Shell scripting

Sistemas Operativos

Enrique Soriano, Gorka Guardiola

GSYC

28 de noviembre de 2018

Universidad
Rey Juan Carlos

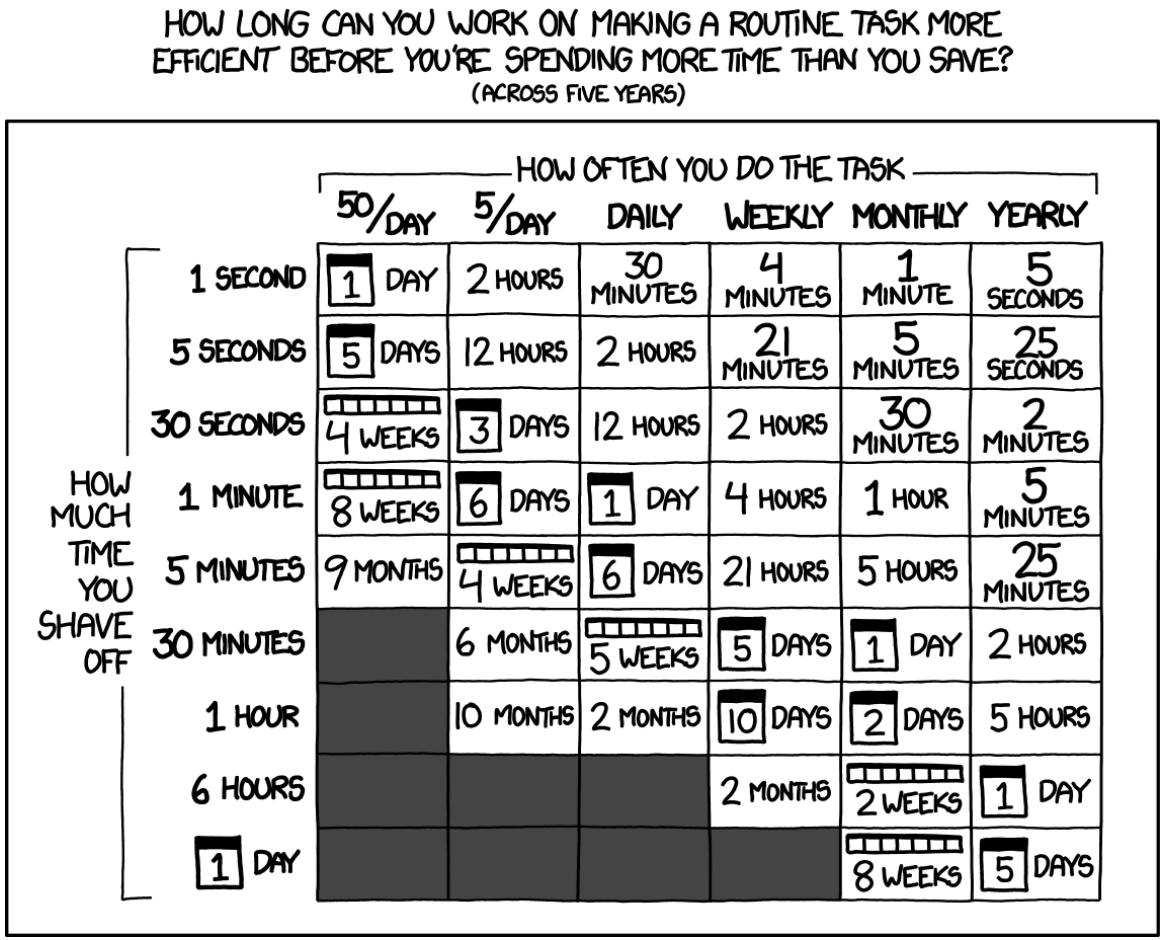
¿Cuándo hago un script de Shell?

- Pasos para realizar una tarea:
  1. Mirar si hay alguna herramienta que haga lo que queremos → buscar en el manual.
  2. Si no encontramos, intentar combinar distintas herramientas → **programar un script de Shell**. La primera aproximación debe ser pipelines de filtros, etc.
     IDEA: combinar herramientas que hacen bien una única tarea para llevar a cabo tareas más complejas.
  3. Si no podemos, hacer una herramienta → programada en C, Python, Java, Ada, Go, ...

¿Qué tipo de cosas ?

- La shell es especialmente buena
  - Para tareas que hago una vez
  - Para automatizar tareas (con un IDE, Makefile)
  - Para procesar texto

Automatizar



Shells

- `sh` es la shell original de Unix, escrita por Ken Thompson. Fue rescrito por Stephen Bourne en 1979 para Unix Version 7: *bourne shell*.
- Los sistemas derivados usan distintas shells: `sh`, `ash`, `bash`, `dash`, `ksh`, `csh`, `tcsh`, `zsh`, `rc`, etc.
- Cada una tiene sus características, pero también tienen mucho en común.
- En sistemas modernos, `/bin/sh` suele ser un enlace simbólico a su shell por omisión para ejecutar scripts. En Ubuntu y Debian es dash[1].
- Política: los scripts que tienen `#!/bin/sh` deben usar únicamente las características POSIX (IEEE Std 1003.1-2017): el subconjunto común que tienen la mayoría de las shells. Así, los scripts pueden ser portables entre distintos sistemas.

[1] No confundir con el shell por omisión para un terminal, que es bash.

Un script:

- Tiene que tener permisos de ejecución.
- Hay comandos que se implementan dentro del shell (no se ejecuta un fichero externo al shell, es una parte del propio shell). Se llaman *built-in*.
- El comando *built-in* `exit` sale del script con el status indicado en su argumento.
- Si un script no sale con `exit`, deja el status que tiene `$?`.

```
#!/bin/sh

# este es un hola mundo en sh
# esto es un comentario

echo hello world
exit 0
```

Un script:

- Una ventaja de la shell, es que puedo probar de forma interactiva
- No escribo el script directamente, voy probando los comandos
- O ni siquiera escribo el script (escribo los comandos directamente)

Ya conocemos:

- | es un pipe
- & ejecuta un comando en background
- $ se usa para las variables. $var es lo mismo que ${var}
- " y ' se usan para escapar cadenas (las dobles expanden algunas cosas)
- < , < y >> son redirecciones
  - 5> para redir del fd 5
  - 5>&3 para hacer dup del 5 al 3
  - Cuando hay varios el orden importa ej: `2>&1 >/dev/null`
- \ se usa para escapar caracteres
- && , || para ejecución condicional
- Globbing (wildcards): ? * $[a-z]$ etc.

Parámetros posicionales

- Se pueden acceder a los parámetros que se han pasado al script con $1, $2, $3 ...
- $0 expande al nombre con el que se ha invocado el script.
- $# expande al número de parámetros (sin contar el 0).
- $* expande a los parámetros posicionales.
- "$*" expande a "$1 $2 ..."
- $@ expande a los parámetros posicionales (igual que $* pero separados)
- "$@" expande a "$1" "$2" ...
- `shift` desplaza los parámetros (p. ej. $4 pasará a ser $3). Se actualiza el valor de $#.
    - Útil para parámetros optativos (pongo lo que sea, o hago shift, el resto igual)

Agrupaciones

- Si queremos ejecutar comandos en un subshell:

(comando; comando; ...)

- Si queremos ejecutar una agrupación de comandos en el shell actual:

{ comando; comando; ... }

Ejemplo:

```
$> { echo uno; echo dos; } | tr o 0
un0
d0s
$> { echo los ficheros de /tmp son; ls /tmp; } > ficheros
```

Agrupaciones

- Ejecutar en un subshell útil
- Para no cambiar el entorno en el shell actual (`cd`, `export`)

Ejemplo:

```
#sigo en tmp al final:
$> pwd; (cd /etc; ls apt;); pwd;
/tmp
apt.conf.d      sources.list
preferences.d   sources.list.d   trusted.gpg    trusted.gpg.d
/tmp
#BLA no existe al final:
$> echo z$BLA; (export BLA=bla; echo $BLA;); echo z$BLA;
z
bla
z
$>
```

Sustitución de comando

- Se sustituye el comando por su salida.
- Se puede escribir de dos formas:

  `$(comando)`

  `‘comando‘`

```
$>  l=$(wc -l /tmp/a | cut -d' ' -f1)
$>  echo $l
31
$>
```

if

Las condiciones depende del status de salida del comando: éxito es verdadero, fallo es falso.

```
if comando
then
        comandos
elif comando
then
        comandos
else
        comandos
fi
```

case

Los casos pueden contener patrones de globbing.

```
case palabra  in
patrón1)
     comandos
     ;;
patrón2 | patrón3)
     comandos
     ;;
*)   # este es el default
     comandos
     ;;
esac
```

Bucles

```
while comando
do
        comandos
done
```

```
for variable in palabra1 palabra2 palabraN
do
        comandos
done
```

Read

- El comando `read` lee una línea de su entrada estándar y la guarda en la variable que se le pasa como argumento.
- Se puede usar para procesar la entrada línea a línea en un bucle.
- Solo debemos hacer eso cuando no tenemos ningún filtro o pipeline que nos sirva para hacer lo que queremos.

Read

- Por ejemplo, esto itera 2 veces:

```
echo 'a b
c d' > /tmp/e

while read line
do
  echo $line
done <  /tmp/e
```

- Esto itera 4 veces:

```
for x in `cat /tmp/e`
do
  echo $x
done
```

Variable IFS

- Esta variable contiene los caracteres que se usan como separadores entre campos.
- Por omisión contiene el tabulador, espacio y el salto de línea.
- Hay que tener cuidado: cambiar el valor de esta variable rompe las cosas.
- Mejor en subshell.

```
$>  for i in $(echo uno dos tres) ; do echo $i ; done
uno
dos
tres
$>  export IFS=-
$>  for i in $(echo uno dos tres) ; do echo $i ; done
uno dos tres
$
```

Funciones

- Se pueden definir funciones. Sus parámetros se acceder como los parámetros posicionales. P. ej.:

```
hello () {
        echo hola $1
        shift
        echo adios $1
}

...
# ejecutamos la función
hello uno dos
```

Test

El comando `test` sirve para comprobar condiciones de distinto tipo.

Ficheros:

- -f fichero
  si existe el fichero
- -d dir
  si existe el directorio

Test

Cadenas:

- -n String1
  si la longitud de la string no es cero
- -z String1
  si la longitud de la string es cero
- String1 = String2
  si son iguales
- String1 != String2
  String1 and String2 variables no son idénticas
- String1
  si la string no es nula

Test

Enteros:

- Integer1 -eq Integer2
  si los enteros Integer1 e Integer2 son iguales.

Otros operadores:

- -ne: not equal
- -gt: greater than
- -ge: greater or equal
- -lt: less than
- -le: less or equal

Test

Test también se puede usar así:

- Esto:

```
[  $a -eq $b  ]
```

- es lo mismo que esto:

```
test $a -eq $b
```

Operaciones aritméticas

Para operaciones básicas con enteros podemos usar el propio shell. También podemos usar el comando `bc`.

- Esto:

```
$((5 + 7))
```

- se reemplaza por

```
12
```

Filtros útiles

- `sort`
  ordena las líneas de varias formas.
- `uniq`
  elimina líneas contiguas repetidas.
- `tail`
  muestra las últimas líneas.

P. ej:

```
$> ps | tail +3 # a partir de la 3ª
$> ps | tail -3 # las 3 ultimas
$> seq 1 1000 | sort
$> seq 1 1000 | sort -n
```

Sort

- Puede recibir una lista de columnas (empezando por la 1)
- Y un separador
- Y ordena por esos campos como clave (es un intervalo de campos)
- Ojo con estabilidad (`-s`)

```
$> cat x.txt
1-2-4
2-3-3
2-2-1
2-1-4
$> sort -k2 -t\- x.txt
2-1-4
2-2-1
1-2-4
2-3-3
$> sort -k1,2 -t\- x.txt
1-2-4
2-1-4
2-2-1
2-3-3
```

Comandos útiles

- `diff`
  compara ficheros de texto línea a línea
- `cmp`
  compara ficheros binarios byte a byte

P. ej:

```
$> diff -n fich1 fich2
$> cmp fich1 fich2
```

Tr

- Traduce caracteres. El primer argumento es el conjunto de caracteres a traducir. El segundo es el conjunto al que se traducen. El enésimo carácter del primer conjunto se traduce por el enésimo carácter del segundo.
- `-d`
  Borra los caracteres del único conjunto que se le pasa como argumento.
- Se le pueden dar rangos, p. ej.
  `$> cat fichero | tr a-z A-Z`

Expresiones regulares (*regexp*)

- Es un lenguaje formal para describir/buscar cadenas de caracteres.
- Parecidas a los patrones de la Shell o de globbing, pero más potentes.
- Veremos las que se llaman *extended regular expressions*. Es un estádar de POSIX.
- Una string encaja con sí misma, por ejemplo `'a'` con `'a'`.

Expresiones regulares (*regexp*)

- .
  encaja con cualquier carácter, por ejemplo 'a'.
- `[conjunto]`
  encaja con cualquier carácter en el conjunto, por ejemplo `[abc]` encaja con `'a'`. Se pueden especificar rangos, p. ej. `[a-zA-Z]`.
- `[^conjunto]`
  encaja con cualquier carácter que **no esté** en el conjunto, por ejemplo `[^abc]` NO encaja con `'a'`, sin embargo sí encaja con `'z'`.

Expresiones regulares (*regexp*)

- ^
  encaja con *principio de línea*.
- $
  encaja con *final de línea*.
- Una regexp e_1 concatenada a otra regexp e_2, $e_1 e_2$, encaja con una string si una parte p_1 de la string encaja con e_1 y otra parte contigua, p_2, encaja con e_2.

P. ej:
`'az'` encaja con la regexp `[a-d]z`

Expresiones regulares (*regexp*)

- `exp*`
  encaja si aparece **cero o más veces** la regexp que lo precede.
- `exp+`
  encaja si aparece **una o más veces** la regexp que lo precede.

P. ej:

`'aaa'` encaja con la regexp `a*`

`'baaa'` encaja con la regexp `ba+`

`'bb'` encaja con la regexp `ba*`

`'bb'` no encaja con la regexp `ba+`

Expresiones regulares (*regexp*)

- `exp?`
  encaja si aparece **cero o una vez** la regexp que lo precede. Se utiliza para partes opcionales.
- `(exp)`
  agrupa expresiones regulares.

P. ej:
`'az', 'av', 'a'` encajan con la regexp az?
`'abab'` encaja con la regexp `(ab)+`
`'abab', 'ababab', 'abababababa'` encajan con la regexp `(ab)+`

Expresiones regulares (*regexp*)

- `exp | exp`
  si encaja con alguna de las regexp que están separadas por la barras
- `\`
  carácter de escape: hace que el símbolo pierda su significado especial.

P. ej:
`'aass'` encaja con la regexp `(aass|booo)`
`'hola*'` encaja con la regexp `a\*`

Egrep

- Filtra líneas usando expresiones regulares.
- `-v`
  Realiza lo inverso: imprime las líneas que no encajan.
- `-n`
  Indica el número de línea.
- `-e`
  indica que el siguiente argumento es una expresión.
- `-q`
  silencioso, no saca nada por la salida (cuando solo nos interesa el status de salida).

Sed

- Stream Editor
- Editor de flujos de texto con comandos.
- Basado en Ed (editor con comandos, tatarabuelo de vi).
- Muchas de las cosas de sed, igual en ed.

Sed

- Es un editor: aplica el comando de sed a cada línea que lee y escribe el resultado por su salida. Sin el modificador `-n`, escribe todas las líneas después de procesarlas.
- Si queremos usar expresiones regulares extendidas, hay que usar la opción -E.
- Comandos:
  `q` $\rightarrow$ Sale del programa.
  `d` $\rightarrow$ Borra la línea.
  `p` $\rightarrow$ Imprime la línea. (correr con -n)
  `r` $\rightarrow$ Lee e inserta un fichero.
  `s` $\rightarrow$ Sustituye. $\leftarrow$ la que más se usa!!!

Sed

- Direcciones:
  `número` → actúa sobre esa línea.
  `/regexp/` → líneas que encajan con la regexp.
  `$` → la última línea.
- Se pueden usar intervalos:
  `número,número` → actúa en ese intervalo.
  `número,$` → desde la línea *número* hasta la última.
  `número,/regexp/` → desde la línea *número* hasta la primera que encaje con regexp.

Sed

Ejemplos:
`sed '3,6d'` $\rightarrow$ borra las líneas de la 3 a la 6
`sed -E -n '/BEGIN|begin/,/END|end/p'` $\rightarrow$ imprime las líneas entre esas regexp
`sed '3q'` $\rightarrow$ imprime las 3 primeras líneas.
`sed -n '13,$p'` $\rightarrow$ imprime desde la línea 13 hasta la última.
`sed -E '/[Hh]ola/d'` $\rightarrow$ borra las líneas que contienen 'Hola' u 'hola'.

Sed

Sustitución

- `sed 's/regexp/sustitución/'` $\rightarrow$ sustituye la primero subcadena que encaja con la exp. por la cadena *sustitución*.
- `sed 's/regexp/sustitución/g'` $\rightarrow$ sustituye todas las subcadenas de la línea que encajan con la exp. por la cadena *sustitución*.
- `sed 's/(regexp)regexp.../ \1 sustitución/g'` $\rightarrow$ usa las subcadenas que encajaron con las agrupaciones en la cadena de sustitución.

Sed

Ejemplos

`sed 's/[0-9]/X/'` $\rightarrow$ el primer dígito de la línea se sustituye por una X.

`sed 's/[0-9]/X/g'` $\rightarrow$ todos los dígitos de la línea se sustituyen por una X.

```
sed 's/^([A-Za-z]+)[ ]+([A-Z]+)/NOMBRE:\1 NOTA:\2/g'
```

hacer mykill.sh

AWK

Imprimir:

- Lenguaje completo de programación de texto.
- Útil, veremos sólo la superficie.

AWK

Imprimir:

- print
  Sentencia que imprime los operandos. Si se separan con comas, inserta un espacio. Al final imprime un salto de línea.
- printf()
  Función que imprime, ofrece control sobre el formato de forma similar a la función de libc para C:

```
$> ls -l | awk '{ printf("Size:%08d KBytes\n", $6) }'
```

AWK

Variables:

- $0
  La línea que está procesando.
- $1, $2 ...
  El primer, segundo... campo de la línea.
- NR
  Número del registro (línea) que se está procesando.
- Ejemplo
  para imprimir la tercera y segunda columna de un csv:

```
$> cat a.txt|awk -F, '{printf("%d\t%d", $3, $2)}'
```

AWK

Variables:

- NF
  Número del campos del registro que se está procesando.
- var=contenido
  Se pueden declarar variables dentro del programa. Con el modificador -v se pueden pasar variables al programa.

```
$> ls -l | awk '
{
size=$6 ; printf("Size: %08d KBytes\n", size)
}'
```

AWK

```
patrón { programa }
```

Actuando sólo en unas líneas, que se ajustan a un patrón, que puede ser:

- Expresión regular
  Se procesan las líneas que encajen con la regexp.

```
$> ls -l | awk '/[Dd]esktop/{ print $1 }'

$> ls -l | awk '$1 ~  /[Dd]esktop/ { print $1 }'
```

AWK

- Expresión de relación
  Se comparan valores y se evalúa la expresión.

```
$> ls -l | awk ' NR >= 5 && NR <= 10 { print $1 }'
```

AWK

Inicialización y finalización:

```
BEGIN{
...
}

patrón{
...
}

END{
...
}
```

AWK

Arrays asociativos:

- Cómodos, imprimir duplicadas:

```
$> awk '{dups[$1]++} END{for (num in dups) {print num,dups[num]}}' data
```

Recorrer un árbol

- Para recorrese un árbol de ficheros
  - `du -a .`
  - `find .`

Join

- `join`
- Extremadamente útil
- Hace un *join* relacional de dos columnas (tienen que estar ordenadas)

```
$> echo '
> a bla
> b ble
> c blo' > a.txt
$> echo '
a ta
b te
c to' > b.txt
$> join a.txt b.txt
a bla ta
b ble te
c blo to
```

Peligro

- Ojo con las redirecciones
- Esto crea un fichero vacío
- ¿Por qué?

```
$> echo '
> a bla
> b ble
> c blo' > a.txt
$> cat a.txt | tail > a.txt
$> cat a.txt
```

Join

- `join` quita las que no están en alguno de los dos (inner join)
- Tienen que estar ordenadas, usar sort antes
- Igual que sort puede usar diferentes campos

Otros comandos

- Para ejecutar comandos sobre una lista
  - `echo a b c |xargs ls -l`

Comandos texto

- `cut` y `paste`
- Mejor dominar `awk` (se puede hacer todo)