Bash scripting de supervivencia

Jon Latorre Martínez
Metabolik Bio Hacklab
mailto:moebius@etxea.net

versión 0.1

28 de Julio de 2004

Este documento es la ayuda para un taller desarrollado en el hacklab Metabolik. Intenta ser un documento para una primera y rápida aproximación a la programación de guiones de comandos.

Introducción

Esto pretender ser un taller rápido y sucio de bash scripting. Así que no esperes el manual definitivo de programación en guiones de bash, o un documento académico. Su tono será desenfadado, sus contenidos puede que no se ajuste a la realidad (ya sabes, cualquier parecido con la realidad es pura casualidad, y cualquier script que funcione una suerte xDDD).

¿Porque bash y no otra shell?

Pues porque es la que conozco :) Bueno, de verdad este documento habla sobre bash ya que es el interprete de comandos mas extendido en Linux. Muchos de los ejemplos serán validos para otros interpretes, pero otros serán específicos de bash.

También conviene recordar que hay mas lenguajes interpretados para los que se pueden hacer scripts (perl, python, php). Perl es algo confuso para los novatos (a mi me sigue pareciendo que son jeroglíficos mas que guiones :), pero en pocas lineas se pueden hacer viguerías (sobre todo con expresiones regulares). Python esta muy de moda y es bastante potente (se puede hacer de todo con el, lo mismo una aplicación p2p que un instalador), aunque consume bastantes recursos. Y php es una opción interesante, ya que tiene muchas funciones (bases de datos, "parseo" de textos, etc) y es bastante sencillo (su sintaxis se parece a la de C por ejemplo), además podemos utilizarlo desde consola o en un servidor web.

Pero realmente, cualquier administrador de maquinas Unix, usuario medio e incluso usuario novato le sacara mucho partido a la programación de guiones de shell. Tareas repetitivas y tediosas se pueden automatizar con unas pocas lineas.

¿Conocimientos previos?

Pues abrir una terminal bash :) Bueno, en serio, saber moverse mínimamente por un sistema Linux y algunos conceptos básicos de programación (que es una variable, etc etc).

Retroalimentación

Si ves que se me ha ido la pinza mucho, que lo que digo esta mal o cualquier error grave, no dudes en mandarme un correo o en editar tu mismo este documento (si lo estas leyendo desde un wiki).

Por ultimo recordar que este documento esta licenciado bajo la licencia Creative Commons, que permite redistribuirlo y modificarlo con la única limitación de nombrar al autor (osea yo, moebius, aka Jon Latorre) y usar una licencia del mismo tipo.

Suerte y que los rm -rf * te sean propicios :)

Changelog

28-07-2004: Primera versión online

Precalentamiento. Mama, quiero ser script!

Un guión o script no es mas que un fichero de texto plano que contiene una lista de comandos de cierto lenguaje. Estos lenguajes se llaman interpretados, ya que en vez de compilar un fichero fuente hasta obtener uno ejecutable son interpretados directamente de la fuente. Ya hemos dicho que hay muchos lenguajes interpretados (perl, python, php, etc) pero nostros nos vamos a centrar en los guiones de comandos de consola.

Lo primero que tendremos que saber hacer es como llamar a estos guiones. Una primera manera sería llamando al intérprete de comandos (bash en este caso) pasandole como parámetro nuestro guión:

```
bash miscript.sh
```

o

```
sh miscript.sh
```

Otra manera válida es activando el bit de ejecución en los permisos del fichero:

```
chmod +x miscript.sh
./miscrit.sh
```

para estoy hay que tener en cuenta 2 cosas

- Debemos indicar que interprete vamos a usar para ejecutar el guion. Normalmente esto se hace añadiendo al guion como primera linea una que comience con #!interprete. Si vamos a hacer guiones de shell esto mismo se puede lograr asegurandose que la extensión es .sh.
- Para ejecutarlo deberemos indicar la ruta al fichero, o incluir el directorio del guion en nuestro path (directorios donde se buscan ejecutables), o copiarlo a alguno de los directorios de nuestro path (a mi personalmente me gusta /usr/local/bin).

Hola mundo!

Empezaremos con el típico hola mundo. Pero vamos a aprovechar y de paso ver algunas de las ventajas de la programación en shell.

```
#!/bin/bash
FECHA=$(date)
echo "Hola mundo"
echo "Ahora mismo son"; echo $FECHA
echo "Es hora   de despedirse.!"
```

Lo primero es indicar cual va a ser el interprete de nuestro guión. Hemos indicado explícitamente que queremos usar bash, pero podíamos haber indicado /bin/sh y nuestro script sería mas genérico (siempre que no usemos funciones exclusivas de bash).

Lo siguiente que hemos hecho es almacenar en una variable (FLECHA, las mayúsculas no son obligatorias, pero hacen mas legible el código) el resultado de ejecutar un comando (date, que nos dará la fecha actual). En esta caso FLECHA es el nombre de la variable, y $FLECHA es su contenido. Para asignar un contenido a una variable usamos = (ojo con los espacios). Las variables en shell no tienen tipos. Normalmente funcionan siempre como cadenas de textos (se pueden concatenar etc).

Luego hemos pasado a sacar distinta información por pantalla. Como se ve, los comandos se pueden separar con un salto de linea, o con el uso de punto y coma. Asimismo, si un comando es demasiado largo y queremos dividirlo en varias lineas, podemos usar la contra barra (\) para hacerlo (realmente lo que hacemos es "escapar", evitar que se interprete, el retorno de carro que nos da una linea nueva.

También hemos hecho uso del valor contenido en una variable, al llamarla con el símbolo de dolar delante.

Comunicándonos con el exterior

Ya hemos visto como sacar información al exterior para comunicarnos con el usuario (echo). Pero también querremos que el usuario introduzca algún dato. Para ello tenemos el comando read, su sintaxis es:

READ

```
read VARIABLE
```

Con esto lo que escriba el usuario seria almacenado en la variable VARIABLE. Un ejemplo:

```
#!/bin/bash
echo -n "Introduce: \t"
read ENTRADA
echo "Has tecleado: $ENTRADA"
```

SELECT

Select esta a medio camino entre una orden lectura de entrada y una de control de flujo. Es por ello que la veremos mas a fondo en la sección de abajo. Por ahora solo deciros que select se usa para que el usuario elija una opción entra una lista prefijada.

redirecciones y tuberías

Todo comando en Unix tiene 3 tuberías: Entrada estándar, salida estándar y salida de error estándar. Estas entradas podemos redireccionarlas a ficheros (con los símbolos < y >) o a otros comandos (con una "tubería", el símbolo |). Esto nos permite ir conectado comando cono piezas de un lego y lograr resultados sorprendentes.

Podemos imaginarnos un comando Unix como una pieza de fontanería con 3 bocas. Si por ejemplo queremos unir la salida de un comando a la entrada de otro, podemos hacer fácilmente con una tubería (pipe) (|)

```
ls | sort
```

Esto nos haría un listado del directorio actual y lo ordenada por orden alfabético. Pero también podemos enviar la salida de un comando a un fichero:

```
ls > fichero.txt
```

Y como no, también podemos hacer que un comando tome su entrada de un fichero:

```
sort < fichero.txt
```

Si queremos que la salida de un comando se añada a un fichero, en vez de sustituir su contenido, podemos usar >>.

Hemos dicho que teníamos 3 tuberías. Como redireccionamos pues la salida de errores estándar? Pues todas las tuberías están numeradas(o entrada, 1 salida, 2 errores) así que podemos hacer uso de su numero al redireccionar:

```
comando 2> errores.txt
```

también podemos redireccionar una salida sobre otra...

```
comando 2>&1
```

...y mandarlo todo a un fichero:

```
comando >> salida.txt 2>&1
```

```
=--=
 ||
```

Control del flujo

Con lo explicado hasta hora podríamos empezar a hacer nuestros pinitos, agrupando comandos en listas que se ejecutan siempre secuencialmente. Pero llegara un momento en el que querremos controlar el flujo del programa (si cumple una condición haz una cosa, repite esto tantas veces, etc).

IF

es la estructura de control mas básica: Si se cumple la condición haz esto, sin haz esto otro. Esto en código sería:

```
if condición
then
 lista de comandos
else
 lista de comandos2
fi
```

La condición puede ser de distintos tipos como veremos mas abajo. La orden else y lista de comandos2 son opcionales, si no las necesitas no hace falta usarlas.

Un ejemplo tonto:

```
PREGUNTAR="SI"
if [ $PREGUNTAR == "SI" ]
then
 echo "Me han pedido que te pregunte cual es tu nombre?"
else
 echo "No me apetece saber como te llamas"
fi
```

WHILE

Con while repetiremos una lista de comandos siempre que la condición sea cierta. Hay que recordar que con while se comprueba la condición antes de ejecutar los comandos, con lo cual puede que no se ejecuten ni una sola vez. También disponemos de until, cuyo funciona miento es similar a while, solo que en esta ocasión la condición sera negada (repetir mientras condición sea falsa, en vez del repetir mientras condición sea cierta que sería un while).

Su sintaxis es:

```
while [ condición ]
do
 comando1
```

```
 comando2
 comando3
 ....
done
```

Ejemplo tonto:

```
SALIR=0
while [ ! $SALIR ]
do

 if [ ]
 then
  SALIR=1
 fi

done
```

Que también se podría escribir usando until:

```
SALIR=0
until [ $SALIR ]
do

 if [ ]
 then
  SALIR=1
 fi

done
```

FOR

El funcionamiento de for en shell es distinto del funcionamiento tradicional de for en lenguajes como C. La sintaxis seria la siguiente:

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

Estas lineas lo que harán será ir asignando a variable cada uno de los elementos de la lista y ejecutar comandos tantas veces como elementos tenga la lista. Por ejemplo un listado de ficheros en un directorio podría ser...

```
for fichero in $(ls .)
do
 echo "Este es el fichero $fichero"
done
```

Si queremos un for mas "tradicional" podemos hacer uso del comando seq, que nos generar una lista:

```
for I in $(seq 1 10)
do
 echo "Estamos en la posición $I de 10"
done
```

Otra manera de usar for es:

```
for (( expr1 ; expr2 ; expr3 ))
do
 lista comandos
done
```

funcionamiento raro: First, the arithmetic expression expr1 is evaluated according to the rules described below under ARITHMETIC EVALUATION. The arithmetic expression expr2 is then evaluated repeatedly until it evaluates to zero. Each time expr2 evaluates to a non-zero value, list is executed and the arithmetic expression expr3 is evaluated. If any expression is omitted, it behaves as if it evaluates to 1. The return value is the exit status of the last command in list that is executed, or false if any of the expressions is invalid.

CASE

Con case podemos comprar una variable con varios valores distintos. Su sintaxis es la siguiente:

```
case $VARIABLE in
 patron1 )
  comandos1
;;
 patron2 )
  comandos2
;;
 patron3 )
  comandos3
```

```
;;
esac
```

En el caso de arriba se comprobaría $VARIABLE con los distintos patrones y en caso de coincidencia se ejecutaría los comandos consecuentes. Un ejemplo de un posible uso de case es el procesado de los parámetros pasados a un guión:

```
#!/bin/bash
case $1 in
 -h )
echo "Aquí va la ayuda"
;;
 -e )
echo "Opción -e "
;;
esac
```

SELECT

ya hemos dicho que esta orden es un poco curiosa. Veamos su sintaxis y luego la analizaremos:

```
select VARIABLE in uno dos tres cuatro
do
 comandos
done
```

Esta orden select nos mostrara por la salida de errores estándar una lista numera que contendrá todas las palabras después de in (en este caso uno dos tres cuatro). Luego se quedara esperando nuestra elección. Si tecleamos uno de los numero correspondientes a alguna de las palabras en la lista, se le asignara dicho valor a la variable VARIABLE y se ejecutaran los comandos. Si tecleamos un numero fuera del rango o cualquier otra cosa, la variable se quedara sin contenido. Lo tecleado se almacenara en la variable $REPLY. Los comandos se ejecutaran cada vez que elegíamos una opción hasta que ordenemos un comando break.

Si no indicamos la lista de palabras, se nos mostrara una lista con los parámetros pasados al comando.

Select nos puede ser útil por ejemplo para crear el menú principal de un programa junto con la orden case. Por ejemplo[o]:

```
select ACCION in Empezar Repetir Acabar
do
 case $ACCION in

 "Empezar")
  echo "El usuario quiere empezar"
  ;;
 "Repetir")
  echo "El usuario quiere repetir"
  ;;
 "Acabar")
  echo "El usuario quiere salir"
  break
  ;;
 *)
  echo "No se que quiere el usuario"
  ;;
 esac
done
```

Condicionales, evaluación aritmética,....

Condicionales

Podemos hacer una evaluación condicional para una sentencia IF o WHILE haciendo uso de [] o test. Por ejemplo podemos hacer lo siguiente y el resultado será el mismo:

```
cadena1="un texto"
cadena2="otro texto"
if test "$cadena1" = "$cadena1"
then
 echo "cadenas iguales"
else
 echo "cadenas distintas"
fi
if [ "$cadena1" == "$cadena1" ]
then
 echo "cadenas iguales"
else
 echo "cadenas distintas"
fi
```

También podemos usar ((condición)) para provocar una evaluación aritmética de la condición. Esto también lo lograremos con let.

las condiciones pueden negarse (! condición), sumarse (vamos, hacer un or, cierto si se cumple alguna de las 2)(condicion1 || condicion2) o multiplicarse (hacer un and, un "y", solo será cierto si ambas condiciones son ciertas)(condicon1 && condicion2)

cadenas

```
cadena1 = cadena2 Verdadero si las 2 cadenas son iguales
cadena1 != cadena2 Verdadero si las 2 cadenas son distintas
-z cadena  Verdadero si la cadena esta vacía (su longitud es cero)
cadena   Verdadero si la cadena no esta vacía
                      (su longitud es mayor de cero)
-n cadena  Verdadero si la longitud de la cadena es mayor que cero
cadena1 == cadena2 Verdadero si las 2 cadenas son iguales (solo en bash)
```

ficheros

```
-e fichero  Verdadero si el fichero existe.
-d fichero  Verdadero si fichero existe y es un directorio.
-f fichero  Verdadero si fichero existe y se un fichero regular.
-L fichero  Verdadero si fichero existe y se un enlace simbólico.
-r fichero  Verdadero si fichero existe y se puede leer.
-w fichero  Verdadero si fichero existe y se puede escribir.
-x fichero  Verdadero si fichero existe y se ejecutable.
fichero1 -nt fichero2 Verdadero si fichero1 mas actual
                      (según la fecha de modificación) que fichero2.
fichero1 -ot fichero2 Verdadero si fichero1 mas antiguo que fichero2.
fichero1 -ef fichero2 Verdadero si fichero1 y fichero2 tiene el mismo
                      numero de device e inodo.
```

Estas son las mas comunes, hay unas cuantas mas. man bash :)

Numéricas

Evaluación aritmética: let y expr

Si queremos hacer cálculos aritméticos (que una expresión sea evaluada aritméticamente) podemos hacer uso del comando expr:

```
echo "Vamos a hacer unos cálculos:"
echo "2 mas 2 : $(expr 2 + 2 )
DOS=2
echo "4 entre 2 : $(expr 4 / $DOS )"
```

Algunos caracteres deben ser escapado para evitar que bash los interprete (si queremos usar < y > tendremos que usar \< y \>). Mas información en man expr :)

Pero si estamos usando bash podemos hacer uso de la orden interna let :

```
id++ id--       Se procesa el valor y después se aumenta o decrementa
++id --id       Aumenta o decrementar el valor de la variable y se procesa
- +            unary minus and plus
! ~            Negación lógica y de bits
**             Exponencial
* / %          Multiplicar, dividir, resto
+ -            Sumar, restar
<< >>          Rotar bit a izquierda y derecha
<= >= < >      Comparaciones
== !=          Igualdad, desigualdad
&              bitwise AND
^              bitwise exclusive OR
|              bitwise OR
&&             logical AND
||             logical OR
expr1?expr2:expr3  Evaluación condicional: Si expr1 entonces expr2, sino expr3
= *= /= %= += -= <<= >>= &= ^= |=   asignaciones
```

Si necesitamos calculos matematicos mas avanzados, o trabajar con distintas bases podemos usar la potente herramienta bc.

```
 AÑADIR DOCU DE BC ALGUN DIA :P
```

ejemplos:

```
CONT=0
for I in $(seq 1 100)
do
 echo "Estamos en el bucle numero $CONT"
 let CONT++
done
```

Algunas variables importantes $algo ($0,$1,&?,$#,....)

A la hora de ejecutar un guión de interprete de comandos tenemos unas cuantas variables que nos puede ayudar. Como todas las variables en shell empiezan por el símbolo del dolar($). Veamos algunas:

$numero

Las variables nombradas como $ mas un números ($0,$1,$2,...) representan los distintos parámetros recibidos. $0 representa el nombre del propio guión de comandos.

$#

Esta variable indica el numero de parámetros que ha recibido nuestro guión al ser llamado. Es útil por ejemplo para comprobar que el usuario ha introducido el numero de parámetros adecuado, o para saber cuantos parámetros ha introducido para ir procesándolos (esto se hace mas fácil con shift). Un ejemplo:

```
if [ $# -ne 3 ]; then
echo "Es necesario introducir 3 parámetros: $0 parametro1 parametro2 parametro3"
exit -1
fi
```

$?

Representa el estado de la salida del comando anterior. De esta manera podremos saber si el comando ejecutado anteriormente ha finalizado exitosamente o a ocurrido algún error.

[[Explicar como usar errores de salida y como generarlos (exit)]]

$*, $@

Estas dos variables nos devuelven todos los parámetros. La diferencia está en que $* nos los va a devolver agrupados como una sola separada y los parámetros separados por espacios (bueno, realmente por el primer carácter de la variable IFS). Mientras que $@ nos devuelve los parámetros pasados al guión como una lista de palabras. Esto es:

$* equivale a : "$1 $2 $3 ..." (recordar que no siempre es espacio, comprobar IFS) $@ equivale a : "$1" "$2 "$3" ...

Como procesar los parámetros pasados a un script

Vamos a verlo con un ejemplo practico:

```
#!/bin/bash
echo "Hemos recibido $# parámetros"
while (( $# ))
do
 case $1 in
  -h )
 echo "Aquí va la ayuda"
  ;;
  -e )
 echo "Opción -e "
 ;;
 esac

 shift
done
```

Explicación:

- Entramos en un while que se repetirá mientras el numero de parámetros sea distinto de cero
- Con un case comparamos cada parámetro con las distintas opciones
- Hacemos un shift. Esto lo que hará sera rotar una posición todos los parámetros (el segundo será el primero, el tercero el segundo, etc) y decrementa el numero de parámetros

$_

Anterior comando ejecutado.

IFS

Separador de elementos de una lista. Normalmente espacio, tabulador y salto de carro. Ajustándolo podemos hacer mas útiles nuestros FOR-s

```
IFS=":"
```

```
echo "Directorios en el PATH..."
for DIR in $PATH
do
   echo $DIR
done
```

Algunos ejemplos de IFS utiles para cambiar entre sperador por tabuladores, retornos, etc:

1. Whitespace == :Space:Tab:Line Feed:Carriage Return:

WSP_IFS=$'\x20'$'\x09'$'\x0A'$'\x0D'

1. No Whitespace == Line Feed:Carriage Return

No_WSP=$'\x0A'$'\x0D'

1. Field separator for dotted decimal ip addresses

ADR_IFS=${No_WSP}'.'

Comas, comillas y comilla invertida

En ciertas ocasiones queremos que un texto en concreto sea tratado de cierta manera, ya sea porque no queremos que interprete caracteres especiales, o por que queremos que se ejecute un comando y se reemplace por su salida.

Comillas simples

Within single quotes all characters are quoted -- including the backslash. The result is one word.

```
grep :${gid}: /etc/group | awk -F: '{print $1}'
```

Comillas dobles

Con comillas dobles las variables serán substituidas por sus valores. Pero no se tendrá expansión de comodines de ficheros (como *.jpg, etc)

Within double quotes you have variable subsitution (ie. the dollar sign is interpreted) but no file name generation (ie. * and ? are quoted). The result is one word.

```
if [ ! "${parent}" ]; then
 parent=${people}/${group}/${user}
fi
```

Back Quotes

(COMO SE LLAMA) Con comilla "" lo que lograremos es que el comando se ejecute y sea substituido por su salida:

```
FECHA=`date`
echo "LA fecha es: $FECHA"
```

Programas auxiliares: cat, cut, grep, tr,...

cat

cut

grep

awk

Muy potente pero igual demasiado para este docu?

find, ese desconocido

date

Ideal para backups.

tar

wc

tr

Este programa lo que hace es substituir una cadena de texto por otra. Pero puede hacer mucho mas. Podemos eliminar cierta cadena, o reducir a una solo aparición esa cadena. sintaxis:

```
tr cadena1 cadena2 Cambia cadena1 por cadena2
tr -s cadena1  Reduce a una sola ocurrencia cadena1
tr -d cadena1  Elimina cadena1
```

Ejemplos:

```
TEXTO="_____hola_____mundo_____"
echo $TEXTO
Salida > _____hola_____mundo_____
echo $TEXTO | tr -s "_"
Salida > _hola_mundo_
echo $TEXTO | tr -s "_" | tr "_" " "
Salida >  hola mundo
echo $TEXTO | tr -d "_"
Salida > holamundo
```

Funciones

Se definen de manera muy sencilla

```
funcion() {
código
código
código
}
```

Dentro de la función podemos usar $0,$1...etc para acceder a los parámetros pasados a la función (no al guión). SI queremos indicar un valor de retorno deberemos usar return valor en vez de exit.

La funcion tiene que ser definida siempre antes de usarla.

Llamando a otros guiones

(comandos)

Ya lo hemos estado usando en los ejemplos anteriores. Si llamamos a una serie de comandos o a un script de la manera $() será reemplazo por el valor de retorno de dichos comandos o guión.

```
echo "En este directorio tenemos : $(ls -l)
```

sh command

Si llamamos a un guión usando sh (p.j.: sh miguion.sh) este guión se ejecutara en una shell separada.

. command

Pero si ejecutamos un guión de shell llamando con un punto (.) delante este se ejecutara dentro de la misma shell en la que se esta ejecutado el guión principal. Esto es, las variables que declaremos en alguno de los 2 guiones serán validas para el otro, etc.

Se suele utilizar a modo de leer ficheros de configuración.

```
#Vamos a leer la configuración del usuario
. ~/usuario.conf
```

Interfaces

A menudo deseamos que nuestro guión se comunique con un humano. Si comunicarnos a través de la consola no nos vale podemos recurrir a varios guis listos para usar desde shell script.

dialog y derivados

Dialog se basa en ncurses para poder representar diálogos de manera muy sencilla. Con el tiempo han salido derivados de dialog que son compatibles a nivel sintaxis. Por lo tanto saber manejar dialog es muy interesante.

Dialog tiene predefinidos varios tipos de ventanas:

```
calendar, checklist, fselect, gauge,
infobox,  inputbox,  input- menu,
```

```
menu,  msgbox  (message),  password,
radiolist, tailbox, tailboxbg, textbox,
timebox, and yesno (yes/no).
```

Elegiremos en cada momento la que mejor se adecue a nuestras necesidades

Luego tenemos una serie de parámetros comunes a todas las ventanas. Titulo, texto y tamaño son las mas comunes. Luego, en funcion de que tipo de ventana hayamos elegido tendremos unas opciones especificas.

A la hora de recuperar los datos introducidos por el usuario en el dialogo tendremos que usar la slaida estandar o la salida estandar de errores, depende del tipo de dialogo. Si queremos saber si el usuario a pulsado cancelar tendremos que fijarnos en el estado de salida de dialog (con $? por ejemplo).

Algunos ejemplos:

Xdialog

Como dialog pero para las X. Su sintaxis es compatible, así que podremos usar los mismos diálogos en ncurses o X, muy útil si no sabemos si el guión sera ejecutado en una maquina con X o no.

Kdialog

Mas de lo mismo pera esta vez usando qt y kde. Nos da un aspecto inmejorable si usamos KDE comoe scritorio.

gdialog

Zenity

Basado en gtk. Su sintaxis es parecida a la de Dialog, pero hay cambios. Nos ofrece un aspecto mucho mas refinado al usar ls librerías gtk2.

Debugeando

- Usar echos en todos los sitios posibles
    - Al entrar en bucles, funciones, etc
- Ejecutar bash -n script para comprobar errores de sintaxis
- Use the command set -v to get a verbose dump of each line the shell reads. Use set +v to turn off verbose mode.
- Use the command set -x to see what each command expands to. Again, set +x turns this mode off.

Optimizaciones

A la hora de programar guiones de shell podemos hacer uso de ciertos "trucos" para optimizarlos y reducir el numero de lineas. A continuación añado unos ejemplos sacados del documento del e-ghost de Pablo y Eduardo.

Trucos y curiosidades

Ejecutar un shell script remoto

Si alguna vez necesitas ejecutan un guion de comandos que este en una maquina remota (accesible por web) puedes usar lynx para ello.

```
lynx -dump URL_AL_SCRIPT | bash
```

Por ejemplo:

```
lynx -dump http://www.freeos.com/guides/lsst/scripts/for6 | bash
```

Comunicarse con otro guiones a traves de una red

Usando nc

Ejemplos

Para no aburrirnos con ejemplos vamos a hacer que sea mas divertido intentando resolver en vivo y en directo algunos problemas típicos de scripting.

Renombrar ficheros

Enunciado

Solución

Preguntado contraseñas

Enunciado

Solución

Backups

Este problema se planteo en la competición de scripting de la Euskal Encounter XII. Es el típico problema de creación de backups.

Enunciado

Script de backup (¡oh no, otro no!)

La idea es que en una misma máquina concurren varios usuarios. Cada usuario tendrá en un fichero backup.txt los archivos y directorios que quiere incluir o excluir en el backup, por ejemplo:

```
-----------------------
I/home/luis/documentos/
E/home/luis/documentos/personal/
I/home/luis/importante.txt
-----------------------
```

El script a desarrollar debe recorrer los ficheros de configuración de todos los usuarios y hacer un .tar.gz según los archivos que indica dicho fichero backup.txt.

Una vez juntado todo en el tgz, el script debe dar la posibilidad de grabarlo en un directorio o mandarlo por correo.

Solución

Esta es la solución que se me ocurre a mi (puede que este mal :)

```
#!/bin/bash
for I in $(ls ./home/)
do
# echo "El usuaro es: $I"
# cd home/$I
 meter=""
 excluir=""
 echo "Reset? $meter $excluir"
 for linea in $(cat home/$I/backup.txt)
 do
  accion=`echo $linea | cut -f 1 -d "/"`
  fichero=`echo $linea | tr -d "I" | tr -d "E" `
  case $accion in
  'I')
    meter=$meter" $fichero"
    ;;
  'E')
    excluir=$excluir" --exclude $fichero"
    ;;
   esac
 done
 echo "tar cvzf /tmp/backup-$I.tar.gz $meter $excluir"
 Xdialog --title "Que desea hacer con el backup?" \\
  --radiolist "Eliga que desa hacer con su backup \\
  /tmp/backup-$I.tar.gz" 0 0 0 \\
  correo "Enviar por correo" off  \\
  "copiar" "copiar a..." off > /tmp/user.cfg 2>&1
  accion_user=$(cat /tmp/user.cfg)
 case $accion_user in
  'correo')
    Xdialog --title "e-mail destino"  \\
    --inputbox "Eliga el correo electronico \\
    de destino" 0 0 > /tmp/destino.log 2>&1
    destino=$(cat /tmp/destino.log)
    echo "Se enviara $destino"
    echo "mv /tmp/backup-$I.tar.gz \\
    /tmp/backup-$I-$(date  +%Y-%m-%d).tar.gz"
    cat > /tmp/mail.txt <<EOF
~*
```

```
0
/tmp/backup-$I-$(date  +%Y-%m-%d).tar.gz
~.
EOF
   mailto -s "Backup del usuario $I \\
   en el dia $(date  +%Y-%m-%d)" \\
   $destino < mail.txt
  ;;
  'copiar')
    Xdialog --fselect "/tmp/backup-$I-$(date  +%Y-%m-%d).tar.gz" \\
    0 0 > /tmp/destino.log 2>&1
    destino=$(cat /tmp/destino.log)
    echo "Se movera a $destino"
    echo "mv /tmp/backup-$I.tar.gz $destino"
  ;;
 esac
done
```

URLs

Shell Scripting: programación con comandos de shell

Escrito por un miembro del grupo de software libre de la universidad de Deusto (e-ghost) tras sus cursillos de verano.

```
Álvaro Uría (Fermat - fermat00 AT euskalnet DOT net)
http://www.e-ghost.Deusto.es/docs/shellScriptin.html
```

Programación en Bash Shell

Cursillo impartido en la Universidad de Deusto por el grupo de software libre de la misma (e-ghost)

```
Pablo Garaizar Sagarminaga
Eduardo González de la Herrán
http://www.e-ghost.Deusto.es/cursillosjulio/ficheros/BashShell/bash.sxi
```

Taller de bash

Taller impartido por kleenux (Asociación de Usuarios de Software Libre de Elche: http://www.kleenux.org)

```
Autor: Juan J. Martínez <jjm_ATTTTTT_usebox.net>,
con la colaboración de Paco Brufal <pbrufal_ATTTTT_mutoid.org>
http://blackshell.usebox.net/pub/shell/taller_sh/
```

Bases de la programación para el Bash

Muy completo (incluye hasta uso de sockets en bash)

```
Por Xento Figal http://xinfo.sourceforge.net
http://xinfo.sourceforge.net/documentos/bash-scripting/bash-script-2.0.html
```

How to write a shell script

```
http://vertigo.hsrl.rutgers.edu/ug/shell_help.html
```

que a su vez esta basado en:

```
An Introduction to Shell Programing by:
Reg Quinton
Computing and Communications Services
The University of Western Ontario
London, Ontario N6A 5B7
Canada
```

Shell Scripts and Awk

```
http://www-h.eng.cam.ac.uk/help/tpl/unix/scripts/scripts.html
```

Shell scripts in 20 pages, Russell Quong

```
http://quong.best.vwh.net/shellin20/#LtohTOCentry-26
```

man bash :)

Como no, no podía faltar un típico man. La man de bash es bastante extensa pero para buscar información concreta es bastante útil. También es interesante revisar las man de lso comandos auxiliares que usemos (cut, tr, grep, etc).

- http://www.freeos.com/guides/lsst/ch08.html

-
-
-
-

Licencia