



Bisoños Usuarios de GNU/Linux de Mallorca y Alrededores | Bergantells Usuaris de GNU/Linux de Mallorca i Afegitons

Dale potencia a tus bourne-shell scripts ^(17803 lectures)

Per Domingo Fiesta Segura, [C2H5OH](http://etanol.doesntexist.org) (<http://etanol.doesntexist.org>)

Creat el 18/08/2004 04:35 modificat el 18/08/2004 04:35

*Aquí os propongo algunos trucos para programar scripts. Supongo que los que estéis interesados ya sabéis **algo** sobre shellscripts.*

Índice

1. [Manipulando las expansiones de variables](#)⁽¹⁾
2. [Magia con el \\$IFS](#)⁽²⁾
3. [Manejo de descriptores de fichero](#)⁽³⁾
- 3.1. [Introducción a redirecciones](#)⁽⁴⁾
- 3.2. [Operaciones de redirección](#)⁽⁵⁾
- 3.3. [Modo de uso](#)⁽⁶⁾
- 3.4. [Otras redirecciones](#)⁽⁷⁾
4. [Manejo de señales](#)⁽⁸⁾
5. [Colofón](#)⁽⁹⁾

Para hacer el artículo más breve y llevadero me limitaré a exponer ejemplos con alguna explicación ocasional. El objetivo es despertar vuestra curiosidad y que experimentéis por iniciativa propia.

Todo lo que aquí se comenta está explicado en la página man del sh (1)

1. Manipulando las expansiones de variables

Ya que siempre estamos manejando texto que entra, texto que sale, tuberías y demás historias, existen algunos mecanismos prácticos antes de recurrir al *Perl*, *Awk* o *Sed*. Y de paso nos ahorramos algunos ciclos.

Disponemos de unos pocos operadores para controlar la expansión de las variables del shell, aquí os describo los más sencillos.

- **Valor alternativo:** `${variable:-valor}` Si variable **NO** está definida entonces la expansión produce valor, si variable sí está definida se expande su contenido.
- **Valor fijo:** `${variable:+valor}` Al contrario que el anterior si variable está definida entonces la expansión es valor, si variable **NO** está definida entonces se produce una cadena vacía "".
- **Eliminar prefijo:** `${variable#prefijo}` Donde prefijo es un patrón (expresión regular en formato sh). Dicho patrón se aplica al contenido de variable por el principio y si hay coincidencia, ésta se omite en la expansión. En caso de que no coincida se expande variable de forma normal
- **Eliminar sufijo:** `${variable%sufijo}` Igual que el anterior salvo que se busca el patrón desde el final del contenido de variable.

Los operadores para eliminar prefijo y sufijo buscan la coincidencia más corta, existen versiones para buscar la coincidencia más larga: `${variable##prefijo}` y `${variable%%sufijo}` respectivamente. Además, donde indicamos el prefijo o sufijo podemos poner cualquier expresión que pueda ser evaluada por el shell. Como por ejemplo:



```

etanol@botijo:~$ texto=piri-jandemor
etanol@botijo:~$ prefijo=piri-
etanol@botijo:~$ echo ${texto#piri-}
jandemor
etanol@botijo:~$ echo ${texto#$prefijo}
jandemor
etanol@botijo:~$ echo ${texto#$(echo piri-)}
jandemor

```

Hay unos cuantos más, pero estos me han parecido los más útiles. Vamos con unos ejemplos:

```

#!/bin/sh
# join.sh: Script para concatenar todos los ficheros 00-nombre
# 01-nombre 02-nombre etc... en nombre. Esto se hace para todos los ficheros
# que empiecen por dos dígitos y un guión en el directorio actual.

for parte in [0-9][0-9]-*; do # El comando ls(1) ya hace "sort"
    cat $parte >>${parte#[0-9][0-9]-}
done

```

Otro ejemplo:

```

#!/bin/sh
# decent-extension.sh extensión_vieja extensión_nueva: Script para renombrar
# todos los ficheros con "extensión_vieja" por "extensión_nueva". Por ejemplo:
# ./decent-extension.sh TXT txt

for fichero in *.$1; do
    mv -v $fichero ${fichero%.$1}.$2
done

```

2. Magia con el \$IFS

IFS son las siglas de *Inter Field Separator*, que significa **separador de campos**. Se trata de una variable muy especial para el shell, en ella se encuentran todos los caracteres que pueden ser **delimitadores** de campo o de palabra. Viene a ser como el FS del *awk*

Esta característica del shell no tiene mucho más misterio. Para verificarlo usaremos un ejemplo:

```

#!/bin/sh
# which.sh nombre_binario: Implementación alternativa del conocido programa.
#
# Esto sirve de emergencia en caso de no disponer o no encontrar el verdadero
# comando "which".

IFS=":"
for dir in $PATH; do
    if test -x "${dir%}/$1"; then
        # Eliminamos la "/" del final si la hay, por si acaso
        echo "${dir%}/$1"
        exit 0
    fi
done
exit 1

```



3. Manejo de descriptores de fichero

Una de las primeras cosas que se aprende a hacer con el shell (tanto interactivamente como en script) es utilizar las redirecciones de la entrada/salida estándar a ficheros alternativos.

Imagino que sabéis lo que es un descriptor de fichero. Para los que no lo sepan basta con decir que es un índice de una tabla de información sobre ficheros abiertos que gestiona el sistema operativo para cada proceso; por eso el descriptor de fichero es un entero.

3.1. Introducción a redirecciones

Las redirecciones no son más que operaciones sencillas sobre ficheros que podemos realizar desde el shell. Podemos abrir ficheros para lectura, para escritura destructiva^[1] (primero trunca y luego escribe), para escritura constructiva (abre y empieza a escribir desde el final) y para lectura/escritura.

Normalmente es más fácil de entender cuando uno piensa en términos de **entrada** y **salida** de texto, pero en algunos casos resulta más cómodo imaginar las llamadas a sistema que se ejecutarán al interpretar la redirección.

Si no se indican los descriptores de ficheros de forma explícita por defecto se asume 0 (entrada estándar) para las redirecciones de **entrada** (apertura de ficheros para **lectura**) y 1 (salida estándar) para las redirecciones de **salida** (apertura de ficheros para **escritura**). También se toma 0 por defecto para lectura/escritura.

3.2. Operaciones de redirección

Las que todos conocemos. Abriendo ficheros y asociándolos a descriptores. Recordemos que si no se indica el descriptor se toma 0 para entrada o entrada/salida y 1 para salida y *append*.

- **Entrada o lectura:** `num < fichero`
- **Salida o escritura:** `num > fichero`
- **Append:** `num >> fichero`
- **Lectura escritura:** `num <> fichero` (¿Esto lo usa alguien?)

También podemos duplicar, cerrar y mover descriptores de ficheros **que ya estén abiertos**.

- **Copia descriptor de entrada:** `copia < &original)`
- **Copia descriptor de salida:** `copia > &original`
- **Cerrar descriptor de entrada:** `descriptor < &-`
- **Cerrar descriptor de salida:** `descriptor > &-`
- **Mover descriptor de entrada:** `nuevo < &original- (original es cerrado)`
- **Mover descriptor de salida:** `nuevo > &original- (original es cerrado)`

Tanto `copia`, `original`, `nuevo` como `descriptor` deben de ser (o deben expandirse a) un número entero que se corresponda con un descriptor **abierto**. Nuevamente insistir en que si el número antes del operador de redirección (`<`, `>`) no está presente se tomarán los valores por defecto 0 ó 1.

3.3. Modo de uso

A niveles prácticos, los operadores para copiar, cerrar y mover descriptores no tienen en cuenta que el fichero sea de lectura y/o escritura. Es decir, si tenemos el descriptor 5 abierto y lo queremos copiar al 7 da igual que hagamos `7<&5` o `7>&5`. Sólo resulta útil diferenciar `<` de `>` cuando el parámetro de la izquierda es la entrada o la salida estándar^[2].

ACLARACIÓN

En una línea de comandos las redirecciones se evalúan de izquierda a derecha. Pongamos por ejemplo el típico caso en que queremos guardar la salida de un programa que da un error para mandarlo a algún desarrollador. Si ejecutamos `programa > fichero.log` los errores no se guardarán porque salen por un descriptor distinto (nº 2 o `STDERR`).



Como queremos mandar tanto STDOUT como STDERR a `fichero.log` probamos con programa `2>&1 >fichero.log`. Esto haría que lo que sale por STDOUT fuera a `fichero.log` pero lo que sale por STDERR saldrá por la terminal, es decir, lo que era STDOUT **antes** de redireccionarlo a `fichero.log`.

¿Cómo solucionamos el problema? Pues **primero** haciendo que STDERR apunte a STDOUT y luego redirigir STDOUT, es decir: programa `>fichero.log 2>&1`. Si usamos bash, existe una abreviatura para redirigir STDOUT y STDERR a la misma vez: programa `&>fichero.log`; ojo que esta abreviatura **NO** es estándar de *bourne shell*, sólo la implementa bash. Si todavía os quedan dudas leed la página man del bash(1) en la sección **REDIRECTION**.

Veamos ahora cómo utilizar los descriptores de ficheros con un ejemplo:

```
#!/bin/sh
# copia2.sh: Crea DOS copias de un fichero de texto.
# Sintaxis:
#
# copia2 origen nombre_copia1 nombre_copia2

escribe2()
{
    echo $@ >&5 # Salida estándar al descriptor 5
    echo $@ >&6 # Salida estándar al descriptor 6
}

test $# -lt 3 && exit 1

# Así es como abrimos o cerramos descriptores de forma permanente en un
# script. Las redirecciones se ejecutan de izquierda a derecha. Así
# pues no es lo mismo hacer:
#   exec >test.log 2<&1
# que hacer:
#   exec 2<&1 >test.log
# Más explicaciones en la página man del sh(1) sección REDIRECTION.

# desc. 5: copia1 (salida)
# desc. 6: copia2 (salida)
# desc. 7: origen (entrada)
exec 5>$2 6>$3 7<$1

IFS="" # Esto es para que el "read" coja toda la línea en una sola variable.
while read linea <&7; do
    escribe2 $linea
done
```

Otro ejemplo de propina:

```
#!/bin/sh
# cat2.sh: Implementación cutre del cat en un script.

if test -r ${1:-""}; then
    exec <$1
fi

IFS=""
while read l; do
    echo "$l"
done
```

Podéis encontrar más ejemplos sencillos y demostraciones prácticas que ilustran las ventajas en tiempo de ejecución que existen al utilizar descriptores, [aquí](#)⁽¹⁰⁾.



3.4. Otras redirecciones

Quizá muchos se pregunten: *Si hay un operador >>, ¿también hay un <<?*. Pues, efectivamente, lo hay. Pero su uso tiene poco que ver con lo hasta ahora mencionado. El operador << se utiliza para crear lo que se conoce como *here document* que no es más que incrustar un fichero de texto (para ser utilizado como entrada estándar) dentro de un script. Por ejemplo:

```
#!/bin/sh

cat <<MARCA_FIN_TEXTO
En un lugar de la mancha, de cuyo nombre no quiero acordarme...
...
...
...y vivieron felices y comieron perdices.
MARCA_FIN_TEXTO
```

[1] Sé que no es un adjetivo muy apropiado.

[2] Para aquellos que necesiten una explicación más técnica, pueden empezar por `man dup2`.

4. Manejo de señales

Otra funcionalidad que no se enseña de buenas a primeras en los tutoriales de scripting es la gestión de señales desde un shell script. Al contrario de lo que pueda parecer, es sencillísimo.

El secreto reside en el uso del comando *trap* para asociar señales a comandos. Como ya he dicho, es un comando **muy** sencillo:

```
trap argumento señal1 señal2 señal3 ...
```

El parámetro *argumento* es el comando que se debe ejecutar cuando se reciba(n) la(s) señal(es) indicada(s) a continuación de éste. Si *argumento* no se indica o es un guión (-) se restaura el comportamiento original asociado a la(s) señal(es) indicada(s). Si *argumento* es la cadena vacía (" " ó ' ') la(s) señal(es) indicada(s) son ignoradas por completo, siempre que sea posible. Si el comando indicado por *argumento* contiene espacios tendréis que entrecomillarlo debidamente.

Para verlo más claro, un ejemplo:

```
#!/bin/sh
# porculo.sh: Cómo sacar de quicio a los amigos con un script inútil

# SIGINT se produce la hacer Ctrl + c en una terminal
senales_bloqueadas="SIGINT"

pregunta()
{
    echo " Así que quieres salir, ¿eh?"
    echo " Contesta:"
    echo " ¿Cuántas muñecas oculta el cuerpo de una mujer?"
    # Obviamente, al hacer Ctrl + c aquí el shell se detiene.
    read num
    if test $num -eq 5; then
        echo " Muy bien, sal cuando quieras."
        trap - $senales_bloqueadas
    else
        echo " Sigue soñando."
    fi
}

trap pregunta $senales_bloqueadas

echo "Tú mismo, yo me lo tomo con calma..."
```



```
while : ; do
    sleep 1d
done
```

5. Colofón

Y, a partir de aquí sólo es probar, probar y probar. Cuanto más se aprende es preguntándose uno mismo: ¿Es posible hacer esto o lo otro?. Y probar. Además, programar scripts es pura diversión cuando uno le coge el truquillo.

Lista de enlaces de este artículo:

1. http://bulma.net/body.phtml?nIdNoticia=2080&nIdPage=1#sec_1
2. http://bulma.net/body.phtml?nIdNoticia=2080&nIdPage=1#sec_2
3. http://bulma.net/body.phtml?nIdNoticia=2080&nIdPage=2#sec_3
4. http://bulma.net/body.phtml?nIdNoticia=2080&nIdPage=2#sec_3_1
5. http://bulma.net/body.phtml?nIdNoticia=2080&nIdPage=2#sec_3_2
6. http://bulma.net/body.phtml?nIdNoticia=2080&nIdPage=2#sec_3_3
7. http://bulma.net/body.phtml?nIdNoticia=2080&nIdPage=2#sec_3_4
8. http://bulma.net/body.phtml?nIdNoticia=2080&nIdPage=3#sec_4
9. http://bulma.net/body.phtml?nIdNoticia=2080&nIdPage=3#sec_5
10. http://www.los-gatos.ca.us/davidbu/faster_sh.html

E-mail del autor: linuxdummybody_ARROBA_ yahoo.es

Podrás encontrar este artículo e información adicional en: <http://bulma.net/body.phtml?nIdNoticia=2080>