Construcciones de CLI (Interfaz de línea de comandos) e información útil

El propósito de este artículo no es ser un tutorial de CLI, sino más bien ser una exposición de construcciones comunes utilizadas en shell scripting para lograr un objetivo de manera eficiente. También hay secciones que simplemente ayudan a entender un tema determinado.

Construye

rev | cut | rev

A menudo es útil revertir una cadena y luego usar cut. Por ejemplo, tome un paquete Slackware y obtenga su nombre:

```
echo dejavu-fonts-ttf-2.33-noarch-1 | rev | cut -d - -f 1-3 --complement | rev
ls -1 /var/log/packages | rev | cut -d - -f 1-3 --complement | rev
```

O si desea obtener la ruta completa de un archivo, menos el sufijo.

```
echo /proc/config.gz | rev | cut -d. -f1 --complement | rev
```

Reemplazar un sufijo

Digamos que querías hacer un script de conversión de video y que necesitabas cambiar el sufijo.

```
input=test.mkv
output="$(basename "$input" .mkv).avi"
```

find | xargs

Esta es una interacción especial entre find y xargs que permite tratar espacios en los nombres de archivos. Es muy rápido porque muchos comandos como `rm` , `rmdir` y `shred` toman múltiples entradas de archivos en la línea de comandos. Una construcción genérica es algo como:

```
find . -type f -print0 | xargs -0 "$command"
```

Puede reemplazar `$ command` con cualquier comando que necesite para ejecutar en los archivos siempre que sea compatible con la entrada de múltiples archivos. Si tiene una lista de archivos, aún puede conservar espacios:

```
tr '\n' '\0' < "$file" | xargs -0 "$command"
```

comm antes y después

Esta construcción es útil para aplicaciones de gestión de paquetes. Desde la página de manual de comm:

```
With  no  options,  produce  three-column  output.
Column one contains lines unique to FILE1,
column two contains lines unique to FILE2, and
column three contains lines common to both files.
```

Las opciones `'-1' '-2' '-3'` suprimen las columnas respectivas. Supongamos que desea registrar los archivos que se agregaron a `/usr` después de ejecutar el comando `$ 1` :

```
# before, make install, after
find /usr > "$tmp/before"
$1
find /usr > "$tmp/after"

# sort
sort "$tmp/before" > "$tmp/before-sorted"
sort "$tmp/after" > "$tmp/after-sorted"

# create log
comm -13 "$tmp/before-sorted" "$tmp/after-sorted" > "$log/$name"
```

Tenga en cuenta que comm requiere archivos ordenados. Aquí `-1` suprime las líneas exclusivas de antes, `-3` suprime las líneas presentes en ambos archivos, por lo que queda con la columna 2 que contiene archivos exclusivos después de los archivos agregados. A muchas personas les gustaría usar diff para comparar archivos, pero es principalmente para crear parches.

while read line

Esta construcción es común y es útil para leer archivos o ingresar una línea a la vez. Aquí hay un ejemplo que se puede usar para concatenar archivos divididos en orden:

```
base="$(echo "$@" | rev | cut -d. -f1 --complement | rev)"

ls -1 "$base".* | sort -V | while read line
do
    cat "$line" >> "$base"
done
```

También tenga en cuenta que `sort -V` es una clasificación de versión y es útil en los casos en que `ls` ordena los sufijos de forma incorrecta. La forma habitual de evitar esto es nombrar sufijos numerados con `0` como `file.001` , pero puede desbordarse y es por eso que `sort -V` es útil.

for i in

Aquí hay un ejemplo para extraer todos los rpms en un directorio:

```
for i in *.rpm
do
    rpm2cpio "$i" | cpio -id --quiet
done
```

También puede usar `seq` para hacer que `i` sea un contador de bucle:

```
for i in $(seq 1 100)
do
    echo "$i"
done
```

Tenga en cuenta que no hay comillas alrededor de `$(seq)` porque de lo contrario, citaría la secuencia numérica completa y eso no funcionaría bien.

Enlaces externos

- http://www.commandlinefu.com/commands/browse

Comilla

Las comillas pueden parecer complicadas, y las razones de ello son oscuras, pero tiene un propósito y no es tan complicado.

Doble comillas

La razón para la comilla doble es para preservar espacios, como espacios en nombres de archivos. La doble cita de una variable o una sustitución de comando lo convierte en un solo argumento. Un ejemplo:

```
bash-4.2$ ls
file with spaces.txt  filewithoutspaces.txt
bash-4.2$ rm -f file with spaces.txt
bash-4.2$ ls
file with spaces.txt  filewithoutspaces.txt
bash-4.2$ rm -f "file with spaces.txt"
bash-4.2$ ls
filewithoutspaces.txt
bash-4.2$ rm -f filewithoutspaces.txt
bash-4.2$ ls
bash-4.2$
```

Claramente necesitas encomillar un archivo con espacios. Puede usar comillas simples aquí, porque no hay variables dentro de las comillas. No debes encomillar en este caso:

```
bash-4.2$ for i in $(seq 1 10); do printf "$i "; done; echo;
1 2 3 4 5 6 7 8 9 10
bash-4.2$ for i in "$(seq 1 10)"; do printf "$i "; done; echo;
1
2
3
4
5
6
7
8
9
10
bash-4.2$
```

Tampoco debe encomillar en ningún caso en que un comando requiera múltiples variables y se las asigne dentro de una variable encomillada. Una variable encomillada se toma entonces como el único argumento, en lugar de múltiples argumentos. Un ejemplo:

```
bash-4.2$ ls
file with spaces.txt  filewithoutspaces.txt
bash-4.2$ file1="file with spaces.txt"
bash-4.2$ file2="filewithoutspaces.txt"
bash-4.2$ rm -f "$file1 $file2"
bash-4.2$ ls
file with spaces.txt  filewithoutspaces.txt
bash-4.2$ rm -f "$file1" "$file2"
bash-4.2$ ls
bash-4.2$
```

También tenga en cuenta que puede y debe encomillar dentro de las sustituciones de comandos, como se muestra en el ejemplo anterior “reemplazar un sufijo” y:

```
mkdir "$(basename "$(pwd)")"
```

Esto crea un directorio dentro del directorio actual llamado el mismo nombre que el directorio actual. Si pwd se expande en algo con espacios, el comando funcionará.

Comilla simple

La razón para la comilla simple es para escapar caracteres especiales de la shell, mientras se los pasa a un comando para que pueda usarlos. Debe usar comillas simples para cada argumento pasado a otro programa que contenga caracteres de shell para que sean interpretados por ese programa y **NO** por el shell. Ejemplo:

```
bash-4.2$ find -name *.txt
./list.txt
bash-4.2$ find -name '*.txt'
./list.txt
./results/002.txt
./results/006.txt
./results/013.txt
./results/wipe.txt
bash-4.2$
```

Aquí el sehell se expande * antes de encontrarla. Debe ingresar la entrada de comillas a awk , find , sed y grep , ya que cada uno de ellos utiliza caracteres especiales que se superponen con las del shell, y por lo tanto deben ser Protegido de la expansión del shell.

Enlaces externos

- http://www.grymoire.com/Unix/Quote.html

Expresiones regulares

Básico

- . coincide con cualquier carácter individual.
- \ evita del siguiente carácter.

Recuerde que debe evitar . usando \. Si quieres un verdadero .

```
bash-4.2$ cat test.txt
testtxt
test.txt
bash-4.2$ sed 's/.txt//g' test.txt
tes
test
bash-4.2$ sed 's/\.txt//g' test.txt
testtxt
test
```

- [ ] es una clase y combina cualquier cosa dentro de los corchetes para un solo personaje. Ejemplos:
  - [Yy] coincide con Y o y.
  - [a-z0-9] incluye un rango, y en este caso coincide con a a través de z y de 0 a 9.
  - [^ a-z] niega el rango, por lo que, en este caso, se ajusta a cualquier cosa excepto a a z.
- ^ coincide con el principio de una línea. Ejemplo: ^ a coincide con una a al principio de una línea.
- $ coincide con el final de una línea. Ejemplo: a $ coincide con una a al final de una línea.
- \ < coincide con el principio de una palabra. Ejemplo: \ <a coincide con una a al principio de una palabra.

- `\>` coincide con el final de una palabra. Ejemplo: a `\>` coincide con una a al final de una palabra.
- Ejemplo: `\ <[tT] he \>` coincide con la palabra `el` o `El` .
- `*` coincide con cualquier número del carácter anterior o nada = ningún carácter. Ejemplo: `[0-9] *` que coincidirá con cualquier número de números. `. *` coincide con cualquier número de cualquier cosa.

Expresiones regulares extendidas

El programa debe admitir lo siguiente para que funcionen. Por ejemplo, para grep debe ejecutar `egrep` o `grep -E` .

- `+` coincide con cualquier número del carácter anterior, como `*` , pero debe haber al menos uno para que coincida, por lo que no coincidirá con nada o ningún carácter.
- `?` hace que el carácter anterior sea opcional (puede faltar), y se compara como máximo una vez.
- `(|)` actúa como una declaración OR. Ejemplo: `(it | her | this)` coincide con cualquiera de esas palabras.
- `a {3}` coincide con `aaa` = 3 a's.
- `a {4,8}` coincide con a al menos 4 veces y como máximo 8 veces, por lo que `aaaa` , `aaaaa` , `aaaaaa` , `aaaaaaa` y `aaaaaaaa` .
- `{0,}` = *
- `{1,}` = +
- `{, 1}` = ?

Enlaces externos

- http://www.grymoire.com/Unix/Regular.html
- http://www.regular-expressions.info/

Comandos útiles e información

stat

Stat es la forma más precisa de determinar:

- Tamaño del archivo en bytes:

```
stat -c '%s' file.txt
```

- Permisos de archivos en octal:

```
stat -c '%a' file.txt
```

awk variable por defecto

Un punto importante es que las variables awk se establecen en cero de forma predeterminada. Esto puede causar problemas en algunas situaciones. Ejemplo:

```
echo -ne '-321\n-14\n-1\n-34\n-4\n' | awk 'BEGIN{max=""}{if ($1 > max) max=$1; if ($1 < min) min=$1;}END{print min"\t"max}'
```

Esto funciona correctamente porque max se establece en una cadena vacía y, por lo tanto, tiene un valor más bajo que cualquier número. Intenta eliminar la cláusula BEGIN y ver qué pasa. También tenga en cuenta que la adición de `min = “”` a la cláusula BEGIN también falla.

no hay prueba de directorio de datos

Puede usar esto para probar si un directorio no contiene datos. Por ejemplo, dirá `0` si el directorio solo contiene archivos vacíos y directorios = sin datos.

```
du -s directory
```

cmp

Esto puede comparar dos archivos byte a byte, y puede ser más útil que las sumas de comprobación. Por ejemplo, después de grabar un CD / DVD, puede ejecutar:

```
cmp slackware.iso /dev/sr0
```

Debe decir lo siguiente si el disco se quemó correctamente:

```
cmp: EOF on slackware.iso
```

shell matemáticas

Recuerde que las utilidades de shell como `let` y `expr` solo hacen matemáticas enteras. Para el punto flotante, use `bc` o `awk` .

shell GUI

Existen numerosos programas que le permiten crear GUI desde un script de shell.

- Xdialog is fully dialog compatible.
- A function library that increases shell GUI portability.
- Gtkdialog has advanced features for customized GUIs.

Enlaces externos

- Guide to awk, bash, sed, find, and more
- Advanced bash guide
- Cheat sheets on Linux CLI programs, the main site is also useful
- Guide to file permissions

Sources

- Cité `man comm`
- Usé `man grep` para la sección de expresiones regulares.
- Escrito por H_TeXMeX_H
- Traducido por: Victor 2019/02/07 20:04 (UTC)

howtos, software, author htexmexh