.

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

`/` divideix dos operands.

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

`%` mòdul de dos operands. Residu de la divisió dels dos operands.

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

++ incrementa l'operand en una unitat.

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

-- decrementa l'operand en una unitat.

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

Consultar els `man`:

```
man let
man expr
```

3.4. Operadors relacionals

Els **operadors relacionals** comparen dos valors.

== Compara si els dos operands són iguals. El resultat és cert si són iguals i fals en cas contrari.

!= Compara si els dos operands són diferents. El resultat és cert si són diferents i fals en cas contrari.

< Compara si el primer operand (el de l'esquerra) és més petit que el segon (el de la dreta). El resultat és cert si el primer operand és més petit que el segon. El resultat és fals en altre cas.

> Compara si el primer operand és més gran que el segon. El resultat és cert si el primer operand és més gran que el segon. El resultat és fals en altre cas.

<= Compara si el primer operand és més petit o igual que el segon. El resultat és cert si el primer operand és més petit o igual que el segon. El resultat és fals si passa altrament.

>= Compara si el primer operand és més gran o igual que el segon. El resultat és cert si el primer operand és més gran o igual que el segon. El resultat és fals si passa altrament.

3.5. Operadors lògics o booleans

Realitzen operacions lògiques.

`&&` **Intersecció o *AND* lògica**. El resultat és cert si els dos operands són certs. El resultat és fals si passa altrament.

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

`||` **Unió o *OR* lògica**. El resultat és cert si un dels dos operands és cert o tots dos són certs. El resultat és fals si passa altrament.

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

`!` **∂Negació**. Aquest operador és unari. El resultat és cert si l'operand és fals. El resultat és fals si passa altrament.

3.6. Operadors bit a bit

Realitzen operacions bit a bit.

`&` Intersecció o *AND* bit a bit.

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

`|` Unió o *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
```

<< Desplaçament aritmètic cap a l'esquerra.

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

>> Desplaçament aritmètic cap a la dreta.

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

Addicionalment, Bash té un compendi de **paraules reservades** del llenguatge que formen part del seu lèxic i només poden ser utilitzades per a la construcció de comandes. Aquestes paraules reservades són:

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

Veurem l'ús d'aquestes paraules reservades en apartats posteriors.

4. Expansions

Durant la fase d'anàlisi del lèxic, l'intèrpret de Bash divideix les comandes amb paraules reconegudes. En aquest procés es detecten patrons i caràcters interns d'expressions que se substitueixen per valors.

Referència

Vegeu l'enllaç «3.5 Shell Expansions».

4.1. Expansió de claus

L'**expansió de claus** es la que es produeix en primer lloc. Les expansions de claus "`{...}`" substitueixen l'expressió per les cadenes de caràcters generades.

Per a no interpretar-ho com a literal, l'expressió no ha d'estar entre cometes.

```
$ 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. Expansió de les titlles

Expandeix el caràcter «~» i els successius fins a la primera barra obliqua «/». Si aquesta no apareix es tracten tots els caràcters en l'expansió.

El resultat de l'expansió serà determinada en funció dels caràcters que succeeixin la titlla. Els exemples més habituals són:

- ~ valor de `$HOME` si està definida al directori *home* de l'usuari.
- ~+ valor de `$PWD`.
- ~- valor de `$OLDPWD`.

```
# Expansió de ~ per $HOME.
$ HOME=/tmp
$ echo ~
/tmp

# Expansió de ~ per la home de l'usuari quan $HOME no està definit.
$ unset HOME
$ echo  ~
/home/user

# Expansió de ~+ per $PWD.
$ pwd
/tmp
$ echo ~+/abc
/tmp/abc

# Expansió de ~- per $OLDPWD.
$ pwd
/home/user
$ cd /tmp/
```

```
$ echo ~-
/home/user
```

4.3. Expansió de paràmetres

L'expansió de paràmetres és del tipus `${parametre}` o `$parametre`.

Les claus són obligatòries quan «parametre» és un paràmetre posicional de més d'un dígit (per exemple `${10}`) o quan volem indicar que hi ha caràcters després del «parametre» que no formen part del mateix nom. Quan es fa l'expansió, «parametre» és substituït pel seu valor.

Vegeu també

Vegeu l'apartat «Variables especials» del present mòdul.

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

4.4. Expansió de comandes

Se substitueix la comanda de l'expressió per la seva sortida després d'executar-la en una *subshell*.

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

4.5. Expansió aritmètica

Se substitueix pel resultat de l'operació aritmètica.

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

4.6. Substitució de processos

Substitueix l'entrada o la sortida d'una comanda via fitxers. De manera que la sortida o l'entrada del procés és l'entrada o sortida de l'altre.

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

4.7. Divisió de paraules

La divisió de paraules utilitza un delimitador per a poder-les separar.

Aquest delimitador està marcat per un caràcter i el seu valor és el de la variable `$IFS`.[2] El valor per defecte del `$IFS` és espai, tabulador i nova línia (`\t\n`).

[(2)]Prové de l'acrònim *Internal Field Separator*.

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

4.8. Expansió de nom de fitxers

També conegut com a *globbing*, l'expansió del nom dels fitxers cerca els caràcters asterisc (`*`), claudàtors (`[...]`) o el signe d'interrogació (`?`).

Si hi ha algun d'aquests caràcters, la paraula es tracta com a patró i s'utilitza per a obtenir una llista dels fitxers que coincideixin amb aquest patró.

```
$ ls
fitxer1  fitxer11  fitxer2  fitxer3  LLEGEIX-ME  script.sh
$ ls fitxer?
fitxer1  fitxer2  fitxer3
$ ls *[1-2]
fitxer1  fitxer11  fitxer2
$ ls *[^1-2]
fitxer3  LLEGEIX-ME  script.sh
$ ls s*
script.sh
```

Per a més informació, vegeu la pàgina del `man`:

```
man 7 glob
```

4.9. Eliminació de cometes

És la darrera expansió, elimina les cometes (`'`), les cometes dobles (`"`) i la barra obliqua inversa (`\`) que no siguin resultat de les expansions.

5. Variables

Les **variables** són una de les parts més importants dels *scripts*. Durant l'execució d'un *script* podem necessitar emmagatzemar dades temporalment perquè es puguin tractar amb posterioritat. Aquestes dades es poden desar en variables.

Amb l'ús de variables podem prendre decisions en el nostre *script* en funció del valor que tenen, és a dir, podem dotar el nostre *script* de comportaments en funció de les variables.

Quan es defineix una variable amb Bash no és obligatori definir el tipus. El tipus bàsic és l'*string*. Malgrat que la variable contingui només dígits, podem realitzar operacions aritmètiques bàsiques amb enters.

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

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

També es pot declarar el tipus de variable amb el `built-in declare`.

Per a més informació, vegeu l'ajuda amb el `built-in help`:

```
help declare
```

5.1. Variables d'entorn

A banda de les variables que podem definir dins d'un *script*, amb Bash hi ha les anomenades *variables d'entorn*.

Les **variables d'entorn** tenen la finalitat de configurar l'entorn en què estem executant Bash.

Les variables d'entorn poden ser globals (exportades) o locals, la diferència principal entre elles és l'àmbit. Les **variables globals** són visibles per a tots els subprocessos que creï la *shell*, mentre que les **variables locals** només són visibles des de la *shell* que les ha definit.

Per a veure el valor de les variables d'entorn globals podem fer-ho amb la comanda `printenv`:

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

La comanda `env` també ens mostrarà les variables d'entorn:

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

Tant `printenv` com `env`, quan les executem sense arguments, mostren les variables d'entorn. Addicionalment, la comanda `env`, quan li indiquem arguments, permet assignar variables d'entorn abans d'executar qualsevol *script*.

Si volem mostrar el valor d'una variable en concret, podem fer servir `echo` o `printenv`:

```
$ echo $HOME
/home/user

$ printenv HOME
/home/user
```

`echo` requereix que li indiquem el símbol de dòlar (`$`) amb el nom de la variable perquè Bash internament li indiqui el valor de la variable com a argument per a poder-lo mostrar. `echo` té el propòsit general de mostrar línies de text per pantalla i no pas de resoldre valors de variables.

La comanda `printenv` sí que està dissenyada especialment per a resoldre noms de variables i és capaç de resoldre el seu valor internament via la funció `getenv` de la `stdlib` de `glibc`.

Les variables (i mètodes) locals els podem veure amb el `built-in set`:

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

5.2. Variables d'usuari

Podem crear les nostres variables amb el valor que desitgem realitzant una assignació de la manera següent:

variable=valor

On *variable* és el nom de la variable i *valor* és el seu valor.

Per exemple:

```
$ var="món"
$ echo "Hola, $var!"
Hola, món!
```

La variable `var` declarada és local i el seu àmbit és la *shell* actual i les seves *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
```

Veiem que en aquest cas, tant la *shell* com la *subshell* (creada amb els parèntesis «( )») tenen definida la variable `var`. `BASH_SUBSHELL` és una variable interna de Bash que ens diu si la *shell* és una *subshell* o no. Retorna el valor «0» per a indicar-nos que no és una *subshell* o el valor «1» per a indicar que sí que ho és. El punt i coma («;») el fem servir per a separar diferents comandes.

En canvi, si obrim una nova *shell* (subprocés de la *shell* actual) veurem que no existeix:

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

(sortida en blanc)
```

Perquè estigui disponible en una nova *shell*, hem de declarar-la com a variable global. Per a fer-ho, utilitzem la comanda `export`:

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

Quan l'exportem com a variable global, tots els subprocessos creats per la *shell* actual tindran definida la variable.

També podem assignar el valor d'una variable a partir de la sortida d'una comanda. Bash suporta les *backticks* (`` ` ` ``) o `$()`.

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

Per a definir una variable de només lectura:

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

Per a eliminar una variable utilitzem el `built-in unset`:

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

(sortida en blanc)
```

Les variables declarades com a només lectura no poden ser eliminades.

5.3. Variables especials

Bash proporciona una sèrie de variables especials relacionades amb els paràmetres per a ser utilitzades en els *scripts*. Aquestes variables són només de lectura per a l'usuari. Internament hi ha *built-ins* que sí que modifiquen el valor d'algunes d'aquestes, per exemple la comanda shift (que modifica un *array* intern anomenat `dollar_vars`).

- `$@`: *array* amb tots els arguments.
- `$*`: cadena de caràcters amb tots els arguments.
- `$#`: nombre d'arguments.
- `$_`: nom de l'*script* i darrer argument de la comanda executada anteriorment.
- `$-`: opcions habilitades via *set* de la *shell*.
- `$?`: valor del codi d'estat.
- `$$`: PID de la *shell*.
- `$!`: PID del darrer procés llençat al *background*.
- `$n`: arguments posicionals. `$0` correspon al nom de la comanda. `$1` a `$9` són els arguments. Per a accedir a partir del desè argument (inclòs), hem de fer-ho amb `${n}`, és a dir, `${10}`, `${11}`, etc.

Vegeu també

Vegeu els apartats «*Arrays*» i «Funcions» del present mòdul.

6. *Arrays*

Un ***array*** és una estructura de dades que permet emmagatzemar diversos valors als quals s'hi accedeix a partir d'un índex.

Amb Bash, podem accedir als elements de l'*array* de manera global o individual.

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

# Accés global, mostra tots els elements de l'array.
$ echo ${array[@]}
a b c d e f
$ echo ${array[*]}
a b c d e f

# Accés individual, amb l'índex s'accedeix a la posició de l'element.
$ echo ${array[5]}
f

# Mida de l'array.
$ echo ${#array[@]}
6
```

Com es pot observar, un *array* és una variable que conté sis elements (`a`, `b`, `c`, `d`, `e`, `f`). Per a accedir a tots els elements de l'*array* ho fem amb l'índex `@` o `*`. Per a accedir a cada element individual de l'*array* ho fem amb els índex numèrics. Amb Bash, el primer element de l'*array* comença per l'índex zero (`0`) i no l'u (`1`). La mida de l'*array* l'obtenim amb l'operador `#`.

També podem inicialitzar un *array* posició per posició (o amb `declare`), eliminar qualsevol element individual de l'*array*, afegir elements nous o accedir a determinades posicions via un índex i un desplaçament:

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

# Mostrem tots els elements.
$ echo ${array[@]}
a b c

# Eliminem l'element de la posició 1.
$ unset array[1]
$ echo ${array[@]}
a c

# S'ha de tenir present que quan eliminem un element d'una posició de l'array, la resta de
l'array es manté intacte. Els elements mantenen el seu índex.
$ echo ${array[0]} ${array[2]}
a c
$ array[1]='b'
$ echo ${array[@]}
a b c
```

```
# Afegim un element fent ús de l'índex.
$ array[3]='d'
$ echo ${array[@]}
a b c d

# Afegim diversos elements de cop.
$ array=(${array[@]} e f)
$ echo ${array[@]}
a b c d e f

# Elements del tercer al cinquè.
$ echo ${array[@]:2:3}
c d e

# Darrer element. Notar l'espai en blanc després de ':'.
$ echo ${array[@]: -1}
f

# Substituim tots els elements "b" per "c".
$ echo ${array[@]/b/c}
a c c d e f

# Esborrem l'array.
$ unset array
$ echo ${array[@]}

(sortida en blanc)
```

7. Funcions

Les **funcions** són blocs de codi que fan unes determinades accions i poden ser cridades des d'altres parts del codi i les vegades que es vulgui.

Hi ha dues maneres per a declarar una funció:

1) Especificar la paraula reservada `function` a la seva signatura.
2) Utilitzar parèntesis a la seva signatura.

```
# Definim la funció "hola" emprant function a la signatura.
$ function hola {
 echo "Hola, món!"
 }

# Definim la funció "adeu" sense emprar function a la signatura. Fem servir parèntesis
per a indicar que és una funció.
$ adeu() {
 echo "Adéu!"
 }

# Executem les funcions.
$ hola
Hola, món!
$ adeu
Adéu!
```

El pas de paràmetres a una funció és posicional i es fa durant la crida a aquesta:

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

Dintre de les funcions també es poden definir variables d'àmbit local:

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

A més de rebre arguments, les funcions amb Bash poden retornar codis d'estat.

Per a més informació, vegeu:

```
man exit
```

Els **codis d'estat** són nombres enters [0-255] que serveixen per a indicar, generalment, si el resultat d'una comanda (o funció) s'ha executat amb èxit o no.

Quan una comanda retorna un codi d'estat diferent de zero significa que hi ha hagut un error en l'execució de la comanda. Si el codi d'estat retornat és zero, significa que la comanda s'ha executat amb èxit.

El codi d'estat retornat es pot consultar amb la variable especial `$?`, que emmagatzema el valor de retorn de la darrera comanda executada.

No és obligatori retornar explícitament un codi d'estat en una funció. Si no ho especifiquem, Bash executarà la funció i si s'executa amb èxit o no li assignarà el codi de retorn corresponent.

```
# Creem dues funcions, "f()" i "ff()". La primera s'executarà amb èxit i la segona amb error.
$ f() {
 echo "f"
 }
$ ff() {
 comanda_no_existent
 }

# Cridem a les funcions i consultem el codi d'estat de sortida.
$ f
f
$ echo $?
0
$ ff
bash: comanda_no_existent: command not found
$ echo $?
127
```

Els diferents codis d'estat de diverses comandes amb *pipes* es poden comprovar amb la variable interna `PIPESTATUS`.

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

Per a retornar un codi d'estat s'utilitza la paraula reservada `return`.

```
$ f() {
 # Cos de la funció.
 return 1
 }
$ f
$ echo $?
1
```

Amb Bash no podem retornar des de les funcions valors que no siguin numèrics perquè siguin tractats com a codis d'estat. Si volem retornar valors des de les funcions podem utilitzar variables de diferents maneres:

```
# En aquest exemple, s'utilitza una variable des del codi que crida a la funció per a desar
el valor de retorn.
```

```
$ f() {
 echo "valor"
 }

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

# També directament.
echo "$(f)"
valor

# En aquest exemple, s'utilitza una variable definida a la funció per a desar el valor de retorn
i és llegida pel codi que crida a la funció.

$ f() {
 v="valor"
 }

$ echo "$v"
valor

# En aquest exemple, s'aprofita la variable especial $? de codi d'estat per a retornar un
resultat enter. Aquest mètode és il·lustratiu i no és recomanat. Per a utilitzar aquest
mètode, s'han d'establir unes pautes sobre els codis d'estat, ja que s'està aprofitant
aquest mètode per a retornar un resultat que està fora del propòsit dels codis d'estat.

$ f() {
 return 22
 }

$ f
$ echo $?
22

# Amb el darrer exemple, podem tenir efectes no desitjats en funció de l'escenari.

$ f() {
 # Codi que calcula diverses operacions matemàtiques.
 # Retornem un valor que suposarem dinàmic.
 return 350
 }

$ f
$ echo $?
94

# S'ha obtingut un valor no esperat i diferent al retornat per la funció. Com que s'estan
aprofitant els codis d'estat per a retornar valors, aquests estan dintre del rang [0, 255].
Qualsevol valor fora del rang (com el 350) implica que es retornarà al mòdul del codi
d'estat per 256 que és el rang definit, és a dir, 350 mod 256 = 94.
```

8. Sentències condicionals

8.1. La sentència `if`

La sentència `if` decideix quan s'executarà o no una acció o grups d'accions determinades.

La decisió d'executar-se o no és determinada per una expressió lògica. Si el resultat de l'expressió és certa, les accions s'executaran. En cas contrari, no s'executaran i es passarà a avaluar l'expressió lògica següent o a executar l'alternativa si n'hi ha.

Les diferents sintaxis del condicional `if` són les següents:

1) Sentència `if`

Avalua la condició i, si es compleix, executa el bloc de codi.

```
if condició; then
  # Codi a executar si es compleix la condició.
fi
```

2) Sentència `if/else`

Avalua la condició. Si es compleix executa el bloc de codi de dins de l'`if`; si no es compleix, executa el bloc de codi de dins de l'`else`.

```
if condició; then
  # Codi a executar si es compleix la condició.
else
  # Codi a executar si no es compleix la condició.
fi
```

3) Sentència `if/elif/else`

Avalua la condició de l'`if` i si es compleix, executa el seu bloc de codi. Si la primera condició no es compleix, s'avalua la condició `elif` següent. Si es compleix, s'executa el bloc de codi de l'`elif`. Si no es compleix cap de les condicions anteriors, s'executa el codi de dins de l'`else` (si existeix).

```
if condició; then
  # Codi a executar si es compleix la condició if.
elif condició; then
  # Codi a executar si es compleix la condició elif.
else
  # Codi a executar si no es compleix cap condició anterior.
fi
```

8.1.1. Format de les condicions

Les **condicions** han de tenir un format específic. Actualment, hi ha tres formats:

1) Condicions amb claudàtors simples

Tenen la forma següent:

```
if [ condició ]; then
<...>
fi
```

Aquesta és la versió portable a altres *shells* POSIX.

Suporta tres tipus de condicions:

Vegeu també

Es detallen cadascun dels diferents tipus de condicions en l'apartat «Tipus de condicions».

- Condicions basades en cadenes de caràcters.
- Condicions basades en nombres (aritmètiques).
- Condicions basades en fitxers.

2) Condicions amb claudàtors dobles

```
if [[ condició ]]; then
<...>
fi
```

Aquesta és la versió millorada del condicional i és suportada per Bash. D'entre les millores es pot destacar:

- Suport d'expansions en cadenes de caràcters i fitxers.
- Opció de posar les cadenes de caràcters entre cometes.
- Permet combinar expressions dintre d'una mateixa condició.

3) Condicions amb parèntesi

```
if ( comanda ); then
<...>
fi
```

Aquests condicionals comproven el codi d'estat retornat després d'executar la comanda en una *subshell*.

4) Condicions amb doble parèntesi

```
if (( condició )); then
<...>
fi
```

Aquests condicionals comproven els resultats aritmètics.

5) Condicions sense signes de puntuació

```
if comanda; then
<...>
fi
```

6) Unió i intersecció de condicions

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

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

8.1.2. Tipus de condicions

Les condicions poden estar compostes per diferents tipus d'expressions a avaluar.

1) Condicions de cadenes de caràcters

Les **condicions basades en cadenes** de caràcters avaluen expressions sobre les seves propietats, per exemple, la longitud, l'ordre alfabètic o la igualtat entre elles.

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

Les expressions utilitzades en cadenes de caràcters són:

- <: la cadena de caràcters de l'esquerra té precedència alfabètica a la cadena de caràcters de la dreta.
- >: la cadena de caràcters de la dreta té precedència alfabètica a la cadena de caràcters de l'esquerra.
- =: les cadenes de caràcters són iguals. També és compatible amb patrons.
- ==: ídem a l'anterior.

- `!=`: les cadenes de caràcters són diferents.

- `=~`: la cadena de caràcters coincideix amb l'expressió regular.

```
# Compara si "abc" és igual a "abc". El resultat és cert.
if [[ "abc" == "abc" ]]; then
  echo "iguals"
fi

# Compara si "abc" és diferent a "cde". El resultat és cert.
if [[ "abc" != "cde" ]]; then
  echo "diferents"
fi

# La cadena "ab" precedeix alfabèticament a "ac".
if [[ "ac" < "ab" ]]; then
  echo "\"ac\" té precedència"
else
  echo "\"ab\" té precedència"
fi
```

2) Condicions de nombres

Les **condicions basades en nombres enters** avaluen expressions sobre les propietats d'aquests nombres com ara la igualtat o quin dels dos nombres és menor o major.

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

Les expressions utilitzades en nombres són:

- `-eq` compara si els dos nombres són iguals.

- `-ne` compara si els dos nombres no són iguals.

- `-gt` compara si el nombre de l'esquerra és major que el nombre de la dreta.

- `-lt` compara si el nombre de l'esquerra és menor que el nombre de la dreta.

- `-ge` compara si el nombre de l'esquerra és major o igual que el nombre de la dreta.

- `-le` compara si el nombre de l'esquerra és menor o igual que el nombre de la dreta.

```
# Nombres iguals.
if [[ 1 -eq 001 ]]; then
  echo "iguals"
fi

# 4 és més gran que 1.
if [[ 4 -gt 1 ]]; then
  echo "4 és major"
```

```
fi
```

3) Condicions de fitxers

Les **condicions basades en fitxers** avaluen expressions sobre les propietats dels fitxers. Aquestes propietats poden ser l'existència i el tipus de fitxer, entre d'altres.

```
if [[ -e fitxer ]]; then
<...>
fi
```

Hi ha moltes expressions sobre fitxers. Es recomana consultar la pàgina del `man` per a poder tenir una llista completa d'aquestes. A continuació s'enumeren algunes d'aquestes:

- `-e` comprova si el fitxer existeix.
- `-d` comprova si el fitxer existeix i si és un directori.
- `-r` comprova si el fitxer existeix i si té permisos de lectura.
- `-w` comprova si el fitxer existeix i si té permisos d'escriptura.
- `-x` comprova si el fitxer existeix i si té permisos d'execució.

```
if [[ -e fitxer1.txt ]]; then
  echo "existeix"
fi
```

Per a més informació, vegeu la pàgina del `man`:

```
man test
```

8.2. El condicional `case`

El **condicional `case`** s'utilitza habitualment per a simplificar quan es tenen molts condicionals possibles i això comporta tenir moltes sentències `if`/`else` niades.

La sintaxi del condicional `case` és la següent:

```
case expressió in
condició 1)
  # Codi a executar per al cas1.
  ;;
condició N)
  # Codi a executar per al casN.
  ;;
esac
```

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

- `;&` continua amb l'execució de les comandes de la disposició següent de la sentència de control `case`.

- `;;&` continua amb la disposició següent de la sentència de control `case` i executa les comandes si es compleix la condició.

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

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

9. Sentències iteratives

9.1. La sentència `for`

La **sentència `for`** la fem servir per a definir bucles que executaran una seqüència de comandes un nombre determinat de vegades.

La seva sintaxi és la següent:

```
for expressió; do
  # Codi a executar.
done
```

L'expressió estàndard és:

```
for element in [LLISTA]; do
  # Codi a executar.
done
```

Per exemple:

```
# Iterar sobre una llista d'elements.
for elem in a b c; do
  echo $elem
done

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

# Iterar sobre un rang de nombres amb increment
for num in {0..10..2}; do
  echo $num
done
```

L'expressió també pot ser de l'estil de C:

`for` (inicialització; condició de finalització; increment)

Per exemple:

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

9.2. La sentència `while`

La `sentència while` la fem servir per a definir bucles que executaran una seqüència de comandes mentre es compleixi una condició.

La seva sintaxi és la següent:

```
while [[ condició ]]; do
  # codi a executar
done
```

Per exemple:

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

9.3. La sentència `until`

La **sentència `until`** és igual que la sentència `while` però amb l'única diferència que la seva condició és negada, és a dir, el bucle s'executa mentre la condició no es compleixi.

La seva sintaxi és la següent:

```
until [[ condició ]]; do
  # Codi a executar.
done
```

Exemple:

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

9.4. Les sentències `break` i `continue`

Les **sentències `break` i `continue`** poden ser utilitzades en els bucles `for`, `while` i `until` per a sortir del bucle en el cas del `break` o per a saltar la part de codi restant del bucle en el cas del `continue`.

Exemple d'ús de `break`:

```
# Bucle infinit del qual se surt amb break quan el valor de i és 5.
i=0
while true; do
  echo $i
  if [[ $((++i)) -eq 5 ]]; then
```

```
    break
  fi
done
```

Exemple d'ús de `continue`:

```
# Bucle en què a la segona iteració no imprimeix la segona cadena a causa del continue.
for i in {1..3}; do
  echo "[$i] primera"
  if [[ $i -eq 2 ]]; then
    continue
  fi
  echo "[$i] segona"
done
```

10. Depuració d'*scripts*

Per a poder **depurar (o *debuggar*) *scripts*** amb Bash, s'habilita el mode de depuració executant l'*script* amb l'opció `-x`.

Per exemple, l'*script* senzill següent:

```
#!/bin/bash

echo "prova de depuració"
echo "executem date: $(date -d "01/01/2019")"
echo "fi"
```

I la seva execució:

```
$ ./debug.sh
prova de depuració
executem date: Tue 01 Jan 2019 12:00:00 AM CET
fi
```

Amb el mode de depuració es mostren traces addicionals per a cada comanda executada per l'*script*:

```
$ bash -x debug.sh
+ echo 'prova de depuració'
prova de depuració
++ date -d 01/01/2019
+ echo 'executem date: Tue 01 Jan 2019 12:00:00 AM CET'
executem date: Tue 01 Jan 2019 12:00:00 AM CET
+ echo fi
fi
```

Per a depurar parts concretes de l'*script*, s'habilita amb *set* el mode entre les línies que es vulgui depurar:

```
#!/bin/bash

echo "prova de depuració"
set -x
echo "executem date: $(date -d "01/01/2019")"
set +x
echo "fi"
```

Durant l'execució s'imprimeixen únicament les traces de les línies entre el *set*:

```
$ ./debug.sh
prova de depuració
++ date -d 01/01/2019
+ echo 'executem date: Tue 01 Jan 2019 12:00:00 AM CET'
executem date: Tue 01 Jan 2019 12:00:00 AM CET
+ set +x
fi
```

Bibliografia

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

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» (4a. edició). LinuxCommand.org.