Formation éclair à Awk

Emmanuel Halbwachs

03 juillet 2023

Contents

1 Awk en deux mots

- vieil outil (1977) mais toujours vert
- développé par Alfred Aho, Peter Weinberger et Brian Kernighan -> AWK
- plusieurs versions
  - mawk : Michael's Awk
  - gawk : GNU Awk (un peu plus lent mais avec des extensions intéressantes)

```
update-alternatives --list awk
```

```
/usr/bin/gawk
/usr/bin/mawk
```

2 Invocation de Awk, fichier en entrée

On utilise ce fichier `foo.txt` pour les premiers exemples :

```
cat foo.txt
```

```
foo 17
bar 32
baz 42
qux 87
quux 3
```

Si on lance *awk* tel quel sur le fichier, on n'obtient qu'un message d'erreur :

```
awk foo.txt
```

```
awk: ligne de commande:1: foo.txt
awk: ligne de commande:1:    ^ syntax error
```

C'est parce qu'on n'a pas spécifié de code à *awk*. Spécifions du code vide (rien) :

```
awk '{}' foo.txt
```

```
(rien)
```

On n'obtient rien. Si on demande d'imprimer chaque ligne, on obtient (presque) la même chose que *cat* :

```
awk '{print}' foo.txt
```

```
foo 17
bar 32
baz 42
qux 87
quux 3
```

Bien sûr, on peut aussi fournir les données sur l'entrée standard au lieu de donner un fichier en argument (*use of useless cat* juste pour l'illustration) :

```
cat foo.txt | awk '{print}'
```

On observe déjà deux concepts :

- le code est entre `{}`
- le code est executé pour chaque ligne du fichier d'entrée

Bien que ce soit le cas à 99,9 %, le code n'a pas forcément de rapport avec le contenu du fichier lu en entrée :

```
awk '{print "Hello World!"}' foo.txt
```

```
Hello World!
Hello World!
```

```
Hello World!
Hello World!
Hello World!
```

On note que le code a bien été exécuté 5 fois, car le fichier d'entrée comporte 5 lignes.

Le code peut être spécifié en argument d'*awk* (option `-f`) comme précédemment pour des *one-liners*, mais aussi dans un fichier :

```
cat print.awk
```

```
{
    print
}
```

```
awk -f print.awk foo.txt
```

```
foo 17
bar 32
baz 42
qux 87
quux 3
```

On peut aussi utiliser un *shebang* en début de fichier et le rendre exécutable, pour pouvoir l'invoquer tel quel :

```
chmod u+x demo01
cat demo01
```

```
#!/usr/bin/awk -f
{
    print
}
```

```
./demo01 foo.txt
```

```
foo 17
bar 32
baz 42
qux 87
quux 3
```

3 Enregistrement et champ (*record*, *field*)

Awk utilise la notion d'enregistrement (*record*) et de champ (*field*) pour traiter les données d'entrée. Les enregistrements sont séparés par un *record separator* (variable `RS`) et les champs par un *field separator* (variable `FS`). Les valeurs par défauts sont respectivement `SPACE` et `\n`.

Par défaut, les données d'entrée sont découpées en lignes, elle-mêmes découpées en champs séparés par une espace.

Chaque champ va dans une variable `$1`, `$2`, etc.

La variable `$0` contient tout l'enregistrement (tous les champs).

Affichage respectif du 1er et 2e champ :

```
awk '{print $1}' foo.txt
echo
awk '{print $2}' foo.txt
```

```
foo
bar
baz
qux
quux

17
32
42
87
3
```

Illustration de la variable `$0` qui contient tout l'enregistrement (tous les champs) :

```
awk '{print $0}' foo.txt
```

```
foo 17
bar 32
baz 42
qux 87
quux 3
```

Il arrive fréquemment que l'on ait besoin de changer la valeur de `FS` et il y a une option pour cela (`-F`) :

```
head -5 /etc/passwd
echo
echo "UID LOGIN"
head -5 /etc/passwd | awk -F: '{print $3, $1}'
```

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync

UID LOGIN
0 root
1 daemon
2 bin
3 sys
4 sync
```

4 Comment Awk exécute un script

Un script Awk contient 3 grandes sections :

```
BEGIN {
    [...]
```

```
}

{
    [...]
}

END {
    [...]
}
```

D'abord la section `BEGIN`, optionnelle. Puis la section principale, qui est exécutée pour chaque enregistrement. Et enfin la section `END`, optionnelle. Comme on le devine, les sections `BEGIN` et `END` sont exécutées au tout debut et à la toute fin du traitement de l'entrée.

Un exemple pour illustrer :

```
head -5 /etc/passwd | awk '
BEGIN {
    FS      = ":"
    counter = 0
}
{
    counter++
    print $1
}
END {
    print "Terminé. Nombre de lignes lues : " counter
}
'
```

```
root
daemon
bin
sys
sync
Terminé. Nombre de lignes lues : 5
```

Il est possible de mettre un test devant la section principale :

```
awk -F: 'BEGIN { print "Logins à 3 lettres :" } $1 ~ /^...$/ { print $1 }' /etc/
    passwd
```

Voici une réécriture du *one-liner* ci-dessus pour plus de clarté :

```
awk -F: '
BEGIN {
    print "Logins à 3 lettres :"
}

$1 ~ /^...$/ {
    print $1
}

' /etc/passwd
```

```
Logins à 3 lettres :
bin
```

```
sys
man
irc
tss
```

La section `END` s'utilise en général pour faire un traitement postérieur au traitement de chaque ligne du fichier d'entrée. P. ex. ici pour compter les lignes (`NR` est une variable spéciale contenant le numéro de l'enregistrement courant) :

```
awk 'END { print NR }' foo.txt
```

```
5
```

On peut voir ici que finalement la section principale elle aussi n'est pas obligatoire. On pourrait réécrire le *Hello, World!* classique sans fichier d'entrée de cette façon, avec uniquement une section `BEGIN` :

```
awk 'BEGIN { print "Hello, World!" }'
```

```
Hello, World!
```

5 Afficher : `print()` et `printf()`

Comme dans pas mal de langages, Awk a une instruction simple pour écrire sur la sortie : `print`. Quelques illustrations :

```
echo dummy | awk '{print "Hello""World""!"}'
echo dummy | awk '{print "Hello" "World" "!"}'
echo dummy | awk '{print "Hello"    "World"    "!"}'
echo World | awk '{print "Hello" $1 "!"}'
echo World | awk '{print "Hello " $1 "!"}'
echo World | awk '{print "Hello ", $1, "!"}'
echo World | awk -v OFS=';' '{print "Hello ", $1, "!"}'
```

```
HelloWorld!
HelloWorld!
HelloWorld!
HelloWorld!
Hello World!
Hello  World !
Hello ;World;!
```

On peut voire que si les chaînes sont séparées par zéro, un ou plusieurs espaces, elles sont concaténées.

Si on les sépare avec une virgule, c'est la variable `OFS` (Output Field Separator) qui sert de séparateur.

Pour les écritures qui nécessitent plus de contrôle, il y a le classique `printf`. L'utilisation de `printf` est vraiment similaire à celle du C et dans d'autres langages, donc je ne développe pas ici les formats. Voici juste un exemple pour illustrer :

```
awk '{printf("%3d %5s\n", $2, $1)}' foo.txt
echo
awk '{printf("%.2f\t%+5s\n", $2, $1)}' foo.txt
```

```
 17   foo
 32   bar
 42   baz
 87   qux
  3  quux

17.00     foo
32.00     bar
42.00     baz
87.00     qux
3.00     quux
```

6 Fonctions utiles les plus courantes

6.1 Substitution : `gsub()`

Une substitution globale, équivalente à `s/foo/bar/g`, peut s'appliquer à toute la ligne ou à un champ donné :

```
awk '/qu/ {print; gsub(/qu/, "tu"); print}' foo.txt
echo
awk '/7/ {print; gsub(/7/, "777", $2); print}' foo.txt
```

```
qux 87
tux 87
quux 3
tuux 3

foo 17
foo 1777
qux 87
qux 8777
```

6.2 Séparation : `split()`

On a parfois besoin de découper un champ selon un motif. P. ex. si un champ est une adresse IPv4 au format classique, on pourrait vouloir récupérer le 4e octet :

```
printf "foo 10.20.30.17\nbar 10.20.30.83\n" | \
awk '{split($2, octet, /\./); print octet[4]}'
```

```
17
83
```

Le résultat de `split()` est mis dans le 2^{e} argument qui est un tableau. On voit ici que les index des tableaux commencent à 1.

7 Tableaux associatifs

Les index des tableaux associatifs (*hash*) sont des chaînes. On peut parcourir le tableau avec une boucle `for`. Attention, les résultats ne sont pas dans l'ordre du fichier d'entrée, comme cela arrive souvent avec les tableaux associatifs dans différents langages.

```
awk '{ tableau[$1] = $2 }
     END { for (key in tableau) {
```

```
            printf("%s : %s\n", key, tableau[key])
        }
    }' foo.txt
```

```
foo : 17
baz : 42
qux : 87
quux : 3
bar : 32
```

Pour la petite histoire, les tableaux simples (dont l'index numérique commence à 1) sont en fait des tableaux associatifs avec une clé qui est une chaîne qui ne contient que des caractères numériques. Mais on n'a pas besoin de le savoir pour les utiliser comme des tableaux classiques :

```
echo dummy | awk 'BEGIN {
        tableau[1] = "foo"
        tableau[2] = "bar"
        tableau[3] = "baz"
    }
    END {
        for (i = 1 ; i <= 3 ; i++) {
            printf("%d : %s\n", i, tableau[i])
        }
    }'
```

```
1 : foo
2 : bar
3 : baz
```

On peut voir que la syntaxe de la boucle `for` est la même que celle du langage C. Il faut une entrée, ici le `echo dummy`, pour déclencher le bloc `END`.

8 Plusieurs fichiers en entrée

En plus du fichier `foo.txt`, nous avons besoin d'un second fichier exemple `bar.txt` :

```
cat foo.txt
```

```
foo 17
bar 32
baz 42
qux 87
quux 3
```

```
cat bar.txt
```

```
foo rouge
bar vert
baz bleu
qux noir
quux blanc
```

On souhaite combiner les données des deux fichiers et faire une jointure sur le 1er champ. Pour cela, il faut lire les deux fichiers dans le même script et faire un test sur la variable `FILENAME` qui contient le nom du fichier en cours de traitement :

```
awk '
FILENAME == "foo.txt" {
    number[$1] = $2
}
FILENAME == "bar.txt" {
    color[$1] = $2
}
END {
    for (key in number) {
        printf("%-7s %6d %s\n", key, number[key], color[key])
    }
}
' foo.txt bar.txt
```

```
foo         17 rouge
baz         42 bleu
qux         87 noir
quux         3 blanc
bar         32 vert
```

Pour chaque fichier, on constitue un tableau associatif avec une clé commune. Dans la section `END`, on boucle sur la clé pour chaque tableau associatif.

9 Quelques preuves

Pour voir les valeurs par défaut de `RS` et `FS` :

```
echo dummy | awk '{ printf("RS = |%s|\nFS = |%s|\n", RS, FS) }'
echo
echo dummy | awk '{ printf("RS = |%s|\nFS = |%s|\n", RS, FS) }' | cat -vte
```

```
RS = |
|
FS = | |

RS = |$
|$
FS = | |$
```

On voit bien que RS est un passage à la ligne et FS une espace.

10 Maintenant de vrais exemples, vraiment utiles

L'idée est d'illustrer quelques éléments vus ici ainsi que de nouveaux éléments dans le contexte d'outils réels et utiles.

10.1 Recherche dans les logs d'Airwave

Soit le fichier `client_all_1662019197.csv` un export d'Airwave.

On recherche :

- le SSID eduroam

- les connexions sur les AP wifi de Meudon (`ap-m-`)
- dans une plage de dates donnée

```
debut=$(date '+%s' -d '2022-08-01 00:00')
fin=$(date '+%s' -d '2022-08-31 23:59')
awk -F, -v OFS=, -v debut=$debut -v fin=$fin '
{
    username     = $1
    ssid         = $2
    mac          = $3
    connect_time = $4
    last_ap_id   = $5
    if ( ssid         ~ /eduroam/ &&
         last_ap_id   ~ /ap-m-/   &&
         connect_time >= debut    &&
         connect_time <= fin      ) {
        cmd = "date -d @" connect_time
        cmd | getline human_date
        close(cmd)
        print username, ssid, mac, human_date, last_ap_id
    }
}' airwave_client_all_1662019197.csv | column -t -s,
```

Les concepts qui sont illustrés par cet exemple :

- passage de variables entre le script shell enveloppe et Awk avec l'option `-v`
- modification du séparateur de sortie `OFS`
- appel à une commande shell externe et récupération du résultat avec `cmd | getline`

10.2 Dédoublonnage de statistiques Git

La commande `git shortlog --summary --numbered --email` est très pratique mais il peut y avoir des doublons, par exemple :

```
  1650<TAB>Stéphane Aicardi <stephane.aicardi@obspm.fr>
   538<TAB>Stephane Aicardi <stephane.aicardi@obspm.fr>
[...]
   444<TAB>Stephane Vaillant <Stephane.Vaillant@obspm.fr>
   413<TAB>Stephane Vaillant <stephane.vaillant@obspm.fr>
```

On veut consolider les statistiques en ne prenant que l'adresse e-mail, normalisée en minuscules.

```
cd ~/git-repo/puppet
git shortlog --summary --numbered --email | awk -F'\t' '
{
    gsub(/^.*</, "", $2)
    gsub(/>/, "", $2)
    e_mail = tolower($2)
    commits[e_mail] += $1
}
END{
    for (e_mail in commits) {
        printf("%5d %s\n", commits[e_mail], e_mail) | "sort -k1,1nr"
    }
}'
```

Les concepts qui sont illustrés par cet exemple :

- modification du délimiteur de champ (`FS`) à `\t`

- utilisation de `gsub()` sur le 2e champ pour supprimer tout ce qui est avant et après l'adresse e-mail
- utilisation de `tolower()` pour normaliser en minuscules
- utilisation d'un tableau associatif avec l'e-mail comme clé et comme valeur le cumul des 1ers champs
- utilisation d'une section `END` pour afficher les résultats finaux
- appel à une commande shell sans réutilisation du résultat avec `| "commande"`

10.3 Affichage atténuation signal optique sur les Juniper

Sur les routeurs Junipers, on cherche à afficher de façon synthétique l'atténuation du signal optique reçu pour chaque interface comportant un *transceiver* SFP optique.

C'est ce que fait le script `/usr/local/bin/releve-attenuation-sfp-juniper`, qui s'appuie sur un script Awk `/usr/local/bin/releve-attenuation-sfp-juniper.awk`.

Le script est un peu long pour être intégré ici, le visualiser directement sur `sionet`.

Les concepts qui sont illustrés par cet exemple :

- utilisation de deux fichiers dont on combine les données (« jointure »)
- utilisation de tableaux associatifs
- utilisation de `split()` en cascade pour isoler la donnée voulue
- formatage un peu évolué de l'affichage avec `printf()`
- appel à la commande shell `sort` dans un script Awk

10.4 Reboot via PoE de tout ce qui correspond à un *device type* LLDP

Pour des raisons que certaines personnes devinent aisément, on a le besoin de rebooter électriquement, en même temps, tout les AP wifi d'un campus. C'est l'objet du script `poe-reboot-device-type`.

On est capable d'avoir le port du switch PoE pour chaque AP wifi avec la commande

```
netdisco-search-neighbours-of-device-type
```

On souhaite agréger ces données pour avoir pour chaque switch PoE l'ensemble des ports occupés par les AP wifi pour faire un *off/on* via PoE. C'est l'objet du script Awk embarqué dans le script `poe-reboot-device-type` que l'on va lire ensemble.

L'extrait intéressant est le suivant :

```
awk -F"|" -v rancid_cmd_pattern="$RANCID_CMD_PATTERN" '
  {
      sw   = $1
      port = $2
      # Concatène les ports dans un hash, clé = switch
      if (! ports[sw]) {
          ports[sw] = port
      } else {
          ports[sw] = ports[sw] "," port
      }
  }
  END {
      for (s in ports) {
          printf(rancid_cmd_pattern, ports[s], s)
      }
  }' $tmp_switch_ports_filtered > $tmp_shell_script
```

Les concepts qui sont illustrés par cet exemple :

- passage de variables entre le script shell enveloppe et Awk avec l'option `-v`
- utilisation d'un tableau associatif avec le nom du switch comme clé et comme valeur la concaténation au fil de l'eau des ports
- utilisation d'une section `END` pour utiliser les résultats finaux et construire un script shell exécutable

11 Liens utiles

Il y a un gadzillion de tutoriels Awk sur le web, je ne saurais pas en recommander un en particulier. J'aime bien le livre O'Reilly « Sed & Awk ».

11.1 Awk reference card (10 pages)

On trouve aussi plein de *reference cards*. J'en recommande une :

http://ewald.cas.usf.edu/teaching/2004F/5156/packet/packet2.docs/07.gawkcard.pdf