Scripting en Linux

Índice

Introdución

Un *shell-script* pode ser un simple ficheiro de texto que conteña un ou varios comandos. É habitual que os *scripts* teñan a extensión "**.sh**", aínda que non é obrigatorio, pero por claridade soe facerse así. Un exemplo sería:

```
#!/bin/bash
# Comentario
echo "Ola mundo"
```

Podemos comentar as dúas liñas que contén:

1. Denomínase *shebang* e a súa utilidade é indicar cal é o intérprete que se vai utilizar (*bash*, neste caso).
2. A segunda liña é un comentario. O intérprete ignorará o seu contido. Convén dicir que BASH é unha linguaxe que non soporta comentarios multiliña, se queremos facer isto teremos que por # en cada liña.
3. Comando *echo* máis a cadea a amosar por pantalla.

Se executamos este *script* amosará por pantalla a cadea "`Ola mundo`". Pero, como podemos executar un *script*? Hai varios xeitos:

1. Utilizando o intérprete antes do nome do arquivo.

```
bash meuscript.sh
```

2. Co carácter punto.

```
./meuscript.sh
```

ou

```
. meuscript.sh
```

É importante lembrar de dotar de permisos de execución a dito arquivo para poder executalo sen problemas. Co intérprete diante non daría fallo pero sen el non deixa executalo se non teño permisos de execución.

En canto á sintaxe podemos indicar o seguinte:

- Non obriga ao uso de tabulacións. *BASH* non o fará pero eu si. Queremos escribir sempre código lexible.
- É moi sensible ao uso ou á ausencia de espazos.

Variables

Non definiremos o que é unha variable porque supoñemos que xa falaron delas no módulo de Programación. O que si comentarei é que podemos facer con elas:

- **Definilas**. As variables son locais ao proceso ou script no que se definen. Tamén convén dicir que estarán dispoñibles despois de definilas, non antes, coma se fose un proceso secuencial (*batch*) sen volta atrás. Ollo co nomeado porque é "*case sensitive*", co cal non lle da igual que sexa maiúscula ou minúscula. Tamén tede coidado se poñedes caracteres estranos, comezades por números,...

```
MIVARIABLE=""
MIVARIABLE
```

- **Inicializalas**. Non se pode deixar espazo antes nin despois do igual. O valor sempre será tido en conta como unha cadea de caracteres.

```
MIVARIABLE="Ola"
MIVARIABLE ="Ola" # Dará un erro
MIVARIABLE= "Ola" # Tamén dará un erro
```

- **Acceder** ao seu valor.

```
$MIVARIABLE
```

- **Eliminalas**.

```
unset MIVARIABLE
```

Podemos dicir que hai varios tipos de variables. As que nos poden interesar serían:

- **Cadeas**.

```
cadea="A miña cadea"
```

- **Números**. Ollo porque, por defecto, *BASH* non se leva ben cos decimais.

```
numero=23
```

- **Resultado de execución de comandos**. Realmente isto pode ser unha variante das cadeas, xa que o que se almacenaría sería o resultado da execución en forma de cadea.

```
resultado=`ps -aux` # Ou $(ps -aux)
```

- **Listas**.

EXERCICIO: Crear un script sinxelo denominado `proba01.sh` no que se amose, con 3 variables diferentes, o nome do alumno, a súa idade e o resultado de execución dun comando de Linux.

Podemos realizar operacións sobre as variables numéricas utilizando a dobre paréntese:

```
numero=23
((numero++))
echo $numero
```

Deste xeito, a partir dese momento, `numero` valerá 24 e non 23. Ou tamén:

```
numero=23
((numero+=2))
echo $numero
```

Deste xeito, a partir dese momento, `numero` valerá 25 e non 23.

EXERCICIO: No script anterior amosa a idade que terá o alumno o ano que ven e dentro de 10 anos.

Comiñas

En BASH podemos usar 3 tipos de comiñas:

- **Simples**: non interpreta caracteres coma o $, polo que non poderemos concatenar cadeas con valores de variables.

```
nome='Gerardo'
```

- **Dobres**: o contrario que as simples si que interpreta caracteres especiais coma o $.

```
nome='Gerardo'
apelido='Otero'
completo="$nome $apelido"

echo $completo
```

- **Invertidas**: ou comiña francesa, executará o contido dentro delas.

```
a=ls

echo `$a' # Amosará por pantalla $a
echo "$a" # Amosará por pantalla ls
echo `$a` # Amosará por pantalla o contido do directorio onde nos atopemos
```

Operadores

Podemos usar os operadores típicos como poden ser:

- suma (+)
- resta (-)
- multiplicación (*)
- división (/)
- módulo (%)
- E (&&)
- OU (||)

Para realizar unha operación aritmética poderemos facelo dos seguintes xeitos:

```
expr 2 \* 2 # Executa 2x2 e amosa o resultado, ollo co carácter de escape
let "var2=2 * 2"; echo $var2 # O comando let avalía expresións matemáticas
echo $((2 * 2))
```

EXERCICIO: Realiza algunha pequena proba con cada un dos operadores. Se queres podes obviar os operadores lóxicos (AND e OR) ata que vexamos os bucles.

Comparadores

Podemos usar os seguintes comparadores recomendados para os números:

- menor que (-lt)
- menor ou igual que (-le)
- maior que (-gt)
- maior ou igual que (-ge)
- igual (-eq)
- distinto (-ne)

E os recomendados para cadeas:

- menor que (\<)
- maior que (\>)
- igual (= ou ==)
- distinto (!=)
- cadea NON baleira (-n)
- cadea baleira (-z)

EXERCICIO: Realiza algunha proba con cada un dos comparadores anteriores. (*)

Operadores ficheiros

Podemos usar os seguintes comparadores:

- existe o ficheiro ou directorio (-e)
- existe o ficheiro NON directorio (-f)
- o ficheiro non está baleiro (-s)
- é un directorio (-d)
- se ten os permisos indicados (-r | -w | -x)
- máis recente (-nt)
- máis antigo (-ot)

```
#!/bin/bash

read -p "Introduce o nome dun ficheiro: " f

if [ -e $f ]
then
       echo "O ficheiro existe"
       if [ -s $f ]
       then
              echo "Non está baleiro"
       else
              echo "Está baleiro"
       fi
else
       echo "Ese ficheiro non existe"
fi
```

EXERCICIO: Realiza algunha proba con cada un dos operadores anteriores. (*)

Condicionais

A sintaxe é semellante a outras linguaxes de programación. Vexamos un exemplo:

```
if [ condición ]
then
       sentenzas en caso de que SI se cumpra
else
       sentenzas en caso de que NON se cumpra
fi
```

Cousas a ter en conta:

- Usamos os corchetes en vez dos parénteses.
- Ollo cos espazos entre a condición e os corchetes.
- Ollo co `then`, vai noutra liña aparte.
- Hai que finalizar a sentenza coa palabra chave `fi`.

Un exemplo práctico sería:

```
num1=1
num2=2

if [ $num1 -ne $num2 ]
then
       echo "Son números diferentes"
else
       echo "Son números iguais"
fi
```

ou tamén

```
if [ $num1 != $num2 ]
then
       echo "Son números diferentes"
else
       echo "Son números iguais"
fi
```

aínda que a recomendación é utilizar `-ne` para números e `!=` para texto. Así que, se son cadeas, sendo estritos debería ser:

```
if [ $cad1 != $cad2 ]
then
       echo "Son cadeas diferentes"
else
       echo "Son cadeas iguais"
fi
```

Isto daría un erro:

```
if [ $cad1 -ne $cad2 ]
then
	echo "Son cadeas diferentes"
else
	echo "Son cadeas iguais"
fi
```

Sen embargo isto non daría erro:

```
if [[ $cad1 -ne $cad2 ]]
then
	echo "Son cadeas diferentes"
else
	echo "Son cadeas iguais"
fi
```

OLLO, porque lembremos que BASH é unha linguaxe interpretada, é dicir, vaise interpretando liña a liña, así que dependendo do valor que introduzamos podo que dea un erro o seguinte código:

```
cad1=Pepe
cad2=Pep

if [ $cad1 \> $cad2 ]
then
	echo "$cad1 maior que $cad2"
else
	echo "$cad1 NON maior que $cad2"
fi
```

Así non da erro pero se mudamos os valores das variables si que o pode dar:

```
cad1="Pepe Domingo"
cad2="Pep"

if [ $cad1 \> $cad2 ]
then
	echo "$cad1 maior que $cad2"
else
	echo "$cad1 NON maior que $cad2"
fi
```

Neste caso, cando substitúe $cad1 polo seu valor "Pepe Domingo" trata de executar:

```
if [ Pepe Domingo \> ... ]
```

E iso da un erro de "demasiados parámetros", xa que despois de Pepe está esperando un operador de comparación. Como podemos solucionar isto? Así:

```
cad1=”Pepe Domingo”
cad2=”Pep”

if [ “$cad1” \> “$cad2” ]
then
       echo “$cad1 maior que $cad2”
else
       echo “$cad1 NON maior que $cad2”
fi
```

Deste xeito xa entende como todo o que hai dentro de “cad1” como un único parámetro.

Podemos introducir condicións no ELSE con ELIF do seguinte xeito:

```
#!/bin/bash
read -p “Introduce un numero:” num1 # Xa veremos o que fai isto
read -p “Introduce outro numero:” num2 # Xa veremos o que fai isto

if [[ “$num1” -eq “$num2” ]]
then
       echo “$num1 e $num2 son iguais”
elif [[ “$num1” -gt “$num2” ]]
then
       echo “$num1 é maior que $num2”
elif [[ “$num1” -lt “$num2” ]]
then
       echo “$num1 é menor que $num2”
fi # Fixádevos que só leva un fin de IF, porque só hai un IF
```

(Estamos usando “`read -p`” que veremos no seguinte apartado)

Sería equivalente do seguinte xeito:

```
#!/bin/bash
read -p “Introduce un numero:” num1 # Xa veremos o que fai isto
read -p “Introduce outro numero:” num2 # Xa veremos o que fai isto

if [ $num1 == $num2 ]
then
       echo “$num1 e $num2 son iguais”
elif [ $num1 \> $num2 ]
then
       echo “$num1 é maior que $num2”
elif [ $num1 \< $num2 ]
then
       echo “$num1 é menor que $num2”
fi
```

Outro condicional sería o **case**:

```
case $variable in
       valor1)
       accións
       ;;
       valor2)
       accións
       ;;
       valor3|valor4)
       accións
       ;;
       *) # Calquera cousa
       accións
esac
```

Por exemplo:

```
case $num3 in
       4)
       echo cuatro
       ;;
       5)
       echo cinco
       ;;
       6|7)
       echo seis ou siete
       ;;
       *) # Calquera cousa
       echo outro
esac
```

(*): Agora xa podes realizar os exercicios anteriores

Entrada

Podemos pasar parámetros de entrada ao noso script. Por exemplo:

```
./proba.sh 2
```

Estamos a executar o script incluído no arquivo `proba.sh` e estamos a pasar o parámetro `2`. Para acceder aos parámetros de entrada utilizaremos `$` e a posición. Neste caso para acceder ao valor `2` usaremos `$1`. Con `$0` accederemos ao string co nome do script, neste caso `./proba.sh`. Pode incluír a ruta ou non (dependendo do que escribísemos nos).

Tamén podemos acceder a todos os parámetros con `$*`, e devolverá unha cadea cos mesmos.

```
./proba.sh 2
```

Para saber o número de parámetros introducimos `$#`.

Podemos introducir datos dun xeito interactivo coa función read.

```
echo "como te chamas?"
read nome
echo "Que tal, $nome"
```

Ou sen utilizar o echo

```
read -p "como te chamas?" nome
echo "Que tal, $nome"
```

EXERCICIO: Realiza un script no que se lle amose ao usuario actual co que está logeado e, despois, se lle permita modificar este nome. Será unha modificación "virtual" non se mudará realmente o usuario de sistema. Deberá amosarse unha mensaxe na que se pida o novo nome. En caso de que o usuario pulse enter quererá dicir que non se muda de usuario e o valor será o mesmo que xa se tiña gardado.

Bucles

Hai 3 tipos de bucles en Bash:

- **while**

```
while condición
do
	accións
done
```

Un exemplo sería:

```
i=0
while (( i <= 5 ))
do
	echo $i
	sleep 1 # Para facelo efecto de espera
	((i++)) # Moi importante
done
```

- **for**

```
for i in lista # Soe utilizarse con listas ou coleccións
do
	accións
done
```

Un exemplo sería:

```
lista=(un dous tres catro cinco)

echo "O tamanho da lista é: {#lista[@]}"

echo "Se quero acceder ao contido"
for i in ${lista[@]}
do
	echo $i
done

echo "Se quero acceder á posición"
for i in ${!lista[@]}
do
	echo "A posición é: $i"
	echo "O contido da posición $i é: ${lista[i]}"
done
```

Tamén se pode empregar para percorrer un directorio:

```
for file in /home/alumno/*
do
	echo $file
done
```

E usando sintaxe de C:

```
for (( a=1; a<=10; a++))
do
       echo $a
done
```

- **until**

```
until condición
do
       accións
done
```

Un exemplo sería:

```
i=0
until (( i >= 5 ))
do
       echo $i
       sleep 1 # Para facelo efecto de espera
       ((i++)) # Moi importante
       # ou i=$(($i+1))
done
```

EXERCICIO: Realiza un script que percorra o home do usuario alumno e, para cada elemento, indica si é un arquivo ou un directorio.

Funcións

Hai varios xeitos de declarar funcións en Bash pero, sen dúbida, o máis recomendable sería o seguinte:

```
function miFuncion() {
	accións
}
```

Unha vez declarada, para chamala tan só hai que facer referencia ao seu nome:

```
miFuncion # Sen parénteses
```

Tamén, ao igual que as chamadas dos scripts, podemos facer uso dos parámetros, do seguinte xeito:

```
function miFuncion() {
	suma=$(($1+$2));
	echo $suma;
}

…
miFuncion 2 3
```

Unha das características fundamentais das funcións é a de retornar valores, en Bash podemos facer o seguinte:

```
function miFuncion() {
	suma=$(($1+$2));
	return $suma;
}

…
miFuncion 2 3
echo $?
```