Tutoriel awk

par nyal

Date de publication : 10/01/2005

Dernière mise à jour :

Voici un tutoriel sur awk et ses méandres. J'espère qu'il vous sera utile...

I - Introduction

I-1 - Objet et objectifs

Ce tutoriel a pour but de présenter toutes les possibilités de gawk. Des bases jusqu'aux possibilités les plus poussées. Bien sur, ce tutoriel n'est pas exhaustif mais il essaye de s'en approcher en proposant des exemples concrets de code. Ces exemples sont extraits de fichiers et non de lignes de commande shell. Une partie est consacrée pour l'utilisation de gawk directement en ligne de commande.

Pour bien comprendre l'utilité de gawk, les connaissances de bases du shell doivent être acquises.

I-2 - Présentation

Awk est un langage de programmation datant de 1977, date de son apparition dans le monde Unix. Il tire son nom des trois programmeurs qui l'ont développé : Alfred V. Aho, Peter J. Weinberger et Brian W. Kernighan.

Cette utilitaire a été crée dans le but de remplacer les commandes grep et sed. Sa grande souplesse lui a permis de connaître un succès immédiat. Et de nouvelles versions sont apparues au fil du temps : nawk et gawk aujourd'hui.

Aujourd'hui encore, cet utilitaire est toujours utilisé du fait de sa ressemblance avec le langage C, de sa souplesse et de sa présence sur la majorité des systèmes d'exploitation Unix. Il est encore utilisé en administration système et dans les scripts Shell en tant que commande.

II - Un programme awk

II-1 - Fonctionnement

Awk fonctionne en lisant des données. Ces données peuvent être ainsi traitées par l'utilisateur. Utilisateur qui peut choisir de lire des données provenant de fichiers ou du canal de l'entrée standard (via un pipe par exemple).

Par conséquent, Awk a pour but premier de jouer un rôle de filtre bien qu'il ne se limite pas qu'à cela.

Nous allons maintenant présenter en détail les différentes façons de lancer un programme awk.

La première manière est d'insérer en haut du fichier awk une ligne qu'on nomme shebang dont voici la syntaxe :

```
#! /usr/bin/awk -f
```

Pour connaître le chemin de l'exécutable awk, il vous suffit d'utiliser la commande suivante sous votre Shell :

```
nyal bash $ which awk
/usr/bin/awk
nyal bash $
```

Il ne vous reste plus qu'à mettre les droits d'exécution au fichier et vous pourrez le lancer comme un exécutable.

Une autre méthode consiste à lancer le fichier en utilisant la commande awk sous un Shell avec l'option -f. Voici différents lancement possibles :

```
nyal bash $ awk -f fichier_awk < fichier_texte
nyal bash $ ls | awk -f fichier_awk
```

Maintenant que vous savez comment lancer un programme awk, nous allons expliquer la structure d'un programme.

II-2 - Structure

Nous allons étudier comment awk nous permet de lire les données d'un fichier en expliquant les trois parties qui composent un programme en awk. Un programme Awk est composé de lignes de programmations ainsi que de lignes de commentaire. Les commentaires commencent après le caractère #.

II-2-a - Le coeur du programme

Ce coeur va être composé de conditions et d'actions qui vont être exécutées si la condition (ou critère) est remplie :

```
Condition { Action }
...
Condition { Action }
```

L'accolade ouvrante doit être sur la même ligne que la condition, sinon une erreur va être produite. Un moyen de l'éviter est de faire :

```
Condition \
  {
    Action
  }
...
Condition {
    Action
}
```

Voici donc d'autres syntaxes possibles. Vous pouvez bien entendu enchaîner les conditions/actions sur une même ligne. Et si une action ne possède pas de condition, elle est exécutée automatiquement.

Une condition est une expression qui va être analysée par awk. Si cette condition est réalisée alors l'action va être exécutée. Cette action peut correspondre à plusieurs instructions sur une ou plusieurs lignes. Si deux instructions se suivent sur la même ligne, il faut alors les séparer par un point-virgule :

```
1 < 2 {
    print "Hello World"
    print "Hello World 2"; print "Again" }
```

Maintenant nous allons voir comment awk exécute ce programme.

Awk va rappeler le coeur du programme qui est composé de critères/actions pour chaque enregistrement. Un enregistrement est représenté le plus souvent par une ligne. Par conséquent, si vous avez un fichier de 4 lignes que vous allez traiter, votre coeur du programme va être appelé quatre fois. Et pour chaque passage, c'est un nouvel enregistrement qui va être traité.

Bien sûr vous vous demandez comment accéder a l'enregistrement. Il y a pour cela un ensemble de variables à disposition, rempli par awk. Cela est expliqué en détail dans le chapitre suivant.

Il reste maintenant à expliquer les deux autres parties d'un programme en awk.

II-2-b - Le démarrage : BEGIN

Au démarrage d'un programme, awk peut exécuter des instructions avant le coeur du programme.

Ces instructions doivent être placées dans un bloc qui se nomme BEGIN

```
BEGIN {
    instructions
}
```

Il peut y avoir plusieurs blocs BEGIN. Ils seront exécutés les uns après les autres n'importe où dans le fichier :

```
BEGIN {
    print "démarrage1"
}
Coeur du programme
...
BEGIN \
{
    print "démarrage2"
}
```

Le bloc BEGIN doit être obligatoirement suivi de son accolade ouvrante sur la même ligne. (A moins d'utiliser le caractère backslash devant le retour chariot)

Les bloc BEGIN sont très utiles pour initialiser des variables et donc préparer la suite du programme.

II-3-c - La fin : END

Contrairement aux blocs BEGIN, les blocs END sont exécutés à la fin du programme. Une fois que tous les enregistrements ont été traités par le coeur du programme. Il possède les mêmes propriétés que le bloc BEGIN :

```
END {
    print "fin du programme";
}
END \
{
    print "C'est vraiment fini"
    print ".."
}
```

Les blocs END peuvent permettre d'afficher des résultats du coeur du programme ainsi que d'autres messages.

Vous savez maintenant comment se compose un programme Awk. Nous allons maintenant voir toutes les possibilités du langage.

```
# Format d'un programme awk

BEGIN {
    print "Démarrage du programme"
}

{ print "Pas de critères. Donc ce message s'affiche autant de fois qu'il y a d'enregistrement" }

END {
    print "Fin du programme"
}
```

III - Les variables

III-1 - Les types

Awk est un langage non typé, c'est à dire que vous n'avez pas besoin de déclarer les variables selon ce que vous voulez stocker dedans. Ce système est donc différent de celui du C pour permettre une plus grande souplesse.

Awk propose deux types de variables :

- Les variables utilisateur : variables simples et tableaux associatifs
- Les variables systèmes

Ces variables vous permettront de développer des programmes en réalisant des opérations arithmétiques, en stockant des données, ...

III-2 - Utilisation

Une variable est composée de lettres, chiffres et soulignés (underscore). La taille du nom n'est pas limitée mais le premier caractère se doit d'être différent d'un chiffre.

Contrairement à un langage comme le C, les variables n'ont pas besoin d'être déclarées et une place mémoire leur est réservée. De plus, la variable est affectée soit de la valeur 0 ou d'une chaîne vide "" selon le contexte par défaut.

Vous pouvez affecter à une variable des valeurs entières, décimales et ascii.

```
variable1 = 2
variable_2 = "Hello world"
variable_3 = 1.56
```

Ce système de variable est géré par awk. Awk va décider s'il faut selon le contexte convertir ou non la variable en texte ou nombre.

Ainsi si la variable est utilisée pour une comparaison avec du texte ou un affichage, elle sera convertie en texte. Et si une chaîne de caractères (dans une variable aussi) est employée dans une opération arithmétique, awk tentera la conversion en une valeur numérique. Mais si cette conversion est impossible (la chaîne commence par un caractère différent d'un chiffre), awk affectera la valeur 0 pour la chaîne.

Par conséquent, vous pouvez savoir si une variable est de type numérique en utilisant le système d'interprétation de awk :

```
i = i + 0 # ou pour une condition aussi
i + 0
```

Awk va essayer de transformer la variable i en une valeur numérique. Et ainsi, si la variable commence par une valeur autre qu'un chiffre, la valeur 0va être affectée dans l'opération. Ce qui fera 0 + 0 et donc la condition ne sera pas remplie. La valeur 0 signifie que la condition n'est pas réalisée et que l'action ne sera pas exécutée.

III-3 - Les tableaux associatifs

Les tableaux sont des variables qui associent des indexes avec des valeurs. Ces indexes peuvent être soit des nombres, soit des chaînes de caractères :

```
tableau[0] = "Hello"
tableau[1] = "World"
tableau["question"] = 1
```

Il est possible d'utiliser comme indexe une variable prédéfinie auparavant:

```
val = "question"
tableau[val] = 1
```

Awk possède des systèmes de boucles pour les tableaux associatifs. Ils seront abordés dans le chapitre concernant les systèmes de contrôle.

Enfin il n'y a pas de syntaxe permettant d'assigner plusieurs valeurs à un tableau directement. De même les valeurs d'un tableau ne peuvent redevenir indéfinies.

III-4 - Les variables système

Awk découpe les enregistrements en champs. Les enregistrements sont découpés selon les espaces et tabulations. Chaque champ est affecté dans les variables $1, $2, $3 ... Le numéro correspondant au nombre de champs de l'enregistrement. $0 permet d'accéder à la totalité de l'enregistrement.

Pour travailler avec ces variables, le signe $ peut être suivi d'une variable. La valeur définit alors le champ concerné dans l'enregistrement :

```
i = 1
print $i # Affiche le champ numéro 1
```

Ainsi vous pouvez utiliser la variable système NF pour afficher le dernier champ d'un enregistrement :

```
print $NF
```

Variable	Descriptif	Valeur par défaut
ARGC	Cette variable contient le nombre d'arguments provenant de la ligne de commande. Les options au programme gawk ne sont pas prises en compte.	
ARGIND	Cette variable permet de connaître quel fichier est en cours en traitement en contenant l'indexe de ARGV.	
ARGV	Ce tableau contient les arguments de la ligne de commande. Il est indexé de 0 à ARGC - 1. Vous pouvez changer dynamiquement le contenu du tableau. Cela a pour incidence de changer les fichiers en traitement par	

Variable	Descriptif	Valeur par défaut
	exemple.	
CONVFMT	La variable contient le format de conversion des nombres.	%.6g
ENVIRON	Tableau contenant l'environnement courant. Les éléments sont indexés par le nom des variables d'environnement (ENVRION["PATH"]). Si vous changez les valeurs du tableau, les programmes qui seront lancés par awk utiliseront toujours l'environnement de base.	
FILENAME	La variable contient le nom du fichier en cours de traitement. Si aucun fichier n'est spécifié alors FILENAME contient '-' et l'entrée standard est utilisée. FILENAME n'est pas encore défini dans les blocs BEGIN.	
FNR	La variable contient le numéro de l'enregistrement pour le fichier courant. FNR reprend la valeur 1 si un nouveau fichier est traité.	
FS	contient la chaîne permettant la séparation des champs. FS peut contenir une expression régulière.	La caractére espace
IGNORECASE	Cette variable permet de contrôler la casse pour les expressions régulières et les opérations sur les chaînes. Si IGNORECASE contient une valeur différente de 0 alors la casse n'est plus prise en compte.	0
NF	La variable contient le nombre de champs pour un enregistrement. (Les champs sont séparés par des espaces et tabulations)	
NR	NR contient le nombre total d'enregistrements depuis le début du script.	
OFMT	Le format de sortie pour les nombres.	%.6g
ORS	ORS contient la chaîne permettant la séparation des enregistrements. La chaîne peut être constituée d'une multitude de caractères.	\n

Variable	Descriptif	Valeur par défaut
SUBSEP	La variable contient le caractère utilisé pour séparer les différents élément définissant un indexe du tableau. L'utilisation de cette variable sera primordiale pour l'émulation des tableaux multi dimensions.	\034 (fs)

III-5 - Exemples

Ce script permet de changer le formatage d'affichage du fichier /etc/passwd :

```
#! /usr/bin/awk -f

BEGIN {
    FS=":" # Changement du séparateur de champ
}
{
    print "Utilisateur : " $1
    print "Répertoire Courant : " $6
    print ""
}
```

IV - Les opérateurs

IV-1 - Les opérateurs de comparaisons

Ces opérateurs servent dans les conditions (ou structures de contrôles) pour comparer les valeurs numériques ou des chaînes de caractères. Il n'y a pas de différences d'opérateurs pour ces comparaisons. Awk fera la comparaison selon le contexte.

Opérateur	Signification
==	égale à
!=	différent de
<=	inférieur ou égale à
>=	supérieur ou égale à
<	inférieur à
>	supérieur à

Ces opérateurs peuvent être employés avec des variables.

Pour comparer le contenu d'une variable avec une chaîne fixe, cette chaîne doit être placée entre guillemet contrairement aux valeurs numériques qui n'ont pas besoin de guillemet :

```
$2 == "Hello"
NR > 2
nb_person >= 4
```

Ces comparaisons vont pouvoir être liées par des opérateurs dits logiques.

IV-2 - Les opérateurs logiques

Opérateur	Signification
&&	Le ET-Logique
||	Le OU-Logique
!	La Négation

Les deux premiers opérateurs vont permettre d'enchaîner les comparaisons. Le **&&** va tester la comparaison suivante si la première est vérifiée :

```
NR > 1 && NR < 20
```

Dans cette exemple, la condition est vérifiée si la ligne d'enregistrement est supérieure à 1 **ET** inférieure à 20. Mais si une des deux comparaison est faussée alors la condition n'est pas vérifiée.

Contrairement au ET-Logique, le **||** va tester la condition suivante si et seulement si, la comparaison précédente n'est pas vérifiée :

```
NR == 6 || NR == 20
```

La condition est vérifiée si la ligne d'enregistrement est la 6 **OU** la 20.

Vous pouviez associer des comparaisons grâce aux parenthèses :

```
( NR >= 1 && NR < 20 ) || NR == 34
```

La comparaison à droite **||** sera testée si la comparaison (totalité) entre parenthèses n'est verifiée.

Enfin, l'opérateur de négation va définir comme vraie, le contraire de la comparaison (ou d'un ensemble de comparaisons entre parenthèses) :

```
!( NR >= 1 && NR < 20 )
```

La condition sera donc vrai pour les enregistrements qui ne sont pas supérieurs à 1 et inférieurs à 20.

Lors de ces comparaisons, il sera possible d'utiliser des opérateurs arithmétiques.

IV-3 - Les opérateurs arithmétiques

Opérateur	Signification
+	Addition
-	Soustraction
*	Multiplication
/	Division
%	Modulo (reste d'une division entière)
^	Puissance

Ces opérateurs peuvent servir dans les conditions ou l'affectation, et la manipulation des valeurs numériques des variables :

```
NR % 2 == 1 {
    nb_ligne_impaire = nb_ligne_impaire + 1
}
```

Les variables sont souvent objets de manipulations arithmétiques (comme dans l'exemple), il existe donc des formules pour abréger l'écriture :

Formule	Equivalence
my_var = my_var + Valeur	my_var += Valeur
my_var = my_var - Valeur	my_var -= Valeur
my_var = my_var * Valeur	my_var *= Valeur
my_var = my_var / Valeur	my_var /= Valeur
my_var = my_var % Valeur	my_var %= Valeur

Enfin, nous allons voir les derniers opérateurs qui sont aussi connus dans la plupart des autres langages. Ce sont les opérateurs de décrémentation et d'incrémentation :

Opérateur	Signification
++	Incrémentation (Rajoute +1 a la valeur de la variable)
--	Décrémentation

Opérateur	Signification
	(Enlève -1 a la valeur de la variable)

Ces opérateurs doivent être utilisés avec prudence. Nous allons voir pourquoi :

```
i = 10
j = i++
print j, i
```

Cet extrait de programme va afficher 10 pour **j** et 11 pour **i**. Ceci peut paraître troublant mais cela se produit car la valeur de la variable **i** n'est augmentée qu'après l'affectation. L'incrémentation se passe après l'affectation car l'opérateur se situe après la variable (en suffixe).

Si vous aviez placés l'opérateur en préfixe, alors l'incrémentation se serait passée avant l'affectation :

```
i = 10
j = ++i
print j, i
```

Dans cet extrait, les deux variables ont la même valeur : 11.

IV-4 - Exemples

Ce script permet d'afficher les lignes paires d'un fichier :

```
#! /usr/bin/awk -f

NR % 2 == 1 { print $0 }
```

Ce script permet d'afficher les 7 premières lignes d'un fichier avec le numéro (demarrant a 1) :

```
#! /usr/bin/awk -f

BEGIN {
    n = 7
}

NR <= n { print NR, $0 }
```

Ce script peut être écrit de la manière suivante :

```
#! /usr/bin/awk -f

NR <= n { print NR, $0 }
```

Ainsi il affichera les **n** premières lignes des fichiers. Mais il faudra l'appeler de la manière suivante avec le shell :

```
nyal $ ./aff_n_lines.awk n=7 fichier_texte
```

L'argument **b=7** ne sera pas pris comme un fichier à lire à cause de sa syntaxe avec le égale. Awk assignera dans le script la valeur à la variable. Ainsi 7 sera assigné à la variable n. Cette valeur est affectée avant que le script soit lancé.

V - Structures de contrôles

V-1 - Introduction

Le langage awk ne serait pas un langage complet si on ne pouvait agir sur le déroulement du script. Ainsi awk dispose de structures de contrôles.

Si vous avez déjà utilisé un langage comme le C, vous vous êtes déjà familiarisés avec les structures qui sont quasiment toutes les mêmes (avec des rajouts). Attention, les structures de contrôles ne peuvent être utilisées en tant que condition première (ou critère).

V-2 - Les décisions

L'expression **if** permet l'exécution d'une ou plusieurs instructions en fonction de la réalisation de la condition. Une condition se définit avec les opérateurs logiques et de comparaisons. (cf. chapitre précédent)

```
if (condition)
    instruction
```

Vous pouvez regrouper plusieurs instructions en utilisant les accolades. L'expression **if** peut être suivie d'une expression **else** qui sera réalisée si l'expression **if** échoue.

```
if (condition)
{
    instruction1
    instruction2
}
else
{
    instruction1
    instruction2
}
```

V-3 - Les boucles

V-3-a - Les boucles while

Awk dispose de deux types de boucles while. Voici leur syntaxe :

```
while (condition)
{
    instructions
}

do
{
    instructions
} while (condition)
```

La boucle **do-while** exécute les instructions une fois avant de tester la condition. Par conséquent les instructions sont exécutées au moins une fois contrairement à la boucle **while** classique ou **for**.

V-3-b - Les boucles for

Comme pour les boucles **while**, il existe deux types de boucles **for** :

```
for (instruction de début; condition; instruction de comptages)
{
    instructions
}

for (variable in Tableau)
{
    instructions
}
```

La première boucle **for** est la boucle classique présente dans de nombreux langages de programmation (C, Perl, ...). Elle peut être remplacée trés facilement par une boucle **while** et inversement.

La seconde boucle est plus originale. Elle permet de parcourir un tableau associatif. Un tableau associatif peut être indexé par des chaînes de caractères. Il est donc beaucoup plus difficile de parcourir le tableau. Par conséquent, awk propose cette boucle (Elle permet de parcourir n'importe quel tableau. Qu'il soit indexé par des nombres ou des caractères alphanumériques). Malheureusement, les valeurs ne seront pas triées. Mais il existe des méthodes pour trier que nous verrons plus tard.

V-4 - Les sauts contrôlés

Les sauts contrôlés vont permettre d'interrompre un traitement. Cependant awk ne dispose pas d'instructions tel le **goto**. Elles rendent souvent le code confus. Cependant les sauts de contrôles permettront de remplacer avantageusement le goto.

V-4-a - Les contrôles de boucles

Dans cette catégorie, ce sont les mots clés **break** et **continue** qui sont utilisés.

- Le **break** permet de sortir de la boucle active,
- Le mot clé **continue** permet de faire un nouveau passage dans la boucle (Les instructions après **continue** ne seront pas prises en compte). Cependant le nouveau passage se fera si et seulement si la condition est remplie.

Il faut savoir qu'il n'y a pas de moyen de sortir de deux boucles imbriquées grâce à un seul **break** (ou de faire un **continue**). Ces mots clés ne fonctionnent que sur la boucle active.

V-4-b - Les contrôles du programme

Cette catégorie est composée des mots clés **next** et **exit**. Lorsque awk rencontre le mot clé next, il arrête le traitement de l'enregistrement en cours et il passe au suivant. Tandis que **exit** permet l'arrêt complet des traitements. Awk exécute tout de même les blocs END. (Si vous mettez un **exit** dans un bloc END, alors le programme s'arrête définitivement)

V-5 - Exemples

V-5-a - Le script PATH

Ce script permet de récupérer le chemin d'un fichier à partir d'un chemin complet (avec le fichier) :

script PATH
```
#! /usr/bin/awk -f

BEGIN {
    FS="/"
}
{
    path=""
    for (i = 1; i < (NF); i++) {
       path = path $i "/"
    }
    print path
}
```

Voici le résultat :

```
nyal $ cat fichier_test
./home/nyal/.xsession
./home/nyal/.Xdefault
/etc/rc.d/rc.modules
/etc/termcap
nyal $ cat fichier_test | ./path.awk
./home/nyal/
./home/nyal/
/etc/rc.d/
/etc/
```

V-5-b - le script STOCKAGE PASSWD

Ce script permet juste de montrer les possibilités des tableaux associatifs. Il stocke les utilisateurs du fichier passwd (/etc/paswd) avec son home (chemin du compte).

script STOCKAGE PASSWORD
```
#! /usr/bin/awk -f

BEGIN {
    FS=":"
}
{
    tabl_stock[$1] = $6
}
END {
    for (name in tabl_stock) {
       print "Nom : " name " | Home : " tabl_stock[name]
    }
}
```

V-5-c - Le script REVERSE LINE

Ce script permet de renverser les champs d'un enregistrement :

script REVERSE LINE
```
#! /usr/bin/awk -f

# ENTRE : Comment allez vous ?
# SORTIE : ? vous allez Comment

{
    reverse=""
    i = NF
    while (i > 0) {
       reverse = reverse $i FS
       i--
    }
    print reverse
}
```