2. Shell scripting

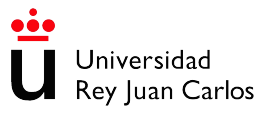
2.1 Scripts para Bourne Again Shell (bash)

Elena García-Morato, Felipe Ortega, Enrique Soriano, Gorka Guardiola

GSyC, ETSIT. URJC.

Laboratorio de Sistemas (LSIS)

9 febrero, 2023



Tema 2 - Shell scripting

2.1 Scripts para Bourne Again Shell (bash)

Contenidos

2.1.1 Scripts de Shell

Scripts de Shell

- Son programas interpretados mediante el mecanismo *hash bang*.
  - Los primeros bytes del contenido de un fichero (normalmente 2 o más) identifican de qué tipo es.
  - Si el fichero empieza por los dos caracteres `#!` (pronunciado *hash bang* en inglés) es un **fichero interpretado**.
  - Ejemplo de contenido de un *script* de Shell.

```
#!/bin/sh
echo hola mundo
```

- El mecanismo permite especificar la ruta a un comando externo que ejecuta el fichero.
- Este mecanismo lo usan los *scripts* de Python, la Shell, etc.

Scripts de Shell

- La primera línea del fichero especifica que es un programa interpretado y que el comando que lo interpreta es la Shell, en la ruta `/bin/sh`.
- Si queremos depurar el programa para detectar errores se puede añadir el parámetro `-x`. Esto provoca que la Shell escriba el comando antes de ejecutarlo.

```
#!/bin/sh
echo hola mundo
pwd
exit 0
```

¿Cuándo crear un script de Shell?

- Antes de plantearte crear un script de Shell, primero comprueba si ya hay alguna herramienta que haga lo que necesitas → busca el manual.
- Si no la encuentras, intenta combinar distintas herramientas en la línea de comandos.
  - Por ejemplo usando pipes, redirecciones, etc.
- Cuando esa estrategia tampoco funciona entonces la solución puede ser programar un **script de Shell**.
- La idea es combinar herramientas que hacen bien una tarea individual para resolver tareas más complejas → Debes **descomponer** el problema en **tareas sencillas**.

Problemas adecuados para un script de Shell

- Tareas que hago una vez y son largas o complicadas de hacer manualmente.
- Automatizar tareas recurrentes (e.g. un Makefile para compilar un proyecto, limpiar archivos de registro, etc.).
- Procesamiento de texto.
  - La Shell tiene programas especialmente potentes para búsqueda en textos, reemplazo de cadenas de caracteres por otras, búsqueda de patrones, etc.

Cuidado con el tiempo que te lleva crear el script...

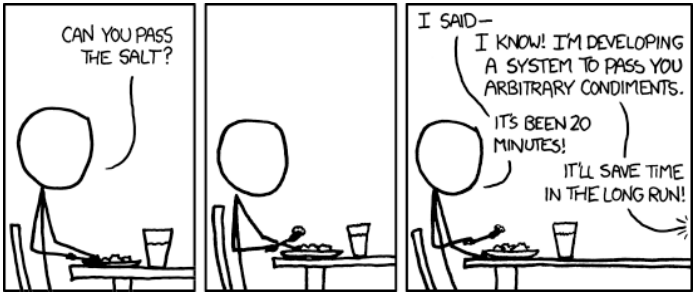


Figura 1: Créditos: https://xkcd.com.

Distintas Shells

- En la cabecera de nuestro *script* podemos elegir el intérprete de Shell que ejecutará nuestro programa.
- `sh` es la Shell original de Unix. Los sistemas derivados de Unix incluyen diferentes Shells: `sh`, `dash`, `ksh`, `csh`, `zsh`, etc.
- Todas tienen mucho en común, pero también características que las diferencian entre sí.
- En los sistemas modernos, la ruta `/bin/sh` suele ser un enlace simbólico a su Shell por defecto para ejecutar *scripts*.
- Los *scripts* deben incluir solo características comunes a todas las Shell para ser portables entre diferentes sistemas. Este conjunto de características comunes se denomina POSIX.

2.1.2 Ejecución *scripts* de Shell

Ejecución de *scripts* de Shell

- El *script* debe tener permisos de ejecución.
- Por ejemplo, si mi *script* está en un fichero `miscript.sh` escribo en la terminal:

```
$ chmod +x miscript.sh
```

- El *script* debe finalizar siempre con el comando integrado `exit` seguido de un argumento numérico (entero).
- El argumento es el código de salida. Por ejemplo, si el script finaliza correctamente:

```
exit 0
```

Ejecución de *scripts* de Shell

- Las líneas de comentario empiezan por #.

```
#!/bin/sh

# Esto es un comentario
# Script que escribe hola mundo por pantalla

echo hello world
exit 0
```

- Recuerda que es muy recomendable incluir comentarios para que otros sepan qué has hecho, y para que tú mismo lo recuerdes pasado un tiempo.

Contenido de *scripts* de Shell

- Un *script* de Shell puede contener todos los elementos que hemos visto hasta ahora en el intérprete de comandos interactivo.
  - Comandos *built-in*.
  - Definición de variables, acceso y modificación de su valor.
  - Pipes, redirecciones, etc.
- Sustitución de comandos: se puede sustituir el comando por su salida de dos formas.
  - `$(comando)`.
  - `'comando'`.
- Ejemplo: si el fichero `/tmp/a` contiene 31 líneas:

```
$ l=$(wc -l /tmp/a)
$ echo $l
31
```

Estado de salida (status)

- Cuando un comando finaliza deja un estado de salida (status) que informa sobre el modo en que finalizó.
- El estado con el que acabó la ejecución del último comando (o *pipeline*) se puede consultar con `$?`.
- Siempre es un número entero.
- 0 indica que el comando o *pipeline* finalizó con éxito.
- Si finaliza con un número positivo es un error (hay algunos valores ya estandarizados).
- Es muy recomendable acabar siempre un script con el comando `exit` seguido del argumento numérico con el código de finalización que queremos devolver.

2.1.3 Parámetros posicionales

Parámetros de entrada posicionales

- Es posible pasar parámetros adicionales en el momento en que invocamos al *script* en la línea de comandos para ejecutarlo.
- Dentro del *script* se pueden recuperar los parámetros que se pasaron al invocarlo con `$1`, `$2`, `$3`, etc. El número indica la posición del parámetro a recuperar.
- `$0` representa el propio nombre con el que se invocó al *script*.
- `$#` representa el número de parámetros (sin contar el 0).
- `$*` devuelve todos los parámetros posicionales (en una sola cadena).
- "`$*`" devuelve "`$1 $2`" ... (en una sola cadena).
- `$@` representa los parámetros posicionales (igulal que `$*` pero separados en varias cadenas).
- "`$@`" devuelve "`$1`" "`$2`" ...
- `shift` desplaza los parámetros. Por ejemplo, `$3` pasa a ser `$2` y se actualiza el valor de `$#`.

Ejemplo: parámetros posicionales

```
$ cat param.sh
#!/bin/sh
echo \$0 es $0
echo \$\# es $#
echo \$\* es $*
echo $1 $2 $3 $4
echo \$\@ es $@
shift
shift
echo $1 $2 $3 $4 $#
$ ./param.sh -a -b -c fich
$0 es ./param.sh
$# es 4
$* es -a -b -c fich
-a -b -c fich
$@ es -a -b -c fich
-c fich 2
```

Ejemplo: parámetros posicionales

```
$ ./param.sh -a fich
$0 es ./param.sh
$# es 2
$* es -a fich
-a fich
$@ es -a fich
0
```

2.1.4 Control de flujo

Sentencia `if`

- Permite decidir si se ejecuta un comando o grupo de comandos según el status de salida con el que acabe el comando de la condición.
- Sintaxis parecida a muchos lenguajes de programación.
  - Primer bloque condicional con `if`.
  - Sucesivos bloques alternativos (va probando si falla el primero) con `elif`.
  - Bloque por defecto (si no entra en ninguno de los anteriores) con `else`.
  - La estructura acaba con `fi`.
- Si el comando de la condición acaba con éxito (status 0) entonces es verdadero, si devuelve fallo es falso.

Sentencia `if`

```
if comando
then
    comando1
    comando2
elif otrocomando
then
    comando3
    comando4
else
    comando 5
    comando 6
fi
```

Sentencia `if`

- Se puede negar la condición del resultado de un comando con el carácter `!`

```
if ! comando
then
    comando1
    comando2
fi
```

Sentencia case

- Ejecución condicional en función de si localiza un patrón.

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

Bucles

- Sólo hay `while` y `for`.

```
# Ejemplo de bucle while
while comando
do
    comandos
done
# Ejemplo de bucle for
for variable in palabra1 palabra2 palabraN
do
    comandos
done
```

Sentencias de control en comandos

- Se puede poner el carácter ; al final de una sentencia para continuar escribiendo en la misma línea.
- Esto es útil en la terminal interactiva, puesto que el carácter de final de línea acabaría el comando

```
while ls|egrep '\.z$'; do
    comandos
done
```

2.1.5 Otros elementos de programación en Shell *scripting*

Comando read

- El comando `read` permite leer una línea (cadena de caracteres) de su entrada estándar y guardarla en la variable que se le pasa como argumento.
- Por ejemplo, esto permite procesar la entrada línea por línea mediante un bucle.
- Por supuesto, si ya tenemos otro comando, filtro o *pipeline* de comandos que ya hace lo que necesitamos es mejor usar esa opción en lugar de `read`.
  - Los comandos ya disponibles son muchísimo más eficientes.

Comando read: ejemplo

```
# En la terminal, creamos un fichero con dos líneas y dos letras en cada línea
$ echo 'a b
  c d' > /tmp/e
# Ahora, creamos y ejecutamos un shell script para leer cada línea
# y repetirla por pantalla. Fichero de entrada: /tmp/e
while read line
do
    echo $line
done <  /tmp/e
# Sin embargo, este otro shell script itera 4 veces (procesa letra a letra).
for x in 'cat /tmp/e'
do
    echo $x
done
```

Variable IFS

- Es una variable que contiene los caracteres reconocidos como separadores entre campos.
- Por defecto, contiene el tabulador, el espacio en blanco y el salto de línea.
- Cuidado: si tocamos indebidamente el valor de esta variable podemos hacer que todo deje de funcionar como es debido.
  - Por ejemplo, el espacio en blanco ya no serviría para separar los argumentos que paso detrás de un comando.

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

Funciones

- Se pueden definir funciones, accediendo a sus parámetros como hacemos con los parámetros posicionales del *script* principal.

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

- Ahora ejecutamos la función.

```
$ hello uno dos
hola uno
adios dos
```

El comando alias

- Define una etiqueta para invocar un comando o conjunto de comandos de forma directa.
- Sin argumentos, `alias` muestra los que hay definidos.
- El comando `unalias` elimina un alias previamente definido.

```
$ alias hmundo='echo hola mundo'
$ hmundo
hola mundo
$ unalias hmundo
$ hmundo
hmundo: command not found
$ alias
alias la='ls -A'
alias ll='ls -alF'
alias ls='ls --color=auto'
$
```

Comando `test`

- Para comprobar diferentes tipos de condiciones.
- Con ficheros:
  - `-f fichero`
    Comprueba si existe el fichero.
  - `-d dir`
    Comprueba si existe el directorio.

Comando test

- Con cadenas de caracteres:
  - `-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`
    Si las cadenas en `String1` y `String2` no son idénticas.
  - `String1`
    . Si la cadena en `String1` no es nula.

Comando test

- Con enteros:
  - `Integer1 -eq Integer2`
    Si los enteros `Integer1` e `Integer2` son iguales.
- Otros posibles operadores con enteros:
  - `-ne` : no igual (comprueba si son distintos).
  - `-gt` : mayor que.
  - `-ge` : mayor o igual que.
  - `-lt` : menor que.
  - `-le` : menor o igual que.

Sintaxis alternativa para `test`

- El comando:

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

- Es equivalente a este otro:

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

Operaciones aritméticas con números enteros

- La Shell permite realizar operaciones básicas que involucren números enteros.
- Otra alternativa es usar el comando bc.

```
$ echo $((5 + 7))
12
$
```

Recorrer árboles de ficheros

- Para recorrer un árbol de ficheros (podemos procesar los resultados en un *script*):
  - `du -a .`
  - `find .`

El comando `join`

- Sirve para calcular la intersección de valores presentes en dos columnas, que tienen que estar previamente ordenadas.

```
$ echo '
a uno
b dos
c tres' > a.txt
$ echo '
a cuatro
b cinco
c seis' > b.txt
$ join a.txt b.txt
a uno cuatro
b dos cinco
c tres seis
```

El comando `join`

- El comando `join` solo devuelve las coincidencias entre ambas tablas, es decir, quita las que solo están en una de las dos.
- Es imprescindible que estén previamente ordenadas, hay que usar antes el comando `sort`.
- Al igual que con `sort`, se pueden usar diferentes campos para calcular la intersección.

El comando xargs

- Permite usar lo que nos viene por la entrada estándar como argumentos de entrada en la ejecución de otro comando.

```
$ echo a b c | xargs ls -l
-rw-rw-r-- 1 jfelipe jfelipe 2 mar    1 16:03 a
-rw-rw-r-- 1 jfelipe jfelipe 2 mar    1 16:03 b
-rw-rw-r-- 1 jfelipe jfelipe 2 mar    1 16:03 c
```

Comandos básicos para cortar y pegar texto

- Comandos `cut` y `paste`.
- Es más potente usar awk (lo veremos en la siguiente sección).

```
$ echo uno dos tres | cut -d' ' -f 1,3
uno tres
$ ps | sed 3q > a
$ seq 1 3 > b
$ paste a b
PID TTY          TIME CMD           1
8462 pts/4       00:00:00 bash      2
11357 pts/4      00:00:00 ps        3
$ paste b a
1       PID TTY          TIME CMD
2       8462 pts/4       00:00:00 bash
3       11357 pts/4      00:00:00 ps
```

Para saber más

- El capítulo 8 *"The Bourne Again Shell (bash)"* y el capítulo 10 *"Programming the Bourne Again Shell (bash)"* de [3] son una excelente fuente de información sobre Bash y shell *scripting*. Accede al libro en O'Reilly Learning.
- Como su propio nombre indica, el libro de Blum y Bresnahan *Linux Command Line and Shell Scripting Bible* [1] es una fuente casi inagotable de información sobre la línea de comandos y la creación de *scripts* para Shell. El apéndice A contiene tablas resumen de comandos de Bash. Accede al libro en O'Reilly Learning.
- Otra referencia actualizada es [2], que contiene un capítulo con un buen resumen sobre búsqueda de patrones y expresiones regulares. Accede al libro en O'Reilly Learning.

Referencias I

[1] R. Blum y C. Bresnahan. *Linux Command Line and Shell Scripting Bible*. 4ª ed. Bible. Wiley, 2021. ISBN: 9781119700913.

[2] G.S. Naik. *Learning Linux Shell Scripting: Leverage the Power of Shell Scripts to Solve Real-World Problems, 2nd Edition*. 2ª ed. Packt Publishing, 2018. ISBN: 9781788993197.

[3] M.G. Sobell y M. Helmke. *A Practical Guide to Linux Commands, Editors, and Shell Programming*. 4ª ed. Pearson Education, nov. de 2017. ISBN: 9780134774619.