Guía de programación en Bash

¿Qué es la shell?

- Es el programa que interpreta las peticiones del usuario para ejecutar programas.
- Hay muchas: **`/bin/sh`**, **`/bin/bash`**, **`/bin/csh`**...
  (en MSDOS era command.com)
- Bash (Bourne-Again SHell) se ha convertido en el estándar de facto.

Inicio de bash

`/bin/bash` se ejecuta de 3 modos diferentes:

- ✔ *Intérprete interactivo de ingreso:* Tras hacer login en modo texto
  - Ejecuta /etc/profile (vbles de entorno y funciones para todo el sistema)
  - Ejecuta ~/.profile (específico del usuario, redefine variables generalmente ejecuta ~/bashrc)
  - Presenta el prompt
- ✔ *Intérprete interactivo de no ingreso:* Terminal en ventana gráfica
  - Copia el entorno padre y luego ejecuta **`~/.bash_rc`**
  - Presenta el prompt
- ✔ *Intérprete no interactivo:* Al lanzar un script

Ciclo de vida de Bash

- Una vez presentado el prompt (que se puede cambiar en variable $PS1 generalemente en ~/bashrc), se queda a la espera de comandos

- Cuando se introduce un comando:
  - Hace sustitución de variables
  - Expande los metacaracteres
  - Maneja redireccionamientos de entrada y salida y las tuberías
  - Realiza la sustitución de comandos
  - Ejecuta el comando, pasándolo al núcleo
  - Cuando finaliza el programa vuelve a presentar el prompt

Requisitos mínimos de un script

- ✔ Fichero textual con permisos de ejecución
- ✔ Primera línea del script (carácter Sha-bang #!)

```
#!/bin/bash
```

Esto no es obligatorio, pero facilita la cosas:

- """Cuando ejecutemos el script, la shell detecta el caracter #! y lanza /bin/bash para que procese el script.
- """Nos asegura que siempre será ejecutado por bash, aunque nuestra shell actual sea otra (csh, sh...)

- ✔ Cada lenguaje script indica su binario:

```
#!/usr/bin/perl
```

- ✔ Se ejecuta como cualquier otro comando:

```
./script.sh
```

```
#!/bin/sh
# Comentario
# /bin/sh es enlace simbolico a /bin/bash en linux

comando1
comando2 \
comando_2_continúa
#podemos partir líneas con \

comando3;comando4
#los comandos se separan por fin de línea, o por ;
```

Variables y parámetros

- Reglas típicas: no comenzar por dígitos, ni carácteres especiales, etc.
- Sólamente aparece sin el $ en:
  - Asignaciones (=, read, cabeceras de bucles)
  - Otros casos especiales (cuando es unset, exportada o señal)
- Empiezan por $ cuando son referenciadas. Bash interpreta el carácter $ como *sustitución de la variable* por su contenido

```
#!/bin/bash

variable=2
variable=$variable + 1
b=$variable
```

Reglas de sustitución de variables

- Los $ se interpretan en el interior de **comillas dobles** (partial quoting)
- Los $ **no** se interpretan en el interior de **comillas simples** (full quoting)
- Si $variable1 causa error de sintaxis, ${variable1} debería funcionar

Ejemplo de sustitución de variables

```
hola="A B   C"

Comando          Salida     Explicación
echo hola        #hola      (no se hace referencia a ninguna variable)

echo $hola       #A B C     (echo recibe A B C, tres argumentos)

echo ${hola}     #A B C     (idéntico al anterior)

echo "$hola"     #A B   C   (echo recibe un único argumento)

echo "${hola}"   #A B   C   (idéntico al anterior)

echo '$hola'     #$hola     ($ se interpreta literalmente)
```

No es necesario declarar variables, pero existen declare y typeset. O también así:

```
variable1=          #Declaración que inicializa a nulo
unset variable1     #Vuelta a nulo
```

Asignación de variables

- No puede haber espacios antes y después del =

```
vble1 =valor1 #Comando vble1 con argumento =valor1
vble1= valor1 #Comando valor1 cuya salida se asigna a la variable vble1
```

- Tipos de asignación

```
a=16+5 ; a=$b       #Asignación normal
let a=16+5          #Asignación con let
for a in 7 8 9 10   #Cabeceras de bucles
echo -n "Dame la IP: " ; read IP  #Cláusula read
```

Argumentos de un script

```
./scriptname 1 2 3 4 5 6 7 8 9 10 11
```

- ✔ scriptname podrá manejarlos con las variables:

$1,${10}	Denotan cada parámetro según su posición
$*	Denotan todos los parámetros separados por espacio
$@	Igual que el anterior pero cada parámetro entrecomillado
$#	Número de argumentos pasados
$0	El nombre del script tal cual ha sido tecleado basename $0 obtiene el nombre normal del fichero (sin path)
shift	Comando que reasigna los argumentos perdiendo todos una posición: $1<--$2, $2<--$3, $3<--$4, $4<--$5 Útil para recorrer los argumentos sin usar las llaves { }

Ejemplo: Script que crea una cuenta local de usuario, usando como parámetros el usuario, contraseña y grupo

```
#!/bin/bash
# alta_usuario.sh

useradd $1 -s /bin/bash -g $3 -p $2 -d /home/$3/$1
cp -R /etc/skel /home/$3
mv /home/$3/skel /home/$3/$1
chown -R $1:$3 /home/$3/$1
chmod 700 /home/$3/$1
```

```
./alta_usuario.sh jsanchez2711 12345 alumnos
```

Entrecomillado \ " '

- **Entrecomillado de un carácter (Backslash \)**

Anteponer un \ a un caracter dice a la shell que lo interprete literalmente. Es similar a hacer un entrecomillado simple en un sólo carácter.

```
echo "\"Hello\""    #imprime "Hello"
echo "\$vble"       #imprime $vble
echo "\\"           #imprime \
echo "'(]\\{}\$\""  #imprime '(]\{}$"
```

- **Entrecomillado completo** (**comillas simples** ')

No se realiza ninguna sutitución de variables ni se contemplan significados especiales de caracteres. Se toma la cadena literalmente. Excepto:

\ para poder escribir una comilla simple

```
echo 'I Can'\''t'   #I Can't
```

Caracteres de escape

- Algunos comandos, como sed y echo, atribuyen significados especiales a ciertos caracteres:

```
\n   Nueva línea
\r   Retorno de carro
\t   Tabulación
\v   Tabulación vertical
\b   Espacio
\a   Alerta (beep)
\0xx octal ASCII equivalente a 0xx
```

```
echo "\t\t\t"       muestra \t\t\t literalmente
echo -e "\t\t\t"    nuestra 3 tabulaciones (-e imprime caracteres de
                    escape)
echo $'\n'          una forma de simular la opción -e
```

Sustitución de comandos

Con el acento grave (`) se ejecutan uno o varios comandos, y bash literalmente pone la salida en otro contexto, como por ejemplo, asignándola a una variable, como argumentos de un comando o como lista en un bucle.

- Forma clásica: Comillas invertidas

```
for i in `ls`
```

```
COMMAND  `echo a b`      2 argumentos, a y b
COMMAND "`echo a b`"     1 argumento: "a b"
COMMAND  `echo`          No argumento
COMMAND "`echo`"         Argumento vacío
```

- También se pueden usar $(comando)

```
a=$(ls -l)      #Asigna el resultado de "ls -l" a la vble a
a=$(uname -m)   #Asigna el resultado de uname -m a la vble a
```

- **Redireccionamiento de salida: De proceso a fichero**
  - La salida que cualquier comando genere en stdout, se puede redireccionar a cualquier otro fichero:

```
ls -l > lista.txt    #sobreescribe en caso de existir lista.txt
ls -l >> lista.txt   #añade en caso de existir lista.txt
```

- **Redireccionamiento de entrada: De fichero a proceso**
  - Cualquier comando que lea su entrada de stdin puede redireccionar su entrada para que venga de otro fichero:

```
read a < fichero
```

- **Redireccionamiento de la salida de errores**
  - Cualquier comando que genere mensajes de error a stderr puede redireccionar estos mensajes en otro fichero:

```
ls -l 2> ls_errores
ls -l 2>> ls_errores
```

- **Redireccionamiento de entrada y salida estándar**

```
grep palabra < fichero_donde_buscar.txt > lineas_econtradas.txt
```

- Utilidades:

```
comando &> file     #Redirige stderr y stdout
: > file            #Crea un fichero vacío
: >> file           #Crea un fichero vacío, o no hace nada si ya existe
```

Si se quiere vaciar un fichero, este método es el más eficiente (similar a touch si el fichero no existe)

```
exec 3<> File  #Abre el File para R/W asignándole el descriptor 3
i>&j           #Redirige el descriptor i a j. Toda salida del fichero
               apuntado por i se envía al fichero apuntado por j.
2>&1           Redirige los errores a stdout
```

TUBERÍAS

- Se utiliza para enlazar comandos. Aquí no hay ficheros, se usa sólo entre procesos.
- La salida de un comando se convierte en la entrada del siguiente.

```
cat alumnos.txt | sort | uniq | lp
```

Ejemplo

El fichero alumnos.txt contiene una línea:

12345,"Sanchez Perez ","Juan",24/11/1988,"1DA","1"

Construir un login de usuario consistente en:

inicial_del_nombre + primer_apellido + dia_nacimiento + mes_de_nacimiento

```
FILE=alumnos.txt

APES=`cut -f 2 -d, | cut -f2 -d\" | tr 'A-Z' 'a-z'`
NAME=`cut -f 3 -d, $FILE | cut -f2 -d\" | tr 'A-Z' 'a-z'`
FNAC=`cut -f 4 -d, $FILE | tr -d '/' | cut -c 1-4`

LOGIN=`echo "$NAME" | cut -c 1``echo "$APES" | cut -f1 -d" " `$FNAC
```

condicionales: test, [], if/then

Operadores que evalúan la condición y devuelven 0 para verdadero, y 1 para falso

- **test** <condicional>
  - Desde la linea de comandos se ejecuta /usr/bin/test, extensión de bash
  - Desde un script es parte de sh-utils (no llama al binario anterior)

- **[** <condicional>
  - El corchete izquierdo es sinónimo de test, pero más eficiente.
  - No tiene por qué cerrarse con], pero generalmente ello incurre a error por otras circunstancias ajenas al comando.

- **[[<condicional>]]**
  - Método diferente de comparación, más intuitivo y parecido a los lenguajes típicos. Más versátil y completo. Adaptado de ksh88

if/then

- Se suele combinar con alguno de las tres opciones anteriores para ejecutar comandos en caso de que la comparación sea verdadero o falso.

```
if test -z "$1"           #Verdadero si hay argumentos
if /usr/bin/test -z "&1" #Mismo resultado, pero con el programa "test"
if [ -z "$1" ]            #Funcionalmente idéntico
if [ -z "$1"              #Debería funcionar, pero podría devolver error
if /usr/bin/[ -z "$1" ]  #De nuevo, funcionalmente válido
```

- if/then también puede usarse con comandos directamente. En este caso más que evaluar, simplemente devuelve su código de error.

```
if cmp a b &> /dev/null      verdadero si son idénticos (cmp devuelve 0)
if grep -q $USR /etc/passwd  verdadero si usuario encontrado
if [ 0 ]                      0 es verdadero
if [ 1 ]                      1 es verdadero
```

sintaxis común de if

```
if [ condici·n1 ]
then
     ...
elif [ condici·n2 ]
then
     ...
else
     ...
fi
```

Alternativa: ((condición))

- ✔ Ejecuta comparaciones aritméticas numéricas

```
#!/bin/bash
(( 4 > 2 ))
echo $?        Devuelve 0, true
```

Operadores para testear ficheros

```
     if [ -op ruta_del_fichero ]

-op  Devuelve verdadero si el fichero:
-e   Existe
-s   No tiene tamaño cero
-f   Es regular (no es directorio ni dispositivo)
-d   Es un directorio
-b   Es un dispositivo de bloques (floppy, cdrom, etc.)
-c   Es un dispositivo de caracteres (teclado, m·dem, etc.)
-h   Es un enlace simbólico (también -L)
-S   Es un socket
-r   Tiene permisos de lectura (para el usuario que ejecuta el programa)
-w   Tiene permisos de escritura
-x   Tiene permisos de ejecución
-O   Yo soy el propietario del fichero
-G   Su grupo es el mismo que el mío
f1 -nt f2     f1 es más reciente que f2 (nt = newer than)
f1 -ot f2     f1 es más antiguo que f2  (ot = older than)
```

Comparación entre enteros (cadenas con dígitos)

```
if [ "$a" -eq "$b" ]       es igual (equal)
if [ "$a" -ne "$b" ]       no es igual (not equal)
if [ "$a" -gt "$b" ]       mayor (greater than)
if [ "$a" -ge "$b" ]       mayor o igual (greater or equal)
if [ "$a" -lt "$b" ]       menor (less than)
if [ "$a" -le "$b" ]       menor o igual (less or equal)
```

Comparación entre cadenas

```
if [ "$a" = "$b" ]         son iguales
if [ "$a" == "$b" ]        son iguales
if [ "$a" != "$b" ]        son diferentes

if [ "$a" \> "$b" ]        mayor, en orden alfabético ASCII
if [ "$a" \<  "$b" ]       menor, en orden alfabético ASCII

if [ -z "$cadena" ]        la cadena es nula (de longitud cero)
if [ -n "$cadena" ]        la cadena no es nula
```

Composición de comparaciones

- Opción 1:

```
if [ exp1 -a exp2 ]      #AND lógico
if [ exp1 -o exp2 ]      #OR lógico
```

- Opción 2:

```
if [ exp1 ] && [ exp2 ]  #AND lógico
if [ exp1 ] || [ exp2 ]  #OR lógico
```

```
if [ -f /usr/bin/firefox -a -f /root/archivo.html ]; then
     firefox /root/archivo.html &
fi
```

- ! niega el resultado de la comparación (NOT lógico)

```
if [ ! -e "$file" ]      #si no existe el fichero...
```

Operadores aritméticos

```
+ - * /          Suma, resta, multiplicaciñon, división
**               Potencia
%                Resto de la división entera
+=               Incrementa una variable con una constante
-= *= /= %=      Decrementa, multiplica, divide o se queda con el resto
```

- Bash no entiende aritmética real de punto flotante.
  Los decimales se interpretan como cadenas.

Ejemplos:

```
let "z = 5 ** 3"       # z = 5 * 5 * 5 = 125
let "z += 5"           # z = z + 5 = 130
let "z /= 3"           # z = z / 3 = 43

a=1.5
let "b = a + 1.3"      # Error
```

Operadores lógicos

```
<<,>>  Desplazamiento binario hacia la izquierda o hacia la derecha
&      Y binario
|      O binario
~      No binario
!      No lógico
&&     Y lógico
||     O lógico
```

Constantes

```
let "decimal = 32"
let "octal = 032"               #Precedido de 0 es octal
let "hexadecimal = 0x32"        #Precedido de 0x es hexadecimal
let "binario = 2#1111001101"    #Precedido de 2# es binario
let "base32 = 32#77"
```

for

```
for variable in lista
do
     ...
done
```

- Si in lista es omitido, se usan automáticamente los parámetros pasados al script o a la función.

- En lista puede ir cualquier lista de elementos:

```
for file in * ; do ...
for file in $(ls) ; do ...
for file in `ls` ; do ...
```

- Alternativas

```
for a in 1 2 3 4 5 6 7 8 9 10
for a in `1 step 10`
for ((a=1; a <= 10 ; a++))    #construcción similar a C
```

Ejemplo: En todos los homes de los profesores hay que copiar el archivo /usr/share/instituto/monitorprofe y llamarlo .monitorprofe. Si ya existe se mantendrá, pero hay que asegurarse que la línea que define RetardoGH sea RetardoGH=12

```
#!/bin/sh

archivo=/usr/share/instituto/monitorprofe

for i in `ls /home/profesor` ; do

  if [ ! -f /home/profesor/$i/.monitorprofe ]; then
      cp -f $archivo /home/profesor/$i/.monitorprofe
  else
      grep -v Retardo /home/profesor/$i/.monitorprofe > /tmp/tmpkk
      echo 'RetardoGH="12"' >> /tmp/tmpkk
      mv /tmp/tmpkk /home/profesor/$i/.monitorprofe
  fi
  chown $i:profesor /home/profesor/$i/.monitorprofe
done
```

while

```
while [ condición ]
do
     ...
done
```

Mientras la condición sea verdadera, se ejecuta el bloque

```
while [ "$a" -le $limite ]; do ...
```

until

```
until [ condición ]
do
     ...
done
```

Mientras la condición sea falsa, se ejecuta el bloque

break y continue

- **break**: Termina el bucle. El programa sigue ejecutándose justo después del "done" de ese bucle.

- **Continue**: Salta a la siguiente iteración del bucle. Admite como parámetro el número de iteración.

```
for i in 1 2 3 4 5 6 7 8 9 10; do
  if [ "$i" -eq 4 ]; then
    continue 2
  fi
  echo -n "$i "
done
#Salida con continue 2: 1 2 3 2 3 2 3 2 3...
#Salida con continue: 1 2 3 5 6 7 8 9 10
```

case

```
case "$variable" in
"$condicion1") ...
               ;;
"$condicion2") ...
               ;;
* )           ...
               ;;
esac
```

ejemplo de case

Script que acepta parámetros tipo: script.sh --h --conf archivo

```
while [ $# -gt 0 ]; do
  case "$1" in
    -h|--help)
       echo "Mostrar ayuda"
       ;;
    -c|--conf)
       CONF_FILE=$2
       shift
       ;;
    -d|--debug)
       DEBUG=1
       ;;
    *) echo "Parámetro no reconocido";;
  esac

  shift
done
```