Divertindo-se com Awk

Aprenda a usar o Awk, uma poderosa linguagem para manipulação de arquivos texto. Veja como ele pode substituir outros comandos e facilitar a sua vida.

O Awk é uma linguagem de script criada com basicamente um objetivo: comparar texto e tomar uma atitude de acordo com o texto localizado (ou seja, um processador de textos). O nome da linguagem é a junção das iniciais dos sobrenomes dos seus autores: Alfred Aho, Pete Weinberger e Brian Kernighan. Quem já leu o "Livro do Dragão" de compiladores, o "C Programming Language", ou escreveu um Hello, World deve ficar muito grato a eles... e quem usa Awk também!

A sintaxe mais básica do Awk é:

```
awk '/padrão/ { ação }'
```

Ou seja, toda vez que ele achar algo que case com /padrão/, ele vai executar uma determinada ação. A gente pode testar isso fácil:

```
$ awk '/localhost/ { print $0 }' /etc/hosts
127.0.0.1               localhost
```

Esse comando lê o arquivo /etc/hosts e procura onde existir o padrão "localhost", quando achar, ele imprime a linha na tela (o tal do $0). Como achar um padrão e colar a linha na tela é uma atividade muito comum (vide o grep) essa é a ação default do Awk então podemos reescrever o comando acima como:

```
$ awk '/localhost/' /etc/hosts
127.0.0.1               localhost
```

Bom, agora já sabemos como usar o awk como um "grep" alternativo -;). E, onde está "localhost" pode-se usar qualquer padrão, inclusive expressões regulares:

```
$ awk '/^[0-9]/' /etc/hosts
127.0.0.1               localhost
192.168.0.1             optimus.mylab optimus
192.168.0.2             rachael.mylab rachael
...
192.168.0.10            megaman.mylab megaman
```

Com o mesmo arquivo de exemplo (/etc/hosts), podemos fazer mais uma brincadeira:

```
$ awk '/localhost/ { print $2 }' /etc/hosts
localhost
```

O $2 significa o segundo campo da linha, o primeiro é o $1 e o terceiro o $3 (e as pessoas inteligentes já sacaram que o quarto é $4, o quinto $5 e assim sucessivamente). E, ao contrário do que muita gente pensa, não é fácil de fazer com o "cut":

```
$ grep localhost /etc/hosts | cut -d' ' -f2
127.0.0.1               localhost
```

Não funcionou porque entre o 127.0.0.1 e o localhost tem Tabs, e não espaços. Trocando os tabs por espaços, também não dá certo:

```
$ grep localhost /etc/hosts | cut -d' ' -f2
```

O grep acabou de mostrar o segundo espaço da linha. Para mostrar o localhost, precisaríamos fazer:

```
$ grep localhost /etc/hosts | cut -d'  '  -f16
localhost
```

Claro, usando tr -s " " dá para sumir com os espaços excedentes... Para isso que a inteligência do awk vem a calhar. Com um só comando a gente consegue detectar e mostrar o segundo campo, não importando se são Tabs, espaços ou a quantidade deles. Além de poder usar o Awk como um "grep" alternativo, podemos usá-lo como um "cut" também -;).

E, em termos de "cut" existe uma feature avançadíssima do Awk que quando menos se espera é tremendamente útil. Por exemplo, o /etc/hosts aqui é assim:

```
127.0.0.1               localhost
192.168.0.1             optimus.mylab optimus
192.168.0.2             rachael.mylab rachael
...
192.168.0.10            megaman.mylab megaman
```

Se eu quiser listar todos os IPs e o apelido de cada host, poderia listar $1 e $3; mas, com isso, não iria listar o 'localhost' que está no segundo campo. O que fazer?

```
$ awk '/^[0-9]/ { print $1"\t"$NF}' /etc/hosts
127.0.0.1       localhost
192.168.0.1     optimus
192.168.0.2     rachael
...
192.168.0.10    megaman
```

NF é o número de campos de cada linha. Então imprimimos o primeiro campo ($1) e o último ($NF). O "\t" é para imprimir um TAB e deixar tudo alinhadinho. E ainda combinamos dois comandos em um -;).

Parecido com o NF, o Awk tem uma segunda variável chamada NR: Número de Registro. Como normalmente um registro é uma linha, pode-se usar esse NR para indicar em qual linha o Awk está processando:

```
$ awk '{ if (NR==1) print $0 }' /etc/group
root::0:root
```

Usamos o comando "if" (se) e comparamos o número de registro, se for igual a 1, imprimimos a linha. Com isso conseguimos imprimir uma linha arbitrária qualquer. Com um pouco de imaginação, podemos fazer uma versão tosca do "head":

```
$ awk '{ if (NR<=10) print $0 }' /etc/group
root::0:root
bin::1:root,bin,daemon
daemon::2:root,bin,daemon
sys::3:root,bin,adm
adm::4:root,adm,daemon
tty::5:
disk::6:root,adm,punk
lp::7:lp
mem::8:
kmem::9:
```

Ao invés do compararmos se o NR é igual a 1, verificamos se ele é menor ou igual a 10. Todas as operações de comparação "normal" funcionam: ==, !=, <=, >=, < e >, sem mistérios.

Depois de fazer o head, porque não inventar um novo comando tail? Esse é um pouco mais complicado, nós só sabemos o tamanho do arquivo quando ele já chegou no final, então temos que armazenar as linhas de texto em algum lugar. Esse exemplo vai ficar um pouco complicado:

```
$ awk '{ TEXT[NR]=$0 }
> END {
>       for (i=NR-9;i<=NR;i++) {
>         print TEXT[i]
>     }
> }' /etc/group
pop::90:pop
scanner::93:
nobody::98:nobody
nogroup::99:
users::100:
console::101:
_ntp::82:_ntp
postdrop:x:104:
haldaemon:x:61:
messagebus:x:60:
```

A primeira linha do nosso programa Awk realiza uma ação específica para todas as linhas encontradas: armazena o conteúdo da linha dentro da variável TEXT[NR] e, a gente lembra que NR é o número da linha. A segunda parte do programa é a mais "complicada". Nela a gente utiliza um padrão especial "END", que sempre "casa" com o final do arquivo (no nosso caso, quando todas as linhas foram lidas) utiliza um padrão especial "END", que sempre "casa" com o final do arquivo (no nosso caso, quando todas as linhas foram lidas); quando chegamos no final do arquivo, fazemos um laço de repetição:

```
for (i=NR-9;i<=NR;i++) { print TEXT[i] }
```

A tradução disso é:

```
i é igual a NR-9
para cada i que você encontrar e for menor ou
igual a NR
imprima o conteudo de TEXT[i]
e some 1 em i.
```

Ou seja, imprima as últimas 10 linhas do texto. É visível que isso de laço de repetição é algo muito útil, vale a pena guardar essa idéia. O padrão "END" também é extremamente útil e ele tem um irmão o "BEGIN" que "casa" com o início do arquivo (antes de qualquer linha ser lida). É uma boa para imprimir cabeçalhos ou algo do tipo.

Já que com a substituição do tail nós acabamos fazendo um script awk mais complexo, vamos chutar o pau da barraca e substituir o wc. O wc conta a quantidade de caracteres, palavras e linhas de um determinado texto:

```
$ wc /etc/fstab
 12  72 794 /etc/fstab
```

O primeiro número são as linhas, o segundo as palavras e o terceiro os caracteres. Contar as linhas todos nós já sabemos:

```
$ awk 'END { print NR }' /etc/fstab
12
```

Lembrando, o NR é o número do registro (linha); quando chegamos no evento END (final do arquivo), o NR contém o número da última linha. Agora vamos contar as palavras:

```
$ awk '{ WORD+=NF }
> END { print WORD }' /etc/fstab
72
```

Esse precisou de um pouco mais de imaginação, o que são as palavras dentro de uma linha se não os campos dela? Por isso, para cada linha lida, o conteúdo de NF é somado na variável WORD e, no final do arquivo, essa variável é impressa na tela. Para a última parte do wc, que é a contagem de caracteres, vamos precisar de um novo comando: length.

```
$ awk '{ CHAR+=length($0) }
> END { print CHAR }' /etc/fstab
782
```

O comando length($VAR) pega o conteúdo da variável e verifica o tamanho dela em caracteres. Ou seja, CHAR+=length($0) soma na variável CHAR todos os caracteres que tem na linha corrente. E, no final,

imprimimos a variável CHAR. Só tem um probleminha soma na variável CHAR todos os caracteres que tem na linha corrente. E, no final, imprimimos a variável CHAR. Só tem um probleminha... o número está errado! O wc informa 794 caracteres e o Awk informa só 782. O que acontece? A resposta é simples, o wc conta os caracteres de "salto de linha" (vulgo, o enter) e o Awk não. Para resolver essa disparidade, podemos somar no final a quantidade de saltos de linha que tem no arquivo:

```
$ awk '{ CHAR+=length($0) }
> END { print CHAR+NR }' /etc/fstab
794
```

E assim fica correto -:) Para finalizar, vamos colocar para imprimir todas as informações juntas, como o wc:

```
$ awk '{ CHAR+=length($0) ; WORD+=NF }
> END { print NR,WORD,CHAR+NR }' /etc/fstab
12 72 794
```

Pronto! Mais um comando substituído com sucesso!

Claro que o wc, o tail e o head ficaram muito maiores que os originais; mas serviram para mostrar vários recursos do Awk e a ensinar como usá-los. Já a substituição de grep+cut por Awk é bem interessante, além de ser extremamente mais versátil.

Umas últimas brincadeiras antes de fechar o artigo:

```
$ awk -F: '{ if ($3>=1000)
> { print $1 } }' /etc/passwd
punk
marina
toledo
garoto
tamiris
infomedia
slackshow
```

Isso mostra todos os usuários que não são do sistema na máquina. A novidade é o -F:, o -F determina qual vai ser o separador de campo, no caso, nós trocamos o espaço/tab pelo dois pontos, que é o caracter utilizado para separar as informações no /etc/passwd. Depois é só comparar o terceiro campo (UID) e verificar se é maior ou igual a 1000 (no slackware os usuários começam no 1000) e imprimir o primeiro -;).

Outro exemplo bom:

```
# ps aux | \
awk '/^usuario/ { system("kill -9 "$2) }'
```

Esse serve para derrubar um usuário (e todos os programas dele que estiverem sendo executados). A novidade aí é utilizar a instrução "system", com ela podemos executar um comando do sistema (no nosso caso o "kill -9") é fácil adaptar para ao invés de matar por um determinado usuário, matar pelo nome do programa (sim... para matar aqueles programas mal educados).

Esse é interessante para imprimir só o nome dos pacotes instalados, sem a versão e arquitetura:

```
$ ls -1 /var/log/packages/ | \
> awk 'BEGIN { FS="-" ; OFS="-" }
> { NF=NF-3 ; print $0 }'
a2ps
aaa_base
aaa_elflibs
...
x11-fonts-misc
x11-fonts-scale
x11-xdmx
...
ytalk
zlib
zsh
```

Utilizamos nesse comando o "BEGIN", que é executado antes de processar o arquivo. Como ele "casa" antes de qualquer linha de entrada ser processada, é o lugar ideal para colocar inicialização de variáveis. E, no nosso caso nós iniciamos duas delas: FS, que indica o separador de campo (é identico ao -F) e o OFS que é o separador de campo para saída; sem o OFS o nome do "x11-xdmx" sairia "x11 xdmx" que não é o que queremos. Já na outra linha vem um macete... os nomes dos pacotes do slackware são assim:

```
nome-versao-arquitetura-versao_do_pacote
```

E "nome" pode ter qualquer tamanho, separado por hífens. O que nós sabemos é que depois do nome, existem apenas três outros campos. Aí vem o truque. Para cada linha nós dizemos que ela possui três campos a menos (NF=NF-3) e depois é só imprimir a linha completa -;).

Bom, e com essas últimas dicas terminamos o artigo -:). Espero que tenha sido útil.

Piter PUNK
<piterpk@terra.com.br>