

Introducción a la programación en bash

Gonzalo Rivero, sfish@rect.unsa.edu.ar

v0.0, 12 Mayo 2003

Es una pequeña introducción a la programación de scripts de shell, asumo que el lector tiene conocimientos (aunque sea mínimos) de programación. Voy a mostrar la sintaxis básica de las órdenes mas usadas con algun ejemplo corto, y al final un ejemplo grande que integre todo

Contents

1	Introducción	2
1.1	Que es bash?	2
1.2	Descarga	2
1.3	Bibliografía	2
2	Comandos	2
2.1	Comandos simples	2
2.2	Ejecución de procesos en "background"	3
2.3	Comandos largos	3
2.4	Comandos múltiples	4
3	Redirección	4
4	Variables	5
4.1	Parámetros	5
4.2	Variables especiales	5
4.3	Variables de entorno	6
4.4	Arrays	6
5	Comodines	6
6	Metacaracteres	6
7	Scripts- programación	7
7.1	Encabezado	7
7.2	Comentarios	7
7.3	Escritura en la salida estandar	7
7.4	Lectura de la entrada estandar	8
7.5	Operaciones aritméticas	8
7.6	Esctructuras de control	8
7.6.1	if then else fi	8

7.6.2	Case	10
7.6.3	while y until	10
7.6.4	for	10
7.7	Funciones	11
8	Comandos utiles	11
9	Un ejemplo largo: front-end para cdparanoia y oggenc	11

1 Introducción

1.1 Que es bash?

Tomado de la página de manual de bash: Bash es intérprete de comandos compatible con sh, que ejecuta comandos leídos de la entrada standard o de un archivo. También incorpora características útiles de los shell Korn y C (ksh y csh).

1.2 Descarga

Es posible que algunas (si no todas) partes de este documento esten mal escritas (desde el punto de vista gramatical), pido disculpas, es que no soy un poeta.

Es mucho mas probable que haya muchos errores de ortografía, pero dicen que herrar es umano :-)

Tambien es posible que muchos contenidos esten incompletos o incorrectos.

En cualquiera de los tres casos, se agradecen aportes, sugerencias, criticas no destructivas, etc...

1.3 Bibliografía

Esto es casi una copia fiel (en este orden) de:

- Uso y programacion de Bash, de Ariel Pereira, disponible en: <http://www.starlinux.net>
- Introduccion a la programación de script de shell con bash, de Guillermo Ontañón Ledesma, disponible en <http://grulla.hispalinux.es>
- Advanced Bash Scripting Guide, de Mendel Cooper <http://www.linuxdoc.org.ar/LDP/abs/html/index.html>
- Manual de bash, `man bash`

2 Comandos

2.1 Comandos simples

Bash suele ser el shell predeterminado, por lo que las formas mas comunes de ingresarle comandos son:

- Ingresar al sistema en alguna consola (en modo texto)
- Abrir alguna terminal dentro de X (xterm, kterm, wterm, gnome-terminal, ...)

Una vez dentro de la terminal, se muestra el prompt:

```
usuario@maquina:directorio_actual$
```

Esto indica que el shell está esperando el ingreso de ordenes, el prompt puede ser diferente en cada sistema (se puede personalizar), y, si el usuario es root, muestra un # al final de la línea en vez del \$. En ambos casos, significa que el shell está listo para ejecutar comandos.

Los comandos normalmente tienen la forma:

```
$ comando parametro1  
parametro2...parametron
```

ejemplo:

```
radiou@operacion:~/Gonzalo$ ps  
PID TTY          TIME CMD  
461 pts/1        00:00:00 bash  
485 pts/1        00:00:00 ps
```

2.2 Ejecución de procesos en "background"

Si necesito ejecutar algun comando que se de antemano que va a tomar mucho tiempo su ejecucion, por ejemplo comprimir un directorio grande como /var, y necesito hacer otras cosas, es posible ejecutar el programa que lo hace en segundo plano, y el shell no esperará que finalice el comando para dejarme ejecutar otros. Esto se hace con el simbolo & al finalizar el comando:

```
usuario@maquina:$comando &
```

ejemplo:

```
usuario@maquina:$ mc &  
[1] 513
```

Luego de lanzar el proceso en background, bash muestra un número entre corchetes y otro fuera de los corchetes. El primero indica el número de tarea en la terminal actual, y el otro muestra el numero de proceso (que es único para cada proceso corriendo en el sistema).

Para traer un proceso que corre en segundo plano a primer plano (solo puede haber un proceso en primer plano) se usa el comando **fg** (de foreground).

Otra forma de mandar un proceso que se esté ejecutando en primer plano a "background" es mediante la combinación de teclas control-z, que pone al proceso en estado "Stopped" y despues usando el comando **bg**.

```
usuario@maquina:$ fg numero_tarea  
usuario@maquina:$ bg numero-tarea
```

2.3 Comandos largos

A veces, la longitud de un comando es tan larga que no entra en una sola linea; cuando esto sucede, automáticamente continuamos escribiendo en la línea inferior, lo cual a veces es difícil de leer porque puede quedar algo entrecortado. Se puede escribir un comando en varias lineas, cortándolo cuando el usuario desee (y no cuando bash desee :-), con una \ al final de la línea y despues ENTER . Con la \ el ENTER pierde su significado y se puede seguir escribiendo en la línea siguiente.

2.4 Comandos múltiples

Se pueden ejecutar varios programas con un solo comando. Si quiero ejecutar varios programas con una sola línea puedo hacerlo de tres maneras distintas:

- Secuencialmente: bash esperará que finalice el comando anterior, y ejecutará el siguiente. Los comandos se separan con un punto y coma. Ejemplo:

```
usuario@maquina:$ ls; who;
BUGS                      CHANGELOG  fish.pl      MP3-Info-0.91.tar.gz  rip~
CDDb_get-1.66             COPYING    INSTALLING   README
CDDb_get-1.66.tar.gz FAQ        MP3-Info-0.91  rip
  PID TTY          TIME CMD
   461 pts/1    00:00:00 bash
  1544 pts/1    00:00:00 ps
```

ejecuta primero ls y luego who

- Simultaneamente: Estos pueden ejecutarse de dos maneras distintas:
 - Independientes unos de otros: los comandos se separan con &. Ejemplo:

```
usuario@maquina:$ ls & ps
```

- Interconectados: la salida del primero, puede ser usada por otro comando. Se separan los comandos con una barra. Ejemplo:

```
usuario@maquina:$ ls|grep bash
bash-intro.aux
bash-intro.log
bash-intro.out
bash-intro.pdf
#bash-intro.sgml#
bash-intro.sgml
bash-intro.sgml~
bash-intro.tex
bash-intro.toc
```

En este caso, la salida estandar de `ls` es enviada a la entrada estandar de `grep`. Lo cual nos lleva a la siguiente seccion

3 Redirección

Si un programa escribe algo en pantalla (stdout, salida estandar), se puede hacer que su salida vaya a un archivo usando el signo `>`. Si el archivo no existe se crea uno nuevo, si existe es sobrescrito; para agregar algo al final de un archivo, se usa `>>`.

Si un programa lee algo del teclado (stdin, entrada estandar), se puede hacer que lea desde un archivo usando el signo `<`.

Cuando un programa quiere leer o escribir un archivo, lo primero que hace es solicitarle al SO su apertura, si la solicitud tiene éxito, el sistema devuelve un descriptor de archivo (un número entero que el sistema utilizará para leer o escribir dicho archivo). Cuando comienza la ejecución de cada programa, el sistema le entrega tres descriptors que identifican tres archivos: entrada estandar (con 0, normalmente el teclado), salida estandar (con 1, normalmente la pantalla) y error estandar (con 2, normalmente la pantalla, stderr).

La forma de redireccionar la entrada de un comando es:

```
usuario@maquina:$ comando < archivo
```

La forma de redireccionar la salida de un comando es:

```
usuario@maquina:$ comando > archivo
```

Pero esto redirecciona la salida estandar. Se puede hacer tambien redireccionamiento entre descriptores. Por ejemplo, si quiero guardar los errores de algun programa, se hace de la siguiente manera:

```
usuario@maquina:$ comando 2 > archivo
```

Si quiero que lo que el programa escribe a algún descriptor, m, a otro descriptor, n, se hace de la siguiente manera:

```
usuario@maquina:$ comando m > & n
```

4 Variables

Para asignar un valor a una variable, no es necesario declararla, basta con escribir su nombre con un signo igual (=) y su valor:

```
usuario@maquina:$ VARIABLE=valor
```

Como la mayoría de los comandos se escriben en minúscula, para diferenciar comandos de variables, se utilizan mayúsculas para el nombre de estas últimas, aunque no es obligatorio.

Para referenciar a una variable, el nombre tiene que ir precedido de \$. Ej:

```
usuario@maquina:$ HOLA=hola
usuario@maquina:$ echo $HOLA
```

4.1 Parámetros

Cuando se ejecuta un script, los parámetros son pasados mediante las *variables parámetro*: El primer parámetro es almacenado en la variable \$1, el segundo en la variable \$2, ..., y el noveno se almacena en la variable \$9.

La variable \$0 contiene el nombre del ejecutable.

4.2 Variables especiales

- \$* Contiene todos los parámetros pasados al script, util cuando son mas de 9
- \$# Contiene el número de parámetros pasados al script
- \$? Contiene el estado de finalización del último comando que es cero si finalizó bien
- \$\$ Contiene el PID (identificador de proceso) del proceso actual

4.3 Variables de entorno

Algunas de las mas comunes son:

- HOME Contiene el directorio home del usuario (/home/usuario)
- PATH Contiene los directorios donde bash busca ejecutables
- PS1 Contiene el simbolo de espera (prompt) de bash. (\$ ó # para root)
- PS2 Contiene el simbolo de espera del shell secundario, cuando se usan comandos de varias lineas
- MAIL Ruta al archivo de la casilla de correo del usuario (normalmente /var/spool/mail/usuario)

4.4 Arrays

Bash provee arrays unidimensionales, no hay un limite máximo en el tamaño de un array, ni es necesario que los miembros de un array se indexen de forma contigua.

La forma de utilizarlas es:

```
var_array[indice]=valor
```

Y para referenciarlas se usa la sintaxis:\${var_array[indice]} un ejemplo:

```
#!/bin/sh
for i in `seq 1 5`
do
    arreglo[$i]=$i
done
echo ${arreglo[3]}
```

Basicamente lo que hace es crear un array "arreglo" de cinco elementos y muestra el tercero al salir del ciclo, para una explicacion de los comandos usados, ver mas abajo.

5 Comodines

- Asterisco (*): Los parámetros que contengan un *, serán reemplazados por nombres de archivo que, en la posicion en que aparece el asterisco, contengan cualquier cadena de caracteres (como en DOS). No reemplaza nombres de archivo que comiencen con un punto.
- Interrogacion (?): Reemplaza un único caracter (como en DOS)
- Corchetes: Sustituye un único caracter, pero dentro de los corchetes se indican los valores que pueden usarse en la sustitución.

6 Metacaracteres

- Barra invertida(\): no se interpreta el siguiente caracter. Ej:

```
usuario@maquina:$ echo *
2dskx-1.0.tgz ABM_gral_imp.h abs-guide.ps.gz ADM_programacion.cpp ADM_programacion_imp.cpp bajada.htm
usuario@maquina:$ echo \*
*
```

- Apostrofos(' '): No se interpreta ningun caracter que esté "dentro" de ellos.
- Comillas(""): Solo se interpretan los caracteres \$,'.' y \, los demas no se interpretan. Ejemplo:

```
usuario@maquina:$ HOLA=hola
usuario@maquina:$ echo "$HOLA*"
hola*
usuario@maquina:$ echo '$HOLA*'
$HOLA*
```

- Comillas invertidas (""). Se utiliza para sustitución de comandos. Es una forma de pasar el resultado de un comando en los parámetros de otro. Ejemplo:

```
usuario@maquina:$ Y0='whoami'
usuario@maquina:$ echo Soy el usuario $Y0
Soy el usuario radiou
```

- " " : Indica el directorio home del usuario. Ejemplo:

```
usuario@maquina:$ echo ~
/home/radiou
```

7 Scripts- programación

7.1 Encabezado

La primera linea de los scripts debe ser:

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

ó

```
#!/bin/sh
```

Si el archivo tiene permisos de ejecución, la primera linea indica que interprete se requiere para su ejecución (en este caso, bash :-)

7.2 Comentarios

Si se agrega el caracter #, bash ignora todo lo que le siga hasta el final de la linea.

7.3 Escritura en la salida estandar

Si se necesita comunicar alguna accion al usuario, se dispone del comando echo:

```
echo [opciones] cadena_de_texto
```

Donde opciones puede ser:

- -m No se escribe el caracter de fin de linea
- -e Habilita la interpretación de caracteres especiales (para que funcione la cadena debe estar entre ""):
 - \a Alerta sonora (bell)

- \b Retroceso
- \c Suprime el caracter de nueva linea
- \f Alimentacion de hoja (util cuando se envia a la impresora)
- \n Retorno de carro y avance de linea
- \r Retorno de carro
- \t Tab horizontal
- \v Tab vertical

7.4 Lectura de la entrada estandar

Se puede hacer de dos formas:

- Mediante linea de comandos, con parámetros
- Interactivamente, con la orden read:

```
read var1 var2 ... varn
```

Si no se le indican variables, la linea ingresada se guarda en la variable `REPLY`

7.5 Operaciones aritméticas

Se utiliza el comando `expr`:

```
expr ARG1 operacion ARG2
```

Operacion puede ser:

+ suma

- resta

/ division

* multiplicación

Es importante la \ antes de * para la multiplicar, si no se pone, bash interpreta el *. Tambien es importante el espacio entre los argumentos y la operación.

7.6 Estructuras de control

7.6.1 if then else fi

Sintaxis:

```
if comando1
then
    comando2
    ...
    comandon
else
    comandon+1
    ...
fi
```


Si comando finaliza exitosamente (devuelve 0), se ejecutan los comandos entre **then** y **else**. Si comando devuelve un numero distinto de cero, se ejecutan los comandos entre **else** y **fi**.

fi (if al revés) marca el final del bloque **if** y no puede faltar.

Comando test Evalúa una expresión y retorna cero si la expresion es verdadera o distinto de cero si es falsa. Sintaxis:

```
test EXPRESSION
```

EXPRESSION puede ser una evaluación de:

- Cadena de caracteres:
 - **-z** CADENA Verdadero si la longitud de la cadena es cero
 - **-m** CADENA Verdadero si la longitud de la cadena es distinta de cero
 - **CADENA1 = CADENA2** Verdadero si las cadenas son iguales
 - **CADENA1 != CADENA2** Verdadero si las cadenas son distintas
 - **CADENA** Verdadero si la cadena no es nula
- Enteros:
 - **INT1 -eq INT2** verdadero si $INT1=INT2$
 - **INT1 -gt INT2** verdadero si $INT1 > INT2$
 - **INT1 -lt INT2** verdadero si $INT1 < INT2$
 - **INT1 -ge INT2** verdadero si $INT1 \geq INT2$
 - **INT1 -le INT2** verdadero si $INT1 \leq INT2$
- Archivos: suponiendo que FILE contiene el path y el nombre de un archivo:
 - **-b** FILE verdadero si existe FILE y es un dispositivo de bloques
 - **-c** FILE verdadero si existe FILE y es un dispositivo de caracteres
 - **-d** FILE verdadero si existe FILE y es un directorio
 - **-e** FILE verdadero si existe FILE
 - **-r** FILE verdadero si existe FILE y tengo permisos de lectura
 - **-w** FILE verdadero si existe FILE y tengo permisos de escritura
 - **-x** FILE verdadero si existe FILE y tengo permisos de ejecución
- Logicos: si EXP1 y EXP2 contienen expresiones lógicas:
 - **!EXP** Negación
 - **EXP1 -a EXP2** Y lógico
 - **EXP1 -o EXP2** O lógico

Escribiendo la expresion entre corchetes, se obtiene el mismo resultado. Ejemplo:

```
$HOLA=hola
if [$HOLA="hola"]
then
    echo $HOLA
fi
```

7.6.2 Case

Se utiliza para condiciones tipo "multiple choice". Sintaxis:

```
case valor in
patron1)
    comando11;
    comando12;
    ...
    comando1n;;
patron2)
    comando21;
    ...
    comando2n;;
...
esac
```

Verifica si **valor** cumple con alguno de los patrones y ejecuta los comandos asociados con la primer coincidencia que encuentra. El **;;** sirve para delimitar cada bloque de comandos, y **esac** (case al vesre :-)) es el fin de case.

7.6.3 while y until

while se utiliza para ejecutar un grupo de comandos mientras se cumpla una condicion:

```
while condicion do
    comando1
    ...
    comandon
done
```

until se utiliza para ejecutar un grupo de comandos mientras una condicion sea falsa:

```
until condicion do
    comando1
    ...
    comandon
done
```

Para forzar la salida de un bloque case, while o until se utiliza la orden break

7.6.4 for

Es otro tipo de bucle, solo que no se basa en en el cumplimiento (o no) de alguna condicion, sino que se ejecuta una cantidad determinada de veces, sintaxis:

```
for VAR in val1 val2 ... valn
do
    comando1
    comando2
    ...
    comandon
done
```

```
<p>La variable VAR contendrá cada uno de los valores val1, val2, ... , valn especificados despues de in. P
<tscreen><verb>
for VAR in 'seq a b'
do
    comando1
    ...
    comandon
done
```

Donde a y b son el valor inicial y final del ciclo, respectivamente, mas informacion de seq: `man seq`

7.7 Funciones

Se pueden agrupar varios comandos bajo un mismo nombre. Sintaxis:

```
[function] nombre_func(){
comando1;
...
comandon;
}
```

la palabra `function` no es obligatoria. Para poder usar una funcion, tiene que haber sido definida antes.

8 Comandos utiles

Se presenta a continuacion una pequeña descripcion de comandos externos a bash, que pueden ayudar en la construccion de nuestros scripts:

- **grep**: busca ocurrencias de un patron (que es una expresion regular) en un archivo o en la entrada estandar, e imprime la linea si encuentra el patron. Mas información en `man grep`
- **sort**: Ordena lineas de texto. Mas información `man sort`
- **tr**: Sustituye o elimina caracteres de la entrada estandar y los escribe en la salida estandar. `man tr`
- **awk**: lenguaje de escaneo y manipulacion de texto. `man awk`
- **sed**: editor de flujos, no interactivo. `man sed`
- **dialog**: sirve para crear cuadros de dialogo en la consola, util cuando tenemos programas interactivos y queremos darle una apariencia "profesional". `man dialog`. Una variante es `gdialog`, que utiliza las librerias de `gnome2` y sirve para crear aplicaciones para `XFree86`
- **whiptail**: otra variante de `dialog`, las ventanas se ven ligeramente distintas

9 Un ejemplo largo: front-end para cdparanoia y oggenc

El siguiente ejemplo es (como su nombre lo indica) un front-end para el `cdparanoia` y `oggenc`. `cdparanoia` extrae pistas de audio de un cd de audio y las graba en formato wav. `oggenc` codifica las pistas a formato Ogg Vorbis, formato que es similar al mp3, pero con dos (en mi humilde opinion) ventajas fundamentales:

- Está totalmente libre de patentes

- Los archivos generados ocupan entre un 30% y un 40% menos que los mp3 y suenan igual o mejor

Para la "interfaz" con el usuario se utiliza dialog, por lo que es necesario instalado dialog, oggenc y cdparanoia.

Las vorbis tools (incluye oggenc) pueden bajarse de <http://www.xiph.org/ogg/vorbis/>.

cdparanoia se baja de <http://www.xiph.org/paranoia/>

dialog se baja de <http://freshmeat.net/redir/dialog/13503/url.homepage/dialog>

En los sitios se incluye documentación adicional de como usar

Intencionalmente no tiene soporte de CDDA (algunos no tenemos conexiones permanentes con la red :-)

Sigue el script:

```
#!/bin/bash
#Ejemplo de sustitucion de comandos:
#Nombre de la banda:
BANDA='dialog --backtitle "Ripper echo x fish" --stdout --inputbox "Nombre de la banda" 0 0 "no se"'
# Nombre del disco:
DISCO='dialog --backtitle "Ripper echo x fish" --stdout --inputbox "Nombre del disco" 0 0 "tampoco s'

#Ejemplo de case:
case 'dialog --backtitle "Ripper echo x fish" --stdout --menu "Cambiar algo?" 0 0 0 D "Disco" B "Banda"
D)
DISCO='dialog --inputbox "Nombre del disco" 0 0'
;;
B)
BANDA='dialog --inputbox "Nombre de la banda" 0 0 "no se"'
;*)
esac

#Como usar el valor de una variable:
#Crear Directorios
mkdir "$BANDA"
cd "$BANDA"
pwd
mkdir "$DISCO"
cd "$DISCO"
pwd

#Redireccion:
#El comando cdparanoia -sQe devuelve información del cdaudio,
#escribe todo en la salida de error
#Calcular cantidad de pistas y demas
cdparanoia -sQe &2> tmp.tmp

#El comando wc cuenta la cantidad de lineas, palabras, bytes de un archivo
CANT='wc -l tmp.tmp | awk '{print $1}''

#Borro las cantidad de lineas con información innecesaria de
#cdparanoia (ver la salida de cdparanoia -sQe)
let "CANT -= 13"
case 'dialog --backtitle "Ripper echo x fish" --stdout --menu "Hay $CANT canciones en el disco, extrae'
S)
INI=1
```

```

    FIN=$CANT
    ;;
N)
##echo Primera pista a extraer:
INI='dialog --backtitle "Ripper echo x fish" --stdout --inputbox "Primera pista a extraer" 0 0'
#echo Ultima pista a extraer:
FIN='dialog --backtitle "Ripper echo x fish" --stdout --inputbox "ultima pista a extraer" 0 0 '
;*)
esac

#Ejemplo de for y uso de arrays:
#Nombrar las canciones...
for ji in `seq $INI $FIN`
do
    CANCION[$ji]='dialog --stdout --backtitle "Nombre de las canciones" --inputbox "Nombre de la cancion '
done
f=0
h=0
j=1

# Rapiar y encodiar canciones
#echo 0 | dialog --backtitle "Ripper echo x fish" --gauge "Extrayendo cancion $INI..." 7 60 0
for i in `seq $INI $FIN`
do
    h='expr $f \* 100'
    j='expr $FIN - $INI + 1'
#solo una barra de progreso:
    expr $h / $j | dialog --backtitle "Ripper echo x fish" --gauge "Extrayendo cancion $i..." 7 60

#ejemplo de comandos interconectados a traves de tuberias
# con el - le indico a cdparanoia que escriba en la salida estandar
# con el menos al final de la linea le digo a oggenc que su entrada sea
# la salida estandar, el dd ibs=5M lo puse así porque así estaba en otro
# frontend al que le robé la idea :-)
#Otra opcion para hacer esto sería:
# cdparanoia -q -Smax $i $i.wav ##Se guarda la pista en un archivo.wav
# oggenc -q -q3 -t "${CANCION[i]}" -a "$BANDA" -l "$DISCO" -o pista-$i.ogg $i.wav
# rm $i.wav
#
#pero usando eso, se desperdicia mucho espacio en el disco (10.5 mb+ o - por
#cada minuto de audio), el mío hace extraccion y codificación "al vuelo"
    cdparanoia -q -Smax $i - | dd ibs=5M | oggenc -Q -q 3 -t "${CANCION[i]}" -a "$BANDA" -l "$DISCO" -
    f='expr $f + 1'
    echo $f
done
echo $f
h='expr $f \* 100'
expr $h / $j | dialog --backtitle "Ripper echo x fish" --gauge "Extrayendo cancion $i..." 7 60
rm tmp.tmp
#fin

```

Para saber que hacen los comandos usados (cdparanoia, oggenc, expr, dialog, etc), ver las correspondientes páginas de manual, [man comando](#)

Tarea, cambiar el programa para que use gdialg. Una pequeña pista, gdialg escribe todo en la salida de error, así que hay que usar la redirección...