Documento
Grupo de Meteorología, Santander (CSIC-UC)
D-GMS:2009.1

Un pequeño manual de Shells

L. Fita
Grupo de Meteorología, Universidad de Cantabria, Spain
versión:2–*Septiembre 2009*
correspondencia: lluis.fita@unican.es

Resumen

Con las 'shells' o scripts se consigue la automatización de qualquier comando o acción que se ejecute en linux. Esto permite crear sistemas muy complejos de ejecución que otorgan una gran potencialidad al sistema operativo linux. En estas notas se introducen los concetos básicos para empezar a desarrollar shells.

1. Introducción

Sin lugar a dudas, una de las potencialidades del Linux es su línea de comandos. Aunque para muchos usuari@s 'entorno gráfico'-dependientes lo puedan encontrar una aberración. Dado que la mayoría de aplicaciones de linux se pueden ejecutar desde la línea de comandos, con los *shells* o *scripts* podemos automatizar, sincronizar o relacionar dichas aplicaciones. Para los usuari@s con más antigüedad equivaldrían a los antiguos archivos de lotes (los viejos `.bat`).

Hay muy pocas limitaciones en cuanto a uso de los *shells* en linux. Su principal problema es el poco control de errores que ofrece y la dificultad de interpretación/búsqueda de errores de scripts heredados. Al estar basada en comandos directos del sistema operativo, no precisa de la instalación de ningún software/paquete extra. Es un lenguaje bastante intuitivo y la dificultad suele recaer mas en saber qué instrucción del sistema operativo utilizar y el manejo de la misma. Esto hace muchas veces que shells que hagan los mismo hay tantas como usuari@s de linux...

El linux ofrece varias *shells* distintas: `ksh, csh, tcsh, Bourne Shell`, etc. que tienen pequeñas diferencias de lenguaje. La mas primitiva es la *Bourne Shell* y de las más avanzadas la `tcsh`.

En esta nota técnica, sólo se dan unas pinzeladas de lo que se puede hacer con la *shell*. Se recomienda encarecidamente visitar manuales más extensos para aprofundizar en su conocimiento (mirar anexo).

2. Comandos en bash

La primera instrucción que se pone en un *Shell*-script es que *shell* queremos utilizar. Para utilizar la `bash`, en la primera línea se escribe: (por regla general, los archivos de *bash* se les pone la extensión `.bash`)

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

NOTAS:

1. Esta instrucción no seria necesaria de por sí, ya que se pueden ejecutar las scripts llamándolos directamente con la *shell* necesária por ejemplo:
   `/bin/bash archivo.bash`
2. La línea de comandos ya tiene por defecto una *shell*, que puede coincidir con el tipo de lenguaje utilizado en la script. Para saber que *shell* se dispone por defecto se ejecuta el comando:
   `echo $SHELL`

En lo que sigue, las escripts se describirán con la llamada a la *shell* en su encabezamiento. Para hacer que una script sea ejecutable le tendremos que dar permisos de ejecución. Para eso.

```
chmod +x archivo.bash
```

Los carácteres básicos en una script.

- **#** comentario
- \ continuación de instrucción a la siguiente línea
- **;** poner dos instrucciones en la misma línea
- | captura de salida de la instrucción anterior para la siguiente (*pipe*)
- > captura de la salida de un comando a un fichero de salida (normalmente)
- >> captura de la salida de un comando a un fichero de salida (normalmente) a partir de la última línea
- **&** ejecución de un comando liberando el terminal (sin esperar su finalización, en modo *'background'*)
- **$()** captura el valor de la instrucción que esté entre paréntesis

2.1 Instrucciones básicas

Las instrucciones mas básicas de *shell* son:

- **echo:** Con ella se escribe la salida en pantalla
  ```
  #!/bin/bash
  echo "Hola mundo"
  ```
  ```
  Hola mundo
  ```
- **variables:** Las variables en una script no se tienen que declarar. Por lo gerneral no hay tipos de variables y su tratamiento dependerá con que comando las tratemos.

Las variables en una script se llaman con el símbolo **$**. Así se llamaría a *pi* como `$ pi` o `${pi}`. **NOTA: El nombre de una variable de shell no puede empezar con un número**. Así por ejemplo:

```
#/bin/bash
pi=3.141593
echo "pi= "${pi}" el numero"
```

```
pi= 3.141593 el numero
```

- **condicionales:**
  - **if:** Con ella se hacen las estructuras condicionales mas simples. Su estructura es del tipo:

```
if test [condicion]
then
  (...)
else
  (...)
fi
```

El análisi de la condición se hace por medio de distintas opciones, algunas de las cuales se detallan:

-a El archivo existe
-d El archivo existe y es un directorio
= Los strings son iguales
-eq, -ne, -lt, -le, -gt, -ge comparaciones aritméticas estándares

Al escribir `if test ! [condición]` la condición pasa a ser falsa. Un ejemplo con la instrucción.

```
#!/bin/bash
num1=1
num2=3
if test $num1 -gt $num2
then
  echo "El numero 1 es mas \
  grande que el 2"
  echo "num1 > num2 -->"${num1}\
  " > "${num2}
else
  echo "El numero 1 es mas \
  pequeno que el 2"
  echo "num1 < num2 -->"${num1}\
  " < "${num2}
fi
```

```
El numero 1 es mas pequeno que el 2
num1 < num2 –>1 < 3
```

  - **case:** Con ella se hacen estructuras de múltiples opciones. Su estructura es del tipo

```
case [variable] in
[opci\'on 1])

;;
[opci\'on 2])

;;
(...)
[opci\'on n])

;;
*)
[todos los otros valores]

esac
```

Un ejemplo muy básico con un programa de referencia que se va a llamar `romanos.bash`:

```
#!/bin/bash
num=3
case $num in
1)
rom='i'
palabra='romano'
;;
2)
rom='ii'
palabra='romanos'
;;
3)
rom='iii'
palabra='romanos'
;;
4)
rom='iv'
palabra='romanos'
;;
5)
rom='v'
palabra='romanos'
esac
echo "Tenemos "${num}${palabra}\
": "${rom}
```

```
./romanos.bash
Tenemos 3 romanos: iii
```

- **Bucles:** Hay distintas maneras de hacer un bucle. Las mas usuales son:
  - **for:** Con este se hacen bucles normalmente a un tipo de opciones. Su estructura general es la que sigue:

```
for [variable] in [grupo de valores]
do
  (...)

done
```

También se dispone de las instrucciones `continue/break`

  - **while:** Con este se hacen bucles meintras que la condición se cumpla, su estructura básica:

```
while [condici\'on]
do
  (...)
done
```

Un ejemplo básico de bucle añadiéndolo en el ejemplo anterior:

```
#!/bin/bash
numeros='3 4 1 2'
for num in $numeros
do
  case $num in
  1)
    rom='i'
    palabra='romano'
  ;;
  2)
    rom='ii'
    palabra='romanos'
  ;;
```

```
    3)
      rom='iii'
      palabra='romanos'
    ;;
    4)
      rom='iv'
      palabra='romanos'
    ;;
    5)
      rom='v'
      palabra='romanos'
    esac
    echo "Tenemos "${num}" "${palabra}\
    ": "${rom}
done
```

```
./romanos.bash
Tenemos 3 romanos: iii
Tenemos 4 romanos: iv
Tenemos 1 romano: i
Tenemos 2 romanos: ii
```

- **read:** Con esta instrucción se interactúa con el usuario esperando la entrada en el terminal. En el ejemplo anterior se añade (quitando el bucle):

```
#!/bin/bash
echo "Quantos romanos tenemos?"
read num
case $num in
1)
  (....)
esac
echo "Tenemos "${num}${palabra}\
": "${rom}
```

Por ejemplo:

```
./romanos.bash
Cuantos romanos tenemos?
3
Tenemos 3 romanos: iii
```

- **exit:** Con esta instrucción paramos la ejecución de una script. Si en el ejemplo anterior se añade una condición mas en el `case`

```
(...)
6)
  echo "Hay demasiados romanos!"
  exit
;;
(...)
```

Nos daría:

```
./romanos.bash
Cuantos romanos tenemos?
6
Hay demasiados romanos!
```

- **sleep:** Con esta instrucción paramos una script un determinado número de segundos [n]. Su semántica:

```
sleep [n]
```

2.1.1 Funciones

En las shells es también recomendable hacer y trabajar con funciones. Se adjunta un ejemplo muy sencillo:

```
#!/bin/bash
suma () {
  arg1=$1
  arg2=$2
  val=‘expr $arg1 + $arg2‘
  echo $val
}

echo "Dame a"
read a

echo "Dame b"
read b

echo "su suma: "$(suma $a $b)
```

Nos da:

```
Dame a
1
Dame b
2
su suma: 3
```

2.2 Argumentos

Las scripts de *Shell* tienen la capacidad de poder trabajar hasta con 9 argumentos cuando se las llama des de la línea de comandos. Dentro de la script se las llama cómo `$1`, `$2`, ..., `$9`. Se considera argumento a cada 'palabra' (conjunto de carácteres separados por un espacio) que sigue a la script. Dentro de la script tenemos otras variables relacionadas con $, tales como:

- **$0** Nombre de la script ejecutándose
- **$*** Argumentos mandados a la script
- **$#** Número de argumentos pasados a la script

Asi por ejemplo retocando el código de la script anterior.

```
#!/bin/bash
echo "Script: "$0
echo "Argumentos: "$*
echo "Numero de argumentos: "$#
num=$1
palabra=$2
case $num in
1)
  rom='i'
;;
2)
  rom='ii'
  palabra=${palabra}s
;;
3)
  rom='iii'
  palabra=${palabra}s
;;
4)
```

```
  rom='iv'
  palabra=${palabra}s
;;
5)
  rom='v'
  palabra=${palabra}s
;;
6)
  palabra=${palabra}s
  echo "Hay demasiados "${palabra}
  exit
esac
echo "Tenemos "${num}" "${palabra}\
": "${rom}
```

Nos da:

```
./romanos.bash 4 galo
Script: ./romanos.bash
Argumentos: 4 galo
Numero de argumentos: 2
Tenemos 4 galos: iv
```

Hay otras variables estándard de linux que se pueden llamar des de una script. Unas cuantas son (para *bash*):

- **$HOME:** Home del usuario
- **$HOSTNAME:** Nombre de la máquina
- **$SHELL:** Shell que interpreta los comandos
- **$PATH:** Path del sistema
- **$LANG:** Lenguaje del sistema
- **$PWD:** Actual posición en el sistema de archivos
- **$RANDOM:** Un número aleatorio entre 0 y 32767

3. Interacción con instrucciones linux

Las scripts pueden interactuar con qualquier comando de linux. Algunos ejemplos muy senzillos se detallan a continuación. Se recomienda mirar la aydua en línea (`man instruicción`) de estas y de muchas mas instrucciones que estan en la línea de comandos. Las salidas de las insrtucciones se pueden tratar de distintas maneras, algunas de ellas:

- `comando`: Salida en pantalla de los resultados
- `comando > archivo`: Se captura la salida del comando en un fichero
- `comando >> archivo`: Se captura la salida del comando en un fichero volcando su contenido después de la última línea
- `variable='comando'`: Se almacena la salida del comando en la variable

- **expr:** Con este comando se hacen operaciones entre variables. Es de las maneras mas senzillas de hacer operaciones. Las operaciones numéricas se hacen solo con numeros enteros. Es una instrucción sencilla, pero muy completa. Entre sus funciones estan
  - **expr** [**num1**] [**+-*/%**] [**num2**]**:** Sumar, restar, multiplicar, dividir y dar el resto entre el número [num1] y el [num2]
  - **expr substr** [**variable**] [**In**] [**Nc**]**:** De la variable [variable] des de su caracter [In] dar [Nc] carácteres
  - **expr length** [**variable**]**:** Longitud de [variable]
  - **expr index** [**variable**] [**c**]**:** Da la posición del carácter [c] en [variable]
- **pwd:** Da la posición actual en el sistema de archivos (igual que $PWD)
- **date:** Permite trabajar con fechas.
- **grep:** Nos selecciona/deselecciona esas líneas de un fichero que contengan una palabra dada
- **sed:** Es una herramienta muy potente. Una de sus funcionalidades es que nos permite substituir palabras espcíficas dentro de un fichero
- **cat:** Esta instrucción nos da por pantalla el contenido de un fichero.
  - Contenido de un fichero.

    ```
    #!/bin/bash
    ls -l * > ficheros.inf
    ## La fecha de hoy en formato
    ## [aaaa]-[mm]-[dd]
    hoy='date +%Y-%m-%d'
    echo "Los ficheros del directorio \
    que tienen fecha de hoy:"
    cat ficheros.inf | grep ${hoy}
    ```

    ```
    -rw-r–r– 1 lluis lluis 1619 2009-09-30 10:30 datos2.inf -rwxr-xr-x 1 lluis lluis 2707 2009-09-30 10:29 datos.inf -rw-r–r– 1 lluis lluis 80 2009-09-30 10:27 gmuc_2009_12.aux -rw-r–r– 1 lluis lluis 26368 2009-09-30 10:27 gmuc_2009_12.dvi -rw-r–r– 1 lluis lluis 15587 2009-09-30 10:27 gmuc_2009_12.log -rw-r–r– 1 lluis lluis 53518 2009-09-30 10:27 gmuc_2009_12.pdf -rw-r–r– 1 lluis lluis 15807 2009-09-30 10:34 gmuc_2009_12.tex -rwxr-xr-x 1 lluis lluis 186 2009-09-30 10:34 hoy.bash
    ```

  - Escrivir un fichero des de una script. La etiqueta EOF indica hasta donde está el contenido del fichero.

    ```
    (
    cat << EOF
    (...)
    EOF
    ) > fichero
    ```
- **wc:** Contador de líneas, palabras y carácteres
- **tail:** Nos da las [n] líneas últimas de un fichero
- **head:** Nos da las [n] líneas primeras de un fichero

Un ejemplo un poco semnzillo con algunas de las interacciones se muestra a continuación (llamado `instrucciones.bash`):

```
#!/bin/bash
if test $1 = '-h'
then
echo "****************************"
echo "***   Shell para dar los  ***"
echo "***   los ficheros que    ***"
echo "*** cumplan una condicion ***"
```

```
echo "*** dada y con un nombre  ***"
echo "***   de una longitud     ***"
echo "***   superior dada hay   ***"
echo "***    en un directorio   ***"
echo "*****************************"
echo "instrucciones.bash 'DIR'\
(directorio) 'CON'(condicion) \
'LON'(longitud)"
else
ficheros=‘ls -1 $1/*$2*‘
for fichero in ${ficheros}
do
  flong=‘expr length ${fichero}‘
  if test ${flong} -ge $3
  then
    echo "Fichero: '"${fichero}"' \
    longitud de su nombre: "${flong}
  fi
### Fin de los ficheros encontrados
done
fi
```

En el ejemplo anterior se a puesto una primera condición *if* con la que obtenemos la descripción de la script en la misma línea de comandos. Nos da:

```
./instrucciones.bash /usr/bin a 43
Fichero:  '/usr/bin/gnome-default-applications-
properties'  longitud de su nombre: 46
Fichero:  '/usr/bin/nautilus-file-management-
properties'  longitud de su nombre: 44
Fichero:  '/usr/bin/rarian-sk-get-extended-
content-list'  longitud de su nombre: 44
Fichero:  '/usr/bin/scrollkeeper-get-index-from-
docpath'  longitud de su nombre: 44
```

O bien

```
./instrucciones.bash -h
*********************************
**    Shell para dar los     ***
**      los ficheros que     ***
** cumplan una condicion ***
** dada y con un nombre  ***
**       de una longitud     ***
**    superior dada hay     ***
**      en un directorio     ***
*********************************
instrucciones.bash              'DIR'(directorio)
'CON'(condicion) 'LON'(longitud)
```

Un ejemplo bastante mas completo está en el apéndice llamado `sinus_hoy.bash`

3.1 AWK

Con este programa se gana un entorno similar al 'C' para trabajar con contenidos de ficheros o con salidas de intrucciones. Además permite la creación de funciones las cuáles podemos llamar dentro de una script. Tiene una semántica própia distinta a la de Shell. Opera con números reales, funciones matemáticas, formatos de salids, etc. Es muy completo y extenso, se dan tres ejemplos muy senzillos:

- **Captura:** En el siguiente ejemplo se da el tercer y séptimo elemento de la instrucción `ls -l`
  `ls -l | awk '{print $3 $7}'`
- **Funciones:** Se contruyen a partir de las instrucciones BEGIN y/o END. Un ejemplo es la función `column.awk`, la cual da la columna [col] de la línea [row] de un fichero/instruccion:
  ```
  # Gives the value at a given row
  # and column of a matrix
  # col: Col number
  # row: Row number
  // { num_rows++; column[num_rows] = $col }
  END {print column[row]}
  ```
  La llamada de esta función podria ser:
  `ls -l | awk -f olumn.awk row=3 col=7`

Anexo

A. Manuales

Básico: ttp://www.gnu.org/software/bash/manual/bashref.html

Wiki: http://es.wikibooks.org/wiki/El_Manual_de_BASH_Scripting_Bsico_para_Principiantes

Avanzado: http://tldp.org/LDP/abs/html/

AWK: http://www.vectorsite.net/tsawk.html

B. sinus_hoy.bash

```
#!/bin/bash
if test $1 = '-h'
then
echo "**************************"
echo "***  Shell para dinujar ***"
echo "*** una curva sinusoide ***"
echo "***    en funcion del   ***"
echo "*** segundo y el minuto ***"
echo "**************************"
echo "sinus_hoy.bash 'N'(numero de frames) 'SALT'(salto entre frames [seg])"
else
istep=1
while test $istep -le $1
do
## La hora, minuto y segundo de hoy
## en formato [hh], [mm] y [ss]
Hhoy=‘date +%H‘
Mhoy=‘date +%M‘
Shoy=‘date +%S‘
## Generacion de un archivo GNUplot
(
cat << EOF
set terminal png
set output 'sinus_hoy_@HH@@MM@@SS@.png'
A=@HH@/24.
B=@MM@/30.
C=@SS@/30.
set xrange[-3*pi:3*pi]
set label sprintf("A= %.3g", A) at graph 0.1, 0.9
set label sprintf("B= %.3g", B) at graph 0.1, 0.85
set label sprintf("C= %.3g", C) at graph 0.1, 0.8
plot A*sin(B*pi+C*x) t "Asinus(Bpi+C*x)" w l lt 1 lw 3
EOF
) > sinus_hoy.gnu

## Substitucion de las variables
## por sus valores
sed -e 's/'@HH@'/'${Hhoy}'/g' sinus_hoy.gnu > stepA.inf
sed -e 's/'@MM@'/'${Mhoy}'/g' stepA.inf > stepB.inf
sed -e 's/'@SS@'/'${Shoy}'/g' stepB.inf > sinus_hoy_${Hhoy}${Mhoy}${Shoy}.gnu

## Dibujo de la grafica
##
gnuplot sinus_hoy_${Hhoy}${Mhoy}${Shoy}.gnu
```

```
## Visualizacion
##
display sinus_hoy_${Hhoy}${Mhoy}${Shoy}.png &

## Se dejan pasar $2 segundos
sleep $2

## Se elimina la visualizacion
killall display
istep=‘expr $istep + 1‘

### Fin del bucle
done

## Vaciado del directorio
sleep $2
rm *.png
rm *.gnu
fi
```