Sebastian Duda, 123RF

Más allá de los básicos de Bash

JUEGOS SHELL

Incluso los principiantes pueden beneficiarse de un mayor conocimiento de los numerosos comandos integrados en la shell de Bash. **POR BRUCE BYFIELD**

La mayoría de los usuarios de GNU/Linux saben que la línea de comandos utiliza Bash (Bourne Again Shell) [1], pero lo que pocos saben es que, por defecto, usan la versión más sencilla de Bash. La verdad es que Bash es un comando muy parecido a cualquier otro. Suele ser el que ejecuta otros comandos – es decir, un intérprete de comandos – pero, al igual que cualquier otra aplicación, su comportamiento puede alterarse radicalmente mediante las opciones que añadimos cuando lo iniciamos o cuando ejecutamos algunas de sus instrucciones integradas. Además, podemos incluso elegir una shell completamente diferente a través de la cual interactuar con nuestro sistema.

La verdad es que muchas de las opciones de la shell y de los comandos integrados [2] son de interés sólo si escribimos scripts. Los usuarios del escritorio probablemente no encuentren demasiado uso, por ejemplo, para un login shell o comandos integrados como *continue* o *declare*. Con todo, incluso los principiantes en el prompt de comandos pueden beneficiarse aprendiendo más acerca de las opciones disponibles.

Modificar Bash es fácil. Desde cualquier ordenador o terminal, podemos iniciar Bash u otra shell como cualquier otro comando. En Gnome, podemos añadir opciones ejecutando Gnome Terminal y creando un perfil a medida. Una vez creado, seleccionamos *Editar | Perfiles | Edita | Título y Comando*. A continuación marcamos el cuadro de comprobación *Ejecutar shell personalizada en vez de mi shell* (Figura 1). En el campo *Comando personalizado*, introducimos la forma del comando Bash que deseamos ejecutar, luego, directamente debajo de él, seleccionamos lo que deseamos que ocurra cuando salimos del comando.

En KDE, de manera similar, abrimos Konsole y creamos un nuevo perfil seleccionando *Preferencias | Administrar Perfil*. Lo marcamos y pulsamos *Editar Perfil | General* para modificar el comando Bash (Figura 1). Si añadimos el widget de Konsole a nuestro panel o a la vista carpeta, presenta una lista de perfiles desde la que podemos elegir el que se ajuste a nuestras necesidades.

Efectos de Opciones

La estructura básica del comando Bash no es distinta de cualquier otro comando: *bash [opciones] [argumentos]*. Y, como cualquier otro comando, *bash* posee una mezcla de opciones de una sola letra comenzando con un guión (-), herencia de encarnaciones tempranas, junto con opciones más largas que comienzan con un doble guión (--) que fueron añadidos por el proyecto GNU.

Sin embargo, como comando, *bash* se comporta de forma inusual. El primer argumento después de la opción puede ser un fichero completo de comandos que Bash ejecuta en lugar de esperar nuestra entrada en el teclado. Incluso más importante, cuando Bash se inicia, se refiere a */etc/profile*, el perfil de Bash genérico, y *~/.profile* o *~./.bash_profile*, el perfil personalizado en el directorio de inicio de la cuenta actual, así como */etc/bash.bashrc* y *~/.bashrc* para shells sin logins. Añadiendo la opción *--noprofile* podemos evitar la lectura de ficheros de perfil, y la de los ficheros **bashrc*, añadiendo la opción *--norc*. En su lugar, podemos forzar a Bash a que utilice un sustituto para todos los ficheros *bashrc* con la opción *--rcfile [file]*.

Muchas de las opciones Bash son de interés principalmente para usuarios avanzados, aunque podríamos intentar ejecutar *--verbose* durante un momento para ver qué variables de entorno se llaman por cada comando (Figura 2). Muchos, como los que cambian los ficheros de arranque, modifican los recursos que usa Bash.

Sin embargo, podríamos intentar ejecutar el comando *bash --debugger -O extdebug* para ejecutar Bash en modo depuración. Otra opción moderadamente avanzada es iniciar una shell restringida con *-r*, *--restricted* o *-O restricted_shell*. Una shell restringida es exactamente como lo que suena: una en la que no están permitidas algunas opciones básicas. Entre éstas se incluyen el cambio a otros directorios, el apagado de restricciones con *set*, y otras nueve o diez opciones [3].

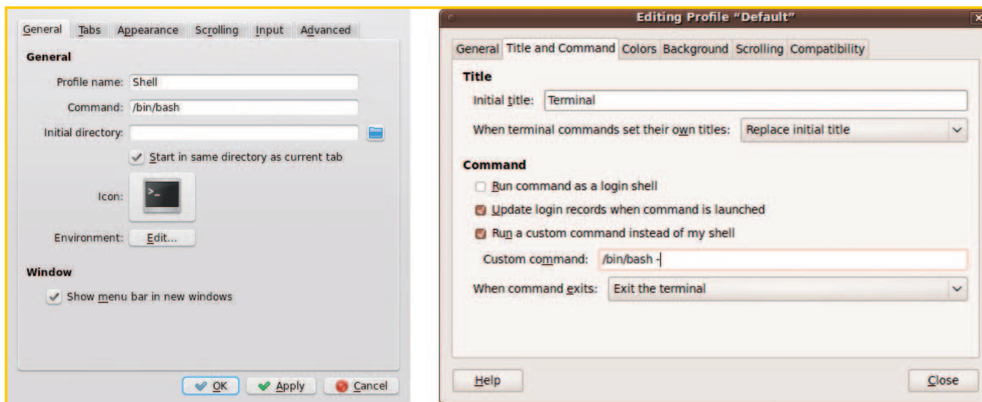

Figura 1: Si trabajamos con KDE (izquierda) o Gnome (derecha), podemos ejecutar una versión modificada de Bash u otra shell en general, haciendo un simple cambio en nuestro escritorio.

Algunos administradores de sistemas utilizan una shell restringida para impedir que los usuarios cambien de shell, aunque esta técnica es una táctica de seguridad débil, ya que no es necesario ser muy experto para arrancar una shell que sea interactiva. En cambio, una shell restringida es más valiosa cuando deseamos comprobar programas que podrían tener fallos o ser inseguros (Figura 3). La ejecución de dichos programas en una shell limitada puede minimizar cualquier daño que causen.

Quizás la opción más interesante sea *-O [modificaciónn]*, la cual afecta a Bash en un número simple de formas aunque productivas. Podemos iniciar las mismas modificaciones con el comando integrado de Bash *shopt -s [opción]*.

Además de las opciones *extdebug* y *restricted_shell* ya mencionadas, algunas de las modificaciones más útiles de manera inmediata (mods) afectan a la navegación. Por ejemplo, si ejecutamos *bash -O autocd*, Bash interpreta el nombre de un subdirectorio que se introduce como alias del comando *cd* para moverse a ese subdirectorio (Figura 4). Otro mod útil es *cd-spell*, el cual intenta corregir los errores ortográficos de los nombres de los directorios automáticamente – aunque suele requerir el path completo y falla cuando usamos abreviaturas como *./* para el directorio actual. De manera similar, *dirspell* corrige errores ortográficos de nombres de directorios en el autocompletado de nombres de ficheros.

```
bruce@nanday:~$ cd /home/bruce
cd /home/bruce
echo -ne "\033]0;${USER}@${HOSTNAME}: ${PWD}\007"
bruce@nanday:~$ 
```

Figura 2: Si nos preguntamos qué ocurre cuando ejecutamos un comando, debemos usar Bash con la opción verbose.

Otro mod para el autocompletado de nombres de ficheros es *nocaseglob*, que ignora la distinción habitual entre mayúsculas y minúsculas. También, *checkjobs* ordena a Bash a que muestre el estado de las tareas en ejecución o parados cuando abandonamos Bash. El mod *mailwarm* nos dice cuándo ha sido leído un fichero que contiene correo desde la última vez que Bash accedió a él.

Comandos Integrados

Muchos de los comandos integrados de Bash son para scripting, tema que trataremos otro día, pero además de *shopt*, con sus modificaciones del comportamiento de Bash, los comandos integrados también incluyen funciones que son útiles en la informática de escritorio diaria.

```
bruce@nanday:~$ bash -r
bruce@nanday:~$ cd /etc
bash: cd: restricted
bruce@nanday:~$ 
```

Figura 3: Una shell restringida hace exactamente lo que su nombre implica: ejecutar una shell en la que algunas funcionalidades, como cambiar de directorios, no están permitidas.

De hecho, los comandos integrados de Bash son inevitables. Si alguna vez hemos introducido *pwd* para comprobar en qué directorio estamos, o si creamos un alias de modo que introducir *ls* es equivalente a escribir *ls --color=auto*, entonces, hemos utilizado los comandos integrados de Bash. Por ejemplo, ni siquiera podemos navegar en el prompt sin usar el comando integrado *cd*.

Algunos de los comandos ofrecen información básica sobre nuestro sistema. Por ejemplo, el comando *type* nos dice si un comando es un alias o un builtin o da la ruta al ejecutable (Figura 5). No podemos ver los comandos del sistema como *apt-get* a menos que nos hayamos registrado como root, aunque *type* puede satisfacer nuestra curiosidad acerca de los comandos que estamos usando y cómo se le considera a cada uno de ellos. Por ejemplo, *cp* es un comando esencial y está situado en */bin*, mientras *sudo*, que nos permite asumir las funciones de root desde otra cuenta, es menos esencial para nuestro sistema, y es por eso por lo que se encuentra situado en */usr/bin*. También, podemos usar *type* para comprobar y ver si un comando es un alias, aunque podríamos preferir usar el comando *alias* en su lugar.

Otro comando integrado que nos da información es *jobs*, el cual lista procesos y si están ejecutándose o están detenidos, algo a veces difícil de saber si estamos ejecutando comandos en segundo plano, especialmente si siguen la tradición Unix de no proporcionar mensajes al terminar.

Introduciendo el comando sin argumentos se obtiene una lista de los procesos que son propiedad de la cuenta del usuario actual y el estado de cada uno de ellos. Si deseamos acabar un proceso porque se porta mal, podemos introducir *job -l* para encontrar su ID de proceso, luego, para finalizarlo, introducimos *kill [IDproceso]*.

Otros comandos integrados nos permiten tanto obtener información como editar el comportamiento del sistema. Por ejemplo, el sencillo comando *set* lista todas las variables de entorno, mientras que otras opciones nos permiten desactivar o activar ciertos comportamientos del sistema como la expansión de llaves (*-B*), que es la capacidad de Bash para sustituir una variable introducida entre llaves por su valor.

De manera parecida, *ulimit -a* nos da información sobre las limitaciones de recursos del sistema, como el número máximo de hilos permitidos o el número máximo de procesos que un único usuario puede tener en propiedad (Figura 6). Pero añadimos un número, y podemos resetear el límite. Por ejemplo, *ulimit -x 10*

```
bruce@nanday:~$ bash -O autocd
bruce@nanday:~$ download
cd download
bruce@nanday:~/download$ bash -O cdspell
bruce@nanday:~/download$ cd /ect
/etc
```

Figura 4: Aquí, la opción *-O* le dice a Bash que trate el nombre de un directorio, que se le pasa como opción, como el equivalente de la instrucción de cambiar a ese directorio. A continuación vemos cómo Bash intenta corregir los nombres de los directorios que contengan errores ortográficos.

establece el número de bloqueos de ficheros en *10*. También podemos seleccionar si un límite es duro o blando – esto es, si sólo el usuario root puede elevarlo, o si lo puede hacer cualquiera.

Los usuarios preocupados por la seguridad también estarán interesados en la utilidad umask. Ejecutando el comando *umask* sin opciones, nos dice los permi-

```
bruce@nanday:/etc$ type mv ls cd
mv is /bin/mv
ls is aliased to `ls --color=auto'
cd is a shell builtin
```

Figura 5: Podemos usar el comando type para descubrir si un comando es un alias, un comando integrado o un comando externo.

sos por defecto que se conferirán a un nuevo fichero cuando se crea en el sistema. Si ejecutamos *umask* en el sistema que estamos utilizando ahora, encuentro que el predeterminado está configurado a *0077* en notación octal, mientras que si ejecuto *umask -S*, encuentro que, en notación simbólica, los permisos por defecto son *u=rwx,g=, o=* (Figura 7). Ambas

```
bruce@nanday:~$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 38912
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 38912
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

Figura 6: El comando ulimit -a muestra las limitaciones de los recursos del sistema para nuestro ordenador. La tabla lista un recurso, su medida, la opción para listarla separadamente y cualquier límite que se encuentre impuesto actualmente en él. También podemos usar el comando ulimit para cambiar límites.

notaciones significan lo mismo: El propietario de un fichero puede leer, escribir o ejecutarlo, pero nadie de su grupo o nadie más, excepto root, puede hacer ninguna de estas cosas. Si quisiera permitir que otros usuarios leyeran nuevos ficheros, podría introducir *umask 4477* para cambiar los permisos predeterminados.

Otro comando integrado que merece la pena conocer es *history*, el cual administra el historial almacenado de comandos para cada usuario en el fichero *.bash-history* en su directorio de inicio. Sin argumentos, lista todos los comandos introducidos en la línea de comandos, con el más antiguo en primer lugar. Por privacidad, podemos introducir *history -d[posición]* para eliminar un comando específico, o *history -c* para limpiar el historial completo.

Shells Alternativas

Cuanto más avanzadas nuestras habilidades informáticas, más beneficios podemos obtener de Bash, tanto en términos de personalización como de funcionalidad. Los ejemplos dados aquí no son más que una introducción a algunos de los muy diversos que podemos encontrar cuando comenzamos a alejarnos de los predeterminados de Bash.

Una vez que hemos explorado Bash minuciosamente, es posible que deseemos explorar otras shells. La mayoría de las distribuciones más importantes contienen varias, instalando sus ejecutables en el directorio */bin*. Para usarlas, lo único que necesitamos es editar los perfiles de Gnome Terminal o Konsole y la variable de entorno *$BASH* para hacer referencia a ellas.

Relativamente pocos usuarios se molestan en saber nada de la Bourne shell original o con la shell C original (csh) o Korn shell (ksh), a pesar de que aún poseen los suficientes admiradores como para que las grandes distribuciones como Debian las incluyan. Sin embargo, la mayoría de los usuarios que exploran shells alternativas hoy son los que más probablemente deseen la funcionalidad añadida de sus sucesores, tales como *tcsh* [4] o *zsh* [5].

Se dice que las shells como tcsh y zsh son a Bash lo que Bash es a la línea de comandos de Windows – en otras palabras, enormemente superiores. Esto es una exageración y, en el caso de tcsh, probablemente se origina en el hecho de que tcsh es la shell por defecto de FreeBSD. Pero, sin duda, estas shells poseen numerosas funcionalidades de las que carece Bash o que posee de forma menos sofisticada. Por ejemplo, tcsh no sólo tiene una sintaxis de scripting similar a la que tiene el lenguaje de programación en C, sino que también posee autocompletado de palabras programable y corrección ortográfica. De forma parecida, zsh cuenta con un corrector ortográfico, autocompletado programable y redireccionamiento múltiple.

Los nuevos usuarios posiblemente deseen explorar shells desarrolladas recientemente como *fish* [6], cuya meta es conseguir trabajar con la línea de comandos fácilmente. Fish incluye resaltado de sin-

```
bruce@nanday:~$ umask
0077
bruce@nanday:~$ umask -S
u=rwx,g=,o=
```

Figura 7: Usamos umask para listar o establecer los permisos por defecto, bien en notación octal (sin opción), bien en notación simbólica (-S).

taxis e historial y autocompletado de pestañas mejorados.

Cada una de estas alternativas exige un poco de aprendizaje, aunque a menudo los comandos serán lo bastante parecidos como para que tengamos sólo los mínimos problemas para cada shell. Como Bash, con sus opciones y comandos integrados, sin embargo, estas alternativas shell enfatizan uno de los puntos más importantes de la línea de comandos: Al igual que en el escritorio, no tenemos por qué conformarnos con lo que nos dan. Si exploramos, pronto descubriremos que podemos hacer cosas a nuestra manera. ■

RECURSOS

[1] Bash: *http://www.gnu.org/software/bash/*

[2] Comandos integrados de Bash: *http://www.faqs.org/docs/bashman/bashref_55.html#SEC55*

[3] Shell restringidal: *http://www.faqs.o docs/rg/bashman/bashref_75.html*

[4] Tcsh: *http://www.tcsh.org/Welcome*

[5] Zsh: *http://www.zsh.org/*

[6] Fish: *http://fishshell.org/index.php*