Bash con ejemplos, parte 2

1. Más fundamentos de programación en bash

Argumentos

Empezaremos con unas nociones básicas sobre el manejo de argumentos en la línea de comandos, para luego pasar a algunos esquemas básicos de bash.

En el programa de ejemplo en el artículo introductorio, usamos la variable de entorno "$1", que se refería al primer argumento suministrado en la línea de comandos. Podemos usar "$2", "$3", etc. para referirnos al segundo y tercer argumentos, respectivamente, y así de forma sucesiva. Aquí tenemos un ejemplo:

Listado de Código 1.1: Referente a los argumentos pasados al guión:

```
#!/usr/bin/env bash

echo el nombre del guión es $0
echo el primer argumento es $1
echo el segundo argumento es $2
echo el decimoséptimo argumento es $17
echo el número de argumentos es $#
```

El ejemplo es autoexplicativo, excepto por dos detalles. Primero: "$0" se expande al nombre del propio guión, tal y como se ha llamado en la línea de comandos; y, segundo: "$#" se expande al número de argumentos pasados al guión. Juega un poco con el guión anterior, pasándole distintos tipos de argumentos para familiarizarte con su manera de funcionar.

A veces resulta útil poder referenciar todos los argumentos en un solo bloque desde el guión. Para este propósito bash nos ofrece la variable de entorno "$@", que se expande a todos los parámetros suministrados, concatenados y separados por espacios en blanco. Veremos un ejemplo de su uso al estudiar los bucles, más adelante en este mismo artículo.

Esquemas de programación en bash

Si alguna vez has programado en un lenguaje procedimental como C, Pascal, Python, or Perl, estarás familiarizado con los esquemas estándares de la programación, como las sentencias "if", los bucles "for" y algunos más. Bash tiene sus propias versiones de los mismos. En las próximas secciones introduciré algunos esquemas de bash y explicaré las diferencias entre estos esquemas y otros similares con los que puede que te hayas encontrado al usar otros lenguajes de programación. Si no has programado demasiado antes, no te preocupes. Hay ejemplos e información suficientes para que puedas seguir el texto.

Amor condicional

Si alguna vez has programado algo, relacionado con archivos, en C, sabrás que se requiere un esfuerzo significativo para saber si un fichero dado es más nuevo que otro. Eso es porque C no tiene una sintaxis interna para realizar dicha comparación; en lugar de eso dos llamadas a stat() y dos estructuras stat son necesarias para poder realizar dicha comparación "a mano". En contraste, bash puede realizar esta operación

mediante operadores estándar que posee. Por eso, determinar si "/tmp/miarchivo es legible" es tan sencillo como comprobar si "$mivar es mayor que cuatro".

La lista siguiente muestra los operadores de comparación más frecuentemente usados en bash. También encontrarás un ejemplo de como usar cada opción de forma correcta. El ejemplo podría usarse inmediatamente después del "if":

Listado de Código 1.2: Operadores de comparación en bash

```
if [ -z "$mivar" ]
then
     echo "mivar no está definida"
fi
```

A veces hay varias formas de realizar una misma comparación, los siguientes ejemplos funcionan de forma idéntica:

Listado de Código 1.3: Dos formas de hacer una comparación

```
if [ "$mivar" -eq 3 ]
then
     echo "mivar igual a 3"
fi

if [ "$mivar" = "3" ]
then
     echo "mivar igual a 3"
fi
```

En el ejemplo de arriba, ambas comparaciones hacen lo mismo, pero, mientras que la primera una un operador de comparación aritmético, la segunda usa un operador de comparación de cadenas de texto.

Peculiaridades de la comparación de cadenas

Si bien la mayoría de las veces se pueden omitir las comillas dobles alrededor de las cadenas y las variables que las contienen, no se considera una buena práctica de programación. ¿Por qué? Todo funcionará perfectamente mientras la variable no contenga un espacio o un carácter de tabulación. En ese caso bash se confundirá. Aquí hay un ejemplo de comparación que fallará por este motivo:

Listado de Código 1.4: Ejemplo de comparación defectuosa

```
if [ $mivar = "foo bar oni" ]
then
     echo "si"
fi
```

En el ejemplo de arriba, la condición se cumplirá tan solo si $mivar contiene la cadena "foo". Si contuviera una cadena con espacios, como "foo var oni", el guión fallará con este error:

Listado de Código 1.5: Error cuando la variable contiene espacios

```
[: too many arguments
```

En este caso, los espacios en "$mivar" (que contiene "foo var oni") confunden al

intérprete bash. Tras expandir "$mivar", la comparación queda de esta forma:

Listado de Código 1.6: Comparación final

```
[ foo bar oni = "foo bar oni" ]
```

Si la variable de entorno no se ha puesto entre comillas dobles, bash piensa que se han colocado más argumentos de la cuenta entre los corchetes. La solución obvia a este problema pasa por encerrar el argumento entre comillas dobles. Recuerda: si tomas el buen hábito de poner siempre tus variables entre comillas dobles, eliminarás de raíz muchos errores similares de programación. Así es como esta comparación debería haberse escrito:

Listado de Código 1.7: Forma correcta de escribir comparaciones

```
if [ "$mivar" = "foo bar oni" ]
then
    echo "si"
fi
```

Este código funcionará como se espera, sin sorpresas desagradables.

Nota: Si quieres que tus variables de entorno se expandan, debes usar comillas dobles. Recuerda que las comillas simples desactivan la expansión de variables y del historial.

Esquemas de bucle: "for"

Ahora que hemos comentado las sentencias de bifurcación condicional "if" empezaremos con los bucles. El bucle "for" estándar. Aquí hay un ejemplo básico:

Listado de Código 1.8: Ejemplo básico de for

```
#!/usr/bin/env bash

for x in uno dos tres cuatro
do
    echo número $x
done

Salida:
número uno
número dos
número tres
número cuatro
```

¿Qué ha pasado exactamente? La parte "for x" del bucle define una variable que llamaremos variable de control del bucle, que se llama "$x", y a la cual se le asignan de forma sucesiva los valores "uno", "dos", "tres" y "cuatro". Tras cada asignación, el cuerpo del bucle (la parte entre "do" y "done") se ejecuta una vez. En el cuerpo nos referimos a la variable de control del bucle "$x" usando la sintaxis estándar de bash para la expansión de variables, como se haría con cualquier otra variable de entorno. For siempre acepta una lista de palabras tras "in". En este caso hemos usado cuatro palabras en castellano, pero la lista de palabras puede contener también nombres de archivo e incluso comodines. El siguiente ejemplo ilustra como usar los comodines estándar de bash en un bucle for:

Listado de Código 1.9: Usando comodines estándar del intérprete

```
#!/usr/bin/env bash

for miarchivo in /etc/r*
do
    if [ -d "$miarchivo" ]
    then
      echo "$miarchivo (dir)"
    else
      echo "$miarchivo"
    fi
done

salida:

/etc/rc.d (dir)
/etc/resolv.conf
/etc/resolv.conf~
/etc/rpc
```

Este código itera sobre cada archivo en /etc que empiece con una "r". Para ello, bash, primero expande el comodín en /etc/r*, reemplazando esa ruta con la cadena /etc/rc.d /etc/resolv.conf /etc/resolv.conf~ /etc/rpc antes de iterar. Una vez dentro del bucle, el operador condicional "-d" se usa en dos líneas que hacen dos cosas distintas, dependiendo de si "miarchivo" es un directorio o no. Si lo es, entonces la cadena " (dir)" se añade al final de la línea.

Podemos usar múltiples comodines o incluso variables de entorno en la lista de palabras:

Listado de Código 1.10: Comodines múltiples y variables de entorno

```
for x in /etc/r??? /var/lo* /home/drobbins/mystuff/* /tmp/${MYPATH}/*
do
    cp $x /mnt/mydira
done
```

Bash ejecutará todas las expansiones de comodines y de variables que sean posibles, creando -potencialmente- una muy larga lista de palabras.

Si bien todos los ejemplos de expansión de comodines se han realizado con rutas absolutas, también se pueden usar rutas relativas, como estas:

Listado de Código 1.11: Usando rutas relativas

```
for x in ../* miscosas/*
do
    echo $x es un fichero inútil
done
```

En el anterior ejemplo, bash expande los comodines en relación al directorio actual. Tal y como se usarían rutas relativas en la línea de comandos. Juega un poco con la expansión de comodines. Ten en cuenta que, si usas rutas absolutas con tus comodines, bash expandirá el comodín a una lista de rutas absolutas. De cualquier otra forma, bash usará rutas relativas en la lista de palabras resultante de la expansión. Si solo quieres referirte a los archivos en el directorio activo (por ejemplo, cuando escribas for x in *), la lista resultante no tendrá ningún tipo de prefijo de ruta añadido. Recuerda que la información de ruta precedente se podría eliminar, de todas formas, con basename, tal y como en este ejemplo:

Listado de Código 1.12: Recortar ruta antepuesta a un nombre de archivo

```
for x in /var/log/*
do
    echo `basename $x` es un fichero contenido en /var/log
done
```

Muchas veces puede ser útil realizar bucles que operen sobre los parámetros suministrados en la línea de comandos. El siguiente es un ejemplo sobre como usar la variable "$@", que se introdujo al principio de este mismo artículo:

Listado de Código 1.13: Ejemplo de uso de la variable $@

```
#!/usr/bin/env bash

for cosa in "$@"
do
    echo has escrito ${cosa}.
done

salida:

$ allargs hola a todos
has escrito hola
has escrito a
has escrito todos
```

Aritmética del intérprete

Antes de aprender un segundo tipo de esquema de bucle, es una buena idea aprender algo sobre la aritmética de bash. Si, es cierto, bash puede realizar operaciones simples con enteros. Tan solo es necesario encerrar la operación entre estos dos pares: "$((" y "))", y bash evaluará la expresión. Aquí hay algunos ejemplos:

Listado de Código 1.14: Contando en bash

```
$ echo $(( 100 / 3 ))
33
$ mivar="56"
$ echo $(( $mivar + 12 ))
68
$ echo $(( $mivar - $mivar ))
0
$ mivar=$(( $mivar + 1 ))
$ echo $mivar
57
```

Ahora que ya estás familiarizado con las operaciones matemáticas, es el momento de presentar dos nuevos esquemas iterativos: "while" y "until".

Más esquemas iterativos: "while" y "until"

Una sentencia "while" se ejecutará iterativamente mientras una condición dada sea cierta, y tiene el siguiente formato:

Listado de Código 1.15: Modelo de sentencia while

```
while [ condición ]
do
    comandos
```

```
done
```

Las sentencias "while" se pueden usar típicamente para ejecutar una tarea un número dado de veces, como en el ejemplo siguiente:

Listado de Código 1.16: Repetir la sentencia 10 veces

```
mivar=0
while [ $mivar -ne 10 ]
do
    echo $mivar
    mivar=$(( $mivar + 1 ))
done
```

Como puedes ver, usamos las expansiones matemáticas de bash para asegurarnos de que, eventualmente, la condición se rompa, y así también el bucle.

Las sentencias "until" nos proveen con la funcionalidad inversa de "while". Repiten el bucle mientras la condición sea falsa, aquí hay un bucle "until" que funciona de forma idéntica al ejemplo "while" anterior:

Listado de Código 1.17: Ejemplo de bucle until

```
mivar=0
until [ $mivar -eq 10 ]
do
    echo $mivar
    mivar=$(( $mivar + 1 ))
done
```

Sentencias case

Las sentencias case "Case" son esquemas condicionales, un ejemplo:

Listado de Código 1.18: Ejemplo de código usando case

```
case "${x##*.}" in
     gz)
           gzunpack ${SROOT}/${x}
           ;;
     bz2)
           bz2unpack ${SROOT}/${x}
           ;;
     *)
           echo "formato de archivo no reconocido."
           exit
           ;;
esac
```

En este ejemplo, bash expande "${x##*.}". En el programa, "$x" es el nombre de un archivo, y "${x##*.}" tiene el efecto de recortar todo el texto excepto el que vaya tras el último punto en el nombre del archivo. Tras eso bash compara lo que queda con los valores listados en los apartados marcados con ")". El resultado de "${x##*.}" se compara por tanto con "gz", luego "bz2" y finalmente "*". Si "${x##*.}" coincide con alguno de dichos patrones o cadenas, las líneas tras el paréntesis ")" se ejecutan, hasta los dos punto y coma siguientes. En ese punto, bash continúa ejecutando las líneas que haya tras la palabra de cierre "esac". Si ningún patrón o cadena coincide, ninguna línea de código dentro de case se ejecuta. No obstante, en este caso concreto, al menos una línea será ejecutada, debido al uso de un comodín "*" como una de las posibilidades,

que coincidirá con todo lo que no coincidan "gz" o "bz2".

Funciones y contexto

En bash, puedes definir funciones, de forma similar a como se definen en los lenguajes procedimentales como Pascal, C y otros. Dichas funciones pueden aceptar argumentos de forma similar a como los aceptan los guiones. Aquí tenemos una definición de función de ejemplo:

Listado de Código 1.19: Ejemplo de definición de función

```
tarview() {
    echo -n "Mostrando contenido de $1 "
    if [ ${1##*.} = tar ]
    then
        echo "(tar descomprimido)"
        tar tvf $1
    elif [ ${1##*.} = gz ]
    then
        echo "(tar comprimido con gzip)"
        tar tzvf $1
    elif [ ${1##*.} = bz2 ]
    then
        echo "(tar comprimido con bzip2)"
        cat $1 | bzip2 -d | tar tvf -
    fi
}
```

Nota: Este mismo ejemplo de arriba podría escribirse usando una sentencia "case". ¿Podrías tú decirnos como?

Arriba definimos una función llamada "tarview" que acepta un argumento, un tarball de alguna clase. Cuando la función se ejecuta, identifica el tipo de tarball al que el argumento apunta, (si es un tarball descomprimido, o comprimido con bzip2 o gzip), imprime una línea informativa, y muestra el contenido del tarball. Así es como debemos llamar a esta función (da igual que sea desde un guión o desde la misma línea de comandos, tras ser escrita en el mismo intérprete, pegada, o volcada usando source.

Listado de Código 1.20: Llamando a la función de arriba

```
$ tarview shorten.tar.gz
Displaying contents of shorten.tar.gz (gzip-compressed tar)
drwxr-xr-x ajr/abbot         0 1999-02-27 16:17 shorten-2.3a/
-rw-r--r-- ajr/abbot      1143 1997-09-04 04:06 shorten-2.3a/Makefile
-rw-r--r-- ajr/abbot      1199 1996-02-04 12:24 shorten-2.3a/INSTALL
-rw-r--r-- ajr/abbot       839 1996-05-29 00:19 shorten-2.3a/LICENSE
....
```

Como puedes ver, los argumentos usan el mismo mecanismo de referenciación dentro de la función que los usados en un guión para referenciar a los argumentos de línea de comandos. La macro "$#" se expande también al número de argumentos. La única cosa que puede no funcionar completamente como se espera es la variable "$0", que, o bien se expande a la cadena "bash" (si la función se llamó desde la línea de comandos, de forma interactiva, o bien al nombre del guión que la llamó.

Nota: Úsalas de forma interactiva: No olvides que las funciones, como la de arriba, pueden ser incluidas en tu ~/.bashrc o tu ~/.bash_profile para que estén disponibles siempre que estés usando bash.

Visibilidad de las variables

A menudo necesitarás crear variables de entorno dentro de una función. Si bien es posible, hay algo que deberías saber. En la mayoría de lenguajes compilados (como C), cuando se crea una variable dentro de una función, ésta es colocada en un contexto separado. Si en C defines una función llamada mifunción, y dentro defines una variable "x", ninguna otra variable "x" definida en una función diferente se verá afectada, eliminando efectos colaterales.

Ésto, que es completamente cierto en C, no lo es en bash. En bash, cuando creas una variable de entorno en cualquier lugar dentro de una función, se añade al contexto global. Esto significa que siempre se corre el peligro de sobreescribir otra variable, y que la variable trascenderá al lapso de vida de la función:

Listado de Código 1.21: Tratamiento de variables en bash

```
#!/usr/bin/env bash

mivar="hola"

mifunc() {

    mivar="un dos tres"
    for x in $mivar
    do
        echo $x
    done
}

mifunc

echo $mivar $x
```

Cuando este guión se ejecuta, produce la salida "un dos tres tres", mostrando como la variable "$mivar" definida en la función ha sobreescrito a la global "$mivar", y como la variable de control "$x" continúa existiendo incluso tras salir de la función en la que fue definida, y a su vez sobreescribiendo cualquier otra posible "$x" que estuviera ya definida.

En este simple ejemplo, el error es fácil de ver y se puede compensar tan solo cambiando los nombres de las variables. Sin embargo, la mejor forma de acometer el problema pasa por prevenir la posibilidad de que ninguna variable pueda sobreescribir a una definida de forma global, mediante el uso del comando "local". Cuando se usa el comando "local" para crear variables dentro de una función, las mismas se mantendrán en un entorno local a la función, y no interferirán con ninguna otra variable global. A continuación, un ejemplo de como implementar dicha definición, para que ninguna variable global sea sobreescrita:

Listado de Código 1.22: Asegurándose de no sobreescribir variables globales

```
#!/usr/bin/env bash

mivar="hola"

mifunc() {
    local x
    local mivar="un dos tres"
    for x in $mivar
    do
        echo $x
```

```
    done
}

mifunc

echo $mivar $x
```

Esta función producirá como resultado "hola" -- la "$mivar" global no se sobreescribe, y "$x" no existirá fuera de mifunc. En la primera línea de la función creamos "$x", una variable local que se usa después, mientras en la segunda línea (local mivar="one two three") creamos otra "$mivar", y le asignamos un valor. La primera forma es ideal para las variables de control de bucles, ya que no podemos hacerlo directamente en la forma "for local x in $mivar". Esta función no sobreescribe ninguna variable anterior. Se recomienda diseñar de esta forma todas las funciones, tan solo se deberá omitir la declaración local de las variables cuando se pretenda, de forma consciente, escribir en una variable global.

Resumiendo

Ahora que hemos cubierto lo más esencial de la funcionalidad de bash, es hora de ver como desarrollar una aplicación entera en bash. En el próximo capítulo veremos como hacer eso. ¡Hasta entonces!

Sumario: En su artículo introductorio sobre bash, Daniel Robbins nos habló sobre algunos elementos básicos del lenguaje interpretado, y algunas razones para usar bash. En este segundo asalto, Daniel lo retoma donde lo dejó, comenzando con algunas construcciones básicas en bash, como los condicionales (if-then), los bucles, y algunas cosas más.