

Bash

Apuntes para Profesionales

Chapter 12: Arrays

Section 12.1: Array Assignments

List Assignment

If you are familiar with Perl, C, or Java, you might think that Bash would use commas to separate items in a list. However, this is not the case; instead, Bash uses spaces.

```
# Array in Perl
my @array = (1, 2, 3, 4);

# Array in Bash
array=(1 2 3 4)
```

Create an array with new elements:

```
array=(first element second element third element)
```

Subscript Assignment

Create an array with explicit element indices:

```
array[[1]]=fourth element [[4]]=fifth element
```

Assignment by index

```
array[0]=first element
array[1]=second element
```

Assignment by name (associative array)

```
declare -A array
array[first]=first element
array[second]=second element
```

Dynamic Assignment

Create an array from the output of other command, for example use `seq`:

```
array=( $(seq 1 10) )
```

Assignment from script's input arguments:

```
array=( "$@" )
```

Assignment within loops:

```
while read -r line; do
    array+=( "$line" )
done
# Array append
# Assignment by index
# Increment index
# command substitution
```

Bash Notes for Professionals

Chapter 21: Quoting

Section 21.1: Double quotes for variable and command substitution

Variable substitutions should only be used inside double quotes.

```
echo $(cat /dev/urandom | fold -w 20 | tr -dc 'a-z0-9' | fold -w 1 | xargs -n 1 shuf | fold -w 1 | xargs -n 1 shuf)
# prints 20 random characters
echo $(ls | fold -w 20 | tr -dc 'a-z0-9' | fold -w 1 | xargs -n 1 shuf | fold -w 1 | xargs -n 1 shuf)
# prints 20 random characters
```

Outside of double quotes, `$var` takes the value of `var`, splits it into whitespace-delimited parts, and interprets each part as a glob (wildcard) pattern. Unless you want this behavior, always put `$var` inside double quotes: `"$var"`.

The same applies to command substitutions: `$(command)` is the output of `command`. `$(command)` is the result of splitting on the output.

```
echo "$var" # good
echo "$(command)" # good
echo "$var" # also works, assignment is implicitly double-quoted
echo "$var" # also works, assignment is implicitly double-quoted
echo "$var" # also works, assignment is implicitly double-quoted
echo "$var" # also works, assignment is implicitly double-quoted
```

Command substitutions get their own quoting contexts. Writing arbitrary nested substitutions is easy because the parser will keep track of nesting depth instead of greedily searching for the first `"` character. The StackOverflow syntax highlighter parses this wrong, however. For example:

```
echo "formatted text: $(printf '%s\n' $(cat /dev/urandom | fold -w 20 | tr -dc 'a-z0-9' | fold -w 1 | xargs -n 1 shuf | fold -w 1 | xargs -n 1 shuf))"
```

Variable arguments to a command substitution should be double-quoted inside the expansions as well:

```
echo "$(command "$arg1" "$arg2")"
```

Section 21.2: Difference between double quote and single quote

Double quote	Single quote
Allows variable expansion	Prevents variable expansion
Allows history expansion if enabled	Prevents history expansion
Allows command substitution	Prevents command substitution
<code>+</code> and <code>@</code> can have special meaning	<code>+</code> and <code>@</code> are always literals
Can contain both single quote or double quote	Single quote is not allowed inside single quote
<code>\$</code> , <code>~</code> , <code>`</code> , <code>^</code> can be escaped with <code>\</code> to prevent their special meaning. All of them are literals.	

Properties that are common to both:

- Prevents globbing
- Prevents word splitting

Examples:

Bash Notes for Professionals

Chapter 36: Internal variables

Section 36.1: Bash internal variables at a glance

Variable	Details
\$*	Function/script positional parameters (arguments). Expand as follows: \$* is the same as "\$1" "\$2" ... (Note that it generally makes no sense to leave those arguments unquoted)
\$@	Function/script positional parameters (arguments). Expand as follows: \$@ is the same as "\$1" "\$2" ... (Note that it generally makes no sense to leave those arguments unquoted)
\$1	Process ID of the last process that executed <code>bash</code> , which does not have to be a space. Background (note that it's not necessarily the same as the job's process group ID when job control is enabled)
\$2	ID of the process that executed <code>bash</code>
\$3	Exit status of the last command
\$4	Positional parameters, where $n=1, 2, 3, \dots, 9$
\$5	Positional parameters (same as above), but n can be > 9
\$6	In scripts, path with which the script was invoked with <code>bash -e "script" "\$@"</code> args: same (the first argument after the inline script, otherwise, the arg[0] that <code>bash</code> received)
\$7	Last field of the last command
\$8	Internal field separator
\$9	PATH environment variable used to look-up executables
\$0	Previous working directory
\$1	Present working directory
\$2	Array of function names in the execution call stack
\$3	Array containing source paths for elements in <code>FUNCTIONS</code> array. Can be used to get the script path.
\$4	Associative array containing all currently defined aliases
\$5	Array of matches from the last <code>grep</code> match
\$6	Bash version string
\$7	An array of 6 elements with Bash version information
\$8	Absolute path to the currently executing Bash shell itself (theoretically determined by <code>bash</code> itself on <code>argv[0]</code> and the value of <code>\$PATH</code> may be wrong in corner cases)
\$9	Bash subshell level
\$10	Real (most effective if different) User ID of the process running <code>bash</code>
\$11	Primary command line prompt; see Using the PS* variables
\$12	Secondary command line prompt (used for additional input)
\$13	Tertiary command line prompt (used for additional input)
\$14	Quaternary command line prompt (used in select loop)
\$15	A pseudo-random integer between 0 and 32767
\$16	Variable used by <code>read</code> by default when no variable is specified. Also used by <code>select</code> to return the user-supplied value
\$17	Array variable that holds the exit status values of each command in the most recently executed foreground pipeline.

Bash Notes for Professionals

Traducido por:

rortegag

100+ páginas

de consejos y trucos profesionales

Contenidos

Acerca de	1
Capítulo 1: Introducción a Bash	2
Sección 1.1: Hola Mundo	2
Sección 1.2: Hola Mundo con variables	3
Sección 1.3: Hola Mundo con entradas de usuario	4
Sección 1.4: Importancia de las citas en las cadenas de texto.....	5
Sección 1.5: Visualización de la información de los módulos Bash.....	5
Sección 1.6: Hola Mundo en modo “Debug”	6
Sección 1.7: Manejo de argumentos con nombre.....	7
Capítulo 2: Script shebang	8
Sección 2.1: Env shebang	8
Sección 2.2: Direct shebang	8
Sección 2.3: Otros shebangs.....	8
Capítulo 3: Navegar por los directorios.....	9
Sección 3.1: Directorios absolutos y relativos	9
Sección 3.2: Cambiar al último directorio.....	9
Sección 3.3: Cambiar al directorio personal	9
Sección 3.4: Cambiar al directorio del script	9
Capítulo 4: Listado de archivos	11
Sección 4.1: Listar archivos en un formato de listado largo	11
Sección 4.2: Lista de los diez archivos modificados más recientemente	12
Sección 4.3: Listar todos los archivos, incluidos los archivos de puntos.....	12
Sección 4.4: Listar archivos sin usar ‘ls’	12
Sección 4.5: Listar archivos	13
Sección 4.6: Listar archivos en formato de árbol.....	13
Sección 4.7: Listar archivos ordenados por tamaño	13
Capítulo 5: Utilizar cat.....	14
Sección 5.1: Concatenar archivos	14
Sección 5.2: Imprimir el contenido de un archivo	14
Sección 5.3: Escribir en un archivo.....	15
Sección 5.4: Mostrar caracteres no imprimibles	15
Sección 5.5: Lectura de la entrada estándar	15
Sección 5.6: Mostrar números de línea con salida.....	16
Sección 5.7: Concatenar archivos comprimidos con gzip	16
Capítulo 6: grep.....	17
Sección 6.1: Cómo buscar un patrón en un archivo	17
Capítulo 7: Aliasing.....	18
Sección 7.1: Anular un alias.....	18

Sección 7.2: Crear un alias	18
Sección 7.3: Eliminar un alias	18
Sección 7.4: Los BASH_ALIASES es un array interno de bash assoc	19
Sección 7.5: Expandir alias	19
Sección 7.6: Lista de alias	19
Capítulo 8: Trabajos y procesos.....	20
Sección 8.1: Gestión del trabajo	20
Sección 8.2: Comprobar qué proceso se ejecuta en un puerto específico.....	22
Sección 8.3: Desaprobar el trabajo de fondo.....	22
Sección 8.4: Listar los trabajos actuales	22
Sección 8.5: Buscar información sobre un proceso en ejecución	22
Sección 8.6: Listar todos los procesos.....	22
Capítulo 9: Redirección	24
Sección 9.1: Redireccionamiento de la salida estándar	24
Sección 9.2: Añadir vs Truncar	24
Sección 9.3: Redireccionamiento de STDOUT y STDERR	24
Sección 9.4: Uso de tuberías con nombre.....	25
Sección 9.5: Redirección a direcciones de red.....	27
Sección 9.6: Imprimir mensajes de error en stderr	27
Sección 9.7: Redirigir varios comandos al mismo archivo.....	27
Sección 9.8: Redireccionamiento de STDIN.....	28
Sección 9.9: Redireccionamiento de STDERR	28
Sección 9.10: Explicación de STDIN, STDOUT y STDERR	28
Capítulo 10: Estructuras de control	30
Sección 10.1: Ejecución condicional de listas de comandos.....	30
Sección 10.2: Declaración if.....	31
Sección 10.3: Recorrer un array en bucle.....	32
Sección 10.4: Uso del bucle for para iterar números en una lista	32
Sección 10.5: continue y break	33
Sección 10.6: Interrupción del bucle.....	33
Sección 10.7: Bucle while.....	34
Sección 10.8: Bucle for con sintaxis tipo C	34
Sección 10.9: Bucle until.....	35
Sección 10.10: Sentencia switch con case	35
Sección 10.11: Bucle for sin parámetro de lista de palabras.....	35
Capítulo 11: Comandos true, false y :	36
Sección 11.1: Bucle infinito.....	36
Sección 11.2: Función return	36
Sección 11.3: Código que siempre/nunca será ejecutado	36
Capítulo 12: Arrays	37
Sección 12.1: Asignación de arrays	37

Sección 12.2: Acceso a los elementos del array.....	37
Sección 12.3: Modificación de arrays.....	38
Sección 12.4: Iteración del array.....	39
Sección 12.5: Longitud del array.....	39
Sección 12.6: Arrays asociativos.....	40
Sección 12.7: Recorrer un array con bucles.....	40
Sección 12.8: Destruir, eliminar o anular un array.....	41
Sección 12.9: Array a partir de cadena de caracteres.....	41
Sección 12.10: Lista de índices inicializados.....	42
Sección 12.11: Lectura de un archivo completo en un array.....	42
Sección 12.12: Función de inserción de arrays.....	43
Capítulo 13: Arrays asociativos.....	44
Sección 13.1: Examinar los arrays assoc.....	44
Capítulo 14: Funciones.....	46
Sección 14.1: Funciones con argumentos.....	46
Sección 14.2: Función simple.....	47
Sección 14.3: Gestión de flags y parámetros opcionales.....	47
Sección 14.4: Imprimir la definición de la función.....	48
Sección 14.5: Una función que acepta parámetros con nombre.....	48
Sección 14.6: Valor de retorno de una función.....	48
Sección 14.7: El código de salida de una función es el código de salida de su último comando.....	49
Capítulo 15: Expansión de parámetros Bash.....	50
Sección 15.1: Modificación de las mayúsculas y minúsculas de los caracteres alfabéticos.....	50
Sección 15.2: Longitud del parámetro.....	50
Sección 15.3: Reemplazar patrón en cadena de caracteres.....	51
Sección 15.4: Subcadenas de texto y subarreglos.....	51
Sección 15.5: Borrar un patrón del principio de una cadena de caracteres.....	53
Sección 15.6: Indirección de parámetros.....	53
Sección 15.7: Expansión de parámetros y nombres de archivo.....	54
Sección 15.8: Sustitución de valores por defecto.....	54
Sección 15.9: Eliminar un patrón del final de una cadena de caracteres.....	55
Sección 15.10: Desprendimiento durante la expansión.....	55
Sección 15.11: Error si la variable está vacía o no está definida.....	56
Capítulo 16: Copiar (cp).....	57
Sección 16.1: Copiar un solo archivo.....	57
Sección 16.2: Copiar carpetas.....	57
Capítulo 17: Buscar (find).....	58
Sección 17.1: Buscar un archivo por nombre o extensión.....	58
Sección 17.2: Ejecutar comandos en un archivo encontrado.....	58
Sección 17.3: Búsqueda de archivos por hora de acceso/modificación.....	59
Sección 17.4: Búsqueda de archivos por tamaño.....	60

Sección 17.5: Filtrar la ruta	61
Sección 17.6: Búsqueda de archivos por tipo.....	61
Sección 17.7: Búsqueda de archivos por extensión.....	61
Capítulo 18: Usar sort	62
Sección 18.1: Ordenar la salida del comando.....	62
Sección 18.2: Hacer que el producto sea único.....	62
Sección 18.3: Ordenación numérica.....	62
Sección 18.4: Ordenar por claves.....	63
Capítulo 19: Búsqueda de fuentes	65
Sección 19.1: Obtener un archivo.....	65
Sección 19.2: Búsqueda de un entorno virtual	65
Capítulo 20: Here documents y here strings.....	66
Sección 20.1: Ejecutar comando con here document	66
Sección 20.2: Documentos con sangría.....	66
Sección 20.3: Crear un archivo	67
Sección 20.4: Here strings.....	67
Sección 20.5: Ejecutar varios comandos con sudo	67
Sección 20.6: Cadenas de caracteres con límite	68
Capítulo 21: Citar	69
Sección 21.1: Comillas dobles para la sustitución de variables y comandos.....	69
Sección 21.2: Diferencia entre comillas dobles y simples	69
Sección 21.3: Saltos de línea y caracteres de control.....	70
Sección 21.4: Citar texto literal.....	70
Capítulo 22: Expresiones condicionales	72
Sección 22.1: Pruebas de tipo de archivo	72
Sección 22.2: Comparación y correspondencia de cadenas de caracteres	72
Sección 22.3: Comprobar el estado de salida de un comando.....	74
Sección 22.4: Prueba de una línea.....	74
Sección 22.5: Comparación de archivos	74
Sección 22.6: Pruebas de acceso a archivos.....	75
Sección 22.7: Comparaciones numéricas.....	75
Capítulo 23: Scripting con parámetros	76
Sección 23.1: Análisis de múltiples parámetros	76
Sección 23.2: Análisis de argumentos mediante un bucle for	77
Sección 23.3: Script envolvente (Wrapper script)	77
Sección 23.4: Acceso a los parámetros.....	78
Sección 23.5: Dividir cadena de caracteres en un array en Bash.....	79
Capítulo 24: Sustituciones del historial de Bash.....	80
Sección 24.1: Referencia rápida	80
Sección 24.2: Repetir el comando anterior con sudo.....	81

Sección 24.3: Búsqueda en el historial de comandos por patrón	81
Sección 24.4: Cambiar al directorio recién creado con !#:N.....	81
Sección 24.5: Usando !\$.....	82
Sección 24.6: Repetir el comando anterior con una sustitución	82
Capítulo 25: Matemáticas	83
Sección 25.1: Matemáticas con dc	83
Sección 25.2: Matemáticas utilizando las capacidades de bash.....	83
Sección 25.3: Matemáticas con bc	84
Sección 25.4: Matemáticas con expr	84
Capítulo 26: Aritmética Bash	86
Sección 26.1: Aritmética simple con (()).....	86
Sección 26.2: Comando aritmético	86
Sección 26.3: Aritmética simple con expr	86
Capítulo 27: Ámbito	87
Sección 27.1: Ámbito dinámico en acción.....	87
Capítulo 28: Sustitución de procesos	88
Sección 28.1: Comparar dos archivos de la web.....	88
Sección 28.2: Alimentar un bucle while con la salida de un comando	88
Sección 28.3: Concatenar archivos	88
Sección 28.4: Transmitir un archivo a través de varios programas a la vez.....	88
Sección 28.5: Con el comando paste	89
Sección 28.6: Para evitar el uso de una subcápsula.....	89
Capítulo 29: Finalización programable	90
Sección 29.1: Finalización sencilla mediante la función	90
Sección 29.2: Completado sencillo de opciones y nombres de archivo.....	90
Capítulo 30: Personalizar PS1	91
Sección 30.1: Colorear y personalizar el indicador del terminal	91
Sección 30.2: Mostrar el nombre de la rama git en el prompt del terminal	92
Sección 30.3: Mostrar la hora en el prompt del terminal.....	92
Sección 30.4: Mostrar una rama git usando PROMPT_COMMAND.....	93
Sección 30.5: Cambiar el indicador PS1.....	93
Sección 30.6: Mostrar el estado de retorno del comando anterior y la hora.....	94
Capítulo 31: Expansión de llaves.....	96
Sección 31.1: Modificar la extensión del nombre de archivo.....	96
Sección 31.2: Crear directorios para agrupar archivos por mes y año	96
Sección 31.3: Crear una copia de seguridad de dotfiles.....	96
Sección 31.4: Utilizar incrementos.....	96
Sección 31.5: Uso de la expansión de llaves para crear listas	96
Sección 31.6: Crear varios directorios con subdirectorios.....	97
Capítulo 32: getopts : análisis sintáctico inteligente de parámetros de posición	98

Sección 32.1: pingnmap	98
Capítulo 33: Depurar	100
Sección 33.1: Comprobación de la sintaxis de un script con "-n"	100
Sección 33.2: Depuración con bashdb	100
Sección 33.3: Depuración de un script bash con "-x"	100
Capítulo 34: Concordancia de patrones y expresiones regulares	102
Sección 34.1: Obtener grupos capturados a partir de una coincidencia regex con una cadena de caracteres	102
Sección 34.2: Comportamiento cuando un glob no coincide con nada	102
Sección 34.3: Comprobar si una cadena de caracteres coincide con una expresión regular	103
Sección 34.4: Correspondencia Regex	103
Sección 34.5: El global *	103
Sección 34.6: El global **	104
Sección 34.7: El global ?	104
Sección 34.8: El global []	105
Sección 34.9: Búsqueda de archivos ocultos	105
Sección 34.10: Coincidencia de mayúsculas y minúsculas	106
Sección 34.11: Globalización ampliada	106
Capítulo 35: Cambiar de shell	108
Sección 35.1: Encontrar el shell actual	108
Sección 35.2: Lista de shells disponibles	108
Sección 35.3: Cambiar el Shell	108
Capítulo 36: Variables internas	109
Sección 36.1: Resumen de las variables internas de Bash	109
Sección 36.2: \$@	110
Sección 36.3: \$#	110
Sección 36.4: \$HISTSIZE	111
Sección 36.5: \$FUNCNAME	111
Sección 36.6: \$HOME	111
Sección 36.7: \$IFS	111
Sección 36.8: \$OLDPWD	112
Sección 36.9: \$PWD	112
Sección 36.10: \$1 \$2 \$3 etc.	112
Sección 36.11: \$*	112
Sección 36.12: \$!	113
Sección 36.13: \$?	113
Sección 36.14: \$\$	113
Sección 36.15: \$RANDOM	113
Sección 36.16: \$BASHPID	113
Sección 36.17: \$BASH_ENV	113
Sección 36.18: \$BASH_VERSION	114
Sección 36.19: \$BASH_VERSION	114

Sección 36.20: \$EDITOR	114
Sección 36.21: \$HOSTNAME	114
Sección 36.22: \$HOSTTYPE	114
Sección 36.23: \$MACHTYPE	114
Sección 36.24: \$OSTYPE	114
Sección 36.25: \$PATH	115
Sección 36.26: \$PPID	115
Sección 36.27: \$SECONDS	115
Sección 36.28: \$SHELLOPTS	115
Sección 36.29: \$_	115
Sección 36.30: \$GROUPS	116
Sección 36.31: \$LINENO	116
Sección 36.32: \$SHLVL	116
Sección 36.33: \$UID	117
Capítulo 37: Control del trabajo	118
Sección 37.1: Listar procesos en segundo plano	118
Sección 37.2: Traer a primer plano un proceso en segundo plano	118
Sección 37.3: Reiniciar proceso en segundo plano detenido	118
Sección 37.4: Ejecutar comando en segundo plano	118
Sección 37.5: Detener un proceso en primer plano	118
Capítulo 38: Declaración case	119
Sección 38.1: Declaración de caso simple	119
Sección 38.2: Declaración de case con caída	119
Sección 38.3: Pasar sólo si el patrón o patrones siguientes coinciden	119
Capítulo 39: ¿Leer un archivo (flujo de datos, variable) línea por línea (y/o campo por campo)?	121
Sección 39.1: Recorrer un fichero línea por línea	121
Sección 39.2: Recorrer la salida de un comando campo por campo	121
Sección 39.3: Leer líneas de un archivo en un array	121
Sección 39.4: Leer líneas de una cadena de caracteres en un array	121
Sección 39.5: Recorrer una cadena de caracteres línea por línea	122
Sección 39.6: Recorrer en bucle la salida de un comando línea por línea	122
Sección 39.7: Leer un fichero campo por campo	122
Sección 39.8: Leer una cadena de caracteres campo por campo	123
Sección 39.9: Leer campos de un archivo en un array	123
Sección 39.10: Leer campos de una cadena de caracteres en un array	123
Sección 39.11: Lee el archivo (/etc/passwd) línea por línea y campo por campo	123
Capítulo 40: Secuencia de ejecución del archivo	125
Sección 40.1: profile vs .bash_profile (y .bash_login)	125
Capítulo 41: Dividir archivos	126
Sección 41.1: Dividir un archivo	126

Capítulo 42: Transferencia de archivos mediante scp.....	127
Sección 42.1: scp transferir archivo.....	127
Sección 42.2: scp transferir varios archivos.....	127
Sección 42.3: Descarga de archivos mediante scp	127
Capítulo 43: Tuberías.....	128
Sección 43.1: Uso de &.....	128
Sección 43.2: Mostrar todos los procesos paginados.....	128
Sección 43.3: Modificar la salida continua de un comando	129
Capítulo 44: Gestión de la variable de entorno PATH	130
Sección 44.1: Añadir una ruta a la variable de entorno PATH.....	130
Sección 44.2: Eliminar una ruta de la variable de entorno PATH.....	130
Capítulo 45: Separación de palabras	132
Sección 45.1: ¿Qué, cuándo y por qué?	132
Sección 45.2: Efectos negativos de la división de palabras.....	132
Sección 45.3: Utilidad de la división de palabras.....	133
Sección 45.4: Dividir por cambios de separador.....	133
Sección 45.5: Dividir con IFS	134
Sección 45.6: IFS y división de palabras	134
Capítulo 46: Evitar la fecha utilizando printf	136
Sección 46.1: Obtener la fecha actual.....	136
Sección 46.2: Establecer la variable a la hora actual.....	136
Capítulo 47: Uso de "trap" para reaccionar ante señales y eventos del sistema	137
Sección 47.1: Introducción: limpiar archivos temporales.....	137
Sección 47.2: Capturar SIGINT o Ctrl+C.....	137
Sección 47.3: Acumular una lista de trabajos trampa para ejecutar a la salida	138
Sección 47.4: Terminar procesos hijos al salir.....	138
Sección 47.5: Reaccionar al cambiar el tamaño de la ventana del terminal.....	138
Capítulo 48: Cadena de mando y operaciones.....	140
Sección 48.1: Contar la ocurrencia de un patrón de texto.....	140
Sección 48.2: Transferir la salida cmd de root a un archivo de usuario.....	140
Sección 48.3: Encadenamiento lógico de comandos con && y 	140
Sección 48.4: Encadenamiento en serie de comandos con punto y coma.....	140
Sección 48.5: Encadenar comandos con 	140
Capítulo 49: Tipo de shells	142
Sección 49.1: Iniciar un shell interactivo.....	142
Sección 49.2: Detectar el tipo de Shell.....	142
Sección 49.3: Introducción a los archivos punto	142
Capítulo 50: Salida de scripts en color (multiplataforma).....	143
Sección 50.1: color-output.sh	143
Capítulo 51: co-procesos.....	144

Sección 51.1: Hola Mundo	144
Capítulo 52: Variables tipográficas.....	145
Sección 52.1: Declarar variables de tipado débil	145
Capítulo 53: Trabajos a horas concretas	146
Sección 53.1: Ejecutar el trabajo una vez a una hora determinada	146
Sección 53.2: Haciendo trabajos a horas especificadas repetidamente usando systemd.timer	146
Capítulo 54: Manejo del indicador del sistema	148
Sección 54.1: Uso de la variable de entorno PROMPT_COMMAND	148
Sección 54.2: Utilizar PS2	149
Sección 54.3: Utilizar PS3	149
Sección 54.4: Utilizar PS4	149
Sección 54.5: Utilizar PS1	150
Capítulo 55: El comando cut.....	151
Sección 55.1: Sólo un carácter delimitador	151
Sección 55.2: Los delimitadores repetidos se interpretan como campos vacíos.....	151
Sección 55.3: Sin citar	151
Sección 55.4: Extraer, no manipular	152
Capítulo 56: Bash en Windows 10	153
Sección 56.1: Léeme	153
Capítulo 57: Comando cut	154
Sección 57.1: Mostrar la primera columna de un fichero	154
Sección 57.2: Mostrar las columnas x a y de un fichero	154
Capítulo 58: Variables globales y locales.....	155
Sección 58.1: Variables globales	155
Sección 58.2: Variables locales	155
Sección 58.3: Mezcla de ambos	155
Capítulo 59: Scripts CGI.....	156
Sección 59.1: Método de solicitud: GET	156
Sección 59.2: Método de solicitud: POST /w JSON	158
Capítulo 60: Palabra clave Select.....	161
Sección 60.1: La palabra clave select se puede utilizar para obtener el argumento de entrada en un formato de menú	161
Capítulo 61: Cuándo utilizar eval	162
Sección 61.1: Uso de eval	162
Sección 61.2: Uso de eval con getopt.....	163
Capítulo 62: Trabajar en red con Bash	164
Sección 62.1: Comandos de red	164
Capítulo 63: Paralelo.....	166
Sección 63.1: Paralelizar tareas repetitivas en listas de archivos.....	166
Sección 63.2: Paralelizar STDIN.....	166

Capítulo 64: Descodificar URL	168
Sección 64.1: Ejemplo sencillo	168
Sección 64.2: Uso de printf para decodificar una cadena de caracteres	168
Capítulo 65: Patrones de diseño	169
Sección 65.1: Patrón de publicación/suscripción (Pub/Sub).....	169
Capítulo 66: Peligros	171
Sección 66.1: Espacios en blanco al asignar variables	171
Sección 66.2: Los comandos fallidos no detienen la ejecución del script	171
Sección 66.3: Falta la última línea de un archivo	171
Apéndice A: Atajos de teclado	173
Sección A.1: Atajos de edición.....	173
Sección A.2: Atajos de llamada	173
Sección A.3: Macros	173
Sección A.4: Fijaciones de teclas personalizadas	173
Sección A.5: Control del trabajo	174
Créditos	175

Acerca de

Este libro ha sido traducido por rortegag.com

Si desea descargar el libro original, puede descargarlo desde:

<https://goalkicker.com/BashBook/>

Si desea contribuir con una donación, hazlo desde:

<https://www.buymeacoffee.com/GoalKickerBooks>

Por favor, siéntase libre de compartir este PDF con cualquier persona de forma gratuita, la última versión de este libro se puede descargar desde:

<https://goalkicker.com/BashBook/>

Este libro Bash Apuntes para Profesionales está compilado a partir de la [Documentación de Stack Overflow](#), el contenido está escrito por la hermosa gente de Stack Overflow. El contenido del texto está liberado bajo Creative Commons BY-SA, ver los créditos al final de este libro quién contribuyó a los distintos capítulos. Las imágenes pueden ser copyright de sus respectivos propietarios a menos que se especifique lo contrario.

Este es un libro no oficial gratuito creado con fines educativos y no está afiliado con los grupo(s) o empresa(s) oficiales de Bash ni Stack Overflow. Todas las marcas comerciales y marcas registradas son propiedad de sus respectivos propietarios de la empresa.

No se garantiza que la información presentada en este libro sea correcta ni exacta. Utilícelo bajo su propia responsabilidad.

Envíe sus comentarios y correcciones a web@petercv.com

Capítulo 1: Introducción a Bash

Versión	Fecha de publicación
0.99	1989-06-08
1.01	1989-06-23
2.0	1996-12-31
2.02	1998-04-20
2.03	1999-02-19
2.04	2001-03-21
2.05b	2002-07-17
3.0	2004-08-03
3.1	2005-12-08
3.2	2006-10-11
4.0	2009-02-20
4.1	2009-12-31
4.2	2011-02-13
4.3	2014-02-26
4.4	2016-09-15

Sección 1.1: Hola Mundo

Shell interactivo

El shell Bash se utiliza habitualmente de forma **interactiva**: Te permite introducir y editar comandos, y luego los ejecuta cuando pulsas la tecla `Return`. Muchos sistemas operativos basados en Unix y similares a Unix utilizan Bash como shell por defecto (especialmente Linux y macOS). El terminal entra automáticamente en un proceso interactivo del shell Bash al iniciarse.

Emita `Hello World` tecleando lo siguiente:

```
echo "Hello World"
#> Hello World # Ejemplo de salida
```

Notas

- Puedes cambiar el shell simplemente escribiendo el nombre del shell en el terminal. Por ejemplo: `sh`, `bash`, etc.
- `echo` es un comando de Bash que escribe los argumentos que recibe en la salida estándar. Por defecto, añade una nueva línea a la salida.

Shell no interactivo

El shell Bash también puede ejecutarse de forma **no interactiva** desde un script, haciendo que el shell no requiera interacción humana. El comportamiento interactivo y el comportamiento de script deben ser idénticos - una importante consideración de diseño de Unix V7 Bourne shell y transitivamente Bash. Por lo tanto, todo lo que se puede hacer en la línea de comandos se puede poner en un archivo de secuencia de comandos para su reutilización.

Siga estos pasos para crear un script `Hello World`:

1. Crea un nuevo archivo llamado `hello-world.sh`
`touch hello-world.sh`
2. Haga ejecutable el script ejecutando `chmod +x hello-world.sh`

3. Añade este código:

```
#!/bin/bash
echo "Hello World"
```

Línea 1: La primera línea del script debe comenzar con la secuencia de caracteres `#!`, conocida como *shebang*¹. El shebang ordena al sistema operativo que ejecute `/bin/bash`, el intérprete de comandos Bash, pasándole la ruta del script como argumento.

Por ejemplo, `/bin/bash hello-world.sh`

Línea 2: Utiliza el comando `echo` para escribir `Hello World` en la salida estándar.

4. Ejecute el script `hello-world.sh` desde la línea de comandos utilizando una de las siguientes opciones:
 - `./hello-world.sh` - el más utilizado y recomendado
 - `/bin/bash hello-world.sh`
 - `bash hello-world.sh` - suponiendo que `/bin` esté en `$PATH`
 - `sh hello-world.sh`

Para un uso real en producción, debería omitir la extensión `.sh` (que de todos modos es engañosa, ya que se trata de un script Bash, no un script sh) y quizás mover el archivo a un directorio dentro de su `PATH` para que esté disponible independientemente de su directorio de trabajo actual, al igual que un comando del sistema como `cat` o `ls`.

Los errores más comunes son:

1. Olvidar aplicar el permiso de ejecución en el archivo, es decir, `chmod +x hello-world.sh`, lo que resulta en la salida de `./hello-world.sh: Permission denied`.
2. Editar el script en Windows, lo que produce caracteres de final de línea incorrectos que Bash no puede manejar.

Un síntoma común es : `command not found` donde el retorno de carro ha forzado el cursor al principio de línea, sobrescribiendo el texto antes de los dos puntos en el mensaje de error.

El script se puede arreglar utilizando el programa `dos2unix`.

Un ejemplo de uso: `dos2unix hello-world.sh`

dos2unix edita el archivo en línea.

3. Usar `sh ./hello-world.sh`, sin darse cuenta de que `bash` y `sh` son shells distintos con características distintas (aunque como Bash es retrocompatible, el error contrario es inofensivo).

De todos modos, confiar simplemente en la línea shebang del script es mucho mejor que escribir explícitamente `bash` o `sh` (o `python` o `perl` o `awk` o `ruby` o...) antes del nombre de archivo de cada script.

Una línea shebang común para hacer tu script más portable es usar `#!/usr/bin/env bash` en lugar de codificar una ruta a Bash. De esta manera, `/usr/bin/env` tiene que existir, pero más allá de ese punto, `bash` sólo necesita estar en tu `PATH`. En muchos sistemas, `/bin/bash` no existe, y debes usar `/usr/local/bin/bash` o alguna otra ruta absoluta; este cambio evita tener que averiguar los detalles de eso.

¹ También llamado *sha-bang*, *hashbang*, *pound-bang*, *hash-pling*.

Sección 1.2: Hola Mundo con variables

Crea un nuevo archivo llamado `hello.sh` con el siguiente contenido y dale permisos de ejecución con `chmod +x hello.sh`.

Ejecutar a través de: `./hello.sh`

```
#!/usr/bin/env bash
```

```
# Note that spaces cannot be used around the `=` assignment operator
whom_variable="World"
# Use printf to safely output the data
printf "Hello, %s\n" "$whom_variable"
#> Hello, World
```

Esto imprimirá `Hello, World` en la salida estándar cuando se ejecute.

Para decirle a `bash` dónde está el script necesitas ser muy específico, señalándole el directorio que lo contiene, normalmente con `./` si es tu directorio de trabajo, donde `.` es un alias del directorio actual. Si no especifica el directorio, `bash` intenta localizar el script en uno de los directorios contenidos en la variable de entorno `$PATH`.

El siguiente código acepta un argumento `$1`, que es el primer argumento de la línea de comandos, y lo muestra en una cadena formateada, a continuación de `Hello, .`

Ejecutar a través de: `./hello.sh World`

```
#!/usr/bin/env bash
printf "Hello, %s\n" "$1"
#> Hello, World
```

Es importante tener en cuenta que `$1` tiene que ir entre comillas dobles, no simples. `"$1"` se expande al primer argumento de la línea de comandos, como se desee, mientras que `"$1"` se evalúa a la cadena literal `$1`.

Nota de seguridad:

Lea [**Implicaciones de seguridad de olvidar entrecomillar una variable en shells bash**](#) para entender la importancia de colocar el texto de la variable entre comillas dobles.

Sección 1.3: Hola Mundo con entradas de usuario

Lo siguiente pedirá al usuario que introduzca datos, y luego los almacenará como una cadena de texto (texto) en una variable. La variable se utiliza entonces para dar un mensaje al usuario.

```
#!/usr/bin/env bash
echo "Who are you?"
read name
echo "Hello, $name."
```

El comando `leer` aquí lee una línea de datos de la entrada estándar en el `name` de la variable. A continuación, se hace referencia a ella mediante `$name` y se imprime en la salida estándar mediante `echo`.

Ejemplo de salida:

```
$ ./hello_world.sh
Who are you?
Matt
Hello, Matt.
```

Aquí el usuario introdujo el nombre "Matt", y este código se utilizó para decir `Hello, Matt..`

Y si desea añadir algo al valor de la variable mientras la imprime, utilice llaves alrededor del nombre de la variable como se muestra en el siguiente ejemplo:

```
#!/usr/bin/env bash
echo "What are you doing?"
read action
echo "You are ${action}ing."
```


Ejemplo de salida:

```
$ ./hello_world.sh
What are you doing?
Sleep
You are Sleeping.
```

Aquí, cuando el usuario introduce una acción, se añade "ing" a esa acción mientras se imprime.

Sección 1.4: Importancia de las citas en las cadenas de texto

Las comillas son importantes para la expansión de cadenas de texto en bash. Con ellas, puedes controlar cómo bash analiza y expande tus cadenas de texto.

Hay dos tipos de citación:

Débil: *utiliza comillas dobles: "*

Fuerte: *utiliza comillas simples: '*

Si quieres golpear para ampliar tu argumento, puedes usar **citas débiles**:

```
#!/usr/bin/env bash
world="World"
echo "Hello $world"
#> Hello World
```

Si no quieres golpear para ampliar tu argumento, puedes utilizar las **citas fuertes**:

```
#!/usr/bin/env bash
world="World"
echo 'Hello $world'
#> Hello $world
```

También puede utilizar escape para evitar la expansión:

```
#!/usr/bin/env bash
world="World"
echo "Hello \$world"
#> Hello $world
```

Si desea información más detallada, aparte de los detalles para principiantes, puede seguir leyéndola aquí.

Sección 1.5: Visualización de la información de los módulos Bash

help <command>

Esto mostrará la página de ayuda (manual) de Bash para el built-in especificado.

Por ejemplo, `help unset` mostrará:

```
unset: unset [-f] [-v] [-n] [name ...]
Unset values and attributes of shell variables and functions.

For each NAME, remove the corresponding variable or function.

Options:
-f treat each NAME as a shell function
-v treat each NAME as a shell variable
-n treat each NAME as a name reference and unset the variable itself
  rather than the variable it references

Without options, unset first tries to unset a variable, and if that fails,
tries to unset a function.

Some variables cannot be unset; also see `readonly'.

Exit Status:
Returns success unless an invalid option is given or a NAME is read-only.
```

Para ver una lista de todos los módulos integrados con una breve descripción, utilice

`help -d`

Sección 1.6: Hola Mundo en modo “Debug”

```
$ cat hello.sh
#!/bin/bash
echo "Hello World"
$ bash -x hello.sh
+ echo Hello World
Hello World
```

El argumento `-x` le permite recorrer cada línea del script. Un buen ejemplo es éste:

```
$ cat hello.sh
#!/bin/bash
echo "Hello World\n"
adding_string_to_number="s"
v=$(expr 5 + $adding_string_to_number)

$ ./hello.sh
Hello World
```

expr: non-integer argument

El error indicado arriba no es suficiente para rastrear el script; sin embargo, usar la siguiente forma le da una mejor idea de dónde buscar el error en el script.

```
$ bash -x hello.sh
+ echo Hello World\n
Hello World

+ adding_string_to_number=s
+ expr 5 + s
expr: non-integer argument
+ v=
```

Sección 1.7: Manejo de argumentos con nombre

```
#!/bin/bash

deploy=false
uglify=false

while (( $# > 1 )); do case $1 in
    --deploy) deploy="$2";;
    --uglify) uglify="$2";;
    *) break;
esac; shift 2
done

$deploy && echo "will deploy... deploy = $deploy"
$uglify && echo "will uglify... uglify = $uglify"

# cómo ejecutar
# chmod +x script.sh
# ./script.sh --deploy true --uglify false
```

Capítulo 2: Script shebang

Sección 2.1: Env shebang

Para ejecutar un archivo de script con el ejecutable `bash` que se encuentra en la variable de entorno `PATH` mediante el ejecutable `env`, la **primera línea** de un archivo de script debe indicar la ruta absoluta al ejecutable `env` con el argumento `bash`:

```
#!/usr/bin/env bash
```

La ruta `env` en el shebang se resuelve y se utiliza sólo si se lanza directamente un script como este:

```
script.sh
```

El script debe tener permisos de ejecución.

El shebang se ignora cuando se indica explícitamente un intérprete `bash` para ejecutar un script:

```
bash script.sh
```

Sección 2.2: Direct shebang

Para ejecutar un archivo de script con el intérprete `bash`, la **primera línea** de un archivo de script debe indicar la ruta absoluta al ejecutable `bash` a utilizar:

```
#!/bin/bash
```

La ruta `bash` en el shebang se resuelve y se utiliza sólo si se lanza directamente un script como éste:

```
./script.sh
```

El script debe tener permisos de ejecución.

El shebang se ignora cuando se indica explícitamente un intérprete `bash` para ejecutar un script:

```
bash script.sh
```

Sección 2.3: Otros shebangs

Existen dos tipos de programas que el kernel conoce. Un programa binario se identifica por su cabecera ELF (**E**xtenable **L**oadable **F**ormat), que normalmente es producida por un compilador. El segundo son scripts de cualquier tipo.

Si un fichero comienza en la primera línea con la secuencia `#!` entonces la siguiente cadena tiene que ser una ruta de un intérprete. Si el kernel lee esta línea, llama al intérprete nombrado por esta ruta y le da todas las palabras siguientes en esta línea como argumentos al intérprete. Si no hay ningún fichero llamado "something" o "wrong":

```
#!/bin/bash something wrong
echo "This line never gets printed"
```

`bash` intenta ejecutar su argumento "something wrong" que no existe. También se añade el nombre del archivo de script. Para ver esto claramente usa un `echo` shebang:

```
#!/bin/echo something wrong
# y ahora llama a este script llamado «thisscript» así:
# thisscript one two
# la salida será:
something wrong ./thisscript one two
```

Algunos programas como `awk` utilizan esta técnica para ejecutar scripts más largos que residen en un archivo de disco.

Capítulo 3: Navegar por los directorios

Sección 3.1: Directorios absolutos y relativos

Para cambiar a un directorio absolutamente especificado, utilice el nombre completo, empezando por una barra /, así:

```
cd /home/username/project/abc
```

Si desea cambiar a un directorio cercano al actual, puede especificar una ubicación relativa. Por ejemplo, si ya se encuentra en `/home/username/project`, puede introducir así el subdirectorio `abc`:

```
cd abc
```

Si desea ir al directorio situado por encima del directorio actual, puede utilizar el alias `..`. Por ejemplo, si estuvieras en `/home/username/Project/abc` y quisieras ir a `/home/username/project`, entonces harías lo siguiente:

```
cd ..
```

Esto también puede denominarse “subir” un directorio.

Sección 3.2: Cambiar al último directorio

Para el shell actual, esto le lleva al directorio anterior en el que se encontraba, sin importar dónde estuviera.

```
cd -
```

Si lo haces varias veces, estarás en el directorio actual o en el anterior.

Sección 3.3: Cambiar al directorio personal

El directorio por defecto es el directorio personal (`$HOME`, normalmente `/home/username`), por lo que `cd` sin ningún directorio te lleva allí

```
cd
```

O podría ser más explícito:

```
cd $HOME
```

Un acceso directo para el directorio personal es `~`, por lo que también podría utilizarse.

```
cd ~
```

Sección 3.4: Cambiar al directorio del script

En general, existen dos tipos de **scripts** Bash:

1. Herramientas del sistema que operan desde el directorio de trabajo actual
2. Herramientas de proyecto que modifican archivos en relación con su propio lugar en el sistema de archivos.

Para el segundo tipo de scripts, es útil cambiar al directorio donde se almacena el script. Esto se puede hacer con el siguiente comando:

```
cd "$(dirname "$(readlink -f "$0")")"
```

Este comando ejecuta 3 comandos:

1. `readlink -f "$0"` determina la ruta al script actual (\$0)
2. `dirname` convierte la ruta al script en la ruta a su directorio
3. `cd` cambia el directorio de trabajo actual al directorio que recibe de `dirname`

Capítulo 4: Listado de archivos

Opción	Descripción
-a, --all	Lista todas las entradas, incluidas las que empiezan por punto
-A, --almost-all	Lista todas las entradas excluyendo <code>.</code> y <code>..</code>
-c	Ordenar archivos por tiempo de modificación
-d, --directory	Listar entradas de directorio
-h, --human-readable	Mostrar los tamaños en formato legible (por ejemplo, K, M)
-H	Igual que arriba sólo que con potencias de 1000 en lugar de 1024
-l	Mostrar contenidos en formato de lista larga
-o	Formato de lista larga sin información de grupo
-r, --reverse	Mostrar el contenido en orden inverso
-s, --size	Imprimir el tamaño de cada archivo en bloques
-S	Ordenar por tamaño de archivo
--sort=WORD	Ordenar el contenido por una palabra. (por ejemplo, tamaño, versión, estado)
-t	Ordenar por hora de modificación
-u	Ordenar por última hora de acceso
-v	Ordenar por versión
-1	Listar un archivo por línea

Sección 4.1: Listar archivos en un formato de listado largo

La opción `-l` del comando `ls` imprime el contenido de un directorio especificado en un formato de listado largo. Si no se especifica ningún directorio, se muestra por defecto el contenido del directorio actual.

```
ls -l /etc
```

Ejemplo de salida:

```
total 1204
drwxr-xr-x 3 root root 4096 Apr 21 03:44 acpi
-rw-r--r-- 1 root root 3028 Apr 21 03:38 adduser.conf
drwxr-xr-x 2 root root 4096 Jun 11 20:42 alternatives
...
```

La salida muestra primero el `total`, que indica el tamaño total en **bloques** de todos los ficheros del directorio listado. A continuación, muestra ocho columnas de información para cada fichero del directorio listado. A continuación, se muestran los detalles de cada columna de la salida:

Nº de columna	Ejemplo	Descripción
1.1	d	Tipo de archivo (véase la tabla siguiente)
1.2	rw-r-xr-x	Cadena de permisos
2	3	Número de enlaces duros
3	root	Nombre del propietario
4	root	Grupo del propietario
5	4096	Tamaño del archivo en bytes
6	Apr 21 03:44	Tiempo de modificación
7	acpi	Nombre del archivo

Tipo de archivo

El tipo de archivo puede ser uno de los siguientes caracteres.

Carácter	Tipo de archivo
-	Fichero normal
B	Archivo especial de bloque
c	Fichero especial de caracteres
C	Fichero de alto rendimiento («datos contiguos»)
d	Directorio
D	Door (archivo IPC especial sólo en Solaris 2.5+)
l	Enlace simbólico
M	Archivo fuera de línea («migrado») (Cray DMF)
n	Archivo especial de red (HP-UX)
p	FIFO (tubería con nombre)
P	Puerto (archivo de sistema especial sólo en Solaris 10+)
s	Socket
?	Otro tipo de archivo

Sección 4.2: Lista de los diez archivos modificados más recientemente

Lo siguiente listará hasta diez de los archivos modificados más recientemente en el directorio actual, utilizando un formato de listado largo (`-l`) y ordenado por tiempo (`-t`).

```
ls -lt | head
```

Sección 4.3: Listar todos los archivos, incluidos los archivos de puntos

Un **dotfile** es un fichero cuyos nombres empiezan por `.`. Normalmente son ocultos por `ls` y no listados a menos que se solicite.

Por ejemplo, la siguiente salida de `ls`:

```
$ ls
bin pki
```

Las opciones `-a` o `--all` listan todos los archivos, incluidos los dotfiles.

```
$ ls -a
.      .ansible      .bash_logout  .bashrc      .lessht      .puppetlabs   .viminfo
..     .bash_history  .bash_profile bin           pki           .ssh
```

La opción `-A` o `--almost-all` listará todos los ficheros, incluidos los dotfiles, pero no listará los `.` y `..` implícitos. Tenga en cuenta que `.` es el directorio actual y `..` es el directorio padre.

```
$ ls -A
.ansible      .bash_logout  .bashrc      .lessht      .puppetlabs   .viminfo
.bash_history .bash_profile bin           pki           .ssh
```

Sección 4.4: Listar archivos sin usar 'ls'

Utilice las funciones de [expansión de nombres de archivo](#) y [expansión de llaves](#) del intérprete de comandos Bash para obtener los nombres de archivo:

```
# mostrar los archivos y directorios que se encuentran en el directorio actual
printf "%s\n" *
# mostrar sólo los directorios del directorio actual
printf "%s\n" */
# mostrar sólo (algunos) archivos de imagen
printf "%s\n" *.{gif,jpg,png}
```

Para capturar una lista de archivos en una variable para su procesamiento, suele ser una buena práctica utilizar un `array bash`:

```
files=( * )

# iterar sobre ellos
for file in "${files[@]}; do
    echo "$file"
done
```

Sección 4.5: Listar archivos

El comando `ls` lista el contenido de un directorio especificado, **excluyendo** dotfiles. Si no se especifica ningún directorio, se muestra por defecto el contenido del directorio actual.

Los ficheros listados se ordenan alfabéticamente, por defecto, y se alinean en columnas si no caben en una línea.

```
$ ls
apt  configs  Documents  Fonts      Music      Programming  Templates      workspace
bin  Desktop  eclipse    git        Pictures   Public       Videos
```

Sección 4.6: Listar archivos en formato de árbol

El comando `tree` muestra el contenido de un directorio especificado en formato de árbol. Si no se especifica ningún directorio, se muestra por defecto el contenido del directorio actual.

Ejemplo de salida:

```
$ tree /tmp
/tmp
├── 5037
├── adb.log
└── evince-20965
    └── image.FPWTJY.png
```

Utilice la opción `-L` del comando `tree` para limitar la profundidad de visualización y la opción `-d` para listar sólo los directorios.

Ejemplo de salida:

```
$ tree -L 1 -d /tmp
/tmp
└── evince-20965
```

Sección 4.7: Listar archivos ordenados por tamaño

La opción `-S` del comando `ls` ordena los archivos en orden descendente según su tamaño.

```
$ ls -l -S ./Fruits
total 444
-rw-rw-rw-  1 root root  295303   Jul 28 19:19 apples.jpg
-rw-rw-rw-  1 root root  102283   Jul 28 19:19 kiwis.jpg
-rw-rw-rw-  1 root root   50197   Jul 28 19:19 bananas.jpg
```

Si se utiliza con la opción `-r`, el orden se invierte.

```
$ ls -l -S -r ./Fruits
total 444
-rw-rw-rw-  1 root root   50197   Jul 28 19:19 bananas.jpg
-rw-rw-rw-  1 root root  102283   Jul 28 19:19 kiwis.jpg
-rw-rw-rw-  1 root root  295303   Jul 28 19:19 apples.jpg
```

Capítulo 5: Utilizar cat

Opción	Detalles
-n	Imprimir números de línea
-v	Mostrar los caracteres no imprimibles utilizando la notación <code>^</code> y <code>M-</code> excepto <code>LFD</code> y <code>TAB</code>
-T	Mostrar caracteres <code>TAB</code> como <code>^I</code>
-E	Mostrar caracteres de salto de línea (LF) como <code>\$</code>
-e	Igual que <code>-vE</code>
-b	Número de líneas de salida no vacías, anula <code>-n</code>
-A	equivalente a <code>-vET</code>
-s	suprimir las líneas de salida vacías repetidas, <code>s</code> se refiere a squeeze

Sección 5.1: Concatenar archivos

Este es el objetivo principal de `cat`.

```
cat file1 file2 file3 > file_all
```

`cat` también se puede utilizar de forma similar para concatenar archivos como parte de una tubería, p.ej.

```
cat file1 file2 file3 | grep foo
```

Sección 5.2: Imprimir el contenido de un archivo

```
cat file.txt
```

imprimirá el contenido de un archivo.

Si el fichero contiene caracteres no ASCII, puede mostrarlos simbólicamente con `cat -v`. Esto puede resultar muy útil en situaciones en las que, de otro modo, los caracteres de control serían invisibles.

```
cat -v unicode.txt
```

Sin embargo, a menudo, para un uso interactivo, es mejor utilizar un paginador interactivo como `less` o `more`. (`less` es mucho más potente que `more` y se aconseja utilizar `less` más a menudo que `more`).

```
less file.txt
```

Pasar el contenido de un archivo como entrada a un comando. Un enfoque que suele considerarse mejor (`UUOC`) es utilizar la redirección.

```
tr A-Z a-z <file.txt # como alternativa a cat file.txt | tr A-Z a-z
```

En caso de que sea necesario listar el contenido hacia atrás desde su final, se puede utilizar el comando `tac`:

```
tac file.txt
```

Si desea imprimir el contenido con números de línea, utilice `-n` con `cat`:

```
cat -n file.txt
```

Para mostrar el contenido de un archivo de forma completamente inequívoca, byte a byte, la solución estándar es un volcado hexadecimal. Esto es bueno para fragmentos muy breves de un archivo, como cuando no se conoce la codificación exacta. La utilidad estándar de volcado hexadecimal es `od -cH`, aunque la representación es un poco engorrosa; sustitutos comunes incluyen `xxd` y `hexdump`.

```
$ printf 'Hello world' | xxd
0000000: 48c3 ab6c 6cc3 b620 77c3 b672 6c64 H..ll.. w..rld
```

Sección 5.3: Escribir en un archivo

```
cat >file
```

Le permitirá escribir el texto en el terminal que se guardará en un archivo llamado *file*.

```
cat >>file
```

hará lo mismo, excepto que añadirá el texto al final del archivo.

N.B: `Ctrl+D` para terminar de escribir texto en el terminal (Linux)

Un documento here puede utilizarse para insertar el contenido de un archivo en una línea de comandos o un script:

```
cat <<END >file
Hello, World.
END
```

El token después del símbolo de redirección `<<` es una cadena de texto arbitraria que debe aparecer sola en una línea (sin espacios en blanco iniciales ni finales) para indicar el final del documento. Puede añadir comillas para evitar que el shell realice la sustitución de comandos y la interpolación de variables:

```
cat <<'fnord'
Nothing in `here` will be $changed
fnord
```

(Sin las comillas, aquí se ejecutaría como un comando, y `$changed` se sustituiría por el valor de la variable cambiada `--` o nada, si fuera indefinida).

Sección 5.4: Mostrar caracteres no imprimibles

Esto es útil para ver si hay caracteres no imprimibles, o caracteres no ASCII.

Por ejemplo, si ha copiado y pegado el código de la web, es posible que tenga comillas como `"` en lugar de `"` estándar.

```
$ cat -v file.txt
$ cat -vE file.txt # Útil para detectar espacios finales.
```

Por ejemplo

```
$ echo '"' | cat -vE # echo | será reemplazado por el archivo actual.
M-bM-^@M-^] $
```

También puede utilizar `cat -A` (A de Todos) que es equivalente a `cat -vET`. Mostrará caracteres TAB (mostrados como `^I`), caracteres no imprimibles y el final de cada línea:

```
$ echo '"`' | cat -A
M-bM-^@M-^] ^I`$
```

Sección 5.5: Lectura de la entrada estándar

```
cat < file.txt
```

La salida es la misma que `cat file.txt`, pero lee el contenido del fichero desde la entrada estándar en lugar de directamente desde el fichero.

```
printf "first line\nSecond line\n" | cat -n
```

El comando `echo` antes de `|` da salida a dos líneas. El comando `cat` actúa sobre la salida para añadir números de línea.

Sección 5.6: Mostrar números de línea con salida

Utilice el indicador `--number` para imprimir los números de línea antes de cada línea. Alternativamente, `-n` hace lo mismo.

```
$ cat --number file
1 line 1
2 line 2
3
4 line 4
5 line 5
```

Para omitir las líneas vacías al contar las líneas, utilice el parámetro `--number-nonblank`, o simplemente `-b`.

```
$ cat -b file
1 line 1
2 line 2
3 line 4
4 line 5
```

Sección 5.7: Concatenar archivos comprimidos con gzip

Los archivos comprimidos con `gzip` pueden concatenarse directamente en archivos gzip más grandes.

```
cat file1.gz file2.gz file3.gz > combined.gz
```

Esta es una propiedad de `gzip` que es menos eficiente que concatenar los archivos de entrada y gzippear el resultado:

```
cat file1 file2 file3 | gzip > combined.gz
```

Una demostración completa:

```
echo 'Hello world!' > hello.txt
echo 'Howdy world!' > howdy.txt
gzip hello.txt
gzip howdy.txt
cat hello.txt.gz howdy.txt.gz > greetings.txt.gz
gunzip greetings.txt.gz
cat greetings.txt
```

Lo que resulta en

```
Hello world!
Howdy world!
```

Observe que `greetings.txt.gz` es un **único archivo** y se descomprime como el **único archivo** `greeting.txt`. Esto contrasta con `tar -czf hello.txt howdy.txt > greetings.tar.gz`, que mantiene los archivos separados dentro del tarball.

Capítulo 6: grep

Sección 6.1: Cómo buscar un patrón en un archivo

Para encontrar la palabra **foo** en el archivo *bar*:

```
grep foo ~/Desktop/bar
```

Para encontrar todas las líneas que **no** contienen foo en el archivo *bar*:

```
grep -v foo ~/Desktop/bar
```

Para utilizar encontrar todas las palabras que contienen foo al final (Expansión Wildcard):

```
grep "*foo" ~/Desktop/bar
```

Capítulo 7: Aliasing

Los alias de shell son una forma sencilla de crear nuevos comandos o de envolver comandos existentes con código propio. En cierto modo se solapan con las funciones del shell, que sin embargo son más versátiles y, por lo tanto, a menudo deberían preferirse.

Sección 7.1: Anular un alias

A veces es posible que desee pasar por alto un alias temporalmente, sin desactivarlo. Para trabajar con un ejemplo concreto, considere este alias:

```
alias ls='ls --color=auto'
```

Y digamos que quieres usar el comando `ls` sin desactivar el alias. Tienes varias opciones:

- Utilice el `command` integrado: `command ls`
- Utilice la ruta completa del comando `/bin/ls`
- Añada una `\` en cualquier parte del nombre del comando, por ejemplo: `\ls`, o `l\ls`
- Cita el comando: `"ls"` o `'ls'`

Sección 7.2: Crear un alias

```
alias word='command'
```

Al invocar `word` se ejecutará el `command`. Cualquier argumento suministrado al alias simplemente se añade al objetivo del alias:

```
alias myAlias='some command --with --options'
myAlias foo bar baz
```

A continuación, se ejecutará el intérprete de comandos:

```
some command --with --options foo bar baz
```

Para incluir varios comandos en el mismo alias, puede encadenarlos con `&&`. Por ejemplo:

```
alias print_things='echo "foo" && echo "bar" && echo "baz"'
```

Sección 7.3: Eliminar un alias

Para eliminar un alias existente, utilice:

```
unalias {alias_name}
```

Por ejemplo:

```
# crear un alias
$ alias now='date'

# vista previa del alias
$ now
Thu Jul 21 17:11:25 CEST 2016

# eliminar el alias
$ unalias now

# comprobar si se ha eliminado
$ now
-bash: now: command not found
```


Sección 7.4: Los BASH_ALIASES es un array interno de bash assoc

Los alias son atajos de comandos con nombre que se pueden definir y utilizar en instancias interactivas de bash. Se guardan en un array asociativa llamada BASH_ALIASES. Para utilizar este var en un script, debe ser ejecutado dentro de un shell interactivo

```
#!/bin/bash -li
# ¡nota el -li de arriba! -l hace que se comporte como un intérprete
# de comandos de inicio de sesión
# -i hace que se comporte como un intérprete de comandos interactivo
#
# shopt -s expand_aliases no funcionará en la mayoría de los casos
echo There are ${#BASH_ALIASES[*]} aliases defined.

for ali in "${!BASH_ALIASES[@]}"; do
    printf "alias: %-10s triggers: %s\n" "$ali" "${BASH_ALIASES[$ali]}"
done
```

Sección 7.5: Expandir alias

Asumiendo que `bar` es un alias para `someCommand -flag1`.

Escriba `bar` en la línea de comandos y pulse `Ctrl + alt + e`.

obtendrá `someCommand -flag1` donde estaba `bar`.

Sección 7.6: Lista de alias

```
alias -p
```

listará todos los alias actuales.

Capítulo 8: Trabajos y procesos

Sección 8.1: Gestión del trabajo

Crear trabajo

Para crear una tarea, basta con añadir un `&` después de la orden:

```
$ sleep 10 &
[1] 20024
```

También puede convertir un proceso en ejecución en un trabajo pulsando `Ctrl + z`:

```
$ sleep 10
^Z
[1]+  Stopped sleep 10
```

Fondo y primer plano de un proceso

Para traer el Proceso al primer plano, se utiliza el comando `fg` junto con `%`.

```
$ sleep 10 &
[1] 20024
```

```
$ fg %1
sleep 10
```

Ahora puedes interactuar con el proceso. Para volver a ponerlo en segundo plano puedes utilizar el comando `bg`. Debido a la sesión de terminal ocupada, es necesario detener primero el proceso pulsando `Ctrl + z`.

```
$ sleep 10
^Z
[1]+  Stopped sleep 10
```

```
$ bg %1
[1]+ sleep 10 &
```

Debido a la pereza de algunos Programadores, todos estos comandos también funcionan con un solo `%` si sólo hay un proceso, o para el primer proceso de la lista. Por ejemplo:

```
$ sleep 10 &
[1] 20024
```

```
$ fg % # para traer un proceso a primer plano 'fg %' también funciona.
sleep 10
```

o simplemente

```
$ % # la pereza no conoce fronteras, '%' también trabaja.
sleep 10
```

Además, basta con escribir `fg` o `bg` sin ningún argumento para realizar el último trabajo:

```
$ sleep 20 &
$ sleep 10 &
$ fg
sleep 10
^C
$ fg
sleep 20
```

Eliminación de trabajos en curso

```
$ sleep 10 &  
[1] 20024  
$ kill %1  
[1]+ Terminated sleep 10
```

El proceso `sleep` se ejecuta en segundo plano con el identificador de proceso (pid) 20024 y el número de trabajo 1. Para hacer referencia al proceso, puede utilizar el pid o el número de tarea. Si utiliza el número de tarea, debe anteponerle %. La señal de `kill` enviada por defecto por `kill` es `SIGTERM`, que permite al proceso de destino salir con gracia.

A continuación, se muestran algunas señales `kill` comunes. Para ver una lista completa, ejecute `kill -l`.

Nombre de la señal	Valor de la señal	Efecto
SIGHUP	1	Cuelgue
SIGINT	2	Interrupción por teclado
SIGKILL	9	Señal de muerte
SIGTERM	15	Señal de terminación

Iniciar y matar procesos específicos

Probablemente la forma más fácil de matar un proceso en ejecución es seleccionándolo a través del nombre del proceso como en el siguiente ejemplo utilizando el comando `pkill` como

```
pkill -f test.py
```

(o) una forma más segura usando `pgrep` para buscar el process-id real

```
kill $(pgrep -f 'python test.py')
```

El mismo resultado puede obtenerse utilizando `grep` sobre `ps -ef | grep nombre_del_proceso` y luego matando el proceso asociado con el pid (id de proceso) resultante. Seleccionar un proceso usando su nombre es conveniente en un entorno de pruebas, pero puede ser realmente peligroso cuando el script se usa en producción: es virtualmente imposible estar seguro de que el nombre coincidirá con el proceso que realmente se quiere matar. En esos casos, el siguiente enfoque es realmente mucho más seguro.

Inicie el script que finalmente matará con el siguiente enfoque. Supongamos que el comando que desea ejecutar y eventualmente matar es `python test.py`.

```
#!/bin/bash  
if [[ ! -e /tmp/test.py.pid ]]; then # Comprobar si el archivo ya existe  
    python test.py & # +y si es así no ejecute otro proceso.  
    echo $! > /tmp/test.py.pid  
else  
    echo -n "ERROR: The process is already running with pid "  
    cat /tmp/test.py.pid  
    echo  
fi
```

Esto creará un archivo en el directorio `/tmp` que contiene el pid del proceso `python test.py`. Si el archivo ya existe, asumiremos que el comando ya se está ejecutando y el script devolverá un error.

Luego, cuando quieras matarlo utiliza el siguiente script:

```
#!/bin/bash  
if [[ -e /tmp/test.py.pid ]]; then # Si el fichero no existe, el  
    kill `cat /tmp/test.py.pid` # + el proceso no está en marcha. Inútil  
    rm /tmp/test.py.pid # + intentar matarlo.  
else  
    echo "test.py is not running"  
fi
```

que matará exactamente el proceso asociado con su comando, sin depender de ninguna información volátil (como la cadena usada para ejecutar el comando). Incluso en este caso, si el archivo no existe, el script asume que quieres matar un proceso no en ejecución.

Este último ejemplo se puede mejorar fácilmente para ejecutar el mismo comando varias veces (añadiendo al archivo `pid` en lugar de sobrescribirlo, por ejemplo) y para gestionar los casos en los que el proceso muere antes de ser matado.

Sección 8.2: Comprobar qué proceso se ejecuta en un puerto específico

Para comprobar qué proceso se ejecuta en el puerto 8080

```
lsof -i :8080
```

Sección 8.3: Desaprobar el trabajo de fondo

```
$ gzip extremelylargefile.txt &
$ bg
$ disown %1
```

Esto permite que un proceso de larga duración continúe una vez que su shell (terminal, ssh, etc) se cierra.

Sección 8.4: Listar los trabajos actuales

```
$ tail -f /var/log/syslog > log.txt
[1]+ Stopped tail -f /var/log/syslog > log.txt

$ sleep 10 &

$ jobs
[1]+ Stopped tail -f /var/log/syslog > log.txt
[2]- Running sleep 10 &
```

Sección 8.5: Buscar información sobre un proceso en ejecución

`ps aux | grep <termino-busqueda>` muestra los procesos que coinciden con el término de búsqueda

Por ejemplo:

```
root@server7:~# ps aux | grep nginx
root 315 0.0 0.3 144392 1020 ? Ss May28 0:00 nginx: master process
/usr/sbin/nginx
www-data 5647 0.0 1.1 145124 3048 ? S Jul18 2:53 nginx: worker process
www-data 5648 0.0 0.1 144392 376 ? S Jul18 0:00 nginx: cache manager process
root 13134 0.0 0.3 4960 920 pts/0 S+ 14:33 0:00 grep --color=auto nginx
root@server7:~#
```

Aquí, la segunda columna es el id del proceso. Por ejemplo, si quieres matar el proceso nginx, puedes usar el comando `kill 5647`. Siempre es aconsejable utilizar el comando `kill` con `SIGTERM` en lugar de `SIGKILL`.

Sección 8.6: Listar todos los procesos

Hay dos formas comunes de listar todos los procesos de un sistema. Ambas listan todos los procesos ejecutados por todos los usuarios, aunque difieren en el formato de salida (la razón de las diferencias es histórica).

```
ps -ef # lists all processes
ps aux # lists all processes in alternative format (BSD)
```

Se puede utilizar para comprobar si una aplicación determinada se está ejecutando. Por ejemplo, para comprobar si el servidor SSH (sshd) se está ejecutando:

```
ps -ef | grep sshd
```

Capítulo 9: Redirección

Parámetro

descriptor de archivo interno
dirección
descriptor de archivo externo o ruta

Detalles

Un número entero
Una de las opciones `>`, `<` o `<>`
& seguido de un número entero para el descriptor de fichero o una ruta.

Sección 9.1: Redireccionamiento de la salida estándar

`>` redirigir la salida estándar (también conocida como `STDOUT`) del comando actual a un archivo o a otro descriptor.

Estos ejemplos escriben la salida del comando `ls` en el archivo `file.txt`

```
ls >file.txt  
> file.txt ls
```

El archivo de destino se crea si no existe; en caso contrario, este archivo se trunca.

El descriptor de redirección por defecto es la salida estándar o `1` cuando no se especifica ninguno. Este comando es equivalente a los ejemplos anteriores con la salida estándar indicada explícitamente:

```
ls 1>file.txt
```

Nota: la redirección es inicializada por el shell ejecutado y no por el comando ejecutado, por lo tanto, se realiza antes de la ejecución del comando.

Sección 9.2: Añadir vs Truncar

Truncar `>`

1. Crea el archivo especificado si no existe.
2. Truncar (eliminar el contenido del archivo)
3. Escribir en el archivo

```
$ echo "first line" > /tmp/lines  
$ echo "second line" > /tmp/lines  
$ cat /tmp/lines  
second line
```

Añadir `>>`

1. Crea el archivo especificado si no existe.
2. Añadir archivo (escritura al final del archivo).

```
# Sobrescribir archivo existente  
$ echo "first line" > /tmp/lines  
# Añadir una segunda línea  
$ echo "second line" >> /tmp/lines  
$ cat /tmp/lines  
first line  
second line
```

Sección 9.3: Redireccionamiento de STDOUT y STDERR

Los descriptors de archivo como `0` y `1` son punteros. Cambiamos a qué apuntan los descriptors de archivo con la redirección. `>/dev/null` significa que `1` apunta a `/dev/null`.

Primero apuntamos 1 (STDOUT) a `/dev/null` y luego apuntamos 2 (STDERR) a lo que apunta 1.

```
# STDERR se redirige a STDOUT: redirigido a /dev/null,  
# redirigiendo efectivamente tanto STDERR como STDOUT a /dev/null  
echo 'hello' > /dev/null 2>&1  
  
Version ≥ 4.0
```

Esto se puede resumir en lo siguiente:

```
echo 'hello' &> /dev/null
```

Sin embargo, esta forma puede ser indeseable en producción si la compatibilidad con el shell es una preocupación, ya que entra en conflicto con POSIX, introduce ambigüedad de análisis, y los shells sin esta característica lo malinterpretarán:

```
# Código actual  
echo 'hello' &> /dev/null  
echo 'hello' &> /dev/null 'goodbye'  
  
# Comportamiento deseado  
echo 'hello' > /dev/null 2>&1  
echo 'hello' 'goodbye' > /dev/null 2>&1  
  
# Comportamiento real  
echo 'hello' &  
echo 'hello' & goodbye > /dev/null
```

NOTA: Se sabe que `&>` funciona como se desea tanto en Bash como en Zsh.

Sección 9.4: Uso de tuberías con nombre

A veces es posible que desee que un programa emita algo y lo introduzca en otro, pero no puede utilizar una tubería estándar.

```
ls -l | grep ".log"
```

Podría simplemente escribir en un archivo temporal:

```
touch tempFile.txt  
ls -l > tempFile.txt  
grep ".log" < tempFile.txt
```

Esto funciona bien para la mayoría de las aplicaciones, sin embargo, nadie sabrá lo que hace `tempFile` y alguien podría eliminarlo si contiene la salida de `ls -l` en ese directorio. Aquí es donde una tubería con nombre entra en juego:

```
mkfifo myPipe  
ls -l > myPipe  
grep ".log" < myPipe
```

`myPipe` es técnicamente un archivo (todo lo es en Linux), así que hagamos `ls -l` en un directorio vacío en el que acabamos de crear una tubería:

```
mkdir pipeFolder  
cd pipeFolder  
mkfifo myPipe  
ls -l
```

El resultado es:

```
prw-r--r-- 1 root root 0 Jul 25 11:20 myPipe
```

Fíjate en el primer carácter de los permisos, aparece como una tubería, no como un archivo.

Ahora vamos a hacer algo guay.

Abre un terminal, y toma nota del directorio (o crea uno para que la limpieza sea fácil), y haz una tubería.

```
mkfifo myPipe
```

Ahora pongamos algo en la tubería.

```
echo "Hello from the other side" > myPipe
```

Verás que esto cuelga, el otro lado de la tubería sigue cerrado. Vamos a abrir el otro lado de la tubería y dejar que las cosas a través de.

Abra otro terminal y vaya al directorio en el que se encuentra la tubería (o si lo conoce, añádalo a la tubería):

```
cat < myPipe
```

Observará que después de que salga `hello from the other side`, el programa del primer terminal termina, al igual que el del segundo terminal.

Ahora ejecute los comandos a la inversa. Empieza con `cat < myPipe` y luego haz eco de algo en él. Todavía funciona, porque un programa esperará hasta que algo se pone en la tubería antes de terminar, porque sabe que tiene que conseguir algo.

Las tuberías con nombre pueden ser útiles para mover información entre terminales o entre programas.

Las tuberías son pequeñas. Una vez llenos, el escritor se bloquea hasta que algún lector lee el contenido, por lo que es necesario ejecutar el lector y el escritor en terminales diferentes o ejecutar uno u otro en segundo plano:

```
ls -l /tmp > myPipe &
cat < myPipe
```

Más ejemplos de uso de tuberías con nombre:

- Ejemplo 1 - todos los comandos en el mismo terminal / mismo Shell

```
$ { ls -l && cat file3; } >mypipe &
$ cat <mypipe
# Salida: Imprime los datos de ls -l y luego imprime el contenido de file3 en pantalla
```

- Ejemplo 2 - todos los comandos en el mismo terminal / mismo Shell

```
$ ls -l >mypipe &
$ cat file3 >mypipe &
$ cat <mypipe
# Salida: Imprime en pantalla el contenido de mypipe.
```

Tenga en cuenta que primero se muestra el contenido de `file3` y luego los datos de `ls -l` (configuración LIFO).

- Ejemplo 3 - todos los comandos en el mismo terminal / mismo Shell

```
$ { pipedata=$(<mypipe) && echo "$pipedata"; } &
$ ls >mypipe
# Salida: Imprime la salida de ls directamente en pantalla
```

Tenga en cuenta que la variable `$pipedata` no está disponible para su uso en el terminal principal / shell principal, ya que el uso de `&` invoca un subshell y `$pipedata` sólo estaba disponible en este subshell.

- Ejemplo 4 - todos los comandos en el mismo terminal / mismo Shell

```
$ export pipedata
$ pipedata=$(<mypipe) &
$ ls -l *.sh >mypipe
$ echo "$pipedata"
# Salida: Imprime correctamente el contenido de mypipe
```

Esto imprime correctamente el valor de la variable `$pipedata` en el shell principal debido a la declaración de exportación de la variable. El terminal principal/ shell principal no se cuelga debido a la invocación de un shell en segundo plano (`&`).

Sección 9.5: Redirección a direcciones de red

Version ≥ 2.04

Bash trata algunas rutas como especiales y puede hacer algunas comunicaciones de red escribiendo a `/dev/{udp|tcp}/host/port`. Bash no puede configurar un servidor de escucha, pero puede iniciar una conexión, y para TCP puede leer los resultados al menos.

Por ejemplo, para enviar una simple petición web se podría hacer:

```
exec 3</dev/tcp/www.google.com/80
printf 'GET / HTTP/1.0\r\n\r\n' >&3
cat <&3
```

y los resultados de la página web por defecto de `www.google.com` se imprimirán en `stdout`.

Igualmente

```
printf 'HI\n' >/dev/udp/192.168.1.1/6666
```

enviaría un mensaje UDP que contiene `HI\n` a un oyente en `192.168.1.1:6666`

Sección 9.6: Imprimir mensajes de error en stderr

Los mensajes de error se incluyen generalmente en un script con fines de depuración o para proporcionar una experiencia de usuario enriquecida. Basta con escribir un mensaje de error como este:

```
cmd || echo 'cmd failed'
```

puede funcionar para casos sencillos, pero no es lo habitual. En este ejemplo, el mensaje de error contaminará la salida real del script mezclando tanto los errores como la salida exitosa en `stdout`.

En resumen, el mensaje de error debe ir a `stderr` y no a `stdout`. Es bastante simple:

```
cmd || echo 'cmd failed' >/dev/stderr
```

Otro ejemplo:

```
if cmd; then
    echo 'success'
else
    echo 'cmd failed' >/dev/stderr
fi
```

En el ejemplo anterior, el mensaje de éxito se imprimirá en `stdout` mientras que el mensaje de error se imprimirá en `stderr`.

Una forma mejor de imprimir el mensaje de error es definir una función:

```
err(){
    echo "E: $*" >>/dev/stderr
}
```

Ahora, cuando tenga que imprimir un error:

```
err "My error message"
```

Sección 9.7: Redirigir varios comandos al mismo archivo

```
{
    echo "contents of home directory"
    ls ~
} > output.txt
```

Sección 9.8: Redireccionamiento de STDIN

< lee de su argumento derecho y escribe en su argumento izquierdo.

Para escribir un archivo en `STDIN` debemos *leer* `/tmp/a_file` y *escribir* en `STDIN` es decir `0</tmp/a_file`.

Nota: El descriptor de archivo interno es por defecto `0` (`STDIN`) para <

```
$ echo "b" > /tmp/list.txt
$ echo "a" >> /tmp/list.txt
$ echo "c" >> /tmp/list.txt
$ sort < /tmp/list.txt
a
b
c
```

Sección 9.9: Redireccionamiento de STDERR

2 es `STDERR`.

```
$ echo_to_stderr 2>/dev/null # echos nothing
```

Definiciones:

`echo_to_stderr` es un comando que escribe `"stderr"` en `STDERR`

```
echo_to_stderr () {
    echo stderr >&2
}
$ echo_to_stderr
stderr
```

Sección 9.10: Explicación de STDIN, STDOUT y STDERR

Los comandos tienen una entrada (`STDIN`) y dos tipos de salidas, la salida estándar (`STDOUT`) y el error estándar (`STDERR`).

Por ejemplo:

STDIN

```
root@server~# read
Type some text here
```

La entrada estándar se utiliza para proporcionar entrada a un programa. (Aquí estamos usando el builtin `read` para leer una línea desde `STDIN`).

STDOUT

```
root@server~# ls file
file
```

La salida estándar se utiliza generalmente para la salida "normal" de un comando. Por ejemplo, `ls` lista archivos, por lo que los archivos se envían a `STDOUT`.

STDERR

```
root@server~# ls anotherfile
ls: cannot access 'anotherfile': No such file or directory
```

El error estándar se utiliza (como su nombre indica) para los mensajes de error. Como este mensaje no es una lista de archivos, se envía a `STDERR`.

STDIN, STDOUT y STDERR son los tres *flujos estándar*. Se identifican al shell por un número en lugar de un nombre:

0 = Entrada estándar

1 = Salida estándar

2 = Error estándar

Por defecto, STDIN se conecta al teclado, y tanto STDOUT como STDERR aparecen en el terminal. Sin embargo, podemos redirigir STDOUT o STDERR a lo que necesitemos. Por ejemplo, supongamos que sólo necesitamos la salida estándar y que todos los mensajes de error impresos en el error estándar deben suprimirse. Es entonces cuando utilizamos los descriptores 1 y 2.

Redireccionamiento de STDERR a /dev/null

Tomando el ejemplo anterior,

```
root@server~# ls anotherfile 2>/dev/null
root@server~#
```

En este caso, si hay algún STDERR, será redirigido a /dev/null (un archivo especial que ignora cualquier cosa que se introduzca en él), por lo que no obtendrá ninguna salida de error en el shell.

Capítulo 10: Estructuras de control

Parámetro a [o test

Operadores de archivos

```
-e "$file"
-d "$file"
-f "$file"
-h "$file"
```

Comparadores de cadenas de texto

```
-z "$str"
-n "$str"
"$str" = "$str2"
```

```
"$str" != "$str2"
```

Comparadores de enteros

```
"$int1" -eq "$int2"
"$int1" -ne "$int2"
"$int1" -gt "$int2"
"$int1" -ge "$int2"
"$int1" -lt "$int2"
"$int1" -le "$int2"
```

Detalles

Detalles

Devuelve true si el archivo existe.

Devuelve true si el archivo existe y es un directorio

Devuelve true si el archivo existe y es un fichero normal

Devuelve true si el archivo existe y es un enlace simbólico

Detalles

Verdadero si la longitud de la cadena de texto es cero

Verdadero si la longitud de la cadena de texto es distinta de cero

Verdadero si la cadena de texto `$str` es igual a la cadena de texto `$str2`. No es lo mejor para enteros. Puede funcionar, pero será inconsistente

Verdadero si las cadenas de texto no son iguales

Detalles

Verdadero si los números enteros son iguales

Verdadero si los enteros no son iguales

Verdadero si `int1` es mayor que `int 2`

Verdadero si `int1` es mayor o igual que `int2`

Verdadero si `int1` es menor que `int 2`

Verdadero si `int1` es menor o igual que `int2`

Sección 10.1: Ejecución condicional de listas de comandos

Cómo utilizar la ejecución condicional de listas de comandos

Cualquier comando, expresión o función incorporada, así como cualquier comando o script externo puede ejecutarse condicionalmente utilizando los operadores `&&(and)` y `||(or)`.

Por ejemplo, esto sólo imprimirá el directorio actual si el comando `cd` tuvo éxito.

```
cd my_directory && pwd
```

Del mismo modo, esto saldrá si el comando `cd` falla, evitando la catástrofe:

```
cd my_directory || exit
rm -rf *
```

Al combinar varias sentencias de esta forma, es importante recordar que (a diferencia de muchos lenguajes de estilo C) estos operadores no tienen precedencia y son asociativos a la izquierda.

Así, esta declaración funcionará como se espera...

```
cd my_directory && pwd || echo "No such directory"
```

- Si `cd` tiene éxito, se ejecuta `&& pwd` y se imprime el nombre del directorio de trabajo actual. A menos que `pwd` falle (una rareza) el `|| echo ...` no se ejecutará.
- Si el `cd` falla, se saltará el `&& pwd` y se ejecutará el `|| echo ...`.

Pero esto no (si estás pensando `if...then...else`)...

```
cd my_directory && ls || echo "No such directory"
```

- Si el `cd` falla, se salta el `&& ls` y se ejecuta el `|| echo ...`.
- Si el `cd` tiene éxito, se ejecuta el `&& ls`.
 - Si el `ls` tiene éxito, el `|| echo ...` es ignorado. (*hasta aquí todo bien*)
 - **PERO... si el `ls` falla, el `|| echo ...` también se ejecutará.**

Es el **ls**, no el **cd**, el comando anterior.

Por qué utilizar la ejecución condicional de listas de comandos

La ejecución condicional es un poco más rápida que `if . . . then`, pero su principal ventaja es que permite a las funciones y scripts salir antes de tiempo, o "cortocircuitar".

A diferencia de muchos lenguajes como C donde la memoria es explícitamente asignada para structs y variables (y por lo tanto debe ser desasignada), **bash** maneja esto bajo las cubiertas. En la mayoría de los casos, no tenemos que limpiar nada antes de salir de la función. Una sentencia **return** desasignará todo lo local a la función y retomará la ejecución en la dirección de retorno en la pila.

Volver de funciones o salir de scripts tan pronto como sea posible puede mejorar significativamente el rendimiento y reducir la carga del sistema al evitar la ejecución innecesaria de código. Por ejemplo...

```
my_function () {  
    ### COMPROBAR SIEMPRE EL CÓDIGO DE RETORNO  
    # se requiere un argumento. "" se evalúa como falso(1)  
    [[ "$1" ]] || return 1  
    # funciona con el argumento. salida en caso de fallo  
    do_something_with "$1" || return 1  
    do_something_else || return 1  
    # Exito! No se han detectado fallos, o no estaríamos aqui  
    return 0  
}
```

Sección 10.2: Declaración if

```
if [[ $1 -eq 1 ]]; then  
    echo "1 was passed in the first parameter"  
elif [[ $1 -gt 2 ]]; then  
    echo "2 was not passed in the first parameter"  
else  
    echo "The first parameter was not 1 and is not more than 2."  
fi
```

La **fi** de cierre es necesaria, pero las cláusulas **elif** y/o **else** pueden omitirse.

El punto y coma que precede a **then** es la sintaxis estándar para combinar dos órdenes en una sola línea; sólo puede omitirse si **then** se traslada a la línea siguiente.

Es importante entender que los corchetes `[[` no forman parte de la sintaxis, sino que se tratan como un comando; es el código de salida de este comando lo que se está comprobando. Por lo tanto, siempre debes incluir espacios alrededor de los corchetes.

Esto también significa que el resultado de cualquier comando puede ser probado. Si el código de salida del comando es un cero, la sentencia se considera verdadera.

```
if grep "foo" bar.txt; then  
    echo "foo was found"  
else  
    echo "foo was not found"  
fi
```

Las expresiones matemáticas, cuando se colocan dentro de paréntesis dobles, también devuelven 0 o 1 de la misma manera, y también se pueden probar:

```
if (( $1 + 5 > 91 )); then  
    echo "$1 is greater than 86"  
fi
```

También puede encontrar sentencias **if** con corchetes simples. Éstas están definidas en el estándar POSIX y se garantiza que funcionan en todos los shells compatibles con POSIX, incluido Bash. La sintaxis es muy similar a la de Bash:

```
if [ "$1" -eq 1 ]; then
    echo "1 was passed in the first parameter"
elif [ "$1" -gt 2 ]; then
    echo "2 was not passed in the first parameter"
else
    echo "The first parameter was not 1 and is not more than 2."
fi
```

Sección 10.3: Recorrer un array en bucle

bucle **for**:

```
arr=(a b c d e f)
for i in "${arr[@]";do
    echo "$i"
done

O

for ((i=0;i<${#arr[@]};i++));do
    echo "${arr[$i]}"
done
```

bucle **while**:

```
i=0
while [ $i -lt ${#arr[@]} ];do
    echo "${arr[$i]}"
    i=$((expr $i + 1))
done

O

i=0
while (( $i < ${#arr[@]} ));do
    echo "${arr[$i]}"
    ((i++))
done
```

Sección 10.4: Uso del bucle for para iterar números en una lista

```
#!/bin/bash
for i in {1..10}; do # {1..10} se expande a «1 2 3 4 5 6 7 8 9 10»
    echo $i
done
```

Esto produce lo siguiente:

```
1
2
3
4
5
6
7
8
8
10
```

Sección 10.5: continue y break

Ejemplo de **continue**

```
for i in [series]
do
    command 1
    command 2
    if (condition) # Condicion para saltar el comando 3
        continue # saltar al siguiente valor de la "serie"
    fi
    command 3
done
```

Ejemplo de **break**

```
for i in [series]
do
    command 4
    if (condition) # Condición para romper el bucle
    then
        command 5 # Comando si es necesario romper el bucle
        break
    fi
    command 6 # Comando a ejecutar si la "condicion" nunca es verdadera
done
```

Sección 10.6: Interrupción del bucle

Romper bucle múltiple:

```
arr=(a b c d e f)
for i in "${arr[@]}";do
    echo "$i"
    for j in "${arr[@]}";do
        echo "$j"
        break 2
    done
done
```

Salida:

```
a
a
```

Romper bucle único:

```
arr=(a b c d e f)
for i in "${arr[@]}";do
    echo "$i"
    for j in "${arr[@]}";do
        echo "$j"
        break
    done
done
```


Salida:

a
a
b
a
c
a
d
a
e
a
f
a

Sección 10.7: Bucle while

```
#!/bin/bash
i=0

while [ $i -lt 5 ] # Mientras i sea inferior a 5
do
    echo "i is currently $i"
    i=$((i+1)) # No la falta de espacios entre los paréntesis. Esto hace que no sea una expresión de prueba
done # finaliza el bucle
```

Observe que hay espacios alrededor de los paréntesis durante la prueba (después de la sentencia while). Estos espacios son necesarios.

Este bucle tiene salida:

i is currently 0
i is currently 1
i is currently 2
i is currently 3
i is currently 4

Sección 10.8: Bucle for con sintaxis tipo C

El formato básico del bucle **for** en C es:

```
for (( variable assignment; condition; iteration process ))
```

Notas:

- La asignación de la variable dentro del bucle **for** estilo C puede contener espacios a diferencia de la asignación habitual.
- Las variables dentro del bucle **for** estilo C no van precedidas de \$.

Ejemplo:

```
for (( i = 0; i < 10; i++ ))
do
    echo "The iteration number is $i"
done
```

También podemos procesar múltiples variables dentro de un bucle **for** al estilo C:

```
for (( i = 0, j = 0; i < 10; i++, j = i * i ))
do
    echo "The square of $i is equal to $j"
done
```

Sección 10.9: Bucle until

El bucle **until** se ejecuta hasta que la condición es verdadera

```
i=5
until [[ i -eq 10 ]]; do # Comprueba si i=10
    echo "i=$i" # Imprime el valor de i
    i=$((i+1)) # Incrementa i en 1
done
```

Salida:

```
i=5
i=6
i=7
i=8
i=9
```

Cuando **i** llega a 10 la condición en el bucle **until** se convierte en verdadera y el bucle termina.

Sección 10.10: Sentencia switch con case

Con la sentencia **case** se pueden comparar valores con una variable.

El argumento pasado a **case** se expande e intenta coincidir con cada uno de los patrones.

Si se encuentra una coincidencia, se ejecutan los comandos hasta **;;**.

```
case "$BASH_VERSION" in
    [34]*)
        echo {1..4}
        ;;
    *)
        seq -s" " 1 4
esac
```

Los patrones no son expresiones regulares, sino coincidencia de patrones de shell (también conocidos como globos).

Sección 10.11: Bucle for sin parámetro de lista de palabras

```
for arg; do
    echo arg=$arg
done
```

Un bucle **for** sin un parámetro de lista de palabras iterará sobre los parámetros posicionales en su lugar. En otras palabras, el ejemplo anterior es equivalente a este código:

```
for arg in "$@"; do
    echo arg=$arg
done
```

En otras palabras, si se sorprende escribiendo **for i in "\$@"; do ...; done**, elimine la parte **in** y escriba simplemente **for i; do ...; done**.

Capítulo 11: Comandos true, false y :

Sección 11.1: Bucle infinito

```
while true; do
    echo ok
done
```

O

```
while ;; do
    echo ok
done
```

O

```
until false; do
    echo ok
done
```

Sección 11.2: Función return

```
function positive() {
    return 0
}
function negative() {
    return 1
}
```

Sección 11.3: Código que siempre/nunca será ejecutado

```
if true; then
    echo Always executed
fi
if false; then
    echo Never executed
fi
```

Capítulo 12: Arrays

Sección 12.1: Asignación de arrays

Asignación de listas

Si está familiarizado con Perl, C o Java, podría pensar que Bash utilizaría comas para separar los elementos del array, sin embargo, este no es el caso; en su lugar, Bash utiliza espacios:

```
# Array en Perl
my @array = (1, 2, 3, 4);
```

```
# Array en Bash
array=(1 2 3 4)
```

Crea un array con nuevos elementos:

```
array=('first element' 'second element' 'third element')
```

Asignación de subíndices

Crea un array con índices de elementos explícitos:

```
array=[3]='fourth element' [4]='fifth element')
```

Asignación por índice

```
array[0]='first element'
array[1]='second element'
```

Asignación por nombre (array asociativo)

Version ≥ 4.0

```
declare -A array
array[first]='First element'
array[second]='Second element'
```

Asignación dinámica

Crea un array a partir de la salida de otro comando, por ejemplo, utiliza **seq** para obtener un rango de 1 a 10:

```
array=(`seq 1 10`)
```

Asignación a partir de los argumentos de entrada del script:

```
array=("$@")
```

Asignación dentro de bucles:

```
while read -r; do
    # array+=("$REPLY") # Añadir array
    array[$i]="$REPLY" # Asignación por índice
    let i++ # Índice de incremento
done < <(`seq 1 10`) # sustitución de comandos
echo ${array[@]} # salida: 1 2 3 4 5 6 7 8 9 10
```

donde `$REPLY` es siempre la entrada actual

Sección 12.2: Acceso a los elementos del array

Imprimir elemento en índice 0

```
echo "${array[0]}"
```

Version < 4.3

Imprimir el último elemento utilizando la sintaxis de expansión de subcadenas de texto

```
echo "${arr[@]: -1 }"
```

Version ≥ 4.3

Imprimir el último elemento utilizando la sintaxis de subíndice

```
echo "${array[-1]}"
```

Imprimir todos los elementos, cada uno citado por separado

```
echo "${array[@]}"
```

Imprimir todos los elementos como una sola cadena de texto entre comillas

```
echo "${array[*]}"
```

Imprime todos los elementos a partir del índice 1, cada uno citado por separado

```
echo "${array[@]:1}"
```

Imprimir 3 elementos del índice 1, cada uno citado por separado

```
echo "${array[@]:1:3}"
```

Operaciones de cadena de texto

Si se refiere a un solo elemento, se permiten las operaciones de cadena de texto:

```
array=(zero one two)
echo "${array[0]:0:3}" # da cero (caracteres en posición 0, 1 y 2 en la cadena cero)
echo "${array[0]:1:3}" # da cero (caracteres en las posiciones 1, 2 y 3 de la cadena cero)
```

así `${array[$i]:N:M}` da una cadena de texto a partir de la posición N (empezando por 0) en la cadena de texto `${array[$i]}` con M caracteres siguientes.

Sección 12.3: Modificación de arrays

Cambiar índice

Inicializar o actualizar un elemento concreto del array

```
array[10]="eleventh element" # porque empieza por 0
```

Version ≥ 3.1

Añadir

Modifica el array, añadiendo elementos al final si no se especifica ningún subíndice.

```
array+=('fourth element' 'fifth element')
```

Sustituye todo el array por una nueva lista de parámetros.

```
array=("${array[@]}" "fourth element" "fifth element")
```

Añade un elemento al principio:

```
array=("new element" "${array[@]}")
```

Insertar

Inserta un elemento en un índice dado:

```
arr=(a b c d)
# insertar un elemento en el índice 2
i=2
arr=("${arr[@]:0:$i}" 'new' "${arr[@]:$i}")
echo "${arr[2]}" # salida: nuevo
```

Borrar

Elimina índices de arrays utilizando la función **unset**:

```
arr=(a b c)
echo "${arr[@]}" # salidas: a b c
echo "${!arr[@]}" # salidas: 0 1 2
unset -v 'arr[1]'
echo "${arr[@]}" # salidas: a c
echo "${!arr[@]}" # salidas: 0 2
```

Fusionar

```
array3=("${array1[@]}" "${array2[@]}")
```

Esto también funciona con arrays dispersos.

Reindexación de un array

Esto puede ser útil si se han eliminado elementos de un array, o si no está seguro de si hay huecos en el array. Para recrear los índices sin huecos:

```
array=("${array[@]}")
```

Sección 12.4: Iteración del array

La iteración de arrays viene en dos gustos, **foreach** y el clásico bucle **for**:

```
a=(1 2 3 4)
# bucle foreach
for y in "${a[@]"; do
    # actuar sobre $y
    echo "$y"
done
# bucle for clásico
for ((idx=0; idx < ${#a[@]}; ++idx)); do
    # actuar sobre ${a[$idx]}
    echo "${a[$idx]}"
done
```

También puede iterar sobre la salida de un comando:

```
a=($(tr ' ' <<<"a,b,c,d")) # tr puede transformar un personaje en otro
for y in "${a[@]"; do
    echo "$y"
done
```

Sección 12.5: Longitud del array

\${#array[@]} da la longitud del array **\${array[@]}**:

```
array=('first element' 'second element' 'third element')
echo "${#array[@]}" # da una longitud de 3
```

Esto funciona también con cadenas de texto en elementos individuales:

```
echo "${#array[0]}" # da la longitud de la cadena de texto en el elemento 0: 13
```

Sección 12.6: Arrays asociativos

Version \geq 4.0

Declarar un array asociativo

```
declare -A aa
```

Es obligatorio declarar un array asociativo antes de inicializarlo o utilizarlo.

Inicializar elementos

Puedes inicializar los elementos de uno en uno de la siguiente manera:

```
aa[hello]=world
aa[ab]=cd
aa["key with space"]="hello world"
```

También puedes inicializar un array asociativo completo en una sola sentencia:

```
aa=( [hello]=world [ab]=cd ["key with space"]="hello world" )
```

Acceder a un elemento de un array asociativo

```
echo ${aa[hello]}
# Salida: world
```

Listado de claves de arrays asociativos

```
echo "${!aa[@]}"
# Salida: hello ab key with space
```

Listado de valores de arrays asociativos

```
echo "${aa[@]}"
# Salida: world cd hello world
```

Iterar sobre claves y valores de arrays asociativos

```
for key in "${!aa[@]}; do
    echo "Key: ${key}"
    echo "Value: ${array[$key]}"
done
# Salida:
# Key: hello
# Value: world
# Key: ab
# Value: cd
# Key: key with space
# Value: hello world
```

Contar elementos de arrays asociativos

```
echo "${#aa[@]}"
# Salida: 3
```

Sección 12.7: Recorrer un array con bucles

Nuestro array de ejemplo:

```
arr=(a b c d e f)
```

Usando un bucle **for..in**:

```
for i in "${arr[@]"; do
    echo "$i"
done
```

Version ≥ 2.04

Usando el bucle **for** estilo C:

```
for ((i=0;i<${#arr[@]};i++)); do
    echo "${arr[$i]}"
done
```

Usando el bucle **while**:

```
i=0
while [ $i -lt ${#arr[@]} ]; do
    echo "${arr[$i]}"
    i=$((i + 1))
done
```

Version ≥ 2.04

Uso del bucle **while** con condicional numérico:

```
i=0
while (( $i < ${#arr[@]} )); do
    echo "${arr[$i]}"
    ((i++))
done
```

Usando un bucle **until**:

```
i=0
until [ $i -ge ${#arr[@]} ]; do
    echo "${arr[$i]}"
    i=$((i + 1))
done
```

Version ≥ 2.04

Usando un bucle **until** con condicional numérico:

```
i=0
until (( $i >= ${#arr[@]} )); do
    echo "${arr[$i]}"
    ((i++))
done
```

Sección 12.8: Destruir, eliminar o anular un array

Para destruir, borrar o deshacer un array:

```
unset array
```

Para destruir, borrar o deshacer un único elemento del array:

```
unset array[10]
```

Sección 12.9: Array a partir de cadena de caracteres

```
stringVar="Apple Orange Banana Mango"
arrayVar=(${stringVar// / })
```


Cada espacio en la cadena de texto denota un nuevo elemento en el array resultante.

```
echo ${arrayVar[0]} # imprimirá Apple
echo ${arrayVar[3]} # imprimirá Mango
```

Del mismo modo, se pueden utilizar otros caracteres para el delimitador.

```
stringVar="Apple+Orange+Banana+Mango"
arrayVar=(${stringVar//+/ })
echo ${arrayVar[0]} # imprimirá Apple
echo ${arrayVar[2]} # imprimirá Banana
```

Sección 12.10: Lista de índices inicializados

Obtener la lista de índices inicializados en un array

```
$ arr[2]='second'
$ arr[10]='tenth'
$ arr[25]='twenty five'
$ echo ${!arr[@]}
2 10 25
```

Sección 12.11: Lectura de un archivo completo en un array

Leer en un solo paso:

```
IFS=$'\n' read -r -a arr < file
```

Lectura en bucle:

```
arr=()
while IFS= read -r line; do
arr+=("$line")
done
```

Version ≥ 4.0

Utilizando `mapfile` o `readarray` (que son sinónimos):

```
mapfile -t arr < file
readarray -t arr < file
```

Sección 12.12: Función de inserción de arrays

Esta función insertará un elemento en un array en un índice dado:

```
insert(){
    h='
##### insert #####
# Usage:
# insert arr_name index element
#
# Parameters:
# arr_name : Name of the array variable
# index : Index to insert at
# element : Element to insert
#####
'

    [[ $1 = -h ]] && { echo "$h" >/dev/stderr; return 1; }
    declare -n __arr__=$1 # referencia a la variable array
    i=$2 # índice a insertar en
    el="$3" # elemento para insertar
    # manejar errores
    [[ ! "$i" =~ ^[0-9]+$ ]] && { echo "E: insert: index must be a valid integer" >/dev/stderr;
return 1; }
    (( $1 < 0 )) && { echo "E: insert: index can not be negative" >/dev/stderr; return 1; }
    # Ahora inserta $el en $i
    __arr__=("${__arr__[@]:0:$i}" "$el" "${__arr__[@]:$i}")
}
```

Uso:

insert array_variable_name index element

Ejemplo:

```
arr=(a b c d)
echo "${arr[2]}" # salida: c
# Ahora llama a la función insertar y pasa el nombre de la variable del array,
# el índice en el que insertar
# y el elemento a insertar
insert arr 2 'New Element'
# 'New Element' se insertó en el índice 2 en arr, ahora imprimirlos
echo "${arr[2]}" # salida: New Element
echo "${arr[3]}" # salida: c
```

Capítulo 13: Arrays asociativos

Sección 13.1: Examinar los arrays assoc

Todo el uso necesario se muestra con este fragmento:

```
#!/usr/bin/env bash
declare -A assoc_array=( [key_string]=value \
    [one]="something" \
    [two]="another thing" \
    [ three ]='mind the blanks!' \
    [ " four" ]='count the blanks of this key later!' \
    [IMPORTANT]='SPACES DO ADD UP!!!'
\
    [1]='there are no integers!' \
    [info]="to avoid history expansion " \
    [info2]="quote exclamation mark with single quotes" \
)
echo # solo una linea en blanco
echo now here are the values of assoc_array:
echo ${assoc_array[@]}
echo not that useful,
echo # solo una linea en blanco
echo this is better:

declare -p assoc_array # -p == print

echo have a close look at the spaces above\!\!\!
echo # solo una linea en blanco

echo accessing the keys
echo the keys in assoc_array are ${!assoc_array[*]}
echo mind the use of indirection operator \!
echo # solo una linea en blanco

echo now we loop over the assoc_array line by line
echo note the \! indirection operator which works differently,
echo if used with assoc_array.
echo # solo una linea en blanco

for key in "${!assoc_array[@]}"; do # accessing keys using ! indirection!!!!
printf "key: \"%s\"\\nvalue: \"%s\"\\n\\n" "$key" "${assoc_array[$key]}"
done

echo have a close look at the spaces in entries with keys two, three and four above\!\!\!
echo # solo una linea en blanco
echo # otra linea en blanco

echo there is a difference using integers as keys\!\!\!
i=1
echo declaring an integer var i=1
echo # solo una linea en blanco
echo Within an integer_array bash recognizes arithmetic context.
echo Within an assoc_array bash DOES NOT recognize arithmetic context.
echo # solo una linea en blanco
echo this works: \${assoc_array[\$i]}: ${assoc_array[$i]}
echo this NOT!!: \${assoc_array[i]}: ${assoc_array[i]}
echo # solo una linea en blanco
echo # solo una linea en blanco
echo an \${assoc_array[i]} has a string context within braces in contrast to an integer_array
declare -i integer_array=( one two three )
echo "doing a: declare -i integer_array=( one two three )"
```

```
echo # solo una linea en blanco
```

```
echo both forms do work: \${integer_array[i]} : ${integer_array[i]}
```

```
echo and this too: \${integer_array[\$i]} : ${integer_array[$i]}
```

Capítulo 14: Funciones

Sección 14.1: Funciones con argumentos

En `helloJohn.sh`:

```
#!/bin/bash
greet() {
    local name="$1"
    echo "Hello, $name"
}

greet "John Doe"

# running above script
$ bash helloJohn.sh
Hello, John Doe
```

1. Si no modificas el argumento de ninguna manera, no hay necesidad de copiarlo a una variable `local` - simplemente haz `echo "Hello, $1"`.
2. Puedes utilizar `$1`, `$2`, `$3` y así sucesivamente para acceder a los argumentos dentro de la función.

Nota: para argumentos mayores a 9 `$10` no funcionará (bash lo leerá como `$10`), necesitas hacer `${10}`, `${10}` y así sucesivamente.

3. `$@` se refiere a todos los argumentos de una función:

```
#!/bin/bash
foo() {
    echo "$@"
}

foo 1 2 3 # output => 1 2 3
```

Nota: Prácticamente siempre se deben utilizar comillas dobles alrededor de `"$@"`, como aquí.

La omisión de las comillas hará que el shell expanda los comodines (incluso cuando el usuario los haya entrecomillado específicamente para evitarlo) y, en general, introducirá comportamientos no deseados y, potencialmente, incluso problemas de seguridad.

```
foo "string with spaces;" '$HOME' "*"
# salida => string with spaces; $HOME *
```

4. para argumentos por defecto usar `${1:-default_val}`. Ej:

```
#!/bin/bash
foo() {
    local val=${1:-25}
    echo "$val"
}

foo # salida => 25
foo 30 # salida => 30
```

5. para argumentos por defecto usar `${1:-default_val}`. Ej:

para requerir un argumento use `${var:?error message}`

```
foo() {
    local val=${1:?Must provide an argument}
    echo "$val"
}
```

Sección 14.2: Función simple

En `helloWorld.sh`

```
#!/bin/bash
# Definir una función greet
greet ()
{
    echo "Hello World!"
}
# Llama a la función greet
greet
```

Al ejecutar el script, vemos nuestro mensaje

```
$ bash helloWorld.sh
Hello World!
```

Tenga en cuenta que el origen de un archivo con funciones hace que estén disponibles en su sesión bash actual.

```
$ source helloWorld.sh # o, más fácilmente, ". helloWorld.sh"
$ greet
Hello World!
```

Puede exportar una función en algunos shells, de modo que quede expuesta a procesos hijos.

```
bash -c 'greet' # falla
export -f greet # función de exportación; nota -f
bash -c 'greet' # éxito
```

Sección 14.3: Gestión de flags y parámetros opcionales

La función `getopts` puede usarse dentro de funciones para escribir funciones que incluyan banderas y parámetros opcionales. Esto no presenta ninguna dificultad especial, pero hay que manejar adecuadamente los valores tocados por `getopts`. Como ejemplo, definimos una función `failwith` que escribe un mensaje en `stderr` y sale con el código 1 o un código arbitrario suministrado como parámetro a la opción `-x`:

```
# failwith [-x STATUS] PRINTF-LIKE-ARGV
# Falla con el mensaje de diagnóstico dado
#
# La bandera -x se puede utilizar para transmitir un estado de salida personalizado, en lugar
# del valor 1. Se añade automáticamente una nueva línea a la salida.
failwith()
{
    local OPTIND OPTION OPTARG status

    status=1
    OPTIND=1

    while getopts 'x:' OPTION; do
        case ${OPTION} in
            x) status="${OPTARG}";;
            *) 1>&2 printf 'failwith: %s: Unsupported option.\n' "${OPTION}";;
        esac
    done

    shift $(( OPTIND - 1 ))
    {
        printf 'Failure: '
        printf "$@"
        printf '\n'
    } 1>&2
    exit "${status}"
}
```

Esta función puede utilizarse del siguiente modo:

```
failwith '%s: File not found.' "${filename}"
failwith -x 70 'General internal error.'
```

y así sucesivamente.

Tenga en cuenta que, al igual que para *printf*, no deben utilizarse variables como primer argumento. Si el mensaje a imprimir consiste en el contenido de una variable, se debe utilizar el especificador `%s` para imprimirlo, como en

```
failwith '%s' "${message}"
```

Sección 14.4: Imprimir la definición de la función

```
getfunc() {
declare -f "$@"
}
function func(){
echo "I am a sample function"
}
funcd="$(getfunc func)"
getfunc func # or echo "$funcd"
```

Salida:

```
func ()
{
echo "I am a sample function"
}
```

Sección 14.5: Una función que acepta parámetros con nombre

```
foo() {
while [[ "$#" -gt 0 ]]
do
    case $1 in
        -f|--follow)
            local FOLLOW="following"
            ;;
        -t|--tail)
            local TAIL="tail=$2"
            ;;
    esac
    shift
done

echo "FOLLOW: $FOLLOW"
echo "TAIL: $TAIL"
}
```

Ejemplo de uso:

```
foo -f
foo -t 10
foo -f --tail 10
foo --follow --tail 10
```

Sección 14.6: Valor de retorno de una función

La sentencia **return** en Bash no devuelve un valor como las funciones C, sino que sale de la función con un estado de retorno. Puedes pensar en ella como el estado de salida de esa función.

Si desea devolver un valor de la función a continuación, enviar el valor a `stdout` como este:

```
fun() {  
    local var="Sample value to be returned"  
    echo "$var"  
    # printf "%s\n" "$var"  
}
```

Ahora, si lo haces:

```
var="$(fun)"
```

la salida de `fun` se almacenará en `$var`.

Sección 14.7: El código de salida de una función es el código de salida de su último comando

Considere esta función de ejemplo para comprobar si un host está activo:

```
is_alive() {  
    ping -c1 "$1" &> /dev/null  
}
```

Esta función envía un único ping al host especificado por el primer parámetro de la función. Tanto la salida como la salida de error de `ping` se redirigen a `/dev/null`, por lo que la función nunca mostrará nada. Pero el comando `ping` tendrá código de salida 0 en caso de éxito, y distinto de cero en caso de fallo. Como este es el último (y en este ejemplo, el único) comando de la función, el código de salida de `ping` será reutilizado para el código de salida de la propia función.

Este hecho es muy útil en las sentencias condicionales.

Por ejemplo, si el host `graucho` está activo, conéctate a él con `ssh`:

```
if is_alive graucho; then  
    ssh graucho  
fi
```

Otro ejemplo: comprueba repetidamente hasta que el host `graucho` esté levantado, y luego conéctate a él con `ssh`:

```
while ! is_alive graucho; do  
    sleep 5  
done  
ssh graucho
```


Capítulo 15: Expansión de parámetros Bash

El carácter `$` introduce la expansión de parámetros, la sustitución de comandos o la expansión aritmética. El nombre del parámetro o símbolo a expandir puede ir encerrado entre llaves, que son opcionales, pero sirven para proteger la variable a expandir de los caracteres que le siguen inmediatamente y que podrían interpretarse como parte del nombre.

Más información en el [Manual del usuario de Bash](#).

Sección 15.1: Modificación de las mayúsculas y minúsculas de los caracteres alfabéticos

Version \geq 4.0

Para mayúsculas

```
$ v="hello"
# Sólo el primer carácter
$ printf '%s\n' "${v^}"
Hello
# Todos los caracteres
$ printf '%s\n' "${v^^}"
HELLO
# Alternativo
$ v="hello world"
$ declare -u string="$v"
$ echo "$string"
HELLO WORLD
```

Para minúsculas

```
$ v="BYE"
# Sólo el primer carácter
$ printf '%s\n' "${v,}"
bYE
# Todos los caracteres
$ printf '%s\n' "${v,,}"
bye
# Alternativo
$ v="HELLO WORLD"
$ declare -l string="$v"
$ echo "$string"
hello world
```

Toggle Case

```
$ v="Hello World"
# Todos los caracteres
$ echo "${v~}"
hELLO wORLD
$ echo "${v~}"
hello World
# Sólo el primer carácter
hello World
```

Sección 15.2: Longitud del parámetro

```
# Longitud de una cadena de texto
$ var='12345'
$ echo "${#var}"
5
```

Tenga en cuenta que la longitud en número de caracteres no es necesariamente la misma que el número de bytes (como en UTF-8, donde la mayoría de los caracteres se codifican en más de un byte), ni el número de glifos/grafemas (algunos de los cuales son combinaciones de caracteres), ni es necesariamente la misma que la anchura de visualización.

```
# Número de elementos del array
```

```
$ myarr=(1 2 3)
$ echo "${#myarr[@]}"
3
```

```
# También funciona con parámetros de posición
```

```
$ set -- 1 2 3 4
$ echo "${#@}"
4
```

```
# Pero más comúnmente (y portablemente a otros shells), uno usaría
```

```
$ echo "$#"
4
```

Sección 15.3: Reemplazar patrón en cadena de caracteres

Primera coincidencia:

```
$ a='I am a string'
$ echo "${a/a/A}"
I Am a string
```

Todas las coincidencias:

```
$ echo "${a//a/A}"
I Am A string
```

Coincidencia al principio:

```
$ echo "${a/#I/y}"
y am a string
```

Coincidencia al final:

```
$ echo "${a/%g/N}"
I am a strinN
```

Sustituir un patrón por nada:

```
$ echo "${a/g/}"
I am a strin
```

Añadir prefijo a los elementos del array:

```
$ A=(hello world)
$ echo "${A[@]}/#/R}"
Rhello Rworld
```

Sección 15.4: Subcadenas de texto y subarreglos

```
var='0123456789abcdef'
```

```
# Definir un desplazamiento de origen cero
```

```
$ printf '%s\n' "${var:3}"
3456789abcdef
```

```
# Desplazamiento y longitud de la subcadena de texto
```

```
$ printf '%s\n' "${var:3:4}"
3456
```

```
Version ≥ 4.2
```

```
# La longitud negativa cuenta desde el final de la cadena de texto
$ printf '%s\n' "${var:3:-5}"
3456789ª

# El desplazamiento negativo cuenta desde el final
# Necesita un espacio para evitar confusión con ${var:-6}
$ printf '%s\n' "${var: -6}"
Abcdef

# Alternativa: paréntesis
$ printf '%s\n' "${var:(-6)}"
Abcdef

# Desplazamiento negativo y longitud negativa
$ printf '%s\n' "${var: -6:-5}"
a
```

Se aplican las mismas expansiones si el parámetro es un **parámetro posicional** o el **elemento de un array con subíndice**:

```
# Establecer parámetro posicional $1
set -- 0123456789abcdef

# Definir el desplazamiento
$ printf '%s\n' "${1:5}"
56789abcdef

# Asignar a elemento del array
myarr[0]='0123456789abcdef'

# Definir desplazamiento y longitud
$ printf '%s\n' "${myarr[0]:7:3}"
789
```

Se aplican expansiones análogas a los **parámetros posicionales**, en los que los desplazamientos se basan en uno:

```
# Establecer parámetros de posición $1, $2, ...
$ set -- 1 2 3 4 5 6 7 8 9 0 a b c d e f

# Definir un offset (cuidado $0 (no es un parámetro posicional)
# también se tiene en cuenta aquí)
$ printf '%s\n' "${@:10}"
0
a
b
c
d
e
f

# Definir un desplazamiento y una longitud
$ printf '%s\n' "${@:10:3}"
0
a
b

# No se permiten longitudes negativas para los parámetros de posición
$ printf '%s\n' "${@:10:-2}"
bash: -2: substring expression < 0
```

```
# El desplazamiento negativo cuenta desde el final
# Necesita un espacio para evitar confusión con ${@:-10:2}
$ printf '%s\n' "${@: -10:2}"
7
8

# ${@:0} es $0 que no es de otro modo un parámetro posicional o parte
# de $@
$ printf '%s\n' "${@:0:2}"
/usr/bin/bash
1
```

La expansión de subcadenas de texto puede utilizarse con **arrays indexadas**:

```
# Crear array (índices basados en cero)
$ myarr=(0 1 2 3 4 5 6 7 8 9 a b c d e f)

# Elementos con índice 5 y superior
$ printf '%s\n' "${myarr[@]:12}"
c
d
e
f

# 3 elementos, empezando por el índice 5
$ printf '%s\n' "${myarr[@]:5:3}"
5
6
7

# El último elemento del array
$ printf '%s\n' "${myarr[@]: -1}"
f
```

Sección 15.5: Borrar un patrón del principio de una cadena de caracteres

Coincidencia más corta:

```
$ a='I am a string'
$ echo "${a#a}"
m a string
```

Coincidencia más larga:

```
$ echo "${a##*a}"
string
```

Sección 15.6: Indirección de parámetros

La indirección Bash permite obtener el valor de una variable cuyo nombre está contenido en otra variable. Ejemplo de variables:

```
$ red="the color red"
$ green="the color green"

$ color=red
$ echo "${!color}"
the color red
$ color=green
$ echo "${!color}"
the color green
```

Algunos ejemplos más que demuestran el uso de la expansión indirecta:

```
$ foo=10
$ x=foo
$ echo ${x} # Impresión variable clásica
Foo
```

```
$ foo=10
$ x=foo
$ echo ${!x} # Expansión indirecta
10
```

Un ejemplo más:

```
$ argtester () { for (( i=1; i<="$#"; i++ )); do echo "${i}";done; }; argtester -ab -cd -ef
1 # i ampliado a 1
2 # i ampliado a 2
3 # i ampliado a 3
```

```
$ argtester () { for (( i=1; i<="$#"; i++ )); do echo "${!i}";done; }; argtester -ab -cd -ef
-ab # i=1 --> expandido a $1 ---> expandido al primer argumento enviado a la función
-cd # i=2 --> expandido a $2 ---> expandido al segundo argumento enviado a la función
-ef # i=3 --> expandido a $3 ---> expandido al tercer argumento enviado a la función
```

Sección 15.7: Expansión de parámetros y nombres de archivo

Puede utilizar la expansión de parámetros de Bash para emular operaciones comunes de procesamiento de nombres de archivo como **basename** y **dirname**.

Utilizaremos esta ruta como ejemplo:

```
FILENAME="/tmp/example/myfile.txt"
```

Para emular **dirname** y devolver el nombre de directorio de una ruta de archivo:

```
echo "${FILENAME%/*}"
# Salida: /tmp/example
```

Para emular **basename** **\$FILENAME** y devolver el nombre de archivo de una ruta de archivo:

```
echo "${FILENAME##*/}"
# Salida: myfile.txt
```

Para emular **basename** **\$FILENAME** .txt y devolver el nombre del archivo sin la extensión .txt:

```
BASENAME="${FILENAME##*/}"
echo "${BASENAME%.txt}"
# Salida: myfile
```

Sección 15.8: Sustitución de valores por defecto

\${parameter:-word}

Si el parámetro no está definido o es nulo, se sustituye por la expansión de la palabra. En caso contrario, se sustituye por el valor del parámetro.

```
$ unset var
$ echo "${var:-XX}" # Parámetro no ajustado -> expansión XX
XX
```

```
$ var="" # El parámetro es nulo -> se produce la expansión XX
$ echo "${var:-XX}"
XX
```

```
$ var=23 # El parámetro no es nulo -> se produce la expansión original
$ echo "${var:-XX}"
23
```

`${parameter:=word}`

Si el parámetro no está definido o es nulo, se le asigna la expansión de word. A continuación, se sustituye el valor del parámetro. Los parámetros posicionales y los parámetros especiales no pueden asignarse de esta forma.

```
$ unset var
$ echo "${var:=XX}" # Parámetro no ajustado -> palabra asignada a XX
XX
```

```
$ echo "$var"
XX
```

```
$ var="" # El parámetro es nulo -> la palabra se asigna a XX
$ echo "${var:=XX}"
XX
```

```
$ echo "$var"
XX
```

```
$ var=23 # Parámetro no nulo -> no se produce asignación
$ echo "${var:=XX}"
23
```

```
$ echo "$var"
23
```

Sección 15.9: Eliminar un patrón del final de una cadena de caracteres

Coincidencia más corta:

```
$ a='I am a string'
$ echo "${a%a*}"
I am
```

Coincidencia más larga:

```
$ echo "${a%%a*}"
I
```

Sección 15.10: Desprendimiento durante la expansión

Las variables no tienen que expandirse necesariamente a sus valores: las subcadenas de caracteres pueden extraerse durante la expansión, lo que puede resultar útil para extraer extensiones de archivo o partes de rutas. Los caracteres de expansión conservan su significado habitual, por lo que `.*` se refiere a un punto literal, seguido de cualquier secuencia de caracteres; no es una expresión regular.

```
$ v=foo-bar-baz
$ echo "${v%-*}"
foo
$ echo "${v%-*}"
foo-bar
$ echo "${v##*-}"
baz
$ echo "${v#*-}"
bar-baz
```

También es posible expandir una variable usando un valor por defecto - digamos que quiero invocar el editor del usuario, pero si no ha establecido uno me gustaría darle **vim**.

```
$ EDITOR=nano
$ ${EDITOR:-vim} /tmp/some_file
# abre nano
$ unset EDITOR
$ $ ${EDITOR:-vim} /tmp/some_file
# abre vim
```

Hay dos formas diferentes de realizar esta expansión, que difieren en si la variable relevante está vacía o no. Si se utiliza `:-` se utilizará el valor predeterminado si la variable está vacía o no está definida, mientras que `-` sólo utiliza el valor predeterminado si la variable no está definida, pero utilizará la variable si está definida como una cadena de texto vacía:

```
$ a="set"
$ b=""
$ unset c
$ echo ${a:-default_a} ${b:-default_b} ${c:-default_c}
set default_b default_c
$ echo ${a-default_a} ${b-default_b} ${c-default_c}
set default_c
```

De forma similar a los valores por defecto, se pueden dar alternativas; donde se utiliza un valor por defecto si una variable en particular no está disponible, se utiliza una alternativa si la variable está disponible.

```
$ a="set"
$ b=""
$ echo ${a:+alternative_a} ${b:+alternative_b}
alternative_a
```

Teniendo en cuenta que estas expansiones se pueden anidar, el uso de alternativas resulta especialmente útil a la hora de proporcionar argumentos a las opciones de la línea de comandos;

```
$ output_file=/tmp/foo
$ wget ${output_file:+"-o ${output_file}"} www.stackexchange.com
# se amplía a wget -o /tmp/foo www.stackexchange.com
$ unset output_file
$ wget ${output_file:+"-o ${output_file}"} www.stackexchange.com
# se amplía a wget www.stackexchange.com
```

Sección 15.11: Error si la variable está vacía o no está definida

La semántica de esta opción es similar a la de la sustitución de valores por defecto, pero en lugar de sustituir un valor por defecto, muestra el mensaje de error proporcionado. Las formas son `${VARNAME?ERRMSG}` y `${VARNAME:?ERRMSG}`. La forma con `:` dará error si la variable **no está definida** o está **vacía**, mientras que la forma sin `:` sólo dará error si la variable no está definida. Si se produce un error, se muestra `ERRMSG` y el código de salida se establece en `1`.

```
#!/bin/bash
FOO=
# ./script.sh: line 4: FOO: EMPTY
echo "FOO is ${FOO?EMPTY}"
# FOO es
echo "FOO is ${FOO?UNSET}"
# ./script.sh: line 8: BAR: EMPTY
echo "BAR is ${BAR?EMPTY}"
# ./script.sh: line 10: BAR: UNSET
echo "BAR is ${BAR?UNSET}"
```

Para ejecutar el ejemplo completo de arriba es necesario comentar cada una de las sentencias `echo` que dan error.

Capítulo 16: Copiar (cp)

Opción

-a, **-archive**
-b, **-backup**
-d, **--no-deference**
-f, **--force**
-i, **--interactive**
-l, **--link**
-p, **--preserve**
-R, **--recursive**

Descripción

Combina las opciones d, p y r
Antes de retirarlo, realiza una copia de seguridad
Conservar enlaces
Eliminar destinos existentes sin preguntar al usuario
Mostrar aviso antes de sobrescribir
En lugar de copiar, enlaza los archivos
Conservar los atributos de los archivos siempre que sea posible
Copiar directorios de forma recursiva

Sección 16.1: Copiar un solo archivo

Copiar `foo.txt` de `/path/to/source/` a `/path/to/target/folder/`

```
cp /path/to/source/foo.txt /path/to/target/folder/
```

Copie `foo.txt` de `/path/to/source/` a `/path/to/target/folder/` en un archivo llamado `bar.txt`

```
cp /path/to/source/foo.txt /path/to/target/folder/bar.txt
```

Sección 16.2: Copiar carpetas

Copiar carpeta `foo` en carpeta `bar`

```
cp -r /path/to/foo /path/to/bar
```

Si la carpeta `bar` existe antes de emitir el comando, entonces `foo` y su contenido se copiarán en la carpeta `bar`. Sin embargo, si `bar` no existe antes de emitir el comando, se creará la carpeta `bar` y el contenido de `foo` se colocará en `bar`.

Capítulo 17: Buscar (find)

find es un comando para buscar recursivamente en un directorio archivos (o directorios) que coincidan con un criterio y, a continuación, realizar alguna acción en los archivos seleccionados.

```
find search_path selection_criteria action
```

Sección 17.1: Buscar un archivo por nombre o extensión

Para buscar archivos/directorios con un nombre específico, relativo a **pwd**:

```
$ find . -name "myFile.txt"
./myFile.txt
```

Para buscar archivos/directorios con una extensión específica, utiliza un comodín:

```
$ find . -name "*.txt"
./myFile.txt
./myFile2.txt
```

Para buscar archivos/directorios que coincidan con una de varias extensiones, utiliza el indicador **or**:

```
$ find . -name "*.txt" -o -name "*.sh"
```

Para buscar archivos/directorios cuyo nombre comience por abc y termine con un carácter alfabético seguido de un dígito:

```
$ find . -name "abc[a-z][0-9]"
```

Para buscar todos los archivos/directorios situados en un directorio específico

```
$ find /opt
```

Para buscar sólo archivos (no directorios), utilice **-type f**:

```
find /opt -type f
```

Para buscar sólo directorios (no archivos normales), utilice **-type d**:

```
find /opt -type d
```

Sección 17.2: Ejecutar comandos en un archivo encontrado

A veces necesitaremos ejecutar comandos contra muchos archivos. Esto se puede hacer usando **xargs**.

```
find . -type d -print | xargs -r chmod 770
```

El comando anterior encontrará recursivamente todos los directorios (**-type d**) relativos a **.** (que es su directorio de trabajo actual), y ejecutará **chmod 770** en ellos. La opción **-r** especifica a **xargs** que no ejecute **chmod** si **find** no encontró ningún archivo.

Si los nombres de sus archivos o directorios contienen espacios, este comando puede bloquearse; una solución es utilizar lo siguiente.

```
find . -type d -print0 | xargs -r -0 chmod 770
```

En el ejemplo anterior, las banderas **-print0** y **-0** especifican que los nombres de archivo se separarán utilizando un byte **null**, y permite el uso de caracteres especiales, como espacios, en los nombres de archivo. Esta es una extensión GNU, y puede no funcionar en otras versiones de **find** y **xargs**.

La forma preferida de hacerlo es omitir el comando **xargs** y dejar que **find** llame al subproceso por sí mismo:

```
find . -type d -exec chmod 770 {} \;
```

Aquí, el `{}` es un marcador de posición que indica que desea utilizar el nombre del archivo en ese punto. `find` ejecutará `chmod` en cada archivo individualmente.

También puede pasar todos los nombres de archivo a una única llamada de `chmod`, utilizando:

```
find . -type d -exec chmod 770 {} +
```

Este es también el comportamiento de los fragmentos `xargs` anteriores. (Para llamar a cada archivo individualmente, puede utilizar `xargs -n1`).

Una tercera opción es dejar que bash haga un bucle sobre la lista de nombres de archivo para encontrar las salidas:

```
find . -type d | while read -r d; do chmod 770 "$d"; done
```

Esto es sintácticamente lo más torpe, pero conveniente cuando se desea ejecutar múltiples comandos en cada archivo encontrado. Sin embargo, esto es **inseguro** frente a nombres de archivo con nombres extraños

```
find . -type f | while read -r d; do mv "$d" "${d// /_}"; done
```

que sustituirá todos los espacios en los nombres de archivo por guiones bajos. (Este ejemplo tampoco funcionará si hay espacios en los nombres de directorio iniciales).

El problema con lo anterior es que, aunque `while read -r` espera una entrada por línea, los nombres de archivo pueden contener nuevas líneas (y además, `read -r` perderá cualquier espacio en blanco al final). Puede solucionarlo dando la vuelta a las cosas:

```
find . -type d -exec bash -c 'for f; do mv "$f" "${f// /_}"; done' _ {} +
```

De esta manera, el `-exec` recibe los nombres de archivo en una forma que es completamente correcta y portable; el `bash -c` los recibe como un número de argumentos, que se encontrarán en `$@`, correctamente entrecomillados, etc. (El script necesitará manejar estos nombres correctamente, por supuesto; cada variable que contenga un nombre de archivo necesita estar entre comillas dobles).

El misterioso `_` es necesario porque el primer argumento de `bash -c 'script'` se utiliza para rellenar `$0`.

Sección 17.3: Búsqueda de archivos por hora de acceso/modificación

En un sistema de archivos `ext`, cada archivo tiene almacenada una hora de acceso, modificación y cambio (de estado) asociada a él - para ver esta información se puede usar `stat myFile.txt`; usando banderas dentro de `find`, podemos buscar archivos que fueron modificados dentro de un cierto rango de tiempo.

Para buscar archivos que *hayan* sido modificados en las últimas 2 horas:

```
$ find . -mmin -120
```

Para buscar archivos que *no hayan* sido modificados en las últimas 2 horas:

```
$ find . -mmin +120
```

En el ejemplo anterior sólo se busca en la hora *modificada*; para buscar en las horas de **acceso** o en las horas **modificadas**, utilice `a` o `c` según corresponda.

```
$ find . -amin -120
```

```
$ find . -cmin +120
```

Formato general:

`-mmin n`: Fichero modificado hace *n* minutos

`-mmin -n`: Fichero modificado hace menos de *n* minutos

`-mmin +n`: Fichero modificado hace más de *n* minutos

Buscar archivos que *hayan* sido modificados en los últimos 2 días:

```
find . -mtime -2
```

Buscar archivos que *no hayan* sido modificados en los últimos 2 días

```
find . -mtime +2
```

Utilice `-atime` y `-ctime` para la hora de acceso y la hora de cambio de estado, respectivamente.

Formato general:

`-mtime n`: Fichero modificado hace *nx24* horas

`-mtime -n`: Fichero modificado hace menos de *nx24* horas

`-mtime +n`: Fichero modificado hace más de *nx24* horas

Buscar archivos modificados en un **rango de fechas**, de 2007-06-07 a 2007-06-08:

```
find . -type f -newermt 2007-06-07 ! -newermt 2007-06-08
```

Buscar archivos a los que se ha accedido en un **intervalo de marcas de tiempo** (utilizando los archivos como marca de tiempo), desde hace 1 hora hasta hace 10 minutos:

```
touch -t $(date -d '1 HOUR AGO' +%Y%m%d%H%M.%S) start_date
touch -t $(date -d '10 MINUTE AGO' +%Y%m%d%H%M.%S) end_date
timeout 10 find "$LOCAL_FOLDER" -newerat "start_date" ! -newerat "end_date" -print
```

Formato general:

`-newerXY reference`: Compara la marca de tiempo del archivo actual con la referencia. XY puede tener uno de los siguientes valores: at (tiempo de acceso), mt (tiempo de modificación), ct (tiempo de cambio) y más. `reference` es el *nombre de un archivo* con el que se desea comparar la marca de tiempo especificada (acceso, modificación, cambio) o una cadena que describa un tiempo absoluto.

Sección 17.4: Búsqueda de archivos por tamaño

Buscar archivos de más de 15 MB:

```
find -type f -size +15M
```

Buscar archivos de menos de 12 KB:

```
find -type f -size -12k
```

Buscar archivos de un tamaño exacto de 12 KB:

```
find -type f -size 12k
```

O

```
find -type f -size 12288c
```

O

```
find -type f -size 24b
```

O

```
find -type f -size 24
```

Formato general:

```
find [options] -size n[cwbkMG]
```

Buscar archivos de tamaño n-bloques, donde +n significa más de n-bloques, -n significa menos de n-bloques y n (sin signo) significa exactamente n-bloques.

Tamaño del bloque:

1. c: bytes
2. w: 2 bytes
3. b: 512 bytes (por defecto)
4. k: 1 KB
5. M: 1 MB
6. G: 1 GB

Sección 17.5: Filtrar la ruta

El parámetro `-path` permite especificar un patrón para que coincida con la ruta del resultado. El patrón puede coincidir también con el propio nombre.

Para encontrar sólo archivos que contengan `log` en cualquier parte de su ruta (carpeta o nombre):

```
find . -type f -path '*log*'
```

Para encontrar sólo archivos dentro de una carpeta llamada `log` (en cualquier nivel):

```
find . -type f -path '*/log/*'
```

Para encontrar sólo archivos dentro de una carpeta llamada `log` o `data`:

```
find . -type f -path '*/log/*' -o -path '*/data/*'
```

Para encontrar todos los archivos **excepto** los contenidos en una carpeta llamada `bin`:

```
find . -type f -not -path '*/bin/*'
```

Para encontrar todos los archivos **excepto** los contenidos en una carpeta llamada `bin` o `log` de archivos:

```
find . -type f -not -path '*log' -not -path '*/bin/*'
```

Sección 17.6: Búsqueda de archivos por tipo

Para buscar archivos, utilice la opción `-type f`

```
$ find . -type f
```

Para buscar directorios, utilice el indicador `-type d`

```
$ find . -type d
```

Para buscar dispositivos de bloque, utilice el indicador `-type b`

```
$ find /dev -type b
```

Para encontrar enlaces simbólicos, utilice el indicador `-type l`

```
$ find . -type l
```

Sección 17.7: Búsqueda de archivos por extensión

Para encontrar todos los archivos de una determinada extensión dentro de la ruta actual puede utilizar la siguiente sintaxis `find`. Funciona haciendo uso de la construcción `glob` incorporada en `bash` para buscar todos los nombres que tengan la extensión `.extension`.

```
find /directory/to/search -maxdepth 1 -type f -name "*.extension"
```

Para buscar todos los archivos de tipo `.txt` sólo en el directorio actual, haga lo siguiente

```
find . -maxdepth 1 -type f -name "*.txt"
```

Capítulo 18: Usar sort

Opción	Significado
<code>-u</code>	Hacer que cada línea de salida sea única

`sort` es un comando de Unix que ordena los datos de un archivo en una secuencia.

Sección 18.1: Ordenar la salida del comando

El comando `sort` se utiliza para ordenar una lista de líneas.

Entrada desde un archivo

```
sort file.txt
```

Entrada de un comando

Puede ordenar cualquier comando de salida. En el ejemplo una lista de archivo siguiendo un patrón.

```
find * -name pattern | sort
```

Sección 18.2: Hacer que el producto sea único

Si es necesario que cada línea de la salida sea única, añada la opción `-u`.

Para mostrar el propietario de los archivos de la carpeta

```
ls -l | awk '{print $3}' | sort -u
```

Sección 18.3: Ordenación numérica

Supongamos que tenemos este archivo:

```
test>>cat file
10.Gryffindor
4.Hogwarts
2.Harry
3.Dumbledore
1.The sorting hat
```

Para ordenar numéricamente este fichero, utilice `sort` con la opción `-n`:

```
test>>sort -n file
```

Esto debería ordenar el archivo como se indica a continuación:

```
1.The sorting hat
2.Harry
3.Dumbledore
4.Hogwarts
10.Gryffindor
```

Invertir el orden de clasificación: Para invertir el orden de la clasificación utilice la opción `-r`.

Para invertir el orden de clasificación del archivo anterior utilice:

```
sort -rn file
```

Esto debería ordenar el archivo como se indica a continuación:

```
10.Gryffindor
4.Hogwarts
3.Dumbledore
2.Harry
1.The sorting hat
```

Sección 18.4: Ordenar por claves

Supongamos que tenemos este archivo:

```
test>>cat Hogwarts
Harry      Malfoy      Rowena      Helga
Gryffindor  Slytherin   Ravenclaw   Hufflepuff
Hermione    Goyle       Lockhart    Tonks
Ron         Snape       Olivander   Newt
Ron         Goyle       Flitwick    Sprout
```

Para ordenar este fichero utilizando una columna como clave utilice la opción **-k**:

```
test>>sort -k 2 Hogwarts
```

Esto ordenará el archivo con la columna 2 como clave:

```
Ron         Goyle       Flitwick    Sprout
Hermione    Goyle       Lockhart    Tonks
Harry      Malfoy      Rowena      Helga
Gryffindor  Slytherin   Ravenclaw   Hufflepuff
Ron         Snape       Olivander   Newt
```

Ahora bien, si tenemos que ordenar el fichero con una clave secundaria junto con la clave primaria utilizar:

```
sort -k 2,2 -k 1,1 Hogwarts
```

Esto ordenará primero el archivo con la columna 2 como clave primaria y, a continuación, ordenará el archivo con la columna 1 como clave secundaria:

```
Hermione    Goyle       Lockhart    Tonks
Ron         Goyle       Flitwick    Sprout
Harry      Malfoy      Rowena      Helga
Gryffindor  Slytherin   Ravenclaw   Hufflepuff
Ron         Snape       Olivander   Newt
```

Si necesitamos ordenar un fichero con más de 1 clave, entonces para cada opción **-k** necesitamos especificar dónde termina la ordenación. Así **-k 1, 1** significa que la ordenación comienza en la primera columna y termina en la primera columna.

Opción **-t**

En el ejemplo anterior el fichero tenía el delimitador por defecto - tabulador. En el caso de ordenar un fichero que tiene un delimitador no predeterminado necesitamos la opción **-t** para especificar el delimitador.

Supongamos que tenemos el archivo como a continuación:

```
test>>cat file
```

```
5.|Gryffindor
4.|Hogwarts
2.|Harry
3.|Dumbledore
1.|The sorting hat
```

Para ordenar este archivo según la segunda columna, utiliza:

```
test>>sort -t "|" -k 2 file
```

Esto ordenará el archivo como se indica a continuación:

```
3.|Dumbledore  
5.|Gryffindor  
2.|Harry  
4.|Hogwarts  
1.|The sorting hat
```

Capítulo 19: Búsqueda de fuentes

Sección 19.1: Obtener un archivo

La obtención de un archivo es diferente de la ejecución, en que todos los comandos son evaluados dentro del contexto de la sesión bash actual - esto significa que cualquier variable, función o alias definidos persistirán a lo largo de su sesión.

Crear el archivo que desea fuente `sourceme.sh`

```
#!/bin/bash
```

```
export A="hello_world"
alias sayHi="echo Hi"
sayHello() {
    echo Hello
}
```

Desde su sesión, obtenga el archivo

```
$ source sourceme.sh
```

A partir de aquí, dispones de todos los recursos del fichero de origen

```
$ echo $A
hello_world
```

```
$ sayHi
Hi
```

```
$ sayHello
Hello
```

Tenga en cuenta que el comando `.` es sinónimo de `source`, de modo que puede utilizar simplemente

```
$ . sourceme.sh
```

Sección 19.2: Búsqueda de un entorno virtual

Cuando se desarrollan varias aplicaciones en una misma máquina, resulta útil separar las dependencias en entornos virtuales.

Con el uso de `virtualenv`, estos entornos son de origen en su shell de modo que cuando se ejecuta un comando, se trata de ese entorno virtual.

Lo más habitual es instalarlo mediante `pip`.

```
pip install https://github.com/pypa/virtualenv/tarball/15.0.2
```

Crear un nuevo entorno

```
virtualenv --python=python3.5 my_env
```

Activar el entorno

```
source my_env/bin/actívale
```


Capítulo 20: Here documents y here strings

Sección 20.1: Ejecutar comando con here document

```
ssh -p 21 example@example.com <<EOF
echo 'printing pwd'
echo "\$(pwd)"
ls -a
find '*.txt'
EOF
```

\$ se escapa porque no queremos que sea expandido por la shell actual, es decir, `$(pwd)` se ejecutará en la shell remota.

De otra manera:

```
ssh -p 21 example@example.com <<'EOF'
echo 'printing pwd'
echo "$ (pwd)"
ls -a
find '*.txt'
EOF
```

Nota: El EOF de cierre **debe** estar al principio de la línea (sin espacios en blanco antes). Si se requiere sangría, pueden usarse tabuladores si comienza su heredoc con `<<-`. Consulta los ejemplos de [Documentos con sangría](#) y [Cadenas de caracteres con límite](#) para obtener más información.

Sección 20.2: Documentos con sangría

Puede sangrar el texto dentro de aquí documentos con tabuladores, es necesario utilizar el operador de redirección `<<-` en lugar de `<<`:

```
$ cat <<- EOF
  This is some content indented with tabs `t`.
  You cannot indent with spaces you __have__ to use tabs.
  Bash will remove empty space before these lines.
  __Note__: Be sure to replace spaces with tabs when copying this example.
EOF
```

```
This is some content indented with tabs `t`.
You cannot indent with spaces you __have__ to use tabs.
Bash will remove empty space before these lines.
__Note__: Be sure to replace spaces with tabs when copying this example.
```

Un caso práctico de esto (como se menciona en [man bash](#)) es en los scripts de shell, por ejemplo:

```
if cond; then
  cat <<- EOF
  hello
  there
  EOF
fi
```

Es habitual sangrar las líneas dentro de los bloques de código como en esta sentencia `if`, para una mejor legibilidad. Sin la sintaxis del operador `<<-`, nos veríamos obligados a escribir el código anterior así:

```
if cond; then
  cat << EOF
hello
there
EOF
fi
```

Eso es muy desagradable de leer, y empeora mucho en un guion realista más complejo.

Sección 20.3: Crear un archivo

Un uso clásico de los documentos aquí es crear un archivo escribiendo su contenido:

```
cat > fruits.txt << EOF
apple
orange
lemon
EOF
```

El here-document son las líneas entre el `<< EOF` y el `EOF`.

Este here document se convierte en la entrada del comando `cat`. El comando `cat` simplemente da salida a su entrada, y usando el operador de redirección de salida `>` redirigimos a un archivo `fruits.txt`.

Como resultado, el archivo `fruits.txt` contendrá las líneas:

```
apple
orange
lemon
```

Se aplican las reglas habituales de redirección de salida: si `fruits.txt` no existía antes, se creará. Si ya existía, se truncará.

Sección 20.4: Here strings

Version \geq 2.05b

Puedes alimentar un comando usando here strings como esta:

```
$ awk '{print $2}' <<< "hello world - how are you?"
World

$ awk '{print $1}' <<< "hello how are you
> she is fine"
hello
she
```

También puedes alimentar un bucle `while` con un here string:

```
$ while IFS=" " read -r word1 word2 rest
> do
> echo "$word1"
> done <<< "hello how are you - i am fine"
hello
```

Sección 20.5: Ejecutar varios comandos con sudo

```
sudo -s <<EOF
a='var'
echo 'Running several commands with sudo'
mktemp -d
echo "\$a"
EOF
```

- `$a` necesita ser escapado para evitar que sea expandido por el shell actual

O

```
sudo -s <<'EOF'
a='var'
echo 'Running serveral commands with sudo'
mktemp -d
echo "$a"
EOF
```

Sección 20.6: Cadenas de caracteres con límite

Un heredoc utiliza la cadena de caracteres con límite (*limitstring*) para determinar cuándo dejar de consumir entrada. El limitstring de terminación **debe**:

- Estar al principio de una fila.
- Ser el único texto de la línea **Nota:** Si utiliza <<- el limitstring puede ir precedida de tabuladores \t

Correcto:

```
cat <<limitstring
line 1
line 2
limitstring
```

Esto dará como resultado:

```
line 1
line 2
```

Uso incorrecto:

```
cat <<limitstring
line 1
line 2
    limitstring
```

Dado que `limitstring` en la última línea no está exactamente al principio de la línea, el intérprete de comandos continuará esperando más entradas, hasta que vea una línea que empiece con `limitstring` y no contenga nada más. Sólo entonces dejará de esperar la entrada, y procederá a pasar el here-document al comando `cat`.

Tenga en cuenta que cuando antepone un guión a el limitstring inicial, las tabulaciones al principio de la línea se eliminan antes del análisis sintáctico, por lo que los datos y el limitstring pueden estar sangrados con tabulaciones (para facilitar la lectura en scripts de shell).

```
cat <<-limitstring
    line 1 has a tab each before the words line and has
        line 2 has two leading tabs
    limitstring
```

producirá

```
line 1      has a tab each before the words line and has      line 2 has two leading tabs
```

sin las pestañas iniciales (pero no las internas).

Capítulo 21: Citar

Sección 21.1: Comillas dobles para la sustitución de variables y comandos

Las sustituciones de variables sólo deben utilizarse dentro de comillas dobles.

```
calculation='2 * 3'
echo "$calculation" # imprime 2 * 3
echo $calculation # imprime 2, la lista de archivos del directorio actual, y 3
echo "$(($calculation))" # imprime 6
```

Fuera de las comillas dobles, `$var` toma el valor de `var`, lo divide en partes delimitadas por espacios en blanco e interpreta cada parte como un patrón glob (comodín). A menos que desee este comportamiento, ponga siempre `$var` entre comillas dobles: `"$var"`.

Lo mismo ocurre con las sustituciones de comandos: `"$(mycommand)"` es la salida de `mycommand`, `$(mycommand)` es el resultado de `split+glob` en la salida.

```
echo "$var" # bien
echo "$(mycommand)" # bien
another=$var # también funciona, la asignación está implícitamente entre comillas dobles
make -D THING=$var # ¡MAL! Esto no es una asignación de bash.
make -D THING="$var" # bien
make -D "THING=$var" # también Bien
```

Las sustituciones de comandos tienen sus propios contextos de entrecomillado. Escribir sustituciones anidadas arbitrariamente es fácil porque el analizador mantiene un registro de la profundidad del anidamiento en lugar de buscar ávidamente el primer carácter `"`. Sin embargo, el resaltador de sintaxis de StackOverflow analiza esto incorrectamente. Por ejemplo:

```
echo "formatted text: $(printf "a + b = %04d" "${c}")" # "formatted text: a + b = 0000"
```

Los argumentos variables de una sustitución de comandos también deben ir entre comillas dobles dentro de las expansiones:

```
echo "$(mycommand "$arg1" "$arg2")"
```

Sección 21.2: Diferencia entre comillas dobles y simples

Doble comilla

- Permite una expansión variable
- Permite ampliar el historial si está activado
- Permite sustituir comandos
- * y @ pueden tener un significado especial
- Puede contener comillas simples o dobles

`$`, ```, `"`, `\` pueden escaparse con `\` para evitar su significado especial

Comilla simple

- Evita la expansión variable
- Evita la expansión del historial
- Evita la sustitución de comandos
- * y @ son siempre literales
- No se admiten comillas simples dentro de comillas simples
- Todos ellos son literales

Propiedades comunes a ambos:

- Evita el globbing
- Evita la división de palabras

Ejemplos:

```
$ echo "!cat"
echo "cat file"
cat file
$ echo '!cat'
!cat
echo "\"'\\"
""
$ a='var'
$ echo '$a'
$a
$ echo "$a"
var
```

Sección 21.3: Saltos de línea y caracteres de control

Se puede incluir una nueva línea en una cadena entre comillas simples o dobles. Tenga en cuenta que la barra invertida-nueva línea no produce una nueva línea, el salto de línea se ignora.

```
newline1='
'
newline2="
"
newline3=$'\n'
empty=\

echo "Line${newline1}break"
echo "Line${newline2}break"
echo "Line${newline3}break"
echo "No line break${empty} here"
```

Dentro de las cadenas entre comillas y dólares, se puede utilizar barra invertida-letra o barra invertida-octal para insertar caracteres de control, como en muchos otros lenguajes de programación.

```
echo $'Tab: [\t]'
echo $'Tab again: [\009]'
echo $'Form feed: [\f]'
echo $'Line\nbreak'
```

Sección 21.4: Citar texto literal

Todos los ejemplos de este párrafo imprimen la línea

```
!"#$%&'()*+,-./:;<=>? @[\]^_`{|}~
```

Una barra invertida entrecomilla el carácter siguiente, es decir, el carácter siguiente se interpreta literalmente. La única excepción es una nueva línea: barra invertida-nueva línea se expande a la cadena de caracteres vacía.

```
echo \!\"\\#\\$\\&\\'\\(\\)\\*\\;\\<\\|=\\>\\?\\ \\|@\\[\\]\\]\\^\\`\\{\\|\\}\\|~
```

Todo el texto entre comillas simples (comillas simples `'`, también conocidas como apóstrofe) se imprime literalmente. Incluso la barra invertida se imprime sola, y es imposible incluir una comilla simple; en su lugar, puede detener la cadena de caracteres literal, incluir una comilla simple literal con una barra invertida y volver a iniciar la cadena de caracteres literal. Así, la secuencia de 4 caracteres `'\\''` permite efectivamente incluir una comilla simple en una cadena de caracteres literal.

```
echo '!\"#$%&'\\'()*+,-./:;<=>? @[\]^_`{|}~'
# ^^^^
```

La comilla simple dólar inicia una cadena de caracteres literal `$'...'` como muchos otros lenguajes de programación, donde la barra invertida entrecomilla el carácter siguiente.

```
echo $'!"#$%&'()*+,-./:;<=>? @[\]^_`{|}~'  
# ^^ ^^
```

Las comillas dobles `"` delimitan cadenas semiliterales en las que sólo los caracteres `" \ $ y `` conservan su significado especial. Estos caracteres necesitan una barra invertida delante (tenga en cuenta que si la barra invertida va seguida de algún otro carácter, la barra invertida permanece). Las comillas dobles son útiles sobre todo cuando se incluye una variable o una sustitución de comando.

```
echo "!\"#$%&'()*+,-./:;<=>? @[\]^_`{|}~"  
# ^^ ^^ ^^  
echo "!\"#$%&'()*+,-./:;<=>? @[\]^_`{|}~"  
# ^^ ^ ^^ \[ imprime \[
```

Interactivamente, ten cuidado con que `!` activa la expansión del historial dentro de las comillas dobles: `"!oops"` busca un comando anterior que contenga `oops`; `"\!oops"` no expande el historial pero mantiene la barra invertida. Esto no ocurre en los scripts.

Capítulo 22: Expresiones condicionales

Sección 22.1: Pruebas de tipo de archivo

El operador condicional `-e` comprueba si existe un archivo (incluidos todos los tipos de archivo: directorios, etc.).

```
if [[ -e $filename ]]; then
    echo "$filename exists"
fi
```

También hay pruebas para tipos de archivo específicos.

```
if [[ -f $filename ]]; then
    echo "$filename is a regular file"
elif [[ -d $filename ]]; then
    echo "$filename is a directory"
elif [[ -p $filename ]]; then
    echo "$filename is a named pipe"
elif [[ -S $filename ]]; then
    echo "$filename is a named socket"
elif [[ -b $filename ]]; then
    echo "$filename is a block device"
elif [[ -c $filename ]]; then
    echo "$filename is a character device"
fi
if [[ -L $filename ]]; then
    echo "$filename is a symbolic link (to any file type)"
fi
```

Para un enlace simbólico, aparte de `-L`, estas pruebas se aplican al destino, y devuelven false para un enlace roto.

```
if [[ -L $filename || -e $filename ]]; then
    echo "$filename exists (but may be a broken symbolic link)"
fi
if [[ -L $filename && ! -e $filename ]]; then
    echo "$filename is a broken symbolic link"
fi
```

Sección 22.2: Comparación y correspondencia de cadenas de caracteres

La comparación de cadenas de caracteres utiliza el operador `==` entre cadenas de caracteres entrecomilladas. El operador `!=` niega la comparación.

```
if [[ "$string1" == "$string2" ]]; then
    echo "\$string1 and \$string2 are identical"
fi
if [[ "$string1" != "$string2" ]]; then
    echo "\$string1 and \$string2 are not identical"
fi
```

Si el lado derecho no está entre comillas, se trata de un patrón comodín con el que se compara `$string`.

```
string='abc'
pattern1='a*'
pattern2='x*'
if [[ "$string" == $pattern1 ]]; then
    # el test es verdadero
    echo "The string $string matches the pattern $pattern"
fi
if [[ "$string" != $pattern2 ]]; then
    # el test es false
    echo "The string $string does not match the pattern $pattern"
fi
```

Los operadores `<` y `>` comparan las cadenas de caracteres en orden lexicográfico (no existen los operadores menor-o-igual o mayor-o-igual para cadenas de caracteres).

Existen pruebas unarias para la cadena de caracteres vacía.

```
if [[ -n "$string" ]]; then
    echo "$string is non-empty"
fi
if [[ -z "${string// }" ]]; then
    echo "$string is empty or contains only spaces"
fi
if [[ -z "$string" ]]; then
    echo "$string is empty"
fi
```

Arriba, la comprobación `-z` puede significar que `$string` no está establecida, o que está establecida a una cadena de caracteres vacía. Para distinguir entre vacío y no establecido, utilice:

```
if [[ -n "${string+x}" ]]; then
    echo "$string is set, possibly to the empty string"
fi
if [[ -n "${string-x}" ]]; then
    echo "$string is either unset or set to a non-empty string"
fi
if [[ -z "${string+x}" ]]; then
    echo "$string is unset"
fi
if [[ -z "${string-x}" ]]; then
    echo "$string is set to an empty string"
fi
```

donde `x` es arbitraria. O en forma de tabla:

+-----+-----+-----+-----+				
\$string is: unset empty non-empty				
+-----+-----+-----+-----+				
[[-z \${string}]]	true	true	false	
[[-z \${string+x}]]	true	false	false	
[[-z \${string-x}]]	false	true	false	
[[-n \${string}]]	false	false	true	
[[-n \${string+x}]]	false	true	true	
[[-n \${string-x}]]	true	false	true	
+-----+-----+-----+-----+				

Alternativamente, el estado se puede comprobar en una sentencia `case`:

```
case ${var+x$var} in
    (x) echo empty;;
    ("") echo unset;;
    (x*[:blank:]*) echo non-blank;;
    (*) echo blank
esac
```


Donde `[:blank:]` son caracteres de espaciado horizontal específicos de la configuración regional (tabulador, espacio, etc.).

Sección 22.3: Comprobar el estado de salida de un comando

Estado de salida 0: éxito

Estado de salida distinto de 0: fallo

Para comprobar el estado de salida de un comando:

```
if command; then
    echo 'success'
else
    echo 'failure'
fi
```

Sección 22.4: Prueba de una línea

Puedes hacer cosas como esta:

```
[[ $s = 'something' ]] && echo 'matched' || echo "didn't match"
[[ $s == 'something' ]] && echo 'matched' || echo "didn't match"
[[ $s != 'something' ]] && echo "didn't match" || echo "matched"
[[ $s -eq 10 ]] && echo 'equal' || echo "not equal"
(( $s == 10 )) && echo 'equal' || echo 'not equal'
```

Una prueba de línea para el estado de salida:

```
command && echo 'exited with 0' || echo 'non 0 exit'
cmd && cmd1 && echo 'previous cmds were successful' || echo 'one of them failed'
cmd || cmd1 # Si cmd falla prueba cmd1
```

Sección 22.5: Comparación de archivos

```
if [[ $file1 -ef $file2 ]]; then
    echo "$file1 and $file2 are the same file"
fi
```

“Mismo archivo” significa que la modificación de uno de los archivos en su lugar afecta al otro. Dos ficheros pueden ser el mismo, aunque tengan nombres diferentes, por ejemplo, si son enlaces duros, o si son enlaces simbólicos con el mismo destino, o si uno es un enlace simbólico que apunta al otro.

Si dos ficheros tienen el mismo contenido, pero son distintos (de modo que la modificación de uno no afecta al otro), entonces `-ef` informa de que son diferentes. Si desea comparar dos ficheros byte a byte, utilice la utilidad `cmp`.

```
if cmp -s -- "$file1" "$file2"; then
    echo "$file1 and $file2 have identical contents"
else
    echo "$file1 and $file2 differ"
fi
```

Para obtener una lista legible de diferencias entre archivos de texto, utilice la utilidad `diff`.

```
if diff -u "$file1" "$file2"; then
    echo "$file1 and $file2 have identical contents"
else
    : # se han enumerado las diferencias entre los ficheros
fi
```

Sección 22.6: Pruebas de acceso a archivos

```
if [[ -r $filename ]]; then
    echo "$filename is a readable file"
fi
if [[ -w $filename ]]; then
    echo "$filename is a writable file"
fi
if [[ -x $filename ]]; then
    echo "$filename is an executable file"
fi
```

Estas pruebas tienen en cuenta los permisos y la propiedad para determinar si el script (o los programas lanzados desde el script) pueden acceder al archivo.

Cuidado con las condiciones de carrera (TOCTOU): que la prueba tenga éxito ahora no significa que siga siendo válida en la línea siguiente. Por lo general, es mejor intentar acceder a un archivo, y manejar el error, en lugar de probar primero y luego tener que manejar el error de todos modos en caso de que el archivo haya cambiado en el ínterin.

Sección 22.7: Comparaciones numéricas

Las comparaciones numéricas utilizan los operadores `-eq` y sus amigos

```
if [[ $num1 -eq $num2 ]]; then
    echo "$num1 == $num2"
fi
if [[ $num1 -le $num2 ]]; then
    echo "$num1 <= $num2"
fi
```

Existen seis operadores numéricos:

- `-eq` igual
- `-ne` no igual
- `-le` menor o igual
- `-lt` menor que
- `-ge` mayor o igual que
- `-gt` mayor que

Tenga en cuenta que los operadores `<` y `>` dentro de `[[...]]` comparan cadenas de caracteres, no números.

```
if [[ 9 -lt 10 ]]; then
    echo "9 is before 10 in numeric order"
fi
if [[ 9 > 10 ]]; then
    echo "9 is after 10 in lexicographic order"
fi
```

Los dos lados deben ser números escritos en decimal (o en octal con un cero a la izquierda). Como alternativa, utilice la sintaxis de expresión aritmética `((...))`, que realiza cálculos con **números enteros** en una sintaxis similar a la de C/Java/....

```
x=2
if ((2*x == 4)); then
    echo "2 times 2 is 4"
fi
((x += 1))
echo "2 plus 1 is $x"
```

Capítulo 23: Scripting con parámetros

Sección 23.1: Análisis de múltiples parámetros

Para analizar muchos parámetros, la forma preferida de hacerlo es utilizando un bucle **while**, una sentencia **case** y **shift**.

shift se utiliza para saltar el primer parámetro de la serie, haciendo que lo que antes eran **\$2**, ahora sean **\$1**. Esto es útil para procesar argumentos de uno en uno.

```
#!/bin/bash
# Cargar los parámetros definidos por el usuario
while [[ $# > 0 ]]
do
    case "$1" in
        -a|--valueA)
            valA="$2"
            shift
            ;;
        -b|--valueB)
            valB="$2"
            shift
            ;;
        --help|*)
            echo "Usage:"
            echo "  --valueA \"value\""
            echo "  --valueB \"value\""
            echo "  --help"
            exit 1
            ;;
    esac
    shift
done

echo "A: $valA"
echo "B: $valB"
```

Entradas y salidas

```
$ ./multipleParams.sh --help
Usage:
  --valueA "value"
  --valueB "value"
  --help
```

```
$ ./multipleParams.sh
A:
B:
```

```
$ ./multipleParams.sh --valueB 2
A:
B: 2
```

```
$ ./multipleParams.sh --valueB 2 --valueA "hello world"
A: hello world
B: 2
```

Sección 23.2: Análisis de argumentos mediante un bucle for

Un ejemplo sencillo que ofrece las opciones:

Opc	Opc. Alt.	Detalles
-h	--help	Mostrar ayuda
-v	--version	Mostrar información de la versión
-dr path	--doc-root path	Una opción que toma un parámetro secundario (una ruta)
-i	--install	Una opción booleana (verdadero/falso)
-*	--	Opción no válida

```
#!/bin/bash
dr=''
install=false

skip=false
for op in "$@";do
    if $skip;then skip=false;continue;fi
    case "$op" in
        -v|--version)
            echo "$ver_info"
            shift
            exit 0
            ;;
        -h|--help)
            echo "$help"
            shift
            exit 0
            ;;
        -dr|--doc-root)
            shift
            if [[ "$1" != "" ]]; then
                dr="${1%/\\}"
                shift
                skip=true
            else
                echo "E: Arg missing for -dr option"
                exit 1
            fi
            ;;
        -i|--install)
            install=true
            shift
            ;;
        -*)
            echo "E: Invalid option: $1"
            shift
            exit 1
            ;;
    esac
done
```

Sección 23.3: Script envolvente (Wrapper script)

Wrapper script es un script que envuelve otro script o comando para proporcionar funcionalidades extra o simplemente para hacer algo menos tedioso.

Por ejemplo, el **egrep** actual en el nuevo sistema GNU/Linux está siendo reemplazado por un script envoltorio llamado **egrep**. Así es como se ve:

```
#!/bin/sh
exec grep -E "$@"
```

Así, cuando se ejecuta `egrep` en tales sistemas, en realidad se está ejecutando `grep -E` con todos los argumentos reenviados.

En caso general, si quieres ejecutar un script/comando `exmp` de ejemplo con otro script `mexmp` entonces el script `mexmp` envoltorio tendrá el siguiente aspecto:

```
#!/bin/sh
exmp "$@" # Añadir otras opciones antes de "$@"
# o
# full/path/to/exmp "$@"
```

Sección 23.4: Acceso a los parámetros

Cuando se ejecuta un script Bash, los parámetros pasados al script se nombran de acuerdo con su posición: `$1` es el nombre del primer parámetro, `$2` es el nombre del segundo parámetro, y así sucesivamente.

Un parámetro ausente simplemente se evalúa como una cadena de caracteres vacía. La comprobación de la existencia de un parámetro puede hacerse de la siguiente manera:

```
if [ -z "$1" ]; then
    echo "No argument supplied"
fi
```

Obtener todos los parámetros

`$@` y `$*` son formas de interactuar con todos los parámetros del script. Haciendo referencia a la [página man de Bash](#), vemos que:

- `$*`: Expande a los parámetros posicionales, empezando por uno. Cuando la expansión se produce entre comillas dobles, se expande a una sola palabra con el valor de cada parámetro separado por el primer carácter de la variable especial IFS.
- `$@`: Expande a los parámetros posicionales, empezando por uno. Cuando la expansión se produce entre comillas dobles, cada parámetro se expande a una palabra separada.

Obtener el número de parámetros

`$#` obtiene el número de parámetros pasados a un script. Un caso de uso típico sería comprobar si se ha pasado el número apropiado de argumentos:

```
if [ $# -eq 0 ]; then
    echo "No arguments supplied"
fi
```

Ejemplo 1

Recorre todos los argumentos y comprueba si son archivos:

```
for item in "$@"
do
    if [[ -f $item ]]; then
        echo "$item is a file"
    fi
done
```

Ejemplo 2

Recorre todos los argumentos y comprueba si son archivos:

```
for (( i = 1; i <= $#; ++ i ))
do
    item=${@:$i:1}
    if [[ -f $item ]]; then
        echo "$item is a file"
    fi
done
```

Sección 23.5: Dividir cadena de caracteres en un array en Bash

Digamos que tenemos un parámetro String y queremos dividirlo por comas

```
my_param="foo,bar,bash"
```

Para dividir esta cadena de caracteres por comas podemos utilizar;

```
IFS=',' read -r -a array <<< "$my_param"
```

Aquí, IFS es una variable especial llamada Separador interno de campo que define el carácter o caracteres utilizados para separar un patrón en tokens para algunas operaciones.

Para acceder a un elemento individual:

```
echo "${array[0]}"
```

Para iterar sobre los elementos:

```
for element in "${array[@]}"
do
    echo "$element"
done
```

Para obtener tanto el índice como el valor:

```
for index in "${!array[@]}"
do
    echo "$index ${array[index]}"
done
```

Capítulo 24: Sustituciones del historial de Bash

Sección 24.1: Referencia rápida

Interacción con el historial

Listar todos los comandos anteriores

history

Borrar el historial, útil si has introducido una contraseña por accidente.

history -c

Designadores de eventos

Se expande a la línea n del historial de bash

!n

Se amplía hasta el último comando

!!

Se expande hasta el último comando que empieza por "text"

!text

Se expande hasta el último comando que contiene "text"

!?text

Se expande al comando n líneas atrás

!-n

Se expande al último comando con la primera aparición de "foo" sustituida por "bar".

^foo^bar^

Se amplía al comando actual

!#

Designadores de palabras

Se separan con **:** del designador del evento al que se refieren. Los dos puntos pueden omitirse si la palabra designadora no empieza por un número: **!^** es lo mismo que **!:^**.

Se expande al primer argumento del comando más reciente

!^

*# Se expande hasta el último argumento del comando más reciente (abreviatura de **!:\$**)*

!\$

Se expande hasta el tercer argumento del comando más reciente

!:3

Se expande a los argumentos x a y (inclusive) del último comando.

x e y pueden ser números o los caracteres de anclaje ^ \$

!:x-y

Se expande a todas las palabras del último comando excepto la 0ª

*# Equivalente a **!:^-\$***

!*

Modificadores

Modifican el evento precedente o el designador de palabra.

```
# Sustitución en la expansión utilizando la sintaxis sed
# Permite banderas antes de la s y separadores alternativos
:s/foo/bar/ # sustituye bar por la primera aparición de foo
:gs|foo|bar| # sustituye todos los foo por bar
```

```
# Eliminar la ruta inicial del último argumento ("cola")
:t
```

```
# Elimina la ruta final del último argumento ("head")
:h
```

```
# Eliminar la extensión de archivo del último argumento
:r
```

Si la variable HISTCONTROL de Bash contiene ignorespace o ignoreboth (o, alternatively, HISTIGNORE contiene el patrón []*), puede evitar que sus comandos se almacenen en el historial de Bash anteponiéndoles un espacio:

```
# Este comando no se guardará en el historial
foo
```

```
# Este comando se guardará
bar
```

Sección 24.2: Repetir el comando anterior con sudo

```
$ apt-get install r-base
E: Could not open lock file /var/lib/dpkg/lock - open (13: Permission denied)
E: Unable to lock the administration directory (/var/lib/dpkg/), are you root?
$ sudo !!
sudo apt-get install r-base
[sudo] password for <user>:
```

Sección 24.3: Búsqueda en el historial de comandos por patrón

Pulsa **Control + R** y escriba un patrón.

Por ejemplo, si ha ejecutado recientemente **man 5 crontab**, puede encontrarlo *rápidamente empezando a teclear "crontab"*. El prompt cambiará así:

```
(reverse-i-search)`cr': man 5 crontab
```

La **`cr'** es la cadena de caracteres que he escrito hasta ahora. Esta es una búsqueda incremental, por lo que a medida que continúe escribiendo, el resultado de la búsqueda se actualiza para que coincida con el comando más reciente que contenía el patrón.

Pulsa las teclas de flecha izquierda o derecha para editar el comando seleccionado antes de ejecutarlo, o la tecla **Enter** para ejecutarlo.

Por defecto, la búsqueda encuentra el comando ejecutado más recientemente que coincida con el patrón. Para ir más atrás en el historial pulsa **Control + R** de nuevo. Puede pulsarlo repetidamente hasta que encuentre el comando deseado.

Sección 24.4: Cambiar al directorio recién creado con !#:N

```
$ mkdir backup_download_directory && cd ! #:1
mkdir backup_download_directory && cd backup_download_directory
```


Esto sustituirá el enésimo argumento del comando actual. En el ejemplo `!#:1` se sustituye por el primer argumento, es decir, `backup_download_directory`.

Sección 24.5: Usando `!$`

Puede utilizar `!$` para reducir las repeticiones cuando utilice la línea de comandos:

```
$ echo ping
ping
$ echo !$
ping
```

También puede basarse en la repetición

```
$ echo !$ pong
ping pong
$ echo !$, a great game
pong, a great game
```

Observa que en el último ejemplo no obtuvimos `ping pong, a great game` porque el último argumento pasado al comando anterior fue `pong`, podemos evitar problemas como este añadiendo comillas. Continuando con el ejemplo, nuestro último argumento fue `game`:

```
$ echo "it is !$ time"
it is game time
$ echo "hooray, !$!"
hooray, it is game time!
```

Sección 24.6: Repetir el comando anterior con una sustitución

```
$ mplayer Lecture_video_part1.mkv
$ ^1^2^
mplayer Lecture_video_part2.mkv
```

Este comando sustituirá `1` por `2` en el comando ejecutado anteriormente. Sólo reemplazará la primera aparición de la cadena de caracteres y es equivalente a `!!:s/1/2/`.

Si desea reemplazar todas las ocurrencias, debe utilizar `!!:gs/1/2/` o `!!:as/1/2/`.

Capítulo 25: Matemáticas

Sección 25.1: Matemáticas con dc

dc es uno de los programas más antiguos de Unix.

Utiliza la notación polaca inversa, lo que significa que primero se apilan los números y luego las operaciones. Por ejemplo, $1+1$ se escribe como `1 1 +`.

Para imprimir un elemento de la parte superior de la pila utilice el comando `p`.

```
echo '2 3 + p' | dc
5
```

o

```
dc <<< '2 3 + p'
5
```

Puede imprimir el elemento superior varias veces

```
dc <<< '1 1 + p 2 + p'
2
4
```

Para los números negativos, utilice el prefijo `_`.

```
dc <<< '_1 p'
-1
```

También puedes utilizar las mayúsculas de la `A` a la `F` para los números entre `10` y `15` y `.` como punto decimal

```
dc <<< 'A.4 p'
10.4
```

dc utiliza precisión arbitraria, lo que significa que la precisión está limitada únicamente por la memoria disponible. Por defecto, la precisión está fijada en 0 decimales.

```
dc <<< '4 3 / p'
1
```

Podemos aumentar la precisión utilizando el comando `k`. `2k` utilizará

```
dc <<< '2k 4 3 / p'
1.33
```

```
dc <<< '4k 4 3 / p'
1.3333
```

También se puede utilizar en varias líneas

```
dc << EOF
1 1 +
3 *
p
EOF
6
```

bc es un preprocesador para **dc**.

Sección 25.2: Matemáticas utilizando las capacidades de bash

También se pueden realizar cálculos aritméticos sin necesidad de utilizar otros programas como éste.

Multipliación:

```
echo $((5 * 2))  
10
```

División:

```
echo $((5 / 2))  
2
```

Módulo:

```
echo $((5 % 2))  
1
```

Exponenciación:

```
echo $((5 ** 2))  
25
```

Sección 25.3: Matemáticas con bc

bc es un lenguaje de calculadora de precisión arbitraria. Puede utilizarse de forma interactiva o ejecutarse desde la línea de comandos.

Por ejemplo, puede imprimir el resultado de una expresión:

```
echo '2 + 3' | bc  
5
```

```
echo '12 / 5' | bc  
2
```

Para la aritmética de poste flotante, puede importar la biblioteca estándar **bc -l**:

```
echo '12 / 5' | bc -l  
2.400000000000000000000000000000
```

Puede utilizarse para comparar expresiones:

```
echo '8 > 5' | bc  
1
```

```
echo '10 == 11' | bc  
0
```

```
echo '10 == 10 && 8 > 3' | bc  
1
```

Sección 25.4: Matemáticas con expr

expr o **Evaluar expresiones** evalúa una expresión y escribe el resultado en la salida estándar.

Aritmética básica:

```
expr 2 + 3  
5
```

Al multiplicar, hay que escapar del signo *****:

```
expr 2 \* 3  
6
```

También puede utilizar variables:

```
a=2
expr $a + 3
5
```

Tenga en cuenta que sólo admite números enteros, por lo que una expresión como ésta:

```
expr 3.0 / 2
```

arrojará un error `expr: not a decimal number: '3.0'.`

Admite expresiones regulares para coincidir con patrones

```
expr 'Hello World' : 'Hell\(.*\)rld'
o Wo
```

O encontrar el índice del primer carácter de la cadena de caracteres de búsqueda

Esto arrojará `expr: syntax error` en **Mac OS X**, porque utiliza **BSD expr** que no tiene el comando `index`, mientras que `expr` en Linux es generalmente **GNU expr**.

```
expr index hello l
3
```

```
expr index 'hello' 'lo'
3
```

Capítulo 26: Aritmética Bash

Parámetro	Detalles
EXPRESIÓN	Expresión a evaluar

Sección 26.1: Aritmética simple con (())

```
#!/bin/bash
echo $(( 1 + 2 ))
```

Salida: 3

```
# Using variables
#!/bin/bash
var1=4
var2=5
((output=$var1 * $var2))
printf "%d\n" "$output"
```

Salida: 20

Sección 26.2: Comando aritmético

- **let**

```
let num=1+2
let num="1+2"
let 'num= 1 + 2'
let num=1 num+=2
```

Necesita comillas si hay espacios o caracteres globbing. Por lo tanto, se producirá un error:

```
let num = 1 + 2 # incorrecto
let 'num = 1 + 2' # correcto
let a[1] = 1 + 1 # incorrecto
let 'a[1] = 1 + 1' # correcto
```

- **(())**

```
((a=$a+1)) # añadir 1 a
((a = a + 1)) # como arriba
((a += 1)) # como arriba
```

Podemos utilizar **(())** en **if**. Algún ejemplo:

```
if (( a > 1 )); then echo "a is greater than 1"; fi
```

La salida de **(())** puede asignarse a una variable:

```
result=$((a + 1))
```

O se utiliza directamente en la salida:

```
echo "The result of a + 1 is $((a + 1))"
```

Sección 26.3: Aritmética simple con expr

```
#!/bin/bash
expr 1 + 2
```

Salida: 3

Capítulo 27: Ámbito

Sección 27.1: Ámbito dinámico en acción

El ámbito dinámico significa que las búsquedas de variables se producen en el ámbito en el que se *llama* a una función, no en el que está *definida*.

```
$ x=3
$ func1 () { echo "in func1: $x"; }
$ func2 () { local x=9; func1; }
$ func2
in func1: 9
$ func1
in func1: 3
```

En un lenguaje de ámbito léxico, `func1` buscaría *siempre* el valor de `x` en el ámbito global, porque `func1` está *definida* en el ámbito local.

En un lenguaje de ámbito dinámico, `func1` busca en el ámbito donde es *llamada*. Cuando se llama desde dentro de `func2`, primero busca en el cuerpo de `func2` un valor de `x`. Si no estuviera definido allí, buscaría en el ámbito global, desde donde se llamó a `func2`.

Capítulo 28: Sustitución de procesos

Sección 28.1: Comparar dos archivos de la web

A continuación, se comparan dos ficheros con `diff` utilizando la sustitución de procesos en lugar de crear ficheros temporales.

```
diff <(curl http://www.example.com/page1) <(curl http://www.example.com/page2)
```

Sección 28.2: Alimentar un bucle while con la salida de un comando

Esto alimenta un bucle `while` con la salida de un comando `grep`:

```
while IFS=":" read -r user _
do
    # "$user" contiene el nombre de usuario en /etc/passwd
done < <(grep "hello" /etc/passwd)
```

Sección 28.3: Concatenar archivos

Es bien sabido que no se puede utilizar el mismo archivo para la entrada y la salida en el mismo comando. Por ejemplo,

```
$ cat header.txt body.txt >body.txt
```

no hace lo que usted quiere. Cuando `cat` lea `body.txt`, ya habrá sido truncado por la redirección y estará vacío. El resultado final será que `body.txt` sólo contendrá el contenido de `header.txt`.

Se podría pensar en evitar esto con la sustitución de procesos, es decir, que el comando

```
$ cat header.txt <(cat body.txt) > body.txt
```

forzará a que el contenido original de `body.txt` se guarde de alguna manera en algún buffer en algún lugar antes de que el archivo sea truncado por la redirección. Esto no funciona. El `cat` entre paréntesis comienza a leer el archivo sólo después de que todos los descriptores de archivo han sido configurados, al igual que el externo. No tiene sentido intentar utilizar la sustitución de procesos en este caso.

La única forma de anteponer un fichero a otro es crear uno intermedio:

```
$ cat header.txt body.txt >body.txt.new
$ mv body.txt.new body.txt
```

que es lo que `sed` o `perl` o programas similares hacen bajo la alfombra cuando se les llama con una opción de edición *in situ* (normalmente `-i`).

Sección 28.4: Transmitir un archivo a través de varios programas a la vez

Esto cuenta el número de líneas en un archivo grande con `wc -l` mientras que simultáneamente lo comprime con `gzip`. Ambos se ejecutan simultáneamente.

```
tee >(wc -l >&2) < bigfile | gzip > bigfile.gz
```

Normalmente `tee` escribe su entrada en uno o más archivos (y `stdout`). Podemos escribir en comandos en lugar de archivos con `tee > (command)`.

Aquí el comando `wc -l >&2` cuenta las líneas leídas de `tee` (que a su vez está leyendo de `bigfile`). (El recuento de líneas se envía a `stderr` (`>&2`) para evitar que se mezcle con la entrada a `gzip`). La salida estándar de `tee` se envía simultáneamente a `gzip`.

Sección 28.5: Con el comando paste

```
# La sustitución de procesos con el comando paste es habitual
# Para comparar el contenido de dos directorios
paste <( ls /path/to/directory1 ) <( ls /path/to/directory2 )
```

Sección 28.6: Para evitar el uso de una subcápsula

Un aspecto importante de la sustitución de procesos es que nos permite evitar el uso de un sub-shell al canalizar comandos desde el shell.

Esto se puede demostrar con un sencillo ejemplo a continuación. Tengo los siguientes archivos en mi carpeta actual:

```
$ find . -maxdepth 1 -type f -print
foo bar zoo foobar foozoo barzoo
```

Si canalizo a un bucle `while/read` que incremente un contador de la siguiente manera:

```
count=0
find . -maxdepth 1 -type f -print | while IFS= read -r _; do
    ((count++))
done
```

`$count` ahora *no* contiene 6, porque fue modificado en el contexto del sub-shell. Cualquiera de los comandos que se muestran a continuación se ejecutan en un contexto de sub-shell y el alcance de las variables utilizadas dentro se pierden después de que el sub-shell termina.

```
command &
command | command
( command )
```

La sustitución del proceso resolverá el problema evitando el uso del operador de tubería `|` como en

```
count=0
while IFS= read -r _; do
    ((count++))
done <<(find . -maxdepth 1 -type f -print)
```

Esto mantendrá el valor de la variable `count` ya que no se invocan sub-shells.

Capítulo 29: Finalización programable

Sección 29.1: Finalización sencilla mediante la función

```
_mycompletion() {  
    local command_name="$1" # no utilizado en este ejemplo  
    local current_word="$2"  
    local previous_word="$3" # no utilizado en este ejemplo  
    # COMPREPLY es un array que tiene que ser llenada con las posibles terminaciones  
    # compgen se utiliza para filtrar las terminaciones coincidentes  
    COMPREPLY=( $(compgen -W 'hello world' -- "$current_word") )  
}  
complete -F _mycompletion mycommand
```

Ejemplo de uso:

```
$ mycommand [TAB][TAB]  
hello world  
$ mycommand h[TAB][TAB]  
$ mycommand hello
```

Sección 29.2: Completado sencillo de opciones y nombres de archivo

```
# La siguiente función de shell se utilizará para generar terminaciones para  
# el comando "nuance_tune".  
_nuance_tune_opts ()  
{  
    local curr_arg prev_arg  
    curr_arg=${COMP_WORDS[COMP_CWORD]}  
    prev_arg=${COMP_WORDS[COMP_CWORD-1]}  
  
    # La opción "config" toma un arg archivo, por lo que obtener una lista de los archivos en el  
    # directorio actual. Una sentencia case es probablemente innecesaria aquí, pero deja  
    # espacio para personalizar los parámetros para otras banderas.  
    case "$prev_arg" in  
        -config)  
            COMPREPLY=( $( /bin/ls -1 ) )  
            return 0  
            ;;  
    esac  
  
    # Utiliza compgen para completar todas las opciones conocidas.  
    COMPREPLY=( $(compgen -W '-analyze -experiment -generate_groups -compute_thresh -config -  
output -help -usage -force -lang -grammar_overrides -begin_date -end_date -group -dataset -  
multiparses -dump_records -no_index -confidencelevel -nrecs -dry_run -rec_scripts_only -  
save_temp -full_trc -single_session -verbose -ep -unsupervised -write_manifest -remap -  
noreparse -upload -reference -target -use_only_matching -histogram -stepsize' -- $curr_arg)  
    );  
}  
  
# El parámetro -o indica a Bash que procese las terminaciones como nombres de archivo, cuando  
# proceda.  
complete -o filenames -F _nuance_tune_opts nuance_tune
```

Capítulo 30: Personalizar PS1

Sección 30.1: Colorear y personalizar el indicador del terminal

Así es como el autor configura su variable personal PS1:

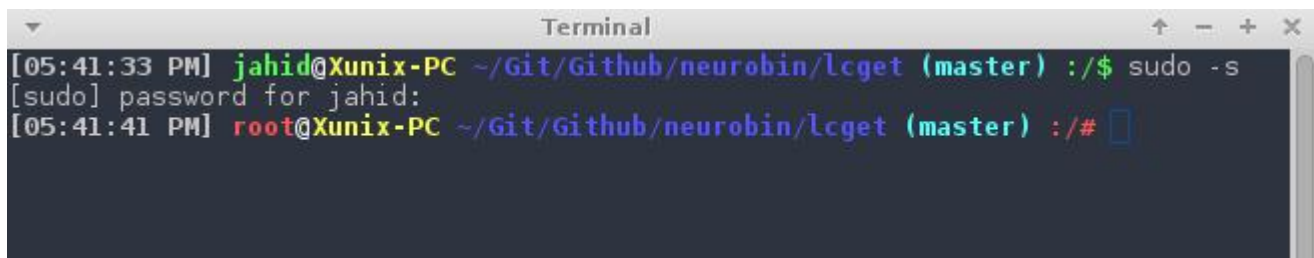
```
gitPS1(){
    gitps1=$(git branch 2>/dev/null | grep '*')
    gitps1="${gitps1:+ ($gitps1/#\* /)}"
    echo "$gitps1"
}

# Please use the below function if you are a mac user
gitPS1ForMac(){
    git branch 2> /dev/null | sed -e '/^\^*/d' -e 's/* \(.*/ (\1)/'
}

timeNow(){
    echo "$(date +%r)"
}

if [ "$color_prompt" = yes ]; then
    if [ x$EUID = x0 ]; then
        PS1='\[\033[1;38m\]$(timeNow)\[\033[00m\]\[\033[1;31m\]\u\[\033[00m\]\[\033[1;37m\]@
\[\033[00m\]\[\033[1;33m\]\h\[\033[00m\]\[\033[1;34m\]\w\[\033[00m\]\[\033[1;36m\]$(gitPS1)
\[\033[00m\] \[\033[1;31m\]:#\[\033[00m\] '
    else
        PS1='\[\033[1;38m\]$(timeNow)\[\033[00m\]\[\033[1;32m\]\u\[\033[00m\]\[\033[1;37m\]@
\[\033[00m\]\[\033[1;33m\]\h\[\033[00m\]\[\033[1;34m\]\w\[\033[00m\]\[\033[1;36m\]$(gitPS1)
\[\033[00m\] \[\033[1;32m\]:/$\[\033[00m\] '
    fi
else
    PS1='[$(timeNow)] \u@\h \w$(gitPS1) :/$ '
fi
```

Y así es como se ve mi consola:



Color de referencia:

```
# Colores
txtblk='\e[0;30m' # Negro - Regular
txtred='\e[0;31m' # Rojo
txtgrn='\e[0;32m' # Verde
txtylw='\e[0;33m' # Amarillo
txtblu='\e[0;34m' # Azul
txtpur='\e[0;35m' # Morado
txtcyn='\e[0;36m' # Cian
txtwht='\e[0;37m' # Blanco
bldblk='\e[1;30m' # Negro - Negrita
bldred='\e[1;31m' # Rojo
bldgrn='\e[1;32m' # Verde
bldylw='\e[1;33m' # Amarillo
bldblu='\e[1;34m' # Azul
bldpur='\e[1;35m' # Morado
bldcyn='\e[1;36m' # Cian
bldwht='\e[1;37m' # Blanco
```

```

unkblk='\e[4;30m' # Negro - Subrayado
undred='\e[4;31m' # Rojo
undgrn='\e[4;32m' # Verde
undylw='\e[4;33m' # Amarillo
undblu='\e[4;34m' # Azul
undpur='\e[4;35m' # Morado
undcyn='\e[4;36m' # Cian
undwht='\e[4;37m' # Blanco
bakblk='\e[40m' # Negro - Fondo
bakred='\e[41m' # Rojo
badgrn='\e[42m' # Verde
bakylw='\e[43m' # Amarillo
bakblu='\e[44m' # Azul
bakpur='\e[45m' # Morado
bakcyn='\e[46m' # Cian
bakwht='\e[47m' # Blanco
txtrst='\e[0m' # Restablecer texto

```

Notas:

- Haga los cambios en `~/.bashrc` o `/etc/bashrc` o `~/.bash_profile` o `~/.profile` (dependiendo del sistema operativo) y guárdelo.
- En el caso de `root`, es posible que también tengas que editar el archivo `/etc/bash.bashrc` o `/root/.bashrc`
- Ejecute `source ~/.bashrc` (específica de la distro) después de guardar el archivo.
- Nota: si ha guardado los cambios en `~/.bashrc`, recuerde añadir `source ~/.bashrc` en su `~/.bash_profile` para que este cambio en `PS1` quede registrado cada vez que se inicie la aplicación Terminal.

Sección 30.2: Mostrar el nombre de la rama git en el prompt del terminal

Usted puede tener funciones en la variable `PS1`, sólo asegúrese de comillas simples o utilizar escape para caracteres especiales:

```

gitPS1(){
    gitps1=$(git branch 2>/dev/null | grep '*')
    gitps1="${gitps1:+ ($gitps1/#\* /)}"
    echo "$gitps1"
}
PS1='\u@\h:\w$(gitPS1)$ '

```

Aparecerá un mensaje como éste:

```
user@Host:/path (master)$
```

Notas:

- Haga los cambios en `~/.bashrc` o `/etc/bashrc` o `~/.bash_profile` o `~/.profile` (dependiendo del sistema operativo) y guárdelo.
- Ejecute `source ~/.bashrc` (específica de la distro) después de guardar el archivo.

Sección 30.3: Mostrar la hora en el prompt del terminal

```

timeNow(){
    echo "$(date +%r)"
}
PS1='[$(timeNow)] \u@\h:\w$ '

```

Aparecerá un mensaje como éste:

```
[05:34:37 PM] user@Host:/path$
```

Notas:

- Haga los cambios en `~/.bashrc` o `/etc/bashrc` o `~/.bash_profile` o `~/.profile` (dependiendo del sistema operativo) y guárdelo.
- Ejecute `source ~/.bashrc` (específica de la distro) después de guardar el archivo.

Sección 30.4: Mostrar una rama git usando PROMPT_COMMAND

Si estás dentro de una carpeta de un repositorio `git` puede estar bien mostrar la rama actual en la que estás. En `~/.bashrc` o `/etc/bashrc` añade lo siguiente (se requiere `git` para que esto funcione):

```
function prompt_command {  
    # Comprobar si estamos dentro de un repositorio git  
    if git status > /dev/null 2>&1; then  
        # Obtener sólo el nombre del branch  
        export GIT_STATUS=$(git status | grep 'On branch' | cut -b 10-)  
    else  
        export GIT_STATUS=""  
    fi  
}
```

```
# Esta función se llama cada vez que se muestra PS1  
PROMPT_COMMAND=prompt_command
```

```
PS1="\$GIT_STATUS \u@\h:\w\$ "
```

Si estamos en una carpeta dentro de un repositorio `git` esto saldrá:

```
branch user@machine:~$
```

Y si estamos dentro de una carpeta normal:

```
user@machine:~$
```

Sección 30.5: Cambiar el indicador PS1

Para cambiar `PS1`, basta con cambiar el valor de la variable de shell `PS1`. El valor se puede establecer en `~/.bashrc` o `/etc/bashrc`, dependiendo de la distribución. `PS1` se puede cambiar a cualquier texto plano como:

```
PS1="hello "
```

Además del texto sin formato, se admiten varios caracteres especiales con barra diagonal inversa:

Formato	Acción
<code>\a</code>	un carácter de campana ASCII (07)
<code>\d</code>	la fecha en formato "DiaSemana Mes Fecha" (por ejemplo, "Tue May 26")
<code>\D{format}</code>	el formato se pasa a <code>strftime(3)</code> y el resultado se inserta en la cadena de consulta; un formato vacío da como resultado una representación de la hora específica de la localidad. Los corchetes son obligatorios
<code>\e</code>	un carácter de escape ASCII (033)
<code>\h</code>	el nombre de host hasta el primer '.'
<code>\H</code>	el nombre de host
<code>\j</code>	el número de trabajos gestionados actualmente por el shell
<code>\l</code>	el nombre base del dispositivo terminal del intérprete de comandos
<code>\n</code>	nueva línea
<code>\r</code>	retorno de carro
<code>\s</code>	el nombre del intérprete de comandos, el nombre base de <code>\$0</code> (la parte que sigue a la barra final)
<code>\t</code>	la hora actual en formato de 24 horas HH:MM:SS
<code>\T</code>	la hora actual en formato de 12 horas HH:MM:SS
<code>\@</code>	la hora actual en formato de 12 horas am/pm
<code>\A</code>	la hora actual en formato de 24 horas HH:MM
<code>\u</code>	el nombre de usuario del usuario actual
<code>\v</code>	la versión de bash (por ejemplo, 2.00)
<code>\V</code>	la versión de bash, versión + nivel de parche (por ejemplo, 2.00.0)
<code>\w</code>	el directorio de trabajo actual, con <code>\$HOME</code> abreviado con una tilde
<code>\W</code>	el nombre base del directorio de trabajo actual, con <code>\$HOME</code> abreviado con una tilde
<code>\!</code>	el número de historial de este comando
<code>\#</code>	el número de este comando
<code>\\$</code>	si el UID efectivo es 0, un #, en caso contrario un \$
<code>\nnn*</code>	el carácter correspondiente al número octal <code>nnn</code>
<code>\</code>	una barra invertida
<code>\[</code>	iniciar una secuencia de caracteres no imprimibles, que podrían utilizarse para incrustar una secuencia de control de terminal en el prompt
<code>\]</code>	poner fin a una secuencia de caracteres no imprimibles

Así, por ejemplo, podemos poner PS1 a:

```
PS1="\u@\h:\w\$ "
```

Y saldrá:

```
user@machine:~$
```

Sección 30.6: Mostrar el estado de retorno del comando anterior y la hora

A veces necesitamos una pista visual para indicar el estado de retorno del comando anterior. El siguiente fragmento de código lo pone en la cabecera del PS1.

Ten en cuenta que la función `__stat()` debe ser llamada cada vez que se genere un nuevo PS1, o de lo contrario se quedaría con el estado de retorno del último comando de tu `.bashrc` o `.bash_profile`.

```
# - CODIGOS COLOR ANSI - #
Color_Off="\033[0m"
###-Regular-###
Red="\033[0;31m"
Green="\033[0;32m"
Yellow="\033[0;33m"
####-Negrita-####

function __stat() {
    if [ $? -eq 0 ]; then
        echo -en "$Green ✓ $Color_Off "
    else
        echo -en "$Red ✗ $Color_Off "
    fi
}

PS1='${__stat}'
PS1+="[\t] "
PS1+="\e[0;33m\u@\h\e[0m:\e[1;34m\w\e[0m \n$ "

export PS1
```



```
✓ [22:50:55] wenzhong@musicforever:~
$ date
Sun Sep  4 22:51:00 CST 2016
✓ [22:51:00] wenzhong@musicforever:~
$ date_
-bash: date_: command not found
✗ [22:51:12] wenzhong@musicforever:~
$
```

Capítulo 31: Expansión de llaves

Sección 31.1: Modificar la extensión del nombre de archivo

```
$ mv filename.{jar,zip}
```

Esto se expande a `mv filename.jar filename.zip`.

Sección 31.2: Crear directorios para agrupar archivos por mes y año

```
$ mkdir 20{09..11}-{01..12}
```

El comando `ls` mostrará que se han creado los siguientes directorios:

```
2009-01 2009-04 2009-07 2009-10 2010-01 2010-04 2010-07 2010-10 2011-01 2011-04 2011-07 2011-10
2009-02 2009-05 2009-08 2009-11 2010-02 2010-05 2010-08 2010-11 2011-02 2011-05 2011-08 2011-11
2009-03 2009-06 2009-09 2009-12 2010-03 2010-06 2010-09 2010-12 2011-03 2011-06 2011-09 2011-12
```

Al poner un `0` delante del `9` en el ejemplo, los números se rellenan con un solo `0`. También puede rellenar los números con varios ceros, por ejemplo:

```
$ echo {001..10}
001 002 003 004 005 006 007 008 009 010
```

Sección 31.3: Crear una copia de seguridad de dotfiles

```
$ cp .vimrc{,.bak}
```

Esto se expande en el comando `cp .vimrc .vimrc.bak`.

Sección 31.4: Utilizar incrementos

```
$ echo {0..10..2}
0 2 4 6 8 10
```

Un tercer parámetro para especificar un incremento, es decir `{start..end..increment}`.

El uso de incrementos no se limita a los números

```
$ for c in {a..z..5}; do echo -n $c; done
afkpuz
```

Sección 31.5: Uso de la expansión de llaves para crear listas

Bash puede crear fácilmente listas a partir de caracteres alfanuméricos.

```
# lista de la a a la z
$ echo {a..z}
a b c d e f g h i j k l m n o p q r s t u v w x y z

# invertir de z a a
$ echo {z..a}
z y x w v u t s r q p o n m l k j i h g f e d c b a

# dígitos
$ echo {1..20}
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

# con ceros a la izquierda
$ echo {01..20}
```

```
01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20
```

```
# dígito inverso
```

```
$ echo {20..1}
```

```
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
```

```
# invertido con ceros a la izquierda
```

```
$ echo {20..01}
```

```
20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01
```

```
# combinación de múltiples llaves
```

```
$ echo {a..d}{1..3}
```

```
a1 a2 a3 b1 b2 b3 c1 c2 c3 d1 d2 d3
```

La expansión de llaves es la primera que tiene lugar, por lo que no puede combinarse con ninguna otra.

Sólo se pueden utilizar caracteres y dígitos.

Esto no funcionará: **echo** {\$(date +%H)..24}

Sección 31.6: Crear varios directorios con subdirectorios

```
mkdir -p toplevel/sublevel_{01..09}/{child1,child2,child3}
```

Esto creará una carpeta de nivel superior llamada `toplevel`, nueve carpetas dentro de `toplevel` llamadas `sublevel_01`, `sublevel_02`, etc. Luego dentro de esos subniveles: carpetas `child1`, `child2`, `child3`, dándole:

```
toplevel/sublevel_01/child1
```

```
toplevel/sublevel_01/child2
```

```
toplevel/sublevel_01/child3
```

```
toplevel/sublevel_02/child1
```

etc. Encuentro esto muy útil para crear múltiples carpetas y subcarpetas para mis propósitos específicos, con un comando bash. Sustituya las variables para ayudar a automatizar/analizar la información dada a la secuencia de comandos.

Capítulo 32: getopts : análisis sintáctico inteligente de parámetros de posición

Parámetro	Detalles
optstring	Los caracteres opcionales que deben reconocerse
name	Nombre donde se almacena la opción analizada

Sección 32.1: pingnmap

```
#!/bin/bash
# Script name : pingnmap
# Scenario : The systems admin in company X is tired of the monotonous job
# of pinging and nmapping, so he decided to simplify the job using a script.
# The tasks he wish to achieve is
# 1. Ping - with a max count of 5 -the given IP address/domain. AND/OR
# 2. Check if a particular port is open with a given IP address/domain.
# And getopts is for her rescue.
# A brief overview of the options
# n : meant for nmap
# t : meant for ping
# i : The option to enter the IP address
# p : The option to enter the port
# v : The option to get the script versión
```

```
while getopts ':nti:p:v' opt
# putting : in the beginnig suppresses the errors for invalid options
do
case "$opt" in
    'i')ip="${OPTARG}"
        ;;
    'p')port="${OPTARG}"
        ;;
    'n')nmap_yes=1;
        ;;
    't')ping_yes=1;
        ;;
    'v')echo "pingnmap version 1.0.0"
        ;;
    *) echo "Invalid option $opt"
        echo "Usage : "
        echo "pingmap -[n|t|i|p]|v]"
        ;;
esac
done
if [ ! -z "$nmap_yes" ] && [ "$nmap_yes" -eq "1" ]
then
    if [ ! -z "$ip" ] && [ ! -z "$port" ]
    then
        nmap -p "$port" "$ip"
    fi
fi

if [ ! -z "$ping_yes" ] && [ "$ping_yes" -eq "1" ]
then
    if [ ! -z "$ip" ]
    then
        ping -c 5 "$ip"
    fi
fi
```

```
shift $(( OPTIND - 1 )) # Processing additional arguments
if [ ! -z "$@" ]
then
    echo "Bogus arguments at the end : $@"
fi
```

Salida

```
$ ./pingnmap -nt -i google.com -p 80
```

```
Starting Nmap 6.40 ( http://nmap.org ) at 2016-07-23 14:31 IST
Nmap scan report for google.com (216.58.197.78)
Host is up (0.034s latency).
rDNS record for 216.58.197.78: maa03s21-in-f14.1e100.net
PORT STATE SERVICE
80/tcp open http
```

```
Nmap done: 1 IP address (1 host up) scanned in 0.22 seconds
PING google.com (216.58.197.78) 56(84) bytes of data.
64 bytes from maa03s21-in-f14.1e100.net (216.58.197.78): icmp_seq=1 ttl=57 time=29.3 ms
64 bytes from maa03s21-in-f14.1e100.net (216.58.197.78): icmp_seq=2 ttl=57 time=30.9 ms
64 bytes from maa03s21-in-f14.1e100.net (216.58.197.78): icmp_seq=3 ttl=57 time=34.7 ms
64 bytes from maa03s21-in-f14.1e100.net (216.58.197.78): icmp_seq=4 ttl=57 time=39.6 ms
64 bytes from maa03s21-in-f14.1e100.net (216.58.197.78): icmp_seq=5 ttl=57 time=32.7 ms
```

```
--- google.com ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4007ms
rtt min/avg/max/mdev = 29.342/33.481/39.631/3.576 ms
$ ./pingnmap -v
pingnmap version 1.0.0
$ ./pingnmap -h
Invalid option ?
Usage :
pingmap -[n|t|i|p]|v]
$ ./pingnmap -v
pingnmap version 1.0.0
$ ./pingnmap -h
Invalid option ?
Usage :
pingmap -[n|t|i|p]|v]
```

Capítulo 33: Depurar

Sección 33.1: Comprobación de la sintaxis de un script con "-n"

La opción `-n` permite comprobar la sintaxis de un script sin tener que ejecutarlo:

```
~> $ bash -n testscript.sh
testscript.sh: line 128: unexpected EOF while looking for matching `''
testscript.sh: line 130: syntax error: unexpected end of file
```

Sección 33.2: Depuración con bashdb

Bashdb es una utilidad similar a `gdb`, en la que puedes hacer cosas como establecer puntos de interrupción en una línea o en una función, imprimir el contenido de variables, puedes reiniciar la ejecución de scripts y mucho más.

Normalmente puede instalarlo a través de su gestor de paquetes, por ejemplo, en Fedora:

```
sudo dnf install bashdb
```

O consíguelo en la página de inicio. Luego puedes ejecutarlo con tu script como parámetro:

```
bashdb <YOUR SCRIPT>
```

Aquí tienes algunos comandos para empezar:

```
l - show local lines, press l again to scroll down
s - step to next line
print $VAR - echo out content of variable
restart - reruns bashscript, it re-loads it prior to execution.
eval - evaluate some custom command, ex: eval echo hi
```

```
b set breakpoint on some line
c - continue till some breakpoint
i b - info on break points
d - delete breakpoint at line #
```

```
shell - launch a sub-shell in the middle of execution, this is handy for manipulating variables
```

Para más información, recomiendo consultar el manual:

<http://www.rodericksmith.plus.com/outlines/manuals/bashdbOutline.html>

Véase también la página de inicio:

<http://bashdb.sourceforge.net/>

Sección 33.3: Depuración de un script bash con "-x"

Utilice `"-x"` para habilitar la salida de depuración de las líneas ejecutadas. Puede ejecutarse en toda una sesión o script, o habilitarse mediante programación dentro de un script.

Ejecuta un script con la salida de depuración activada:

```
$ bash -x myscript.sh
```

O

```
$ bash --debug myscript.sh
```

Activa la depuración dentro de un script bash. Se puede volver a activar opcionalmente, aunque la salida de depuración se restablece automáticamente cuando se sale del script.

```
#!/bin/bash
set -x # Activar depuración
# algo de código aquí
set +x # Desactiva la salida de depuración.
```

Capítulo 34: Concordancia de patrones y expresiones regulares

Sección 34.1: Obtener grupos capturados a partir de una coincidencia regex con una cadena de caracteres

```
a='I am a simple string with digits 1234'
pat='(.*) ([0-9]+)'
[[ "$a" =~ $pat ]]
echo "${BASH_REMATCH[0]}"
echo "${BASH_REMATCH[1]}"
echo "${BASH_REMATCH[2]}"
```

Salida:

```
I am a simple string with digits 1234
I am a simple string with digits
1234
```

Sección 34.2: Comportamiento cuando un glob no coincide con nada

Preparación

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

En caso de que el global no coincida con nada, el resultado viene determinado por las opciones `nullglob` y `failglob`. Si ninguna de ellas está activada, Bash devolverá el propio global si no se encuentra nada.

```
$ echo no*match
no*match
```

Si `nullglob` está activado, no se devuelve nada (`null`):

```
$ shopt -s nullglob
$ echo no*match
```

```
$
```

Si `failglob` está activado, se devuelve un mensaje de error:

```
$ shopt -s failglob
$ echo no*match
bash: no match: no*match
$
```

Tenga en cuenta que la opción `failglob` sustituye a la opción `nullglob`, es decir, si `nullglob` y `failglob` están ambas activadas, en caso de que no haya coincidencia se devuelve un error.

Sección 34.3: Comprobar si una cadena de caracteres coincide con una expresión regular

Version ≥ 3.0

Comprueba si una cadena de caracteres consta exactamente de 8 dígitos:

```
$ date=20150624
$ [[ $date =~ ^[0-9]{8}$ ]] && echo "yes" || echo "no"
yes
$ date=hello
$ [[ $date =~ ^[0-9]{8}$ ]] && echo "yes" || echo "no"
no
```

Sección 34.4: Correspondencia Regex

```
pat='^[0-9]+([0-9]+)'\ns='I am a string with some digits 1024'\n[[ $s =~ $pat ]] # $pat no debe estar citada\necho "${BASH_REMATCH[0]}"\necho "${BASH_REMATCH[1]}"
```

Salida:

```
I am a string with some digits 1024
1024
```

En lugar de asignar la regex a una variable (`$pat`) también podríamos hacer:

```
[[ $s =~ [^0-9]+([0-9]+) ]]
```

Explicación

- La construcción `[[$s =~ $pat]]` realiza la correspondencia regex
- Los grupos capturados, es decir, los resultados de las coincidencias, están disponibles en un array denominado `BASH_REMATCH`
- El índice 0 del array `BASH_REMATCH` es la coincidencia total
- El índice `i` del array `BASH_REMATCH` es el grupo capturado `i`, donde `i = 1, 2, 3 ...`

Sección 34.5: El global *

Preparación

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

El asterisco `*` es probablemente el global más utilizado. Simplemente coincide con cualquier cadena de caracteres.

```
$ echo *acy
macy stacy tracy
```

Un solo `*` no coincidirá con archivos y carpetas que residan en subcarpetas

```
$ echo *
emptyfolder folder macy stacy tracy
$ echo folder/*
folder/anotherfolder folder/subfolder
```

Sección 34.6: El global `**`

Version ≥ 4.0

Preparación

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -s globstar
```

Bash es capaz de interpretar dos asteriscos adyacentes como un único global. Con la opción `globstar` activada, puede utilizarse para buscar carpetas que se encuentren a mayor profundidad en la estructura de directorios.

```
echo **
emptyfolder folder folder/anotherfolder folder/anotherfolder/content
folder/anotherfolder/content/deepfolder folder/anotherfolder/content/deepfolder/file
folder/subfolder folder/subfolder/content folder/subfolder/content/deepfolder
folder/subfolder/content/deepfolder/file macy stacy tracy
```

El `**` puede considerarse una expansión de la ruta, independientemente de su profundidad. Este ejemplo coincide con cualquier archivo o carpeta que empiece por `deep`, independientemente de la profundidad a la que esté anidado:

```
$ echo **/deep*
folder/anotherfolder/content/deepfolder folder/subfolder/content/deepfolder
```

Sección 34.7: El global `?`

Preparación

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

El `?` simplemente coincide exactamente con un carácter.

```
$ echo ?acy
macy
$ echo ??acy
stacy tracy
```

Sección 34.8: El global []

Preparación

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

Si es necesario buscar caracteres específicos, se puede utilizar '[]'. Cualquier carácter dentro de '[]' se comparará exactamente una vez.

```
$ echo [m]acy
macy
$ echo [st][tr]acy
stacy tracy
```

Sin embargo, el global [] es más versátil que eso. También permite una coincidencia negativa e incluso la coincidencia de rangos de caracteres y clases de caracteres. Una coincidencia negativa se consigue utilizando ! o ^ como primer carácter después de [. Podemos coincidir con stacy mediante.

```
$ echo [!t][^r]acy
stacy
```

Aquí le estamos diciendo a bash que sólo queremos que coincida con los archivos que no comienzan con una t y la segunda letra no es una r y el archivo termina en acy.

Los rangos pueden emparejarse separando un par de caracteres con un guión (-). Cualquier carácter que se encuentre entre esos dos caracteres (ambos inclusive) será coincidente. Por ejemplo, [r-t] equivale a [rst].

```
$ echo [r-t][r-t]acy
stacy tracy
```

Las clases de caracteres pueden emparejarse mediante [:class:], por ejemplo, para emparejar archivos que contengan un espacio en blanco.

```
$ echo *[:blank:]*
file with space
```

Sección 34.9: Búsqueda de archivos ocultos

Preparación

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

La opción `dotglob` incorporada en Bash permite hacer coincidir archivos y carpetas ocultos, es decir, archivos y carpetas que empiezan por `.`


```
$ shopt -s dotglob
$ echo *
file with space folder .hiddenfile macy stacy tracy
```

Sección 34.10: Coincidencia de mayúsculas y minúsculas

Preparación

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

Si se activa la opción `nocaseglob`, el global no distinguirá entre mayúsculas y minúsculas.

```
$ echo M*
M*
$ shopt -s nocaseglob
$ echo M*
macy
```

Sección 34.11: Globalización ampliada

Version ≥ 2.02

Preparación

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

La opción `extglob` incorporada en Bash puede ampliar las capacidades de coincidencia de un global

```
shopt -s extglob
```

Los siguientes sub-patrones comprenden globales extendidos válidos:

- `?(lista-patrón)` - Coincide con cero o una ocurrencia de los patrones dados
- `*(lista-patrón)` - Coincide con cero o más ocurrencias de los patrones dados
- `+(lista-patrón)` - Coincide con una o más ocurrencias de los patrones dados
- `@(lista-patrón)` - Coincide con uno de los patrones dados
- `!(lista-patrón)` - Coincide con todo excepto con uno de los patrones dados

La lista de patrones es una lista de globales separados por `|`.

```
$ echo *([r-t])acy  
stacy tracy
```

```
$ echo *([r-t]|m)acy  
macy stacy tracy
```

```
$ echo ?([a-z])acy  
macy
```

La propia `lista-patrón` puede ser otro global extendido anidado. En el ejemplo anterior hemos visto que podemos hacer coincidir `tracy` y `stacy` con `*([r-t])`. Este global extendido se puede utilizar dentro del global extendido negado `!([lista-patrón])` para que coincida con `macy`

```
$ echo !(*([r-t]))acy  
macy
```

Coincide con cualquier cosa que **no** empiece con cero o más apariciones de las letras `r`, `s` y `t`, lo que deja sólo a `macy` como posible coincidencia.

Capítulo 35: Cambiar de shell

Sección 35.1: Encontrar el shell actual

Hay varias formas de determinar el caparazón actual

```
echo $0
ps -p $$
echo $SHELL
```

Sección 35.2: Lista de shells disponibles

Para listar las conchas de acceso disponibles:

```
cat /etc/shells
```

Ejemplo:

```
$ cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/dash
/bin/bash
/bin/rbash
```

Sección 35.3: Cambiar el Shell

Para cambiar el bash actual ejecuta estos comandos

```
export SHELL=/bin/bash
exec /bin/bash
```

para cambiar el bash que se abre al inicio edita `.profile` y añade estas líneas.

Capítulo 36: Variables internas

Una visión general de las variables internas de Bash, dónde, cómo y cuándo usarlas.

Sección 36.1: Resumen de las variables internas de Bash

Variable	Detalles
<code>\$*</code> / <code>\$@</code>	Parámetros posicionales de función/script (argumentos). Expándalos como sigue: <code>\$*</code> y <code>\$@</code> son lo mismo que <code>\$1 \$2 ...</code> (tenga en cuenta que generalmente no tiene sentido dejarlas sin comillas) <code>"\$*"</code> es lo mismo que <code>"\$1 \$2 ..." 1</code> <code>"\$@"</code> es lo mismo que <code>"\$1" "\$2" ...</code> 1. Los argumentos están separados por el primer carácter de <code>\$IFS</code> , que no tiene por qué ser un espacio.
<code>\$#</code>	Número de parámetros posicionales pasados al script o función
<code>\$!</code>	ID de proceso del último comando (el más alto en el caso de las canalizaciones) del trabajo más reciente puesto en segundo plano (tenga en cuenta que no es necesariamente el mismo que el ID del grupo de procesos del trabajo cuando el control de trabajos está activado).
<code>\$\$</code>	ID del proceso que ejecutó bash
<code>\$?</code>	Estado de salida del último comando
<code>\$n</code>	Parámetros de posición, donde <code>n=1, 2, 3, ..., 9</code>
<code>\${n}</code>	Parámetros posicionales (igual que arriba), pero <code>n</code> puede ser <code>> 9</code>
<code>\$0</code>	En scripts, ruta con la que se invocó al script; con bash <code>-c 'printf "%s\n" "\$0"'</code> <code>nameargs</code> : <code>name</code> (el primer argumento después del script en línea), de lo contrario, el <code>argv[0]</code> que recibió bash .
<code>_</code>	Último campo del último comando
<code>\$IFS</code>	Separador de campo interno
<code>\$PATH</code>	Variable de entorno <code>PATH</code> utilizada para buscar ejecutables
<code>\$OLDPWD</code>	Directorio de trabajo anterior
<code>\$PWD</code>	Directorio de trabajo actual
<code>\$FUNCNAME</code>	Array de nombres de funciones en la pila de llamadas de ejecución
<code>\$BASH_SOURCE</code>	Array que contiene las rutas de origen de los elementos de la matriz <code>FUNCNAME</code> . Se puede utilizar para obtener la ruta del script.
<code>\$BASH_ALIASES</code>	Array asociativo que contiene todos los alias definidos actualmente
<code>\$BASH_REMATCH</code>	Array de coincidencias a partir de la última coincidencia regex
<code>\$BASH_VERSION</code>	Cadena de versión de Bash
<code>\$BASH_VERSINFO</code>	Un array de 6 elementos con información sobre la versión de Bash
<code>\$BASH</code>	Ruta absoluta a la shell Bash que se está ejecutando en ese momento (heurísticamente determinada por bash basándose en <code>argv[0]</code> y el valor de <code>\$PATH</code> ; puede ser errónea en casos puntuales).
<code>\$BASH_SUBSHELL</code>	Nivel de subshell de Bash
<code>\$UID</code>	ID de usuario real (no efectivo si es diferente) del proceso que ejecuta bash .
<code>\$PS1</code>	Línea de comandos principal; véase Uso de las variables PS
<code>\$PS2</code>	Línea de comandos secundaria (utilizada para entradas adicionales)
<code>\$PS3</code>	Línea de comandos terciaria (utilizada en el bucle de selección)
<code>\$PS4</code>	Línea de comandos cuaternaria (utilizada para añadir información a la salida detallada)
<code>\$RANDOM</code>	Un número entero pseudoaleatorio entre 0 y 32767
<code>\$REPLY</code>	Variable utilizada por read por defecto cuando no se especifica ninguna variable. También utilizada por SELECT para devolver el valor proporcionado por el usuario.
<code>\$PIPESTATUS</code>	Variable de array que contiene los valores de estado de salida de cada comando en el canal en primer plano ejecutado más recientemente.

La asignación de variables no debe tener espacios antes y después. `a=123` no `a = 123`. Este último (un signo igual rodeado de espacios) de forma aislada significa ejecutar el comando `a` con los argumentos `=` y `123`, aunque también se ve en el operador de comparación de cadenas (que sintácticamente es un argumento de `[` o `[[` o la prueba que esté utilizando).

Sección 36.2: `$@`

`"$@"` se expande a todos los argumentos de la línea de comandos como palabras separadas. Es diferente de `"$*"`, que se expande a todos los argumentos como una sola palabra.

`"$@"` es especialmente útil para recorrer argumentos y manejar argumentos con espacios.

Consideremos que estamos en un script que invocamos con dos argumentos, así:

```
$ ./script.sh "_1_2_" "_3_4_"
```

Las variables `$*` o `$@` se expandirán en `$1_2_$2`, que a su vez se expanden en `1_2_3_4` por lo que el bucle de abajo:

```
for var in $*; do # same for var in $@; do
    echo "\\<"$var"\\>"
done
```

se imprimirá tanto para

```
<1>
<2>
<3>
<4>
```

Mientras que `"$*"` se expandirá en `"$1_2_$2"` que a su vez se expandirá en `"_1_2_3_4_"` y así el bucle:

```
for var in "$*"; do
    echo "\\<"$var"\\>"
done
```

sólo invocará `echo` una vez e imprimirá

```
<_1_2_3_4_>
```

Y finalmente `"$@"` se expandirá en `"$1" "$2"`, que se expandirá en `"_1_2_3_4_"` y así el bucle

```
for var in "$@"; do
    echo "\\<"$var"\\>"
done
```

imprimirá

```
<_1_2_>
<_3_4_>
```

preservando así tanto el espaciado interno en los argumentos como la separación de argumentos. Observe que la construcción `for var in "$@"; do ...` es tan común e idiomática que es la predeterminada para un bucle `for` y puede abreviarse a `for var; do ...`.

Sección 36.3: `$#`

Para obtener el número de argumentos de línea de comandos o parámetros posicionales - escriba:

```
#!/bin/bash
echo "$#"
```

Cuando se ejecuta con tres argumentos, el ejemplo anterior dará como resultado la salida:

```
~> $ ./testscript.sh firstarg secondarg thirdarg
3
```

Sección 36.4: \$HISTSIZE

El número máximo de comandos recordados:

```
~> $ echo $HISTSIZE
1000
```

Sección 36.5: \$FUNCNAME

Para obtener el nombre de la función actual - escriba:

```
my_function()
{
    echo "This function is $FUNCNAME" # El resultado será "This function is my_function"
}
```

Esta instrucción no devolverá nada si la escribe fuera de la función:

```
my_function
```

```
echo "This function is $FUNCNAME" # El resultado será "This function is"
```

Sección 36.6: \$HOME

El directorio personal del usuario

```
~> $ echo $HOME
/home/user
```

Sección 36.7: \$IFS

Contiene la cadena de caracteres del separador interno de campos que bash utiliza para dividir cadenas al hacer bucles, etc. Por defecto son los caracteres de espacio en blanco: `\n` (nueva línea), `\t` (tabulador) y espacio. Cambiando esto a otra cosa le permite dividir cadenas de caracteres usando diferentes caracteres:

```
IFS=","
INPUTSTR="a,b,c,d"
for field in ${INPUTSTR}; do
    echo $field
done
```

El resultado de lo anterior es:

```
a
b
c
d
```

Notas:

- Esto es responsable del fenómeno conocido como división de palabras.

Sección 36.8: \$OLDPWD

OLDPWD (**OLD**Print**W**orking**D**irectory) contiene el directorio anterior al último comando **cd**:

```
~> $ cd directory
directory> $ echo $OLDPWD
/home/user
```

Sección 36.9: \$PWD

PWD (**P**rint**W**orking**D**irectory) El directorio de trabajo actual en el que se encuentra en este momento:

```
~> $ echo $PWD
/home/user
~> $ cd directory
directory> $ echo $PWD
/home/user/directory
```

Sección 36.10: \$1 \$2 \$3 etc..

Parámetros posicionales pasados al script desde la línea de comandos o desde una función:

```
#!/bin/bash
# $n es el parámetro posicional n
echo "$1"
echo "$2"
echo "$3"
```

El resultado de lo anterior es:

```
~> $ ./testscript.sh firstarg secondarg thirdarg
firstarg
secondarg
thirdarg
```

Si el número de argumentos posicionales es superior a nueve, deben utilizarse llaves.

```
# "set -- " establece parámetros de posición
set -- 1 2 3 4 5 6 7 8 nine ten eleven twelve
# la siguiente línea mostrará 10 no 1 como valor de $1 el dígito 1
# se concatenará con el siguiente 0
echo $10 # resultado 1
echo ${10} # resultado ten
# para mostrarlo claramente:
set -- arg{1..12}
echo $10
echo ${10}
```

Sección 36.11: \$*

Devolverá todos los parámetros posicionales en una única cadena de caracteres.

testscript.sh:

```
#!/bin/bash
echo "$@"
```

Ejecute el script con varios argumentos:

```
./testscript.sh firstarg secondarg thirdarg
```

Salida:

```
firstarg secondarg thirdarg
```

Sección 36.12: \$!

El ID de proceso (pid) del último trabajo ejecutado en segundo plano:

```
~> $ ls &
testfile1 testfile2
[1]+  Done                  ls
~> $ echo $!
21715
```

Sección 36.13: \$?

El estado de salida de la última función o comando ejecutado. Normalmente 0 significará OK cualquier otra cosa indicará un fallo:

```
~> $ ls *.blah;echo $?
ls: cannot access *.blah: No such file or directory
2
~> $ ls;echo $?
testfile1 testfile2
0
```

Sección 36.14: \$\$

El ID de proceso (pid) del proceso actual:

```
~> $ echo $$
13246
```

Sección 36.15: \$RANDOM

Cada vez que se hace referencia a este parámetro, se genera un número entero aleatorio entre 0 y 32767. Asignar un valor a esta variable alimenta el generador de números aleatorios ([fuente](#)).

```
~> $ echo $RANDOM
27119
~> $ echo $RANDOM
1349
```

Sección 36.16: \$BASHPID

ID del proceso (pid) de la instancia actual de Bash. Esto no es lo mismo que la variable \$\$, pero a menudo da el mismo resultado. Esto es nuevo en Bash 4 y no funciona en Bash 3.

```
~> $ echo "\${$ pid = $$ BASHPID = $BASHPID}"
${$ pid = 9265 BASHPID = 9265
```

Sección 36.17: \$BASH_ENV

Una variable de entorno que apunta al archivo de inicio de Bash que se lee cuando se invoca un script.

Sección 36.18: \$BASH_VERSIONINFO

Un array que contiene la información completa de la versión dividida en elementos, mucho más conveniente que \$BASH_VERSION si sólo buscas la versión principal:

```
~> $ for ((i=0; i<=5; i++)); do echo "BASH_VERSIONINFO[$i] = ${BASH_VERSIONINFO[$i]}"; done
BASH_VERSIONINFO[0] = 3
BASH_VERSIONINFO[1] = 2
BASH_VERSIONINFO[2] = 25
BASH_VERSIONINFO[3] = 1
BASH_VERSIONINFO[4] = release
BASH_VERSIONINFO[5] = x86_64-redhat-linux-gnu
```

Sección 36.19: \$BASH_VERSION

Muestra la versión de bash que se está ejecutando, esto le permite decidir si puede utilizar alguna característica avanzada:

```
~> $ echo $BASH_VERSION
4.1.2(1)-release
```

Sección 36.20: \$EDITOR

El editor por defecto que será invocado por cualquier script o programa, normalmente **vi** o **emacs**.

```
~> $ echo $EDITOR
vi
```

Sección 36.21: \$HOSTNAME

El nombre de host asignado al sistema durante el arranque.

```
~> $ echo $HOSTNAME
mybox.mydomain.com
```

Sección 36.22: \$HOSTTYPE

Esta variable identifica el hardware, puede ser útil para determinar qué binarios ejecutar:

```
~> $ echo $HOSTTYPE
x86_64
```

Sección 36.23: \$MACHTYPE

Al igual que \$HOSTTYPE, también incluye información sobre el sistema operativo y el hardware.

```
~> $ echo $MACHTYPE
x86_64-redhat-linux-gnu
```

Sección 36.24: \$OSTYPE

Devuelve información sobre el tipo de sistema operativo que se ejecuta en la máquina, p. ej.

```
~> $ echo $OSTYPE
linux-gnu
```

Sección 36.25: \$PATH

La ruta de búsqueda para encontrar los binarios de los comandos. Algunos ejemplos comunes son `/usr/bin` y `/usr/local/bin`.

Cuando un usuario o script intenta ejecutar un comando, se buscan las rutas en `$PATH` para encontrar un archivo coincidente con permiso de ejecución.

Los directorios en `$PATH` están separados por un carácter `:`.

```
~> $ echo "$PATH"
/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin
```

Así, por ejemplo, dado el `$PATH` anterior, si escribes `lss` en el prompt, el shell buscará `/usr/kerberos/bin/lss`, luego `/usr/local/bin/lss`, luego `/bin/lss`, luego `/usr/bin/lss`, en este orden, antes de concluir que no existe tal comando.

Sección 36.26: \$PPID

El ID de proceso (pid) del script o shell padre, es decir, el proceso que invocó al script o shell actual.

```
~> $ echo $$
13016
~> $ echo $PPID
13015
```

Sección 36.27: \$SECONDS

El número de segundos que se ha estado ejecutando un script. Esto puede ser bastante grande si se muestra en el shell:

```
~> $ echo $SECONDS
98834
```

Sección 36.28: \$SHELLOPTS

Una lista de sólo lectura de las opciones que bash proporciona al iniciarse para controlar su comportamiento:

```
~> $ echo $SHELLOPTS
braceexpand:emacs:hashall:histexpand:history:interactive-comments:monitor
```

Sección 36.29: \$_

Muestra el último campo del último comando ejecutado, útil para obtener algo que pasar a otro comando:

```
~> $ ls *.sh; echo $_
testscript1.sh testscript2.sh
testscript2.sh
```

Proporciona la ruta del script si se utiliza antes de cualquier otro comando:

test.sh:

```
#!/bin/bash
echo "$_"
```

Salida:

```
~> $ ./test.sh # ejecutando test.sh
./test.sh
```

Sección 36.30: \$GROUPS

Un array que contiene el número de grupos a los que pertenece el usuario:

```
#!/usr/bin/env bash
echo You are assigned to the following groups:
for group in ${GROUPS[@]}; do
    IFS=: read -r name dummy number members << (getent group $group )
    printf "name: %-10s number: %-15s members: %s\n" "$name" "$number" "$members"
done
```

Sección 36.31: \$LINENO

Muestra el número de línea del script actual. Muy útil para depurar scripts.

```
#!/bin/bash
# esta es la línea 2
echo something # esta es la línea 3
echo $LINENO # La salida será de 4
```

Sección 36.32: \$SHLVL

Cuando se ejecuta el comando bash se abre un nuevo shell. La variable de entorno `$SHLVL` contiene el número de niveles de shell sobre los que se ejecuta el shell actual.

En una nueva ventana de terminal, la ejecución del siguiente comando producirá resultados diferentes en función de la distribución de Linux en uso.

```
echo $SHLVL
```

Usando *Fedora 25*, la salida es "3". Esto indica, que al abrir un nuevo shell, un comando bash inicial se ejecuta y realiza una tarea. El comando bash inicial ejecuta un proceso hijo (otro comando bash) que, a su vez, ejecuta un comando bash final para abrir el nuevo shell. Cuando se abre el nuevo shell, se ejecuta como un proceso hijo de otros 2 procesos shell, de ahí la salida de "3".

En el siguiente ejemplo (dado que el usuario está ejecutando Fedora 25), la salida de `$SHLVL` en un nuevo shell será "3". A medida que se ejecuta cada comando bash, `$SHLVL` se incrementa en uno.

```
~> $ echo $SHLVL
3
~> $ bash
~> $ echo $SHLVL
4
~> $ bash
~> $ echo $SHLVL
5
```

Se puede ver que ejecutar el comando 'bash' (o ejecutar un script bash) abre un nuevo shell. En comparación, la ejecución de un script ejecuta el código en el shell actual.

test1.sh

```
#!/usr/bin/env bash
echo "Hello from test1.sh. My shell level is $SHLVL"
source "test2.sh"
```

test2.sh

```
#!/usr/bin/env bash
echo "Hello from test2.sh. My shell level is $SHLVL"
```

run.sh

```
#!/usr/bin/env bash
echo "Hello from run.sh. My shell level is $SHLVL"
./test1.sh
```

Ejecutar:

```
chmod +x test1.sh && chmod +x run.sh
./run.sh
```

Salida:

```
Hello from run.sh. My shell level is 4
Hello from test1.sh. My shell level is 5
Hello from test2.sh. My shell level is 5
```

Sección 36.33: \$UID

Una variable de sólo lectura que almacena el número de identificación de los usuarios:

```
~> $ echo $UID
12345
```

Capítulo 37: Control del trabajo

Sección 37.1: Listar procesos en segundo plano

```
$ jobs
[1] Running                sleep 500 & (wd: ~)
[2]- Running              sleep 600 & (wd: ~)
[3]+ Running               ./Fritzing &
```

El primer campo muestra los identificadores de las tareas. Los signos + y - que siguen al identificador de dos tareas indican la tarea por defecto y la siguiente tarea candidata por defecto cuando finaliza la tarea por defecto actual, respectivamente. El trabajo por defecto se utiliza cuando se utilizan los comandos `fg` o `bg` sin ningún argumento.

El segundo campo indica el estado del trabajo. El tercer campo es el comando utilizado para iniciar el proceso.

El último campo (`wd: ~`) indica que los comandos `sleep` se iniciaron desde el directorio de trabajo `~` (Home).

Sección 37.2: Traer a primer plano un proceso en segundo plano

```
$ fg %2
sleep 600
```

`%2` especifica el trabajo nº 2. Si se utiliza `fg` sin ningún argumento si trae al primer plano el último proceso puesto en segundo plano.

```
$ fg %?sle
sleep 500
```

`?sle` se refiere al comando de proceso en segundo plano que contiene "sle". Si varios comandos de fondo contienen la cadena de caracteres, se producirá un error.

Sección 37.3: Reiniciar proceso en segundo plano detenido

```
$ bg
[8]+ sleep 600 &
```

Sección 37.4: Ejecutar comando en segundo plano

```
$ sleep 500 &
[1] 7582
```

Pone el comando `sleep` en background. 7582 es el id de proceso del proceso en segundo plano.

Sección 37.5: Detener un proceso en primer plano

Pulsa `Ctrl + Z` para detener un proceso en primer plano y ponerlo en segundo plano.

```
$ sleep 600
^Z
[8]+ Stopped                sleep 600
```

Capítulo 38: Declaración case

Sección 38.1: Declaración de caso simple

En su forma más simple soportada por todas las versiones de bash, la sentencia case ejecuta el caso que coincide con el patrón. El operador `;;` se rompe después de la primera coincidencia, si la hay.

```
#!/bin/bash
var=1
case $var in
1)
    echo "Antartica"
    ;;
2)
    echo "Brazil"
    ;;
3)
    echo "Cat"
    ;;
esac
```

Salidas:

Antartica

Sección 38.2: Declaración de case con caída

Version \geq 4.0

Desde bash 4.0, se introdujo un nuevo operador `;&` que proporciona un mecanismo de [fall through](#) (caída a través de).

```
#!/bin/bash
var=1
case $var in
1)
    echo "Antartica"
    ;&
2)
    echo "Brazil"
    ;&
3)
    echo "Cat"
    ;&
Esac
```

Salidas:

Antartica
Brazil
Cat

Sección 38.3: Pasar sólo si el patrón o patrones siguientes coinciden

Version \geq 4.0

A partir de Bash 4.0, se introdujo otro operador `;&&` que también proporciona un [fall through](#) sólo si los patrones en la(s) sentencia(s) case subsiguiente(s), si los hay, coinciden.

```
#!/bin/bash
var=abc
case $var in
a*)
    echo "Antartica"
    ;;&
xyz)
    echo "Brazil"
    ;;&
*b*)
    echo "Cat"
    ;;&
esac
```

Salidas:

Antartica
Cat

En el ejemplo siguiente, el `abc` coincide con el primer y el tercer caso, pero no con el segundo. Por lo tanto, el segundo caso no se ejecuta.

Capítulo 39: ¿Leer un archivo (flujo de datos, variable) línea por línea (y/o campo por campo)?

Parámetro	Detalles
IFS	Separador de campo interno
file	Un nombre de archivo/ruta
-r	Evita la interpretación de la barra invertida cuando se utiliza con <code>read</code>
-t	Elimina una nueva línea final de cada línea leída por <code>readarray</code>
-d DELIM	Continúa hasta que se lee el primer carácter de DELIM (con <code>read</code>), en lugar de <code>newline</code>

Sección 39.1: Recorrer un fichero línea por línea

```
while IFS= read -r line; do
    echo "$line"
done <file
```

Si el archivo no puede incluir una nueva línea al final, entonces:

```
while IFS= read -r line || [ -n "$line" ]; do
    echo "$line"
done <file
```

Sección 39.2: Recorrer la salida de un comando campo por campo

Supongamos que el separador de campos es `:`

```
while IFS= read -d : -r field || [ -n "$field" ];do
    echo "**$field**"
done <$(ping google.com)
```

O con una tubería:

```
ping google.com | while IFS= read -d : -r field || [ -n "$field" ];do
    echo "**$field**"
done
```

Sección 39.3: Leer líneas de un archivo en un array

```
readarray -t arr <file
```

O con un bucle:

```
arr=()
while IFS= read -r line; do
    arr+=("$line")
done <file
```

Sección 39.4: Leer líneas de una cadena de caracteres en un array

```
var='line 1
line 2
line3'
readarray -t arr <<< "$var"
```


o con un bucle:

```
arr=()
while IFS= read -r line; do
    arr+=("$line")
done <<< "$var"
```

Sección 39.5: Recorrer una cadena de caracteres línea por línea

```
var='line 1
line 2
line3'
while IFS= read -r line; do
    echo "-$line-"
done <<< "$var"
```

O

```
readarray -t arr <<< "$var"
for i in "${arr[@]}";do
    echo "-$i-"
done
```

Sección 39.6: Recorrer en bucle la salida de un comando línea por línea

```
while IFS= read -r line;do
    echo "**$line**"
done < <(ping google.com)
```

o con una tubería:

```
ping google.com |
while IFS= read -r line;do
    echo "**$line**"
done
```

Sección 39.7: Leer un fichero campo por campo

Supongamos que el separador de campos es `:` (dos puntos) en el fichero de *archivos*.

```
while IFS= read -d : -r field || [ -n "$field" ]; do
    echo "$field"
done <file
```

Por un contenido:

```
first : se
con
d:
    Thi rd:
    Fourth
```

La salida es:

```
**first **
** se
con
d**
**
    Thi rd**
**
    Fourth
**
```

Sección 39.8: Leer una cadena de caracteres campo por campo

Supongamos que el separador de campos es `:`

```
var='line: 1
line: 2
line3'
while IFS= read -d : -r field || [ -n "$field" ]; do
    echo "-$field-"
done <<< "$var"
```

Salida:

```
-line-
- 1
line-
- 2
line3
-
```

Sección 39.9: Leer campos de un archivo en un array

Supongamos que el separador de campos es `:`

```
arr=()
while IFS= read -d : -r field || [ -n "$field" ]; do
    arr+=("$field")
done <file
```

Sección 39.10: Leer campos de una cadena de caracteres en un array

Supongamos que el separador de campos es `:`

```
var='1:2:3:4:
newline'
arr=()
while IFS= read -d : -r field || [ -n "$field" ]; do
    arr+=("$field")
done <<< "$var"
echo "${arr[4]}"
```

Salida:

```
newline
```

Sección 39.11: Lee el archivo (/etc/passwd) línea por línea y campo por campo

```
#!/bin/bash
FILENAME="/etc/passwd"
while IFS=: read -r username password userid groupid comment homedir cmdshell
do
    echo "$username, $userid, $comment $homedir"
done < $FILENAME
```

En un archivo de contraseñas unix, la información del usuario se almacena línea por línea, cada línea consiste en información para un usuario separada por el carácter dos puntos (:). En este ejemplo, mientras se lee el archivo línea por línea, la línea también se divide en campos utilizando el carácter dos puntos como delimitador, lo que se indica mediante el valor dado para `IFS`.

Ejemplo de entrada

```
mysql:x:27:27:MySQL Server:/var/lib/mysql:/bin/bash
pulse:x:497:495:PulseAudio System Daemon:/var/run/pulse:/sbin/nologin
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
tomcat:x:91:91:Apache Tomcat:/usr/share/tomcat6:/sbin/nologin
webalizer:x:67:67:Webalizer:/var/www/usage:/sbin/nologin
```

Ejemplo de salida

```
mysql, 27, MySQL Server /var/lib/mysql
pulse, 497, PulseAudio System Daemon /var/run/pulse
sshd, 74, Privilege-separated SSH /var/empty/sshd
tomcat, 91, Apache Tomcat /usr/share/tomcat6
webalizer, 67, Webalizer /var/www/usage
```

Para leer línea por línea y tener toda la línea asignada a la variable, lo siguiente es una versión modificada del ejemplo. Tenga en cuenta que sólo tenemos una variable por línea de nombre mencionado aquí.

```
#!/bin/bash
FILENAME="/etc/passwd"
while IFS= read -r line
do
    echo "$line"
done < $FILENAME
```

Ejemplo de entrada

```
mysql:x:27:27:MySQL Server:/var/lib/mysql:/bin/bash
pulse:x:497:495:PulseAudio System Daemon:/var/run/pulse:/sbin/nologin
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
tomcat:x:91:91:Apache Tomcat:/usr/share/tomcat6:/sbin/nologin
webalizer:x:67:67:Webalizer:/var/www/usage:/sbin/nologin
```

Ejemplo de salida

```
mysql:x:27:27:MySQL Server:/var/lib/mysql:/bin/bash
pulse:x:497:495:PulseAudio System Daemon:/var/run/pulse:/sbin/nologin
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
tomcat:x:91:91:Apache Tomcat:/usr/share/tomcat6:/sbin/nologin
webalizer:x:67:67:Webalizer:/var/www/usage:/sbin/nologin
```

Capítulo 40: Secuencia de ejecución del archivo

`.bash_profile`, `.bash_login`, `.bashrc` y `.profile` hacen prácticamente lo mismo: configurar y definir funciones, variables y cosas por el estilo.

La principal diferencia es que `.bashrc` se llama al abrirse una ventana no de inicio de sesión pero interactiva, y `.bash_profile` y los demás se llaman para un intérprete de comandos de inicio de sesión. Mucha gente tiene su `.bash_profile` o similar llamada `.bashrc` de todos modos.

Sección 40.1: `profile` vs `.bash_profile` (y `.bash_login`)

`.profile` es leído por la mayoría de los shells al iniciarse, incluyendo `bash`. Sin embargo, `.bash_profile` se usa para configuraciones específicas de `bash`. Para código de inicialización general, ponlo en `.profile`. Si es específico de `bash`, usa `.bash_profile`.

`.profile` no está diseñado específicamente para `bash`, sino `.bash_profile`. (`.profile` es para Bourne y otros shells similares, en los que se basa `bash`) `Bash` recurrirá a `.profile` si no se encuentra `.bash_profile`.

`.bash_login` es una alternativa a `.bash_profile`, si no se encuentra. Generalmente es mejor usar `.bash_profile` o `.profile` en su lugar.

Capítulo 41: Dividir archivos

A veces es útil dividir un archivo en varios archivos separados. Si tienes archivos grandes, puede ser una buena idea dividirlos en trozos más pequeños.

Sección 41.1: Dividir un archivo

Si ejecuta el comando `split` sin ninguna opción, dividirá un archivo en 1 o más archivos separados que contengan hasta 1000 líneas cada uno.

`split file`

Esto creará archivos llamados `xaa`, `xab`, `xac`, etc, cada uno con un máximo de 1000 líneas. Como puede ver, todos ellos llevan por defecto la letra `x` como prefijo. Si el fichero inicial tuviera menos de 1000 líneas, sólo se crearía uno de estos ficheros.

Para cambiar el prefijo, añada el prefijo deseado al final de la línea de comandos

`split file customprefix`

Ahora se crearán archivos con los nombres `customprefixaa`, `customprefixab`, `customprefixac`, etc.

Para especificar el número de líneas de salida por archivo, utilice la opción `-l`. Lo siguiente dividirá un archivo en un máximo de 5000 líneas

`split -l5000 file`

O

`split --lines=5000 file`

Alternativamente, puede especificar un número máximo de bytes en lugar de líneas. Esto se hace utilizando las opciones `-b` o `--bytes`. Por ejemplo, para permitir un máximo de 1MB

`split --bytes=1MB file`

Capítulo 42: Transferencia de archivos mediante scp

Sección 42.1: scp transferir archivo

Para transferir un archivo de forma segura a otra máquina - escriba:

```
scp file1.txt tom@server2:$HOME
```

Este ejemplo presenta la transferencia de `file1.txt` desde nuestro host al directorio home del usuario `tom` del `server2`.

Sección 42.2: scp transferir varios archivos

`scp` también puede utilizarse para transferir múltiples ficheros de un servidor a otro. A continuación, se muestra un ejemplo de transferencia de todos los archivos del directorio `my_folder` con extensión `.txt` al `server2`. En el ejemplo de abajo todos los archivos serán transferidos al directorio home del usuario `tom`.

```
scp /my_folder/*.txt tom@server2:$HOME
```

Sección 42.3: Descarga de archivos mediante scp

Para descargar un archivo del servidor remoto a la máquina local - escriba:

```
scp tom@server2:$HOME/file.txt /local/machine/path/
```

Este ejemplo muestra cómo descargar el archivo llamado `file.txt` desde el directorio personal del usuario `tom` al directorio actual de nuestra máquina local.

Capítulo 43: Tuberías

Sección 43.1: Uso de | &

|& conecta la salida estándar y el error estándar del primer comando con el segundo, mientras que | sólo conecta la salida estándar del primer comando con el segundo.

En este ejemplo, la página se descarga a través de curl. con -v opción curl escribe alguna información en stderr incluyendo, la página descargada se escribe en stdout. El título de la página se encuentra entre <title> y </title>.

```
curl -vs 'http://www.google.com/' |& awk  
'/Host:/{print}/<title>/{match($0,/<title>(.*?)</title>/,a);print a[1]}'
```

La salida es:

```
> Host: www.google.com  
Google
```

Pero con | se imprimirá mucha más información, es decir, la que se envía a stderr porque sólo stdout se canaliza al siguiente comando. En este ejemplo todas las líneas excepto la última (Google) fueron enviadas a stderr por curl:

```
* Hostname was NOT found in DNS cache  
* Trying 172.217.20.228...  
* Connected to www.google.com (172.217.20.228) port 80 (#0)  
> GET / HTTP/1.1  
> User-Agent: curl/7.35.0  
> Host: www.google.com  
> Accept: */*  
>  
* HTTP 1.0, assume close after body  
< HTTP/1.0 200 OK  
< Date: Sun, 24 Jul 2016 19:04:59 GMT  
< Expires: -1  
< Cache-Control: private, max-age=0  
< Content-Type: text/html; charset=ISO-8859-1  
< P3P: CP="This is not a P3P policy! See  
https://www.google.com/support/accounts/answer/151657?hl=en for more info."  
< Server: gws  
< X-XSS-Protection: 1; mode=block  
< X-Frame-Options: SAMEORIGIN  
< Set-Cookie:  
NID=82=jX0yZLPPUE7u13kKNevUCDg8yG9Ze_C03o0IMEopOSKL0mMITEagIE816G55L2wrTlQwgXkhq4ApFvvYEoaWfoEoq  
2T0sBTuQVdsIFULj9b208X3500sAgUnc3a3JnTRBqelMcuS9QkQA;  
expires=Mon, 23-Jan-2017 19:04:59 GMT;  
path=/; domain=.google.com; HttpOnly  
< Accept-Ranges: none  
< Vary: Accept-Encoding  
< X-Cache: MISS from jetsib_appliance  
< X-Loop-Control: 5.202.190.157 81E4F9836653D5812995BA53992F8065  
< Connection: close  
<  
{ [data not shown]  
* Closing connection 0  
Google
```

Sección 43.2: Mostrar todos los procesos paginados

```
ps -e | less
```

`ps -e` muestra todos los procesos, su salida se conecta a la entrada de `more` mediante `|`, `less` página los resultados.

Sección 43.3: Modificar la salida continua de un comando

```
~$ ping -c 1 google.com # salida sin modificar
PING google.com (16.58.209.174) 56(84) bytes of data.
64 bytes from wk-in-f100.1e100.net (16.58.209.174): icmp_seq=1 ttl=53 time=47.4 ms
~$ ping google.com | grep -o '^[0-9]\+[^()]\+' # salida modificada
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
...
```

La tubería (`|`) conecta la salida estándar de `ping` con la entrada estándar de `grep`, que la procesa inmediatamente. Algunos otros comandos como `sed` almacenan por defecto su `stdin`, lo que significa que tiene que recibir suficientes datos, antes de imprimir nada, causando potencialmente retrasos en el procesamiento posterior.

Capítulo 44: Gestión de la variable de entorno PATH

Parámetro	Detalles
PATH	Variable de entorno Path

Sección 44.1: Añadir una ruta a la variable de entorno PATH

La variable de entorno PATH se define generalmente en `~/.bashrc` o `~/.bash_profile` o `/etc/profile` o `~/.profile` o `/etc/bash.bashrc` (archivo de configuración de Bash específico de la distribución).

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/usr/lib/jvm/jdk1.8.0_92/bin:/usr/lib/jvm/jdk1.8.0_92/db/bin:/usr/lib/jvm/jdk1.8.0_92/jre/bin
```

Ahora, si queremos añadir una ruta (por ejemplo `~/bin`) a la variable PATH:

```
PATH=~/bin:$PATH
# o
PATH=$PATH:~/bin
```

Pero esto modificará el PATH sólo en el shell actual (y su subshell). Una vez que salga del shell, esta modificación desaparecerá.

Para hacerlo permanente, tenemos que añadir ese trozo de código al archivo `~/.bashrc` (o lo que sea) y volver a cargar el archivo.

Si ejecutas el siguiente código (en terminal), añadirá `~/bin` al PATH permanentemente:

```
echo 'PATH=~/bin:$PATH' >> ~/.bashrc && source ~/.bashrc
```

Explicación:

- `echo 'PATH=~/bin:$PATH' >> ~/.bashrc` añade la línea `PATH=~/bin:$PATH` al final del archivo `~/.bashrc` (puedes hacerlo con un editor de texto).
- `source ~/.bashrc` recarga el archivo `~/.bashrc`.

Esto es un poco de código (ejecutado en terminal) que comprobará si una ruta ya está incluida y añadirá la ruta sólo si no es así:

```
path=~/bin # ruta que debe incluirse
bashrc=~/bashrc # archivo bash a escribir y recargar
# ejecute el siguiente código sin modificar
echo $PATH | grep -q "\(^|:|:)$path\(:|/|/{0,1}\}$)" || echo "PATH=\$PATH:$path" >> "$bashrc";
source "$bashrc"
```

Sección 44.2: Eliminar una ruta de la variable de entorno PATH

Para eliminar un PATH de una variable de entorno PATH, es necesario editar el archivo `~/.bashrc` o `~/.bash_profile` o `/etc/profile` o `~/.profile` o `/etc/bash.bashrc` (específico de la distro) y eliminar la asignación para esa ruta en particular.

En lugar de encontrar la asignación exacta, podría simplemente hacer un reemplazo en el `$PATH` en su etapa final.

Lo siguiente eliminará `$path` de `$PATH` de forma segura:

```
path=~/bin
PATH="$(echo "$PATH" | sed -e "s#\(^|:|:)$path\(:|/|/{0,1}\}$#" -e 's#:#+:#g' -e 's#^:|:|:##g')"
```

Para hacerlo permanente, tendrá que añadirlo al final de su archivo de configuración bash.

Puedes hacerlo de forma funcional:

```

rpath(){
    for path in "$@";do
        PATH="$(echo "$PATH" |sed -e "s#\(^\|:\)\$(echo "$path" |sed -e 's/[^^]/[&]/g' -e 's/\^/\\^/g')\(:\|/\{0,1\}$\)#\\1\\2#" -e 's#:\++#:#g' -e 's#^\|:##g')"
    done
    echo "$PATH"
}

PATH="$(rpath ~/bin /usr/local/sbin /usr/local/bin)"
PATH="$(rpath /usr/games)"
# etc ...

```

Esto facilitará la gestión de rutas múltiples.

Notas:

- Tendrás que añadir estos códigos en el archivo de configuración de Bash (~/.bashrc o lo que sea).
- Ejecuta `source ~/.bashrc` para recargar el archivo de configuración de Bash (~/.bashrc).

Capítulo 45: Separación de palabras

Parámetro	Detalles
IFS	Separador de campo interno
-x	Imprimir comandos y sus argumentos a medida que se ejecutan (opción de Shell)

Sección 45.1: ¿Qué, cuándo y por qué?

Cuando el shell realiza una *expansión de parámetros*, una *sustitución de comandos* o una *expansión aritmética* o de *variables*, busca límites de palabra en el resultado. Si se encuentra algún límite de palabra, el resultado se divide en varias palabras en esa posición. El límite de palabra se define mediante una variable del shell **IFS** (Internal Field Separator). El valor predeterminado para **IFS** es espacio, tabulador y nueva línea, es decir, la división de palabras se producirá en estos tres caracteres de espacio en blanco si no se evita explícitamente.

```
set -x
var='I am
a
multiline string'
fun() {
    echo "$1-"
    echo "$2*"
    echo "$3."
}
fun $var
```

En el ejemplo anterior, así es como se ejecuta la función `fun`:

```
fun I am a multiline string
```

`$var` se divide en 5 args, sólo se imprimirán `I`, `am` y `a`.

Sección 45.2: Efectos negativos de la división de palabras

```
$ a='I am a string with spaces'
$ [ $a = $a ] || echo "didn't match"
bash: [: too many arguments
didn't match
```

`[$a = $a]` se interpretó como `[I am a string with spaces = I am a string with spaces]`. `[` es el comando de **test** para el que `I am a string with spaces` no es un único argumento, ¡¡¡sino que son 6 argumentos!!!

```
$ [ $a = something ] || echo "didn't match"
bash: [: too many arguments
didn't match
```

`[$a = something]` se interpretaba como `[I am a string with spaces = something]`

```
$ [ $(grep . file) = 'something' ]
bash: [: too many arguments
```

El comando **grep** devuelve una cadena de caracteres multilínea con espacios, así que puedes imaginarte cuántos argumentos hay...:D

Vea qué, cuándo y por qué para lo más básico.

Sección 45.3: Utilidad de la división de palabras

Hay algunos casos en los que la división de palabras puede ser útil:

Rellenando array:

```
arr=$(grep -o '[0-9]\+' file)
```

Esto llenará `arr` con todos los valores numéricos encontrados en el `file`.

Recorrer palabras separadas por espacios:

```
words='foo bar baz'
for w in $words;do
    echo "W: $w"
done
```

Salida:

```
W: foo
W: bar
W: baz
```

Pasar parámetros separados por espacios que no contengan espacios en blanco:

```
packs='apache2 php php-mbstring php-mysql'
sudo apt-get install $packs
```

O

```
packs='
apache2
php
php-mbstring
php-mysql
'
sudo apt-get install $packs
```

Esto instalará los paquetes. Si escribes `$packs` entre comillas, se producirá un error.

Sin citarlo, `$packs` está enviando todos los nombres de paquetes separados por espacios como argumentos a `apt-get`, mientras que citando enviará la cadena de caracteres `$packs` como un único argumento y entonces `apt-get` intentará instalar un paquete llamado `apache2 php php-mbstring php-mysql` (para el primero) que obviamente no existe.

Vea qué, cuándo y por qué para lo más básico.

Sección 45.4: Dividir por cambios de separador

Podemos hacer una simple sustitución de separadores de espacio a nueva línea, como en el siguiente ejemplo.

```
echo $sentence | tr " " "\n"
```

Dividirá el valor de la sentencia variable y lo mostrará línea por línea respectivamente.

Sección 45.5: Dividir con IFS

Para ser más claros, vamos a crear un script llamado `showarg`:

```
#!/usr/bin/env bash
printf "%d args:" $#
printf " <%s>" "$@"
echo
```

Veamos ahora las diferencias:

```
$ var="This is an example"
$ showarg $var
4 args: <This> <is> <an> <example>
```

`$var` se divide en 4 args. IFS es caracteres de espacio en blanco y por lo tanto la división de palabras se produjo en espacios.

```
$ var="This/is/an/example"
$ showarg $var
1 args: <This/is/an/example>
```

En la división de palabras anterior no se produjo porque no se encontraron los caracteres IFS.

Ahora pongamos `IFS=/`

```
$ IFS=/
$ var="This/is/an/example"
$ showarg $var
4 args: <This> <is> <an> <example>
```

El `$var` se divide en 4 argumentos no un único argumento.

Sección 45.6: IFS y división de palabras

Vea qué, cuándo y por qué si no conoce la afiliación de IFS a la división de palabras

establezcamos el IFS sólo con caracteres de espacio:

```
set -x
var='I am
a
multiline string'
IFS=' '
fun() {
    echo "$1-"
    echo "$2*"
    echo ".$3."
}
fun $var
```

Esta vez la división de palabras sólo funcionará en espacios. La función `fun` se ejecutará así:

```
fun I 'am
a
multiline' string
```

`$var` se divide en 3 args. `I`, `am` y `multiline` y `string` se imprimirán

Pongamos el IFS sólo en nueva línea:

```
IFS=$'\n'
```

...

Ahora la diversión se ejecutará como:

```
fun 'I am' a 'multiline string'
```

```
$var
```

 se divide en 3 args. I am, a, multiline string se imprimirán.

Veamos qué ocurre si establecemos IFS como nullstring:

```
IFS=
```

...

Esta vez la diversión se ejecutará así:

```
fun 'I am  
a  
multiline string'
```

Esta vez la diversión se ejecutará así:

```
$var
```

 no se divide, es decir, sigue siendo un único argumento.

Puede evitar la división de palabras estableciendo el IFS en nullstring

Una forma general de evitar la división de palabras es utilizar comillas dobles:

```
fun "$var"
```

evitará la división de palabras en todos los casos mencionados anteriormente, es decir, la función `fun` se ejecutará con un solo argumento.

Capítulo 46: Evitar la fecha utilizando printf

En Bash 4.2, se introdujo una conversión de tiempo integrada en el shell para `printf`: la especificación de formato `%(datefmt)T` hace que `printf` emita la cadena de caracteres de fecha-hora correspondiente a la cadena de formato `datefmt` tal y como la entiende `strftime`.

Sección 46.1: Obtener la fecha actual

```
$ printf '%(%F)T\n'
2016-08-17
```

Sección 46.2: Establecer la variable a la hora actual

```
$ printf -v now '%(%T)T'
$ echo "$now"
12:42:47
```

Capítulo 47: Uso de "trap" para reaccionar ante señales y eventos del sistema

Parámetro	Significado
-p	Lista de trampas instaladas actualmente
-l	Lista de nombres de señales y números correspondientes

Sección 47.1: Introducción: limpiar archivos temporales

Puede utilizar el comando `trap` para "atrapar" señales; este es el equivalente en shell de la llamada `signal()` o `sigaction()` en C y la mayoría de los otros lenguajes de programación para atrapar señales.

Uno de los usos más comunes de `trap` es limpiar los archivos temporales tanto en una salida esperada como inesperada.

Desgraciadamente, no hay suficientes shell scripts que lo hagan :-)

```
#!/bin/sh

# Hacer una función de limpieza
cleanup() {
    rm --force -- "${tmp}"
}

# Atrapa el grupo especial "EXIT", que siempre se ejecuta cuando el shell sale.
trap cleanup EXIT

# Crear un archivo temporal
tmp="$(mktemp -p /tmp tmpfileXXXXXX)"

echo "Hello, world!" >> "${tmp}"

# No es necesario rm -f "$tmp". La ventaja de usar EXIT es que sigue funcionando
# incluso si hubo un error o si usaste exit.
```

Sección 47.2: Capturar SIGINT o Ctrl+C

La trampa se reinicia para las subshells, por lo que el `sleep` seguirá actuando ante la señal `SIGINT` enviada por `^C` (normalmente saliendo), pero el proceso padre (es decir, el script de la shell) no lo hará.

```
#!/bin/sh

# Ejecuta un comando en la señal 2 (SIGINT, que es lo que envía ^C)
sigint() {
    echo "Killed subshell!"
}
trap sigint INT

# O utilice el comando no-op para no obtener salida
# trap : INT

# Este será asesinado en el primer ^C
echo "Sleeping..."
sleep 500

echo "Sleeping..."
sleep 500
```


Y una variante que aún permite salir del programa principal pulsando `^C` dos veces en un segundo:

```
last=0
allow_quit() {
    [ $(date +%s) -lt $(( $last + 1 )) ] && exit
    echo "Press ^C twice in a row to quit"
    last=$(date +%s)
}
trap allow_quit INT
```

Sección 47.3: Acumular una lista de trabajos trampa para ejecutar a la salida

¿Te has olvidado alguna vez de añadir un `trap` para limpiar un archivo temporal o realizar otras tareas al salir?

¿Alguna vez has puesto una trampa que canceló otra?

Este código hace que sea fácil añadir las cosas que se deben hacer en la salida de un elemento a la vez, en lugar de tener una gran declaración `trap` en algún lugar de su código, que puede ser fácil de olvidar.

```
# on_exit y add_on_exit
# Uso:
# add_on_exit rm -f /tmp/foo
# add_on_exit echo "I am exiting"
# tempfile=$(mktemp)
# add_on_exit rm -f "$tempfile"
# Basado en http://www.linuxjournal.com/content/use-bash-trap-statement-cleanup-temporary-files
function on_exit()
{
    for i in "${on_exit_items[@]}"
    do
        eval $i
    done
}
function add_on_exit()
{
    local n=${#on_exit_items[*]}
    on_exit_items[$n]="$*"
    if [[ $n -eq 0 ]]; then
        trap on_exit EXIT
    fi
}
```

Sección 47.4: Terminar procesos hijos al salir

Las expresiones trampa no tienen por qué ser funciones o programas individuales, también pueden ser expresiones más complejas.

Combinando `jobs -p` y `kill`, podemos matar todos los procesos hijos del intérprete de comandos al salir:

```
trap 'jobs -p | xargs kill' EXIT
```

Sección 47.5: Reaccionar al cambiar el tamaño de la ventana del terminal

Existe una señal `WINCH` (WINdowCHange), que se dispara cuando se cambia el tamaño de una ventana de terminal.

```
declare -x rows cols

update_size(){
    rows=$(tput lines) # obtener líneas reales del terminal
    cols=$(tput cols) # obtener columnas reales del terminal
    echo DEBUG terminal window has no $rows lines and is $cols characters wide
}

trap update_size WINCH
```

Capítulo 48: Cadena de mando y operaciones

Existen algunos medios para encadenar comandos. Simples como sólo un `;` o más complejas como cadenas lógicas que se ejecutan dependiendo de algunas condiciones. La tercera es la canalización de comandos, que efectivamente entrega los datos de salida al siguiente comando de la cadena.

Sección 48.1: Contar la ocurrencia de un patrón de texto

El uso de una tubería hace que la salida de un comando sea la entrada del siguiente.

```
ls -l | grep -c ".conf"
```

En este caso, la salida del comando `ls` se utiliza como entrada del comando `grep`. El resultado será el número de ficheros que incluyen `".conf"` en su nombre.

Esto se puede utilizar para construir cadenas de comandos posteriores tan largas como sea necesario:

```
ls -l | grep ".conf" | grep -c .
```

Sección 48.2: Transferir la salida cmd de root a un archivo de usuario

A menudo uno quiere mostrar el resultado de un comando ejecutado por root a otros usuarios. El comando `tee` permite escribir fácilmente un archivo con permisos de usuario desde un comando ejecutado como root:

```
su -c ifconfig | tee ~/results-of-ifconfig.txt
```

Sólo `ifconfig` se ejecuta como root.

Sección 48.3: Encadenamiento lógico de comandos con `&&` y `||`

`&&` encadena dos órdenes. El segundo sólo se ejecuta si el primero sale con éxito. `||` encadena dos comandos. Pero el segundo sólo se ejecuta si el primero sale con error.

```
[ a = b ] && echo "yes" || echo "no"
```

```
# si desea ejecutar más comandos dentro de una cadena lógica, utilice llaves
# que designan un bloque de comandos
# Necesitan un ; antes del corchete de cierre para que bash pueda diferenciar de otros usos
# de llaves
```

```
[ a = b ] && { echo "let me see."
              echo "hmmm, yes, i think it is true" ; } \
|| { echo "as i am in the negation i think "
      echo "this is false. a is a not b." ; }
```

```
# ten en cuenta el uso del signo de continuación de línea \
# sólo necesario para encadenar bloque sí con || ....
```

Sección 48.4: Encadenamiento en serie de comandos con punto y coma

Un punto y coma separa sólo dos comandos.

```
echo "i am first" ; echo "i am second" ; echo " i am third"
```

Sección 48.5: Encadenar comandos con `|`

El `|` toma la salida del comando izquierdo y la canaliza como entrada al comando derecho. Tenga en cuenta que esto se hace en un subshell. Por lo tanto, no puede establecer valores de variables del proceso de llamada dentro de una tubería.

```
find . -type f -a -iname '*.mp3' | \  
    while read filename; do  
        mute --noise "$filename"  
    done
```

Capítulo 49: Tipo de shells

Sección 49.1: Iniciar un shell interactivo

bash

Sección 49.2 Detectar el tipo de Shell

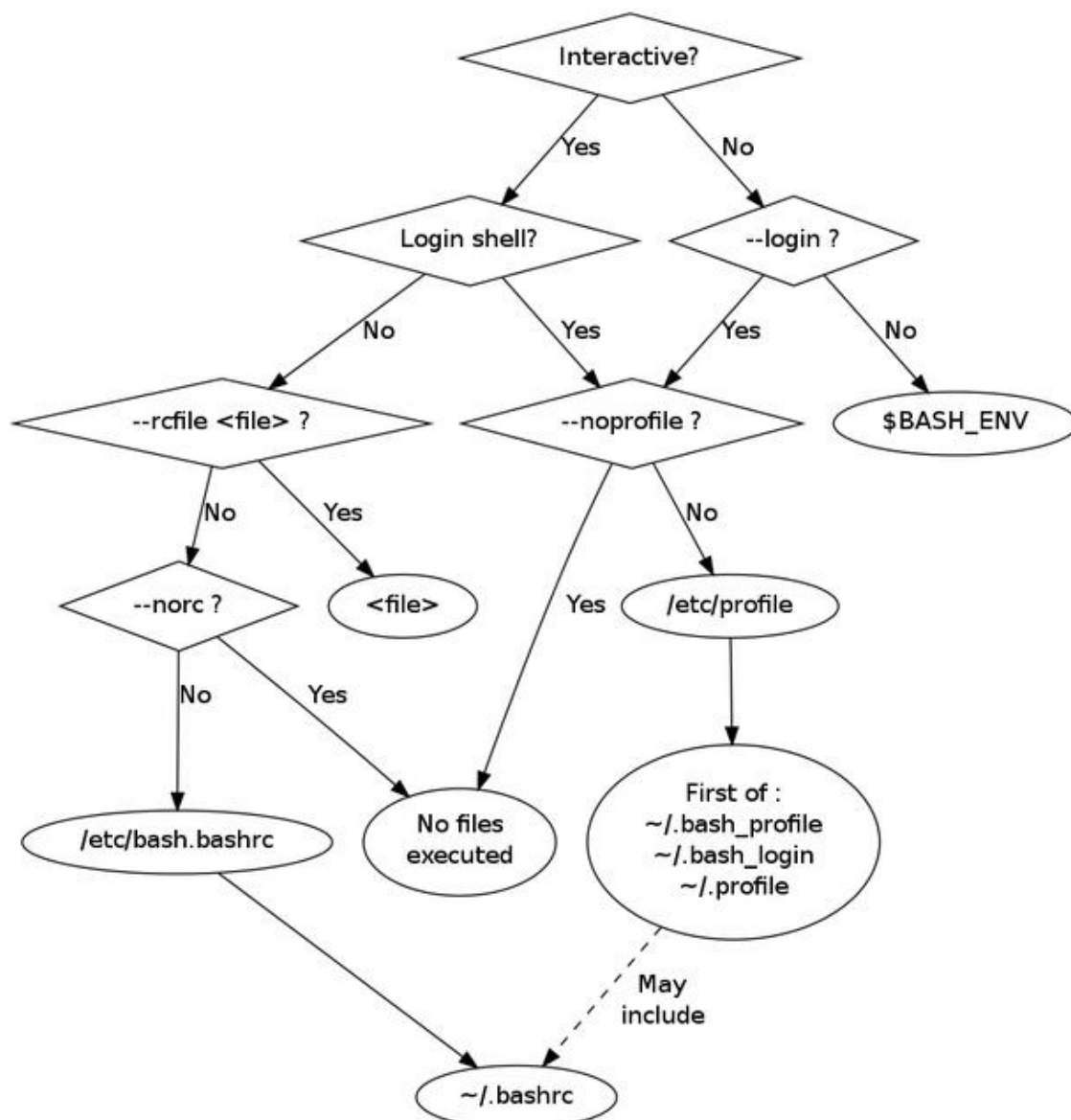
```
shopt -q login_shell && echo 'login' || echo 'not-login'
```

Sección 49.3: Introducción a los archivos punto

En Unix, los archivos y directorios que comienzan con un punto suelen contener configuraciones para un programa o una serie de programas específicos. Los archivos que empiezan por punto suelen estar ocultos para el usuario, por lo que tendría que ejecutar `ls -a` para verlos.

Un ejemplo de archivo punto es `.bash_history`, que contiene los últimos comandos ejecutados, suponiendo que el usuario esté utilizando Bash.

Existen varios archivos que se obtienen cuando se entra en el intérprete de comandos Bash. La imagen de abajo, tomada de [este sitio](#), muestra el proceso de decisión detrás de la elección de los archivos a la fuente en el arranque.



Capítulo 50: Salida de scripts en color (multiplataforma)

Sección 50.1: color-output.sh

En la sección de apertura de un script bash, es posible definir algunas variables que funcionan como ayudantes para colorear o formatear la salida del terminal durante la ejecución del script.

Las distintas plataformas utilizan diferentes secuencias de caracteres para expresar el color. Sin embargo, existe una utilidad llamada `tput` que funciona en todos los sistemas `*nix` y devuelve cadenas de caracteres de colores de terminal específicas de la plataforma a través de una API multiplataforma coherente.

Por ejemplo, para almacenar la secuencia de caracteres que hace que el texto del terminal se vuelva rojo o verde:

```
red=$(tput setaf 1)
green=$(tput setaf 2)
```

O, para almacenar la secuencia de caracteres que restablece el texto a la apariencia por defecto:

```
reset=$(tput sgr0)
```

Entonces, si el script BASH necesita mostrar salidas de diferentes colores, esto se puede lograr con:

```
cho "${green}Success!${reset}" echo "${red}Failure.${reset}"
```

Capítulo 51: co-procesos

Sección 51.1: Hola Mundo

```
# crear el coproceso
coproc bash

# envíale una orden (echo a)
echo 'echo Hello World' >&"${COPROC[1]}"

# leer una línea de su salida
read line <&"${COPROC[0]}"

# mostrar la línea
echo "$line"
```

La salida es "Hello World".

Capítulo 52: Variables tipográficas

Sección 52.1: Declarar variables de tipado débil

declare es un comando interno de bash. (el comando interno usa **help** para mostrar “manpage”). Se usa para mostrar y definir variables o mostrar cuerpos de funciones.

Sintaxis: **declare** [opciones] [nombre[=valor]]...

```
# se utilizan para definir
# un entero
declare -i myInteger
declare -i anotherInt=10
# un array con valores
declare -a anArray=( one two three )
# an array asociativo
declare -A assocArray=( [element1]="something" [second]=anotherthing )
# tenga en cuenta que bash reconoce el contexto de cadena de caracteres dentro de []

# existen algunos modificadores
# contenido en mayúsculas
declare -u big='this will be uppercase'
# lo mismo para las minúsculas
declare -l small='THIS WILL BE LOWERCASE'

# array de sólo lectura
declare -ra constarray=( eternal true and unchangeable )

# exportar entero a entorno
declare -xi importantInt=42
```

También puede utilizar el **+** que quita el atributo dado. Mayormente inútil, sólo para completar.

Para visualizar variables y/o funciones también hay algunas opciones

```
# impresión de variables y funciones definidas
declare -f
# restringir la salida sólo a funciones
declare -F # si la depuración imprime el número de línea y el nombre de archivo definidos en too
```


Capítulo 53: Trabajos a horas concretas

Sección 53.1: Ejecutar el trabajo una vez a una hora determinada

Nota: **at** no está instalado por defecto en la mayoría de las distribuciones modernas.

Para ejecutar un trabajo una vez en otro momento que no sea ahora, en este ejemplo de las 5 de la tarde, puede utilizar

```
echo "somecommand &" | at 5pm
```

Si desea capturar la salida, puede hacerlo de la forma habitual:

```
echo "somecommand > out.txt 2>err.txt &" | at 5pm
```

en entiende muchos formatos de tiempo, por lo que también se puede decir

```
echo "somecommand &" | at now + 2 minutes
echo "somecommand &" | at 17:00
echo "somecommand &" | at 17:00 Jul 7
echo "somecommand &" | at 4pm 12.03.17
```

Si no se indica el año o la fecha, se asume que la próxima vez que ocurra será a la hora especificada. Así, si indica una hora que ya ha pasado hoy, asumirá que es mañana, y si indica un mes que ya ha pasado este año, asumirá que es el año que viene.

Esto también funciona junto con **nohup** como era de esperar.

```
echo "nohup somecommand > out.txt 2>err.txt &" | at 5pm
```

Hay algunos comandos más para controlar los trabajos temporizados:

- **atq** lista todos los trabajos temporizados (**atqueue**)
- **atrm** elimina un trabajo temporizado (**atremove**)
- **batch** hace básicamente lo mismo que **at**, pero ejecuta trabajos sólo cuando la carga del sistema es inferior a 0,8

Todos los comandos se aplican a los trabajos del usuario conectado. Si se ha iniciado sesión como root, por supuesto se gestionan los trabajos de todo el sistema.

Sección 53.2: Haciendo trabajos a horas especificadas repetidamente usando systemd.timer

systemd proporciona una implementación moderna de **cron**. Para ejecutar un script periódicamente se necesita un servicio y un archivo de temporizador. Los archivos de servicio y temporizador deben colocarse en `/etc/systemd/{system,user}`. El archivo de servicio:

```
[Unit]
```

```
Description=my script or programm does the very best and this is the description
```

```
[Service]
```

```
# el type es importante
```

```
Type=simple
```

```
# programa/script a llamar. Usar siempre rutas absolutas
```

```
# y redirigir STDIN y STDERR ya que no hay terminal mientras se ejecuta
```

```
ExecStart=/absolute/path/to/someCommand >>/path/to/output 2>/path/to/STDERRoutput
```

```
# ¡¡¡¡NO instalar sección!!!! Es gestionado por las propias instalaciones del temporizador.
```

```
# [Install]
```

```
# WantedBy=multi-user.target
```

A continuación, el archivo del temporizador:

```
[Unit]
Description=my very first systemd timer
[Timer]
# La sintaxis para las especificaciones de fecha/hora es Y-m-d H:M:S
# un * significa «cada», y también se puede dar una lista de elementos separados por comas
# *-*- * ,15,30,45:00 dice cada año, cada mes, cada día, cada hora,
# en el minuto 15,30,45 y cero segundos

OnCalendar=*-*- * *:01:00
# este se ejecuta cada hora a un minuto cero segundo por ejemplo 13:01:00
```

Capítulo 54: Manejo del indicador del sistema

Escape

Detalles

<code>\a</code>	Un personaje de campanillas.
<code>\d</code>	La fecha, en formato "Weekday Month Date" (por ejemplo, "Tue May 26").
<code>\D{FORMAT}</code>	El <code>FORMAT</code> se pasa a <code>strftime(3)</code> y el resultado se inserta en la cadena de caracteres de consulta; un <code>FORMAT</code> vacío da como resultado una representación de la hora específica de la localidad. Los corchetes son obligatorios.
<code>\e</code>	Un carácter de escape. <code>\033</code> también funciona, por supuesto.
<code>\h</code>	El nombre de host, hasta la primera <code>.</code> (es decir, sin la parte del dominio)
<code>\H</code>	El nombre de host eventualmente con parte de dominio
<code>\j</code>	El número de trabajos gestionados actualmente por el shell.
<code>\l</code>	El nombre base del dispositivo terminal del intérprete de comandos.
<code>\n</code>	Una nueva línea.
<code>\r</code>	Un retorno de carro.
<code>\s</code>	El nombre del intérprete de comandos, el nombre base de <code>\$0</code> (la parte que sigue a la barra final).
<code>\t</code>	La hora, en formato de 24 horas HH:MM:SS.
<code>\T</code>	La hora, en formato de 12 horas HH:MM:SS.
<code>@</code>	La hora, en formato de 12 horas am/pm.
<code>\A</code>	La hora, en formato de 24 horas HH:MM.
<code>\u</code>	El nombre del usuario actual.
<code>\v</code>	La versión de Bash (por ejemplo, 2.00)
<code>\V</code>	La versión de Bash, versión + nivel de parche (por ejemplo, 2.00.0)
<code>\w</code>	El directorio de trabajo actual, con <code>\$HOME</code> abreviado con una tilde (utiliza la variable <code>\$PROMPT_DIRTRIM</code>).
<code>\W</code>	El nombre base de <code>\$PWD</code> , con <code>\$HOME</code> abreviado con tilde.
<code>!</code>	El número de historial de este comando.
<code>#</code>	El número de este comando.
<code>\$</code>	Si el uid efectivo es <code>0</code> , <code>#</code> , en caso contrario <code>\$</code> .
<code>\NNN</code>	El carácter cuyo código ASCII es el valor octal <code>NNN</code> .
<code>\</code>	Una barra invertida.
<code>\[</code>	Inicia una secuencia de caracteres no imprimibles. Esto podría utilizarse para incrustar una secuencia de control de terminal en el prompt.
<code>\]</code>	Finaliza una secuencia de caracteres no imprimibles.

Sección 54.1: Uso de la variable de entorno `PROMPT_COMMAND`

Cuando se ejecuta el último comando de una instancia bash interactiva, se muestra la variable `PS1` evaluada. Antes de mostrar `PS1` bash comprueba si `PROMPT_COMMAND` está activado. El valor de esta variable debe ser un programa o script invocable. Si esta variable está activada este programa/script es llamado ANTES de que se muestre el prompt `PS1`.

```
# sólo una función estúpida, vamos a utilizar para demostrar
# comprobamos la fecha si Hora es 12 y Minuto es menor que 59
lunchbreak(){
    if (( $(date +%H) == 12 && $(date +%M) < 59 )); then
        # e imprimir color \033[ inicia la secuencia de escape
        # 5; es atributo intermitente
        # 2; significa negrita
        # 31 dice rojo
        printf "\033[5;1;31m mind the lunch break\033[0m\n";
    else
        printf "\033[33m still working...\033[0m\n";
    fi;
}

# activarlo
export PROMPT_COMMAND=lunchbreak
```

Sección 54.2: Utilizar PS2

PS2 se muestra cuando un comando se extiende a más de una línea y bash espera más pulsaciones. También se muestra cuando se introduce un comando compuesto como **while...do..done** y similares.

```
export PS2="would you please complete this command?\n"
# ahora introduzca un comando que se extienda al menos dos líneas para ver PS2
```

Sección 54.3: Utilizar PS3

Cuando se ejecuta la sentencia **select**, muestra los elementos dados prefijados con un número y luego muestra el prompt PS3:

```
export PS3=" To choose your language type the preceding number : "
select lang in EN CA FR DE; do
    # compruebe aquí la entrada hasta que sea válida.
    break
done
```

Sección 54.4: Utilizar PS4

PS4 se muestra cuando bash está en modo de depuración.

```
#!/usr/bin/env bash
# activar la depuración
set -x

# define a stupid_func
stupid_func(){
    echo I am line 1 of stupid_func
    echo I am line 2 of stupid_func
}

# configuración del indicador PS4 "DEBUG"
export PS4='\nDEBUG level:$SHLVL subshell-level: $BASH_SUBSHELL \nsource-file:${BASH_SOURCE}
line:${LINENO} function:${FUNCNAME[0]}:${FUNCNAME[0]}(): }\nstatement: '

# una declaración normal
echo something

# llamada de función
stupid_func

# una serie de comandos que se ejecutan en una subshell
( ls -l | grep 'x' )
```

Sección 54.5: Utilizar PS1

PS1 es el prompt normal del sistema que indica que bash espera a que se escriban comandos. Entiende algunas secuencias de escape y puede ejecutar funciones o programas. Como bash tiene que posicionar el cursor después del prompt de displays, necesita saber cómo calcular la longitud efectiva de la cadena de caracteres del prompt. Para indicar secuencias de caracteres no imprimibles dentro de la variable PS1 se utilizan llaves de escape: `\[` una secuencia no imprimible de caracteres `\]`. Todo esto es válido para todas las variables PS*.

(El signo de intercalación negro indica el cursor)

```
# todo lo que no sea una secuencia de escape se imprimirá literalmente
export PS1="literal sequence " # El prompt es ahora:
literal sequence █

# \u == user \h == host \w == directorio de trabajo actual
# tenga en cuenta las comillas simples para evitar la interpretación del shell
export PS1='\u@\h:\w > ' # \u == user, \h == host, \w dirección actual de trabajo
looser@host:/some/path > █

# ejecutar algunos comandos en PS1
# siguiente línea se establecerá el color de primer plano a red, si user==root,
# si no, restablece los atributos por defecto
# $( (($EUID == 0)) && tput setaf 1)
# más tarde restableceremos los atributos por defecto con
# $( tput sgr0 )
# suponiendo que sea root:
PS1="\[$( (($EUID == 0)) && tput setaf 1 \)\u\[$(tput sgr0)\]@\w:\w \ $"
looser@host:/some/path > █ # si no es root entonces <red>root<default>@host....
```

Capítulo 55: El comando cut

Parámetro	Detalles
-f, --fields	Selección sobre el terreno
-d, --delimiter	Delimitador para la selección por campos
-c, --characters	Selección basada en caracteres, delimitador ignorado o error
-s, --only-delimited	Suprimir las líneas sin caracteres delimitadores (se imprimen tal cual)
--complement	Selección invertida (extraer todos los campos/caracteres <i>excepto</i> los especificados)
--output-delimiter	Especifique cuándo debe ser diferente del delimitador de entrada

El comando **cut** es una forma rápida de extraer partes de líneas de archivos de texto. Pertenece a los comandos más antiguos de Unix. Sus implementaciones más populares son la versión GNU encontrada en Linux y la versión FreeBSD encontrada en MacOS, pero cada sabor de Unix tiene la suya propia. Vea más abajo las diferencias. Las líneas de entrada se leen de **stdin** o de archivos listados como argumentos en la línea de comandos.

Sección 55.1: Sólo un carácter delimitador

No puede tener más de un delimitador: si especifica algo como **-d " , ; : "**, algunas implementaciones utilizarán sólo el primer carácter como delimitador (en este caso, la coma.) Otras implementaciones (por ejemplo, GNU **cut**) le darán un mensaje de error.

```
$ cut -d " , ; : " -f2 <<<"J.Smith,1 Main Road,cell:1234567890;land:4081234567"
cut: the delimiter must be a single character
Try `cut --help' for more information.
```

Sección 55.2: Los delimitadores repetidos se interpretan como campos vacíos

```
$ cut -d, -f1,3 <<<"a,,b,c,d,e"
a,b
```

es bastante obvio, pero con cadenas de caracteres delimitadas por espacios puede ser menos obvio para algunos

```
$ cut -d ' ' -f1,3 <<<"a b c d e"
a b
```

cut no puede utilizarse para analizar argumentos como hacen el shell y otros programas.

Sección 55.3: Sin citar

No hay forma de proteger el delimitador. Las hojas de cálculo y los programas similares de tratamiento de CSV suelen reconocer un carácter de entrecomillado de texto que permite definir cadenas de caracteres que contienen un delimitador. Con **cut** no se puede.

```
$ cut -d, -f3 <<<'John,Smith,"1, Main Street"'
"1
```

Sección 55.4: Extraer, no manipular

Sólo puede extraer partes de líneas, no reordenar ni repetir campos.

```
$ cut -d, -f2,1 <<< 'John,Smith,USA' ## Igual que -f1,2
John,Smith
$ cut -d, -f2,2 <<< 'John,Smith,USA' ## Igual que -f2
Smith
```

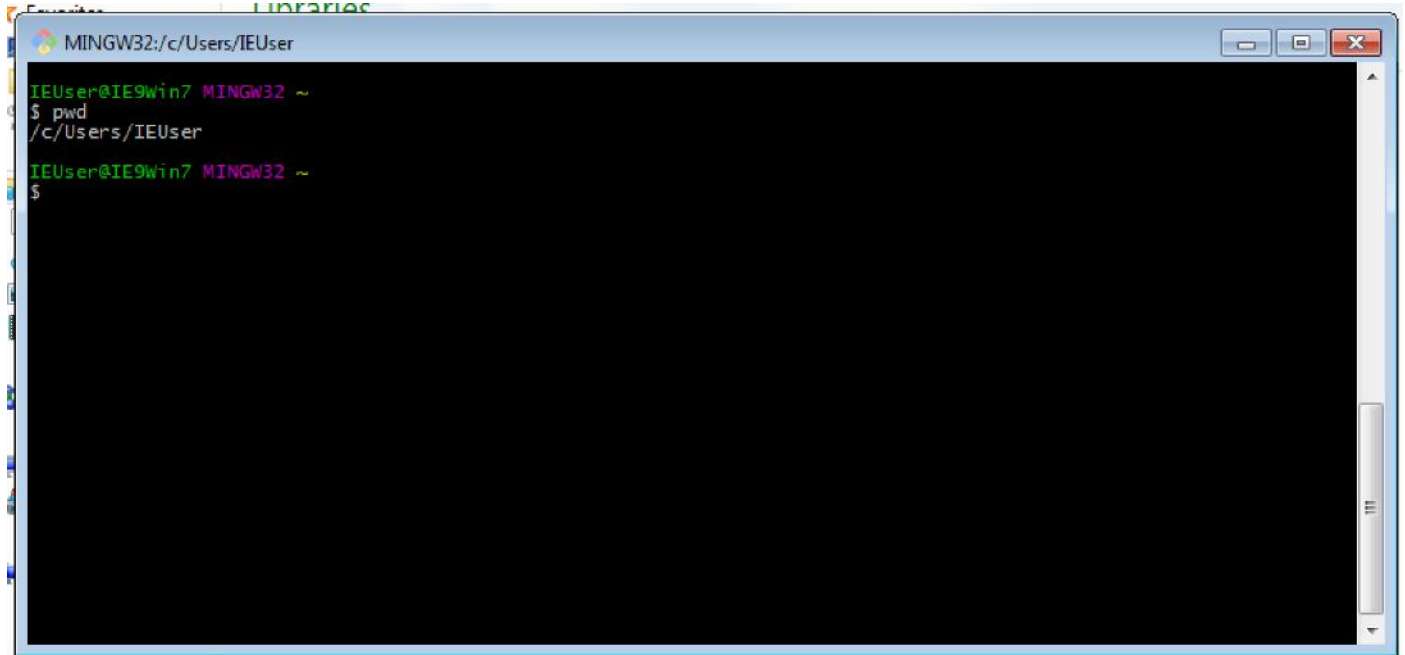
Capítulo 56: Bash en Windows 10

Sección 56.1: Léeme

La forma más sencilla de usar Bash en Windows es instalar Git para Windows. Se entrega con Git Bash que es un Bash real. Puedes acceder a él con el acceso directo en:

Start > All Programs > Git > Git Bash

Comandos como **grep**, **ls**, **find**, **sed**, **vi**, etc. funcionan.

A screenshot of a Git Bash terminal window. The title bar reads 'MINGW32:/c/Users/IEUser'. The terminal content shows the prompt 'IEUser@IE9win7 MINGW32 ~' followed by the command '\$ pwd' and the output '/c/Users/IEUser'. The prompt is followed by another '\$' indicating it is ready for the next command. The window has standard Windows window controls (minimize, maximize, close) in the top right corner.

Capítulo 57: Comando cut

Opción

`-b LIST, --bytes=LIST`
`-c LIST, --characters=LIST`

`-f LIST, --fields=LIST`
`-d DELIMITER`

Descripción

Imprime los bytes listados en el parámetro `LIST`
Imprimir caracteres en las posiciones especificadas en el parámetro `LIST`
Imprimir campos o columnas
Se utiliza para separar columnas o campos

En Bash, el comando `cut` es útil para dividir un archivo en varias partes más pequeñas.

Sección 57.1: Mostrar la primera columna de un fichero

Suponga que tiene un archivo con el siguiente aspecto

```
John Smith 31
Robert Jones 27
...
```

Este fichero tiene 3 columnas separadas por espacios. Para seleccionar sólo la primera columna, haga lo siguiente.

```
cut -d ' ' -f1 filename
```

Aquí la bandera `-d`, especifica el delimitador, o lo que separa los registros. El indicador `-f` especifica el número de campo o columna. Esto mostrará la siguiente salida.

```
John
Robert
...
```

Sección 57.2: Mostrar las columnas x a y de un fichero

A veces, es útil mostrar un rango de columnas en un fichero. Supongamos que tiene este fichero

```
Apple California 2017 1.00 47
Mango Oregon 2015 2.30 33
```

Para seleccionar las 3 primeras columnas haga

```
cut -d ' ' -f1-3 filename
```

Esto mostrará el siguiente resultado

```
Apple California 2017
Mango Oregon 2015
```

Capítulo 58: Variables globales y locales

Por defecto, cada variable en bash es **global** a cada función, script e incluso al shell externo si estás declarando tus variables dentro de un script.

Si quieres que tu variable sea local a una función, puedes usar **local** para que esa variable sea una nueva variable independiente del ámbito global y cuyo valor sólo será accesible dentro de esa función.

Sección 58.1: Variables globales

```
var="hello"
```

```
function foo(){  
    echo $var  
}
```

```
foo
```

Obviamente mostrará "hello", pero también funciona a la inversa:

```
function foo() {  
    var="hello"  
}
```

```
foo  
echo $var
```

También mostrará "hello"

Sección 58.2: Variables locales

```
function foo() {  
    local var  
    var="hello"  
}
```

```
foo  
echo $var
```

No mostrará nada, ya que **var** es una variable local a la función **foo**, y su valor no es visible desde fuera de ella.

Sección 58.3: Mezcla de ambos

```
var="hello"
```

```
function foo(){  
    local var="sup?"  
    echo "inside function, var=$var"  
}
```

```
foo  
echo "outside function, var=$var"
```

Salida

```
inside function, var=sup?  
outside function, var=hello
```

Capítulo 59: Scripts CGI

Sección 59.1: Método de solicitud: GET

Es bastante fácil llamar a un CGI-Script vía GET.
Primero necesitarás la `url` codificada del script.

A continuación, añade un `?` seguido de las variables.

- Cada variable debe tener dos secciones separadas por `=`. La primera sección debe ser siempre un nombre único para cada variable, mientras que la segunda parte sólo contiene valores
- Las variables están separadas por `&`
- La longitud total de la cadena de caracteres no debe superar los **255** caracteres
- Los nombres y valores necesitan ser codificados en html (reemplazar: `</ , / ? : @ & = + $`)

Sugerencia:

Cuando se usan **formularios html** el método de petición puede ser generado por sí mismo. Con **Ajax** se puede codificar todo a través de `encodeURIComponent` y `encodeURIComponent`.

Ejemplo:

```
http://www.example.com/cgi-bin/script.sh?var1=Hello%20World!&var2=This%20is%20a%20Test.&
```

El servidor debe comunicarse únicamente a través de **Cross-Origin Resource Sharing** (CORS), para que las peticiones sean más seguras. En este caso, utilizamos **CORS** para determinar el tipo de datos que queremos utilizar.

Hay muchos tipos de datos entre los que podemos elegir, los más comunes son...

- **text/html**
- **text/plain**
- **application/json**

Al enviar una petición, el servidor también creará muchas variables de entorno. Por ahora las variables de entorno más importantes son `$REQUEST_METHOD` y `$QUERY_STRING`.

El **Request Method** tiene que ser GET ¡nada más!

El **Query String** incluye todos los `datos` `html-encoded`.

El Script

```
#!/bin/bash
```

```
# CORS es la forma de comunicarse, así que vamos a responder al servidor en primer lugar
echo "Content-type: text/html" # establecer el tipo de datos que queremos utilizar
echo "" # no necesitamos más reglas, la línea vacía inicia esto.
```

```
# CORS están grabados en piedra y cualquier comunicación a partir de ahora será como leer un documento html.
```

```
# Por lo tanto, necesitamos crear cualquier stdout en formato html!
```

```
# crear una estructura html y enviarla a stdout
```

```
echo "<!DOCTYPE html>"
```

```
echo "<html><head>"
```

```
# El contenido se creará en función del método de solicitud
```

```
if [ "$REQUEST_METHOD" = "GET" ]; then
```

```
# Tenga en cuenta que las variables de entorno $REQUEST_METHOD y $QUERY_STRING pueden ser procesadas directamente por el shell.
```

```
# Hay que filtrar la entrada para evitar el cross site scripting.
```

```
Var1=$(echo "$QUERY_STRING" | sed -n 's/^.*var1=\([^&]*\).*$/\1/p') # leer valor de "var1"
```

```
Var1_Dec=$(echo -e $(echo "$Var1" | sed 's/+ / /g;s/%(..)/\\x\\1/g;')) # descodificación html
```

```
Var2=$(echo "$QUERY_STRING" | sed -n 's/^.*var2=\([^&]*\).*$/\1/p')
```

```
Var2_Dec=$(echo -e $(echo "$Var2" | sed 's/+ / /g;s/%(..)/\\x\\1/g;'))
```

```
# crear contenido para stdout
```

```
echo "<title>Bash-CGI Example 1</title>"
```

```
echo "</head><body>"
```

```
echo "<h1>Bash-CGI Example 1</h1>"
```

```
echo "<p>QUERY_STRING: ${QUERY_STRING}<br>var1=${Var1_Dec}<br>var2=${Var2_Dec}</p>" # imprimir los valores en stdout
```

```
else
```

```
echo "<title>456 Wrong Request Method</title>"
```

```
echo "</head><body>"
```

```
echo "<h1>456</h1>"
```

```
echo "<p>Requesting data went wrong.<br>The Request method has to be \"GET\" only!</p>"
```

```
fi
```

```
echo "<hr>"
```

```
echo "$SERVER_SIGNATURE" # otra variable de entorno
```

```
echo "</body></html>" # cerrar html
```

```
exit 0
```

El documento html tendrá este aspecto...

```
<html>
  <head>
    <title>Bash-CGI Example 1</title>
  </head>
  <body>
    <h1>Bash-CGI Example 1</h1>
    <p>QUERY_STRING: var1=Hello%20World!&amp;var2=This%20is%20a%20Test.&amp;<br>var1=Hello
    World!<br>var2=This is a Test.</p>
    <hr>
    <address>Apache/2.4.10 (Debian) Server at example.com Port 80</address>
  </body>
</html>
```

La **salida** de las variables tendrá este aspecto...

```
var1=Hello%20World!&var2=This%20is%20a%20Test.&  
Hello World!  
This is a Test.  
Apache/2.4.10 (Debian) Server at example.com Port 80
```

Efectos secundarios negativos...

- Toda la codificación y decodificación no se ve bien, pero es necesario
- La petición será de lectura pública y dejará una bandeja detrás
- El tamaño de una petición es limitado
- Necesita protección contra Cross-Side-Scripting (XSS)

Sección 59.2: Método de solicitud: POST /w JSON

El uso del método de solicitud **POST** en combinación con **SSL** hace que la transferencia de datos sea más segura.

Además...

- La mayor parte de la codificación y decodificación ya no es necesaria.
- La URL será visible para cualquiera y necesita ser codificada. Los datos se enviarán por separado, por lo que deben protegerse mediante SSL.
- El tamaño de los datos es casi ilimitado.
- Todavía necesita protección contra Cross-Side-Scripting (XSS)

Para mantener este escaparate simple queremos recibir **datos JSON** y la comunicación debe ser sobre **Cross-Origin Resource Sharing** (CORS).

El siguiente script también demostrará dos **Content-Types** diferentes.

```
#!/bin/bash
```

```
exec 2>/dev/null # No queremos que se imprima ningún mensaje de error en stdout
trap "response_with_html && exit 0" ERR # respuesta con un mensaje html cuando se produce un error
y cierra el script
```

```
function response_with_html(){
    echo "Content-type: text/html"
    echo ""
    echo "<!DOCTYPE html>"
    echo "<html><head>"
    echo "<title>456</title>"
    echo "</head><body>"
    echo "<h1>456</h1>"
    echo "<p>Attempt to communicate with the server went wrong.</p>"
    echo "<hr>"
    echo "$SERVER_SIGNATURE"
    echo "</body></html>"
}
```

```
function response_with_json(){
    echo "Content-type: application/json"
    echo ""
    echo "{\"message\": \"Hello World!\"}"
}
```

```
if [ "$REQUEST_METHOD" = "POST" ]; then
```

```
    # La variable de entorno $CONTENT_TYPE describe el tipo de datos recibidos
```

```
    case "$CONTENT_TYPE" in
```

```
        application/json)
```

```
            # La variable de entorno $CONTENT_LENGTH describe el tamaño de los datos
```

```
            read -n "$CONTENT_LENGTH" QUERY_STRING_POST # leer flujo de datos
```

```
            # Las siguientes líneas evitarán XSS y comprobarán que los datos JSON sean válidos.
```

```
            # Pero estos Símbolos necesitan ser codificados de alguna manera antes de enviarlos a este script
```

```
            QUERY_STRING_POST=$(echo "$QUERY_STRING_POST" | sed "s/'//g" | sed
's/\$///g;s/`//g;s/\*//g;s/\\//g' ) # elimina algunos símbolos (como \ * ` $ ') para
evitar XSS con Bash y SQL.
```

```
            QUERY_STRING_POST=$(echo "$QUERY_STRING_POST" | sed -e :a -e 's/<[^>]*>//g;/</N;//ba')
```

```
            # elimina la mayoría de las declaraciones html para evitar XSS en los documentos
```

```
            JSON=$(echo "$QUERY_STRING_POST" | jq .) # json encode - Esta es una forma bastante
segura de comprobar si el código json es válido.
```

```
            ;;
```

```
        *)
```

```
            response_with_html
```

```
            exit 0
```

```
            ;;
```

```
    esac
```

```
else
```

```
    response_with_html
```

```
    exit 0
```

```
fi
```

```
# Algunos comandos...
```

```
response_with_json
```

```
exit 0
```

Obtendrá `{"message": "Hello World!"}` como respuesta al enviar **datos JSON** vía **POST** a este script. Todo lo demás recibirá el documento html.

También es importante la variable `$JSON`. Esta variable está libre de XSS, pero todavía podría tener valores erróneos en ella y necesita ser verificada primero. Por favor, téngalo en cuenta.

Este código funciona de forma similar sin JSON.

Podrías obtener cualquier dato de esta manera.

Sólo tienes que cambiar el **Content-Type** para sus necesidades.

Ejemplo:

```
if [ "$REQUEST_METHOD" = "POST" ]; then
    case "$CONTENT_TYPE" in
        application/x-www-form-urlencoded)
            read -n "$CONTENT_LENGTH" QUERY_STRING_POST
            text/plain)
                read -n "$CONTENT_LENGTH" QUERY_STRING_POST
            ;;
        esac
    fi
```

Por último, pero no por ello menos importante, no olvide responder a todas las solicitudes; de lo contrario, los programas de terceros no sabrán si han tenido éxito.

Capítulo 60: Palabra clave Select

La palabra clave Select se puede utilizar para obtener el argumento de entrada en un formato de menú.

Sección 60.1: La palabra clave select se puede utilizar para obtener el argumento de entrada en un formato de menú

Supongamos que queremos que el user **SELECT** palabras clave de un menú, podemos crear un script similar a

```
#!/usr/bin/env bash
```

```
select os in "linux" "windows" "mac"
do
    echo "${os}"
    break
done
```

Explicación: Aquí la palabra clave **SELECT** se utiliza para recorrer una lista de elementos que se presentarán en el símbolo del sistema para que el usuario elija. Fíjese en la palabra clave **break** para salir del bucle una vez que el usuario hace una elección. De lo contrario, ¡el bucle será interminable!

Resultados: Al ejecutar este script, se mostrará un menú con estos elementos y se pedirá al usuario que los seleccione. Tras la selección, se mostrará el valor y se volverá al símbolo del sistema.

```
>bash select_menu.sh
1) linux
2) windows
3) mac
#? 3
mac
>
```


Capítulo 61: Cuando utilizar eval

Lo primero y más importante: ¡saber lo que se hace! En segundo lugar, aunque deberías evitar el uso de `eval`, si su uso hace que el código sea más limpio, adelante.

Sección 61.1: Uso de eval

Por ejemplo, considere lo siguiente que establece el contenido de `$@` al contenido de una variable dada:

```
a=(1 2 3)
eval set -- "${a[@]}"
```

Este código suele ir acompañado de `getopt` o `getopts` para establecer `$@` a la salida de los analizadores de opciones antes mencionados, sin embargo, también se puede utilizar para crear una función pop simple que pueda operar sobre variables de forma silenciosa y directa sin tener que almacenar el resultado en la variable original:

```
isnum()
{
    # ¿el argumento es un número entero?
    local re='^[0-9]+$'
    if [[ -n $1 ]]; then
        [[ $1 =~ $re ]] && return 0
        return 1
    else
        return 2
    fi
}

isvar()
{
    if isnum "$1"; then
        return 1
    fi
    local arr="$(eval eval -- echo -n "${$1}")"
    if [[ -n ${arr[@]} ]]; then
        return 0
    fi
    return 1
}

pop()
{
    if [[ -z $@ ]]; then
        return 1
    fi

    local var=
    local isvar=0
    local arr=()

    if isvar "$1"; then # comprobemos si se trata de una variable o simplemente de un array vacío
        var="$1"
        isvar=1
        arr=($(eval eval -- echo -n "${$1[@]}")) # si es un var, obtener su contenido
    else
        arr=($@)
    fi

    # necesitamos invertir el contenido de $@ para poder desplazar
    # el último elemento en la nada
}
```

```

arr=($(awk <<<"${arr[@]}" '{ for (i=NF; i>1; --i) printf("%s ",$i); print $1; }'

# establecer $@ a ${arr[@]} para que podamos ejecutar shift contra ella.
eval set -- "${arr[@]}"

shift # eliminar el último elemento

# devolver el array a su orden original
arr=($(awk <<<"$@" '{ for (i=NF; i>1; --i) printf("%s ",$i); print $1; }'

# hacer eco del contenido en beneficio de los usuarios y de los arrays vacíos
echo "${arr[@]}"

if ((isvar)); then
    # establece el contenido del var original en el nuevo array modificado
    eval -- "$var=(${arr[@]})"
fi
}

```

Sección 61.2: Uso de eval con getopt

Mientras que **eval** puede no ser necesario para una función como **pop**, sin embargo, es necesario siempre que utilice **getopt**:

Considere la siguiente función que acepta **-h** como opción:

```

f()
{
    local __me__="${FUNCNAME[0]}"
    local argv=$(getopt -o 'h' -n $__me__ -- "$@" )

    eval set -- "$argv"

    while ;; do
        case "$1" in
            -h)
                echo "LOLOLOLOL"
                return 0
                ;;
            --)
                shift
                break
                ;;
        Done
    echo "$@"
}

```

Sin **eval set -- "\$argv"** genera

-h --

en lugar del deseado **(-h --)** y posteriormente entra en un bucle infinito porque

-h --

no coincide con **--** o **-h**.

Capítulo 62: Trabajar en red con Bash

Bash suele utilizarse en la gestión y el mantenimiento de servidores y clusters. Debe incluirse información relativa a los comandos típicos utilizados por las operaciones de red, cuándo utilizar cada comando para cada propósito, y ejemplos de aplicaciones únicas y/o interesantes del mismo.

Sección 62.1: Comandos de red

`ifconfig`

El comando anterior mostrará todas las interfaces activas de la máquina y también dará la información de

1. Dirección IP asignada a la interfaz
2. Dirección MAC de la interfaz
3. Dirección de difusión
4. Bytes de transmisión y recepción

Algunos ejemplos

`ifconfig -a`

El comando anterior también muestra la interfaz de desactivación

`ifconfig eth0`

El comando anterior sólo mostrará la interfaz `eth0`

`ifconfig eth0 192.168.1.100 netmask 255.255.255.0`

El comando anterior asignará la IP estática a la interfaz `eth0`

`ifup eth0`

El comando anterior habilitará la interfaz `eth0`

`ifdown eth0`

El siguiente comando desactivará la interfaz `eth0`

`ping`

El comando anterior (Packet Internet Grouper) es para probar la conectividad entre los dos nodos

`ping -c2 8.8.8.8`

El comando anterior hará `ping` o probará la conectividad con el servidor de google durante 2 segundos.

`traceroute`

El comando anterior se utiliza en la solución de problemas para averiguar el número de saltos necesarios para llegar al destino.

`netstat`

El comando anterior (Estadísticas de red) proporciona información sobre las conexiones y su estado.

`dig www.google.com`

El comando anterior (agrupador de información de dominio) consulta la información relacionada con DNS

`nslookup www.google.com`

El comando anterior consulta el DNS y encuentra la dirección IP correspondiente al nombre del sitio web.

`route`

El comando anterior se utiliza para comprobar la información de ruta de la red. Básicamente muestra la tabla de enrutamiento

```
router add default gw 192.168.1.1 eth0
```

El comando anterior añadirá la ruta por defecto de la red de la interfaz `eth0` a `192.168.1.1` en la tabla de enrutamiento.

```
route del default
```

El comando anterior eliminará la ruta por defecto de la tabla de enrutamiento

Capítulo 63: Paralelo

Opción	Descripción
<code>-j n</code>	Ejecutar n trabajos en paralelo
<code>-k</code>	Mantener el mismo orden
<code>-X</code>	Múltiples argumentos con sustitución de contexto
<code>--colsep regexp</code>	Dividir entrada en <code>regexp</code> para sustituciones posicionales
<code>{ } { . } { / } { / . } { # }</code>	Cadenas de caracteres de remplazo
<code>{ 3 } { 3 . } { 3 / } { 3 / . }</code>	Cadenas de caracteres de sustitución posicional
<code>-S sshlogin</code>	Por ejemplo: <code>foo@server.example.com</code>
<code>--trc {} .bar</code>	Abreviatura de <code>--transfer --return {} .bar --cleanup</code>
<code>--onall</code>	Ejecuta el comando dado con el argumento en todos los sshlogins
<code>--nonall</code>	Ejecuta el comando dado sin argumentos en todos los sshlogins
<code>--pipe</code>	Dividir <code>stdin</code> (entrada estándar) en varios trabajos.
<code>--recend str</code>	Separador final de registro para <code>--pipe</code> .
<code>--restart str</code>	Separador de inicio de grabación para <code>--pipe</code> .

Los trabajos en GNU Linux se pueden paralelizar usando GNU `parallel`. Un trabajo puede ser un único comando o un pequeño script que tiene que ejecutarse para cada una de las líneas de la entrada. La entrada típica es una lista de archivos, una lista de hosts, una lista de usuarios, una lista de URLs, o una lista de tablas. Un trabajo también puede ser un comando que lee de una tubería.

Sección 63.1: Paralelizar tareas repetitivas en listas de archivos

Muchos trabajos repetitivos pueden realizarse de forma más eficiente si se utilizan más recursos del ordenador (es decir, CPU y RAM). A continuación, se muestra un ejemplo de ejecución de varios trabajos en paralelo.

Suponga que tiene una `< list of files >`, digamos la salida de `ls`. Además, que estos archivos están comprimidos `bz2` y el siguiente orden de tareas necesitan ser operados en ellos.

1. Descomprimir los archivos `bz2` usando `bzcat` a `stdout`
2. `grep` (por ejemplo, filtro) líneas con palabra(s) clave específica(s) utilizando `grep <some key word>`.
3. Canalizar la salida para concatenarla en un único archivo comprimido con `gzip`.

Ejecutar esto utilizando un bucle `while` puede tener este aspecto

```
filenames="file_list.txt"
while read -r line
do
  name="$line"
  ## coge las lineas con los puppies en el
  bzcat $line | grep puppies | gzip >> output.gz
done < "$filenames"
```

Usando GNU `Parallel`, podemos ejecutar 3 trabajos paralelos a la vez simplemente haciendo

```
parallel -j 3 "bzcat {} | grep puppies" ::: $( cat filelist.txt ) | gzip > output.gz
```

Este comando es simple, conciso y más eficiente cuando el número de archivos y su tamaño es grande. Los trabajos se inician en `parallel`, la opción `-j 3` lanza 3 trabajos en paralelo y la entrada a los trabajos en paralelo se toma con `::: .` La salida se canaliza finalmente a `gzip > output.gz`

Sección 63.2: Paralelizar STDIN

Ahora, imaginemos que tenemos un archivo grande (por ejemplo, 30 GB) que hay que convertir, línea por línea. Digamos que tenemos un script, `convert.sh`, que hace esta `<task>`. Podemos enviar el contenido de este archivo a `stdin` para que `parallel` lo tome y trabaje con él en trozos como.

```
<stdin> | parallel --pipe --block <block size> -k <task> > output.txt
```

donde **<stdin>** puede provenir de cualquier cosa como **cat <file>**.

Como ejemplo reproducible, nuestra tarea será **nl -n rz**. Tome cualquier archivo, el mío será **data.bz2**, y páselo a **<stdin>**.

```
bzcat data.bz2 | nl | parallel --pipe --block 10M -k nl -n rz | gzip > ouptput.gz
```

El ejemplo anterior toma **<stdin>** de **bzcat data.bz2 | nl**, donde incluí **nl** sólo como prueba de concepto de que la salida final **ouptput.gz** se guardará en el orden en que se recibió. Entonces, **parallel** divide el **<stdin>** en trozos de tamaño 10 MB, y para cada trozo lo pasa a través de **nl -n rz** donde sólo añade un número justificado (ver **nl --help** para más detalles). Las opciones **--pipe** le dice a **parallel** que divida **<stdin>** en múltiples trabajos y **--block** especifica el tamaño de los bloques. La opción **-k** especifica que debe mantenerse el orden.

El resultado final debería ser algo parecido a

```
000001 1 <data>
000002 2 <data>
000003 3 <data>
000004 4 <data>
000005 5 <data>
...
000587 552409 <data>
000588 552410 <data>
000589 552411 <data>
000590 552412 <data>
000591 552413 <data>
```

Mi archivo original tenía 552.413 líneas. La primera columna representa los trabajos paralelos, y la segunda columna representa la numeración de líneas original que se pasó al paralelo en trozos. Debería notar que el orden en la segunda columna (y el resto del archivo) se mantiene.

Capítulo 64: Decodificar URL

Sección 64.1: Ejemplo sencillo

URL codificada

```
http%3A%2F%2Fwww.foo.com%2Findex.php%3Fid%3Dqwerty
```

Utilice este comando para decodificar la URL

```
echo "http%3A%2F%2Fwww.foo.com%2Findex.php%3Fid%3Dqwerty" | sed -e "s/%\([0-9A-F][0-9AF]\)/\\\\x\1/g" | xargs -0 echo -e
```

URL decodificada (resultado del comando)

```
http://www.foo.com/index.php?id=qwerty
```

Sección 64.2: Uso de printf para decodificar una cadena de caracteres

```
#!/bin/bash
```

```
$ string='Question%20-%20%22how%20do%20I%20decode%20a%20percent%20encoded%20string%3F%22%0AAnswer%20%20%20-%20Use%20printf%20%3A)'\n$ printf '%b\n' "${string//%/\\x}"
```

```
# el resultado
```

```
Question - "how do I decode a percent encoded string?"\nAnswer - Use printf :)
```

Capítulo 65: Patrones de diseño

Realizar algunos patrones de diseño comunes en Bash

Sección 65.1: Patrón de publicación/suscripción (Pub/Sub)

Cuando un proyecto Bash se convierte en una biblioteca, puede resultar difícil añadir nuevas funcionalidades. Los nombres de las funciones, variables y parámetros normalmente necesitan ser cambiados en los scripts que los utilizan. En escenarios como este, es útil desacoplar el código y utilizar un patrón de diseño dirigido por eventos. En dicho patrón, un script externo puede suscribirse a un evento. Cuando ese evento es disparado (publicado) el script puede ejecutar el código que registró con el evento.

pubsub.sh:

```
#!/usr/bin/env bash

#
# Guarda la ruta al directorio de este script en una variable env global
#
DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"

#
# Array que contendrá todos los eventos registrados
#
EVENTS=()

function action1() {
    echo "Action #1 was performed ${2}"
}

function action2() {
    echo "Action #2 was performed"
}

#
# @desc :: Registra un evento
# @param :: string $1 - El nombre del evento. Básicamente un alias para el nombre de una función
# @param :: string $2 - El nombre de la función a llamar
# @param :: string $3 - Ruta completa al script que incluye la función llamada
#
function subscribe() {
    EVENTS+=("${1};${2};${3}")
}

#
# @desc :: Publicar un evento
# @param :: string $1 - El nombre del acontecimiento que se publica
#
function publish() {
    for event in ${EVENTS[@]}; do
        local IFS=";"
        read -r -a event <<< "$event"
        if [[ "${event[0]}" == "${1}" ]]; then
            ${event[1]} "${2}"
        fi
    done
}
```



```
#  
# Registrar nuestros eventos y las funciones que los gestionan  
#  
subscribe "/do/work" "action1" "${DIR}"  
subscribe "/do/more/work" "action2" "${DIR}"  
subscribe "/do/even/more/work" "action1" "${DIR}"  
  
#  
# Ejecutar nuestros eventos  
#  
publish "/do/work"  
publish "/do/more/work"  
publish "/do/even/more/work" "again"
```

Ejecutar:

```
chmod +x pubsub.sh  
./pubsub.sh
```

Capítulo 66: Peligros

Sección 66.1: Espacios en blanco al asignar variables

Los espacios en blanco son importantes a la hora de asignar variables.

```
foo = 'bar' # incorrecto
foo= 'bar' # incorrecto
foo='bar' # correcto
```

Los dos primeros darán lugar a errores de sintaxis (o peor aún, a la ejecución de un comando incorrecto). El último ejemplo establecerá correctamente la variable `$foo` al texto `'bar'`.

Sección 66.2: Los comandos fallidos no detienen la ejecución del script

En la mayoría de los lenguajes de scripting, si una llamada a una función falla, puede lanzar una excepción y detener la ejecución del programa. Los comandos Bash no tienen excepciones, pero sí códigos de salida. Un código de salida distinto de cero señala un fallo, sin embargo, un código de salida distinto de cero no detendrá la ejecución del programa.

Esto puede llevar a situaciones peligrosas (aunque hay que reconocer que artificiosas) como ésta:

```
#!/bin/bash
cd ~/non/existent/directory
rm -rf *
```

Si al ir a este directorio falla, Bash ignorará el fallo y pasará al siguiente comando, limpiando el directorio desde donde ejecutó el script.

La mejor forma de solucionar este problema es utilizar el comando `set`:

```
#!/bin/bash
set -e
cd ~/non/existent/directory
rm -rf *
```

`set -e` le dice a Bash que salga del script inmediatamente si cualquier comando devuelve un estado distinto de cero.

Sección 66.3: Falta la última línea de un archivo

El estándar C dice que los ficheros deben terminar con una nueva línea, por lo que, si EOF aparece al final de una línea, es posible que algunos comandos no pasen por alto esa línea. Por ejemplo:

```
$ echo 'one\ntwo\nthree\c' > file.txt

$ cat file.txt
one
two
three

$ while read line ; do echo "line $line" ; done < file.txt
one
two
```

Para asegurarte de que esto funciona correctamente para en el ejemplo anterior, añade una prueba para que continúe el bucle si la última línea no está vacía.

```
$ while read line || [ -n "$line" ] ; do echo "line $line" ; done < file.txt
one
two
three
```

Apéndice A: Atajos de teclado

Sección A.1: Atajos de edición

Atajo

Ctrl	+	a	.
Ctrl	+	e	.
Ctrl	+	k	.
Ctrl	+	u	.
Ctrl	+	w	.
Alt	+	b	.
Alt	+	f	.
Ctrl	+	Alt	+ e
Ctrl	+	y	.
Alt	+	y	.

Descripción

desplazarse al principio de la línea
pasar al final de la línea
Mata el texto desde la posición actual del cursor hasta el final de la línea
Mata el texto desde la posición actual del cursor hasta el principio de la línea
Matar la palabra detrás de la posición actual del cursor
retroceder una palabra
avanzar una palabra
línea de expansión de shell
Devuelve el último texto eliminado a la memoria intermedia del cursor.
Rota el texto matado. Sólo puede hacer esto si el comando anterior es
Ctrl + y o Alt + y.

Matar texto borrará el texto, pero lo guardará para que el usuario pueda volver a insertarlo tirando de él. Similar a cortar y pegar, excepto que el texto se coloca en un anillo de matar que permite almacenar más de un conjunto de texto para ser arrancado de nuevo en la línea de comandos.

Puedes encontrar más información en el [manual de emacs](#).

Sección A.2: Atajos de llamada

Atajo

Ctrl	+	r	.
Ctrl	+	p	.
Ctrl	+	n	.
Ctrl	+	g	.
Alt	+	.	.
Alt	+	n	Alt + .
!!	+	Return	.

Descripción

buscar en el historial hacia atrás
comando anterior en el historial
siguiente comando en el historial
salir del modo de búsqueda en el historial
utilizar la última palabra del comando anterior
repetir para obtener la última palabra del comando anterior + 1
utilizar la enésima palabra del comando anterior
volver a ejecutar el último comando (útil cuando se ha olvidado **sudo**:
sudo !!)

Sección A.3: Macros

Atajo

Ctrl	+	x	,	(
Ctrl	+	x	,)
Ctrl	+	x	,	e

Descripción

iniciar la grabación de una macro
detener la grabación de una macro
ejecutar la última macro grabada

Sección A.4: Fijaciones de teclas personalizadas

Con el comando **bind** es posible definir combinaciones de teclas personalizadas.

El siguiente ejemplo vincula un **Alt + w** a **>/dev/null 2>&1**:

```
bind '"\ew": "\> /dev/null 2>&1\>"
```

Si desea ejecutar la línea inmediatamente, añádale `\C-m` (`Intro`):

```
bind '"\ew"' : "\" >/dev/null 2>&1\C-m\
```

Sección A.5: Control del trabajo

Atajo

Ctrl	+	c
Ctrl	+	z

Descripción

Detener el trabajo actual

Suspender el trabajo actual (enviar una señal SIGTSTP)

Créditos

Muchas gracias a todas las personas de Stack Overflow Documentation que ayudaron a proporcionar este contenido, más cambios pueden ser enviados a web@petercv.com para que el nuevo contenido sea publicado o actualizado.

Traductor al español

[rortegag](#)

Ajay Sangale	Capítulo 1
Ajinkya	Capítulo 20
Alessandro Mascolo	Capítulos 11 y 26
Alexej Magura	Capítulos 9, 12, 36 y 61
Amir Rachum	Capítulo 8
Anil	Capítulo 1
anishsane	Capítulo 5
Antoine Bolvy	Capítulo 9
Archemar	Capítulo 9
Arronical	Capítulo 12
Ashari	Capítulo 36
Ashkan	Capítulos 36 y 43
Batsu	Capítulo 17
Benjamin W.	Capítulos 1, 9, 12, 15, 24, 31, 36, 46 y 47
binki	Capítulo 21
Blachshma	Capítulo 1
Bob Bagwill	Capítulo 1
Bostjan	Capítulo 7
BrunoLM	Capítulo 14
Brydon Gibson	Capítulo 9
Bubblepop	Capítulos 1, 5, 8, 12 y 24
Burkhard	Capítulo 1
BurnsBA	Capítulo 22
Carpetsmoker	Capítulo 47
cb0	Capítulo 28
Chandrahas Aroori	Capítulo 6
chaos	Capítulo 9
charneykaye	Capítulo 50
chepner	Capítulos 15, 27 y 46
Chris Rasys	Capítulo 34
Christopher Bottoms	Capítulos 1, 3 and 5
codeforester	Capítulo 12
Cody	Capítulo 66
Colin Yang	Capítulo 1
Cows quack	Capítulo 30
CraftedCart	Capítulo 1
CrazyMax	Capítulo 64
criw	Capítulo 36
Daniel Käfer	Capítulo 67
Danny	Capítulo 1
Dario	Capítulos 28, 36 y 55
David Grayson	Capítulo 9
Deepak K M	Capítulo 20
deepmax	Capítulo 25
depperm	Capítulos 4 y 35

dhimanta	Capítulo 62
dimo414	Capítulo 14
dingalapadum	Capítulos 7 y 16
divyum	Capítulos 1 y 14
DocSalvager	Capítulo 10
Doctor J	Capítulo 28
DonyorM	Capítulo 10
Dr Beco	Capítulo 36
Dunatotatos	Capítulo 51
Echoes_86	Capítulo 17
Edgar Rokjān	Capítulo 10
edi9999	Capítulo 14
Eric Renouf	Capítulo 9
fedorqui	Capítulos 12, 15, 17, 20, 28 y 34
fifaltra	Capítulos 8 y 53
Flows	Capítulo 18
Gavyn	Capítulos 9, 26, 33 y 36
George Vasiliou	Capítulos 9, 15 y 58
Gilles	Capítulos 21 y 22
glenn jackman	Capítulos 1, 4, 5 y 7
Grexis	Capítulo 15
Grisha Levit	Capítulo 36
gzh	Capítulo 10
hedgar2017	Capítulos 9, 15 y 22
Holt Johnson	Capítulo 4
IO_ol	Capítulo 64
Iain	Capítulos 4 y 20
lamaTacos	Capítulo 35
Inanc Gumus	Capítulo 1
Inian	Capítulos 17 y 28
intboolstring	Capítulos 4, 5 y 7
Jahid	Capítulos 1, 5, 9, 10, 12, 14, 15, 17, 20, 21, 22, 23, 30, 34, 39, 43, 44 y 45
James Taylor	Capítulo 23
Jamie Metzger	Capítulo 31
jandob	Capítulo 29
janos	Capítulos 7, 10, 12, 14, 20 y 24
Jeffrey Lin	Capítulo 49
JepZ	Capítulo 3
jerblack	Capítulo 12
Jesse Chen	Capítulos 15, 26 y 45
JHS	Capítulos 7, 19 y 67
jimsug	Capítulo 24
John Kugelman	Capítulo 12
Jon	Capítulo 63
Jon Ericson	Capítulo 9
Jonny Henly	Capítulo 4
jordi	Capítulo 48
Judd Rogers	Capítulos 9 y 67
Kelum Senanayake	Capítulo 23
ksoni	Capítulo 30
leftaroundabout	Capítulo 17
Leo Ufimtsev	Capítulo 33
liborm	Capítulo 9
lynxlynxlynx	Capítulo 43
m02ph3u5	Capítulo 67

<u>markjwill</u>	Capítulo 12
<u>Markus V.</u>	Capítulo 4
<u>Mateusz Piotrowski</u>	Capítulo 12
<u>Matt Clark</u>	Capítulos 1, 9, 14, 17, 19 y 23
<u>mattmc</u>	Capítulos 36 y 65
<u>Michael Le Barbier Grünewald</u>	Capítulo 14
<u>Mike Metzger</u>	Capítulo 8
<u>miken32</u>	Capítulos 9 y 10
<u>Misa Lazovic</u>	Capítulos 4 y 30
<u>Mohima Chaudhuri</u>	Capítulos 18 y 41
<u>nautical</u>	Capítulo 34
<u>NeilWang</u>	Capítulo 12
<u>Neui</u>	Capítulo 8
<u>Ocab19</u>	Capítulo 58
<u>ormaaj</u>	Capítulo 12
<u>Osaka</u>	Capítulo 4
<u>P.P.</u>	Capítulo 38
<u>Pavel Kazhevets</u>	Capítulo 25
<u>Peter Uhnak</u>	Capítulo 31
<u>phs</u>	Capítulo 47
<u>Pooyan Khosravi</u>	Capítulo 9
<u>Rafa Moyano</u>	Capítulo 42
<u>Reboot</u>	Capítulo 42
<u>Riccardo Petraglia</u>	Capítulo 8
<u>Richard Hamilton</u>	Capítulos 4, 16, 41 y 57
<u>Riker</u>	Capítulos 1 y 40
<u>Roman Piták</u>	Capítulo 47
<u>Root</u>	Capítulos 5, 8 y 9
<u>Sameer Srivastava</u>	Capítulo 8
<u>Samik</u>	Capítulos 4, 5, 10, 12, 14 y 37
<u>Samuel</u>	Capítulo 5
<u>Saqib Rokadia</u>	Capítulo 67
<u>satyanarayan rao</u>	Capítulo 1
<u>Scroff</u>	Capítulo 66
<u>Sergey</u>	Capítulo 14
<u>sjsam</u>	Capítulos 1 y 32
<u>Sk606</u>	Capítulos 8, 12 y 33
<u>Skynet</u>	Capítulo 45
<u>SLePort</u>	Capítulos 5 y 10
<u>Stephane Chazelas</u>	Capítulos 15 y 36
<u>Stobor</u>	Capítulo 20
<u>suleiman</u>	Capítulo 59
<u>Sundeep</u>	Capítulo 1
<u>Sylvain Bugat</u>	Capítulos 2, 4, 9, 14 y 15
<u>Thomas Champion</u>	Capítulo 56
<u>Tim Rijavec</u>	Capítulo 25
<u>TomOnTime</u>	Capítulo 47
<u>Trevor Clarke</u>	Capítulo 1
<u>tripleee</u>	Capítulos 1, 5, 14, 17 y 36
<u>tversteeg</u>	Capítulo 30
<u>uhelp</u>	Capítulos 2, 7, 13, 20, 31, 36, 47, 48, 52, 53 y 54
<u>UNagaswamy</u>	Capítulos 12, 13 y 60
<u>user1336087</u>	Capítulos 1 y 26
<u>vielmetti</u>	Capítulo 5
<u>vmaroli</u>	Capítulo 39

<u>Warren Harper</u>	Capítulo 9
<u>Wenzhong</u>	Capítulo 30
<u>Will</u>	Capítulos 12, 15 y 21
<u>Will Barnwell</u>	Capítulo 24
<u>William Pursell</u>	Capítulos 1, 36 y 49
<u>Wojciech Kazior</u>	Capítulo 36
<u>Wolfgang</u>	Capítulo 9
<u>xhienne</u>	Capítulo 5
<u>ymbirtt</u>	Capítulo 15
<u>zarak</u>	Capítulos 8, 24 y 31
<u>Zaz</u>	Capítulo 1
<u>Мона_Сax</u>	Capítulo 28
<u>南山竹</u>	Capítulos 1, 5, 9, 12 y 17