Tutorial BASH n°1

Programación BASH

INDICE

1. Introducción

Antes de empezar a programar la Shell de Linux, es necesario conocer algunos conceptos previos.

Una shell es un intérprete de comandos, es la aplicación que permite al usuario comunicarse con el sistema operativo y darle órdenes. Existen diferentes shells en Linux, algunos ejemplos:

- Bourne Shell (sh) - La shell clásica que se encuentra en todos los sistemas UNIX.
- Korn Shell (ksh)
- C Shell (csh)
- Bourne Again Shell (bash) - La shell de GNU que se encuentra en todos los sistemas Linux y en muchos otros UNIX. Es la que vamos a utilizar.

La shell no solo es capaz de interpretar comandos, puede programarse usando ficheros de texto que ésta interpretará. Estos ficheros, llamados scripts (guiones) estarán compuestos de comandos y de construcciones y facilidades que ofrece la Shell para facilitar su programación.

El Shell es un lenguaje de programación en sí mismo. Con los Shell-scripts lo que hacemos es escribir programas en el propio lenguaje del Shell, siendo interpretados por éste. Cualquier orden o secuencia de órdenes que pueda ser tecleada directamente en el terminal también puede ser almacenada y ejecutada desde un fichero. Además el Shell proporciona sentencias de lectura y escritura de variables, selectivas, repetitivas, etc., … típicas de los lenguajes de programación, que pueden ser incluidas en la Shell-scripts.

Los scripts de shell son muy útiles para ciertos tipos de tareas:

- **Tareas administrativas**: algunas partes de los sistemas UNIX son scripts de shell, para poder entenderlos y modificarlos es necesario tener alguna noción sobre la programación de scripts.
- **Tareas tediosas** que solo se van a ejecutar una o dos veces, no importa el rendimiento del programa resultante pero si conviene que su programación sea rápida.
- Hacer que **varios programas fun**cionen como un conjunto de una forma sencilla.
- Pueden ser un buen método para **desarrollar prototipos** de aplicaciones más complejas que posteriormente se implementarán en leguajes más potentes.

Conocer a fondo la shell aumenta tremendamente la rapidez y productividad a la hora de usarla, incluso fuera de los scripts.

2. Manejo básico del shell.

Para manejar el Shell es imprescindible conocer los comandos que utiliza. Algunos de los más básicos son:

- **echo**: que muestra los argumentos que recibe por pantalla.
- **ls**: que lista el contenidos (ficheros y subdirectorios) de un directorio
- **cat**: muestra el contenido de un fichero
- **more**: muestra el contenido de un fichero haciendo pausas entre pantalla. cd: permite navegar por la estructura de directorios

2.1. La línea de comandos

En Linux nos encontramos dos **prompt**, **$** y **#.** Cuando el **prompt** es **$,** el **usuario** que está utilizando el Shell es un usuario "**normal**", privilegios de administración del sistema. Si el Shell tiene como **prompt #**, el usuario que ejecuta el Shell es el **root** o **administrador** del **sistema**.

La utilización básica de la línea de comandos es

$comando opciones argumentos

Donde

- **$:** es el prompt del sistema, siendo un usuario normal el que lo ejecuta
- **comando**: es la orden de Linux que queremos ejecutar
- **opciones**: la mayoría de los comandos en Linux tienen una lista de opciones, que permiten modificar el comportamiento de la ejecución.
- **Argumentos**: los comandos de Linux pueden recibir como argumentos ficheros, directorios, etc, sobre los que se va a ejecutar la orden.

Un ejemplo de ejecución, podría ser la siguiente:

$ls

Cuando pulsemos **enter**, el comando ls se ejecutará, mostrando en este caso por pantalla, la lista de ficheros y subdirectorios del directorio actual.

2.2. Patrones de sustitución.

Existen varios caracteres que se pueden utilizar como comodines y pueden sustituir otros caracteres o grupos de caracteres.

Los más importantes son:

***** Que sustituye un conjunto cualesquiera de caracteres.

Ejemplo:

$ls /home/juan/*

? Que sustituye a un único carácter

Ejemplo:

$cp alumno?.txt seguridad/

$rm alumno3.???

[…] cualquiera de los caracteres entre corchetes

Ejemplo:

$echo /usr/[aeiou]*

3. Los caracteres más comunes

Como en cualquier lenguaje de programación, en bash hay una serie de caracteres y palabras reservadas que tienen un significado especial:

#!/bin/bash Todos los scripts de shell empiezan con esta línea, que sirve para decirle al sistema operativo que se trata de un fichero ejecutable y que sepa cuál es el intérprete que lo tiene que interpretar.

Comentario: todo lo que haya tras él en una línea es ignorado.

; Separa dos comandos que se ejecutan uno a continuación del otro:

```
echo "Hoy es:"; date
```

. Seguido del nombre de un fichero, hace que el contenido de ese fichero sea interpretado por la shell como si fuese parte del script, es como un #include de C.

' " Varias formas de entrecomillar cadenas que se explicará más adelante.

` Se ejecuta lo que hay entre las comillas como un comando y se sustituye su salida:

```
echo `date`
```

**** Escapa el siguiente carácter, haciendo que se interprete literalmente.

$ Accede al valor de una variable:

```
echo $PATH
```

~ Equivale al directorio 'home' del usuario, es equivalente a leer el valor de la variable de entorno HOME:

```
ls ~
ls $HOME
```

& Escrito después de un comando, hace que éste se ejecute en segundo plano. Esto quiere decir que el script no esperará a que el comando retorne antes de ejecutar la siguiente instrucción.

4. Variables y parámetros

Al igual que ocurre con todos los lenguajes de programación, una variables, es un conjunto de caracteres que pueden representar diferentes valores, durante la ejecución de una aplicación, en nuestro caso de un script.

Existe un grupo de variables, denominadas variables de entorno, que afectan al comportamiento de la Shell, tanto a la hora de trabajar de forma interactiva como desde los scripts que ésta interpreta. Estas variables pueden ser accedidas y modificadas en la línea de comandos y también en los scripts.

Se puede ver el valor de todas las variables de entorno definidas en un momento dado invocando al comando set sin argumentos.

Algunas variables especialmente útiles y su significado:

- **$HOME** Directorio 'home' del usuario.
- **$PATH** Rutas en las que la shell busca los ejecutables cuando se invoca un comando.
- **$PWD** Nos muestra el directorio actual.
- **$PS1** Equivale al prompt del intérprete de comandos.
- **$PATH** Nos muestra el camino de búsqueda de ficheros ejecutables.
- **$?** Esta variable contiene el valor de salida del último comando ejecutado, es útil para saber si un comando ha finalizado con éxito o ha tenido problemas. Un '0' indica que no ha habido errores, otro valor indica que sí ha habido errores.
- **$UID** Identificador del usuario que ejecuta el script.
- **$!** Identificador de proceso del último comando ejecutado en segundo plano.

Dentro de la Shell, o dentro de un Shell-script, podemos declarar y utilizar variables. A diferencia de los lenguajes de programación estructurados, no hay que declarar las variables a un determinado tipo, , basta con asignar un valor a una variable para crearla. Para acceder al valor que contiene una variable se usa el carácter $, de la siguiente forma:

```
variable=`date`

echo $variable
```

Existe un conjunto de variables especiales denominadas parámetros de un script. Al igual que ocurre con los comandos de la Shell, nuestros Shell-script pueden recibir argumentos. Los valores que reciben estos argumentos, pueden ser accedidos dentro del script, a través de los parámetros posicionales del script. Estas variables, se construye con el **$** y la posición del parámetro, así la variables **$1** hace referencia al primer parámetro, **$2** al segundo y así sucesivamente hasta **${10}** **${11}** etc. Para poder trabajar de forma cómoda con las variables que almacenan los parámetros de un script, se utiliza el **comando shift**, que mueve todos los parámetros una posición a la izquierda, de forma que $1 desaparece y será reemplazo por **$2** y así sucesivamente.

La variable **$0** es una variable especial, y hace referencia al propio Shell script. Otra variable importante es **$#**, que contiene el número de parámetros que ha recibido el script. Por último nos encontramos con **$***, que contiene todos los parámetros juntos en una sola cadena.

Resumiendo:

$0	contiene el nombre de nuestro script
$#	el número de parámetros con los que se ha invocado al shell
$n	los parámetros, con n de 1 a 9 (a $#)
$$	el PID de nuestro proceso
$*	todos los parámetros menos $0

5. Uso de comillas

En general las comillas se usan para prevenir que la shell intérprete ciertos caracteres dentro de una cadena y para que tome una cadena con espacios como una sola palabra.

Comillas dobles (“ ”). En general los caracteres especiales no son interpretados cuando están entre comillas dobles.

Sin embargo algunos de ellos sí son interpretados:

$	Está permitido referenciar variables dentro de las comillas dobles.
****	Se pueden escapar caracteres.
`	Se puede realizar sustitución de comandos, esto es, ejecutar un comando y sustituir la cadena por su salida.
Comillas simples (´)	Dentro de las comillas simples todos los caracteres son interpretados literalmente, ninguno de los caracteres especiales conserva su significado dentro de ellas.
Comillas invertidas (`)	Poner una cadena entre comillas invertidas supone ejecutar su contenido como un comando y sustituir su salida.

6. Redireccionamientos

En principio, la ejecución de cualquier comando en Linux muestra el resultado de su proceso por pantalla. Sin embargo, en algunos casos, la ejecución puede generar errores, o puede interesar que se vuelque en un fichero, etc.

Para interaccionar con la Shell, se considera que la salida por la que se muestra el resultado de la ejecución de un comando es la pantalla, la forma de comunicar la información a la Shell es a través del teclado, y puede que en algunos casos se produzcan errores. Por estos motivos, es Linux nos encontramos los siguientes elementos:

- **Entrada estándar stdin**
- **Salida estándar stdout**
- **Error estándar stderr**

Como en Linux todo es un fichero nos encontramos con una serie de ficheros especiales, que permiten interaccionar con la Shell. Ficheros estándar que están abiertos para todo programa en ejecución:

- Entrada estándar /dev/stdin (ej. Teclado)
- Salida estándar /dev/stdout (ej. Pantalla)
- Salida de error /dev/stderr (ej. Pantalla)

Otros ficheros especiales:

- /dev/null -> "la nada de UNIX"
- /dev/zero -> fuente infinita de ceros
- /dev/random -> datos aleatorios

Con los tres descriptores de ficheros: **stdin**, **stdout** y **stderr** se puede:

1. redirigir la salida estándar (stdout) a un fichero. Es decir, hacer que en vez de mostrar por pantalla el resultado de un comando, se almacene e

   $ ls > salida

   $ cat salida

2. redirigir el error estándar (stderr) a un fichero

   $ ls fichero.txt 2> error

   $ cat error

3. redirigir la entrada estándar (stdin) a fichero

   $ cat < /dev/stdin

4. redirigir stdout a stderr
5. redirigir stderr a stdout
6. redirigir stderr y stdout a un fichero
7. redirigir stderr y stdout a stdout
8. redirigir stderr y stdout a stderr

Para hacer las redirecciones de salida se utiliza **1>** ó **>**. Si se quiere redireccionar el error estándar, se utiliza **2>** y se si quiere redireccionar la entrada, se usa **<**.

Un ejemplo de redirección de salida será:

ls -l > ls-l.txt

En este caso, se creará un fichero llamado 'ls-l.txt' que contendrá lo que se vería en la pantalla si escribiese el comando 'ls -l' y lo ejecutase.

Un ejemplo de redirección de error:

grep da * 2> errores-de-grep.txt

En este caso, se creará un fichero llamado 'errores-de-grep.txt' que contendrá la parte stderr de la salida que daría el comando 'grep da *'.

Un ejemplo de redirección de salida estándar a error.

grep da * 1>&2

En este caso, la parte stdout del comando se envía a stderr; puedes observar eso de varias maneras.

Un ejemplo de error estándar a salida estándar

grep * 2>&1

Esto hará que la salida stderr de un programa se escriba en el mismo descriptor de fichero que stdout. La parte stderr del comando se envía a stdout. Si hace una tubería con less, verás que las líneas que normalmente 'desaparecen' (al ser escritas en stderr), ahora permanecen (porque están en el stdout). Dicho de otra forma, en vez de mandar el error a la salida de error, se envía a la pantalla.

Podemos redireccioanr el error y la salida estándar a un fichero. Esto colocará toda la salida de un programa en un fichero. A veces, esto es conveniente en las entradas, si quieres que un comando se ejecute en absoluto silencio.

rm -f $(find / -name core) &> /dev/null

Esto eliminará todo archivo llamado 'core' en cualquier directorio. Ten en cuenta que tienes que estar muy seguro de lo que hace un comando si le vas a eliminar la salida.

7. Tuberías

Las tuberías permiten utilizar la salida de un programa como la entrada de otro.

Un ejemplo de uso será:

ls -l | sed -e "s/[aeio]/u/g"

En este caso primero se ejecuta el comando ls -l, y luego su salida, en vez de imprimirse en la pantalla, se envía (entuba) al programa sed, que imprime su salida correspondiente.

Otro ejemplo:

ls -l | more

En este caso, la salida del programa ls -l se envía al programa more, que la muestra de forma paginada.

8. Salida de los programas

Todos los comando en Linux, una vez que se ejecutan, devuelven un número para que el Shell sepa si la ejecución ha sido correcta o ha generado errores. Un valor de retorno 0, indica que la ejecución es correcto, un valor diferente, indica que se ha producido un error. Cuando se ejecuta un programa o un comando Linux podemos, aparte de redirigir su entrada y su salida, recoger el resultado de su ejecución y su salida.

El resultado de la salida se muestra en la variable especial **$?**

$?

resultado del último programa ejecutado

Ejemplo:

$ ls fichero_a_listar 2> /dev/null ; echo $?

$ ls > /dev/null ; echo $?

Podemos ejecutar un comando, y hacer que el valor devuelto por la salida, se muestre o se almacene en una variable.

$(comando)

la salida de comando (esto es equivalente al uso de comillas invertidas.

Ejemplo:

$ salida_ls=$(ls) ; echo $salida_ls

exit ENTERO

Es la forma más taxativa de finalizar nuestro programa con el valor de salida ENTERO. Si queremos informar de una **salida correcta, ENTERO valdrá 0**, y si es **incorrecta, cualquier otro valor**.

9. Comando test

El comando test, nos va a permitir evaluar expresiones, comprobando si son verdaderas o falsas. Se utilizan para realizar comprobaciones sobre ficheros, realizar operaciones booleanas o comparaciones relacionales entre números.

Los tests se usan, principalmente, en la estructura **if then else fi** para determinar qué parte del script se va a ejecutar. Un if puede evaluar, además de un test, otras expresiones, como una lista de comandos (usando su valor de retorno), una variable o una expresión aritmética.

Hay dos formas de utilizar el comando test, con el propio comando o con **[ ]**

test expresion

[expresion]

El comando test evalúa expresión, y si evalúa a cierto, devuelve cero (true), o en otro caso 1 (false). Si no hay expresión, test siempre devuelve falso. Este comportamiento puede ser algo confuso, ya en lógica los valores cierto y falso suelen ser al contrario ya que aquí se comprueba el valor de retorno de la ejecución de test, que es 0 si es correcto y 1 si es incorrecto.

9.1. Uso con ficheros

Estos tests toman como argumento el nombre de un fichero y devuelven verdadero o falso:

```
if [ -e fichero ]
then
  echo "Existe el fichero"
fi
```

Las opciones de test más importantes para operar con ficheros son:

-e	Verdadero si el fichero existe.
-f	Comprueba que el fichero es un fichero regular (que no es ni un directorio, ni un dispositivo).
-d	Devuelve verdadero si se trata de un directorio.
-l	Cierto si el fichero es un enlace simbólico.
-r	Cierto si se tiene permiso para leer el fichero.
-w	Cierto si se tiene permiso para escribir el fichero.
-x	Cierto si se tiene permiso para ejecutar el fichero.

9.2. Operadores relacionales

El comando test incorpora una serie de opciones que nos permite realizar comparaciones entre números enteros.

```
if [ $num1 -eq $num2 ]
then
  echo "El número $num1 es igual al número $num2"
fi
```

Las opciones de test para trabajar con operadores relacionales son:

-eq	igual a
-ne	no es igual a
-gt	es mayor que
-ge	es mayor o igual que
-lt	es menor que
-le	es menor o igual que

9.3. Operadores de comparación de cadenas

Con estas opciones de test, podemos comparar dos cadenas, devolviendo verdadero o falso..

= ó ==	Comparación de igualdad
!=	Comparación de desigualdad. El operador! se puede colocar delante de cualquier test para negar su resultado.
< y **>**	Menor que y mayor que.

9.4. Operadores booleanos

! expresion	cierto si expresion es falsa (negación)
expresion1 -a expresion2	cierto si expresion1 y expresion2 son ciertas
expresion1 -o expresion2	cierto si expresion1 o expresion2 son ciertas

10. Operaciones aritméticas

Aunque el lenguaje del Shell-script no está pensado como un lenguaje de programación tradicional, permite realizar operaciones aritméticas entre números, aunque la forma de implementarlo es bastante peculiar.

Si utilizamos la siguiente expresión

$echo 1 + 1

para el Shell es la ejecución de un comando con tres argumentos, por ello si queremos que sustituya la operación por su valor emplearemos:

$((expresión)) evalúa la expresión aritmética y reemplaza el bloque por el resultado

Ejemplo:

$ echo $((1+1))

Algunos operadores aritméticos soportados:

+	Suma
*****	Multiplicación
-	Resta
/	División entera
%	Resto de la división entera
( )	Agrupar operaciones Equivalente al uso de $((expresión)) es la construcción **`expr expresion`** donde expresión puede ser 1 + 1.

11. Estructuras de control

Como en cualquier lenguaje de programación, la shell ofrece estructuras que permiten controlar el flujo de ejecución de los scripts.

11.1 La instrucción if

Se evalúa la expresión y si se obtiene un valor cierto, entonces se ejecuta los comandos pertinentes. La sintaxis general es:

```
if [expresión]
then
  comandos
else
  comandos
fi
```

then, **else**, y **fi** son palabras reservadas y como tales únicamente serán reconocidas después de una nueva línea o ;(punto y coma). Asegúrate de que terminas cada if con su fi correspondiente.

El if se puede anidar:

```
if [expresión1]
then
  ...
elif [expresión2]
then
  ...
else
  ...
fi
```

El operador **&&** se usa para ejecutar un comando, y si es correcto, ejecuta el siguiente comando en la lista.

Por ejemplo, en **comando1 && comando2**, se ejecuta primero comando1 y si es correcto se ejecuta comando2. Esto es equivalente a:

```
if comando1
then
  comando2
fi
```

El operador **||** se usa para ejecutar un comando, y si falla, ejecuta el siguiente comando de la lista. Por ejemplo, en **comando1 || comando2**, se ejecuta primero comando1 y si falla se ejecuta comando2. Esto es equivalente a:

```
comando1
if test $? -ne 0
then
  comando2
fi
```

11.2. La instrucción case

El flujo del programa se controla en base a una palabra dada. Esta palabra se compara con cada patrón hasta que se encuentra uno que haga juego. Cuando se encuentra, se ejecuta el comando asociado y se termina la instrucción.

```
case palabra-dada in
  patrón1)
    comandos
    ;;
  patrón2|patrón3)
    comandos
    ;;
  patrónN)
    comandos
    ;;
 esac
```

Un comando puede asociarse con más de un patrón. Los patrones pueden separarse unos de otros con el símbolo **|** . El orden de chequeo es el orden en que aparecen los patrones.

Para especificar un patrón por defecto se pueden usar comodines:

?	comprueba un carácter
*	comprueba cualquier número de cualquier tipo de caracteres
[nnn]	comprueba cualquiera de los caracteres entre corchetes
[!nnn]	comprueba cualquier carácter que no esté entre los corchetes
[n-n]	comprueba cualquier carácter en el rango

11.3. La instrucción for

La estructura se repite para un conjunto de valores que recorre mediante una variable. El bucle for sigue la siguiente notación general:

```
for variable in conjunto
do
  comandos
  …
done
```

Comandos es una secuencia de uno o mas comandos separados por una línea o por un ; (punto y coma).

Las palabras reservadas do y done deben estar precedidas por una línea o por un;

```
for variable in lista; do comandos; done
```

La variable **$var** toma el valor del siguiente valor de la lista en cada iteración.

Un ejemplo:

```
for i in $(ls *.sh)
do
  if [ -x "$i" ]
  then
    echo "El fichero \"$i\" es ejecutable"
  fi
done
```

11.4. Las instrucciones while y until

La instrucción **while** tiene la forma general:

```
while [expresión]
do
  lista-de-comandos
done
```

La instrucción **until** tiene la forma general:

```
until [expresión]
do
  lista-de-comandos
done
```

Su función es idéntica a while excepto en que el bucle se ejecuta mientras en status expresión no es 0.

La variable $var toma el valor del siguiente valor de la lista en cada iteración. Un ejemplo:

```
for i in $(ls *.sh)
do
  if [ -x "$i" ]
  then
    echo "El fichero \"$i\" es ejecutable"
  fi
done
```

11.5, Las instrucciones break y continue.

La instrucción break termina la ejecución del bucle más interior causando la ejecución de la instrucción **done** más cercana.

Para salir del nivel n, se usa break n, esto causa la ejecución de la instrucción done n. El comando **continue** causa la ejecución de la instrucción **while**, **until** o **for** en la cual comienza el bucle que contiene el comando **continue**.

Ejemplo:

```
#!/bin/bash
while echo "Por favor introduce un comando" ; read respuesta
do
  case "$respuesta" in
    'fin')
      break      # no más comandos
      ;;
    "")
      continue   # comando nulo
      ;;
    *)
      eval $respuesta  # ejecuta el comando
      ;;
  esac
done
```

Mientras el usuario introduzca un comando o un **string** nulo el script continua funcionando. Para pararlo el usuario debe teclear **“fin”.**

12. Funciones

Se pueden utilizar funciones para agrupar trozos de código. Para declarar una función se escribe function mi_func () { mi código }. Llamar a la función es como llamar a otro programa, sólo hay que escribir su nombre mi_func.

12.1. Ejemplo de funciones

```
#!/bin/bash
function salir ()
{
  exit 0
}
function hola ()
{
  echo  "Hola Mundo"
}
```

12.2. Ejemplo de funciones con parámetros

```
#!/bin/bash
function salir ()
{
  exit 0
}
function mostrar ()
{
  echo $1 #Parámetro pasado a la función mostrar
}
```