Shell Scripting

Enrique Ocaña González
chucky_spain@hotmail.com

17 de octubre de 2000

Resumen

Este artículo es una adaptación del libro *Bourne Shell Programming*, escrito por Robert P. Sayle. En él se explica todos los detalles que es necesario conocer para programar scripts de shell.

1 Entrada/Salida

La entrada y la salida de comandos se puede redirigir y encadenar de las siguientes formas:

- **<, > Redirección de la Entrada y Salida Estándar**:

```
$ grep Linux < entrada.txt
$ cat /dev/audio > sonido_grabado.au
$ grep Linux < entrada.txt > salida.txt
```

- **| Pipes (Redirección de la Salida Estándar de un comando a la Entrada Estándar del siguiente)**:

```
$ ps ax | grep "\ R\ "
5348 ttyp0 R 0:00 ps ax
```

- **<< Leer Entrada Estándar del popio script hasta una marca**:

```
$ cat << FIN
> Este es mi mensaje largo
> de 2 lineas
> FIN
Este es mi mensaje largo
de 2 lineas
```

TRUCO: Se pueden expandir variables en el texto, de modo que resulta muy fácil hacer plantillas

```
$ CUENTAS=‘ls /home‘
$ for cuenta in ${CUENTAS}; do cat <<FIN
> He encontrado la cuenta ${cuenta} en el sistema
> FIN
> done
He encontrado la cuenta enrique en el sistema
He encontrado la cuenta gm en el sistema
He encontrado la cuenta magic en el sistema
```

- **>> Redirección de la Salida estándar en modo Append (continuación del fichero)**:

  `$ echo 'La pantalla ha explotado' >> eventos.log`

- ***descriptor*<*fichero*, *descriptor*<<*fichero* Redirección de la entrada de un descriptor a un fichero**:

  `$ sort 0<mi_fichero.txt` (Otra forma alternativa de redirigir la entrada estándar)

- ***descriptor*>*fichero*, *descriptor*>>*fichero* Redirección de la salida de un descriptor a un fichero**:

  `$ ls pepe 2>>errores.txt` (Hemos redirigido el Error Estándar (descriptor nº 2) al fichero errores.txt en modo Append)

- ***descriptor*<*&descriptor_fich* Redirección de la entrada de un descriptor al fichero utilizado por otro descriptor**.

- ***descriptor*>*&descriptor_fich* Redirección de la salida de un descriptor al fichero utilizado por otro descriptor**:

  `$ ls pepe >>salida_y_errores.txt 2>&1` (Hemos redirigido la Salida Estándar (descriptor nº 1) a un fichero en modo Append, y luego hemos redirigido el error estándar (descriptor nº 2) al mismo fichero que está siendo utilizado por el descriptor 1 (el de la salida estándar))

- **<&- Cierra la Entrada Estándar**.

- **>&- Cierra la Salida Estándar**:

  `$ ls >&-` (La salida del comando no va a ninguna parte y por eso `ls` nos va a dar un error)
  `ls: error de escritura: Descriptor de fichero erróneo`

- `exec >>`***fichero*** **Redirección de la Salida Estándar para todas las salidas de los comandos de un script**:

  ```
  #!/bin/sh
  # Script que vuelca el contenido de /home y /tmp al fichero cosas.txt
  exec >>cosas.txt (Esta es la parte que redirige toda la salida de lo que viene a continuación)
  ls /home
  ls /tmp
  ```

- `for ... done>>`***fichero*** **Redirección de la Salida Estándar para todas las operaciones que se ejecuten en un bucle**:

  ```
  $ USERS=‘cat /etc/passwd | cut -f1 -d:‘
  $ for user in ${USERS}
  > do
  > du -sk /home/${user}
  > done >>diskusage 2>/dev/null (Aquí se hace la redirección)
  $ cat diskusage
  2748 /home/ftp
  38931 /home/jac
  109 /home/mlambi
  ```

```
26151 /home/rsayle
5 /home/bobt
8185 /home/mann_al
13759 /home/bheintz
```

1.1 Algunos ficheros interesantes para Entrada/Salida

- **/dev/null**: Descarta cualquier cosa que se le envíe. Útil para que la salida de un comando no salga por pantalla.
- **/dev/tty**: Nuestro propio terminal. Lo que se envía a él nos sale por pantalla

2 Agrupación de comandos

Los comandos simples de un shell pueden ser agrupados de las siguientes formas:

- **; Punto y coma**: Encadena comandos en una sola línea como si fuesen uno solo.

```
$ ls > /tmp/listado; ls -lisa > /tmp/listado_detallado
```

- **( ) Paréntesis**: Ejecuta lo que haya dentro en un subshell, de modo que los cambios que se hagan en las variables no permanecen al volver al shell actual. No es necesario separar los paréntesis de los comandos mediante espacios.

```
$ TMP=permanece; (TMP=no_me_ves; echo $TMP); echo $TMP
```

- **{ } Llaves**: Ejecuta lo que haya dentro utilizando la shell actual. Es necesario separar las llaves de los comandos, terminar el último comando dentro de la llave con un punto y coma y ponerle también el punto y coma a la propia llave (como si en lugar de { y } tuviésemos un comando de sistema).

```
$ TMP=permanece; { TMP=no_me_ves; echo $TMP; }; echo $TMP
```

- **&& Doble ampersand**: Ejecuta el siguiente comando sólo si el anterior tuvo resultado verdadero (igual a 0).

```
$ ls pepe && echo 'El directorio pepe existe'
```

- || **Doble raya vertical**: Ejecuta el siguiente comando sólo si el anterior tuvo resultado falso (distinto de 0).

```
$ ls pepe || echo 'El directorio pepe no existe'
```

3 Expresiones Regulares

Una expresión regular es una secuencia de caracteres que especifica un patrón textual. Se utilizan en muchos comandos de UNIX, como por ejemplo grep, sed, awk y egrep. Mediande ellas se puede indicar a los programas que procesen sólo aquellos elementos que encajen con el patrón. La mayoría de las expresiones regulares utilizan metacaracteres para expresar repetición, existencia o rangos en patrones de caracteres.

A continuación se describen los metacaracteres más comunes:

- **Punto (.)**: Empareja con cualquier carácter excepto avance de línea.
- **Asterisco (*)**: Empareja 0 o más ocurrencias del carácter o expresión regular que le precede.
- **Corchetes []**: Empareja cualquiera de las clases de caracteres encerrados entre los corchetes.
  - **^ tras el corchete de apertura** indica que el emparejamiento se debe realizar a la inversa, es decir, seleccionar aquellas cadenas que no coincidan con lo especificado en los corchetes.
  - **- especifica un rango**.
  - Los **metacaracteres** pierden su significado dentro de los corchetes (incluido el propio carácter de], que debe aparecer al principio de la lista).
- **Acento circunflejo (^)**: Indica "principio de línea".
- **Dólar ($)**: Indica "fin de línea".
- **Barra invertida (\)**: Escapa el metacarácter que venga a continuación.

Este metacarácter sólo se utiliza en los programas sed, grep, egrep y awk:

- **Barra-Llave** $/\{m,n/\}$: Empareja un rango de ocurrencias del carácter simple que precede inmediatamente a la expresión. Se consideran las siguientes variantes:
  - $/\{m/\}$ El patrón encaja exactamente con m repeticiones.
  - $/\{m,/\}$ El patrón encaja con m repeticiones como mínimo.
  - $/\{m,n/\}$ El patrón encaja si aparecen entre m y n repeticiones.

Estos metacaracteres sólo se utilizan en los programas egrep y awk:

- **Más (+)**: Empareja 1 o más ocurrencias del carácter o expresión regular que le precede.
- **Interrogación (?)**: Empareja 0 o 1 ocurrencias del carácter o expresión regular que le precede.
- **Barra vertical (|)**: Empareja con el carácter o expr. que le precede o con la que le sigue.
- **Paréntesis ( )**: Agrupa expresiones regulares.

3.1 Compleción de nombres de fichero en el Shell

El shell utiliza algo parecido a las expresiones regulares para emparejar con los nombres de fichero de un directorio (parecido a lo que ocurre en MS-DOS). Este es el juego de metacaracteres propios que utiliza el shell:

Carácter	Significado
* (Comodín)	Empareja con 0 o más caracteres
?	Empareja con 1carácter
[*abc...*]	Empareja con cualquiera de los caracteres listados
[!*abc...*]	Empareja con lo contrario que [*abc...*]
\ (Escape)	Elimina el significado de cualquier carácter especial, incluido fin de línea

4 Entrecomillado (Quoting)

Estos los los tipos de comillas que se utilizan en scripting:

- **Comillas dobles " "**: Elimina el significado especial de todos los caracteres encerrados entre ellas excepto $, ` y \.

```
$ SYS=Linux ; echo "Este es un sistema $SYS equipado"
Este es un sistema Linux equipado
```

- **Comillas simples ' '**: Elimina el significado especial de todos los caracteres encerrados entre ellas.

```
$ SYS=Linux ; echo 'Este es un sistema $SYS equipado'
Este es un sistema $SYS equipado
```

- **Comillas invertidas ` `**: Sustitución de comando. Se ejecuta el comando que va entre comillas y su salida se pone en lugar del comando.

```
$ echo "El nombre del sistema es \"`hostname`\" "
El nombre del sistema es "chucky"
```

- **Barra invertida** \: Escape. Elimina el significado especial del carácter que le sigue. Dentro de comillas dobles \ también escapa $, ', nueva línea y al propio \.

```
$ echo Un asterisco:  \*
Un asterisco:  *
```

5 Variables

Las variables se **declaran** con un = sin espacios. A menudo es recomendable utilizar comillas.

Para **desreferenciar** una variable se coloca un **$** delante de su nombre. Para diferenciar el nombre de la variable de lo que viene a continuación es aconsejable colocal la variable entre **llaves { }**:

```
#!/bin/sh
NOMBRE='John Smith'
echo $NOMBRE
```

```
$ CARA="*Cara*"; CARACOL="*Caracol*"; echo $CARACOL; echo ${CARA}COL
*Caracol*
*Cara*COL
```

5.1 Sustitución de parámetros

La sustitución de parámetros es un método para dotar de un valor por defecto a una variable en el caso de que no esté definida (valga nulo). Existen varios tipos de sustitución de parámetros:

- $*param* **...o bien...** ${*param*} Sustituye el valor de *param*.
- ${*param*:-*valor*} Sustituye el valor de *param* si no es nulo, en cuyo caso utilizaría *valor*.

```
$ echo "El nombre es ${NOMBRE:-desconocido}"
El nombre es desconocido
$ NOMBRE=pepe; echo "El nombre es ${NOMBRE:-desconocido}"
El nombre es pepe
```

- `${param:=valor}` Sustituye el valor de *param* si no es nulo, en cuyo caso utilizaría *valor* y además asignaría *param=valor*.

```
$ echo "Antes:  ${NOMBRE}, Ahora:  ${NOMBRE:='John Smith'}, Despues:
${NOMBRE}"
Antes:  , Ahora:  'John Smith', Despues:  'John Smith'
```

- `${param:?valor}` Sustituye el valor de *param* si no es nulo, en cuyo caso escribe *valor* en el Error Estándar. Si *valor* se omite, escribe "*param*: *param* null or not set".

```
$ echo "El nombre es ${NOMBRE:?}"
bash:  NOMBRE: parameter null or not set
```

- `${param:+value}` Sustituye *value* si *param* no es nulo. En caso contrario no sustituye nada.

```
$ echo "*${NOMBRE:+Existe}*"; NOMBRE=Pepe; echo "*${NOMBRE:+Existe}*"

**
*Existe*
```

5.2 Propiedades de las variables

Las variables pueden declararse como de **sólo-lectura** mediante **readonly**:

```
$ NOMBRE=Pepe
$ NOMBRE=Juan
$ readonly NOMBRE
$ NOMBRE=Pepe
bash:  NOMBRE: readonly variable
```

Además, los valores de las variables que se asignan en el shell actual (variables locales) no son heredados por los subshells que lancemos desde el actual. Para que los subshells puedan ver las variables locales es necesario **exportarlas**:

```
$ export DISPLAY='localhost:0.0'
```

5.3 Parámetros especiales

A continuación se muestra una tabla con variables del shell que contienen valores especiales:

Variable	Contenido
$0 ... $9	Nombre del ejecutable del shell ($0) y los primeros 9 argumentos
$#	Número de argumentos pasados al shell
$*	Valor de los argumentos como un único valor
$@	Como $*pero cuando se pone entre comillas dobles, pone cada parám. entre com. dobles.
$$	PID del script o la sesión actual
$!	PID del último programa enviado al background
$?	Exit status del último programa no ejecutado en background
$-	Opciones actuales en efecto

5.4 Lectura de datos para almacenar en variables

Cuando se deseen pedir datos del usuario o de la Entrada Estándar se deberá utilizar el comando **read**:

```
$ read PERSONA CIUDAD HOBBY; echo "$PERSONA vive en $CIUDAD y practica $HOBBY"

Pepe New\ York pesca
Pepe vive en New York y practica pesca
```

Los campos en la entrada se separan generalmente por espacios, avances de línea y tabuladores, pero este comportamiento puede ser modificado combiando el valor de la variable de entorno **IFS (Internal Field Separator)**, que contiene los caracteres que actúan de separadores:

```
$ export IFS=":"
$ read PERSONA CIUDAD HOBBY; echo "$PERSONA vive en $CIUDAD y practica $HOBBY"

Pepe:New York:pesca
Pepe vive en New York y practica pesca
```

6 Funciones

Las funciones se declaran así:

```
miFuncion () {
echo 'Hola!!'
echo 'Soy una función :-)'
}

otraFuncion () { echo 'Pues yo soy otra función ;-)'; }
```

El espacio entre el nombre de la función y los paréntesis no es obligatorio, así como tampoco lo es que la apertura de llave aparezca en la misma línea que la declaración de la función. Es importante decir que **los argumentos de una función nunca se declaran**, de modo que los paréntesis deben aparecer siempre vacíos.

Para llamar a la función, simplemente se le llama, pasándole todos los parámetros que sea necesario. Los parámetros se referencian del mismo modo que en el shell: $1, $2, $3... y el resultado de la función se declara con `return` y se lee luego con la variable de entorno $?. Si una función no retorna nada, en $? queda almacenado el valor de retorno de la última sentencia que se ejecutó al llamar a la función.

```
$ felicitacion () {
> echo "Feliz cumpleaños ${1}!!"
> return ${2}
>}
$ felicitacion Jonh 16
Feliz cumpleaños John!!
$ echo $?
16
```

Los cambios realizados en las variables dentro de una función **no son locales**, es decir, permanecen al finalizar la llamada a la función (al contrario de lo que ocurre al abrir un subshell). Del mismo modo, las variables declaradas (mediante una asignación) dentro de una función quedan declaradas para todo el script.

6.1 Inclusión de código

Para incluir (como con el #include de C) el contenido de un fichero (que podría tener muchas funciones útiles definidas) en el script actual se utiliza el comando **punto** (**.**):

```
#!/bin/sh
# FICHERO utilHTML.sh
cabeceraHTML() { echo '<HTML><BODY>'; }
cierreHTML() { echo '</BODY></HTML>'; }
documentoHTML () {
cabeceraHTML
echo ${1}
cierreHTML
}
```

```
#!/bin/sh
# FICHERO saludo.sh
.  utilHTML.sh
documentoHTML 'Hola World Wide Web!!'
```

7 Estructuras condicionales

7.1 Evaluación de condiciones

Existen dos formas de chequear una condición dejando el resultado (0 si es Verdadera o 1 si es Falsa) en la variable $?:

- `test 5 = 5`
- `[ 5 = 5 ]` (En realidad, "[" es un comando del sistema igual que test, por lo que **deben respetarse los espacios**, ya que lo que viene a continuación son en realidad argumentos).

Y estas son todas las posibles comprobaciones que se pueden hacer:

Comprobación	Significado
string	*string* no es nulo
-n *string*	*string* no es nulo
-z *string*	*string* sí es nulo
string1 = *string2*	*string1* es igual a *string2*
string1 != *string2*	*string1* no es igual a *string2*
-eq	igual a (eq va en medio de los valores a comparar)
-ne	no igual
-gt	mayor que
-ge	mayor o igual
-lt	menor que
-le	menor o igual
-b	el fichero es un fichero especial de bloque
-c	es un fichero especial de carácter
-d	es un directorio
-f	es un fichero ordinario
-g	tiene el Set Group ID Bit activado
-k	tiene el Sticky Bit activado
-p	es un Named Pipe
-r	puede ser leído por el proceso actual
-s	tiene longitud distinta de 0
-t	el descriptor de fichero está abierto y asociado a un terminal
-u	tiene el Set User ID Bit activado
-w	puede ser escrito por el proceso actual
-x	puede ser ejecutado por el proceso actual
-a	hace un AND del argumento anterior y siguiente
-o	hace un OR del argumento anterior y siguiente

7.2 Decisión IF

Los **IF**s se declaran de la siguiente manera:

```
if [ $n < 5 ]; then
echo 'Es...'
echo '...menor que 5'
elif [ $n > 5 ]; then
echo 'Es...'
echo '...mayor que 5'
else
echo 'Es...'
echo '...igual a 5'
fi
```

7.3 Decisión CASE

Tambien podemos declarar estructuras **CASE**, donde se hace pattern-matching con cada caso, que sólo será ejecutado si empareja con la expresión regular:

```
case ($opcion) in
-h) echo 'Este script no tiene ayuda'
```

```
;;
-n) echo 'El nombre de la máquina es:'
hostname
;;
*)  echo 'Cualquier opción que no sea -h, -n no es válida'
;;
esac
```

8 Bucles

8.1 Bucles FOR

La forma de declarar un bucle FOR es:

```
for variable in 1 2 3 /tmp/* `ls -lisa` items_de_una_lista
do
echo 'Un item:'
echo $variable
done
```

Si se omite la lista de valores del FOR, el shell utiliza la lista de parámetros posicionales $1...$9 condensada en $@:

```
#!/bin/sh
# FICHERO imprime_parametros.sh
echo $@
for p
do
echo "Un parámetro:  ${p}"
done
```

```
$ imprime_parametros.sh Hola mundo
Hola mundo
Un parámetro:  Hola
Un parámetro:  mundo
```

8.2 Bucles WHILE

El bucle se ejecuta mientras se cumpla la condición:

```
$ i=1
$ while [ ${i} -le 5 ]
> do
> echo ${i}
> i=‘expr ${i} + 1‘
> done
1
2
3
4
5
```

Los bucles WHILE pueden ser utilizados para realizar la técnica conocida como "desplazamiento de argumentos":

```
$ cat printargs
#!/bin/sh
while [ $# -gt 0 ]
do
echo "$@"
shift
done
$ printargs hello world "foo bar" bye
hello world foo bar bye
world foo bar bye
foo bar bye
bye
```

8.3 Bucles UNTIL

El bucle se ejecuta hasta que se cumpla la condición:

```
$ cat printargs2
#!/bin/sh
until [ $# -le 0 ]
do
echo "$@"
shift
done
$ printargs2 hello world "foo bar" bye
hello world foo bar bye
world foo bar bye
foo bar bye
bye
```

8.4 Control de las iteraciones

Existen dos comandos para controlar las iteraciones de un bucle:

- `break`: Sale inmediatamente del bucle
- `continue`: Salta inmediatamente a la siguiente iteración del bucle

9 Procesado de argumentos de línea de comando

Tradicionalmente en UNIX existen dos formas de reconocer parámetros:

- Separadamente: `ping -c 10 -t 1 localhost`
- Apilados: `ls -lisa texto.txt`

La interpretación de argumentos se puede hacer cómodamente gracias al desplazamiento de argumentos:

```
#!/bin/sh
#
# setether:  set an Ethernet interface's IP configuration
#
while [ $# -gt 1 ]
do
case ${1}
a) ARP="arp"
shift
;;
b) BROADCAST=${2}
shift 2
;;
i) IPADDRESS=${2}
shift 2
;;
m) NETMASK=${2}
shift 2
;;
ñ) NETWORK=${2}
shift 2
;;
*) echo "setether:  illegal option:  ${1}"
exit 1
;;
esac
done
INTERFACE=${1}
ifconfig ${INTERFACE} ${IPADDRESS} netmask ${NETMASK} broadcast ${BROADCAST}
${ARP}
route add -net ${NETWORK}
```

Y se puede hacer aún de forma más sencilla utilizando la utilidad **getopts**:

```
#!/bin/sh
#
# setether:  set an Ethernet interface's IP configuration
#
while getopts ab:e:i:m:n:  option
do
case "${option}" in
a) ARP="arp"
b) BROADCAST=${OPTARG};;
e) INTERFACE=${OPTARG};;
i) IPADDRESS=${OPTARG};;
m) NETMASK=${OPTARG};;
ñ) NETWORK=${OPTARG};;
*) echo "setether:  illegal option:  ${option}"
exit 1
;;
esac
done
INTERFACE=${1}
```

```
ifconfig ${INTERFACE} ${IPADDRESS} netmask ${NETMASK} broadcast ${BROADCAST}
${ARP}
route add -net ${NETWORK}
```

Como el parámetro "a" no lleva nada, getopts sabe que no debe buscarle un argumento. Sin embargo, los dos puntos de "b:", "e:", "i:", "m:" y "n:" indican que esto parámetros sí llevan argumentos, que getopts deberá buscar y colocar en la variable de entorno $OPTARG. Además podemos observar cómo ya no es necesario realizar a mano el desplazamiento (shift) de la lista de argumentos, puesto que de eso ya se encarga getopts.

El uso de getopts agiliza la tarea de interpretar argumentos, aunque tiene desventajas como que sólo puede manejar opciones con un único carácter. Si queremos reconocer, por ejemplo, la opción "–help" tendremos que hacerlo a mano.