Introducción a Sistemas Operativos: Usando el shell

Clips xxx
Francisco J Ballesteros

1. Trabajando con tablas

En gran cantidad de casos tendrás datos tabulados como en una hoja de cálculo. Gran número de ficheros de configuración en UNIX presentan este aspecto, y además la salida de muchos comandos está tabulada.

La mayoría de las veces puedes operar filtrando filas y columnas y trabajar con los comandos que hemos visto durante el curso. Para cálculos numéricos puedes utilizar cualquier de las calculadores de línea de comandos, aunque tal vez sea más simple utilizar *expr(1)* para evaluar expresiones numéricas simples, como en:

```
unix$ n=4
unix$ n=‘expr $n ’*’ 2‘
unix$ echo $n
8
unix$
```

No necesitas mucho más.

No obstante, dispones de *awk(1)* que es un lenguaje pensado para operar sobre datos tabulados. Ya lo hemos utilizado para seleccionar campos, pero ahora vamos a ver algunos otros usos típicos y como son los programas de `awk` en general.

La idea es similar a *sed(1)*. Para `awk` los ficheros constan de registros (líneas) que tienen campos (separados por blancos, aunque ya sabes como cambiar esto). Lo que hace `awk` es leer la entrada (o el fichero indicado como argumento) línea a línea y aplicar el programa sobre el mismo.

Un programa en `awk` se suele escribir directamente en el primer argumento en la llamada a `awk`, pero ya sabes que puedes utilizar "`#!`" en scripts para lo que gustes. El programa consta de **reglas**, que tienen una expresión (opcional) que selecciona a qué lineas se aplica la regla y una acción entre llaves (opcional) que indica qué debe hacer `awk` en esas líneas. Si no hay expresión se considera que la acción se aplica a todas las líneas. Si no hay acción se considera que la acción es imprimir.

En las expresiones y en las acciones tienes predefinidas las variables "`$1`" (el primer campo), "`$2`" (el segundo), etc. El registro entero, la línea, es "`$0`". Y además tienes definidas las variables `NR` *number of record* (número de línea) y `NF` *number of field* (número de campos).

Para jugar, vamos a utilizar el fichero `/etc/passwd` que define las cuentas de usuario. Tiene este aspecto:

```
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
daemon:*:1:1:System Services:/var/root:/usr/bin/false
...
```

El nombre de usuario, seguido de su uid, gid, nombre real, home y por último el shell que utiliza. Todos los campos separados por "`:`".

Vamos a imprimir tan sólo el nombre de usuario y el shell que utilizan:

```
unix$ awk -F: '{print $1, $NF}' /etc/passwd
nobody /usr/bin/false
root /bin/sh
daemon /usr/bin/false
...
```

En lugar de print, podemos utilizar `printf`. Las acciones de `awk` utilizan sintáxis casi como la de C y las variables no es preciso declararlas. Sabiendo esto puedes ser optimista y hacer intentos, la página de manual te sacará de dudas en cualquier caso.

```
unix$ awk -F: '{printf("user %s shell %s\n", $1, $NF);}' /etc/passwd
user nobody shell /usr/bin/false
user root shell /bin/sh
user daemon shell /usr/bin/false
...
```

¡Ahora sólo para *root*!

```
unix$ awk -F: '
>    $1 == "root" {
>        printf("user %s shell %s\n", $1, $NF);
>    }' /etc/passwd
user root shell /bin/sh
unix$
```

En este caso la expresión `$1 == "root"` hace que la acción ejecute sólo si el primer campo es el string `root`.

Pero compara con este otro comando:

```
unix$ awk -F: '
>    $1 ~ /root/ {
>        printf("user %s shell %s\n", $1, $NF);
>    }' /etc/passwd
user root shell /bin/sh
user _cvmsroot shell /usr/bin/false
unix$
```

El operador "`~`" permite ver si cierto texto encaja con una expresión regular (escrita entre "`/`"). Por eso aparecen dos cuentas, porque el nombre de usuario contiene `root` en ambas.

Igual que en el caso de `sed`, podemos utilizar como expresión un par de expresiones regulares separadas por una coma, lo que hace que la expresión sea cierta para las líneas comprendidas entre una que encaja con la primera y otra que encaja con la segunda.

```
unix$ seq 15 | awk '/3/,/5/ {print}'
3
4
5
13
14
15
unix$
```

Verás que aparecen las líneas entre la 3 y la 5 (que encajan con las expresiones) y las líneas entre la 13 y la

15 (que también lo hacen). Las expresiones pueden ser mas elaboradas y puedes utilizar los operadores "&&" y "||" casi como en C.

La acción `next` salta el registro en curso. Considerando que `awk` procesa las reglas en el orden en que las escribimos, podemos hacer que imprima un rango de líneas pero salte algunas. Por ejemplo,

```
unix$ seq 15 | awk '
>    $0 == "4" {next}
>    /3/,/5/ {print} '
3
5
13
14
15
unix$
```

Muy útil para utilizar

```
$0 ~ /^#/ {next}
```

al principio del programa de `awk` e ignorar las líneas que comienzan por un "#" (que suele ser el carácter de comentario en el shell y en muchos ficheros).

Las expresiones `BEGIN` y `END` hacen que su acción ejecute antes de procesar la entrada y después de haberla procesado. Podemos utilizar esto para sumar los números de la entrada, por ejemplo:

```
unix$ seq 15 | awk '
>    {tot += $0}
>    END {print tot}'
120
unix$
```

O para obtener la media:

```
unix$ seq 15 | awk '
>    {tot += $0}
>    END {print tot/NR}'
8
unix$
```

Este otro obtiene la media de los números pares, y además los imprime:

```
unix$ seq 15 | awk '
>    $0 % 2 == 0 {
>        print;
>        tot += $0;
>    }
>    END {print tot/NR}'
2
4
6
8
10
12
14
3.73333
```

Pero podríamos haberlo hecho así:

```
unix$ seq 15 | awk '
>    { if ($0 %2 == 0) {
>        print;
>        tot += $0;
>      }
>    }'
2
4
...
```

Simplemente recuerda que las acciones disponen de sintaxis similar a la de C y se optimista. Consulta *awk(1)* para ver qué funciones tienes disponibles y que expresiones y estructuras de control. Si has entendido lo que hemos estado haciendo, seguro que te sobra con la página de manual.

¿Puedes escribir un comando que numere las líneas de un fichero? Para imprimirlo con números de línea, por ejemplo.