



EBook Gratis

APRENDIZAJE

Bash

Free unaffiliated eBook created from
Stack Overflow contributors.

#bash

Tabla de contenido

Acerca de.....	1
Capítulo 1: Empezando con Bash.....	2
Versiones.....	2
Examples.....	2
Hola mundo usando variables.....	2
Hola Mundo.....	3
Shell interactivo.....	3
Notas.....	3
Shell no interactivo.....	4
Visualización de información para Bash incorporados.....	5
Hola mundo con entrada de usuario.....	6
Manejo de argumentos con nombre.....	7
Hola mundo en modo "depurar".....	7
Importancia de citar en cuerdas.....	8
Hay dos tipos de citas:.....	8
Capítulo 2: Abastecimiento.....	9
Examples.....	9
Obtención de un archivo.....	9
Aprovisionamiento de un entorno virtual.....	9
Capítulo 3: Alcance.....	11
Examples.....	11
Ámbito dinámico en acción.....	11
Capítulo 4: Aliasing.....	12
Introducción.....	12
Observaciones.....	12
Examples.....	12
Crear un alias.....	12
Listar todos los alias.....	12
Expandir alias.....	12
Eliminar un alias.....	13

Omitir un alias.....	13
BASH_ALIASES es un arreglo interno de bash assoc.....	13
Capítulo 5: Aquí los documentos y aquí las cadenas.....	15
Examples.....	15
Sangrando aquí documentos.....	15
Aquí cuerdas.....	15
Cadenas de límite.....	16
Crear un archivo.....	17
Ejecutar comando con el documento aquí.....	17
Ejecuta varios comandos con sudo.....	18
Capítulo 6: Aritmética de Bash.....	19
Sintaxis.....	19
Parámetros.....	19
Observaciones.....	19
Examples.....	19
Comando aritmético.....	19
Aritmética simple con (()).	20
Aritmética simple con expr.....	20
Capítulo 7: Arrays.....	21
Examples.....	21
Asignaciones de matrices.....	21
Accediendo a Array Elements.....	22
Longitud de la matriz.....	23
Modificación de Array.....	23
Iteración de Array.....	24
También puede iterar sobre la salida de un comando:.....	24
Destruir, eliminar o desarmar una matriz.....	25
Matrices asociativas.....	25
Lista de índices inicializados.....	26
Buceando a través de una matriz.....	26
Matriz de cuerda.....	27
Función de inserción de matriz.....	28

Leyendo un archivo entero en una matriz.....	28
Capítulo 8: Atajos de teclado.....	30
Observaciones.....	30
Examples.....	30
Recordar atajos.....	30
Edición de atajos.....	30
Control de trabajo.....	31
Macros.....	31
Custom Key Bindings.....	31
Capítulo 9: Bash en Windows 10.....	32
Examples.....	32
Readme.....	32
Capítulo 10: Bash sustituciones de historia.....	34
Examples.....	34
Utilizando! \$.....	34
Referencia rápida.....	34
Interacción con la historia.....	34
Designadores de eventos.....	34
Designadores de palabras.....	35
Modificadores.....	35
Buscar en el historial de comandos por patrón.....	36
Cambie al directorio recién creado con! #: N.....	36
Repita el comando anterior con una sustitución.....	36
Repita el comando anterior con sudo.....	36
Capítulo 11: Cadena de comandos y operaciones.....	38
Introducción.....	38
Examples.....	38
Contando la ocurrencia de un patrón de texto.....	38
transferir la salida del cmd de la raíz al archivo de usuario.....	38
encadenamiento lógico de comandos con && y 	38
Encadenamiento en serie de comandos con punto y coma.....	39
comandos de encadenamiento con 	39

Capítulo 12: Cambiar shell	40
Syntaxis	40
Examples	40
Encuentra el shell actual	40
Cambiar la concha	40
Listar conchas disponibles	40
Capítulo 13: Citando	41
Syntaxis	41
Examples	41
Nuevas líneas y personajes de control	41
Comillas dobles para sustitución de variables y comandos	41
Citando texto literal	42
Diferencia entre comillas dobles y comillas simples	43
Capítulo 14: Comando de corte	45
Introducción	45
Syntaxis	45
Parámetros	45
Examples	45
Mostrar la primera columna de un archivo	45
Mostrar las columnas x a y de un archivo	46
Capítulo 15: Control de trabajo	47
Syntaxis	47
Examples	47
Ejecutar comando en segundo plano	47
Listar procesos de fondo	47
Traer un proceso de fondo al primer plano	47
Detener un proceso de primer plano	48
Reiniciar proceso de fondo detenido	48
Capítulo 16: Copiando (cp)	49
Syntaxis	49
Parámetros	49
Examples	49

Copia un solo archivo.....	49
Copiar carpetas.....	49
Capítulo 17: co-procesos.....	51
Examples.....	51
Hola Mundo.....	51
Capítulo 18: Creando directorios.....	52
Introducción.....	52
Examples.....	52
Mueva todos los archivos que aún no estén en un directorio a un directorio con nombre prop.....	52
Capítulo 19: Cuándo usar eval.....	53
Introducción.....	53
Examples.....	53
Usando Eval.....	53
Usando Eval con Getopt.....	54
Capítulo 20: Declaración del caso.....	56
Examples.....	56
Declaración de un caso simple.....	56
Declaración del caso con caída a través.....	56
Caerse solo si los patrones subsiguientes coinciden.....	57
Capítulo 21: Decodificar URL.....	58
Examples.....	58
Ejemplo simple.....	58
Usando printf para decodificar una cadena.....	58
Capítulo 22: Depuración.....	59
Examples.....	59
Depurando un script bash con "-x".....	59
Comprobando la sintaxis de un script con "-n".....	59
Depurando usigh bashdb.....	59
Capítulo 23: Dividir archivos.....	61
Introducción.....	61
Examples.....	61
Dividir un archivo.....	61

Podemos usar sed con la opción w para dividir un archivo en varios archivos. Los archivos	61
Capítulo 24: División de palabras	64
Sintaxis.....	64
Parámetros.....	64
Observaciones.....	64
Examples.....	64
Partiendo con IFS.....	64
¿Qué, cuándo y por qué?.....	65
IFS y división de palabras.....	65
Malos efectos de la división de palabras.....	66
Utilidad de la división de palabras.....	67
División por cambios de separador.....	68
Capítulo 25: El comando de corte	69
Introducción.....	69
Sintaxis.....	69
Parámetros.....	69
Observaciones.....	70
Examples.....	71
Uso básico.....	71
Sólo un personaje delimitador.....	72
Los delimitadores repetidos se interpretan como campos vacíos.....	72
Sin citar.....	73
Extraer, no manipular.....	73
Capítulo 26: Encontrar	74
Introducción.....	74
Sintaxis.....	74
Examples.....	74
Buscando un archivo por nombre o extensión.....	74
Buscando archivos por tipo.....	75
Ejecutando comandos contra un archivo encontrado.....	75
Encontrar el archivo por tiempo de acceso / modificación.....	76
Buscando archivos por extensión específica.....	78

Encontrar archivos de acuerdo al tamaño.....	78
Filtrar el camino.....	79
Capítulo 27: Escollos.....	80
Examples.....	80
Espacio en blanco al asignar variables.....	80
Falta la última línea en un archivo.....	80
Los comandos fallidos no detienen la ejecución del script.....	80
Capítulo 28: Escribiendo variables.....	82
Examples.....	82
declarar variables de tipo débil.....	82
Capítulo 29: Espacio de nombres.....	83
Examples.....	83
No hay cosas tales como espacios de nombres.....	83
Capítulo 30: Estructuras de Control.....	84
Sintaxis.....	84
Parámetros.....	84
Observaciones.....	85
Examples.....	85
Si declaración.....	85
Mientras bucle.....	86
En bucle.....	86
Usando For Loop para enumerar Iterar sobre números.....	87
Para bucle con sintaxis estilo C.....	87
Hasta Loop.....	87
continuar y romper.....	88
Bucle sobre una matriz.....	88
Ruptura de bucle.....	89
Cambiar declaración con el caso.....	90
Para Loop sin un parámetro de lista de palabras.....	90
Ejecución condicional de listas de comandos.....	90
Cómo utilizar la ejecución condicional de las listas de comandos.....	90
¿Por qué usar la ejecución condicional de las listas de comandos?.....	91

Capítulo 31: Evitando la fecha usando printf	93
Introducción	93
Sintaxis	93
Observaciones	93
Examples	93
Obtén la fecha actual	93
Establecer variable a la hora actual	93
Capítulo 32: Expansión de la abrazadera	94
Observaciones	94
Examples	94
Crear directorios para agrupar archivos por mes y año	94
Crear una copia de seguridad de archivos de puntos	94
Modificar la extensión del nombre de archivo	94
Usar incrementos	94
Usando la expansión de llaves para crear listas	95
Hacer múltiples directorios con sub-directorios	95
Capítulo 33: Expansión del parámetro Bash	97
Introducción	97
Sintaxis	97
Examples	97
Subcadenas y subarreglos	97
Longitud del parámetro	99
Modificando el caso de los caracteres alfabéticos	100
Parámetro indirecto	101
Sustitución de valor por defecto	101
Error si la variable está vacía o sin configurar	102
Eliminar un patrón desde el principio de una cadena	102
Eliminar un patrón del final de una cadena	103
Reemplazar patrón en cadena	103
Munging durante la expansión	104
Expansión de parámetros y nombres de archivos	105
Capítulo 34: Expresiones condicionales	106

Sintaxis.....	106
Observaciones.....	106
Examples.....	106
Comparacion de archivos.....	106
Pruebas de acceso a archivos.....	107
Comparaciones numericas.....	107
Comparación de cadenas y emparejamiento.....	108
Pruebas de tipo de archivo.....	109
Prueba de estado de salida de un comando.....	110
Una prueba de línea.....	110
Capítulo 35: Finalización programable.....	112
Examples.....	112
Finalización simple utilizando la función.....	112
Terminación simple para opciones y nombres de archivos.....	112
Capítulo 36: Funciones.....	114
Sintaxis.....	114
Examples.....	114
Función simple.....	114
Funciones con argumentos.....	115
Valor de retorno de una función.....	116
Manejo de banderas y parámetros opcionales.....	116
El código de salida de una función es el código de salida de su último comando.....	117
Imprimir la definición de la función.....	118
Una función que acepta parámetros nombrados.....	118
Capítulo 37: Gestionando variable de entorno PATH.....	120
Sintaxis.....	120
Parámetros.....	120
Observaciones.....	120
Examples.....	120
Agregar una ruta a la variable de entorno PATH.....	120
Eliminar una ruta de la variable de entorno PATH.....	121
Capítulo 38: getopt: análisis inteligente de parámetros posicionales.....	123

Sintaxis.....	123
Parámetros.....	123
Observaciones.....	123
Opciones.....	123
Examples.....	124
mapa de ping.....	124
Capítulo 39: Grep.....	126
Sintaxis.....	126
Examples.....	126
Cómo buscar un archivo para un patrón.....	126
Capítulo 40: Leer un archivo (flujo de datos, variable) línea por línea (y / o campo por c.....	127
Parámetros.....	127
Examples.....	127
Lee archivo (/ etc / passwd) línea por línea y campo por campo.....	127
Leer las líneas de un archivo en una matriz.....	128
Bucle a través de un archivo línea por línea.....	128
Leer las líneas de una cadena en una matriz.....	129
Corriendo a través de una cuerda línea por línea.....	129
Bucle a través de la salida de una línea de comando por línea.....	129
Lee un archivo campo por campo.....	129
Leer una cadena de campo por campo.....	130
Leer los campos de un archivo en una matriz.....	130
Leer campos de una cadena en una matriz.....	131
Bucle a través de la salida de un campo de comando por campo.....	131
Capítulo 41: Listado de archivos.....	132
Sintaxis.....	132
Parámetros.....	132
Examples.....	132
Lista de archivos.....	133
Lista de archivos en un formato de listado largo.....	133
Tipo de archivo.....	134
Lista de archivos ordenados por tamaño.....	134

Lista de archivos sin usar `ls`	135
Listar los diez archivos modificados más recientemente.....	135
Listar todos los archivos incluyendo archivos de puntos.....	135
Lista de archivos en un formato de árbol.....	136
Capítulo 42: Manejando el indicador del sistema.....	137
Sintaxis.....	137
Parámetros.....	137
Examples.....	138
Usando la variable de entorno PROMPT_COMMAND.....	138
Usando ps2.....	139
Usando ps3.....	139
Usando ps4.....	139
Usando ps1.....	140
Capítulo 43: Mates.....	141
Examples.....	141
Matemáticas usando dc.....	141
Matemáticas usando bc.....	142
Matemáticas usando las capacidades de bash.....	142
Matemáticas usando expr.....	143
Capítulo 44: Matrices asociativas.....	145
Sintaxis.....	145
Examples.....	145
Examinando arrays assoc.....	145
Capítulo 45: Navegando directorios.....	147
Examples.....	147
Cambiar al último directorio.....	147
Cambiar al directorio de inicio.....	147
Directorios absolutos vs relativos.....	147
Cambio al Directorio del Guión.....	148
Capítulo 46: Oleoductos.....	149
Sintaxis.....	149
Observaciones.....	149

Examples.....	149
Mostrar todos los procesos paginados.....	149
Utilizando &.....	149
Modificar la salida continua de un comando.....	150
Capítulo 47: Paralela.....	151
Introducción.....	151
Sintaxis.....	151
Parámetros.....	151
Examples.....	152
Paralelizar tareas repetitivas en la lista de archivos.....	152
Paralelizar STDIN.....	152
Capítulo 48: Patrón de coincidencia y expresiones regulares.....	154
Sintaxis.....	154
Observaciones.....	154
Examples.....	155
Compruebe si una cadena coincide con una expresión regular.....	155
El * glob.....	155
El ** glob.....	156
Los ? glob.....	156
El [] glob.....	157
Coincidencia de archivos ocultos.....	158
Coincidencia insensible a mayúsculas.....	158
Comportamiento cuando un glob no coincide con nada.....	159
Globo extendido.....	159
Coincidencia de expresiones regulares.....	160
Obtener grupos capturados de una coincidencia de expresiones regulares contra una cadena.....	161
Capítulo 49: Patrones de diseño.....	162
Introducción.....	162
Examples.....	162
El patrón de publicación / suscripción (Pub / Sub).....	162
Capítulo 50: Personalizando PS1.....	164
Examples.....	164

Cambiar el indicador de PS1	164
Mostrar una rama git usando PROMPT_COMMAND	165
Mostrar el nombre de la rama git en el indicador de terminal	166
Mostrar hora en terminal	166
Colorear y personalizar el indicador de terminal	167
Mostrar el estado y el tiempo de retorno del comando anterior	168
Capítulo 51: Proceso de sustitución	170
Observaciones	170
Examples	170
Compara dos archivos de la web	170
Alimenta un bucle while con la salida de un comando	170
Con comando de pegar	170
Concatenando archivos	170
Transmitir un archivo a través de múltiples programas a la vez	171
Para evitar el uso de un sub-shell	171
Capítulo 52: Redes con Bash	173
Introducción	173
Examples	173
Comandos de red	173
Capítulo 53: Redirección	175
Sintaxis	175
Parámetros	175
Observaciones	175
Examples	176
Redireccionando salida estándar	176
Redireccionando STDIN	176
Redireccionando tanto STDOUT como STDERR	177
Redireccionando STDERR	177
Anexar vs Truncar	178
Truncar >	178
Añadir >>	178
STDIN, STDOUT y STDERR explicados	178

Redirigiendo múltiples comandos al mismo archivo.....	179
Utilizando tuberías con nombre.....	180
Imprimir mensajes de error a stderr.....	182
Redireccionamiento a direcciones de red.....	183
Capítulo 54: Salida de script en color (multiplataforma).....	184
Observaciones.....	184
Examples.....	184
color-output.sh.....	184
Capítulo 55: Script Shebang.....	186
Sintaxis.....	186
Observaciones.....	186
Examples.....	186
Shebang directo.....	186
Env Shebang.....	186
Otros shebangs.....	187
Capítulo 56: Scripts CGI.....	188
Examples.....	188
Método de solicitud: GET.....	188
Método de solicitud: POST / w JSON.....	190
Capítulo 57: Secuencia de ejecución de archivo.....	193
Introducción.....	193
Observaciones.....	193
Examples.....	193
.profile vs .bash_profile (y .bash_login).....	193
Capítulo 58: Secuencias de comandos con parámetros.....	194
Observaciones.....	194
Examples.....	194
Análisis de múltiples parámetros.....	194
Parámetros de acceso.....	195
Obteniendo todos los parametros.....	195
Obtener el número de parámetros.....	195
Ejemplo 1.....	196

Ejemplo 2.....	196
Análisis de argumentos utilizando un bucle for.....	196
Guión envoltorio.....	197
Dividir cadena en una matriz en Bash.....	198
Capítulo 59: Seleccionar palabra clave.....	199
Introducción.....	199
Examples.....	199
La palabra clave seleccionada se puede usar para obtener el argumento de entrada en un for.....	199
Capítulo 60: strace.....	200
Sintaxis.....	200
Examples.....	200
Cómo observar las llamadas al sistema de un programa.....	200
Capítulo 61: Tipo de conchas.....	201
Observaciones.....	201
Examples.....	201
Introducción a los archivos de puntos.....	201
Iniciar un shell interactivo.....	202
Detectar el tipo de concha.....	203
Capítulo 62: Trabajos en tiempos específicos.....	204
Examples.....	204
Ejecutar el trabajo una vez en un momento específico.....	204
Haciendo trabajos en momentos específicos repetidamente usando systemd.timer.....	204
Capítulo 63: Trabajos y Procesos.....	206
Examples.....	206
Listar trabajos actuales.....	206
Manejo de trabajos.....	206
Creando empleos.....	206
Fondo y primer plano de un proceso.....	206
Matando trabajos en ejecución.....	207
Iniciar y matar procesos específicos.....	208
Listar todos los procesos.....	209

Compruebe qué proceso se está ejecutando en un puerto específico.....	209
Encontrar información sobre un proceso en ejecución.....	209
Desaprobando trabajo de fondo.....	210
Capítulo 64: Transferencia de archivos usando scp.....	211
Syntax.....	211
Examples.....	211
archivo de transferencia de scp.....	211
scp transfiriendo múltiples archivos.....	211
Descargando archivo usando scp.....	211
Capítulo 65: Usando "trampa" para reaccionar a señales y eventos del sistema.....	212
Syntax.....	212
Parámetros.....	212
Observaciones.....	212
Examples.....	213
La captura de SIGINT o Ctl + C.....	213
Introducción: limpiar archivos temporales.....	213
Acumula una lista de trabajo de trampa para ejecutar en la salida.....	214
Matando procesos infantiles en la salida.....	214
reaccionar al cambiar el tamaño de la ventana de terminales.....	215
Capítulo 66: Usando gato.....	216
Syntax.....	216
Parámetros.....	216
Observaciones.....	216
Examples.....	216
Impresión del contenido de un archivo.....	216
Mostrar números de línea con salida.....	217
Leer de entrada estándar.....	218
Concatenar archivos.....	218
Escribir en un archivo.....	218
Mostrar caracteres no imprimibles.....	219
Concatenar archivos comprimidos.....	219
Capítulo 67: Usando orden.....	221

Introducción.....	221
Sintaxis.....	221
Parámetros.....	221
Observaciones.....	221
Examples.....	221
Ordenar orden de salida.....	221
Haz que la salida sea única.....	221
Tipo numérico.....	222
Ordenar por llaves.....	222
Capítulo 68: Utilidad de dormir.....	225
Introducción.....	225
Examples.....	225
\$ dormir 1.....	225
Capítulo 69: variables globales y locales.....	226
Introducción.....	226
Examples.....	226
Variables globales.....	226
Variables locales.....	226
Mezclando los dos juntos.....	226
Capítulo 70: Variables internas.....	228
Introducción.....	228
Examples.....	228
Bash variables internas de un vistazo.....	228
\$ BASHPID.....	230
\$ BASH_ENV.....	230
\$ BASH_VERSINFO.....	230
\$ BASH_VERSION.....	230
\$ EDITOR.....	230
\$ FUNCNAME.....	230
\$ HOME.....	231
\$ HOSTNAME.....	231
\$ HOSTTYPE.....	231

\$ GRUPOS.....	231
\$ IFS.....	231
\$ LINENO.....	232
\$ MACHTYPE.....	232
\$ OLDPWD.....	232
\$ OSTYPE.....	232
\$ PATH.....	233
\$ PPID.....	233
\$ PWD.....	233
\$ SEGUNDOS.....	233
\$ SHELLOPTS.....	233
\$ SHLVL.....	234
\$ UID.....	235
\$ 1 \$ 2 \$ 3 etc	235
PS.....	236
PS.....	236
PS.....	236
PS.....	236
PS.....	237
\$\$.....	237
PS.....	237
\$ HISTSIZE.....	238
\$ RANDOM.....	238
Capítulo 71: verdadero, falso y: comandos.....	240
Sintaxis.....	240
Examples.....	240
Bucle infinito.....	240
Función de retorno.....	240
Código que siempre / nunca será ejecutado.....	240
Creditos.....	242

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: [bash](#)

It is an unofficial and free Bash ebook created for educational purposes. All the content is extracted from [Stack Overflow Documentation](#), which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Bash.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con Bash

Versiones

Versión	Fecha de lanzamiento
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

Examples

Hola mundo usando variables

Cree un nuevo archivo llamado `hello.sh` con el siguiente contenido y `chmod +x hello.sh` permisos ejecutables con `chmod +x hello.sh`.

Ejecutar / Ejecutar vía: `./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` a la salida estándar cuando se ejecute.

Para decirle a `bash` dónde está el script, debe ser muy específico, apuntándolo al directorio que contiene, normalmente con `./` si es su directorio de trabajo, dónde `.` Es un alias del directorio actual. Si no especifica el directorio, `bash` intenta localizar el script en uno de los directorios contenidos en la `$PATH` entorno `$PATH` .

El siguiente código acepta un argumento `$1` , que es el primer argumento de la línea de comando, y lo envía en una cadena con formato, siguiendo a `Hello,`

Ejecutar / Ejecutar vía: `./hello.sh World`

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

Es importante tener en cuenta que `$1` debe cotizarse entre comillas dobles, no comillas simples. `"$1"` expande al primer argumento de la línea de comando, según lo deseado, mientras que `'$1'` evalúa como una cadena literal `$1` .

Nota de seguridad:

Lea las [implicaciones de seguridad de olvidarse de citar una variable en las bases](#) para entender la importancia de colocar el texto de la variable entre comillas dobles.

Hola Mundo

Shell interactivo

El shell Bash se usa comúnmente de forma **interactiva**: **le** permite ingresar y editar comandos y luego ejecutarlos cuando presiona la tecla `Retorno` . Muchos sistemas operativos basados en Unix y similares a Unix usan Bash como su shell predeterminado (en particular, Linux y macOS). El terminal ingresa automáticamente un proceso de shell Bash interactivo en el inicio.

Salida `Hello World` escribiendo lo siguiente:

```
echo "Hello World"
#> Hello World # Output Example
```

Notas

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

Shell no interactivo

El shell Bash también se puede ejecutar de forma **no interactiva** desde un script, lo que hace que el shell no requiera interacción humana. El comportamiento interactivo y el guión deben ser idénticos: una consideración de diseño importante para el shell Bourne de Unix V7 y el tránsito de Bash. Por lo tanto, cualquier cosa que se pueda hacer en la línea de comandos se puede poner en un archivo de script para su reutilización.

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

1. Crea un nuevo archivo llamado `hello-world.sh`

```
touch hello-world.sh
```

2. Haz el script ejecutable 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 `#!` , referido como *shebang* ¹ . El shebang le indica al sistema operativo que ejecute `/bin/bash` , el shell Bash, y le pasa 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 comando usando uno de los siguientes:

- `./hello-world.sh` - más comúnmente usado y recomendado
- `/bin/bash hello-world.sh`
- `bash hello-world.sh` - asumiendo que `/bin` está en tu `$PATH`
- `sh hello-world.sh`

Para un uso de producción real, `.sh` extensión `.sh` (que de todas formas es engañosa, ya que se trata de un script Bash, no de un script `sh`) y tal vez mueva el archivo a un directorio dentro de su `PATH` para que esté disponible sin importar su directorio de trabajo actual, al igual que un comando del sistema como `cat 0 ls .`

Los errores comunes incluyen:

1. Olvidarse de aplicar el permiso de ejecución en el archivo, es decir, `chmod +x hello-world.sh`, dando como resultado la salida de `./hello-world.sh: Permission denied.`
2. Edición del script en Windows, 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 la línea, sobrescribiendo el texto antes de los dos puntos en el mensaje de error.

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

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

`dos2unix` edita el archivo en línea.

3. Usando `sh ./hello-world.sh`, no me `sh ./hello-world.sh` cuenta de que `bash` y `sh` son carcasas distintas con características distintas (aunque Bash es compatible con versiones anteriores, el error opuesto es inofensivo).

De todos modos, simplemente confiar en la línea shebang del script es sumamente preferible a 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 usar con el fin de hacer que tu script sea más portátil es usar `#!/usr/bin/env bash` lugar de codificar de forma rígida una ruta a Bash. De esa manera, `/usr/bin/env` tiene que existir, pero más allá de ese punto, `bash` solo necesita estar en su `PATH`. En muchos sistemas, `/bin/bash` no existe, y debe usar `/usr/local/bin/bash` o alguna otra ruta absoluta; este cambio evita tener que averiguar los detalles de eso.

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

Visualización de información para Bash incorporados

```
help <command>
```

Esto mostrará la página de ayuda (manual) de Bash para la función incorporada especificada.

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 todas las incorporaciones con una breve descripción, use

```
help -d
```

Hola mundo con entrada de usuario

Lo siguiente le pedirá a un usuario que ingrese información, y luego la almacenará como una cadena (texto) en una variable. La variable se utiliza para dar un mensaje al usuario.

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

El comando `read` aquí lee una línea de datos de la entrada estándar en el `name` la variable. Luego se hace referencia a esto usando `$name` y se imprime a una salida estándar usando `echo`.

Ejemplo de salida:

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

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

Y si desea agregar algo al valor de la variable mientras lo imprime, use corchetes 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 ingresa una acción, se agrega "ing" a esa acción mientras se imprime.

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"

# how to run
# chmod +x script.sh
# ./script.sh --deploy true --uglify false
```

Hola mundo en modo "depurar"

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

El argumento `-x` permite caminar a través de cada línea en el script. Un buen ejemplo es aquí:

```
$ 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 anterior no es suficiente para rastrear el script; sin embargo, el uso de la siguiente manera 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
```

Importancia de citar en cuerdas

Las citas son importantes para la expansión de cadenas en bash. Con estos, puedes controlar cómo el bash analiza y expande tus cadenas.

Hay dos tipos de citas:

- **Débil** : *utiliza comillas dobles: "*
- **Fuerte** : *usa comillas simples: '*

Si quiere golpear para expandir su argumento, puede usar **Weak Quoting** :

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

Si no quiere golpear para expandir su argumento, puede usar **Strong Quoting** :

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

También puedes usar escape para evitar la expansión:

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

Para obtener información más detallada además de los detalles para principiantes, puede continuar leyendo [aquí](https://riptutorial.com/es/bash/topic/300/empezando-con-bash) .

Lea Empezando con Bash en línea: <https://riptutorial.com/es/bash/topic/300/empezando-con-bash>

Capítulo 2: Abastecimiento

Examples

Obtención de un archivo

La fuente de un archivo es diferente de la ejecución, ya que todos los comandos se evalúan en el contexto de la sesión de bash actual, lo que significa que cualquier variable, función o alias definidos permanecerán durante la sesión.

Crea el archivo que deseas fuente `sourceme.sh`

```
#!/bin/bash

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

Desde tu sesión, fuente el archivo.

```
$ source sourceme.sh
```

De aquí en adelante, tiene todos los recursos del archivo de origen disponible

```
$ echo $A
hello_world

$ sayHi
Hi

$ sayHello
Hello
```

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

```
$ . sourceme.sh
```

Aprovisionamiento de un entorno virtual.

Al desarrollar varias aplicaciones en una máquina, resulta útil separar las dependencias en entornos virtuales.

Con el uso de `virtualenv`, estos entornos se obtienen de su shell para que cuando ejecute un comando, provenga de ese entorno virtual.

Esto se instala más comúnmente utilizando `pip`.

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

Crear un nuevo entorno.

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

Activar el medio ambiente.

```
source my_env/bin/activate
```

Lea Abastecimiento en línea: <https://riptutorial.com/es/bash/topic/564/abastecimiento>

Capítulo 3: Alcance

Examples

Ámbito dinámico en acción.

El alcance dinámico significa que las búsquedas de variables ocurren en el ámbito donde se *llama* una función, no donde se *define*.

```
$ 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` *siempre* buscaría en el ámbito global el valor de `x`, porque `func1` se *define* en el ámbito local.

En un lenguaje de alcance dinámico, `func1` busca en el ámbito donde se *llama*. Cuando se llama desde `func2`, primero busca en el cuerpo de `func2` un valor de `x`. Si no estuviera definido allí, se vería en el ámbito global, desde donde se llamó a `func2`.

Lea Alcance en línea: <https://riptutorial.com/es/bash/topic/2452/alcance>

Capítulo 4: Aliasing

Introducción

Los alias de shell son una forma sencilla de crear nuevos comandos o de envolver comandos existentes con su propio código. En cierto modo, se superponen con las [funciones de shell](#), que, sin embargo, son más versátiles y, por lo tanto, a menudo se prefieren.

Observaciones

El alias solo estará disponible en el shell donde se emitió el comando alias.

Para persistir el alias considera ponerlo en tu `.bashrc`

Examples

Crear un alias

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

Al invocar la `word` se ejecutará el `command`. Cualquier argumento proporcionado al alias se anexa simplemente al destino del alias:

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

El shell entonces ejecutará:

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

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

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

Listar todos los alias

```
alias -p
```

listará todos los alias actuales.

Expandir alias

Suponiendo que la `bar` es un alias para `someCommand -flag1`.

Escriba `bar` en la línea de comando y luego presione `Ctrl + alt + e`

obtendrá un `someCommand -flag1` donde estaba parado el `bar`.

Eliminar un alias

Para eliminar un alias existente, use:

```
unalias {alias_name}
```

Ejemplo:

```
# create an alias
$ alias now='date'

# preview the alias
$ now
Thu Jul 21 17:11:25 CEST 2016

# remove the alias
$ unalias now

# test if removed
$ now
-bash: now: command not found
```

Omitir un alias

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

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

Y digamos que desea utilizar el comando `ls` sin deshabilitar el alias. Tienes varias opciones:

- Usa el `command builtin`: `command ls`
- Use la ruta completa del comando: `/bin/ls`
- Agregue un `\` en cualquier lugar en el nombre del comando, por ejemplo: `\ls`, o `l\s`
- Cita el comando: `"ls"` o `'ls'`

BASH_ALIASES es un arreglo interno de bash

Los alias se denominan accesos directos de comandos, uno puede definir y usar en instancias de bash interactivas. Se mantienen en una matriz asociativa llamada `BASH_ALIASES`. Para usar esta var en un script, debe ejecutarse dentro de un shell interactivo

```
#!/bin/bash -li
# note the -li above! -l makes this behave like a login shell
# -i makes it behave like an interactive shell
#
# shopt -s expand_aliases will not work in most cases
```



```
echo There are ${#BASH_ALIASES[*]} aliases defined.  
  
for ali in "${!BASH_ALIASES[@]}"; do  
    printf "alias: %-10s triggers: %s\n" "$ali" "${BASH_ALIASES[$ali]}"  
done
```

Lea Aliasing en línea: <https://riptutorial.com/es/bash/topic/368/aliasing>

Capítulo 5: Aquí los documentos y aquí las cadenas.

Examples

Sangrando aquí documentos

Puede sangrar el texto dentro de los documentos aquí con pestañas, necesita usar el operador de redirección `<<-` 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 de uso práctico de esto (como se menciona en `man bash`) está 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 declaración `if` , para una mejor legibilidad. Sin la sintaxis de operador `<<-` , nos veríamos obligados a escribir el código anterior de esta manera:

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

Eso es muy desagradable de leer y empeora mucho en un guión realista más complejo.

Aquí cuerdas

2.05b

Puedes alimentar un comando usando aquí cadenas como esta:

```
$ awk '{print $2}' <<< "hello world - how are you?"
world

$ awk '{print $1}' <<< "hello how are you
> she is fine"
hello
she
```

También puede alimentar a un `while` de bucle con una cadena aquí:

```
$ while IFS=" " read -r word1 word2 rest
> do
> echo "$word1"
> done <<< "hello how are you - i am fine"
hello
```

Cadenas de límite

Un heredoc usa la *cadena de límite* para determinar cuándo dejar de consumir información. La cadena de límite de terminación **debe**

- Estar al comienzo de una línea.
- Sea el único texto en la línea. **Nota:** Si usa `<<-` la cadena de límite puede tener el prefijo de tabs `\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 la `limitstring` de `limitstring` en la última línea no se encuentra exactamente al inicio de la línea, el shell continuará esperando más información hasta que vea una línea que comienza con la `limitstring` y no contiene nada más. Solo entonces dejará de esperar la entrada y procederá a pasar el documento aquí al comando `cat` .

Tenga en cuenta que cuando realiza el prefijo de la cadena de límite inicial con un guión, todas las pestañas al inicio de la línea se eliminan antes del análisis, por lo que los datos y la cadena de límite se pueden sangrar con pestañas (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
```

con las pestañas iniciales (pero no las pestañas internas) eliminadas.

Crear un archivo

Un uso clásico de los documentos de aquí es crear un archivo escribiendo su contenido:

```
cat > fruits.txt << EOF
apple
orange
lemon
EOF
```

El documento aquí son las líneas entre `<< EOF` y `EOF` .

Este documento aquí se convierte en la entrada del comando `cat` . El `cat` de comandos simplemente da salida a su entrada, y utilizando el operador de redirección de salida `>` que redirigir 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 existió antes, será truncado.

Ejecutar comando con el documento aquí

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

\$

se escapa porque no queremos que se expanda por el shell actual, es decir, `$(pwd)` se ejecutará en el shell remoto.

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, se pueden usar pestañas si comienza su heredoc con `<<-` . Vea los ejemplos de [sangría aquí](#) y [cadenas de límite](#) para obtener más información.

Ejecuta varios comandos con sudo.

```
sudo -s <<EOF
a='var'
echo 'Running serveral commands with sudo'
mktemp -d
echo "$a"
EOF
```

- `$a` debe ser evitado 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
```

Lea Aquí los documentos y aquí las cadenas. en línea:

<https://riptutorial.com/es/bash/topic/655/aqui-los-documentos-y-aqui-las-cadenas->

Capítulo 6: Aritmética de Bash

Sintaxis

- `$ ((EXPRESSION))` - Evalúa la expresión y devuelve su resultado.
- `EXPRESSION expr` - Imprime el resultado de EXPRESIÓN a la salida estándar.

Parámetros

Parámetro	Detalles
EXPRESIÓN	Expresión para evaluar.

Observaciones

Se requiere un espacio (") entre cada término (o signo) de la expresión. "1 + 2" no funcionará, pero "1 + 2" funcionará.

Examples

Comando aritmético

- `let`

```
let num=1+2
let num="1+2"
let 'num= 1 + 2'
let num=1 num+=2
```

Necesitas comillas si hay espacios o caracteres globosos. Así que esos obtendrán error:

```
let num = 1 + 2      #wrong
let 'num = 1 + 2'    #right
let a[1] = 1 + 1     #wrong
let 'a[1] = 1 + 1'   #right
```

- `(())`

```
((a=$a+1))           #add 1 to a
((a = a + 1))         #like above
((a += 1))            #like above
```

Podemos usar `(())` en `if`. Algunos ejemplos:

```
if (( a > 1 )); then echo "a is greater than 1"; fi
```

La salida de `(())` se puede asignar a una variable:

```
result=$((a + 1))
```

O utilizado directamente en la salida:

```
echo "The result of a + 1 is $((a + 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

Aritmética simple con `expr`

```
#!/bin/bash
expr 1 + 2
```

Salida: 3

Lea Aritmética de Bash en línea: <https://riptutorial.com/es/bash/topic/3652/aritmetica-de-bash>

Capítulo 7: Arrays

Examples

Asignaciones de matrices

Asignación de lista

Si está familiarizado con Perl, C o Java, podría pensar que Bash usaría comas para separar los elementos de la matriz, sin embargo, este no es el caso; en su lugar, Bash usa espacios:

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

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

Crea una matriz con nuevos elementos:

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

Asignación de subíndices

Crear una matriz 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 (matriz asociativa)

4.0

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

Asignación dinámica

Cree una matriz a partir de la salida de otro comando, por ejemplo, use **seq** para obtener un rango de 1 a 10:

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


Asignación de los argumentos de entrada del script:

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

Asignación dentro de bucles:

```
while read -r; do
    #array+=("$REPLY")      # Array append
    array[$i]="$REPLY"     # Assignment by index
    let i++                # Increment index
done < <(seq 1 10)        # command substitution
echo ${array[@]}          # output: 1 2 3 4 5 6 7 8 9 10
```

donde `$REPLY` es siempre la entrada actual

Accediendo a Array Elements

Elemento de impresión en el índice 0

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

4.3

Imprimir el último elemento utilizando la sintaxis de expansión de subcadena

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

4.3

Imprimir el último elemento usando la sintaxis de los subíndices

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

Imprime todos los elementos, cada uno citado por separado

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

Imprimir todos los elementos como una sola cadena entre comillas

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

Imprima todos los elementos del índice 1, cada uno citado por separado

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

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

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

Operaciones de cadena

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

```
array=(zero one two)
echo "${array[0]:0:3}" # gives out zer (chars at position 0, 1 and 2 in the string zero)
echo "${array[0]:1:3}" # gives out ero (chars at position 1, 2 and 3 in the string zero)
```

entonces `${array[$i]:N:M}` genera una cadena desde la posición `N` (comenzando desde 0) en la cadena `${array[$i]}` con `M` siguiendo los caracteres.

Longitud de la matriz

`${#array[@]}` da la longitud de la matriz `${array[@]}` :

```
array=('first element' 'second element' 'third element')
echo "${#array[@]}" # gives out a length of 3
```

Esto funciona también con cuerdas en elementos individuales:

```
echo "${#array[0]}" # gives out the lenght of the string at element 0: 13
```

Modificación de Array

Cambiar índice

Inicialice o actualice un elemento particular en la matriz

```
array[10]="elevenths element" # because it's starting with 0
```

3.1

Adjuntar

Modifique la matriz, agregando elementos al final si no se especifica ningún subíndice.

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

Reemplace toda la matriz con una nueva lista de parámetros.

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

Agrega un elemento al principio:

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

Insertar

Insertar un elemento en un índice dado:

```
arr=(a b c d)
# insert an element at index 2
i=2
arr=("${arr[@]:0:$i}" 'new' "${arr[@]:$i}")
echo "${arr[2]}" #output: new
```

Borrar

Elimine los índices de matriz utilizando la función `unset` builtin:

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

Unir

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

Esto funciona también para arreglos dispersos.

Re-indexando una matriz

Esto puede ser útil si se han eliminado elementos de una matriz, o si no está seguro de si hay brechas en la matriz. Para recrear los índices sin lagunas:

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

Iteración de Array

La iteración de arreglos viene en dos sabores, `foreach` y el clásico `for-loop`:

```
a=(1 2 3 4)
# foreach loop
for y in "${a[@]"; do
    # act on $y
    echo "$y"
done
# classic for-loop
for ((idx=0; idx < ${#a[@]}; ++idx)); do
    # act on ${a[$idx]}
    echo "${a[$idx]}"
done
```

También puede iterar sobre la salida de un comando:

```
a=($(tr ',' ' ' <<<"a,b,c,d")) # tr can transform one character to another
```

```
for y in "${a[@]"; do
    echo "$y"
done
```

Destruir, eliminar o desarmar una matriz

Para destruir, eliminar o desarmar una matriz:

```
unset array
```

Para destruir, eliminar o desarmar un solo elemento de matriz:

```
unset array[10]
```

Matrices asociativas

4.0

Declara una matriz asociativa

```
declare -A aa
```

La declaración de una matriz asociativa antes de la inicialización o el uso es obligatoria.

Inicializar elementos

Puede inicializar los elementos uno a la vez de la siguiente manera:

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

También puede inicializar una matriz asociativa completa en una sola declaración:

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

Acceder a un elemento de matriz asociativa.

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

Listado de claves de matriz asociativa

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

Listado de valores de matriz asociativa

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

Iterar sobre claves y valores de matriz asociativa

```
for key in "${!aa[@]}"; do
    echo "Key:    ${key}"
    echo "Value:  ${array[$key]}"
done

# Out:
# Key:    hello
# Value:  world
# Key:    ab
# Value:  cd
# Key:    key with space
# Value:  hello world
```

Contar elementos de matrices asociativas.

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

Lista de índices inicializados

Obtener la lista de índices inicializados en una matriz

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

Buceando a través de una matriz

Nuestro ejemplo de matriz:

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

Usando un bucle `for..in`:

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

2.04

Usando C-style `for` loop:

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

```
done
```

Usando `while` loop:

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

2.04

Usando `while` bucle con condicional numérica:

```
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
```

2.04

Usando un bucle `until` con condicional numérico:

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

Matriz de cuerda

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

Cada espacio en la cadena denota un nuevo elemento en la matriz resultante.

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

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

```
stringVar="Apple+Orange+Banana+Mango"
```

```
arrayVar=(${stringVar//+/ })
echo ${arrayVar[0]} # will print Apple
echo ${arrayVar[2]} # will print Banana
```

Función de inserción de matriz

Esta función insertará un elemento en una matriz 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 # reference to the array variable
    i=$2                  # index to insert at
    el="$3"               # element to insert
    # handle errors
    [[ ! "$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; }
    # Now insert $el at $i
    __arr__=("${__arr__[0:$i]}" "$el" "${__arr__[i:$i]}")
}
```

Uso:

```
insert array_variable_name index element
```

Ejemplo:

```
arr=(a b c d)
echo "${arr[2]}" # output: c
# Now call the insert function and pass the array variable name,
# index to insert at
# and the element to insert
insert arr 2 'New Element'
# 'New Element' was inserted at index 2 in arr, now print them
echo "${arr[2]}" # output: New Element
echo "${arr[3]}" # output: c
```

Leyendo un archivo entero en una matriz

Leyendo en un solo paso:

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

Leyendo en un bucle:

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

4.0

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

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

Lea Arrays en línea: <https://riptutorial.com/es/bash/topic/471/arrays>

Capítulo 8: Atajos de teclado

Observaciones

`bind -P` muestra todos los accesos directos configurados.

Examples

Recordar atajos

Atajo	Descripción
Ctrl + r	buscar la historia hacia atrás
Ctrl + p	comando anterior en la historia
Ctrl + n	próximo comando en la historia
Ctrl + g	salir del modo de búsqueda de historial
Alt + .	Usa la última palabra del comando anterior.
	repita para obtener la última palabra del comando + 1 anterior
Alt + n Alt + .	usar la palabra nth del comando anterior
!! + Regresar	ejecuta el último comando otra vez (útil cuando olvidaste sudo: <code>sudo !!</code>)

Edición de atajos

Atajo	Descripción
Ctrl + a	mover al principio de la línea
Ctrl + e	mover al final de la línea
Ctrl + k	Mata el texto desde la posición actual del cursor hasta el final de la línea.
Ctrl + u	Mata el texto desde la posición actual del cursor hasta el principio de la línea.
Ctrl + w	Mata la palabra detrás de la posición actual del cursor
Alt + b	retroceder una palabra
Alt + f	avanzar una palabra

Atajo	Descripción
Ctrl + Alt + e	shell expandir línea
Ctrl + y	Haz retroceder el texto eliminado más recientemente en el búfer en el cursor.
Alt + y	Rotar a través del texto eliminado. Solo puede hacer esto si el comando anterior es Ctrl + y o Alt + y .

El texto de eliminación eliminará el texto, pero lo guardará para que el usuario pueda reinsertarlo tirando. Similar a cortar y pegar, excepto que el texto se coloca en un anillo de eliminación que permite almacenar más de un conjunto de texto para volver a colocarlo en la línea de comandos.

Puedes encontrar más información en el [manual de emacs](#) .

Control de trabajo

Atajo	Descripción
Ctrl + c	Detener el trabajo actual
Ctrl + z	Suspender el trabajo actual (enviar una señal SIGTSTP)

Macros

Atajo	Descripción
Ctrl + x , (empezar a grabar una macro
Ctrl + x ,)	dejar de grabar una macro
Ctrl + x , e	Ejecutar la última macro grabada.

Custome Key Bindings

Con el `bind` de comandos es posible definir atajos de teclado personalizado.

El siguiente ejemplo vincula un `Alt + w a` `>/dev/null 2>&1 :`

```
bind '"\ew"' : "\ " >/dev/null 2>&1 "\ "
```

Si desea ejecutar la línea inmediatamente agregue `\Cm (Enter)` a ella:

```
bind '"\ew"' : "\ " >/dev/null 2>&1 \C-m "\ "
```

Lea Atajos de teclado en línea: <https://riptutorial.com/es/bash/topic/3949/atajos-de-teclado>

Capítulo 9: Bash en Windows 10

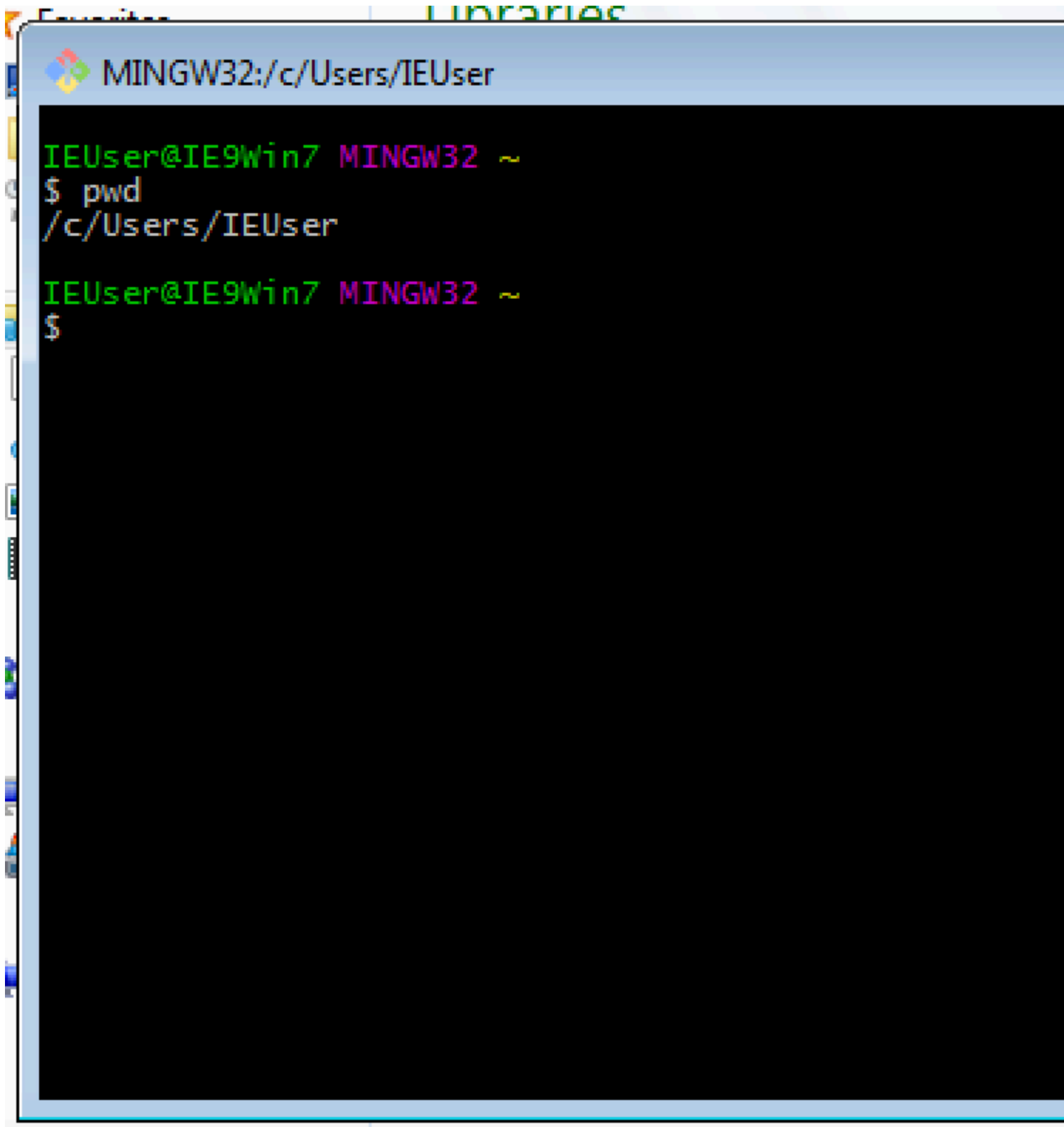
Examples

Readme

La forma más sencilla de usar Bash en Windows es instalar Git para Windows. Se envía con Git Bash, que es un verdadero Bash. Se puede acceder con acceso directo en:

Start > All Programs > Git > Git Bash

Comandos como `grep`, `ls`, `find`, `sed`, `vi`, etc. están funcionando.



```
MINGW32:/c/Users/IEUser

IEUser@IE9Win7 MINGW32 ~
$ pwd
/c/Users/IEUser

IEUser@IE9Win7 MINGW32 ~
$
```

Lea Bash en Windows 10 en línea: <https://riptutorial.com/es/bash/topic/9114/bash-en-windows-10>

Capítulo 10: Bash sustituciones de historia

Examples

Utilizando! \$

Puede usar !\$ Para reducir la repetición al usar la línea de comando:

```
$ echo ping
ping
$ echo !$
ping
```

También puedes construir sobre la repetición.

```
$ echo !$ pong
ping pong
$ echo !$, a great game
pong, a great game
```

Tenga en cuenta que en el último ejemplo no obtuvimos `ping pong, a great game` porque el último argumento pasado al comando anterior fue `pong`, podemos evitar este tipo de problemas agregando comillas. Continuando con el ejemplo, nuestro último argumento fue el `game`:

```
$ echo "it is !$ time"
it is game time
$ echo "hooray, !$!"
hooray, it is game time!
```

Referencia rápida

Interacción con la historia.

```
# List all previous commands
history

# Clear the history, useful if you entered a password by accident
history -c
```

Designadores de eventos

```
# Expands to line n of bash history
!n

# Expands to last command
!!

# Expands to last command starting with "text"
```

```

!text

# Expands to last command containing "text"
! ?text

# Expands to command n lines ago
! -n

# Expands to last command with first occurrence of "foo" replaced by "bar"
^foo^bar^

# Expands to the current command
!#

```

Designadores de palabras

Estos están separados por : desde el designador del evento al que se refieren. Los dos puntos se pueden omitir si la palabra designador no comienza con un número !^ Es igual que !: ^ .

```

# Expands to the first argument of the most recent command
!^

# Expands to the last argument of the most recent command (short for !!: $)
!$

# Expands to the third argument of the most recent command
!:3

# Expands to arguments x through y (inclusive) of the last command
# x and y can be numbers or the anchor characters ^ $
!:x-y

# Expands to all words of the last command except the 0th
# Equivalent to : ^-$
!*

```

Modificadores

Estos modifican el evento precedente o designador de la palabra.

```

# Replacement in the expansion using sed syntax
# Allows flags before the s and alternate separators
:s/foo/bar/ #substitutes bar for first occurrence of foo
:gs|foo|bar| #substitutes bar for all foo

# Remove leading path from last argument ("tail")
:t

# Remove trailing path from last argument ("head")
:h

# Remove file extension from last argument
:r

```

Si la variable Bash HISTCONTROL contiene un ignorespace o uno ignoreboth (o, alternativamente, HISTIGNORE contiene el patrón []*), puede evitar que sus comandos se almacenen en el historial

de Bash añadiéndoles un espacio:

```
# This command won't be saved in the history
foo

# This command will be saved
bar
```

Buscar en el historial de comandos por patrón

Presiona `control r` y escribe un patrón.

Por ejemplo, si recientemente ejecutado `man 5 crontab`, puede encontrarlo rápidamente al *empezar a escribir* "crontab". El aviso cambiará así:

```
(reverse-i-search)`cr': man 5 crontab
```

El ``cr'` es la cadena que escribí hasta ahora. Esta es una búsqueda incremental, por lo que a medida que continúa escribiendo, el resultado de la búsqueda se actualiza para que coincida con el comando más reciente que contenía el patrón.

Presione las teclas de flecha hacia la izquierda o hacia la derecha para editar el comando correspondiente antes de ejecutarlo, o la tecla `Intro` para ejecutar el comando.

Por defecto, la búsqueda encuentra el comando ejecutado más reciente que coincide con el patrón. Para retroceder en el historial, presiona `control r` nuevamente. Puede presionarlo varias veces hasta que encuentre el comando deseado.

Cambie al directorio recién creado con! #: N

```
$ mkdir backup_download_directory && cd !#:1
mkdir backup_download_directory && cd backup_download_directory
```

Esto sustituirá el argumento Nth del comando actual. En el ejemplo `!#:1` se reemplaza con el primer argumento, es decir, `backup_download_directory`.

Repita el comando anterior con una sustitución.

```
$ mplayer Lecture_video_part1.mkv
$ ^1^2^
mplayer Lecture_video_part2.mkv
```

Este comando reemplazará `1` por `2` en el comando ejecutado anteriormente. Solo reemplazará la primera aparición de la cadena y es equivalente a `!!:s/1/2/ .`

Si desea reemplazar *todas las* ocurrencias, debe usar `!!:gs/1/2/ o !!:as/1/2/ .`

Repita 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>:
```

Lea Bash sustituciones de historia en línea: <https://riptutorial.com/es/bash/topic/1519/bash-sustituciones-de-historia>

Capítulo 11: Cadena de comandos y operaciones.

Introducción

Hay algunos medios para encadenar comandos. Los simples como solo un; o más complejas como cadenas lógicas que se ejecutan dependiendo de algunas condiciones. El tercero son los comandos de canalización, que efectivamente transfieren los datos de salida al siguiente comando en la cadena.

Examples

Contando 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 usa como entrada del comando `grep`. El resultado será la cantidad de archivos que incluyen `".conf"` en su nombre.

Esto se puede usar para construir cadenas de comandos subsiguientes siempre que sea necesario:

```
ls -l | grep ".conf" | grep -c .
```

transferir la salida del cmd de la raíz al archivo de usuario

A menudo, uno quiere mostrar el resultado de un comando ejecutado por la raíz a otros usuarios. El comando **tee** permite escribir fácilmente un archivo con permisos de usuario desde un comando que se ejecuta como root:

```
su -c ifconfig | tee ~/results-of-ifconfig.txt
```

Sólo **ifconfig** se ejecuta como root.

encadenamiento lógico de comandos con **&&** y **||**

&& encadena dos comandos. El segundo solo se ejecuta si el primero sale con éxito. **||** encadena dos órdenes. Pero el segundo solo se ejecuta si el primero sale con un error.

```
[ a = b ] && echo "yes" || echo "no"
```

```
# if you want to run more commands within a logical chain, use curly braces
```

```
# which designate a block of commands
# They do need a ; before closing bracket so bash can differentiate from other uses
# of curly braces
[ 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." ; }
# mind the use of line continuation sign \
# only needed to chain yes block with || ....
```

Encadenamiento en serie de comandos con punto y coma.

Un punto y coma separa solo dos comandos.

```
echo "i am first" ; echo "i am second" ; echo " i am third"
```

comandos de encadenamiento con |

El | toma la salida del comando izquierdo y la canaliza como entrada el comando derecho. Tenga en cuenta que esto se hace en una subshell. Por lo tanto, no puede establecer valores de vars del proceso de llamada con una canalización.

```
find . -type f -a -iname '*.mp3' | \
while read filename; do
    mute --noise "$filename"
done
```

Lea Cadena de comandos y operaciones. en línea:

<https://riptutorial.com/es/bash/topic/5589/cadena-de-comandos-y-operaciones->

Capítulo 12: Cambiar shell

Sintaxis

- `echo $0`
- `ps -p $$`
- `echo $SHELL`
- `exportar SHELL = / bin / bash`
- `exec / bin / bash`
- `cat / etc / shells`

Examples

Encuentra el shell actual

Hay algunas maneras de determinar el shell actual

```
echo $0
ps -p $$
echo $SHELL
```

Cambiar la concha

Para cambiar el bash actual ejecuta estos comandos.

```
export SHELL=/bin/bash
exec /bin/bash
```

para cambiar el bash que se abre en el inicio, edite el `.profile` y agregue esas líneas

Listar conchas disponibles

Para listar los shells de inicio de sesión disponibles:

```
cat /etc/shells
```

Ejemplo:

```
$ cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/dash
/bin/bash
/bin/rbash
```

Lea Cambiar shell en línea: <https://riptutorial.com/es/bash/topic/3951/cambiar-shell>

Capítulo 13: Citando

Sintaxis

- `\C` (cualquier caracter excepto una nueva línea)
- 'todo literal excepto comillas simples'; 'esto:' `\` " es una comilla simple '
- `$` 'solo `\\` y `\'` son especiales; `\n` = nueva línea, etc. '
- "`$` variable y otro texto; `\`" `\\` `$` `\`` son especiales"

Examples

Nuevas líneas y personajes de control.

Se puede incluir una nueva línea en una cadena de una sola cadena o entre comillas dobles. Tenga en cuenta que backslash-newline 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 de comillas en dólares, backslash-letter o backslash-octal se pueden usar 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'
```

Comillas dobles para sustitución de variables y comandos.

Las sustituciones de variables solo deben usarse entre comillas dobles.

```
calculation='2 * 3'
echo "$calculation"          # prints 2 * 3
echo $calculation            # prints 2, the list of files in the current directory, and 3
echo "$(($calculation))"     # prints 6
```

Fuera de las comillas dobles, `$var` toma el valor de `var`, lo divide en partes delimitadas por espacios en blanco e interpreta cada parte como un patrón global (comodín). A menos que desee este comportamiento, siempre ponga `$var` entre comillas dobles: `"$var"`.

Lo mismo se aplica a las sustituciones de comandos: "\$ (mycommand) " es la salida de mycommand , \$(mycommand) es el resultado de split + glob en la salida.

```
echo "$var"           # good
echo "$(mycommand)"   # good
another=$var          # also works, assignment is implicitly double-quoted
make -D THING=$var     # BAD! This is not a bash assignment.
make -D THING="$var"   # good
make -D "THING=$var"   # also good
```

Las sustituciones de comandos tienen sus propios contextos de cita. Escribiendo sustituciones arbitrariamente anidados es fácil porque el analizador hará un seguimiento de la profundidad de anidación en lugar de avidez en busca de la primera " carácter El resaltador de sintaxis Stackoverflow analiza este mal, sin embargo, por ejemplo..:

```
echo "formatted text: $(printf "a + b = %04d" "${c}")" # "formatted text: a + b = 0000"
```

Los argumentos variables para una sustitución de comando también deben incluirse entre comillas dobles dentro de las expansiones:

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

Citando texto literal

Todos los ejemplos en este párrafo imprimen la línea.

```
!"#$%&'()*+,-./:;<=>? @[\]^_`{|}~
```

Una barra invertida cita el siguiente carácter, es decir, el siguiente carácter se interpreta literalmente. La única excepción es una nueva línea: backslash-newline se expande a la cadena vacía.

```
echo \!"#$%&'()*+,-./:;<=>? @[\]^_`{|}~
```

Todo el texto entre comillas simples (citas hacia adelante ' , también conocido como apóstrofe) se imprime literalmente. Incluso la barra invertida significa por sí misma, y es imposible incluir una sola cita; en su lugar, puede detener la cadena literal, incluir una comilla simple literal con una barra diagonal inversa y volver a iniciar la cadena literal. Por lo tanto, la secuencia de 4 caracteres '\'' permite efectivamente incluir una comilla simple en una cadena literal.

```
echo '!"#$%&'\'()*+,-./:;<=>? @[\]^_`{|}~'
#          ^^^^
```

Dollar-single-quote inicia una cadena literal '\$'...' como muchos otros lenguajes de programación, donde la barra invertida cita el siguiente carácter.

```
echo $'!"#$%&'\'()*+,-./:;<=>? @[\]^_`{|}~'
#          ^^          ^^
```

Las comillas dobles " delimitan cadenas semi-literales donde solo los caracteres " \ \$ y ` conservan su significado especial. Estos caracteres necesitan una barra invertida delante de ellos (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 "!\"#$%&'()*+,-./:;<=>? @[\]^_`{|}~"
#      ^^          ^  ^^      \[ prints \[
```

¡Interactivamente, ten cuidado con eso ! desencadena la expansión del historial entre comillas dobles: "!oops" busca un comando antiguo que contenga oops ; "\!oops" no hace expansión de historial pero mantiene la barra invertida. Esto no sucede en los scripts.

Diferencia entre comillas dobles y comillas simples

Doble cita	Una frase
Permite expansión variable.	Previene la expansión variable.
Permite la expansión del historial si está habilitado.	Previene la expansión de la historia.
Permite la sustitución de comandos.	Previene la sustitución de comandos
* y @ pueden tener un significado especial	* y @ son siempre literales
Puede contener tanto comillas simples como comillas dobles	No se permite comillas simples dentro de comillas simples
\$, `, ", \ puede escaparse con \ para evitar su significado especial	Todos ellos son literales.

Propiedades que son comunes a ambos:

- Previene el globbing
- Previene la división de palabras

Ejemplos:

```
$ echo "!cat"
echo "cat file"
cat file
$ echo '!cat'
!cat
echo "\"'\\"
""
$ a='var'
$ echo '$a'
```

```
$a  
$ echo "$a"  
var
```

Lea Citando en línea: <https://riptutorial.com/es/bash/topic/729/citando>

Capítulo 14: Comando de corte

Introducción

En Bash, el comando de `cut` es útil para dividir un archivo en varias partes más pequeñas.

Sintaxis

- cortar archivo [opción]

Parámetros

Opción	Descripción
<code>-b LIST , --bytes=LIST</code>	Imprima los bytes listados en el parámetro LIST
<code>-c LIST , --characters=LIST</code>	Imprimir caracteres en posiciones especificadas en el parámetro LIST
<code>-f LIST , --fields=LIST</code>	Imprimir campos o columnas
<code>-d DELIMITER</code>	Se utiliza para separar columnas o campos

Examples

Mostrar la primera columna de un archivo.

Supongamos que tienes un archivo como este

```
John Smith 31
Robert Jones 27
...
```

Este archivo tiene 3 columnas separadas por espacios. Para seleccionar solo la primera columna, haga lo siguiente.

```
cut -d ' ' -f1 filename
```

Aquí el `-d` , especifica el delimitador, o lo que separa los registros. La bandera `-f` especifica el campo o número de columna. Esto mostrará la siguiente salida

```
John
Robert
...
```


Mostrar las columnas x a y de un archivo.

A veces, es útil mostrar un rango de columnas en un archivo. Supongamos que tienes este archivo

```
Apple California 2017 1.00 47  
Mango Oregon 2015 2.30 33
```

Para seleccionar las 3 primeras columnas haz

```
cut -d ' ' -f1-3 filename
```

Esto mostrará la siguiente salida

```
Apple California 2017  
Mango Oregon 2015
```

Lea Comando de corte en línea: <https://riptutorial.com/es/bash/topic/9138/comando-de-corte>

Capítulo 15: Control de trabajo

Sintaxis

- long_cmd &
- trabajos
- fg% JOB_ID
- fg%? PATRON
- fg% JOB_ID

Examples

Ejecutar comando en segundo plano

```
$ sleep 500 &  
[1] 7582
```

Pone el comando sleep en segundo plano. 7582 es el identificador de proceso del proceso en segundo plano.

Listar procesos de fondo

```
$ jobs  
[1]   Running                  sleep 500 &    (wd: ~)  
[2]-  Running                  sleep 600 &    (wd: ~)  
[3]+  Running                  ./Fritzing &
```

El primer campo muestra los identificadores de trabajo. El signo + y - que sigue a la identificación del trabajo para dos trabajos denota el trabajo predeterminado y el siguiente trabajo predeterminado candidato cuando el trabajo predeterminado actual finaliza respectivamente. El trabajo predeterminado se usa cuando los comandos `fg` o `bg` se usan sin ningún argumento.

El segundo campo da el estado del trabajo. El tercer campo es el comando utilizado para iniciar el proceso.

El último campo (wd: ~) dice que los comandos de suspensión se iniciaron desde el directorio de trabajo ~ (Inicio).

Traer un proceso de fondo al primer plano

```
$ fg %2  
sleep 600
```

% 2 especifica el trabajo no. 2. Si se usa `fg` sin ningún argumento, el último proceso se pone en segundo plano.

```
$ fg %?sle  
sleep 500
```

?sle refiere al comando del proceso background que contiene "sle". Si varios comandos de fondo contienen la cadena, producirá un error.

Detener un proceso de primer plano

Presione Ctrl + Z para detener un proceso de primer plano y ponerlo en segundo plano

```
$ sleep 600  
^Z  
[8]+  Stopped                  sleep 600
```

Reiniciar proceso de fondo detenido

```
$ bg  
[8]+ sleep 600 &
```

Lea Control de trabajo en línea: <https://riptutorial.com/es/bash/topic/5193/control-de-trabajo>

Capítulo 16: Copiando (cp)

Sintaxis

- `cp [opciones] fuente de destino`

Parámetros

Opción	Descripción
<code>-a , --archive</code>	Combina las opciones <code>d</code> , <code>p</code> y <code>r</code>
<code>-b , --backup</code>	Antes de la eliminación, hace una copia de seguridad
<code>-d , --no-deference</code>	Conserva enlaces
<code>-f , --force</code>	Eliminar destinos existentes sin preguntar al usuario
<code>-i , --interactive</code>	Mostrar mensaje antes de sobrescribir
<code>-l , --link</code>	En lugar de copiar, vincular archivos en lugar de
<code>-p , --preserve</code>	Conservar atributos de archivo cuando sea posible
<code>-R , --recursive</code>	Copia recursiva de directorios

Examples

Copia un solo archivo

Copie `foo.txt` desde `/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/` to `/path/to/target/folder/` en un archivo llamado `bar.txt`

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

Copiar carpetas

copiar carpeta `foo` en la `bar` carpetas

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

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

Lea Copiando (cp) en línea: <https://riptutorial.com/es/bash/topic/4030/copiando--cp->

Capítulo 17: co-procesos

Examples

Hola Mundo

```
# create the co-process
coproc bash

# send a command to it (echo a)
echo 'echo Hello World' >&"${COPROC[1]}"

# read a line from its output
read line <&"${COPROC[0]}"

# show the line
echo "$line"
```

La salida es "Hello World".

Lea co-procesos en línea: <https://riptutorial.com/es/bash/topic/6933/co-procesos>

Capítulo 18: Creando directorios

Introducción

Manipulación de directorios desde la línea de comando.

Examples

Mueva todos los archivos que aún no estén en un directorio a un directorio con nombre propio

```
ll | grep ^ - | awk -F "." '{imprimir $ 2 "." $ 3}' | awk -F ":" '{print $ 2}' | awk '{$ 1 = ""; imprimir $ 0}' |  
corte -c2- | awk -F "." '{print "mkdir" "$ 1" "; mv" "$ 1". "$ 2" "" "$ 1" "}'> tmp; source tmp
```

Lea Creando directorios en línea: <https://riptutorial.com/es/bash/topic/8168/creando-directorios>

Capítulo 19: Cuándo usar eval

Introducción

En primer lugar: ¡sabe lo que estás haciendo! En segundo lugar, aunque debe evitar usar `eval`, si su uso hace que el código sea más limpio, adelante.

Examples

Usando 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` u `getopts` para establecer `$@` en la salida de los analizadores de opciones mencionados anteriormente. Sin embargo, también puede usarlo para crear una función `pop` simple que puede operar en las variables de forma silenciosa y directamente sin tener que almacenar el resultado para la variable original:

```
isnum()
{
    # is argument an integer?
    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 # let's check to see if this is a variable or just a bare array
    var="$1"
    isvar=1
    arr=($(eval eval -- echo -n "\${$1[@]}")) # if it is a var, get its contents
else
    arr=($@)
fi

# we need to reverse the contents of $@ so that we can shift
# the last element into nothingness
arr=($(awk <<<"${arr[@]}" '{ for (i=NF; i>1; --i) printf("%s ",$i); print $1; }'

# set $@ to ${arr[@]} so that we can run shift against it.
eval set -- "${arr[@]}"

shift # remove the last element

# put the array back to its original order
arr=($(awk <<<"$@" '{ for (i=NF; i>1; --i) printf("%s ",$i); print $1; }'

# echo the contents for the benefit of users and for bare arrays
echo "${arr[@]}"

if ((isvar)); then
    # set the contents of the original var to the new modified array
    eval -- "$var=(${arr[@]})"
fi
}

```

Usando Eval con Getopt

Si bien es posible que eval no sea necesario para una función de estilo `pop`, sin embargo, se requiere cada vez que use `getopt`:

Considere la siguiente función que acepta `-h` como una 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
                ;;
        esac
    done
}

```

```
done  
  
echo "$@"  
}
```

Sin el `set -- "$argv" eval set -- "$argv"` genera `-h --` lugar del deseado `(-h --)` y luego ingresa en un bucle infinito porque `-h --` no coincide `--` o `-h` .

Lea **Cuándo usar eval en línea**: <https://riptutorial.com/es/bash/topic/10113/cuando-usar-eval>

Capítulo 20: Declaración del caso

Examples

Declaración de un caso simple

En su forma más simple admitida por todas las versiones de bash, la declaración de caso ejecuta el caso que coincide con el patrón. ;; El operador se rompe después del primer partido, si lo hay.

```
#!/bin/bash

var=1
case $var in
1)
    echo "Antartica"
    ;;
2)
    echo "Brazil"
    ;;
3)
    echo "Cat"
    ;;
esac
```

Salidas:

```
Antartica
```

Declaración del caso con caída a través

4.0

Desde bash 4.0, se introdujo un nuevo operador ;& que proporciona un mecanismo de [caída](#) .

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

```
var=1
case $var in
1)
    echo "Antartica"
    ;&
2)
    echo "Brazil"
    ;&
3)
    echo "Cat"
    ;&
esac
```

Salidas:

```
Antartica
Brazil
Cat
```

Caerse solo si los patrones subsiguientes coinciden

4.0

Desde Bash 4.0, se introdujo otro operador `;;&` que también [se aplica solo si](#) los patrones en la (s) posterior (es) declaración (es) de caso coinciden.

```
#!/bin/bash

var=abc
case $var in
a*)
    echo "Antartica"
    ;;&
xyz)
    echo "Brazil"
    ;;&
*b*)
    echo "Cat"
    ;;&
esac
```

Salidas:

```
Antartica
Cat
```

En el siguiente ejemplo, el `abc` coincide con el primer y el tercer caso, pero no con el segundo. Entonces, el segundo caso no se ejecuta.

Lea Declaración del caso en línea: <https://riptutorial.com/es/bash/topic/5237/declaracion-del-caso>

Capítulo 21: Decodificar URL

Examples

Ejemplo simple

URL codificada

http% 3A% 2F% 2Fwww.foo.com% 2Findex.php% 3Fid% 3Dqwerty

Usa este comando para decodificar la URL

```
echo "http%3A%2F%2Fwww.foo.com%2Findex.php%3Fid%3Dqwerty" | sed -e "s/%\([0-9A-F][0-9A-F]\)/\\\\x\\1/g" | xargs -0 echo -e
```

URL decodificada (resultado del comando)

<http://www.foo.com/index.php?id=qwerty>

Usando printf para decodificar una cadena

```
#!/bin/bash

$ string='Question%20-%20%22how%20do%20I%20decode%20a%20percent%20encoded%20string%3F%22%0AAnswer%20%20%20-%20Use%20printf%20%3A)'
$ printf '%b\n' "${string//%/\\x}"

# the result
Question - "how do I decode a percent encoded string?"
Answer   - Use printf :)
```

Lea Decodificar URL en línea: <https://riptutorial.com/es/bash/topic/10895/decodificar-url>

Capítulo 22: Depuración

Examples

Depurando un script bash con "-x"

Use "-x" para habilitar la salida de depuración de las líneas ejecutadas. Puede ejecutarse en una sesión o script completo, o habilitarse programáticamente dentro de un script.

Ejecutar un script con salida de depuración habilitada:

```
$ bash -x myscript.sh
```

O

```
$ bash --debug myscript.sh
```

Activa la depuración dentro de un script bash. Opcionalmente, se puede volver a activar, aunque la salida de depuración se restablece automáticamente cuando sale el script.

```
#!/bin/bash
set -x    # Enable debugging
# some code here
set +x    # Disable debugging output.
```

Comprobando la sintaxis de un script con "-n"

La marca -n le permite verificar 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
```

Depurando usigh bashdb

Bashdb es una utilidad similar a gdb, ya que puede hacer cosas como establecer puntos de interrupción en una línea o en una función, imprimir contenido de variables, puede reiniciar la ejecución de scripts y más.

Normalmente puede instalarlo a través de su administrador de paquetes, por ejemplo en Fedora:

```
sudo dnf install bashdb
```

O conseguirlo desde la [página de inicio](#) . Luego puedes ejecutarlo con tu script como parámetro:

```
bashdb <YOUR SCRIPT>
```

Aquí hay algunos comandos para comenzar:

```
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 <line num> set breakpoint on some line
c - continue till some breakpoint
i b - info on break points
d <line #> - delete breakpoint at line #

shell - launch a sub-shell in the middle of execution, this is handy for manipulating
variables
```

Para obtener 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/>

Lea Depuración en línea: <https://riptutorial.com/es/bash/topic/3655/depuracion>

Capítulo 23: Dividir archivos

Introducción

A veces es útil dividir un archivo en varios archivos separados. Si tiene archivos grandes, puede ser una buena idea dividirlo en trozos más pequeños

Examples

Dividir un archivo

Ejecutar 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 están prefijados con la letra `x` por defecto. Si el archivo inicial tuviera menos de 1000 líneas, solo se crearía uno de esos archivos.

Para cambiar el prefijo, agregue el prefijo que desee al final de la línea de comando

```
split file customprefix
```

Ahora se crean archivos llamados `customprefixaa`, `customprefixab`, `customprefixac`, etc.

Para especificar el número de líneas a generar por archivo, use 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 usando las opciones `-b` o `--bytes`. Por ejemplo, para permitir un máximo de 1MB.

```
split --bytes=1MB file
```

Podemos usar `sed` con la opción `w` para dividir un archivo en varios archivos. Los archivos se pueden dividir especificando la dirección de línea o el patrón.

Supongamos que tenemos este archivo fuente que nos gustaría dividir:


```
cat -n sourcefile
```

```
1 En el Ning Nang Nong
2 donde las vacas van bong!
3 y todos los monos dicen BOO!
4 Hay un Nong Nang Ning
5 donde van los arboles ping!
6 Y las teteras jibber jabber joo.
7 En el Nong Ning Nang
```

Comando para dividir el archivo por número de línea:

```
sed '1,3w f1
> 4,7w f2' sourcefile
```

Esto escribe la línea 1 a la línea 3 en el archivo f1 y la línea 4 a la línea 7 en el archivo f2, desde el archivo fuente.

```
cat -n f1
```

```
1 En el Ning Nang Nong
2 donde las vacas van bong!
3 y todos los monos dicen BOO!
```

```
cat -n f2
```

```
1 Hay un Nong Nang Ning
2 donde van los arboles ping!
3 Y las teteras jibber jabber joo.
4 En el Nong Ning Nang
```

Comando para dividir el archivo por contexto / patrón:

```
sed '/Ning/w file1
> /Ping/w file2' sourcefile
```

Esto divide el archivo de origen en archivo1 y archivo2. file1 contiene todas las líneas que coinciden con Ning, file2 contiene líneas que coinciden con Ping.

```
cat file1
```

```
En el Ning Nang Nong
Hay un Nong Nang Ning
En el Nong Ning Nang
```

```
cat file2
```

Donde van los arboles ping!

Lea Dividir archivos en línea: <https://riptutorial.com/es/bash/topic/9151/dividir-archivos>

Capítulo 24: División de palabras

Sintaxis

- Establezca IFS en nueva línea: IFS = \$ '\n'
- Establecer IFS en cadena nula: IFS =
- Establezca IFS en / carácter: IFS = /

Parámetros

Parámetro	Detalles
IFS	Separador de campo interno
-X	Imprimir comandos y sus argumentos a medida que se ejecutan (opción de Shell)

Observaciones

- La división de palabras no se realiza durante las tareas, por ejemplo, `newvar=$var`
- La división de palabras no se realiza en la construcción `[[...]]`
- Use comillas dobles en las variables para evitar la división de palabras

Examples

Partiendo con IFS

Para ser más claros, vamos a crear un script llamado `showarg` :

```
#!/usr/bin/env bash
printf "%d args:" $#
printf " <%s>" "$@"
echo
```

Ahora veamos las diferencias:

```
$ var="This is an example"
$ showarg $var
4 args: <This> <is> <an> <example>
```

`$var` se divide en 4 args. IFS son caracteres de espacios en blanco y, por lo tanto, la división de palabras ocurrió 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 vamos a establecer `IFS=/`

```
$ IFS=/
$ var="This/is/an/example"
$ showarg $var
4 args: <This> <is> <an> <example>
```

El `$var` se divide en 4 argumentos, no un solo argumento.

¿Qué, cuándo y por qué?

Cuando el shell realiza la *expansión de parámetros* , la *sustitución de comandos* , la *expansión variable o aritmética* , busca límites de palabras en el resultado. Si se encuentra un límite de palabra, el resultado se divide en varias palabras en esa posición. El límite de la palabra se define mediante una variable de shell `IFS` (Separador de campo interno). 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 argumentos, solo `I` , `am` y `a` se imprimirán.

IFS y división de palabras

Vea [qué, cuándo y por qué](#), si no sabe acerca de la afiliación de `IFS` a la división de palabras

vamos a configurar el `IFS` en caracteres de espacio solamente:

```
set -x
var='I am
a
multiline string'
```

```
IFS=' '
fun() {
    echo "$1-"
    echo "$2*"
    echo "$3."
}
fun $var
```

Esta vez la división de palabras solo funcionará en espacios. La función de `fun` se ejecutará así:

```
fun I 'am
a
multiline' string
```

`$var` se divide en 3 args. `I`, `am\na\nmultiline` y `string` se imprimirán

Vamos a configurar el IFS a nueva línea solamente:

```
IFS=$'\n'
...
```

Ahora la `fun` será ejecutada como:

```
fun 'I am' a 'multiline string'
```

`$var` se divide en 3 args. `I am`, se imprimirá `a multiline string`

Veamos qué sucede si establecemos IFS en cadena nula:

```
IFS=
...
```

Esta vez la `fun` se ejecutará así:

```
fun 'I am
a
multiline string'
```

`$var` no se divide, es decir, se mantuvo como un solo argumento.

Puede evitar la división de palabras configurando el IFS en cadena nula

Una forma general de evitar la división de palabras es usar comillas dobles:

```
fun "$var"
```

evitará la división de palabras en todos los casos discutidos anteriormente, es decir, la función `fun` se ejecutará con un solo argumento.

Malos efectos 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 solo argumento, ¡¡es más bien 6 argumentos !!

```
$ [ $a = something ] || echo "didn't match"
bash: [: too many arguments
didn't match
```

`[$a = something]` se interpretó como `[I am a string with spaces = something]`

```
$ [ $(grep . file) = 'something' ]
bash: [: too many arguments
```

El comando `grep` devuelve una cadena multilínea con espacios, así que puedes imaginar cuántos argumentos hay ...: D

Vea [qué](#), [cuándo](#) y [por qué](#) para lo básico.

Utilidad de la división de palabras

Hay algunos casos donde la división de palabras puede ser útil:

Rellenando matriz:

```
arr=$(grep -o '[0-9]\+' file)
```

Esto llenará `arr` con todos los valores numéricos encontrados en el *archivo*

Corriendo a través del espacio palabras separadas:

```
words='foo bar baz'
for w in $words;do
    echo "W: $w"
done
```

Salida:

```
W: foo
W: bar
W: baz
```

Pasando parámetros separados por espacios que no contienen espacios en blanco:

```
packs='apache2 php php-mbstring php-mysql'
sudo apt-get install $packs
```

```
packs='
apache2
php
php-mbstring
php-mysql
'
sudo apt-get install $packs
```

Esto instalará los paquetes. Si cita dos veces los `$packs` se producirá un error.

Unquoe `td $packs` está enviando todos los nombres de paquetes separados por espacios como argumentos a `apt-get`, mientras que si lo cita enviará la cadena de `$packs` como un solo argumento y luego `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 básico.

División por cambios de separador.

Simplemente podemos reemplazar los separadores del espacio a una nueva línea, como se muestra en el siguiente ejemplo.

```
echo $sentence | tr " " "\n"
```

Dividirá el valor de la `sentence` variable y lo mostrará línea por línea respectivamente.

Lea [División de palabras en línea](https://riptutorial.com/es/bash/topic/5472/division-de-palabras): <https://riptutorial.com/es/bash/topic/5472/division-de-palabras>

Capítulo 25: El comando de corte

Introducción

El comando de `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 que se encuentra en Linux y la versión FreeBSD que se encuentra en MacOS, pero cada versión de Unix tiene la suya. Vea a continuación las diferencias. Las líneas de entrada se leen desde la `stdin` o desde los archivos enumerados como argumentos en la línea de comando.

Sintaxis

- `cortar -f1,3 #` extraer el primer y tercer **campo** *delimitado por tabuladores* (desde `stdin`)
- `cortar -f1-3 #` extraer *desde el primer hasta el tercer* campo (extremos incluidos)
- `corte -f-3 #` -3 se interpreta como 1-3
- `cut -f2- #` 2- se interpreta *desde el segundo hasta el último*
- `cortar -c1-5,10 #` extraer de `stdin` los **caracteres** en las posiciones 1,2,3,4,5,10
- `cut -s -f1 #` suprimir líneas que no contienen delimitadores
- `corte --complemento -f3 #` (solo corte GNU) extraiga *todos los campos excepto* el tercero

Parámetros

Parámetro	Detalles
-f, - campos	Selección basada en el campo
-d, --delimitador	Delimitador para selección de campo
-c, --caracteres	Selección basada en caracteres, delimitador ignorado o error
-s, - solo delimitado	Suprimir líneas sin caracteres delimitadores (impresos como están).
--complemento	Selección invertida (extraer todos los campos / caracteres <i>excepto los</i> especificados)
- salida-delimitador	Especifique cuándo debe ser diferente del delimitador de entrada.

Observaciones

1. Diferencias de sintaxis.

Las opciones largas en la tabla anterior solo son compatibles con la versión GNU.

2. Ningún personaje recibe tratamiento especial.

El `cut` FreeBSD (que viene con MacOS, por ejemplo) no tiene el interruptor `--complement` y, en el caso de los rangos de caracteres, se puede usar el comando `colrm` lugar:

```
$ cut --complement -c3-5 <<<"123456789"
126789

$ colrm 3 5 <<<"123456789"
126789
```

Sin embargo, hay una gran diferencia, porque `colrm` trata los caracteres TAB (ASCII 9) como tabulaciones reales hasta el siguiente múltiplo de ocho, y los espacios en blanco (ASCII 8) como - 1 de ancho; por el contrario, `cut` trata a todos los caracteres como una columna de ancho.

```
$ colrm 3 8 <<<${'12\tABCDEF'} # Input string has an embedded TAB
12ABCDEF

$ cut --complement -c3-8 <<<${'12\tABCDEF'}
12F
```

3. (Todavía no) Internacionalización

Cuando se diseñó el `cut`, todos los caracteres tenían un byte de largo y la internacionalización no era un problema. Cuando los sistemas de escritura con caracteres más anchos se hicieron populares, la solución adoptada por POSIX fue diferenciar entre el viejo conmutador `-c`, que debería conservar su significado de seleccionar *caracteres*, sin importar cuántos bytes de ancho, e introducir un nuevo conmutador `-b` que debería seleccionar *bytes*, independientemente de la codificación de caracteres actual. En las implementaciones más populares, se introdujo `-b` y funciona, pero `-c` sigue funcionando exactamente igual que `-b` y no como debería. Por ejemplo, con `cut` GNU:

Parece que el filtro de correo no deseado de SE incluye listas negras de textos en inglés con caracteres kanji aislados. No pude superar esta limitación, por lo que los siguientes ejemplos son menos expresivos de lo que podrían ser.

```
# In an encoding where each character in the input string is three bytes wide,
# Selecting bytes 1-6 yields the first two characters (correct)
$ LC_ALL=ja_JP.UTF-8 cut -b1-6 kanji.utf-8.txt
...first two characters of each line...

# Selecting all three characters with the -c switch doesn't work.
# It behaves like -b, contrary to documentation.
$ LC_ALL=ja_JP.UTF-8 cut -c1-3 kanji.utf-8.txt
...first character of each line...
```

```
# In this case, an illegal UTF-8 string is produced.
# The -n switch would prevent this, if implemented.
$ LC_ALL=ja_JP.UTF-8 cut -n -c2 kanji.utf-8.txt
...second byte, which is an illegal UTF-8 sequence...
```

Si sus caracteres están fuera del rango ASCII y desea utilizar el `cut`, siempre debe tener en cuenta el ancho de caracteres en su codificación y utilizar `-b` consecuencia. Si y cuando `-c` comienza a funcionar como se documenta, no tendrá que cambiar sus scripts.

4. Comparaciones de velocidad.

Las limitaciones del `cut` hacen que las personas duden de su utilidad. De hecho, la misma funcionalidad se puede lograr mediante utilidades más potentes y populares. Sin embargo, la ventaja del `cut` es su *rendimiento*. Vea a continuación para algunas comparaciones de velocidad. `test.txt` tiene tres millones de líneas, con cinco campos separados por espacios. Para la prueba `awk`, se usó `mawk`, porque es más rápido que GNU `awk`. El shell en sí (última línea) es, con mucho, el peor desempeño. Los tiempos dados (en segundos) son lo que el comando de `time` da como *tiempo real*.

(Solo para evitar malentendidos: todos los comandos probados dieron la misma salida con la entrada dada, pero por supuesto no son equivalentes y darían salidas diferentes en diferentes situaciones, en particular si los campos estuvieran delimitados por un número variable de espacios)

Mando	Hora
<code>cut -d ' ' -f1,2 test.txt</code>	1.138s
<code>awk '{print \$1 \$2}' test.txt</code>	1.688s
<code>join -al -o1.1,1.2 test.txt /dev/null</code>	1.767s
<code>perl -lane 'print "@F[1,2]"' test.txt</code>	11.390s
<code>grep -o '^\[^\]*\) \[^\]*\)' test.txt</code>	22.925s
<code>sed -e 's/^\[^\]*\) \[^\]*\).*\$/\1 \2/' test.txt</code>	52.122s
<code>while read ab _; do echo \$a \$b; done <test.txt</code>	55.582s

5. Páginas de manual referenciales

- [Opengrupo](#)
- [ÑU](#)
- [FreeBSD](#)

Examples

Uso básico

El uso típico es con archivos de tipo CSV, donde cada línea consta de campos separados por un delimitador, especificados por la opción `-d`. El delimitador predeterminado es el carácter TAB. Supongamos que tiene un archivo de datos `data.txt` con líneas como

```
0 0 755 1482941948.8024
102 33 4755 1240562224.3205
1003 1 644 1219943831.2367
```

Entonces

```
# extract the third space-delimited field
$ cut -d ' ' -f3 data.txt
755
4755
644

# extract the second dot-delimited field
$ cut -d. -f2 data.txt
8024
3205
2367

# extract the character range from the 20th through the 25th character
$ cut -c20-25 data.txt
948.80
056222
943831
```

Como de costumbre, puede haber espacios opcionales entre un interruptor y su parámetro: `-d`, es lo mismo que `-d ,`

GNU `cut` permite especificar una opción `--output-delimiter`: (una característica independiente de este ejemplo es que debe escaparse un punto y coma como delimitador de entrada para evitar su tratamiento especial por parte del shell)

```
$ cut --output-delimiter=, -d\; -f1,2 <<<"a;b;c;d"
a,b
```

Sólo un personaje delimitador.

No puede tener más de un delimitador: si especifica algo como `-d ",;:"`, algunas implementaciones usarán solo el primer carácter como delimitador (en este caso, la coma). Otras implementaciones (por ejemplo, `cut` GNU) 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.
```

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 delimitadas por el espacio podría ser menos obvio para algunos

```
$ cut -d ' ' -f1,3 <<<"a b c d e"  
a b
```

`cut` no se puede usar para analizar argumentos como lo hace el shell y otros programas.

Sin citar

No hay forma de proteger el delimitador. Las hojas de cálculo y el software de manejo de CSV similar generalmente pueden reconocer un carácter de comillas de texto que hace posible definir cadenas que contienen un delimitador. Con el `cut` no puedes.

```
$ cut -d, -f3 <<<'John,Smith,"1, Main Street"  
"1
```

Extraer, no manipular.

Solo puede extraer partes de líneas, no reordenar o repetir campos.

```
$ cut -d, -f2,1 <<<'John,Smith,USA' ## Just like -f1,2  
John,Smith  
$ cut -d, -f2,2 <<<'John,Smith,USA' ## Just like -f2  
Smith
```

Lea El comando de corte en línea: <https://riptutorial.com/es/bash/topic/8762/el-comando-de-corte>

Capítulo 26: Encontrar

Introducción

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

encontrar search_path selection_criteria acción

Sintaxis

- encontrar [-H] [-L] [-P] [-D debugopts] [-Olevel] [ruta ...] [expresión]

Examples

Buscando un archivo por nombre o extensión

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

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

Para encontrar archivos / directorios con una extensión específica, use un comodín:

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

Para encontrar archivos / directorios que coincidan con una de las muchas extensiones, use el indicador `or` :

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

Para buscar archivos / directorios cuyo nombre comience con abc y termine con un carácter alfa siguiendo un dígito:

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

Para encontrar todos los archivos / directorios ubicados en un directorio específico

```
$ find /opt
```

Para buscar solo archivos (no directorios), use `-type f` :

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

Para buscar solo directorios (no archivos normales), use `-type d`:

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

Buscando archivos por tipo

Para encontrar archivos, use la `-type f`

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

Para encontrar directorios, use la `-type d`

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

Para encontrar dispositivos de bloque, use la `-type b`

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

Para encontrar enlaces simbólicos, use la bandera `-type l`

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

Ejecutando comandos contra 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 buscará recursivamente todos los directorios (`-type d`) relativos a `.` (que es su directorio de trabajo actual), y ejecute `chmod 770` en ellos. La opción `-r` especifica a `xargs` para no ejecutar `chmod` si `find` no encontró ningún archivo.

Si los nombres de sus archivos o directorios tienen un carácter de espacio en ellos, este comando puede bloquearse; una solución es usar lo siguiente

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

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

La forma preferida de hacerlo es omitir el comando `xargs` y dejar que `find` llame al subprocesso en sí:

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

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

Alternativamente, puede pasar todos los nombres de archivos a una *so/a* llamada de `chmod`, usando

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

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

Una tercera opción es dejar que `bash` se desplace por la lista de nombres de archivos para `find` resultados:

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

Esto es sintácticamente el más torpe, pero conveniente cuando quiere ejecutar múltiples comandos en cada archivo encontrado. Sin embargo, esto **no** es **seguro** en vista de los nombres de archivos con nombres impares.

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

que reemplazará todos los espacios en los nombres de archivo con guiones bajos. (Este ejemplo tampoco funcionará si hay espacios en los nombres de *directorio* principales).

El problema con lo anterior es que, `while read -r` espera una entrada por línea, los nombres de los archivos pueden contener nuevas líneas (y también, la `read -r` perderá cualquier espacio en blanco al final). Puedes arreglar esto cambiando las cosas:

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

De esta manera, el `-exec` recibe los nombres de los archivos en una forma que es completamente correcta y portátil; `bash -c` recibe como una serie de argumentos, que se encontrarán en `$@`, correctamente citados, etc. (El script deberá manejar estos nombres correctamente, por supuesto; cada variable que contenga un nombre de archivo debe estar en doble citas.)

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

Encontrar el archivo por tiempo de acceso / modificación

En un sistema de archivos `ext`, cada archivo tiene asociado un `stat myFile.txt` Acceso, Modificación y Cambio (Estado), para ver esta información puede usar `stat myFile.txt`; utilizando indicadores dentro de `find`, podemos buscar archivos que fueron modificados dentro de un cierto rango de tiempo.

Para buscar archivos que *han* 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
```

El ejemplo anterior se busca sólo en la hora de *modificación* - para buscar en **un** momento cceso, o **c** colgado veces, utilizar **a** o **c** en consecuencia.

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

Formato general:

`-mmin n` : el archivo fue modificado hace *n* minutos
`-mmin -n` : el archivo se modificó hace menos de *n* minutos
`-mmin +n` : el archivo se modificó hace más de *n* minutos

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

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

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

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

Use `-atime` y `-ctime` para el tiempo de acceso y el tiempo de cambio de estado, respectivamente.

Formato general:

`-mtime n` : el archivo fue modificado *nx24* horas atrás
`-mtime -n` : el archivo se modificó hace menos de *24* horas
`-mtime +n` : el archivo se modificó hace más de *24* horas

Encuentre archivos modificados en un **rango de fechas** , del 2007-06-07 al 2007-06-08:

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

Encuentre los archivos a los que se accede en un **rango de marcas de tiempo** (usando 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` podría 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 cuando* queremos comparar la marca de tiempo especificada (acceso, modificación, cambio) o una *cadena que describe un tiempo absoluto*.

Buscando archivos por extensión específica

Para encontrar todos los archivos de una cierta extensión dentro de la ruta actual, puede usar la siguiente sintaxis de `find`. Funciona haciendo uso de `bash`'s construcción `glob` integrada de `bash`'s para que coincida con todos los nombres que tienen la `.extension`.

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

Para encontrar todos los archivos de tipo `.txt` desde el directorio actual solo, haga

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

Encontrar archivos de acuerdo al tamaño.

Encuentra archivos de más de 15MB:

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

Encuentra archivos de menos de 12KB:

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

Encuentra archivos exactamente del tamaño de 12KB:

```
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]
```

Encuentre archivos de tamaño n-block, donde + n significa más que n-block, -n significa menos que n-block y n (sin ningún signo) significa exactamente n-block

Tamaño de 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

Filtrar el camino

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

Para buscar solo archivos que contengan `log` en cualquier lugar de su ruta (carpeta o nombre):

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

Para buscar solo archivos dentro de una carpeta llamada `log` (en cualquier nivel):

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

Para buscar solo archivos dentro de una carpeta llamada `log` o `data` :

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

Para buscar todos los archivos **excepto** los que se encuentran en una carpeta llamada `bin` :

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

Para buscar todos los archivos, todos los archivos **excepto** los que se encuentran en una carpeta llamada `bin` o archivos de registro:

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

Lea Encontrar en línea: <https://riptutorial.com/es/bash/topic/566/encontrar>

Capítulo 27: Escollos

Examples

Espacio en blanco al asignar variables

El espacio en blanco importa cuando se asignan variables.

```
foo = 'bar' # incorrect
foo= 'bar'  # incorrect
foo='bar'   # correct
```

Los dos primeros darán como resultado errores de sintaxis (o peor, la ejecución de un comando incorrecto). El último ejemplo establecerá correctamente la variable `$foo` en el texto "barra".

Falta la última línea en un archivo

El estándar de C dice que los archivos 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. Como 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 asegurarse de que esto funcione correctamente en el ejemplo anterior, agregue una prueba para que continúe el ciclo si la última línea no está vacía.

```
$ while read line || [ -n "$line" ] ; do echo "line $line" ; done < file.txt
one
two
three
```

Los comandos fallidos no detienen la ejecución del script

En la mayoría de los lenguajes de script, si falla una llamada de función, puede lanzar una excepción y detener la ejecución del programa. Los comandos de Bash no tienen excepciones, pero sí tienen códigos de salida. Sin embargo, un código de salida distinto de cero indica un error, un código de salida distinto de cero no detendrá la ejecución del programa.

Esto puede llevar a situaciones peligrosas (aunque se admiten) como las siguientes:

```
#!/bin/bash
cd ~/non/existent/directory
rm -rf *
```

Si el `cd` -ing a este directorio falla, Bash ignorará el error y pasará al siguiente comando, limpiando el directorio desde donde ejecutó el script.

La mejor manera de lidiar con este problema es hacer uso del comando `set` :

```
#!/bin/bash
set -e
cd ~/non/existent/directory
rm -rf *
```

`set -e` le dice a Bash que salga de la secuencia de comandos inmediatamente si algún comando devuelve un estado distinto de cero.

Lea Escollos en línea: <https://riptutorial.com/es/bash/topic/3656/escollos>

Capítulo 28: Escribiendo variables

Examples

declarar variables de tipo débil

declare es un comando interno de bash. (El comando interno usa la **ayuda** para mostrar "manpage"). Se utiliza para mostrar y definir variables o mostrar cuerpos de funciones.

Sintaxis: **declare [opciones] [nombre [= valor]] ...**

```
# options are used to define
# an integer
declare -i myInteger
declare -i anotherInt=10
# an array with values
declare -a anArray=( one two three)
# an assoc Array
declare -A assocArray=( [element1]="something" [second]=anotherthing )
# note that bash recognizes the string context within []

# some modifiers exist
# uppercase content
declare -u big='this will be uppercase'
# same for lower case
declare -l small='THIS WILL BE LOWERCASE'

# readonly array
declare -ra constarray=( eternal true and unchangeable )

# export integer to environment
declare -xi importantInt=42
```

Puedes usar también el + que quita el atributo dado. Sobre todo inútil, solo para estar completo.

Para mostrar variables y / o funciones hay algunas opciones también.

```
# printing defined vars and functions
declare -f
# restrict output to functions only
declare -F # if debugging prints line number and filename defined in too
```

Lea Escribiendo variables en línea: <https://riptutorial.com/es/bash/topic/7195/escribiendo-variables>

Capítulo 29: Espacio de nombres

Examples

No hay cosas tales como espacios de nombres

```
myfunc() {  
    echo "I will never be executed."  
}  
another_func() {  
    # this "redeclare" overwrites original function  
    myfunc(){ echo "I am the one and only"; }  
}  
# myfunc will print "I will never be executed"  
myfunc  
# but if we call another_func first  
another_func  
# it gets overwritten and  
myfunc  
# no prints "I am the one and only"
```

La última declaración gana. ¡No hay cosas tales como espacios de nombres! Sin embargo, las funciones pueden contener otras funciones.

Lea Espacio de nombres en línea: <https://riptutorial.com/es/bash/topic/6835/espacio-de-nombres>

Capítulo 30: Estructuras de Control

Sintaxis

- `["$ 1" = "$ 2"] #A "["` el corchete es en realidad un comando. Por eso requiere un espacio antes y después.
- `test "$ 1" = "$ 2" #Test` es un sinónimo para el comando `"["`

Parámetros

Parámetro a [o prueba	Detalles
Operadores de archivos	Detalles
<code>-e "\$file"</code>	Devuelve true si el archivo existe.
<code>-d "\$file"</code>	Devuelve true si el archivo existe y es un directorio.
<code>-f "\$file"</code>	Devuelve verdadero si el archivo existe y es un archivo normal
<code>-h "\$file"</code>	Devuelve true si el archivo existe y es un enlace simbólico
Comparadores de cuerdas	Detalles
<code>-z "\$str"</code>	Verdad si la longitud de la cadena es cero
<code>-n "\$str"</code>	Verdad si la longitud de la cadena no es cero
<code>"\$str" = "\$str2"</code>	Verdad si la cadena \$ str es igual a la cadena \$ str2. No es lo mejor para los enteros. Puede funcionar pero será inconsistente.
<code>"\$str" != "\$str2"</code>	Cierto si las cuerdas no son iguales
Comparadores de enteros	Detalles
<code>"\$int1" -eq "\$int2"</code>	Verdad si los enteros son iguales
<code>"\$int1" -ne "\$int2"</code>	Verdad si los enteros no son iguales
<code>"\$int1" -gt "\$int2"</code>	Verdad si int1 es mayor que int 2
<code>"\$int1" -ge "\$int2"</code>	Verdadero si int1 es mayor o igual que int2

Parámetro a [o prueba	Detalles
"\$int1" -lt "\$int2"	Verdad si int1 es menor que int 2
"\$int1" -le "\$int2"	Verdad si int1 es menor o igual que int2

Observaciones

Hay muchos parámetros de comparación disponibles en bash. No todos están listados aquí.

Examples

Si declaración

```
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
```

El cierre de `fi` es necesario, pero se pueden omitir las cláusulas `elif` y / o `else`.

Los puntos y coma antes de `then` son sintaxis estándar para combinar dos comandos en una sola línea; solo se pueden omitir si `then` se mueve a la siguiente línea.

Es importante entender que los corchetes `[[` no son parte de la sintaxis, pero se tratan como un comando; es el código de salida de este comando que se está probando. Por lo tanto, siempre debe 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 declaración 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 entre 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 `if` declaraciones con corchetes simples. Estos se definen en el

estándar POSIX y se garantiza que funcionarán en todas las carcasas compatibles con POSIX, incluyendo 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
```

Mientras bucle

```
#!/bin/bash

i=0

while [ $i -lt 5 ] #While i is less than 5
do
    echo "i is currently $i"
    i=$((i+1)) #Not the lack of spaces around the brackets. This makes it a not a test
expression
done #ends the loop
```

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

Este bucle produce:

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

En bucle

```
#!/bin/bash

for i in 1 "test" 3; do #Each space separated statement is assigned to i
    echo $i
done
```

Otros comandos pueden generar sentencias para pasar. Consulte el ejemplo de "Uso de For Loop para iterar sobre números".

Esto produce:

```
1
test
3
```

Usando For Loop para enumerar Iterar sobre números

```
#!/bin/bash

for i in {1..10}; do # {1..10} expands to "1 2 3 4 5 6 7 8 9 10"
    echo $i
done
```

Esto da como resultado lo siguiente:

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

Para bucle con sintaxis estilo C

El formato básico de estilo C `for` bucle es:

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

Notas:

- La asignación de la variable dentro de C-style `for` bucle puede contener espacios a diferencia de la asignación habitual
- Las variables dentro del estilo C `for` bucle no están precedidas por `$`.

Ejemplo:

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

También podemos procesar múltiples variables dentro de C-style `for` loop:

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

Hasta Loop

Hasta que el bucle se ejecute hasta que la condición sea verdadera

```
i=5
until [[ i -eq 10 ]]; do #Checks if i=10
    echo "i=$i" #Print the value of i
    i=$((i+1)) #Increment i by 1
done
```

Salida:

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

Cuando `i` a 10, la condición en hasta que el bucle se vuelve verdadero y el bucle termina.

continuar y romper

Ejemplo para continuar

```
for i in [series]
do
    command 1
    command 2
    if (condition) # Condition to jump over command 3
        continue # skip to the next value in "series"
    fi
    command 3
done
```

Ejemplo para romper

```
for i in [series]
do
    command 4
    if (condition) # Condition to break the loop
    then
        command 5 # Command if the loop needs to be broken
        break
    fi
    command 6 # Command to run if the "condition" is never true
done
```

Bucle sobre una matriz

for bucle:

```
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
```

while **bucle:**

```
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
```

Ruptura de 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 un solo bucle:

```
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
```

Cambiar declaración con el caso

Con la declaración del `case` puede hacer coincidir los valores con una variable.

El argumento pasado al `case` se expande e intenta hacer coincidir con cada patrón.

Si se encuentra una coincidencia, los comandos hasta `;;` son ejecutados.

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

Los patrones no son expresiones regulares sino que coinciden con los patrones de shell (también conocido como globs).

Para Loop sin un 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 otras palabras, el ejemplo anterior es equivalente a este código:

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

En otras palabras, si te encuentras escribiendo `for i in "$@"; do ...; done`, acaba de caer el `in` parte, y escribir simplemente `for i; do ...; done`

Ejecución condicional de listas de comandos

Cómo utilizar la ejecución condicional de las listas de comandos.

Cualquier comando, expresión o función incorporada, así como cualquier comando externo o secuencia de comandos, se pueden ejecutar condicionalmente usando `&&` (*and*) y `||` (*o*) operadores.

Por ejemplo, esto solo imprimirá el directorio actual si el comando `cd` fue exitoso.

```
cd my_directory && pwd
```

Del mismo modo, esto se cerrará si el comando `cd` falla, previniendo una catástrofe:

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

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

Por lo tanto, esta declaración funcionará como se espera ...

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

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

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

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

- Si el `cd` falla, se omite el `&& ls` y el `|| echo ...` se ejecuta.
- Si el `cd` tiene éxito, se ejecuta `&& ls`.
 - Si la `ls` tiene éxito, la `|| echo ...` se ignora (*hasta ahora tan bueno*)
 - **PERO ... si la `ls` falla, el `|| echo ...` también será ejecutado.**

Es el `ls`, no el `cd`, que es el comando anterior.

¿Por qué usar la ejecución condicional de las listas de comandos?

La ejecución condicional es mucho más rápida que `if...then` pero su principal ventaja es permitir que las funciones y los scripts salgan antes, o "cortocircuito".

A diferencia de muchos lenguajes como `c` donde la memoria se asigna explícitamente para estructuras y variables y similares (y, por lo tanto, debe ser desasignado), `bash` maneja esto bajo las coberturas. En la mayoría de los casos, no tenemos que limpiar nada antes de abandonar la función. Una declaración de `return` desasignará todo lo local a la función y la ejecución de recolección en la dirección de devolución en la pila.

Regresar de las funciones o salir de los scripts lo antes posible puede mejorar significativamente

el rendimiento y reducir la carga del sistema al evitar la ejecución innecesaria de código. Por ejemplo...

```
my_function () {  
  
    ### ALWAYS CHECK THE RETURN CODE  
  
    # one argument required. "" evaluates to false(1)  
    [[ "$1" ]] || return 1  
  
    # work with the argument. exit on failure  
    do_something_with "$1" || return 1  
    do_something_else || return 1  
  
    # Success! no failures detected, or we wouldn't be here  
    return 0  
}
```

Lea Estructuras de Control en línea: <https://riptutorial.com/es/bash/topic/420/estructuras-de-control>

Capítulo 31: Evitando la fecha usando printf

Introducción

En Bash 4.2, se introdujo una conversión de tiempo incorporada en el shell para `printf`: la especificación de formato `%(datefmt)T` hace que `printf %(datefmt)T printf` fecha y hora correspondiente a la cadena de formato `datefmt` como se entiende por `strftime` de `strftime`.

Sintaxis

- `printf '%(dateFmt) T' # dateFmt` puede ser cualquier cadena de formato que reconoce `strftime`
- `printf '%(dateFmt) T' -1 # -1` representa la hora actual (predeterminado para ningún argumento)
- `printf '%(dateFmt) T' -2 # -2` representa la hora a la que se invocó el shell

Observaciones

El uso de `printf -v foo '%(...)T'` es idéntico a `foo=$(date +'...')` y guarda una bifurcación para la llamada a la `date` programa externo.

Examples

Obtén la fecha actual

```
$ printf '%(%F)T\n'
2016-08-17
```

Establecer variable a la hora actual

```
$ printf -v now '%(%T)T'
$ echo "$now"
12:42:47
```

Lea Evitando la fecha usando printf en línea: <https://riptutorial.com/es/bash/topic/5522/evitando-la-fecha-usando-printf>

Capítulo 32: Expansión de la abrazadera

Observaciones

[Manual de referencia de Bash: Expansión de llaves](#)

Examples

Crear directorios para agrupar archivos por mes y año.

```
$ mkdir 20{09..11}-{01..12}
```

Al ingresar el comando `ls` se mostrará que se crearon 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
```

Poner un `0` delante de `9` en el ejemplo garantiza que los números se rellenan con un solo `0`. También puede rellena números con múltiples ceros, por ejemplo:

```
$ echo {001..10}
001 002 003 004 005 006 007 008 009 010
```

Crear una copia de seguridad de archivos de puntos.

```
$ cp .vimrc{,.bak}
```

Esto se expande en el comando `cp .vimrc .vimrc.bak`.

Modificar la extensión del nombre de archivo

```
$ mv filename.{jar,zip}
```

Esto se expande en `mv filename.jar filename.zip`.

Usar 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 está limitado a solo números

```
$ for c in {a..z..5}; do echo -n $c; done  
afkpuz
```

Usando la expansión de llaves para crear listas

Bash puede crear fácilmente listas de caracteres alfanuméricos.

```
# list from a to 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  
  
# reverse from z to 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  
  
# digits  
$ echo {1..20}  
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20  
  
# with leading zeros  
$ echo {01..20}  
01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20  
  
# reverse digit  
$ echo {20..1}  
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1  
  
# reversed with leading zeros  
$ echo {20..01}  
20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01  
  
# combining multiple braces  
$ echo {a..d}{1..3}  
a1 a2 a3 b1 b2 b3 c1 c2 c3 d1 d2 d3
```

La expansión Brace es la primera expansión que tiene lugar, por lo que no se puede combinar con ninguna otra expansión.

Solo se pueden usar caracteres y dígitos.

Esto no funcionará: `echo {$(date +%H)..24}`

Hacer múltiples directorios con sub-directorios

```
mkdir -p toplevel/sublevel_{01..09}/{child1,child2,child3}
```

Esto creará una carpeta de nivel superior llamada `toplevel`, nueve carpetas dentro de `toplevel` llamado `sublevel_01`, `sublevel_02`, etc. luego dentro de esos subniveles: `child1`, `child2`, `child3` carpetas, que le da:

```
toplevel/sublevel_01/child1
```

```
toplevel/sublevel_01/child2  
toplevel/sublevel_01/child3  
toplevel/sublevel_02/child1
```

y así. Encuentro esto muy útil para crear múltiples carpetas y subcarpetas para mis propósitos específicos, con un solo comando bash. Sustituya variables para ayudar a automatizar / analizar la información dada al script.

Lea Expansión de la abrazadera en línea: <https://riptutorial.com/es/bash/topic/3351/expansion-de-la-abrazadera>

Capítulo 33: Expansión del parámetro Bash

Introducción

El carácter `$` introduce la expansión de parámetros, la sustitución de comandos o la expansión aritmética. El nombre del parámetro o el símbolo que se expandirá puede incluirse entre llaves, que son opcionales, pero sirven para proteger la variable que se expandirá de los caracteres que la siguen, lo que podría interpretarse como parte del nombre.

Lea más en el [manual de usuario de Bash](#) .

Sintaxis

- `${parámetro: desplazamiento}` # subcadena comenzando en desplazamiento
- `${parámetro: desplazamiento: longitud}` # subcadena de longitud "longitud" que comienza en desplazamiento
- `${# parámetro}` # Longitud del parámetro
- `${parámetro / patrón / cadena}` # Reemplace la primera aparición del patrón con cadena
- `${parámetro // patrón / cadena}` # Reemplazar todas las apariciones de patrón con cadena
- `${parameters / # patrón / cadena}` # Reemplazar patrón con cadena si el patrón está al principio
- `${parámetro /% patrón / cadena}` # Reemplazar patrón con cadena si el patrón está al final
- `${parámetro # patrón}` # Eliminar la coincidencia más corta del patrón desde el comienzo del parámetro
- `${parámetro ## patrón}` # Eliminar la coincidencia más larga del patrón desde el comienzo del parámetro
- `${parámetro% patrón}` # Eliminar la coincidencia más corta del patrón del final del parámetro
- `${parámetro %% patrón}` # Eliminar la coincidencia más larga del patrón desde el final del parámetro
- `${parámetro: -word}` # Expandir a palabra si el parámetro no está establecido / no definido
- `${parámetro: = palabra}` # Expandir a palabra si parámetro no establecido / no definido y establecer parámetro
- `${parámetro: + palabra}` # Expandir a palabra si el conjunto de parámetros / definido

Examples

Subcadenas y subarreglos

```
var='0123456789abcdef'

# Define a zero-based offset
$ printf '%s\n' "${var:3}"
3456789abcdef
```

```
# Offset and length of substring
$ printf '%s\n' "${var:3:4}"
3456
```

4.2

```
# Negative length counts from the end of the string
$ printf '%s\n' "${var:3:-5}"
3456789a

# Negative offset counts from the end
# Needs a space to avoid confusion with ${var:-6}
$ printf '%s\n' "${var: -6}"
abcdef

# Alternative: parentheses
$ printf '%s\n' "${var:(-6)}"
abcdef

# Negative offset and negative length
$ printf '%s\n' "${var: -6:-5}"
a
```

Las mismas expansiones se aplican si el parámetro es un **parámetro posicional** o el **elemento de una matriz con subíndices** :

```
# Set positional parameter $1
set -- 0123456789abcdef

# Define offset
$ printf '%s\n' "${1:5}"
56789abcdef

# Assign to array element
myarr[0]='0123456789abcdef'

# Define offset and length
$ printf '%s\n' "${myarr[0]:7:3}"
789
```

Las expansiones análogas se aplican a los **parámetros posicionales** , donde las compensaciones se basan en una sola:

```
# Set positional parameters $1, $2, ...
$ set -- 1 2 3 4 5 6 7 8 9 0 a b c d e f

# Define an offset (beware $0 (not a positional parameter)
# is being considered here as well)
$ printf '%s\n' "${@:10}"
0
a
b
c
d
e
f
```

```
# Define an offset and a length
$ printf '%s\n' "${@:10:3}"
0
a
b

# No negative lengths allowed for positional parameters
$ printf '%s\n' "${@:10:-2}"
bash: -2: substring expression < 0

# Negative offset counts from the end
# Needs a space to avoid confusion with ${@:-10:2}
$ printf '%s\n' "${@: -10:2}"
7
8

# ${@:0} is $0 which is not otherwise a positional parameters or part
# of $@
$ printf '%s\n' "${@:0:2}"
/usr/bin/bash
1
```

La expansión de subcadenas se puede utilizar con **matrices indexadas** :

```
# Create array (zero-based indices)
$ myarr=(0 1 2 3 4 5 6 7 8 9 a b c d e f)

# Elements with index 5 and higher
$ printf '%s\n' "${myarr[@]:12}"
c
d
e
f

# 3 elements, starting with index 5
$ printf '%s\n' "${myarr[@]:5:3}"
5
6
7

# The last element of the array
$ printf '%s\n' "${myarr[@]: -1}"
f
```

Longitud del parámetro

```
# Length of a string
$ var='12345'
$ echo "${#var}"
5
```

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

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

# Works for positional parameters as well
$ set -- 1 2 3 4
$ echo "${#@}"
4

# But more commonly (and portably to other shells), one would use
$ echo "$#"
4
```

Modificando el caso de los caracteres alfabéticos.

4.0

A mayúsculas

```
$ v="hello"
# Just the first character
$ printf '%s\n' "${v^}"
Hello
# All characters
$ printf '%s\n' "${v^^}"
HELLO
# Alternative
$ v="hello world"
$ declare -u string="$v"
$ echo "$string"
HELLO WORLD
```

A minúsculas

```
$ v="bYE"
# Just the first character
$ printf '%s\n' "${v,}"
bYE
# All characters
$ printf '%s\n' "${v,,}"
bye
# Alternative
$ v="HELLO WORLD"
$ declare -l string="$v"
$ echo "$string"
hello world
```

Caso de palanca

```
$ v="Hello World"
# All chars
$ echo "${v~~}"
hELLO wORLD
$ echo "${v~}"
# Just the first char
```

```
hello World
```

Parámetro indirecto

Bash **indirección de 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 expansión indirecta:

```
$ foo=10
$ x=foo
$ echo ${x}      #Classic variable print
foo

$ foo=10
$ x=foo
$ echo "${!x}"    #Indirect expansion
10
```

Un ejemplo más:

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

$ argtester () { for (( i=1; i<="$#"; i++ )); do echo "${!i}";done; }; argtester -ab -cd -ef
-ab  # i=1 --> expanded to $1 ---> expanded to first argument sent to function
-cd  # i=2 --> expanded to $2 ---> expanded to second argument sent to function
-ef  # i=3 --> expanded to $3 ---> expanded to third argument sent to function
```

Sustitución de valor por defecto

```
${parameter:-word}
```

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

```
$ unset var
$ echo "${var:-XX}"    # Parameter is unset -> expansion XX occurs
XX
$ var=""              # Parameter is null -> expansion XX occurs
```



```
$ echo "${var:-XX}"
XX
$ var=23                # Parameter is not null -> original expansion occurs
$ echo "${var:-XX}"
23
```

```
${parameter:=word}
```

Si el parámetro no está definido o es nulo, la expansión de la palabra se asigna al parámetro. El valor del parámetro se sustituye. Los parámetros posicionales y los parámetros especiales no pueden asignarse de esta manera.

```
$ unset var
$ echo "${var:=XX}"      # Parameter is unset -> word is assigned to XX
XX
$ echo "$var"
XX
$ var=""                # Parameter is null -> word is assigned to XX
$ echo "${var:=XX}"
XX
$ echo "$var"
XX
$ var=23                # Parameter is not null -> no assignment occurs
$ echo "${var:=XX}"
23
$ echo "$var"
23
```

Error si la variable está vacía o sin configurar

La semántica para esto es similar a la de la sustitución del valor predeterminado, pero en lugar de sustituir un valor predeterminado, se produce un error con el mensaje de error proporcionado. Los formularios son `${VARNAME?ERRMSG}` y `${VARNAME:?ERRMSG}`. La forma con `:` será nuestro error si la variable **no** está **definida o está vacío**, mientras que la forma sin error sólo si la variable *no* está *definida*. Si se produce un error, se emite `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 is
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}"
```

La ejecución del ejemplo completo sobre cada una de las declaraciones de error de error debe ser comentada para continuar.

Eliminar un patrón desde el principio de una cadena

Partido más corto:

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

Partido más largo:

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

Eliminar un patrón del final de una cadena

Partido más corto:

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

Partido más largo:

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

Reemplazar patrón en cadena

Primer partido:

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

Todos los partidos:

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

Partido al principio:

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

Partido al final:

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

Reemplazar un patrón con nada:

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

Añadir prefijo a los elementos de la matriz:

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

Munging durante la expansión

Las variables no necesariamente tienen que expandirse a sus valores: las subcadenas se pueden extraer durante la expansión, lo que puede ser útil para extraer extensiones de archivo o partes de rutas. Los caracteres globales mantienen sus significados habituales, por lo que `.*` 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 predeterminado; digamos que quiero invocar el editor del usuario, pero si no han establecido uno, me gustaría darles `vim`.

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

Hay dos formas diferentes de realizar esta expansión, que difieren en si la variable relevante está vacía o sin configurar. Usando `:-` usará el valor predeterminado si la variable no está establecida o está vacía, mientras que `-` solo usa el valor predeterminado si la variable está desactivada, pero usará la variable si está establecida en la cadena 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
```

Similar a los valores por defecto, se pueden dar alternativas; donde se usa un valor predeterminado si una variable particular no está disponible, se usa 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 se vuelve particularmente útil al proporcionar argumentos a los indicadores de línea de comando;

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

Expansión de parámetros y nombres de archivos

Puede usar Bash Parameter Expansion para emular operaciones comunes de procesamiento de nombres de archivo como `basename` y `dirname`.

Usaremos esto como nuestro camino de ejemplo:

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

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

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

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

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

Para emular el `basename $FILENAME .txt` y devolver el nombre de archivo sin el `.txt` extensión:

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

Lea Expansión del parámetro Bash en línea: <https://riptutorial.com/es/bash/topic/502/expansion-del-parametro-bash>

Capítulo 34: Expresiones condicionales

Sintaxis

- `[[-OP $ nombre de archivo]]`
- `[[$ file1 -OP $ file2]]`
- `[[-z $ cadena]]`
- `[[-n $ cadena]]`
- `[["$ string1" == "$ string2"]]`
- `[["$ string1" == $ patrón]]`

Observaciones

La sintaxis `[[...]]` rodea a las expresiones condicionales incorporadas de bash. Tenga en cuenta que se requieren espacios en ambos lados de los soportes.

Las expresiones condicionales pueden usar operadores unarios y binarios para probar las propiedades de cadenas, enteros y archivos. También pueden usar los operadores lógicos `&&`, `||` y `!`.

Examples

Comparacion 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 archivos pueden ser iguales incluso si tienen nombres diferentes, por ejemplo, si son enlaces duros, o si son enlaces simbólicos con el mismo objetivo, o si uno es un enlace simbólico que apunta al otro.

Si dos archivos tienen el mismo contenido, pero son archivos distintos (para que la modificación de uno no afecte al otro), `-ef` reporta como diferentes. Si desea comparar dos archivos byte a byte, use la utilidad `cmp`.

```
if cmp -s -- "$file1" "$file2"; then
    echo "$file1 and $file2 have identical contents"
else
    echo "$file1 and $file2 differ"
fi
```

Para producir una lista legible por humanos de diferencias entre archivos de texto, use la utilidad `diff`.

```
if diff -u "$file1" "$file2"; then
    echo "$file1 and $file2 have identical contents"
else
    : # the differences between the files have been listed
fi
```

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 toman en cuenta los permisos y la propiedad para determinar si el script (o los programas iniciados desde el script) pueden acceder al archivo.

Cuidado con las [condiciones](#) de la [carrera \(TOCTOU\)](#) : solo porque la prueba sea exitosa ahora no significa que aún sea válida en la siguiente línea. 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 mientras tanto.

Comparaciones numericas

Las comparaciones numéricas utilizan los operadores `-eq` y amigos.

```
if [[ $num1 -eq $num2 ]]; then
    echo "$num1 == $num2"
fi
if [[ $num1 -le $num2 ]]; then
    echo "$num1 <= $num2"
fi
```

Hay seis operadores numéricos:

- `-eq` igual
- `-ne` es igual
- `-le` menos o igual
- `-lt` menos que
- `-ge` mayor o igual
- `-gt` mayor que

Tenga en cuenta que los operadores `<` y `>` dentro de `[[...]]` comparan cadenas, 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 inicial). Alternativamente, use la sintaxis de expresión aritmética `((...))`, que realiza cálculos de **enteros** en una sintaxis similar a C / Java / ...

```
x=2
if ((2*x == 4)); then
    echo "2 times 2 is 4"
fi
((x += 1))
echo "2 plus 1 is $x"
```

Comparación de cadenas y emparejamiento

La comparación de cadenas utiliza el operador `==` entre cadenas *citadas*. 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 no se cita el lado derecho, entonces es un patrón de comodín con el que se `$string1`.

```
string='abc'
pattern1='a*'
pattern2='x*'
if [[ "$string" == $pattern1 ]]; then
    # the test is true
    echo "The string $string matches the pattern $pattern"
fi
if [[ "$string" != $pattern2 ]]; then
    # the test is false
    echo "The string $string does not match the pattern $pattern"
fi
```

Los operadores `<` y `>` comparan las cadenas en orden lexicográfico (no hay operadores menos o iguales o mayores o iguales para las cadenas).

Hay pruebas unarias para la cadena 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 está configurada en una cadena vacía. Para distinguir entre vacío y no definido, use:

```
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 arbitrario. O en [forma de tabla](#) :

	unset	empty	non-empty
<code>[[-z \${string}]]</code>	true	true	false
<code>[[-z \${string+x}]]</code>	true	false	false
<code>[[-z \${string-x}]]</code>	false	true	false
<code>[[-n \${string}]]</code>	false	false	true
<code>[[-n \${string+x}]]</code>	false	true	true
<code>[[-n \${string-x}]]</code>	true	false	true

[Alternativamente](#) , el estado se puede verificar en una declaración de caso:

```
case ${var+x$var} in
    (x) echo empty;;
    ("") echo unset;;
    (x*[:blank:]*) echo non-blank;;
    (*) echo blank
esac
```

Donde `[:blank:]` es la ubicación específica de caracteres espaciados horizontales (tabulación, espacio, etc.).

Pruebas de tipo de archivo

El operador condicional `-e` comprueba si existe un archivo (incluidos todos los tipos de archivos: directorios, etc.).

```
if [[ -e $filename ]]; then
    echo "$filename exists"
fi
```

También hay pruebas para tipos de archivos 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 objetivo y devuelven falso 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

```

Prueba de estado de salida de un comando

Estado de salida 0: éxito

Estado de salida distinto de 0: error

Para probar el estado de salida de un comando:

```

if command;then
    echo 'success'
else
    echo 'failure'
fi

```

Una prueba de 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 #If cmd fails try cmd1
```

Lea Expresiones condicionales en línea: <https://riptutorial.com/es/bash/topic/731/expresiones-condicionales>

Capítulo 35: Finalización programable

Examples

Finalización simple utilizando la función.

```
_mycompletion() {  
    local command_name="$1" # not used in this example  
    local current_word="$2"  
    local previous_word="$3" # not used in this example  
    # COMPREPLY is an array which has to be filled with the possible completions  
    # compgen is used to filter matching completions  
    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
```

Terminación simple para opciones y nombres de archivos

```
# The following shell function will be used to generate completions for  
# the "nuance_tune" command.  
_nuance_tune_opts ()  
{  
    local curr_arg prev_arg  
    curr_arg=${COMP_WORDS[COMP_CWORD]}  
    prev_arg=${COMP_WORDS[COMP_CWORD-1]}  
  
    # The "config" option takes a file arg, so get a list of the files in the  
    # current dir. A case statement is probably unnecessary here, but leaves  
    # room to customize the parameters for other flags.  
    case "$prev_arg" in  
        -config)  
            COMPREPLY=( $( /bin/ls -1 ) )  
            return 0  
            ;;  
        esac  
  
    # Use compgen to provide completions for all known options.  
    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 )  
);  
}  
  
# The -o parameter tells Bash to process completions as filenames, where applicable.
```

```
complete -o filenames -F _nuance_tune_opts nuance_tune
```

Lea Finalización programable en línea: <https://riptutorial.com/es/bash/topic/3162/finalizacion-programable>

Capítulo 36: Funciones

Sintaxis

- Defina una función con la palabra clave de `function` :

```
function f {  
  
}
```

- Definir una función con `()` :

```
f() {  
  
}
```

- Defina una función con la palabra clave de `function` y `()` :

```
function f(){  
  
}
```

Examples

Función simple

En `helloWorld.sh`

```
#!/bin/bash  
  
# Define a function greet  
greet ()  
{  
    echo "Hello World!"  
}  
  
# Call the function greet  
greet
```

Al ejecutar el script, vemos nuestro mensaje.

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

Tenga en cuenta que la [obtención de](#) un archivo con funciones los hace disponibles en su sesión de bash actual.

```
$ source helloWorld.sh    # or, more portably, ". helloWorld.sh"
$ greet
Hello World!
```

Puede `export` una función en algunos shells, para que esté expuesta a procesos secundarios.

```
bash -c 'greet'    # fails
export -f greet    # export function; note -f
bash -c 'greet'    # success
```

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 modifica el argumento de ninguna manera, no es necesario copiarlo en una variable `local` , simplemente haga `echo "Hello, $1"` .
2. Puede usar `$1` , `$2` , `$3` y así sucesivamente para acceder a los argumentos dentro de la función.

Nota: para argumentos más de 9 `$10` no funcionarán (bash lo leerá como `$ 1 0`), debe hacer `${10}` , `${11}` y así sucesivamente.

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

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

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

Nota: Prácticamente siempre debe usar comillas dobles alrededor de `"$@"` , como aquí.

Omitir las comillas hará que el shell amplíe los comodines (incluso cuando el usuario los haya citado específicamente para evitar eso) y, en general, introduzca comportamientos no deseados y posiblemente incluso problemas de seguridad.

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

4. para los argumentos predeterminados use `${1:-default_val}` . P.ej:

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

foo      # output => 25
foo 30   # output => 30
```

5. para requerir un argumento usa `${var:?error message}`

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

Valor de retorno de una función

La declaración de `return` en Bash no devuelve un valor como las funciones C, sino que sale de la función con un estado de retorno. Puedes considerarlo como el estado de salida de esa función.

Si desea devolver un valor de la función, envíe el valor a `stdout` así:

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

Ahora, si lo haces:

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

La salida de `fun` será almacenada en `$var` .

Manejo de banderas y parámetros opcionales.

La *función* incorporada `getopts` se puede usar dentro de las funciones para escribir funciones que se ajusten a indicadores y parámetros opcionales. Esto no presenta ninguna dificultad especial, pero uno tiene que manejar adecuadamente los valores tocados por los `getopts` . Como ejemplo, definimos una función de *falla de fallos* 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
# Fail with the given diagnostic message
```

```
#
# The -x flag can be used to convey a custom exit status, instead of
# the value 1. A newline is automatically added to the output.

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 se puede utilizar de la siguiente manera:

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

y así.

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

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

El código de salida de una función es el código de salida de su último comando.

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

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

Esta función envía un solo ping al host especificado por el primer parámetro de la función. La salida y la salida de error de `ping` se redirigen a `/dev/null`, por lo que la función nunca generará nada. Pero el comando `ping` tendrá el código de salida 0 en caso de éxito, y no cero en caso de error. Como este es el último (y en este ejemplo, el único) comando de la función, el código de

salida de `ping` se reutilizará para el código de salida de la función en sí.

Este hecho es muy útil en declaraciones condicionales.

Por ejemplo, si el host `graucho` está activo, entonces conéctese con `ssh` :

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

Otro ejemplo: verifique repetidamente hasta que el host `graucho` esté activo, y luego conéctese con `ssh` :

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

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"
}
```

Una función que acepta parámetros nombrados.

```
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
```

Lea Funciones en línea: <https://riptutorial.com/es/bash/topic/472/funciones>

Capítulo 37: Gestionando variable de entorno PATH

Sintaxis

- Agregar ruta: `PATH = $ PATH: / new / path`
- Agregar ruta: `PATH = / new / path: $ PATH`

Parámetros

Parámetro	Detalles
CAMINO	Variable de entorno de ruta

Observaciones

Bash archivo de configuración:

Este archivo se obtiene cada vez que se inicia un nuevo shell interactivo de Bash.

En sistemas GNU / Linux es generalmente el archivo `~ / .bashrc`; en Mac es `~ / .bash_profile` o `~ / .profile`

Exportar:

La variable PATH se debe exportar una vez (se realiza de forma predeterminada). Una vez que se exporte, seguirá exportándose y cualquier cambio que se le haga se aplicará de inmediato.

Aplicar los cambios:

Para aplicar los cambios a un archivo de configuración de Bash, debe volver a cargar ese archivo en un terminal (`source /path/to/bash_config_file`)

Examples

Agregar 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 Bash específico de la distro)

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/usr
```

Ahora, si queremos agregar una ruta (por ejemplo, `~/bin`) a la variable PATH:

```
PATH=~/.bin:$PATH
# or
PATH=$PATH:~/.bin
```

Pero esto modificará el PATH solo en el shell actual (y su subshell). Una vez que salga del shell, esta modificación desaparecerá.

Para hacerlo permanente, debemos agregar ese bit de código al archivo `~ / .bashrc` (o lo que sea) y volver a cargar el archivo.

Si ejecuta el siguiente código (en el terminal), agregará `~/bin` a la RUTA permanentemente:

```
echo 'PATH=~/.bin:$PATH' >> ~/.bashrc && source ~/.bashrc
```

Explicación:

- `echo 'PATH=~/.bin:$PATH' >> ~/.bashrc` agrega la línea `PATH=~/.bin:$PATH` al final del archivo `~/.bashrc` (puede hacerlo con un editor de texto)
- `source ~/.bashrc` vuelve a cargar el archivo `~/.bashrc`

Esto es un poco de código (ejecutado en el terminal) que verificará si una ruta ya está incluida y agregará la ruta solo si no:

```
path=~/.bin           # path to be included
bashrc=~/.bashrc      # bash file to be written and reloaded
# run the following code unmodified
echo $PATH | grep -q "\(^|:|:)$path\(:|/|\{0,1\}$\)" || echo "PATH=\$PATH:$path" >> "$bashrc";
source "$bashrc"
```

Eliminar una ruta de la variable de entorno PATH

Para eliminar un PATH de una variable de entorno PATH, debe editar el archivo `~ / .bashrc` o `~ / .bash_profile` // `etc / profile` o `~ / .profile` o `/etc/bash.bashrc` (específico de la distro) y eliminar la asignación de ese camino particular

En lugar de encontrar la asignación exacta, puede hacer un reemplazo en el `$PATH` en su etapa final.

Lo siguiente quitará `$path` de forma segura de `$PATH` :

```
path=~/.bin
PATH="$ (echo "$PATH" | sed -e "s#\(^|:)\$ (echo "$path" | sed -e 's/[^\^]/[&]/g' -e 's/\^\//\^\^/g')\(:|/|\{0,1\}\$\)#1\2# -e 's#:\+:#:q' -e 's#^\:|:##q')"
```

Para hacerlo permanente, deberá agregarlo al final de su archivo de configuración de 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 hará que sea más fácil manejar múltiples rutas.

Notas:

- Deberá agregar estos códigos en el archivo de configuración de Bash (~ / .bashrc o lo que sea).
- Ejecute `source ~/.bashrc` para volver a cargar el archivo de configuración de Bash (~ / .bashrc).

Lea Gestionando variable de entorno PATH en línea:

<https://riptutorial.com/es/bash/topic/5515/gestionando-variable-de-entorno-path>

Capítulo 38: getopt: análisis inteligente de parámetros posicionales

Sintaxis

- nombre optstring de getopt [args]

Parámetros

Parámetro	Detalle
optstring	Los caracteres de opción a reconocer.
nombre	Luego nombre donde se almacena la opción analizada

Observaciones

Opciones

`optstring` : los caracteres opcionales a reconocer.

Si un carácter va seguido de dos puntos, se espera que la opción tenga un argumento, que debe estar separado de él por un espacio en blanco. Los dos puntos (:) (y signo de interrogación ?) No pueden utilizarse como caracteres de opción.

Cada vez que se invoca, `getopts` coloca la siguiente opción en el nombre de la variable de shell, inicializando el nombre si no existe, y el índice del siguiente argumento que se procesará en la variable `OPTIND` . `OPTIND` se inicializa en 1 cada vez que se invoca el shell o un script de shell.

Cuando una opción requiere un argumento, `getopts` coloca ese argumento en la variable `OPTARG` . El shell no restablece `OPTIND` automáticamente; debe restablecerse manualmente entre varias llamadas a `getopts` dentro de la misma invocación de shell si se va a usar un nuevo conjunto de parámetros.

Cuando se encuentra el final de las opciones, `getopts` sale con un valor de retorno mayor que cero.

`OPTIND` se establece en el índice del primer argumento no opcional y el nombre se establece en ? . `getopts` normalmente analiza los parámetros posicionales, pero si se dan más argumentos en `args` , `getopts` analiza en su lugar.

`getopts` puede reportar errores de dos maneras. Si el primer carácter del `optstring` es de dos

puntos (:), se utiliza el informe de errores en silencio. En el funcionamiento normal, los mensajes de diagnóstico se imprimen cuando se encuentran opciones inválidas o argumentos de opciones faltantes.

Si la variable `OPTERR` se establece en 0 , no se mostrarán mensajes de error, incluso si el primer carácter de `optstring` no es dos puntos.

Si se ve una opción no válida, `getopts` lugares `getopts ?` en `name` y, si no está en silencio, imprime un mensaje de error y `OPTARG` . Si `getopts` está en silencio, el carácter de opción encontrado se coloca en `OPTARG` y no se imprime ningún mensaje de diagnóstico.

Si no se encuentra un argumento requerido y `getopts` no está en silencio, se coloca un signo de interrogación (?) En el `name` , se desactiva `OPTARG` y se imprime un mensaje de diagnóstico. Si `getopts` es silenciosa, luego dos puntos (:) se coloca en nombre y `OPTARG` se establece en el carácter de opción.

Examples

mapa de ping

```
#!/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 version

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]

```

Lea getopts: análisis inteligente de parámetros posicionales en línea:

<https://riptutorial.com/es/bash/topic/3654/getopts--analisis-inteligente-de-parametros-posicionales>

Capítulo 39: Grep

Sintaxis

- `grep [OPCIONES] PATRÓN [ARCHIVO ...]`

Examples

Cómo buscar un archivo para un patrón

Para encontrar la palabra **foo** en la *barra de* archivos:

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

Para encontrar todas las líneas que **no** contienen foo en la *barra de* archivos:

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

Para usar encontrar todas las palabras que contienen foo al final (expansión de tarjeta de crédito):

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

Lea Grep en línea: <https://riptutorial.com/es/bash/topic/10852/grep>

Capítulo 40: Leer un archivo (flujo de datos, variable) línea por línea (y / o campo por campo)?

Parámetros

Parámetro	Detalles
IFS	Separador de campo interno
expediente	Un nombre de archivo / ruta
-r	Previene la interpretación de barra invertida cuando se usa con lectura
-t	Elimina una nueva línea final de cada línea leída por <code>readarray</code>
-d DELIM	Continuar hasta que se lea (con <code>read</code>) el primer carácter de DELIM, en lugar de nueva línea

Examples

Lee 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 el archivo de contraseñas de Unix, la información del usuario se almacena línea por línea, y cada línea consta de información para un usuario separado por dos puntos (:). En este ejemplo, al leer el archivo línea por línea, la línea también se divide en campos que utilizan caracteres de dos puntos como delimitador, lo que se indica mediante el valor dado para IFS.

Entrada de muestra

```
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
```

Salida de muestra

```
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 la línea completa asignada a la variable, a continuación se muestra una versión modificada del ejemplo. Tenga en cuenta que solo tenemos una variable por nombre de línea mencionada aquí.

```
#!/bin/bash
FILENAME="/etc/passwd"
while IFS= read -r line
do
    echo "$line"
done < $FILENAME
```

Entrada de muestra

```
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
```

Salida de muestra

```
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
```

Leer las líneas de un archivo en una matriz

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

O con un bucle:

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

Bucle a través de un archivo 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
```

Leer las líneas de una cadena en una matriz

```
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"
```

Corriendo a través de una cuerda 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
```

Bucle a través de la salida de una línea de comando por línea

```
while IFS= read -r line;do
    echo "***$line**"
done < <(ping google.com)
```

o con una pipa:

```
ping google.com |
while IFS= read -r line;do
    echo "***$line**"
done
```

Lee un archivo campo por campo

Supongamos que el separador de campo es : (dos puntos) en el *archivo* .

```
while IFS= read -d : -r field || [ -n "$field" ]; do
    echo "$field"
done <file
```

Para un contenido:

```
first : se
con
d:
    Thi rd:
    Fourth
```

La salida es:

```
**first **
** se
con
d**
**
    Thi rd**
**
    Fourth
**
```

Leer una cadena de campo por campo

Supongamos que el separador de campo 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
-
```

Leer los campos de un archivo en una matriz

Supongamos que el separador de campo es :

```
arr=()
while IFS= read -d : -r field || [ -n "$field" ]; do
    arr+=("$field")
done <file
```

Leer campos de una cadena en una matriz

Supongamos que el separador de campo 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
```

Bucle a través de la salida de un campo de comando por campo

Supongamos que el separador de campo es :

```
while IFS= read -d : -r field || [ -n "$field" ];do
    echo "***$field**"
done < <(ping google.com)
```

O con una pipa:

```
ping google.com | while IFS= read -d : -r field || [ -n "$field" ];do
    echo "***$field**"
done
```

Lea **Leer un archivo (flujo de datos, variable) línea por línea (y / o campo por campo)? en línea:**
<https://riptutorial.com/es/bash/topic/5473/leer-un-archivo--flujo-de-datos--variable--linea-por-linea--y---o-campo-por-campo-->

Capítulo 41: Listado de archivos

Sintaxis

- ls [OPCIÓN] ... [ARCHIVO] ...

Parámetros

Opción	Descripción
-a , --all	Listar todas las entradas, incluyendo las que comienzan con un punto
-A , --almost-all	Listar todas las entradas excluyendo . y ..
-c	Ordenar archivos por hora de cambio
-d , --directory	Listar entradas de directorio
-h , --human-readable	Mostrar tamaños en formato legible para humanos (es decir, K , M)
-H	Igual que el anterior solo con poderes 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 contenidos en orden inverso
-s , --size	Tamaño de impresión de cada archivo en bloques.
-S	Ordenar por tamaño de archivo
--sort=WORD	Ordenar los contenidos por una palabra. (es decir, tamaño, versión, estado)
-t	Ordenar por tiempo de modificación
-u	Ordenar por último tiempo de acceso
-v	Ordenar por versión
-1	Listar un archivo por línea

Examples

Lista de archivos

El comando `ls` enumera los contenidos de un directorio específico, **excluyendo los** archivos de puntos. Si no se especifica ningún directorio, entonces, de forma predeterminada, se enumeran los contenidos del directorio actual.

Los archivos listados están ordenados alfabéticamente, por defecto, y alineados en columnas si no caben en una línea.

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

Lista de archivos en un formato de listado largo

La opción `-l` del comando `ls` imprime el contenido de un directorio específico en un formato de listado largo. Si no se especifica ningún directorio, entonces, de forma predeterminada, se enumeran los contenidos 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 primero muestra el `total`, lo que indica el tamaño total en **bloques** de todos los archivos en el directorio listado. Luego muestra ocho columnas de información para cada archivo en el directorio listado. A continuación se muestran los detalles de cada columna en la salida:

Columna No.	Ejemplo	Descripción
1.1	d	Tipo de archivo (ver tabla abajo)
1.2	rwxr-xr-x	Cadena de permiso
2	3	Número de enlaces duros
3	root	Nombre del dueño
4	root	Grupo propietario
5	4096	Tamaño de archivo en bytes
6	Apr 21 03:44	Tiempo de modificacion

Columna No.	Ejemplo	Descripción
7	acpi	Nombre del archivo

Tipo de archivo

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

Personaje	Tipo de archivo
-	Archivo regular
b	Bloquear archivo especial
c	Archivo especial de caracteres
C	Archivo de alto rendimiento ("datos contiguos")
d	Directorio
D	Puerta (archivo IPC especial solo en Solaris 2.5+)
l	Enlace simbólico
M	Archivo fuera de línea ("migrado") (Cray DMF)
n	Archivo especial de red (HP-UX)
p	FIFO (tubo con nombre)
P	Puerto (archivo de sistema especial en Solaris 10+ solamente)
s	Enchufe
?	Algún otro tipo de archivo

Lista de archivos ordenados por tamaño

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

```
$ 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
```

Cuando se usa con la opción `-r`, el orden se invierte.

```
$ ls -l -S -r -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
```

Lista de archivos sin usar `ls`

Utilice la [expansión de nombre de archivo](#) del shell de Bash y las capacidades de [expansión de refuerzo](#) para obtener los nombres de archivo:

```
# display the files and directories that are in the current directory
printf "%s\n" *

# display only the directories in the current directory
printf "%s\n" */

# display only (some) image files
printf "%s\n" *.{gif,jpg,png}
```

Para capturar una lista de archivos en una variable para su procesamiento, normalmente es una buena práctica usar una [matriz bash](#) :

```
files=( * )

# iterate over them
for file in "${files[@]}"; do
    echo "$file"
done
```

Listar los diez archivos modificados más recientemente

A continuación se enumerarán hasta diez de los archivos modificados más recientemente en el directorio actual, utilizando un formato de listado largo (`-l`) y ordenados por tiempo (`-t`).

```
ls -lt | head
```

Listar todos los archivos incluyendo archivos de puntos

Un archivo de **puntos** es un archivo cuyos nombres comienzan con a `.` . Estos normalmente están ocultos por `ls` y no están listados a menos que sean solicitados.

Por ejemplo la siguiente salida de `ls` :

```
$ ls
bin  pki
```

El `-a` o `--all` opción mostrará una lista de todos los archivos, incluyendo dotfiles.

```
$ ls -a
```

```
.  .ansible      .bash_logout  .bashrc  .lessht  .puppetlabs  .viminfo
.. .bash_history .bash_profile bin      pki      .ssh
```

La opción `-A` o `-A --almost-all` muestra una lista de todos los archivos, incluidos los archivos de puntos, pero no la lista implícita `.` y `..`. 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
```

Lista de archivos en un formato de árbol

El comando de `tree` enumera los contenidos de un directorio específico en un formato similar a un árbol. Si no se especifica ningún directorio, entonces, de forma predeterminada, se enumeran los contenidos del directorio actual.

Ejemplo de salida:

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

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

Ejemplo de salida:

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

Lea Listado de archivos en línea: <https://riptutorial.com/es/bash/topic/366/listado-de-archivos>

Capítulo 42: Manejando el indicador del sistema

Sintaxis

- export PS1 = "algo" # se muestra cuando bash espera un comando para que se escriba
- exportar PS2 = "algo" # se muestra cuando la instrucción se extiende a más líneas
- export PS3 = "solicitud de pregunta para la instrucción de selección" # # rara vez se utiliza la solicitud para seleccionar. Primero configura la PS3 según tus necesidades, luego **selecciona** Call. Ver **ayuda seleccionar**
- export PS4 = "principalmente útil para la depuración; número de línea y así sucesivamente" # utilizado para depurar scripts de bash.

Parámetros

Escapar	Detalles
<code>\una</code>	Un personaje de campana.
<code>\re</code>	La fecha, en formato "Fecha del mes de la semana" (por ejemplo, "Mar 26 de mayo").
<code>\D {FORMATO}</code>	El FORMATO se pasa a <code>`strftime`</code> (3) y el resultado se inserta en la cadena de solicitud; un FORMATO vacío da como resultado una representación temporal específica de la localidad. Los tirantes son necesarios.
<code>\mi</code>	Un personaje de escape. <code>\033</code> funciona por supuesto también.
<code>\h</code>	El nombre de host, hasta el primer <code>`.`</code> . (es decir, ninguna parte de dominio)
<code>\H</code>	El nombre de host finalmente con parte de dominio
<code>\j</code>	El número de trabajos actualmente gestionados por el shell.
<code>\l</code>	El nombre base del nombre del dispositivo terminal del shell.
<code>\norte</code>	Una nueva línea.
<code>\r</code>	Un retorno de carro.
<code>\s</code>	El nombre del shell, el nombre base de <code>`\$ 0`</code> (la parte que sigue a la barra final).
<code>\t</code>	La hora, en formato HH: MM: SS las 24 horas.
<code>\T</code>	La hora, en formato HH: MM: SS de 12 horas.

Escapar	Detalles
@	La hora, en formato de 12 horas am / pm.
\UNA	La hora, en formato HH: MM 24 horas.
\u	El nombre de usuario del usuario actual.
\v	La versión de Bash (por ejemplo, 2.00)
\V	El lanzamiento de Bash, version + patchlevel (por ejemplo, 2.00.0)
\w	El directorio de trabajo actual, con \$ HOME abreviado con una tilde (utiliza la variable \$ PROMPT_DIRTRIM).
\W	El nombre base de \$ PWD, con \$ HOME abreviado con una tilde.
!	El número de historial de este comando.
#	El número de comando de este comando.
PS	Si el uid efectivo es 0, # , de lo contrario \$.
\NNN	El carácter cuyo código ASCII es el valor octal NNN.
\	Una barra invertida.
\[Comience una secuencia de caracteres no imprimibles. Esto podría usarse para incrustar una secuencia de control de terminal en el indicador.
\]	Finaliza una secuencia de caracteres no imprimibles.

Examples

Usando la variable de entorno PROMPT_COMMAND

Cuando se realiza el último comando en una instancia de bash interactiva, se muestra la variable PS1 evaluada. Antes de mostrar el bash de PS1, se muestra si PROMPT_COMMAND está configurado. Este valor de esta var debe ser un programa o script invocable. Si se establece esta var, este programa / script se llama ANTES de que se muestre el indicador PS1.

```
# just a stupid function, we will use to demonstrate
# we check the date if Hour is 12 and Minute is lower than 59
lunchbreak(){
    if (( $(date +%H) == 12 && $(date +%M) < 59 )); then
        # and print colored \033[ starts the escape sequence
        # 5; is blinking attribute
        # 2; means bold
        # 31 says red
        printf "\033[5;1;31mmind the lunch break\033[0m\n";
    else
```

```

        printf "\033[33mstill working...\033[0m\n";
    fi;
}

# activating it
export PROMPT_COMMAND=lunchbreak

```

Usando ps2

PS2 se muestra cuando un comando se extiende a más de una línea y bash espera más pulsaciones. También se muestra cuando se ingresa un comando compuesto como **while ... do..done** y por igual.

```

export PS2="would you please complete this command?\n"
# now enter a command extending to at least two lines to see PS2

```

Usando ps3

Cuando se ejecuta la instrucción de selección, muestra los elementos dados con un número y luego muestra el indicador de PS3:

```

export PS3="  To choose your language type the preceding number : "
select lang in EN CA FR DE; do
    # check input here until valid.
    break
done

```

Usando ps4

PS4 se muestra cuando bash está en modo de depuración.

```

#!/usr/bin/env bash

# switch on debugging
set -x

# define a stupid_func
stupid_func(){
    echo I am line 1 of stupid_func
    echo I am line 2 of stupid_func
}

# setting the PS4 "DEBUG" prompt
export PS4='\nDEBUG level:${SHLVL} subshell-level: ${BASH_SUBSHELL} \nsource-file:${BASH_SOURCE}
line#:${LINENO} function:${FUNCNAME[0]}:${FUNCNAME[0]}(): }\nstatement: '

# a normal statement
echo something

# function call
stupid_func

# a pipeline of commands running in a subshell
( ls -l | grep 'x' )

```

Usando ps1

PS1 es el indicador de sistema normal que indica que bash espera a que se ingresen los comandos. Entiende algunas secuencias de escape y puede ejecutar funciones o programas. Como bash tiene que posicionar el cursor después de que aparezca el indicador, debe saber cómo calcular la longitud efectiva de la cadena del indicador. Para indicar que no se imprimen secuencias de caracteres dentro de la variable PS1, se utilizan llaves de escape: `\[una secuencia de caracteres no imprimible \]`. Todo lo que se dice es válido para todos los PS * vars.

(El cursor negro indica cursor)

```
#everything not being an escape sequence will be literally printed
export PS1="literal sequence " # Prompt is now:
literal sequence █

# \u == user \h == host \w == actual working directory
# mind the single quotes avoiding interpretation by shell
export PS1='\u@\h:\w > ' # \u == user, \h == host, \w actual working dir
looser@host:/some/path > █

# executing some commands within PS1
# following line will set foreground color to red, if user==root,
# else it resets attributes to default
# $( (($EUID == 0)) && tput setaf 1)
# later we do reset attributes to default with
# $( tput sgr0 )
# assuming being root:
PS1="\[$( (($EUID == 0)) && tput setaf 1 \]\u\[$(tput sgr0)\]@\w:\w \$ "
looser@host:/some/path > █ # if not root else <red>root<default>@host....
```

Lea Manejando el indicador del sistema en línea:

<https://riptutorial.com/es/bash/topic/7541/manejando-el-indicador-del-sistema>

Capítulo 43: Mates

Examples

Matemáticas usando dc

dc es uno de los lenguajes más antiguos en Unix.

Está utilizando la [notación de pulido inverso](#), lo que significa que primero apila números y luego operaciones. Por ejemplo, $1+1$ se escribe como `1 1+ .`

Para imprimir un elemento desde la parte superior de la pila, use el comando `p`

```
echo '2 3 + p' | dc
5

or

dc <<< '2 3 + p'
5
```

Puedes imprimir el elemento superior muchas veces

```
dc <<< '1 1 + p 2 + p'
2
4
```

Para números negativos usa el prefijo `_`

```
dc <<< '_1 p'
-1
```

También puede usar letras mayúsculas de la `A` to `F` para los números entre `10` and `15` y `.` como punto decimal

```
dc <<< 'A.4 p'
10.4
```

dc utiliza [una precisión máxima](#), lo que significa que la precisión solo está limitada por la memoria disponible. Por defecto, la precisión se establece 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 puedes usarlo sobre múltiples líneas.

```
dc << EOF  
1 1 +  
3 *  
p  
EOF  
6
```

`bc` es un preprocesador para `dc`.

Matemáticas usando bc

`bc` es un lenguaje de calculadora de precisión arbitraria. Se puede usar 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 publicaciones flotantes, puede importar la biblioteca estándar `bc -l`:

```
echo '12 / 5' | bc -l  
2.40000000000000000000
```

Se puede utilizar para comparar expresiones:

```
echo '8 > 5' | bc  
1  
  
echo '10 == 11' | bc  
0  
  
echo '10 == 10 && 8 > 3' | bc  
1
```

Matemáticas usando las capacidades de bash

El cálculo aritmético también se puede realizar sin involucrar ningún otro programa como este:

Multiplicación:

```
echo $((5 * 2))
```

```
10
```

División:

```
echo $((5 / 2))  
2
```

Modulo

```
echo $((5 % 2))  
1
```

Exposición:

```
echo $((5 ** 2))  
25
```

Matemáticas usando expr

`expr Evaluate expressions` `expr` **O** `Evaluate expressions` **evalúan una expresión y escriben el resultado en la salida estándar**

Aritmética básica

```
expr 2 + 3  
5
```

Al multiplicar, necesitas escapar del signo *

```
expr 2 \* 3  
6
```

También puedes utilizar variables.

```
a=2  
expr $a + 3  
5
```

Tenga en cuenta que solo admite enteros, por lo que una expresión como esta

```
expr 3.0 / 2
```

arrojará un error `expr: not a decimal number: '3.0'.`

Es compatible con la expresión regular para que coincida con los patrones

```
expr 'Hello World' : 'Hell\(.*\)rld'  
o Wo
```

O encuentra el índice del primer char en la cadena de búsqueda

Esto generará el `expr: syntax error` en **Mac OS X** , porque usa **BSD expr** que no tiene el comando de índice, mientras que `expr` en Linux es generalmente **GNU expr**

```
expr index hello l
3

expr index 'hello' 'lo'
3
```

Lea Mates en línea: <https://riptutorial.com/es/bash/topic/2086/mates>

Capítulo 44: Matrices asociativas

Sintaxis

- declare -A assoc_array # sin inicializar
- declare -A assoc_array = ([clave] = "valor" [otra clave] = "cuidado con los espacios" [tres espacios] = "todos los espacios en blanco se resumen")
- echo \${assoc_array[@]} # los valores
- echo \${!assoc_array[@]} # las teclas

Examples

Examinando arrays assoc

Todo el uso necesario se muestra con este fragmento de código:

```
#!/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 # just a blank line
echo now here are the values of assoc_array:
echo ${assoc_array[@]}
echo not that useful,
echo # just a blank line
echo this is better:

declare -p assoc_array # -p == print

echo have a close look at the spaces above\!\!\!
echo # just a blank line

echo accessing the keys
echo the keys in assoc_array are ${!assoc_array[*]}
echo mind the use of indirection operator \!
echo # just a blank line

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 # just a blank line

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 # just a blank line
echo # just another blank line

echo there is a difference using integers as keys\\!\\!\\!
i=1
echo declaring an integer var i=1
echo # just a blank line
echo Within an integer_array bash recognizes arithmetic context.
echo Within an assoc_array bash DOES NOT recognize arithmetic context.
echo # just a blank line
echo this works: \\${assoc_array[\\$i]}: \\${assoc_array[$i]}
echo this NOT!\\!\\! : \\${assoc_array[i]}: \\${assoc_array[i]}
echo # just a blank line
echo # just a blank line
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 # just a blank line

echo both forms do work: \\${integer_array[i]} : \\${integer_array[i]}
echo and this too: \\${integer_array[\\$i]} : \\${integer_array[$i]}

```

Lea Matrices asociativas en línea: <https://riptutorial.com/es/bash/topic/7536/matrices-asociativas>

Capítulo 45: Navegando directorios

Examples

Cambiar al último directorio.

Para el shell actual, esto te lleva al directorio anterior en el que estabas, sin importar dónde estuviera.

```
cd -
```

Si lo hace varias veces, efectivamente "alterna" su presencia en el directorio actual o en el anterior.

Cambiar al directorio de inicio

El directorio predeterminado es el directorio de inicio (`$HOME` , normalmente `/home/username`), por lo que `cd` sin ningún directorio lo lleva allí

```
cd
```

O podrías ser más explícito:

```
cd $HOME
```

Un atajo para el directorio de inicio es `~` , por lo que también podría usarse.

```
cd ~
```

Directorios absolutos vs relativos

Para cambiar a un directorio absolutamente especificado, use el nombre completo, comenzando con una barra invertida `\` , por lo tanto:

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

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

```
cd abc
```

Si desea ir al directorio que se encuentra sobre el directorio actual, puede usar el alias `..` . Por ejemplo, si estuvieras en `/home/username/project/abc` y quisieras ir a `/home/username/project` , harías lo siguiente:

```
cd ..
```

Esto también se puede llamar "subir" un directorio.

Cambio al Directorio del Guión

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

1. Herramientas del sistema que operan desde el directorio de trabajo actual.
2. Herramientas de proyecto que modifican archivos relativos a 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 del script a la ruta de su directorio
3. `cd` cambia el directorio de trabajo actual al directorio que recibe de `dirname`

Lea Navegando directorios en línea: <https://riptutorial.com/es/bash/topic/6784/navegando-directorios>

Capítulo 46: Oleoductos

Sintaxis

- `[time [-p]] [!] command1 [| o | y comando2]...`

Observaciones

Una tubería es una secuencia de comandos simples separados por uno de los operadores de control `|` o `&` ([fuente](#)).

`|` conecta la salida de `command1` a la entrada de `command2`.

`&` conecta la salida estándar y el error estándar de `command1` a la entrada estándar de `command2`.

Examples

Mostrar todos los procesos paginados.

```
ps -e | less
```

`ps -e` muestra todos los procesos, su salida está conectada a la entrada de más vía `|`, `less` pagina los resultados.

Utilizando `&`

`&` conecta la salida estándar y el error estándar del primer comando al segundo mientras que `|` solo conecta la salida estándar del primer comando al segundo comando.

En este ejemplo, la página se descarga a través de `curl`. con la opción `-v` `curl` escribe alguna información en `stderr` incluida, la página descargada se escribe en `stdout`. El título de la página se puede encontrar 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, las que se envían a `stderr` porque solo la `stdout` se canaliza al siguiente comando. En este ejemplo, todas las líneas, excepto la última (Google), se enviaron 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_C03o0IM-
EopOSKL0mMITEagIE816G55L2wrTlQwgXkhq4ApFvvYEoaWF-
oEoq2T0sBTuQVdsIFULj9b2O8X35O0sAgUnc3a3JnTRBqelMcuS9QkQA; 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
```

Modificar la salida continua de un comando.

```
~$ ping -c 1 google.com # unmodified output
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]\+[^()]\+' # modified output
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 canalización (`|`) conecta el `stdout` de `ping` al `stdin` de `grep` , que lo procesa inmediatamente. Algunos otros comandos, como `sed` default, almacenan en búfer su `stdin` , lo que significa que tiene que recibir suficientes datos, antes de que se imprima algo, lo que podría causar demoras en el procesamiento posterior.

Lea Oleoductos en línea: <https://riptutorial.com/es/bash/topic/5485/oleoductos>

Capítulo 47: Paralela

Introducción

Los trabajos en GNU Linux pueden ser paralelos utilizando GNU en paralelo. Un trabajo puede ser un solo comando o un pequeño script que debe ejecutarse para cada una de las líneas en la entrada. La entrada típica es una lista de archivos, una lista de hosts, una lista de usuarios, una lista de URL o una lista de tablas. Un trabajo también puede ser un comando que se lee desde una canalización.

Sintaxis

1. paralelo [opciones] [comando [argumentos]] <list_of_arguments>

Parámetros

Opción	Descripción
-jn	Ejecutar n trabajos en paralelo
-k	Mantener el mismo orden
-X	Múltiples argumentos con reemplazo de contexto
--colsep regexp	Entrada dividida en expresiones regulares para reemplazos posicionales
{ } { . } { / } { / . } { # }	Cuerdas de repuesto
{ 3 } { 3 . } { 3 / } { 3 / . }	Cuerdas de reemplazo posicional
-S sshlogin	Example: foo@server.example.com
--trc {} .bar	Taquigrafía para --transfer --return {} .bar --cleanup
--onall	Ejecuta el comando dado con argumento en todos los sshlogins
--nonall	Ejecute el comando dado sin argumentos en todos los sshlogins
--pipe	Dividir stdin (entrada estándar) a múltiples trabajos.
--recend str	Registre el separador final para --pipe.
--recstart str	Registre el separador de inicio para --pipe.

Examples

Paralelizar tareas repetitivas en la lista de archivos

Muchos trabajos repetitivos se pueden realizar de manera más eficiente si utiliza más recursos de su computadora (es decir, CPU y RAM). A continuación se muestra un ejemplo de ejecución de varios trabajos en paralelo.

Supongamos que tiene una `< list of files >` , digamos el resultado de `ls` . Además, deje que estos archivos estén comprimidos en bz2 y el siguiente orden de tareas debe operarse en ellos.

1. Descomprima los archivos bz2 usando `bzcat` para stdout
2. Líneas de grep (p. Ej., Filtro) con palabras clave específicas que utilizan `grep <some key word>`
3. Canalice la salida para que se concatene en un solo archivo comprimido con `gzip` usando `gzip`

Ejecutar esto usando un bucle while puede verse así

```
filenames="file_list.txt"
while read -r line
do
  name="$line"
  ## grab lines with puppies in them
  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 la cantidad de archivos y el tamaño de los archivos es grande. Los trabajos se inician en `parallel` , la opción `-j 3` inicia 3 trabajos en paralelo y la entrada a los trabajos en paralelo se realiza mediante `:::` . La salida finalmente se canaliza a `gzip > output.gz`

Paralelizar STDIN

Ahora, imaginemos que tenemos 1 archivo grande (por ejemplo, 30 GB) que se debe convertir, línea por línea. Digamos que tenemos un script, `convert.sh` , que hace esto `<task>` . Podemos canalizar el contenido de este archivo a stdin para que lo tomemos en paralelo y lo *trabajemos* en *fragmentos* como:

```
<stdin> | parallel --pipe --block <block size> -k <task> > output.txt
```

donde `<stdin>` puede originarse desde algo como `cat <file>` .

Como ejemplo reproducible, nuestra tarea será `nl -n rz` . Toma cualquier archivo, el mío será `data.bz2` , y `data.bz2 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 `output.gz nl` solo como prueba de concepto de que la salida final `output.gz` se guardará en el orden en que se recibió. Luego, el `parallel` divide el `<stdin>` en trozos de tamaño 10 MB, y por cada fragmento lo pasa a través de `nl -n rz` donde solo agrega un número justificado (consulte `nl --help` para obtener más detalles). Las opciones `--pipe` dice en `parallel` a dividir `<stdin>` en varios trabajos y `--block` especifica el tamaño de los bloques. La opción `-k` especifica que se debe mantener el orden.

Tu salida final debería verse como

```
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 línea original que se pasó a `parallel` en partes. Debe observar que el orden en la segunda columna (y el resto del archivo) se mantiene.

Lea Paralela en línea: <https://riptutorial.com/es/bash/topic/10778/paralela>

Capítulo 48: Patrón de coincidencia y expresiones regulares

Sintaxis

- \$ shopt -u opción # Desactivar la 'opción' incorporada de Bash
- Opción \$ shopt -s # Activar la 'opción' incorporada de Bash

Observaciones

Clases de personajes

Las clases de caracteres válidos para `[]` glob están definidas por el estándar POSIX:

alnum alfa ascii en blanco cntrl dígito gráfico letra inferior espacio punteado palabra superior xdigit

Dentro de `[]` se puede usar más de una clase de caracteres o rango, por ejemplo,

```
$ echo a[a-z[:blank:]0-9]*
```

coincidirá con cualquier archivo que comience con una `a` seguido de una letra minúscula o un espacio en blanco o un dígito.

Sin embargo, debe tenerse en cuenta que un globo `[]` solo puede ser completamente negado y no solo partes de él. El carácter de negación *debe* ser el primer carácter después de la apertura `[`, por ejemplo, esta expresión coincide con todos los archivos que **no** comienzan con una `a`

```
$ echo [^a]*
```

Lo siguiente hace coincidir todos los archivos que comienzan con un dígito o un `^`

```
$ echo [[:alpha:]^a]*
```

No coincide con cualquier archivo o carpeta que comienza con la letra `a` excepción de una `a` porque el `^` se interpreta como un literal `^`.

Escapar de personajes glob

Es posible que un archivo o carpeta contenga un carácter global como parte de su nombre. En este caso, se puede escapar un globo con un `\` precedente en orden para una coincidencia literal. Otro método consiste en utilizar el doble `""` o de un solo `' '` cita para abordar el archivo. Bash no procesa globos que están encerrados dentro de `""` o `' '`.

Diferencia a las expresiones regulares

La diferencia más significativa entre globs y expresiones regulares es que una expresión regular válida requiere un calificador y un cuantificador. Un calificador identifica con *qué* coincidir y un cuantificador indica con qué frecuencia debe coincidir con el calificador. El RegEx equivalente al `*` glob es `.*` Donde `.` significa cualquier carácter y `*` representa cero o más coincidencias del carácter anterior. El RegEx equivalente para el `?` glob es `.{1}`. Como antes, el calificador `.` coincide con cualquier carácter y el `{1}` indica que coincide exactamente una vez con el calificador anterior. Esto no debe confundirse con el `?` cuantificador, que coincide con cero o una vez en un RegEx. El `[]` glob se puede usar de la misma manera en un RegEx, siempre que esté seguido de un cuantificador obligatorio.

Expresiones Regulares Equivalentes

Globo	RegEx
<code>*</code>	<code>.*</code>
<code>?</code>	<code>.</code>
<code>[]</code>	<code>[]</code>

Examples

Compruebe si una cadena coincide con una expresión regular

3.0

Compruebe si una cadena consta de exactamente 8 dígitos:

```
$ date=20150624
$ [[ $date =~ ^[0-9]{8}$ ]] && echo "yes" || echo "no"
yes
$ date=hello
$ [[ $date =~ ^[0-9]{8}$ ]] && echo "yes" || echo "no"
no
```

El `*` glob

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 globo más utilizado. Simplemente coincide con cualquier cadena

```
$ echo *acy
macy stacy tracy
```

Un solo `*` no coincidirá con los archivos y carpetas que residen en subcarpetas

```
$ echo *
emptyfolder folder macy stacy tracy
$ echo folder/*
folder/anotherfolder folder/subfolder
```

El `**` glob

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 solo globo. Con la opción `globstar` activada, se puede usar para hacer coincidir carpetas que residen más profundamente en la estructura del directorio.

```
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 `**` se puede pensar en una expansión del camino, sin importar qué tan profundo sea el camino. Este ejemplo coincide con cualquier archivo o carpeta que comience con `deep`, sin importar qué tan profundo esté anidado:

```
$ echo **/deep*
folder/anotherfolder/content/deepfolder folder/subfolder/content/deepfolder
```

Los `?` glob

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 personaje

```
$ echo ?acy
macy
$ echo ??acy
stacy tracy
```

El `[]` glob

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 existe la necesidad de hacer coincidir caracteres específicos, se puede usar `[]`. Cualquier carácter dentro de `[]` se comparará exactamente una vez.

```
$ echo [m]acy
macy
$ echo [st][tr]acy
stacy tracy
```

El `[]` glob, sin embargo, es más versátil que eso. También permite una coincidencia negativa e incluso gamas coincidentes de caracteres y clases de caracteres. Una coincidencia negativa se logra utilizando `!` o `^` como el primer carácter que sigue a `[]`. Podemos igualar `stacy` por

```
$ echo [!t][^r]acy
stacy
```

Aquí le estamos diciendo a bash que queremos hacer coincidir solo 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 combinarse separando un par de caracteres con un guión (-). Cualquier personaje que se encuentre entre esos dos caracteres adjuntos, inclusive, será igualado. Por ejemplo, `[rt]` es equivalente a `[rst]`

```
$ echo [r-t][r-t]acy
stacy tracy
```

Las clases de caracteres se pueden hacer coincidir con `[:class:]` , por ejemplo, para hacer coincidir los archivos que contienen un espacio en blanco

```
$ echo *[:blank:]*
file with space
```

Coincidencia 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 integrada de Bash *dotglob* permite hacer coincidir archivos y carpetas ocultos, es decir, archivos y carpetas que comienzan con `.`

```
$ shopt -s dotglob
$ echo *
file with space folder .hiddenfile macy stacy tracy
```

Coincidencia insensible a mayúsculas

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
```

Establecer la opción `nocaseglob` coincidirá con el globo de una manera no sensible a mayúsculas y minúsculas

```
$ echo M*
M*
$ shopt -s nocaseglob
$ echo M*
macy
```

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 globo no coincida con nada, el resultado está determinado por las opciones `nullglob` y `failglob` . Si ninguno de ellos está configurado, Bash devolverá el globo si no coincide.

```
$ echo no*match
no*match
```

Si `nullglob` está activado, entonces no se devuelve nada (`null`):

```
$ shopt -s nullglob
$ echo no*match

$
```

Si se activa `failglob` se `failglob` un mensaje de error:

```
$ shopt -s failglob
$ echo no*match
bash: no match: no*match
$
```

Tenga en cuenta que la opción `failglob` reemplaza a la opción `nullglob` , es decir, si están configurados `nullglob` y `failglob` , entonces, en caso de no coincidir, se devuelve un error.

Globo extendido

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 de `extglob` puede extender las capacidades de correspondencia de un globo

```
shopt -s extglob
```

Los siguientes sub-patrones comprenden globos extendidos válidos:

- `?(pattern-list)` : coincide con cero o una aparición de los patrones dados
- `*(pattern-list)` : coincide con cero o más apariciones de los patrones dados
- `+(pattern-list)` - Coincide con una o más apariciones de los patrones dados
- `@(pattern-list)` - Coincide con uno de los patrones dados
- `!(pattern-list)` - Coincide con cualquier cosa excepto con uno de los patrones dados

La `pattern-list` es una lista de globos separados por `|`.

```
$ echo *([r-t])acy
stacy tracy

$ echo *([r-t]|m)acy
macy stacy tracy

$ echo ?([a-z])acy
macy
```

La `pattern-list` sí puede ser otro globo extendido anidado. En el ejemplo anterior, hemos visto que podemos hacer coincidir `tracy` y `stacy` con `*([r-t])`. Este globo extendido en sí mismo puede usarse dentro del globo extendido negado `!(pattern-list)` para hacer coincidir `macy`

```
$ echo !(*([r-t]))acy
macy
```

Coincide con cualquier cosa que **no** comience con cero o más apariciones de las letras `r`, `s` y `t`, lo que deja solo la coincidencia `macy` posible.

Coincidencia de expresiones regulares

```
pat='[^0-9]+([0-9]+)'
```

```
s='I am a string with some digits 1024'
[[ $s =~ $pat ]] # $pat must be unquoted
echo "${BASH_REMATCH[0]}"
echo "${BASH_REMATCH[1]}"
```

Salida:

```
I am a string with some digits 1024
1024
```

En lugar de asignar la expresión regular a una variable (`$pat`) también podríamos hacer:

```
[[ $s =~ [^0-9]+([0-9]+) ]]
```

Explicación

- La construcción `[[$s =~ $pat]]` realiza la coincidencia de expresiones regulares
- Los grupos capturados, es decir, los resultados de las coincidencias están disponibles en una matriz denominada `BASH_REMATCH`
- El índice 0 en la matriz `BASH_REMATCH` es la coincidencia total
- El índice *i*'th en la matriz `BASH_REMATCH` es el *i*'th grupo capturado, donde *i* = 1, 2, 3 ...

Obtener grupos capturados de una coincidencia de expresiones regulares contra una cadena

```
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
```

Lea Patrón de coincidencia y expresiones regulares en línea:

<https://riptutorial.com/es/bash/topic/3795/patron-de-coincidencia-y-expresiones-regulares>

Capítulo 49: Patrones de diseño

Introducción

Lograr algunos patrones de diseño comunes en Bash

Examples

El patrón de publicación / suscripción (Pub / Sub)

Cuando un proyecto Bash se convierte en una biblioteca, puede resultar difícil agregar una nueva funcionalidad. Los nombres de funciones, las variables y los parámetros generalmente deben cambiarse en los scripts que los utilizan. En escenarios como este, es útil desacoplar el código y usar un patrón de diseño dirigido por eventos. En dicho patrón, un script externo puede suscribirse a un evento. Cuando ese evento se activa (publica), el script puede ejecutar el código que registró con el evento.

pubsub.sh:

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

#
# Save the path to this script's directory in a global env variable
#
DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"

#
# Array that will contain all registered events
#
EVENTS=()

function action1() {
    echo "Action #1 was performed ${2}"
}

function action2() {
    echo "Action #2 was performed"
}

#
# @desc    :: Registers an event
# @param   :: string $1 - The name of the event. Basically an alias for a function name
# @param   :: string $2 - The name of the function to be called
# @param   :: string $3 - Full path to script that includes the function being called
#
function subscribe() {
    EVENTS+=("${1}"; "${2}"; "${3}")
}

#
# @desc    :: Public an event
# @param   :: string $1 - The name of the event being published
```

```

#
function publish() {
    for event in ${EVENTS[@]}; do
        local IFS=";"
        read -r -a event <<< "$event"
        if [[ "${event[0]}" == "${1}" ]]; then
            ${event[1]} "$@"
        fi
    done
}

#
# Register our events and the functions that handle them
#
subscribe "/do/work" "action1" "${DIR}"
subscribe "/do/more/work" "action2" "${DIR}"
subscribe "/do/even/more/work" "action1" "${DIR}"

#
# Execute our events
#
publish "/do/work"
publish "/do/more/work"
publish "/do/even/more/work" "again"

```

Correr:

```

chmod +x pubsub.sh
./pubsub.sh

```

Lea Patrones de diseño en línea: <https://riptutorial.com/es/bash/topic/9531/patrones-de-diseno>

Capítulo 50: Personalizando PS1

Examples

Cambiar el indicador de PS1

Para cambiar PS1, solo tienes que cambiar el valor de la variable de shell PS1. El valor se puede configurar en el archivo `~/.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 una serie de caracteres especiales de escape de barra invertida:

Formato	Acción
<code>\a</code>	un personaje de campana ASCII (07)
<code>\d</code>	la fecha en el formato "Fecha del mes de la semana" (por ejemplo, "Mar 26 de mayo")
<code>\D{format}</code>	el formato se pasa a <code>strftime</code> (3) y el resultado se inserta en la cadena de solicitud; un formato vacío da como resultado una representación temporal específica del entorno local. Los tirantes son necesarios.
<code>\e</code>	un personaje 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 actualmente gestionados por el shell
<code>\l</code>	el nombre base del dispositivo terminal del shell
<code>\n</code>	nueva línea
<code>\r</code>	retorno de carro
<code>\s</code>	el nombre del shell, el nombre base de \$ 0 (la parte que sigue a la barra final)
<code>\t</code>	la hora actual en formato HH: MM: SS de 24 horas
<code>\T</code>	la hora actual en formato HH: MM: SS de 12 horas
<code>\@</code>	la hora actual en formato de 12 horas a.m./pm

Formato	Acción
\A	la hora actual en formato HH: MM de 24 horas
\u	el nombre de usuario del usuario actual
\v	la versión de bash (por ejemplo, 2.00)
\V	el lanzamiento de bash, versión + nivel de parche (por ejemplo, 2.00.0)
\w	el directorio de trabajo actual, con \$ HOME abreviado con una tilde
\W	el nombre base del directorio de trabajo actual, con \$ HOME abreviado con una tilde
\!	el número histórico de este comando
\#	el número de comando de este comando
\\$	si el UID efectivo es 0, un #, de lo contrario un \$
\nnn*	el carácter correspondiente al número octal nnn
\	una barra invertida
\[comenzar una secuencia de caracteres no imprimibles, que podrían utilizarse para incrustar una secuencia de control de terminal en el indicador
\]	terminar una secuencia de caracteres no imprimibles

Entonces, por ejemplo, podemos configurar PS1 para:

```
PS1="\u@\h:\w\$ "
```

Y saldrá:

usuario @ máquina: ~ \$

Mostrar una rama git usando PROMPT_COMMAND

Si está dentro de una carpeta de un repositorio de git, puede ser bueno mostrar la rama actual en la que se encuentra. En ~/.bashrc o /etc/bashrc agregue lo siguiente (se requiere git para que esto funcione):

```
function prompt_command {
    # Check if we are inside a git repository
    if git status > /dev/null 2>&1; then
        # Only get the name of the branch
        export GIT_STATUS=$(git status | grep 'On branch' | cut -b 10-)
    else
        export GIT_STATUS=""
    fi
}
```



```

fi
}
# This function gets called every time PS1 is shown
PROMPT_COMMAND=prompt_command

PS1="\$GIT_STATUS \u@\h:\w\$ "

```

Si estamos en una carpeta dentro de un repositorio de git, esto generará:

usuario de la rama @ máquina: ~ \$

Y si estamos dentro de una carpeta normal:

usuario @ máquina: ~ \$

Mostrar el nombre de la rama git en el indicador de terminal

Puedes tener funciones en la variable PS1, solo asegúrate de citarla o usar escape para caracteres especiales:

```

gitPS1() {
    gitpsl=$(git branch 2>/dev/null | grep '*')
    gitpsl="${gitpsl:+ (${gitpsl/#\* /})}"
    echo "$gitpsl"
}

PS1='\u@\h:\w$(gitPS1)$ '

```

Te dará un mensaje como este:

```
user@Host:/path (master)$
```

Notas:

- Realice los cambios en `~/.bashrc` o `/etc/bashrc` o `~/.bash_profile` o `~/.profile` (dependiendo del sistema operativo) y guárdelo.
- Ejecute `source ~/.bashrc` (específico de la distribución) después de guardar el archivo.

Mostrar hora en terminal

```

timeNow() {
    echo "$(date +%r)"
}

PS1='[$(timeNow)] \u@\h:\w$ '

```

Te dará un mensaje como este:

```
[05:34:37 PM] user@Host:/path$
```

Notas:

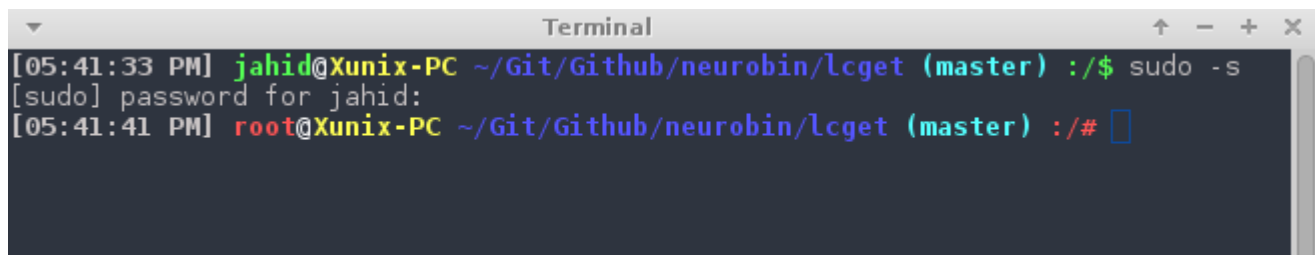
- Realice los cambios en `~/.bashrc` o `/etc/bashrc` o `~/.bash_profile` o `~/profile` (dependiendo del sistema operativo) y guárdelo.
- Ejecute `source ~/.bashrc` (específico de la distribución) después de guardar el archivo.

Colorear y personalizar el indicador de terminal

Así es como el autor configura su variable personal de `PS1` :

```
gitPS1(){
    gitpsl=$(git branch 2>/dev/null | grep '*')
    gitpsl="${gitpsl:+ ($gitpsl/#\* /)}"
    echo "$gitpsl"
}
#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 mensaje:



Referencia de color:

```
# Colors
txtblk='\e[0;30m' # Black - Regular
txtrd='\e[0;31m' # Red
txtgrn='\e[0;32m' # Green
txtylw='\e[0;33m' # Yellow
txtblu='\e[0;34m' # Blue
txtpur='\e[0;35m' # Purple
txtcyn='\e[0;36m' # Cyan
xtwht='\e[0;37m' # White
```

```

bldblk='\e[1;30m' # Black - Bold
bldred='\e[1;31m' # Red
bldgrn='\e[1;32m' # Green
bldylw='\e[1;33m' # Yellow
bldblu='\e[1;34m' # Blue
bldpur='\e[1;35m' # Purple
bldcyn='\e[1;36m' # Cyan
bldwht='\e[1;37m' # White
unkblk='\e[4;30m' # Black - Underline
undred='\e[4;31m' # Red
undgrn='\e[4;32m' # Green
undylw='\e[4;33m' # Yellow
undblu='\e[4;34m' # Blue
undpur='\e[4;35m' # Purple
undcyn='\e[4;36m' # Cyan
undwht='\e[4;37m' # White
bakblk='\e[40m' # Black - Background
bakred='\e[41m' # Red
badgrn='\e[42m' # Green
bakylw='\e[43m' # Yellow
bakblu='\e[44m' # Blue
bakpur='\e[45m' # Purple
bakcyn='\e[46m' # Cyan
bakwht='\e[47m' # White
txtrst='\e[0m' # Text Reset

```

Notas:

- Realice los cambios en `~/.bashrc` o `/etc/bashrc` o `~/.bash_profile` o `~/profile` (dependiendo del sistema operativo) y guárdelo.
- Para `root` también puede que necesite editar el archivo `/etc/bash.bashrc` o `/root/.bashrc`
- Ejecute `source ~/.bashrc` (específico de la distribución) después de guardar el archivo.
- Nota: si ha guardado los cambios en `~/.bashrc`, recuerde agregar la `source ~/.bashrc` en su `~/.bash_profile` para que este cambio en PS1 se registre cada vez que se inicie la aplicación Terminal.

Mostrar el estado y el tiempo de retorno del comando anterior

A veces necesitamos una sugerencia visual para indicar el estado de retorno del comando anterior. El siguiente fragmento de código lo pone a la cabeza de la PS1.

Tenga en cuenta que la función `__stat()` debe llamarse cada vez que se genere un nuevo PS1, o de lo contrario se mantendría en el estado de retorno del último comando de su `.bashrc` o `.bash_profile`.

```

# -ANSI-COLOR-CODES- #
Color_Off="\033[0m"
###-Regular-###
Red="\033[0;31m"
Green="\033[0;32m"
Yellow="\033[0;33m"
####-Bold-####

```

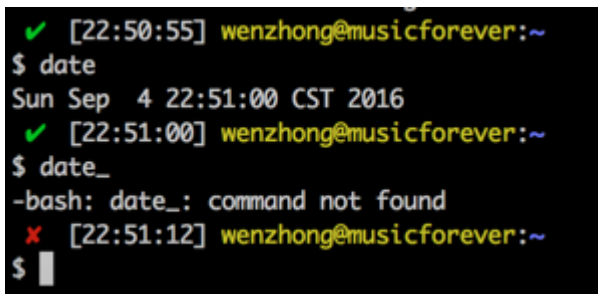
```

function __stat() {
    if [ $? -eq 0 ]; then
        echo -en "$Green ✓ $Color_Off "
    else
        echo -en "$Red ✗ $Color_Off "
    fi
}

PS1='$(__stat)'
PS1+='\n'
PS1+='\e[0;33m\u@\h\e[0m:\e[1;34m\w\e[0m\n$ '

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:~
$

```

Lea Personalizando PS1 en línea: <https://riptutorial.com/es/bash/topic/3340/personalizando-ps1>

Capítulo 51: Proceso de sustitución

Observaciones

La sustitución de procesos es una forma de redirección donde la entrada o salida de un proceso (alguna secuencia de comandos) aparece como un archivo temporal.

Examples

Compara dos archivos de la web.

Lo siguiente compara dos archivos con `diff` mediante la sustitución de procesos en lugar de crear archivos temporales.

```
diff <(curl http://www.example.com/page1) <(curl http://www.example.com/page2)
```

Alimenta un bucle while con la salida de un comando

Esto alimenta un `while` de bucle con la salida de un `grep` comando:

```
while IFS=":" read -r user _
do
    # "$user" holds the username in /etc/passwd
done < <(grep "hello" /etc/passwd)
```

Con comando de pegar

```
# Process substitution with paste command is common
# To compare the contents of two directories
paste <(ls /path/to/directory1) <(ls /path/to/directory1)
```

Concatenando archivos

Es bien sabido que no puede usar el mismo archivo para entrada y salida en el mismo comando. Por ejemplo,

```
$ cat header.txt body.txt >body.txt
```

no hace lo que tu quieres Para cuando el `cat` lee `body.txt`, ya se ha truncado por la redirección y está vacío. El resultado final será que `body.txt` contendrá los contenidos de `header.txt`.

Uno podría pensar para evitar esto con la sustitución de procesos, es decir, que el comando

```
$ cat header.txt <(cat body.txt) > body.txt
```

forzará que los contenidos originales de `body.txt` se guarden de alguna manera en algún búfer en algún lugar antes de que el archivo sea truncado por la redirección. No funciona. El `cat` entre paréntesis comienza a leer el archivo solo después de que se hayan configurado todos los descriptores de archivo, al igual que el externo. No tiene sentido intentar usar la sustitución de procesos en este caso.

La única forma de anteponer un archivo a otro es crear uno intermedio:

```
$ cat header.txt body.txt >body.txt.new
$ mv body.txt.new body.txt
```

que es lo que hacen `sed` o `perl` o programas similares bajo la alfombra cuando se les llama con una opción de *edición en el lugar* (generalmente `-i`).

Transmitir un archivo a través de múltiples programas a la vez

Esto cuenta la cantidad de líneas en un archivo grande con `wc -l` mientras se comprime simultáneamente con `gzip`. Ambos corren concurrentemente.

```
tee >(wc -l >&2) < bigfile | gzip > bigfile.gz
```

Normalmente, `tee` escribe su entrada en uno o más archivos (y `stdout`). Podemos escribir comandos en lugar de archivos con `tee >(command)`.

Aquí el comando `wc -l >&2` cuenta las líneas leídas desde el `tee` (que a su vez se lee desde el `bigfile`). (El recuento de líneas se envía a `stderr` (`>&2`) para evitar que se mezcle con la entrada a `gzip`). La `tee` estándar de `tee` se alimenta simultáneamente a `gzip`.

Para evitar el uso de un sub-shell

Un aspecto importante de la sustitución de procesos es que nos permite evitar el uso de un sub-shell al canalizar los comandos desde el shell.

Esto se puede demostrar con un simple 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 incrementa el 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 se modificó en el contexto de 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 pierde después de que finaliza el sub-shell.

```
command &  
command | command  
( command )
```

La sustitución del proceso resolverá el problema al evitar el uso de la tubería `|` operador como en

```
count=0  
while IFS= read -r _; do  
    ((count++))  
done <<(find . -maxdepth 1 -type f -print)
```

Esto mantendrá el valor de la variable de `count` ya que no se invocan sub-shells.

Lea Proceso de sustitución en línea: <https://riptutorial.com/es/bash/topic/2647/proceso-de-sustitucion>

Capítulo 52: Redes con Bash

Introducción

Bash a menudo se usa en la administración y mantenimiento de servidores y clústeres. Información sobre los comandos típicos utilizados por las operaciones de red, cuándo usar qué comando para qué propósito, y ejemplos / muestras de aplicaciones únicas y / o interesantes de este deben incluirse.

Examples

Comandos de red

```
ifconfig
```

El comando anterior mostrará toda la interfaz activa de la máquina y también proporcionará la información de

1. Dirección IP asignada a la interfaz
2. Dirección MAC de la interfaz
3. Dirección de Difusión
4. Transmitir y recibir bytes

Algun ejemplo

```
ifconfig -a
```

El comando anterior también muestra la interfaz de desactivación

```
ifconfig eth0
```

El comando anterior solo 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 deshabilitará la interfaz eth0


```
ping
```

El comando anterior (Packet Internet Grouper) es 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.

```
tracert
```

El comando anterior es para usar en la resolución de problemas para averiguar la cantidad de saltos tomados para llegar al destino.

```
netstat
```

El comando anterior (estadísticas de red) proporciona la información de conexión 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 verificar la información de la ruta Netwrok. Básicamente te muestra la tabla de enrutamiento.

```
router add default gw 192.168.1.1 eth0
```

El comando anterior agregará la ruta predeterminada 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 predeterminada de la tabla de enrutamiento

Lea Redes con Bash en línea: <https://riptutorial.com/es/bash/topic/10737/redes-con-bash>

Capítulo 53: Redirección

Sintaxis

- comando `</ ruta / a / archivo #` Redirigir la entrada estándar al archivo
- comando `> / ruta / a / archivo #` Redirige la salida estándar a file
- comando `file_descriptor> / ruta / a / archivo #` Redirige la salida de file_descriptor a archivo
- comando `> & file_descriptor #` Redirige la salida a file_descriptor
- comando `file_descriptor> & another_file_descriptor #` Redirige file_descriptor a another_file_descriptor
- comando `<& file_descriptor #` Redirige file_descriptor a la entrada estándar
- comando `&> / ruta / a / archivo #` Redirige la salida estándar y el error estándar al archivo

Parámetros

Parámetro	Detalles
descriptor de archivo interno	Un entero.
dirección	Uno de <code>></code> , <code><</code> o <code><></code>
descriptor de archivo externo o ruta	<code>&</code> seguido de un entero para el descriptor de archivo o una ruta.

Observaciones

Los programas de la consola UNIX tienen un archivo de entrada y dos archivos de salida (las secuencias de entrada y salida, así como los dispositivos, son tratados como archivos por el sistema operativo). Estos son típicamente el teclado y la pantalla, respectivamente, pero cualquiera o todos pueden redirigirse para venir de, o ir a, un archivo u otro programa.

`STDIN` es una entrada estándar, y es así como el programa recibe una entrada interactiva. `STDIN` se le suele asignar el descriptor de archivo 0.

`STDOUT` es una salida estándar. Todo lo que se emita en `STDOUT` se considera el "resultado" del programa. `STDOUT` se suele asignar el descriptor de archivo 1.

`STDERR` es donde se muestran los mensajes de error. Normalmente, cuando se ejecuta un programa desde la consola, `STDERR` muestra en la pantalla y no se puede distinguir de `STDOUT` . `STDERR` se suele asignar el descriptor de archivo 2.

El orden de redireccionamiento es importante.

```
command > file 2>&1
```

Redirige ambos (`STDOUT` y `STDERR`) al archivo.

```
command 2>&1 > file
```

Los redireccionamientos que solamente `STDOUT` , debido a que el descriptor de archivo 2 se redirige al archivo apuntado por el descriptor de archivo 1 (que no es el archivo `file` sin embargo, cuando se evalúa la declaración).

Cada comando en una tubería tiene su propio `STDERR` (y `STDOUT`) porque cada uno es un proceso nuevo. Esto puede crear resultados sorprendentes si espera que una redirección afecte a toda la tubería. Por ejemplo, este comando (envuelto para legibilidad):

```
$ python -c 'import sys;print >> sys.stderr, "Python error!"' \
| cut -f1 2>> error.log
```

imprimirá "error de Python!" a la consola en lugar del archivo de registro. En su lugar, adjunte el error al comando que desea capturar:

```
$ python -c 'import sys;print >> sys.stderr, "Python error!"' 2>> error.log \
| cut -f1
```

Examples

Redireccionando salida estándar

> redirige la salida estándar (también conocida como `STDOUT`) del comando actual a un archivo u 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, de lo contrario, este archivo se trunca.

El descriptor de redirección predeterminado 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.

Redireccionando 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 está predeterminado en `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
```

Redireccionando tanto `STDOUT` como `STDERR`

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

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

```
# STDERR is redirect to STDOUT: redirected to /dev/null,
# effectually redirecting both STDERR and STDOUT to /dev/null
echo 'hello' > /dev/null 2>&1
```

4.0

Esto *puede* reducirse aún más a lo siguiente:

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

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

```
# Actual code
echo 'hello' &> /dev/null
echo 'hello' &> /dev/null 'goodbye'

# Desired behavior
echo 'hello' > /dev/null 2>&1
echo 'hello' 'goodbye' > /dev/null 2>&1

# Actual behavior
echo 'hello' &
echo 'hello' & goodbye > /dev/null
```

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

Redireccionando `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
```

Anexar vs Truncar

Truncar >

1. Crear 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. Crear archivo especificado si no existe.
2. Adjuntar archivo (escritura al final del archivo).

```
# Overwrite existing file  
$ echo "first line" > /tmp/lines  
  
# Append a second line  
$ echo "second line" >> /tmp/lines  
  
$ cat /tmp/lines  
first line  
second line
```

STDIN, STDOUT y STDERR explicados

Los comandos tienen una entrada (STDIN) y dos tipos de salidas, salida estándar (STDOUT) y 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 la [read](#) [orden interna](#) para leer una línea de STDIN.)

Repartir

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

La salida estándar se usa generalmente para la salida "normal" de un comando. Por ejemplo, `ls` lista los 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 usa (como su nombre lo 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 en el shell por un número en lugar de un nombre:

0 = Estándar en
1 = salida estándar
2 = error estándar

De forma predeterminada, STDIN está conectado 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 solo necesita la salida estándar y todos los mensajes de error impresos en el error estándar deben suprimirse. Ahí es cuando utilizamos los descriptores `1` y `2`.

Redireccionando STDERR a / dev / null

Tomando el ejemplo anterior,

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

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

Redirigiendo múltiples comandos al mismo archivo

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

Utilizando tuberías con nombre

A veces es posible que desee generar algo por un programa e ingresarlo en otro programa, pero no puede usar una canalización estándar.

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

Usted 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 qué 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 está 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
```

La salida es:

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

Observe el primer carácter de los permisos, aparece como una canalización, no como un archivo.

Ahora hagamos algo genial.

Abra un terminal y tome nota del directorio (o cree uno para que la limpieza sea fácil) y haga una tubería.

```
mkfifo myPipe
```

Ahora vamos a poner algo en la tubería.

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

Notará que esto cuelga, el otro lado de la tubería todavía está cerrado. Abramos el otro lado de la tubería y dejemos pasar esas cosas.

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

```
cat < myPipe
```

Notará que después `hello from the other side` salir el `hello from the other side`, el programa en el primer terminal termina, al igual que en el segundo terminal.

Ahora ejecuta los comandos a la inversa. Comience con `cat < myPipe` y luego `cat < myPipe` eco de algo en él. Todavía funciona, porque un programa esperará hasta que algo se coloque en la tubería antes de terminar, porque sabe que tiene que obtener algo.

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

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

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

Más ejemplos usando tuberías con nombre:

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

```
$ { ls -l && cat file3; } >mypipe &  
$ cat <mypipe  
# Output: Prints ls -l data and then prints file3 contents on screen
```

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

```
$ ls -l >mypipe &  
$ cat file3 >mypipe &  
$ cat <mypipe  
#Output: This prints on screen the contents of mypipe.
```

`file3` se muestran los primeros contenidos de `file3` y luego se muestran los datos de `ls -l` (configuración LIFO).

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

```
$ { pipedata=$(<mypipe) && echo "$pipedata"; } &  
$ ls >mypipe
```



```
# Output: Prints the output of ls directly on screen
```

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

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

```
$ export pipedata
$ pipedata=$(mypipe) &
$ ls -l *.sh >mypipe
$ echo "$pipedata"
#Output : Prints correctly the contents of 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 bloquea debido a la invocación de un shell de fondo (`&`).

Imprimir mensajes de error a stderr

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

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

Puede funcionar para casos simples pero no es la forma habitual. En este ejemplo, el mensaje de error contaminará la salida real de la secuencia de comandos al mezclar los errores y la salida exitosa en la salida `stdout`.

En resumen, el mensaje de error debe ir a `stderr` 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 la `stdout` mientras que el mensaje de error se imprimirá en `stderr`.

Una mejor manera de imprimir un mensaje de error es definir una función:

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

```
}
```

Ahora, cuando tienes que imprimir un error:

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

Redireccionamiento a direcciones de red

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 al menos los resultados.

Por ejemplo, para enviar una solicitud web simple uno 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 predeterminada de `www.google.com` se imprimirán a la `stdout` estándar.

similar

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

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

Lea Redireccion en línea: <https://riptutorial.com/es/bash/topic/399/redireccion>

Capítulo 54: Salida de script en color (multiplataforma)

Observaciones

`tput` consulta la base de datos terminfo para obtener información dependiente del terminal.

De [tput en Wikipedia](#) :

En informática, `tput` es un comando estándar del sistema operativo Unix que hace uso de las capacidades del terminal.

Dependiendo del sistema, `tput` usa la base de datos terminfo o termcap, así como también busca en el entorno el tipo de terminal.

de [Bash Prompt HOWTO: Capítulo 6. Secuencias de escape ANSI: colores y movimiento del cursor](#) :

- **`tput setab [1-7]`**
 - Establecer un color de fondo utilizando ANSI escape
- **`tput setb [1-7]`**
 - Establecer un color de fondo
- **`tput setaf [1-7]`**
 - Establecer un color de primer plano utilizando ANSI escape
- **`tput setf [1-7]`**
 - Establecer un color de primer plano
- **`negrita`**
 - Establecer el modo en negrita
- **`tput sgr0`**
 - Desactivar todos los atributos (no funciona como se esperaba)

Examples

`color-output.sh`

En la sección inicial de un script de bash, es posible definir algunas variables que funcionan como

ayudantes para colorear o formatear la salida del terminal durante la ejecución del script.

Diferentes plataformas utilizan diferentes secuencias de caracteres para expresar el color. Sin embargo, hay una utilidad llamada `tput` que funciona en todos los sistemas * nix y devuelve cadenas de color de terminal específicas de la plataforma a través de una API consistente multiplataforma.

Por ejemplo, para almacenar la secuencia de caracteres que convierte el texto del terminal en rojo o verde:

```
red=$(tput setaf 1)
green=$(tput setaf 2)
```

O bien, para almacenar la secuencia de caracteres que restablece el texto a la apariencia predeterminada:

```
reset=$(tput sgr0)
```

Luego, si la secuencia de comandos BASH necesitaba mostrar diferentes salidas de color, esto se puede lograr con:

```
echo "${green}Success!${reset}"
echo "${red}Failure.${reset}"
```

Lea Salida de script en color (multiplataforma) en línea:

<https://riptutorial.com/es/bash/topic/6670/salida-de-script-en-color--multiplataforma->

Capítulo 55: Script Shebang

Sintaxis

- Use `/bin/bash` como el intérprete de bash:

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

- Busque el intérprete de bash en la `PATH` entorno `PATH` con ejecutable `env` :

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

Observaciones

Un error común es tratar de ejecutar los archivos de script `\r\n` formato de línea final de Windows en sistemas UNIX / Linux, en este caso el intérprete de script usado en el shebang es:

```
/bin/bash\r
```

Y no se encuentra, pero puede ser difícil de entender.

Examples

Shebang directo

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

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

La ruta `bash` en el shebang se resuelve y se usa solo si un script se inicia directamente así:

```
./script.sh
```

El script debe tener permiso de ejecución.

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

```
bash script.sh
```

Env Shebang

Para ejecutar un archivo de script con el ejecutable de `bash` encontrado en la `PATH` entorno `PATH` mediante el `env` ejecutable, la **primera línea** de un archivo de script debe indicar la ruta absoluta

al ejecutable de `env` con el argumento `bash` :

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

La ruta `env` en el shebang se resuelve y se usa solo si un script se inicia directamente así:

```
script.sh
```

El script debe tener permiso de ejecución.

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

```
bash script.sh
```

Otros shebangs

Hay dos tipos de programas que el kernel conoce. Un programa binario se identifica por su ELF (**E**xtenable **L**oadable **F**ormato) cabecera, que se produce generalmente por un compilador. El segundo son guiones de cualquier tipo.

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

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

Bash intenta ejecutar su argumento "algo mal" que no existe. El nombre del archivo de script también se agrega. Para ver esto claramente use un **eco** shebang:

```
#!/bin/echo something wrong
# and now call this script named "thisscript" like so:
# thisscript one two
# the output will be:
something wrong ./thisscript one two
```

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

Lea Script Shebang en línea: <https://riptutorial.com/es/bash/topic/3658/script-shebang>

Capítulo 56: Scripts CGI

Examples

Método de solicitud: GET

Es bastante fácil llamar a un script CGI a través de `GET` .

Primero necesitarás la `encoded url` del script.

Entonces usted agrega un signo de interrogación `?` 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 tiene valores solo en ella.
- Las variables están separadas por `&`
- La longitud total de la cadena no debe superar los **255** caracteres
- Los nombres y valores deben estar codificados en html (reemplace: `</, /?: @ & = + $`)

Insinuación:

Cuando se utiliza **html-forms**, el método de solicitud puede generarse por sí mismo.

Con **Ajax** 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 a través **del Intercambio de recursos de origen cruzado (CORS)** solamente, para que la solicitud sea más segura. En este escaparate usamos **CORS** para determinar el tipo de `Data-Type` que queremos usar.

Hay muchos `Data-Types` que podemos elegir, los más comunes son ...

- **texto / html**
- **Texto sin formato**
- **aplicación / json**

Al enviar una solicitud, el servidor también creará muchas variables de entorno. Por ahora, las variables de entorno más importantes son `$REQUEST_METHOD` y `$QUERY_STRING` .

El **Método de Solicitud** tiene que ser `GET` nada más!

La **cadena de consulta** incluye todos los `html-encoded data` .

La secuencia de comandos

```
#!/bin/bash

# CORS is the way to communicate, so lets response to the server first
echo "Content-type: text/html"      # set the data-type we want to use
```

```

echo ""      # we dont need more rules, the empty line initiate this.

# CORS are set in stone and any communication from now on will be like reading a html-
document.
# Therefor we need to create any stdout in html format!

# create html sctructure and send it to stdout
echo "<!DOCTYPE html>"
echo "<html><head>"

# The content will be created depending on the Request Method
if [ "$REQUEST_METHOD" = "GET" ]; then

    # Note that the environment variables $REQUEST_METHOD and $QUERY_STRING can be processed
    by the shell directly.
    # One must filter the input to avoid cross site scripting.

    Var1=$(echo "$QUERY_STRING" | sed -n 's/^.*var1=([^&]*).*$/\1/p')      # read value of
"var1"
    Var1_Dec=$(echo -e $(echo "$Var1" | sed 's/+//g;s/%(..)/\\x\1/g;'))      # html decode

    Var2=$(echo "$QUERY_STRING" | sed -n 's/^.*var2=([^&]*).*$/\1/p')
    Var2_Dec=$(echo -e $(echo "$Var2" | sed 's/+//g;s/%(..)/\\x\1/g;'))

    # create content for 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>"      #
print the values to 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"      # an other environment variable
echo "</body></html>"      # close html

exit 0

```

El documento html se verá así ...

```

<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 se verá así ...

```
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 necesaria
- La Solicitud será pública y dejará una bandeja detrás.
- El tamaño de una solicitud es limitado
- Necesita protección contra Cross-Side-Scripting (XSS)

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.

Adicionalmente...

- La mayoría de la codificación y decodificación ya no es necesaria
- La URL será visible para cualquiera y debe estar codificada en url.
Los datos se enviarán por separado y, por lo tanto, se deben proteger 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 **Intercambio de recursos de origen cruzado (CORS)**.

La siguiente secuencia de comandos también mostrará dos tipos de **contenido** diferentes.

```
#!/bin/bash

exec 2>/dev/null      # We dont want any error messages be printed to stdout
trap "response_with_html && exit 0" ERR      # response with an html message when an error
occurred and close the 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

    # The environment variable $CONTENT_TYPE describes the data-type received
    case "$CONTENT_TYPE" in
        application/json)
            # The environment variable $CONTENT_LENGTH describes the size of the data
            read -n "$CONTENT_LENGTH" QUERY_STRING_POST          # read datastream

            # The following lines will prevent XSS and check for valide JSON-Data.
            # But these Symbols need to be encoded somehow before sending to this script
            QUERY_STRING_POST=$(echo "$QUERY_STRING_POST" | sed "s/'//g" | sed
's/\\$//g;s/`//g;s/\\*//g;s/\\\\//g' )          # removes some symbols (like \ * ` $ ') to prevent
XSS with Bash and SQL.
            QUERY_STRING_POST=$(echo "$QUERY_STRING_POST" | sed -e :a -e 's/<[^>]*>//g;/</N;/ba')
# removes most html declarations to prevent XSS within documents
            JSON=$(echo "$QUERY_STRING_POST" | jq .)             # json encode - This is a pretty save
way to check for valide json code
            ;;
        *)
            response_with_html
            exit 0
            ;;
    esac

else
    response_with_html
    exit 0
fi

# Some Commands ...

response_with_json

exit 0

```

Obtendrá {"message":"Hello World!"} Como respuesta cuando envíe **datos JSON** a través de POST a este script. Todo lo demás recibirá el documento html.

Importante es también el variable `$JSON` . Esta variable está libre de XSS, pero aún podría tener valores incorrectos y debe verificarse primero. Por favor tenlo en mente.

Este código funciona de manera similar sin JSON.

Usted podría obtener cualquier información de esta manera.

Solo necesita cambiar el tipo de 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 menos importante, no olvide responder a todas las solicitudes, de lo contrario, los programas de terceros no sabrán si tuvieron éxito.

Lea Scripts CGI en línea: <https://riptutorial.com/es/bash/topic/9603/scripts-cgi>

Capítulo 57: Secuencia de ejecución de archivo

Introducción

`.bash_profile` , `.bash_login` , `.bashrc` , y `.profile` hacen casi lo mismo: configurar y definir funciones, variables y los ordenamientos.

La principal diferencia es que se llama a `.bashrc` en la apertura de una ventana que no es de inicio de sesión pero interactiva, y se llama a `.bash_profile` y los demás para un shell de inicio de sesión. Muchas personas tienen su `.bash_profile` o llamada similar `.bashrc` todos modos.

Observaciones

Otros archivos de la nota son:

- `/etc/profile` , para el código de inicialización de todo el sistema (no específico del usuario).
- `.bash_logout` , se activa al cerrar sesión (piensa cosas de limpieza)
- `.inputrc` , similar a `.bashrc` pero para readline.

Examples

`.profile` vs `.bash_profile` (y `.bash_login`)

`.profile` es leído por la mayoría de los shells en el inicio, incluyendo bash. Sin embargo, `.bash_profile` se usa para configuraciones específicas de bash. Para el código de inicialización general, póngalo en `.profile` . Si es específico para bash, use `.bash_profile` .

`.profile` no está realmente diseñado para bash específicamente, en cambio, `.bash_profile` es. (`.profile` es para Bourne y otros shells similares, que bash está basado en fuera) Bash volverá a caer en `.profile` si no se encuentra `.bash_profile` .

`.bash_login` es un `.bash_login` para `.bash_profile` , si no se encuentra. Generalmente es mejor usar `.bash_profile` o `.profile` en `.profile` lugar.

Lea Secuencia de ejecución de archivo en línea:

<https://riptutorial.com/es/bash/topic/8626/secuencia-de-ejecucion-de-archivo>

Capítulo 58: Secuencias de comandos con parámetros

Observaciones

- `shift` desplaza los parámetros de posición a la izquierda de modo que `$2` convierte en `$1` , `$3` convierte en `$2` y así sucesivamente.
- `"$@"` es una matriz de todos los parámetros posicionales pasados al script / función.
- `"$*"` es una cadena compuesta por todos los parámetros posicionales pasados al script / función.

Examples

Análisis de múltiples parámetros

Para analizar una gran cantidad de parámetros, la forma preferida de hacer esto es usar un bucle *while*, una declaración de *caso*, y el *desplazamiento*.

`shift` se usa para mostrar el primer parámetro de la serie, lo que solía ser `$ 2` , ahora `$ 1` . Esto es útil para procesar argumentos uno a la vez.

```
#!/bin/bash

# Load the user defined parameters
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
```

Parámetros de acceso

Al ejecutar un script de Bash, los parámetros pasados en el 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 faltante simplemente se evalúa como una cadena vacía. La comprobación de la existencia de un parámetro se puede hacer de la siguiente manera:

```
if [ -z "$1" ]; then
    echo "No argument supplied"
fi
```

Obteniendo todos los parametros

`$@` y `$*` son formas de interactuar con todos los parámetros del script. Haciendo referencia a [la página de Bash](#), vemos que:

- `$*` : Se expande a los parámetros posicionales, comenzando desde uno. Cuando la expansión ocurre entre comillas dobles, se expande a una sola palabra con el valor de cada parámetro separado por el primer carácter de la variable especial IFS.
- `$@` : Se expande a los parámetros posicionales, comenzando desde uno. Cuando la expansión ocurre dentro de 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 verificar si

se pasa el número apropiado de argumentos:

```
if [ $# -eq 0 ]; then
    echo "No arguments supplied"
fi
```

Ejemplo 1

Recorre todos los argumentos y verifica 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 verifica si son archivos:

```
for (( i = 1; i <= $#; ++ i ))
do
    item=${@:$i:1}

    if [[ -f $item ]]; then
        echo "$item is a file"
    fi
done
```

Análisis de argumentos utilizando un bucle for

Un ejemplo simple que proporciona las opciones:

Optar	Alt. Optar	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 inválida

```
#!/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

```

Guión envoltorio

El script Wrapper es un script que envuelve otro script o comando para proporcionar funcionalidades adicionales o simplemente para hacer algo menos tedioso.

Por ejemplo, el `egrep` real 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 "$@"

```

Entonces, cuando ejecuta `egrep` en tales sistemas, en realidad está ejecutando `grep -E` con todos los argumentos reenviados.

En el caso general, si desea ejecutar un ejemplo script / comando `exmp` con otro guión `mexmp` entonces el envoltorio `mexmp` guión se verá así:

```

#!/bin/sh
exmp "$@" # Add other options before "$@"

```



```
# or
#full/path/to/exmp "$@"
```

Dividir cadena en una matriz en Bash

Digamos que tenemos un parámetro String y queremos dividirlo con una coma

```
my_param="foo,bar,bash"
```

Para dividir esta cadena por comas podemos utilizar;

```
IFS=',' read -r -a array <<< "$my_param"
```

Aquí, IFS es una variable especial llamada [Separador de campo interno](#) 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
```

Lea [Secuencias de comandos con parámetros en línea](#):

<https://riptutorial.com/es/bash/topic/746/secuencias-de-comandos-con-parametros>

Capítulo 59: Seleccionar palabra clave

Introducción

La palabra clave de selección se puede utilizar para obtener argumentos de entrada en un formato de menú.

Examples

La palabra clave seleccionada se puede usar para obtener el argumento de entrada en un formato de menú

Supongamos que desea que el `user select` palabras clave de un menú, podemos crear una secuencia de comandos 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. Observe 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 de estos elementos y se le pedirá al usuario que realice una selección. Tras la selección, se mostrará el valor, volviendo a la línea de comandos.

```
>bash select_menu.sh
1) linux
2) windows
3) mac
#? 3
mac
>
```

Lea Seleccionar palabra clave en línea: <https://riptutorial.com/es/bash/topic/10104/seleccionar-palabra-clave>

Capítulo 60: strace

Sintaxis

- `strace -c [df] [-ln] [-bexecve] [-eexpr] ... [-Oheadhead] [-Ssortby] -ppid ... / [-D] [-Evar [= val]] ... [-uusername] comando [args]`

Examples

Cómo observar las llamadas al sistema de un programa.

Para un *archivo ejecutable o comando* `exec`, ejecutando esto se enumerarán todas las llamadas del sistema:

```
$ ptrace exec
```

Para mostrar llamadas específicas al sistema use la opción `-e`:

```
$ strace -e open exec
```

Para guardar la salida en un archivo use la opción `-o`:

```
$ strace -o output exec
```

Para encontrar las llamadas al sistema que utiliza un programa activo, use la opción `-p` mientras especifica el pid [\[cómo obtener pid\]](#) :

```
$ sudo strace -p 1115
```

Para generar un informe estadístico de todas las llamadas de sistema utilizadas, use la opción `-c`:

```
$ strace -c exec
```

Lea strace en línea: <https://riptutorial.com/es/bash/topic/10855/strace>

Capítulo 61: Tipo de conchas

Observaciones

Login Shell

Un shell de inicio de sesión es aquel cuyo primer carácter del argumento cero es a, o uno iniciado con la opción `--login`. La inicialización es más completa que en un shell (sub) shell interactivo normal.

Shell interactivo

Una shell interactiva es una iniciada sin argumentos no opcionales y sin la opción `-c` cuya entrada estándar y error están conectadas a los terminales (según lo determinado por `isatty(3)`), o una iniciada con la opción `-i`. `PS1` está configurado y `$:` incluye i si bash es interactivo, lo que permite que un script de shell o un archivo de inicio prueben este estado.

Shell no interactivo

Un Shell no interactivo es un shell en el que el usuario no puede interactuar con el shell. Como en el ejemplo, un shell que ejecuta un script es siempre un shell no interactivo. De todos modos, el script todavía puede acceder a su `tty`.

Configurando un shell de inicio de sesión

Al iniciar sesión:

```
If '/etc/profile' exists, then source it.  
If '~/.bash_profile' exists, then source it,  
else if '~/.bash_login' exists, then source it,  
else if '~/.profile' exists, then source it.
```

Para shells interactivos sin inicio de sesión

Al arrancar:

```
If '~/.bashrc' exists, then source it.
```

Para shells no interactivos.

Al iniciar: si la variable de entorno `ENV` no es nula, expanda la variable y obtenga el archivo nombrado por el valor. Si Bash no se inicia en el modo Posix, busca `BASH_ENV` antes de `ENV`.

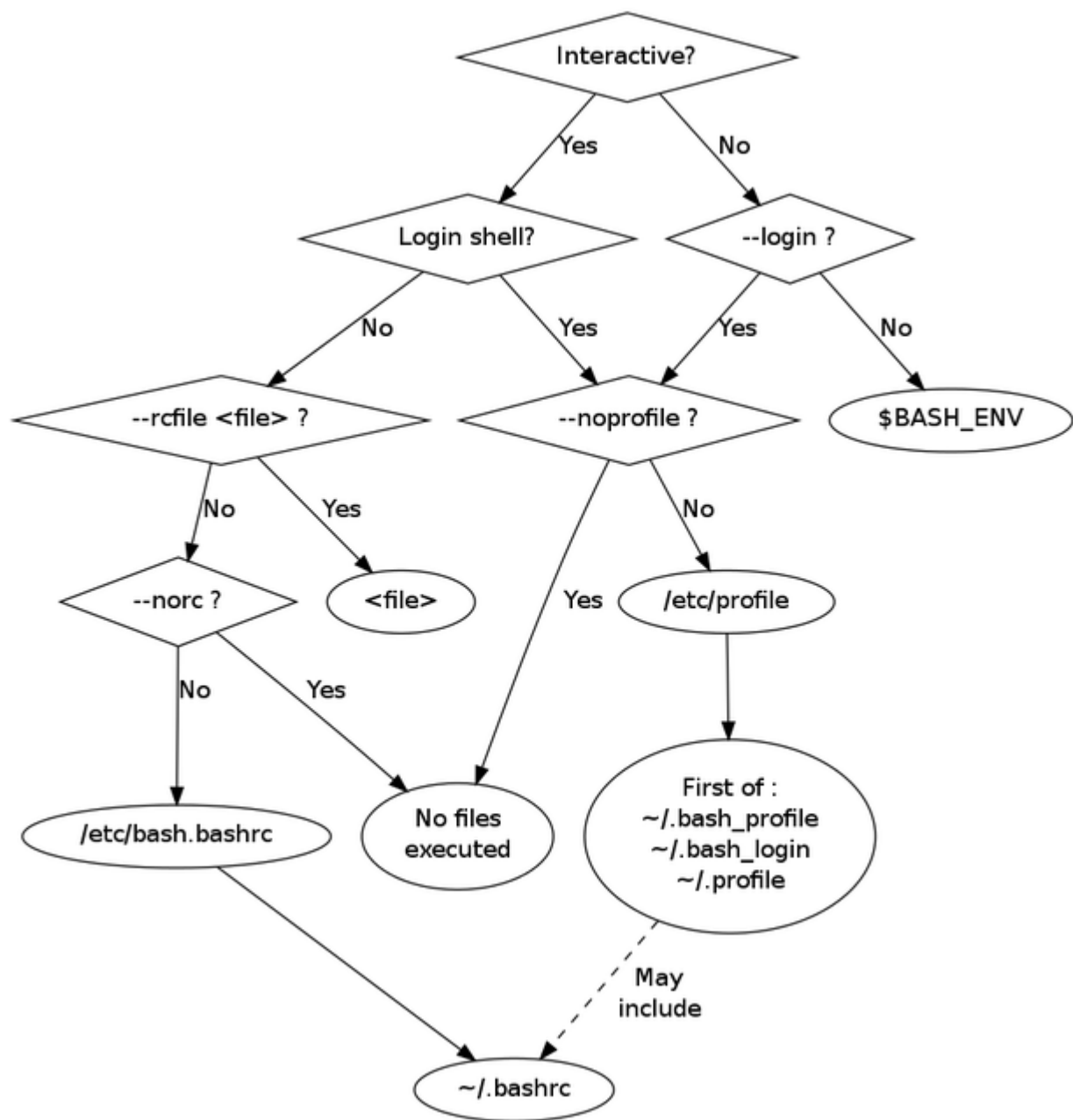
Examples

Introducción a los archivos de puntos.

En Unix, los archivos y directorios que comienzan con un período por lo general contienen configuraciones para un programa específico / una serie de programas. Los archivos de puntos generalmente están ocultos para el usuario, por lo que deberá ejecutar `ls -a` para verlos.

Un ejemplo de un archivo de puntos es `.bash_history` , que contiene los últimos comandos ejecutados, asumiendo que el usuario está utilizando Bash.

Hay varios archivos que se [obtienen](#) cuando se coloca en el shell Bash. La imagen a continuación, tomada de [este sitio](#) , muestra el proceso de decisión detrás de la elección de los archivos de origen en el inicio.



Iniciar un shell interactivo

```
bash
```

Detectar el tipo de concha

```
shopt -q login_shell && echo 'login' || echo 'not-login'
```

Lea Tipo de conchas en línea: <https://riptutorial.com/es/bash/topic/6517/tipo-de-conchas>

Capítulo 62: Trabajos en tiempos específicos

Examples

Ejecutar el trabajo una vez en un momento específico

Nota: **at** no está instalado por defecto en la mayoría de las distribuciones modernas.

Para ejecutar un trabajo una vez en otro momento que ahora, en este ejemplo a las 5 p. M.,
Puede usar

```
echo "somecommand &" | at 5pm
```

Si desea capturar la salida, puede hacerlo de la manera habitual:

```
echo "somecommand > out.txt 2>err.txt &" | at 5pm
```

at entiende muchos formatos de tiempo, por lo que también 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 da un año o una fecha, se supone que la próxima vez que ocurra la hora especificada. Entonces, si das una hora que ya pasó hoy, asumirá mañana, y si das un mes que ya pasó este año, asumirá el próximo año.

Esto también funciona junto con **nohup** como usted esperaría.

```
echo "nohup somecommand > out.txt 2>err.txt &" | at 5pm
```

Hay algunos comandos más para controlar trabajos cronometrados:

- **atq** enumera todos los trabajos cronometrados (**atq** ueue)
- **atrm** elimina un trabajo cronometrado (**atr** e **m** ove)
- **el lote** hace básicamente lo mismo que **at**, pero ejecuta trabajos solo cuando la carga del sistema es inferior a 0,8

Todos los comandos se aplican a los trabajos del usuario que inició sesión. Si ha iniciado sesión como root, los trabajos en todo el sistema se manejan, por supuesto.

Haciendo trabajos en momentos específicos repetidamente usando **systemd.timer**

systemd proporciona una implementación moderna de **cron** . Para ejecutar un script

periódicamente se necesita un servicio y un archivo temporizador. Los archivos de servicio y temporizador se deben colocar 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]
# type is important!
Type=simple
# program|script to call. Always use absolute pathes
# and redirect STDIN and STDERR as there is no terminal while being executed
ExecStart=/absolute/path/to/someCommand >>/path/to/output 2>/path/to/STDERRoutput
#NO install section!!!! Is handled by the timer facilities itself.
#[Install]
#WantedBy=multi-user.target
```

A continuación el archivo temporizador:

```
[Unit]
Description=my very first systemd timer

[Timer]
# Syntax for date/time specifications is Y-m-d H:M:S
# a * means "each", and a comma separated list of items can be given too
# *-*-* *,15,30,45:00 says every year, every month, every day, each hour,
# at minute 15,30,45 and zero seconds

OnCalendar=*-*-* *:01:00
# this one runs each hour at one minute zero second e.g. 13:01:00
```

Lea Trabajos en tiempos específicos en línea: <https://riptutorial.com/es/bash/topic/7283/trabajos-en-tiempos-especificos>

Capítulo 63: Trabajos y Procesos

Examples

Listar 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 &
```

Manejo de trabajos

Creando empleos

Para crear un trabajo, simplemente agregue un solo `&` después del comando:

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

También puede hacer que un proceso en ejecución sea un trabajo presionando `Ctrl + z`:

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

Fondo y primer plano de un proceso.

Para poner el proceso en primer plano, el comando `fg` se usa junto con `%`

```
$ sleep 10 &
[1] 20024

$ fg %1
sleep 10
```

Ahora puedes interactuar con el proceso. Para volver a ponerlo en segundo plano, puede utilizar el comando `bg`. Debido a la sesión de terminal ocupada, primero debe detener el proceso presionando `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 hay un solo proceso, o para el primer proceso en la lista. Por ejemplo:

```
$ sleep 10 &
[1] 20024

$ fg %          # to bring a process to foreground 'fg %' is also working.
sleep 10
```

o solo

```
$ %          # laziness knows no boundaries, '%' is also working.
sleep 10
```

Además, con solo escribir `fg` o `bg` sin ningún argumento, se manejará el último trabajo:

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

Matando trabajos en ejecución

```
$ sleep 10 &
[1] 20024

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

El proceso de suspensión se ejecuta en segundo plano con el ID de proceso (pid) 20024 y el número de trabajo 1. Para hacer referencia al proceso, puede usar el pid o el número de trabajo. Si usa el número de trabajo, debe prefijarlo con %. La señal de eliminación predeterminada enviada por `kill` es `SIGTERM`, que permite que el proceso de destino salga correctamente.

Algunas señales comunes de muerte se muestran a continuación. Para ver una lista completa, ejecute `kill -l`.

Nombre de la señal	Valor de señal	Efecto
SIGHUP	1	Colgar

Nombre de la señal	Valor de señal	Efecto
SIGINT	2	Interrumpir desde el teclado
SIGKILL	9	Señal de matar
SIGTERM	15	Señal de terminación

Iniciar y matar procesos específicos.

Probablemente, la forma más sencilla de eliminar un proceso en ejecución es seleccionándolo a través del nombre del proceso como en el siguiente ejemplo, usando el comando `pkill` como

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

(o) una forma más `pgrep` usar `pgrep` para buscar el identificador de proceso real

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

Se puede obtener el mismo resultado utilizando `grep` sobre `ps -ef | grep name_of_process` luego mata el proceso asociado con el pid resultante (id de proceso). La selección de un proceso con su nombre es conveniente en un entorno de prueba, pero puede ser realmente peligroso cuando se usa el script en producción: es prácticamente imposible asegurarse de que el nombre coincidirá con el proceso que realmente desea eliminar. En esos casos, el siguiente enfoque es realmente seguro.

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

```
#!/bin/bash

if [[ ! -e /tmp/test.py.pid ]]; then    # Check if the file already exists
    python test.py &                    #+and if so do not run another process.
    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` contiene el pid del proceso `python test.py`. Si el archivo ya existe, asumimos que el comando ya se está ejecutando y que el script devuelve un error.

Luego, cuando quieras matarlo usa el siguiente script:

```
#!/bin/bash

if [[ -e /tmp/test.py.pid ]]; then    # If the file do not exists, then the
```

```
kill `cat /tmp/test.py.pid`      #+the process is not running. Useless
rm /tmp/test.py.pid             #+trying to kill it.
else
    echo "test.py is not running"
fi
```

eso matará exactamente el proceso asociado con su comando, sin depender de ninguna información volátil (como la cadena utilizada para ejecutar el comando). Incluso en este caso, si el archivo no existe, la secuencia de comandos asume que desea eliminar un proceso que no se está ejecutando.

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

Listar todos los procesos

Hay dos formas comunes de enumerar todos los procesos en un sistema. Ambos listan todos los procesos que ejecutan todos los usuarios, aunque difieren en el formato que emiten (el motivo de las diferencias es histórico).

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

Esto se puede usar para verificar si una aplicación dada se está ejecutando. Por ejemplo, para verificar si el servidor SSH (sshd) se está ejecutando:

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

Compruebe qué proceso se está ejecutando en un puerto específico

Para comprobar qué proceso se está ejecutando en el puerto 8080

```
lsof -i :8080
```

Encontrar información sobre un proceso en ejecución

`ps aux | grep <search-term>` muestra los procesos que coinciden *con el término de búsqueda*

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 la identificación del proceso. Por ejemplo, si desea finalizar el proceso nginx, puede usar el comando `kill 5647` . Siempre se recomienda utilizar el comando `kill` con `SIGTERM` lugar de `SIGKILL` .

Desaprobando trabajo de fondo

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

Esto permite que un proceso de ejecución prolongada continúe una vez que su shell (terminal, ssh, etc.) esté cerrado.

Lea Trabajos y Procesos en línea: <https://riptutorial.com/es/bash/topic/398/trabajos-y-procesos>

Capítulo 64: Transferencia de archivos usando scp

Sintaxis

- `scp / some / local / directory / file_name user_name @ host_name: destination_file_path`
- `scp nombre_usuario @ nombre_host: origin_file_path / some / local / directory`

Examples

archivo de transferencia de scp

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 de inicio de `tom` usuario del `server2`.

scp transfiriendo múltiples archivos

`scp` también se puede utilizar para transferir varios archivos 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 siguiente ejemplo, todos los archivos se transferirán al directorio principal del usuario `tom`.

```
scp /my_folder/*.txt tom@server2:$HOME
```

Descargando archivo usando scp

Para descargar un archivo desde el 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` del directorio de inicio de `tom` del usuario al directorio actual de nuestra máquina local.

Lea [Transferencia de archivos usando scp en línea](https://riptutorial.com/es/bash/topic/5484/transferencia-de-archivos-usando-scp):

<https://riptutorial.com/es/bash/topic/5484/transferencia-de-archivos-usando-scp>

Capítulo 65: Usando "trampa" para reaccionar a señales y eventos del sistema

Sintaxis

- `trap action sigspec ...` # Ejecutar "action" en una lista de señales
- `trap sigspec ...` # La acción de omisión restablece las trampas para las señales

Parámetros

Parámetro	Sentido
-pag	Listar las trampas actualmente instaladas
-l	Lista de nombres de señales y números correspondientes

Observaciones

La utilidad de `trap` es un shell especial incorporado. Se [define en POSIX](#) , pero bash también agrega algunas extensiones útiles.

Los ejemplos que son compatibles con POSIX comienzan con `#!/bin/sh` , y los ejemplos que comienzan con `#!/bin/bash` usan una extensión de bash.

Las señales pueden ser un número de señal, un nombre de señal (sin el prefijo SIG) o la palabra clave especial `EXIT` .

Los garantizados por POSIX son:

Número	Nombre	Notas
0	SALIDA	Ejecutar siempre en la salida de shell, independientemente del código de salida
1	SIGHUP	
2	SIGINT	Esto es lo que <code>^C</code> envía
3	SIGQUIT	
6	SIGABRT	
9	SIGKILL	
14	SIGALRM	

Número	Nombre	Notas
15	Sigma	Esto es lo que <code>kill</code> envía por defecto

Examples

La captura de SIGINT o Ctl + C

La captura se restablece para las subshells, por lo que el `sleep` seguirá actuando sobre la señal `SIGINT` enviada por `^C` (generalmente al salir), pero el proceso principal (es decir, el script de shell) no lo hará.

```
#!/bin/sh

# Run a command on signal 2 (SIGINT, which is what ^C sends)
sigint() {
    echo "Killed subshell!"
}
trap sigint INT

# Or use the no-op command for no output
#trap : INT

# This will be killed on the first ^C
echo "Sleeping..."
sleep 500

echo "Sleeping..."
sleep 500
```

Y una variante que todavía le permite salir del programa principal presionando `^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
```

Introducción: limpiar archivos temporales

Puedes usar el comando de `trap` para "atrapar" las señales; este es el equivalente de shell de la `signal()` o `sigaction()` llamada en C y la mayoría de los otros lenguajes de programación para capturar señales.

Uno de los usos más comunes de la `trap` es limpiar archivos temporales tanto en una salida esperada como inesperada.

Desafortunadamente no hay suficientes scripts de shell hacer esto :-(


```
#!/bin/sh

# Make a cleanup function
cleanup() {
    rm --force -- "${tmp}"
}

# Trap the special "EXIT" group, which is always run when the shell exits.
trap cleanup EXIT

# Create a temporary file
tmp="$(mktemp -p /tmp tmpfileXXXXXXX)"

echo "Hello, world!" >> "${tmp}"

# No rm -f "$tmp" needed. The advantage of using EXIT is that it still works
# even if there was an error or if you used exit.
```

Acumula una lista de trabajo de trampa para ejecutar en la salida.

¿Alguna vez ha olvidado agregar una `trap` para limpiar un archivo temporal o hacer otro trabajo al salir?

¿Alguna vez has puesto una trampa que canceló otra?

Este código hace que sea fácil agregar las cosas que deben hacerse al salir de un elemento a la vez, en lugar de tener una declaración de `trap` grande en algún lugar de su código, que puede ser fácil de olvidar.

```
# on_exit and add_on_exit
# Usage:
#   add_on_exit rm -f /tmp/foo
#   add_on_exit echo "I am exiting"
#   tempfile=$(mktemp)
#   add_on_exit rm -f "$tempfile"
# Based on 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
}
```

Matando procesos infantiles en la salida

Las expresiones de captura no tienen que ser funciones o programas individuales, también pueden ser expresiones más complejas.

Al combinar los `jobs -p` y `kill` , podemos eliminar todos los procesos secundarios generados del shell en exit:

```
trap 'jobs -p | xargs kill' EXIT
```

reaccionar al cambiar el tamaño de la ventana de terminales

Hay una señal `WINCH` (WINdowCHange), que se activa cuando se cambia el tamaño de una ventana de terminal.

```
declare -x rows cols

update_size(){
    rows=$(tput lines) # get actual lines of term
    cols=$(tput cols)  # get actual columns of term
    echo DEBUG terminal window has no $rows lines and is $cols characters wide
}

trap update_size WINCH
```

Lea Usando "trampa" para reaccionar a señales y eventos del sistema en línea:

<https://riptutorial.com/es/bash/topic/363/usando--trampa--para-reaccionar-a-senales-y-eventos-del-sistema>

Capítulo 66: Usando gato

Sintaxis

- `gato [OPCIONES] ... [ARCHIVO] ...`

Parámetros

Opción	Detalles
-norte	Imprimir números de línea
-v	Mostrar caracteres no imprimibles utilizando la notación ^ y M, excepto LFD y TAB
-T	Mostrar los caracteres TAB como ^I
-MI	Mostrar caracteres de salto de línea (LF) como \$
-mi	Igual que -vE
-segundo	Número de líneas de salida no vacías, anulaciones -n
-UNA	equivalente a -vET
-s	suprimir líneas de salida vacías repetidas, s se refiere a apretar

Observaciones

`cat` puede leer tanto de archivos como de entradas estándar y los concatena a una salida estándar

Examples

Impresión del contenido de un archivo

```
cat file.txt
```

imprimirá el contenido de un archivo.

Si el archivo contiene caracteres no ASCII, puede mostrar esos caracteres simbólicamente con `cat -v`. Esto puede ser muy útil para situaciones donde los caracteres de control serían invisibles.

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

Sin embargo, muy a menudo, para el uso interactivo, es mejor usar un buscapersoas interactivo con `less` o `more`. (`less` es mucho más poderoso que `more` y se recomienda usar `less` más frecuencia que `more`).

```
less file.txt
```

Para pasar el contenido de un archivo como entrada a un comando. Un enfoque generalmente visto como mejor ([UUOC](#)) es utilizar la redirección.

```
tr A-Z a-z <file.txt # as an alternative to cat file.txt | tr A-Z a-z
```

En caso de que el contenido deba aparecer al revés desde su extremo, se puede usar el comando `tac` :

```
tac file.txt
```

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

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

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

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

Mostrar números de línea con salida

Use la `--number` de número 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 líneas vacías al contar líneas, use `--number-nonblank` , o simplemente `-b` .

```
$ cat -b file
```

```
1 line 1
2 line 2

3 line 4
4 line 5
```

Leer de entrada estándar

```
cat < file.txt
```

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

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

El comando `echo` antes | Salidas de dos líneas. El comando `cat` actúa sobre la salida para agregar números de línea.

Concatenar archivos

Este es el propósito principal del `cat` .

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

`cat` también se puede utilizar de forma similar para concatenar archivos como parte de una canalización, por ejemplo,

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

Escribir en un archivo

```
cat >file
```

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

```
cat >>file
```

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

NB: `Ctrl + D` para finalizar la escritura de texto en el terminal (Linux)

Un documento aquí se puede usar para integrar 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 arbitraria que debe aparecer sola en una línea (sin espacios en blanco al principio o al final) para indicar el final del documento aquí. Puede agregar 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, `here` se ejecutaría como un comando, y `$changed` sería sustituido con el valor de la variable `changed`, o nada, si no estaba definida).

Mostrar caracteres no imprimibles

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

por ejemplo, si ha copiado y copiado el código de la web, puede tener citas como " lugar de estándar " .

```
$ cat -v file.txt
$ cat -vE file.txt # Useful in detecting trailing spaces.
```

p.ej

```
$ echo '"      ' | cat -vE # echo | will be replaced by actual file.
M-bM-^@M-^]      $
```

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

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

Concatenar archivos comprimidos

Los archivos comprimidos por `gzip` se pueden concatenar 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 `gzip` 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 ***solo archivo*** y se descomprime como el ***único archivo*** `greeting.txt` . Contrasta esto con `tar -czf hello.txt howdy.txt > greetings.tar.gz` , que mantiene los archivos separados dentro del tarball.

Lea Usando gato en línea: <https://riptutorial.com/es/bash/topic/441/usando-gato>

Capítulo 67: Usando orden

Introducción

sort es un comando de Unix para ordenar datos en archivos en una secuencia.

Sintaxis

- ordenar [opción] nombre de archivo

Parámetros

Opción	Sentido
-u	Haz que cada línea de salida sea única.

Observaciones

Manual de usuario completo de lectura de `sort` [línea](#)

Examples

Ordenar orden de salida

`sort` comando `sort` se usa para ordenar una lista de líneas.

Entrada desde un archivo

```
sort file.txt
```

Entrada desde un comando

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

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

Haz que la salida sea única

Si cada línea de la salida debe ser única, agregue la opción `-u`.

Para mostrar el propietario de los archivos en la carpeta


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

Tipo numérico

Supongamos que tenemos este archivo:

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

Para ordenar este archivo numéricamente, use ordenar con la opción -n:

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

Esto debería ordenar el archivo de la siguiente manera:

```
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 use:

```
sort -rn file
```

Esto debería ordenar el archivo de la siguiente manera:

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

Ordenar por llaves

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 archivo usando una columna como clave, use la opción k:

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

Esto ordenará el archivo con la columna 2 como la clave:

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

Ahora si tenemos que ordenar el archivo con una clave secundaria junto con el uso de la clave principal:

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

Esto primero ordenará el archivo con la columna 2 como clave principal, y luego 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 archivo con más de 1 clave, entonces para cada opción -k necesitamos especificar dónde termina la clasificación. Entonces, -k1,1 significa iniciar la ordenación en la primera columna y terminar la ordenación en la primera columna.

-t opción

En el ejemplo anterior, el archivo tenía la pestaña delimitador predeterminada. En el caso de ordenar un archivo que tiene un delimitador no predeterminado, necesitamos la opción -t para especificar el delimitador. Supongamos que tenemos el archivo de abajo:

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

Para ordenar este archivo según la segunda columna, use:

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

Esto ordenará el archivo de la siguiente manera:

```
3.|Dumbledore
5.|Gryffindor
2.|Harry
4.|Hogwarts
```

1.|The sorting hat

Lea Usando orden en línea: <https://riptutorial.com/es/bash/topic/6834/usando-orden>

Capítulo 68: Utilidad de dormir

Introducción

El comando de reposo se puede usar para pausar un tiempo determinado.

Si desea utilizar una entrada diferente, use como esto Segundos: \$ sleep 1s (segundos es el valor predeterminado) Minutos: \$ sleep 1m Horas: \$ sleep 1h días: \$ sleep 1d

Si desea dormir por menos de un segundo, use \$ sleep 0.5 Puede usar como arriba de acuerdo a sus necesidades.

Examples

\$ dormir 1

Aquí el proceso iniciado esta llamada dormirá durante 1 segundo.

Lea Utilidad de dormir en línea: <https://riptutorial.com/es/bash/topic/10879/utilidad-de-dormir>

Capítulo 69: variables globales y locales

Introducción

De forma predeterminada, cada variable en bash es **global** para cada función, script e incluso el shell externo si está declarando sus variables dentro de un script.

Si desea que su variable sea local a una función, puede usar `local` para tener esa variable como una nueva variable que es independiente del alcance global y cuyo valor solo será accesible dentro de esa función.

Examples

Variables globales

```
var="hello"

function foo(){
    echo $var
}

foo
```

Obviamente saldrá "hola", pero esto funciona al revés también:

```
function foo() {
    var="hello"
}

foo
echo $var
```

También saldrá "hello"

Variables locales

```
function foo() {
    local var
    var="hello"
}

foo
echo $var
```

No generará nada, ya que `var` es una variable local de la función `foo`, y su valor no es visible desde fuera de ella.

Mezclando los dos juntos.

```
var="hello"

function foo(){
    local var="sup?"
    echo "inside function, var=$var"
}

foo
echo "outside function, var=$var"
```

Saldrá

```
inside function, var=sup?
outside function, var=hello
```

Lea variables globales y locales en línea: <https://riptutorial.com/es/bash/topic/9256/variables-globales-y-locales>

Capítulo 70: Variables internas

Introducción

Una visión general de las variables internas de Bash, dónde, cómo y cuándo usarlas.

Examples

Bash variables internas de un vistazo

Variable	Detalles
<code>\$*</code> / <code>\$@</code>	<p>Parámetros posicionales de la función / script (argumentos). Expandir de la siguiente manera:</p> <p><code>\$*</code> y <code>\$@</code> son lo mismo que <code>\$1 \$2 ...</code> (tenga en cuenta que generalmente no tiene sentido dejarlos sin comillas)</p> <p><code>"\$*"</code> es lo mismo que <code>"\$1 \$2 ..."</code> ¹</p> <p><code>"\$@"</code> es lo mismo que <code>"\$1" "\$2" ...</code></p> <p>1. Los argumentos están separados por el primer carácter de <code>\$IFS</code>, que no tiene que ser un espacio.</p>
<code>\$#</code>	Número de parámetros posicionales pasados al script o función
<code>\$!</code>	Id. De proceso del último comando (más a la derecha para canalizaciones) en el último trabajo puesto en segundo plano (tenga en cuenta que no es necesariamente el mismo que el Id. De grupo de proceso del trabajo cuando el control de trabajo está habilitado)
<code>\$\$</code>	ID del proceso que ejecutó <code>bash</code>
<code>\$?</code>	Estado de salida del último comando
<code>\$n</code>	Parámetros posicionales, 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 los scripts, ruta con la que se invocó el script; con <code>bash -c 'printf "%s\n" "\$0" ' name args' : name</code> (el primer argumento después del script en línea), de lo contrario, el <code>argv[0]</code> que recibió <code>bash</code> .
<code>\$_</code>	Último campo del último comando.
<code>\$IFS</code>	Separador de campo interno
<code>\$PATH</code>	Variable de entorno <code>PATH</code> usada para buscar ejecutables

Variable	Detalles
\$OLDPWD	Directorio de trabajo anterior
\$PWD	Directorio de trabajo actual
\$FUNCNAME	Conjunto de nombres de funciones en la pila de llamadas de ejecución
\$BASH_SOURCE	Array que contiene las rutas de origen para los elementos en la matriz <code>FUNCNAME</code> . Se puede utilizar para obtener la ruta del script.
\$BASH_ALIASES	Matriz asociativa que contiene todos los alias definidos actualmente
\$BASH_REMATCH	Arsenal de partidos del último partido regex
\$BASH_VERSION	Bash version string
\$BASH_VERSINFO	Una matriz de 6 elementos con información de la versión Bash
\$BASH	Ruta absoluta al shell Bash actualmente en ejecución (determinado heurísticamente por <code>bash</code> función de <code>argv[0]</code> y el valor de <code>\$PATH</code> ; puede ser incorrecto en los casos de esquina)
\$BASH_SUBSHELL	Bash subshell level
\$UID	ID de usuario real (no es efectivo si es diferente) del proceso que ejecuta <code>bash</code>
\$PS1	Línea de comando primaria; ver Uso de las variables PS *
\$PS2	Indicador de línea de comando secundario (usado para entrada adicional)
\$PS3	Indicador de línea de comando terciario (usado en el ciclo de selección)
\$PS4	Indicador de línea de comando cuaternario (usado para agregar información con salida detallada)
\$RANDOM	Un entero pseudoaleatorio entre 0 y 32767
\$REPLY	Variable utilizada por <code>read</code> por defecto cuando no se especifica ninguna variable. También se utiliza por <code>select</code> para devolver el valor proporcionado por el usuario
\$PIPESTATUS	Variable de matriz que contiene los valores de estado de salida de cada comando en la tubería de primer plano ejecutada más recientemente.

La asignación de variables no debe tener espacio antes y después. `a=123` no `a = 123` . Este último (un signo igual rodeado de espacios) en 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 para `[` o `[[` o la

prueba que sea) utilizando).

\$ BASHPID

Id. De 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
```

\$ BASH_ENV

Una variable de entorno que apunta al archivo de inicio Bash que se lee cuando se invoca un script.

\$ BASH_VERSINFO

Una matriz que contiene la información de la versión completa se divide en elementos, mucho más conveniente que \$BASH_VERSION si solo está buscando la versión principal:

```
~> $ for ((i=0; i<=5; i++)); do echo "BASH_VERSINFO[$i] = ${BASH_VERSINFO[$i]}"; done
BASH_VERSINFO[0] = 3
BASH_VERSINFO[1] = 2
BASH_VERSINFO[2] = 25
BASH_VERSINFO[3] = 1
BASH_VERSINFO[4] = release
BASH_VERSINFO[5] = x86_64-redhat-linux-gnu
```

\$ BASH_VERSION

Muestra la versión de bash que se está ejecutando, esto le permite decidir si puede usar alguna función avanzada:

```
~> $ echo $BASH_VERSION
4.1.2(1)-release
```

\$ EDITOR

El editor predeterminado que estará involucrado por cualquier script o programa, generalmente vi o emacs.

```
~> $ echo $EDITOR
vi
```

\$ FUNCNAME

Para obtener el nombre de la función actual, escriba:

```
my_function()
{
    echo "This function is $FUNCNAME"      # This will output "This function is my_function"
}
```

Esta instrucción no devolverá nada si la escribe fuera de la función:

```
my_function

echo "This function is $FUNCNAME"      # This will output "This function is"
```

\$ HOME

El directorio de inicio del usuario.

```
~> $ echo $HOME
/home/user
```

\$ HOSTNAME

El nombre de host asignado al sistema durante el inicio.

```
~> $ echo $HOSTNAME
mybox.mydomain.com
```

\$ HOSTTYPE

Esta variable identifica el hardware, puede ser útil para determinar qué binarios ejecutar:

```
~> $ echo $HOSTTYPE
x86_64
```

\$ GRUPOS

Una matriz que contiene los números de grupos en los que se encuentra 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
```

\$ IFS

Contiene la cadena del separador de campo interno que utiliza bash para dividir cadenas cuando se realiza un bucle, etc. El valor predeterminado es los caracteres de espacio en blanco: `\n` (nueva línea), `\t` (pestaña) y espacio. Cambiar esto a otra cosa te permite dividir cadenas usando diferentes caracteres:

```
IFS=","
INPUTSTR="a,b,c,d"
for field in ${INPUTSTR}; do
    echo $field
done
```

La salida de lo anterior es:

```
a
b
c
d
```

Notas:

- Esto es responsable del fenómeno conocido como [división de palabras](#) .

\$ LINENO

Muestra el número de línea en el script actual. Sobre todo útil cuando se depuran scripts.

```
#!/bin/bash
# this is line 2
echo something # this is line 3
echo $LINENO # Will output 4
```

\$ MACHTYPE

Similar a \$HOSTTYPE anterior, esto también incluye información sobre el sistema operativo, así como el hardware

```
~> $ echo $MACHTYPE
x86_64-redhat-linux-gnu
```

\$ OLDPWD

OLDPWD (**OLDP rint W** orking **D** irectory) contiene el directorio antes del último comando de `cd` :

```
~> $ cd directory
directory> $ echo $OLDPWD
/home/user
```

\$ OSTYPE

Devuelve información sobre el tipo de sistema operativo que se ejecuta en la máquina, por ejemplo.

```
~> $ echo $OSTYPE
linux-gnu
```

\$ PATH

La ruta de búsqueda para encontrar binarios para comandos. Los ejemplos comunes incluyen `/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
```

Entonces, por ejemplo, dado el `$PATH` anterior, si escribe `lss` en el indicador, 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.

\$ PPID

El ID de proceso (pid) del script o el padre del shell, es decir, el proceso que invoca el script o shell actual.

```
~> $ echo $$
13016
~> $ echo $PPID
13015
```

\$ 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
```

\$ SEGUNDOS

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
```

\$ SHELLOPTS

Se proporciona una lista de solo lectura de las opciones de bash en el inicio para controlar su comportamiento:

```
~> $ echo $SHELLOPTS  
braceexpand:emacs:hashall:histexpand:history:interactive-comments:monitor
```

\$ SHLVL

Cuando se ejecuta el comando bash, se abre un nuevo shell. La variable de entorno \$ SHLVL contiene la cantidad de niveles de shell a los que se ejecuta la shell *actual*.

En una *nueva* ventana de terminal, la ejecución del siguiente comando producirá resultados diferentes según 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 secundario de otros 2 procesos de 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 se configurará en "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
```

Uno puede ver que ejecutar el comando 'bash' (o ejecutar un script bash) abre un nuevo shell. En comparación, la fuente de un script ejecuta el código en el shell actual.

prueba1.sh

```
#!/usr/bin/env bash  
echo "Hello from test1.sh. My shell level is $SHLVL"  
source "test2.sh"
```

prueba2.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
```

\$ UID

Una variable de solo lectura que almacena el número de identificación del usuario:

```
~> $ echo $UID
12345
```

\$ 1 \$ 2 \$ 3 etc ...

Los parámetros posicionales pasados al script desde la línea de comandos o desde una función:

```
#!/bin/bash
# $n is the n'th positional parameter
echo "$1"
echo "$2"
echo "$3"
```

La salida de lo anterior es:

```
~> $ ./testscript.sh firstarg secondarg thirdarg
firstarg
secondarg
thirdarg
```

Si el número de argumentos posicionales es mayor que nueve, se deben usar llaves.

```
# "set -- " sets positional parameters
set -- 1 2 3 4 5 6 7 8 nine ten eleven twelve
# the following line will output 10 not 1 as the value of $1 the digit 1
# will be concatenated with the following 0
echo $10 # outputs 1
echo ${10} # outputs ten
# to show this clearly:
set -- arg{1..12}
echo $10
echo ${10}
```

PS

Para obtener el número de argumentos de línea de comando o parámetros posicionales, escriba:

```
#!/bin/bash
echo "$#"
```

Cuando se ejecuta con tres argumentos, el ejemplo anterior resultará con el resultado:

```
~> $ ./testscript.sh firstarg secondarg thirdarg
3
```

PS

Devolverá todos los parámetros posicionales en una sola cadena.

testscript.sh:

```
#!/bin/bash
echo "$@"
```

Ejecuta el script con varios argumentos:

```
./testscript.sh firstarg secondarg thirdarg
```

Salida:

```
firstarg secondarg thirdarg
```

PS

El ID de proceso (pid) del último trabajo ejecutado en segundo plano:

```
~> $ ls &
testfile1 testfile2
[1]+  Done                  ls
~> $ echo $!
21715
```

PS

Da salida al último campo del último comando ejecutado, útil para hacer que algo pase a otro comando:

```
~> $ ls *.sh;echo $_
testscript1.sh testscript2.sh
testscript2.sh
```

Da la ruta del script si se usa antes que cualquier otro comando:

test.sh:

```
#!/bin/bash
echo "$_"
```

Salida:

```
~> $ ./test.sh # running test.sh
./test.sh
```

Nota: esta no es una manera infalible de obtener la ruta del script

PS

El estado de salida de la última función o comando ejecutado. Por lo general, 0 significa que cualquier otra cosa indicará una falla:

```
~> $ ls *.blah;echo $?
ls: cannot access *.blah: No such file or directory
2
~> $ ls;echo $?
testfile1 testfile2
0
```

\$\$

El ID de proceso (pid) del proceso actual:

```
~> $ echo $$
13246
```

PS

"\$@" 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 [un bucle](#) a través de argumentos y gastos de argumentos con espacios.

Considere que estamos en un script que invocamos con dos argumentos, como este:

```
$ ./script.sh "1 2" "3 4"
```

Las variables \$* o \$@ se expandirán en \$1_\$2 , que a su vez se expandirán en 1_2_3_4 modo que el siguiente bucle:

```
for var in $*; do # same for var in $@; do
```



```
echo \<"$var"\>
done
```

imprimirá para ambos

```
<1>
<2>
<3>
<4>
```

Mientras que "\$*" se expandirá a "\$1_\$2" que a su vez se expandirá a "_1_2_3_4_" y así el bucle:

```
for var in "$*"; do
    echo \<"$var"\>
done
```

solo invocará `echo` una vez e imprimirá

```
<_1_2_3_4_>
```

Y finalmente, "\$@" se expandirá a "\$1" "\$2" , que se expandirá a "_1_" "_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. Tenga en cuenta que la construcción `for var in "$@"; do ...` es tan común e idiomático que es el valor predeterminado para un bucle `for` y se puede reducir `for var; do ...`.

\$ HISTSIZE

El número máximo de comandos recordados:

```
~> $ echo $HISTSIZE
1000
```

\$ RANDOM

Cada vez que se hace referencia a este parámetro, se genera un entero aleatorio entre 0 y 32767. Asignar un valor a esta variable siembra el generador de números aleatorios ([fuente](#)).

```
~> $ echo $RANDOM  
27119  
~> $ echo $RANDOM  
1349
```

Lea Variables internas en línea: <https://riptutorial.com/es/bash/topic/4797/variables-internas>

Capítulo 71: verdadero, falso y: comandos

Sintaxis

- verdadero,: - siempre devuelve 0 como código de salida.
- false: siempre devuelve 1 como código de salida.

Examples

Bucle infinito

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

0

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

0

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

Función de retorno

```
function positive() {
    return 0
}

function negative() {
    return 1
}
```

Código que siempre / nunca será ejecutado.

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

Lea verdadero, falso y: comandos en línea: <https://riptutorial.com/es/bash/topic/6655/verdadero-->

Creditos

S. No	Capítulos	Contributors
1	Empezando con Bash	4444 , Acey , Ajay Sangale , Alessandro Mascolo , Anil , Ashari , Benjamin W. , Blachshma , Bob Bagwill , Bubblepop , Burkhard , Christopher Bottoms , Colin Yang , Community , CraftedCart , Danny , Diego Torres Milano , divyum , dotancohen , fedorqui , Franck Dernoncourt , Functino , Gavyn , glenn jackman , Inanc Gumus , Ingve , intboolstring , J F , Jahid , Jean-Baptiste Yunès , JHS , Jonny Henly , I0b0 , lesmana , Marek Skiba , Matt , Matt Clark , mouviciel , Nathan Arthur , niglesias , rap-2-h , Richard Hamilton , Riker , satyanarayan rao , shloosh , Shubham , sjsam , Sundeep , Sylvain Bugat , Trevor Clarke , tripleee , user1336087 , William Pursell , WMios , Yuki Inoue , Zaz ,
2	Abastecimiento	hgiesel , JHS , Matt Clark
3	Alcance	Benjamin W. , chepner
4	Aliasing	Bostjan , Daniel Käfer , dingalapadum , glenn jackman , intboolstring , janos , JHS , Neui , tripleee , uhelp
5	Aquí los documentos y aquí las cadenas.	Ajinkya , Benjamin W. , Deepak K M , fedorqui , Iain , Jahid , janos , lanox , Stobor , uhelp
6	Aritmética de Bash	Alessandro Mascolo , Ashkan , Gavyn , Jesse Chen , user1336087
7	Arrays	Alexej Magura , Arronical , Benjamin W. , Bubblepop , chepner , Christopher Werby , codeforester , fedorqui , Grisha Levit , Jahid , janos , jerblack , John Kugelman , markjwill , Mateusz Piotrowski , NeilWang , ormaaj , RamenChef , Samik , Sk606 , UNagaswamy , Will ,
8	Atajos de teclado	Daniel Käfer , JHS , Judd Rogers , m02ph3u5 , Saqib Rokadia
9	Bash en Windows 10	Thomas Champion
10	Bash sustituciones de historia	Benjamin W. , Bubblepop , Grexis , janos , jimsug , kalimba , Will Barnwell , zarak
11	Cadena de comandos y operaciones.	jordi , Mateusz Piotrowski , uhelp

12	Cambiar shell	depperm , lamaTacos
13	Citando	Benjamin W. , binki , Gilles , Jahid , Will
14	Comando de corte	Richard Hamilton
15	Control de trabajo	Samik
16	Copiando (cp)	dingalapadum , Richard Hamilton
17	co-procesos	Dunatotatos
18	Creando directorios	Brendan Kelly
19	Cuándo usar eval	Alexej Magura
20	Declaración del caso	P.P.
21	Decodificar URL	Crazy , l0_ol
22	Depuración	Gavyn , Leo Ufimtsev , Sk606
23	Dividir archivos	Mohima Chaudhuri , Richard Hamilton
24	División de palabras	Jahid , Jesse Chen , RamenChef , Skynet
25	El comando de corte	Dario
26	Encontrar	Batsu , Daniel Käfer , Echoes_86 , fedorqui , GiannakopoulosJ , Inian , Jahid , John Bollinger , Laurel , leftaroundabout , Matt Clark , Michael Gorham , Mohima Chaudhuri , Peter , sjsam , tripleee ,
27	Escollos	Cody , Scroff
28	Escribiendo variables	uhelp
29	Espacio de nombres	meatspace , uhelp
30	Estructuras de Control	Dennis Williamson , DocSalvager , DonyorM , Edgar Rokyan , gzh , Jahid , janos , miken32 , Samik , SLePort
31	Evitando la fecha usando printf	Benjamin W. , chepner , kojiro , RamenChef
32	Expansión de la abrazadera	4444 , Benjamin W. , Jamie Metzger , mnoronha , Peter , uhelp , zarak
33	Expansión del parámetro Bash	Benjamin W. , chepner , codeforester , fedorqui , George Vasiliou , Grexis , hedgar2017 , J F , Jahid , Jesse Chen , Stephane Chazelas , Sylvain Bugat , uhelp , Will , WMios , ymbirtt

34	Expresiones condicionales	BurnsBA , Gilles , hedgar2017 , Jahid , Jonny Henly , mnoronha , RamenChef ,
35	Finalización programable	Benjamin W. , jandob
36	Funciones	BrunoLM , Christopher Bottoms , dimo414 , divyum , edi9999 , Jahid , janos , Matt Clark , Michael Le Barbier Grünewald , Neui , Reboot , Samik , Sergey , Sylvain Bugat , tripleee ,
37	Gestionando variable de entorno PATH	Jahid , kojiro , RamenChef
38	getopts: análisis inteligente de parámetros posicionales	pepoluan , sjsam
39	Grep	Chandrabhas Aroori
40	Leer un archivo (flujo de datos, variable) línea por línea (y / o campo por campo)?	Jahid , vmaroli
41	Listado de archivos	Benjamin W. , cswl , depperm , glenn jackman , Holt Johnson , Iain , intboolstring , J F , Jonny Henly , Markus V. , Misa Lazovic , mpromonet , Neui , Osaka , Richard Hamilton , RJHunter , Samik , Sylvain Bugat , teksisto
42	Manejando el indicador del sistema	RamenChef , uhelp
43	Mates	deepmax , Pavel Kazhevets , Tim Rijavec
44	Matrices asociativas	Benjamin W. , uhelp , UNagaswamy
45	Navegando directorios	Christopher Bottoms , JepZ
46	Oleoductos	Ashkan , Jahid , liborm , lynxlynxlynx
47	Paralela	Jon
48	Patrón de coincidencia y expresiones regulares	Benjamin W. , chepner , Chris Rasys , fedorqui , Grisha Levit , Jahid , nautical , RamenChef , suvayu
49	Patrones de diseño	mattmc

50	Personalizando PS1	Blacksilver , Cows quack , Jahid , Jonny Henly , Josh de Kock , Kamal Soni , Misa Lazovic , tversteeg , Wenzhong
51	Proceso de sustitución	Benjamin W. , cb0 , Dario , Doctor J , fedorqui , Inian , Martin Lange , Мона_Сax
52	Redes con Bash	dhimanta
53	Redireccion	Alexej Magura , Antoine Bolvy , Archemar , Benjamin W. , Brydon Gibson , chaos , David Grayson , Eric Renouf , fedorqui , Gavyn , George Vasiliou , hedgar2017 , Jahid , Jon Ericson , Judd Rogers , lesmana , liborm , Matt Clark , miken32 , Neui , Pooyan Khosravi , RamenChef , Root , Stephane Chazelas , Sven Schoenung , Sylvain Bugat , Warren Harper , William Pursell , Wolfgang ,
54	Salida de script en color (multiplataforma)	charneykaye
55	Script Shebang	DocSalvager , Sylvain Bugat , uhelp
56	Scripts CGI	suleiman
57	Secuencia de ejecucion de archivo	Riker
58	Secuencias de comandos con parámetros	Jahid , James Taylor , Kelum Senanayake , Matt Clark , RamenChef
59	Seleccionar palabra clave	UNagaswamy
60	strace	Chandrabhas Aroori
61	Tipo de conchas	Jeffrey Lin , liborm , William Pursell
62	Trabajos en tiempos específicos	fifaltra , uhelp
63	Trabajos y Procesos	Amir Rachum , Bubblepop , DonyorM , fifaltra , J F , Jouster500 , Mike Metzger , Neui , Riccardo Petraglia , Root , Sameer Srivastava , Sk606 , suvayu , u02sgb , WAF , WannaGetHigh , zarak
64	Transferencia de archivos usando scp	Benjamin W. , onur güngör , Pian0_M4n , Rafa Moyano , RamenChef , Reboot , Wojciech Kazior
65	Usando "trampa" para reaccionar a	Benjamin W. , Carpetsmoker , dubek , jackhab , Jahid , jerblack , laconbass , Mike S , phs , Roman Piták , Sriharsha Kalluru ,

	señales y eventos del sistema	suvayu , TomOnTime , uhelp , Will , William Pursell
66	Usando gato	anishsane , Bubblepop , Christopher Bottoms , glenn jackman , intboolstring , Jahid , Matt Clark , Rafa Moyano , Root , Samik , Samuel , SLePort , tripleee , vielmetti , xhienne ,
67	Usando orden	Flows , Mohima Chaudhuri
68	Utilidad de dormir	Ranjit Mane
69	variables globales y locales	George Vasiliou , Ocab19
70	Variables internas	Alexej Magura , Ashari , Ashkan , Benjamin W. , codeforester , criw , Daniel Käfer , Dario , Dr Beco , fedorqui , Gavyn , Grisha Levit , Jahid , mattmc , Savan Morya , Stephane Chazelas , tripleee , uhelp , William Pursell , Wojciech Kazior
71	verdadero, falso y: comandos	Alessandro Mascolo