Bash scripting

Pablo Garaizar Sagarminaga

e-ghost, cursos de verano, 2011

¿Qué vamos a ver?

- El intérprete de comandos:
- Comandos básicos.
- Tuberías.
- Comandos de filtrado.
- Búsquedas y expresiones regulares.
- Otros comandos.
- Scripting.
- Optimizaciones.

Intérprete de comandos

- Shells
  - Existen muchas shells: sh, csh, ksh, bash, etc.
  - Entorno de trabajo:
    - Case sensitive: sensible a mayúsculas (ECHO != echo).
    - Sintáxis: comando arg1 arg2... argn
    - Si un programa no está en el PATH: ./programa
    - Prompt:
      - $: usuario normal
      - #: usuario administrador/a (root)

Comandos básicos

- ls: listar.
- man: manual.
- pwd: directorio actual.
- cd: cambia de directorio.
- echo: escribe texto.
- cat: muestra fichero.
- more: paginador.
- file: muestra tipo de fichero.
- touch: crea fichero.
- rm: borra fichero.
- mkdir: crea directorio.
- rmdir: borra directorio.
- cp: copia ficheros.
- mv: mueve ficheros.
- ln: crea enlaces.
- date: muestra la fecha.

Tuberías

- Un proceso en un sistema UNIX-like tiene inicialmente abiertos 3 “desagües”:
  - 0: STDIN o entrada estándar
  - 1: STDOUT o salida estándar
  - 2: STDERR o salida de error
  - Imaginémonos una refinería:
    - Metes crudo por 0 (STDIN), consigues gasolina por 1 (STDOUT) y bastantes residuos por el “desagüe” 2 (STDERR).

Tuberías

- Redirigiendo la salida de un comando:
  - > : redirigir STDOUT a un fichero
    - **`ls > listado.txt`**
  - >> : redirigir STDOUT al final de un fichero (añadir)
    - **`ls >> listados.txt`**
  - 2> : redirigir STDERR a un fichero
    - **`ls 2> errores.txt`**
  - 2>> : redirigir STDERR al final de un fichero
    - **`ls 2>> errores.txt`**

Tuberías

- Redirigiendo la entrada de un comando:
  - < : redirigir el contenido de un fichero a STDIN
    - **`tr a A < fichero.txt`**
- Tuberías:
  - | : es posible recoger la salida de un desagüe y conducirlo a la entrada de otro comando.
    - **`cat fichero.txt | tr a A`**

Comandos de filtrado

- sort: ordena líneas de ficheros de texto.
- tr: reemplaza un carácter por otro.
- head: muestra las N primeras líneas de un fichero.
- tail: muestra las N últimas líneas de un fichero.
- wc: cuenta líneas, palabras o caracteres.
- cut: permite “cortar” columnas de un fichero en función de un carácter.

Búsquedas y regexps

- Comandos de búsqueda:
  - find: busca ficheros.
  - grep: busca patrones en ficheros.
  - sed: modifica patrones.
  - awk: busca y modifica campos en ficheros.
- Todos usan patrones, expresiones regulares.

Búsquedas y regexps

- grep: busca una cadena o patrón en ficheros y muestra las líneas que la contienen.

```
grep [opciones] patron [fich1 fich2...]
```

- Metacaracteres válidos:
  - . (un carácter cualquiera)
  - * (cero o más ocurrencias del carácter anterior)
  - ^ (principio de línea)
  - $ (fin de línea)
  - [a-f] (cualquier carácter del rango)
  - \< \> (principio y fin de palabra)

Búsquedas y regexps

- Opciones típicas:
  - -c escribe sólo el número de ocurrencias encontradas.
  - -i ignora mayúsculas y minúsculas.
  - -l escribe sólo los nombres de fichero donde ha encontrado el patrón.
  - -s opera en silencio, sólo saca mensajes de error.
  - -v escribe líneas que NO contienen el patrón
  - -w trata el patrón como una palabra entera.
- Uso típico: filtrar la salida de un comando

```
$ comando | grep ...
```

Búsquedas y regexps

- Patrones
  - . (un carácter cualquiera)
  - * (cero o más ocurrencias del carácter anterior)
  - ^ (principio de línea)
  - $ (fin de línea)
  - [a-f] (cualquier carácter del rango)
  - [-AD] (caracteres '-', 'A', 'D')
  - [^56] (todos los caracteres excepto '5' y '6')
  - [5^6] (caracteres '5', '^', '6')
  - [[:alnum:]] (todos los caracteres alfanuméricos)

Búsquedas y regexps

- Quoted Braces (llaves)
  - Se utilizan para buscar un número determinado de repeticiones de la expresión:
    - expresion_regular \{min, max\}
      - **grep '[345]\{2,4\}' fichero**
    - expresion_regular \{exact\}
      - **grep 'r\{2\}' fichero** (busca dos 'r' seguidas)
    - expresion_regular \{min,\}
      - **grep 'er\{2,\}' fichero**

      (busca lineas con al menos dos caracteres 'r' precedidos por una 'e').

Búsquedas y regexps

- Quoted Parentheses (paréntesis)
  - Almacena los caracteres que concuerdan con la expresion regular en un registro.
  - Se pueden utilizar hasta 9 registros en una expresión regular
  - Para referenciar los registros utilizamos \1 a \9
    - Ejemplo: listamos líneas con dos caracteres idénticos seguidos:
    - grep '\(.\)\1' fichero

Búsquedas y regexps

- ¿Qué es awk?
  - Awk es un lenguaje de programación utilizado para manipular texto.
  - Los datos se manipulan como palabras (**campos**) dentro de una línea (**registro**).
  - Un comando awk consiste en un **patrón** y una **acción** que comprende una o más sentencias.
    - `awk 'patron { accion }' fichero ...`

Búsquedas y regexps

- ¿Qué es awk?
  - awk comprueba cada registro de los ficheros indicados en busca de coincidencias con el patrón; si se encuentra alguna se realiza la acción especificada.
  - awk puede actuar como filtro tras un pipe o obtener los datos de la entrada estándar si no se especifica ningún fichero.

Búsquedas y regexps

- Expresiones regulares
  - Las expresiones regulares quedan delimitadas con el carácter '/' (ej: /x/)
  - Además de todas las comentadas hasta ahora, awk puede utilizar:
    - /x+/ Una o más ocurrencias de x
    - /x?/ Una o ninguna ocurrencia de x
    - /x|y/ Tanto x como y
    - (string) Agrupa una cadena para usar con + o ?

Búsquedas y regexps

- Sentencia print
  - Es de las más utilizadas
    - **`awk 'patron { print }' f_ent > f_sal`**
      - este ejemplo sería lo mismo que:

      `$ grep patron f_ent > f_sal`
    - Referenciando campos:
      - $0: Registro completo
      - $1: Primer campo del registro
      - $2: Segundo campo del registro
      - ...

Búsquedas y regexps

- Más funcionalidades:
  - Multiples sentencias en una acción
    - Usando “;” o salto de línea:
  - Comentarios con “#”
  - Comparaciones de expresiones regulares o strings
    - ==, !=, <, <=, >, >=, ~, !~, ||, &&
    - $1 ~ /x/

```
$ awk '$1 ~ /^[T]/ || $3 ~ /^[46]/ { print } fichero
```

Búsquedas y regexps

- Más funcionalidades:
  - Variables de usuario
  - Operaciones aritméticas
  - Estructuras de control
    - if – else if – else
    - while
    - for

Búsquedas y regexps

- find: herramienta muy potente para realizar búsquedas de ficheros. Sintaxis:
  - **`find [path] opciones`**
  - Opciones típicas:
    - -name “nombre”
    - -type (d|f|l|s|...)
    - -maxdepth n
    - -mindepth n
    - -mtime n
    - -exec comando \; ({} indica el fichero)

Usuarios y grupos

- useradd: añade un usuario en el sistema.
- userdel: elimina un usuario del sistema.
- usermod: modifica las propiedades de un usuario.
- groupadd: añade un grupo de usuarios en el sistema.
- groupdel: elimina un grupo del sistema.
- groupmod: modifica las propiedades de un grupo.
- passwd: modifica la contraseña de un usuario.
- gpasswd: añade un usuario a un grupo.

Propietarios y permisos

- chmod: modifica el modo de acceso de un fichero.
- chown: modifica el propietario de un fichero.
- chgrp: modifica el grupo propietario de un fichero.
- who: muestra usuarios conectados al sistema.
- whoami: muestra el nombre de usuario del usuario actual.
- id: muestra las propiedades del usuario y grupo actuales.
- su: cambio de usuario actual.

Gestión de procesos

- top: muestra información de procesador, procesos y memoria.
- ps: muestra la lista de procesos que se están ejecutando.
- pstree: ps en forma de árbol.
- pgrep: ps + grep.
- pidof: muestra el PID del proceso que solicitemos.
- kill: envía una señal a un proceso (PID).
- killall: envía una señal a un proceso (nombre).

Gestión de procesos

- &: al lanzar un proceso lo lanza en segundo plano.
- bg: manda un proceso a ejecutarse en segundo plano.
- fg: lleva un proceso a ejecutarse en primer plano.
- ctrl+z: interrumpe un proceso y lo suspende.
- ctrl+c: interrumpe un proceso y lo para (señal KILL).
- jobs: muestra las tareas de la sesión actual en segundo plano.
- nohup: hace que un proceso cuelgue de init y redirige su salida a nohup.out.
- disown: hace un nohup de un proceso ya en ejecución.

Comandos de backup

- gzip: compresor estándar
- bzip2: compresor más pontente pero más lento
- tar: empaquetador de ficheros (muy útil)
- zcat: cat sobre archivos comprimidos
- zmore: more sobre archivos comprimidos
- zgrep: grep sobre archivos comprimidos
    - `zgrep cadena file.gz`
    - `gzip -c file.gz | grep cadena`

Shell Scripts

- Script = Guión.
- Tareas repetitivas se pueden agrupar en un guión y ejecutarse automáticamente (Batch Processing).
  - Es sencillo ejecutar 4 comandos para crear un buzón de correo.
  - No lo es tanto para crear 20.000 buzones.
  - Es sencillo hacer un bucle que se repita 20.000 veces ;-)

Shell Scripts

- Nuestro primer shell script:
  - Usamos un editor y creamos el fichero hola.sh
  - Contenido de hola.sh:

```
#!/bin/sh

echo hola
```

  - Con #! en la primera línea indicamos quién debería interpretar el resto de comandos (/bin/sh).
  - Posteriormente escribimos los comandos separados por saltos de línea.

Shell Scripts

- Variables:
  - Una variable tiene un **nombre** y un **valor**, y sirve para dotar de dinamismo a nuestros scripts:

```
FECHA="15/07/2004"

echo "Hoy es $FECHA"
```

    - FECHA es el nombre de la variable.
    - $FECHA es su valor.
  - Para asignar un valor, se utiliza "=". ¡¡¡SIN ESPACIOS!!!

Shell Scripts

- Variables de entorno:
  - Al arrancar una shell, ya hay muchas variables definidas, son las variables de entorno.
    - Podemos ver su valor con el comando “env”.
  - Ámbito de una variable:
    - Si se define una variable en una shell, sólo tiene valor en esa shell, a no ser que se exporte a los programas “hijo”.
    - **`export USUARIO="joaquin"`**
    - Si desde esa shell lanzamos un script u otro programa, la variable USUARIO contendrá “joaquin”.

Shell Scripts

- Variables: interactividad
  - Es posible leer del usuario el valor de una variable, dotando a nuestros scripts de interactividad.
  - cat hola.sh

```
#!/bin/sh

echo "Dime tu nombre:"

read NOMBRE

echo "Hola $NOMBRE, encantado de
 conocerte"
```

Shell Scripts

- Variables: argumentos
  - Es posible pasar los parámetros o argumentos que queramos y utilizarlos dentro del script.
    - cat nombre.sh

```
#!/bin/sh
echo "Nombre: $1"
echo "Primer Apellido: $2"
echo "Segundo Apellido: $3"
```

    - ./nombre.sh Juan López Martínez
    - ./nombre.sh "Maria Dolores" Pradera Sánchez

Shell Scripts

- Variables: argumentos
  - $1, $2, $3... ${10}, ${11}: argumentos
  - $0 es el propio script.
    - basename $0: nombre del script.
    - dirname $0: ruta al nombre del script.
  - shift: rota los argumentos hacia la izquierda
    - $1 ahora vale lo que valía $2, $2 lo que valía $3, etc.
    - $0 no cambia.

Shell Scripts

- Variables: argumentos especiales
  - $#: número de argumentos que nos han pasado.
  - $*: todos los argumentos. “$*” = “$1 $2 $3...”
  - $@: todos los argumentos. “$@” = “$1” “$2” “$3”...
  - $_: comando anteriormente ejecutado.
  - $$: PID del propio proceso shell.

Shell Scripts

- Variables: sustitución de comandos
  - Es posible almacenar en una variable el resultado de la ejecución de un comando.
  - Dos sintaxis:
    - Acentos graves: compatibilidad
      - `LISTADO=\`ls\``
    - Con $(): anidable
      - `LISTADO=$(ls)`
      - `LISTADO=$(ls $(cat directorios.txt))`

Shell Scripts

- expr: Permite realizar operaciones aritméticas.
  - Sintaxis: **`expr ARG1 OP ARG2`**

```
$ SUMA=`expr 7 + 5`
```
(¡ojo espacios!)

```
$ echo $SUMA
12
$ expr 7 \> 5
```
(¡ojo escapar operadores!)

```
$ expr \( 7 + 5 \) \* 2
```

  - Ejercicio: Obtener el mes anterior al actual con una sola sentencia expr ;-)

Shell Scripts

- Control del flujo de ejecución:
  - Condiciones: test ó [ ]
    - test “$NOMBRE” == “Juan” (==, !=, >, <, >=, <=)
    - test $DINERO -eq 1000 (-eq, -ne, -gt, -lt, -ge, -le)
    - test -f /etc/passwd (-f, -d, -L, -r, -w, -x)
    - Modifican el valor de $?
      - cero = verdadero
      - no cero = falso (¡¡AL REVÉS QUE EN C!!)

Shell Scripts

- Control del flujo de ejecución:
  - if: alternativa simple. Sintaxis:

```
if comando_if
then
  comandos_then
elif comando_elif
then
  comandos_elif
else
  comandos_else
fi
```

Shell Scripts

- Control del flujo de ejecución:
  - if. Ejemplo:

```
if test "$NOMBRE" == "Juan"
then
  echo "Hola Juanin, ¿qué tal?"
elif test "$NOMBRE" == "Pedro"
then
  echo "Pedreteee, ¡cuánto tiempo!"
else
  echo "No te conozco"
fi
```

Shell Scripts

- Control del flujo de ejecución:
  - case: cómodo para evitar alternativas anidadas. Sintaxis:

```
case $VARIABLE in
   "VALOR1") comandos_valor1
      ;;
   "VALOR2") comandos_valor2
      ;;
   *) comandos_default;
esac
```

Shell Scripts

- Control del flujo de ejecución:
  - case. Ejemplo:

```
case $NOMBRE in
  "Juan") echo "Hola Juanin, ¿qué tal?"
    ;;
  "Pedro") echo "Pedro, ¡cuánto tiempo!"
    ;;
  *) echo "no te conozco";
esac
```

Shell Scripts

- Control del flujo de ejecución:
  - while. Ejecución de 0 a N veces. Sintaxis:

```
while comando
do
   comandos
done
```

Shell Scripts

- Control del flujo de ejecución:
  - while. Ejemplo:

```
N=1
while [ $N -lt 100 ]
do
   echo "Repito, ya voy $N veces"
   N=$(expr $N + 1)
   sleep 1 # Esperamos 1 segundo
done
```

Shell Scripts

- Control del flujo de ejecución:
  - until. Ejecución de 0 a N veces. Idéntico a while con la condición negada. Sintaxis:

```
until comando
do
   comandos
done
```

Shell Scripts

- Control del flujo de ejecución:
  - until. Ejecución de 0 a N veces. Idéntico a while con la condición negada. Sintaxis:

```
N=1
until [ $N -ge 100 ]
do
   echo "Repito, ya voy $N veces"
   N=$(expr $N + 1)
done
```

Shell Scripts

- Control del flujo de ejecución:
  - for: ejecución repetitiva asignando a una variable de control valores de una lista. Sintaxis:

```
for VARIABLE in LISTA
do
   comandos
done
```

Shell Scripts

- Control del flujo de ejecución:
  - for. Ejemplo:

```
for N in “Sopa” “Arroz” “Pan de ajo”
do
  echo “Hoy comemos $N”
done
```

Shell Scripts

- Control del flujo de ejecución:
  - for: la LISTA define la separación de cada elemento por el valor de la variable IFS (que por defecto vale `" \t\n"`). Ejemplo:

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

Shell Scripts

- Control del flujo de ejecución:
  - for. Ejemplos numéricos:

```
for N in 1 2 3 4 5 6 7 8 9 10
do
  echo "N ahora vale $N"
done
for N in $(seq 10)
do
  echo "N ahora vale $N"
done
```

Shell Scripts

- Control del flujo de ejecución:
  - select: muestra las opciones especificadas en LISTA y asigna a VARIABLE la opción escogida. Sintaxis:

```
select VARIABLE in LISTA
do
   comandos
done
```

Shell Scripts

- Control del flujo de ejecución:
  - select: Ejemplo:

```
select OPCION in “Doner Kebab” “Pizza”
do
  case $OPCION in
    “Doner Kebab”) echo “Mmmm...”
      break;;
    “Pizza”) echo “Slurppp!”
      break;;
    *) echo “No sé qué es eso”
  esac
done
```

Shell Scripts

- function:
  - Podemos modularizar los scripts agrupando tareas en funciones.
  - Es necesario que una función esté definida ANTES de que sea llamada.
  - Dentro de una función, $1, $2, $3, etc. serán los parámetros pasados a la función, no al script en sí.

Shell Scripts

- function. Ejemplo:

```
#!/bin/sh
function suma
{
  echo $(expr $1 + $2)
}
suma 4 6
suma 3 234
```

Shell Scripts

- source, .
  - Con source o con “.” podemos incluir el código de otro script en el nuestro:

```
#!/bin/sh

source funciones.sh # ahí se define suma

suma 1 3

suma 12 12312
```

Optimizaciones

- **`$(ls)` → `*`**
  - **`for F in $(ls) == for F in *`**
- **`cat fichero | comando` → `comando < fichero`**
  - **`cat fichero | grep patron`**
  - **`grep patron < fichero`**
- **`echo $VAR | comando` → `comando <<< $VAR`**
  - **`echo $PATH | cut -d: -f1`**
  - **`cut -d: -f1 <<<$PATH`**
- : es como una NOP
  - : >> es lo mismo que touch. Ej: **`:>> fichero`**
  - **`while : ; do echo "siempre"; done`**

Optimizaciones

- Listas AND y OR:
  - AND (&&): **`comando1 && comando2`**

```
if comando1
then
   comando2
fi
```

    - `test –f “/etc/passwd” && echo “passwd encontrado”`

Optimizaciones

- Listas AND y OR:
  - OR (||): **`comando1 || comando2`**

```
if comando1
then
   :
else
   comando2
fi
```

  - **`test -f "/etc/passwd" || echo "error: sin passwd"`**

Optimizaciones

- Bloques:
  - { }: agrupar E/S de comandos:

```
{
  echo "Listado..."; ls;
  echo "Fin de listado"
} > listado.txt
```

  - También se puede hacer con bucles:

```
for F in *
do
  cat $F
done | wc --lines
```

Optimizaciones

- Bloques:
  - ( ): subshells.

```
(sleep 3 && echo fin shell1) &
(sleep 5 && echo fin shell2) &
```

- Señales:

```
trap 'echo capturada señal' 2 3

kill -2 PID
kill -3 PID
```

Referencias

- BASH Programming - Introduction HOW-TO:
  http://www.tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html
- Advanced Bash-Scripting Guide:
  http://www.tldp.org/LDP/abs/html/
- Guía de Administración de Redes con Linux:
  http://es.tldp.org/Manuales-LuCAS/GARL2/garl2/
- UNIX Security:
  http://es.tldp.org/Manuales-LuCAS/SEGUNIX/unixsec-2.1-html
- Kernel HOWTO:
  http://www.tldp.org/HOWTO/Kernel-HOWTO.html