Linux Bash

August 22, 2016

1 Programación en Bash

Multiples actividades en nuestro sistema operativo pueden ser constituidas por elementos repetitivos o por largas secuencias de acciones, y además ser estas actividades de uso frecuente, por lo que resulta conveniente una forma de poder hacer esto con la menor interacción posible. Esto se puede lograr usando los interpretes de comando de las consolas linux.

1.1 Consola linux

Antes que nada es necesario tener claro que bash no es el único interprete de comandos linux que encontraremos y por lo mismo es de cuidado indicar que interprete se usará. Probablemente para los *scripts* más sencillos no veamos ninguna diferencia pero en otros más elaborados lo notemos.

Como un ejemplo, en la distribución linux Debian (y derivados como Ubuntu) por defecto el interprete *shell (sh)* es dash, que con respecto a bash brinda un interprete ligero y rápido para la ejecución de los *scripts* de inicialización del sistema a costo de poseer un menor número de caracteristica (como la existencia de vectores).

Así, el binario asociado a la interpretación de comandos linux mediante `/bin/sh` redirige a dash, mientras que si deseamos ejecutar con bash se debe indicar explicitamente su uso (`bash` o `/bin/bash/`).

```
In [1]: !ls -o /bin/*sh
```

```
-rwxr-xr-x 1 root 1037528 jun 24 10:44 /bin/bash
-rwxr-xr-x 1 root  154072 feb 17 15:25 /bin/dash
lrwxrwxrwx 1 root       4 jun 24 10:44 /bin/rbash -> bash
lrwxrwxrwx 1 root       4 jul 10 11:33 /bin/sh -> dash
lrwxrwxrwx 1 root       7 jul 10 11:33 /bin/static-sh -> busybox
```

1.2 Scripts

Nuestros *scripts* en bash como en cualquier otro lenguaje poseen estructuras definidas claramente. En bash, encontramos:

- Cabecera: Sirve para indicar el interprete a utilizar y la codificación.
- Definición de variables globales: Asignamos valores a variables que serán de acceso en todo el *script*.

- Ayuda: Define la función de ayuda del *script*. Útil para la indicación de uso.
- Funciones: Permite un diseño modular de nuestro *script*.
- Cuerpo principal.

Debemos recordar que antes de ejecutar el *script* debemos asignar los permisos adecuados, ejemplo: `chmod 755 script.sh`.

1.2.1 Cabecera

Nuestra primera linea de cabecera corresponde a la indicación del interprete de comandos a usar, el cual nos interesa que sea bash. En este caso la linea debe lucir como sigue:

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

Esta linea puede usarse por ejemplo, para indicar que un *script* debe ejecutarse con python sin necesidad de ingresar a la consola de python.

Una vez se definio el interprete, ya es posible ejecutar nuestro *script* como `./script.sh`.

La segunda linea obedece a la codificación, lo cual nos permitirá hacer uso de codificación UTF-8 para soporte de caracteres latinos, los cuales por defecto en la codificación ASCII no son posibles (es la versión de 7 bits y no el extendido de 8 bits).

```
# -*- ENCODING: UTF-8 -*-
```

1.2.2 Variables

La asignación de variables en bash se hace de forma directa sin requerir de la declaración de tipo de variable. Las variables en bash soportan números, caracteres y cadenas de caracteres, y se deben asignar sin dejar espacios alrededor del signo igual. Las variables pueden ser de caracter local antecediendo la instrucción `local`.

Para obtener el valor almacenado en una variable se requiere el uso del signo `$` antes de su nombre.

```
In [2]: %%bash
        variable=5
        echo $variable
```

```
5
```

Si el signo `$` es omitido, se interpreta el nombre de la variable como un comando.

```
In [3]: %%bash
        variable=5
        variable
```

```
bash: line 2: variable: command not found
```

La obtención del valor no sirve solo para su impresión o manipulación por otras expresiones sino tambien para la ejecución de la instrucción guardada como cadena de caracteres.

```
In [4]: %%bash
        comando="ls -oh *png"
        $comando
```

```
-rw-rw-r-- 1 cosmoscalibur 13K ago  7 18:35 jupyter.png
```

Existen variables de entorno ya predefinidas asociadas a distintas caracteristicas o información de nuestro sistema. Algunas de ellas son:

```
In [5]: %%bash
        echo $USER
        echo $HOSTNAME
        echo $HOME
```

```
cosmoscalibur
EdAsus
/home/cosmoscalibur
```

Para la lectura de variables recurrimos a la instrucción `read` seguido de la variable deseada. Tambien es posible usar vectores mediante la siguiente descripción:

```
In [6]: %%bash
        vector=( 4 2 8 9 )
        echo ${vector[0]}
        echo ${vector[3]}
        echo ${vector[@]}
        echo ${#vector[@]}
        echo ${vector[*]}
        echo ${vector[@]:2}
```

```
4
9
4 2 8 9
4
4 2 8 9
8 9
```

1.2.3 Funciones

Las funciones poseen una sintaxis tipo C, donde indicamos que se trata de una función con la instrucción `function` seguida del nombre de función y la agrupación de parentesis, y el contenido de la función entre llaves.

```
In [7]: %%bash
        nombre="Edward"
        function imprimir_saludo(){
        echo "Hola $nombre"
        }
        imprimir_saludo
```

```
Hola Edward
```

El paso de variables a las funciones se realiza con ayuda de las variables globales.
Es conveniente que las funciones de **ayuda** se creen al inicio tras la indicación de las variables de inicio. La idea de las funciones de ayuda son funciones que se ejecutan bajo condiciones como directamente un termino clave para mostrar la ayuda o cuando el ingreso de los argumentos es erroneo.

```
In [8]: %%bash
        function imprimir_saludo(){
        echo "Hola $0 $1"
        }
        imprimir_saludo Edward
```

```
Hola bash Edward
```

El paso de variables a funciones y en general *scripts* corresponden al orden de parametros, con la secuencia `$` seguido del número en el orden de la lista, iniciando en `$0` para el nombre de la función o *script*.

1.2.4 Cuerpo principal

Esta sección como el nombre lo dice es la parte que logra la esencia del *script* y donde se hace necesario introducir las estructuras de control.

1.3 Operadores

Operador	Simbolo
Igualdad	`=`
Asignación	`=`
Suma	`+`
Resta	`-`
Multiplicación	`*`
División	`/`
Modulo	`%`
Menor que	`-lt`
Mayor que	`-gt`
Menor igual	`-le`

Operador	Simbolo
Mayor igual	`-ge`
Igual	`-eq`
Diferente	`ne`

Ademas de estos simbolos, para las operaciones aritmeticas se hace necesario indicar la expresión `let` o las indicaciones propias de bash para evaluacion de expresiones (no solo aritmeticas).

```
In [9]: %%bash
        let C=1+1
        echo "C es $C"
        D=$[2+1]
        echo "D es $D"
        E=$((1+3))
        echo "E es $E"

C es 2
D es 3
E es 4
```

Si requieren operaciones más complejas que operaciones con enteros, se debe requerir a una aplicación externa. Se podría hacer llamado a python si lo desea, o a la calculadora por consola.

```
In [10]: %%bash
         echo "Con bash"
         A=$[1/2]
         echo "A es $A"
         echo "con bc"
         echo 1/2 | bc -l
         echo "con python"
         printf "print 1./2" | python

Con bash
A es 0
con bc
.50000000000000000000
con python
0.5
```

1.4 Estructuras de control

1.4.1 Condicionales

La estructura condicional tiene el esquema

```
if [condicion]; then
```

```
    sentencia
elif [condicion]; then
    sentencia
else
    sentencia
fi
```

Los corchetes pueden ser simples o dobles según el efecto de evaluación de la condición. Con corchetes simples se requiere indicar la variable entre comillas mientras que con corchetes dobles se puede omitir las comillas.

```
In [11]: %%bash
         T1="foo"
         T2="bar"
         if [ "$T1" = "$T2" ]; then
             echo expression evaluated as true
         else
             echo expression evaluated as false
         fi
```

```
expression evaluated as false
```

1.4.2 Ciclos

```
In [12]: %%bash
         for i in $( ls ); do
             echo item: $i
         done
```

```
item: Jupyter
item: Notebook
item: Basico.ipynb
item: Jupyter+Notebook+Basico.pdf
item: Jupyter
item: Notebook
item: Intermedio.ipynb
item: Jupyter+Notebook+Intermedio.pdf
item: jupyter.png
item: LICENSE
item: Linux
item: Bash.ipynb
item: Linux
item: Basico.ipynb
item: Linux+Basico.pdf
item: presentacion_herramientas.md
item: presentacion_herramientas.pdf
item: README.md
item: Taller_1.ipynb
```

item: Taller_1.pdf

In [13]:
```
%%bash
for i in `seq 1 10`; do
    echo $i
done
```

```
1
2
3
4
5
6
7
8
9
10
```

In [14]:
```
%%bash
for ((I=0; I<=10; I++)); do
    printf "$I\n"
done
```

```
0
1
2
3
4
5
6
7
8
9
10
```

In [15]:
```
%%bash
COUNTER=0
while [  $COUNTER -lt 10 ]; do
    echo The counter is $COUNTER
    let COUNTER=COUNTER+1
done
```

```
The counter is 0
The counter is 1
The counter is 2
The counter is 3
```

```
The counter is 4
The counter is 5
The counter is 6
The counter is 7
The counter is 8
The counter is 9
```

```
In [16]: %%bash
         COUNTER=20
         until [  $COUNTER -lt 10 ]; do
             echo COUNTER $COUNTER
             let COUNTER-=1
         done
```

```
COUNTER 20
COUNTER 19
COUNTER 18
COUNTER 17
COUNTER 16
COUNTER 15
COUNTER 14
COUNTER 13
COUNTER 12
COUNTER 11
COUNTER 10
```

1.5 Referencias

1. BASH Programming - Introduction HOW-TO.
2. Advanced Bash-Scripting Guide.
3. Writing shell scripts.
4. Teoría de bash.