C02-script-user_advenced-variable-vim - COURS LINUX - MTN

Guillaume ASTIER

26/02/16

Table des matières

Scripting Premier pas

Qu’es-ce qu’un script

Le but ici est de comprendre comment réaliser un script, de le rendre executable, d’automatiser certains processus

Un langage de script est un langage de programmation qui permet de manipuler les fonctionnalités d’un système informatique configuré pour fournir à l’interpréteur de ce langage un environnement et

une interface qui déterminent les possibilités de celui-ci. Le langage de script peut alors s'affranchir des contraintes de bas niveau — prises en charge par l'intermédiaire de l'interface — et bénéficier d'une syntaxe de haut niveau.

script bash

Pour linux et plus précisément en bash (en tant que language) un script consiste à un enchainement de

```
* commande

* script

* variables (local ou environement)
```

le tout dans un contexte qui peut être :

```
*  linéaires

* imbriquées

* conditionnelles

* fonctionnelles
```

Exemple et explication :

Nous allons ici examiner un script et le comprendre ensemble

```
#!/bin/bash
VAR=$1
NBRE=$(echo $VAR|wc -c)
echo "Votre argument à $0 est $VAR"
echo "Il contient *NBRE caractères(s)"
```

Shebang

- Que signifie '#!/bin/bash'

Le shebang, représenté par #!, est un en-tête d'un fichier texte qui indique au système d'exploitation (de type Unix) que ce fichier n'est pas un fichier binaire mais un script (ensemble de commandes) ; sur la même ligne est précisé l'interpréteur permettant d'exécuter ce script.

Vous pouvez retrouver des shebang en python, perl, sh, ksh

Les structures de contrôle:

if/fi for/done while/done case

Les grandes structures de contrôles sont les suivantes :

- if : Conditions
- for : Pour tous les
- while : Tant qu'il y a
- case : pour les cas

Structure du if :

- La condition "if" se contruit de cette manière :

```
if [[ CONDITION ]]
    then
        # Si la condition est remplie fait une action
    else
        # Si la condition n'est pas remplie fait une action
fi
```

- Peut-être ecrit autrement :

```
[[ CONDITION ]] && action si vrai || action si pas vrai
```

Exemple pour le if :

- Complet

```
if [[ -f /tmp/un_fichier ]]
    then
        echo "Le fichier existe"
    else
        touch /tmp/un_fichier
        echo "Creation du fichier"
fi
```

- Rapide

```
[[ -f /tmp/un_fichier ]] && echo "Le fichier existe" && \
touch /tmp/un_fichier && echo "Creation du fichier"
```

Structure du for :

- Ex avec des opé arithmétique :

```
for var in UN DEUX TROI
    do
        cpt=$((cpt+1))
        echo "$var"
    done

retour:

1 UN
2 DEUX
3 TROIS
```

exemple du for:

- Ex avec retour commande et parsing de fichier : -> le fichier /tmp/liste contient :

```
$ cat /tmp/liste
je suis un fichier
sur deux lignes
$ for var in 'cat /tmp/liste '
    do
        cpt=$((cpt+1))
        echo $var
    done
echo "le fichier /tmp/liste contient $cpt mots"
```

exemple du for (retour de la commande)

Retournera :

```
je
suis
un
fichier
sur
deux
lignes
le fichier /tmp/liste contient 7 mots
```

Structure du while :

```
while [[ condition ]] # tant que la condition est vrai
    do
        # fait quelque chose
    done
```

Attention : on peut viste rentrer dans une boucle infinie si l'on n'a pas préciser une confition correct ou un break dans la boucle.

Exemple du while:

```
while [[ "$CPT" != "5" ]]
    do
        CPT=$((CPT+1))
        echo "Je suis à ${CPT} secondes sur 10"
        sleep 1
    done
```

Le retour :

```
Je suis à 1 secondes sur 10
Je suis à 2 secondes sur 10
Je suis à 3 secondes sur 10
Je suis à 4 secondes sur 10
Je suis à 5 secondes sur 10
```

Exemple du while avec fichiers :

```
$ cat /tmp/liste
je suis un fichier
sur deux lignes
$ while read var
        do
                cpt=$((cpt+1))
                echo $var
        done < /tmp/liste
echo "le fichier /tmp/liste contient $cpt lignes"
```

Retournera :

```
je suis un fichier
sur deux lignes
le fichier /tmp/liste contient 2 lignes"
```

Structure de case :

Le principe du case est de compartimenter la réponse.

```
case "$var" in
    ok) echo "processus peut continuer" ;;
    ko) echo "processus doit être arrêté" ; exit 1;;
    *) echo "Aucun retour pour l'instant"
esac
```

Les fonctions

Syntaxe

Le bash propose plusieurs syntaxes pour définir une fonction.

```
maFonctionA(){
    echo 'Hello'
}
maFonctionB(){
    echo 'Word'
}

maFonctionA
maFonctionB
```

Pipe et esperluette

OU/ET (‘||’ et ‘&&’)

En programmation les deux grandes conditions sont le ET ‘&&’ et le OU ‘||’

Si nous prenons l’exemple du if :

```
if [[ "$VAR" == "YES" ]] || [[ "$VAR" == "OK" ]]
        then ... fi

if [[ "$VAR" == "TRUE" ]] && [[ -f /tmp/file_to_populate ]]
        then ... fi
```

Si la commande précédente ...

La notion de ET et de OU peut être aussi utiliser dans des suites de commandes :

- ‘CMD1 && CMD2’ : implique que si la commande CMD1 c’est bien passé alors on lancera la CMD2
- ‘CMD1 || CMD2’ : implique que si la commande CMD1 à retrouné une erreur alors seulement on lanera CMD2

En mode Système (| et &)

- Le cas du PIPE

Le pipe simple ’‘|’’ sous linux indiquer que nous allons rediriger le résultat d’une commande par exemple dans une autre commande :

Ex avec cat et grep.

```
1 cat FILE | grep CHAINE
```

Ici nous allons afficher sur le terminal l’intégralité du fichier FILE mais nous allons filtrer gràce à grep uniquement la chaine de caractère CHAINE

Le cas du ET commercial

Dans le cas ou vous ne voulez pas attendre la fin d’une commande avant de pouvoire reprendre la main sur le processus parant de cette comamnde (génèralement la bash C.A.D votre terminal), vous aller utiliser le ET commercial.

Pour que la démonstration soit plus visuelle et compréhensive, il suffit de lancer un editeur de texte graphique depuis le terminal. Tant que vous n’avez pas quiter ce dernier, vous ne pourrez pas utiliser le temrinal depuis lequel vous avez lancé gedit (par exemple).

Cependant si vous lancez ce dernier en ajoutant &, gedit se lancera et le terminal vous rendra la main.

Gestion des utilisateurs et des groupes (suite)

PAM

PAM (pour Pluggable Authentication Modules) est un système inventé par SUN Microsystems permettant d’intégrer simplement diveres stratégies liées à l’authentification. Depuis 2006, PAM s’est repandu sur les systèmes Linux, Solaris,BSD et même les Unix propriétaires comme IBM-AIX ou HP-UX.

Pour pousser plus loin PAM à un module ‘pam_ usb’ qui permet de s’authentifier avec une clef usb qui contient une partition cryptée. Aucun mot de passe ne vous sera demandé.

PAM permet de changer la façon dont le système va identifier un utilisateur (globalement, ou pour un service donné)

Kerberos

Kerberos est un protocole d’authentification réseau qui repose sur un mécanisme de clés secrètes (chiffrement symétrique) et l’utilisation de tickets, et non de mots de passe en clair, évitant ainsi le risque d’interception frauduleuse des mots de passe des utilisateurs On peut associer Kerberos sous linux à AD (Active Directory) sous windows

NSS

Le Name Service Switch (NSS) autorise le remplacement des traditionnels fichiers Unix de configuration (par exemple /etc/passwd, /etc/group, /etc/hosts) par une ou plusieurs bases de données centralisées. Les mécanismes utilisés pour accéder à ces bases étant configurables.

Autre type de contrôle d’accès : sudo

sudo est un programme conçu pour permettre à un administrateur système de donner des privilèges d’administration limités aux utilisateurs et d’enregistrer dans un journal les actions de l’administrateur (« root ». sudo ne demande que le mot de passe d’un utilisateur normal

aspects de l’authentification

- account : validité du compte, expiration du mot de passe, …
- authentification : vérification de l’identité de l’utilisateur
- password : modification du mot de passe
- session : liste des tâches à effectuer avant la mise à disposition du service (ou après sa terminaison)

configuration de PAM

PAM se configure dans :

- /etc/pam.conf

C'est LE fichier de configuration global

- /etc/pam.d/

C'est le répertoire contenant des fichiers de configuration supplémentaire. Ce path absolu doit être renseigné dans le fichier de configuration global.

Session

Script de démarrage

Les scripts exécutés quand bash est utilisé pour une ouverture de session:

- '/etc/profile' (commun à tous les utilisateurs)
- '$HOME/.bash_profile'
- '$HOME/.bash_login'
- '$HOME/.profile'

Script exécuté lors de la déconnexion

- '$HOME/.bash_logout'

Script exécuté quand bash n'est pas utilisé pour une ouverture de session:

- '$HOME/.bashrc'

Les variables

Declaration de variable

- Que signifie VAR=$1

VAR dans un prmemier temps est le nom arbistraire d'une variable que l'on a nomé. Elle va récupérer la valeur de $1

$1 est le contenue de la chaine passé en argument au script.

Par exemple si le script s'appel 'nombre_de_caracteres.sh' et qu'on l'éxecute avec un arguement comme ci dessous :

```
./nombre_de_caracteres.sh ma_chaine_de_caractere
```

$1 sera égale à 'ma_chaine_de_caractere'

Affectation simple:

```
    $ variable=chaîne
    $ read variable
    chaîne
```

Accèder au contenue de la variable :

```
    $ echo $variable
    $ echo ${variable}     // mieux
```

Utiliser des accolades comme '$VAR' est préférable car permet de protéger le nom de votre variable

Affectation exporté

```
$ export variable=chaîne

$ variable=chaîne; export variable
#sur vieux shells
```

Exporter une variable permet l'accès à cette dernière depuis les sous-shells et les porcessus enfants.

Se comporte de la même manière que des variables 'classiques'

Variable auto (Récupération de données dans un script)

```
$0 : nom du script.
$1 : argument numéro 1
$2 : argument numéro 2
$3 : argument numéro 3
```

```
...
$# : nombre d'arguments passés
$* : tous les arguments concaténés en une chaîne
     unique.
$@ : tous les arguments transformés individuellement
     en chaîne.
```

IFS (Internal Field Separator)

Sous Linux les tabulations,les espaces et les entrées sont considérés comme des séparateurs.

Si nous avions écrit :

```
./nombre_de_caracteres.sh ma chaine de caractere
```

Linux et le script comprendraient non pas une chaine avec des espaces dans une variable mais 4 chaines de caractères dans 4 variables :

- $1 : ma
- $2 : chaine
- $3 : de
- $4 : caractere

Pour protéger une chaine de caractère il faudra utiliser les doubles ou simple côtes

Simple ou double côtes

L'interprétation des variables ne se fait pas de la même manière avec des doubles et simple côte:

Nous avons une variable d'environnement $HOME qui contien /home/isen

si j'écris par exemple :

```
echo "$HOME"
```

Le résultat sera : /home/isen

par contre si j'écris :

```
echo '$HOME'
```

Le résultat donnera : $HOME

Traitement supplémentaire des variables

```
${var-texte}  : renvoie le contenu de var si var est
        définie, sinon renvoie texte
${var:-texte} : renvoie le contenu de var si var est
        définie et non vide, sinon renvoie texte
${#var}       : renvoie la taille de la chaîne $var
${#tab[\\\*]}    : renvoie le nombre d'élément dans
        le tableau
${var:x:y}    : renvoie les "y" caractères de $var à
        partir de du caractère "x"
```

Affectation par “sous-exécution”

Enregistrer le résultat d’une commande dans une variable

Le but est d’affecter à une variable le résultat d’une exécution. Exemples :

```
$ var="$(date)"
$ echo ${var}
Sun Nov 28 19:01:36 CET 2010
```

Le résultat de la commande ‘date’ est affecté à la variable var

```
$ moi="$(whoami)"
$ nbaccount=$(wc -l /etc/passwd | cut -f 1 -d' ')
```

Bash & math

Pour définir et initialiser une variable de type entier, on utilise la syntaxe suivante : $ declare -i v=35

On utilise la commande interne “’ (( )) ’” pour effectuer des opérations arithmétiques :

```
$ (( v++ )); echo $v
    36
    $ echo $(( v * 9 ))
    324
    $ (( r = v - 1 )); echo $r
    35
```

Contrainte : les nombres à virgule flottante ne peuvent être employés.

Les tableaux

Affectation :

```
$ tab[n]=chaîne
$ tab=(chaîne1 chaîne2 chaîne3 ...)
```

Accès :

```
$ echo ${tab[2]} // affiche le 3ème élément du tableau
$ echo ${tab[*]} // affiche tous les éléments
$ echo ${tab[@]} // idem
```

Edition sou LINUX/UNIX en mode TEXTUEL

Sous linux il éxiste plusieurs traitements de texte en mode textuel. Le mode textuel veut simplement dire edition dans un terminal Ce qui implique pas de menu, pas d'image, pas de mise en forme

Aventage :

- Ultra rapide
- Par défaut sur la plupart de distribution UNIX/LINUX
- Portable et multiplateforme
- Accessible depuis le réseau
- Très puissant
- Programmable
- Modulable
- Gestion des fichiers de configuration simplifiée.

Inconveniant :

```
- Nécessite de la pratique
- Apprentissage compliqué
- esthétiquement pauvre
```

Editeur de texte type terminal

Voici les editeur texte les plus retrouvés sous Linux et les plus utilisés

- Vi/Vim (langage macro en propre)
- Emacs
- nano

Vim logo

Figure 1: Vim logo

Vim move

Figure 2: Vim move

- jupp / joe

Dans ce cours nous nous pencherons sur vim qui est un portage plus accéssible de vi

Vim est LE plus utilisé et le plus pratique pour l'édition de code et fichier de configuration.

Les Bases de vim

Les modes sous VIM

Sous VIM il existe 3 grands modes qui permetent d'utiliser ce traitement de texte.

```
1 * le mode interactif
2 * le mode insertion
3 * le mode commande
```

Mode interactif

C'est le mode par défaut par lequel vous commencez en lançant Vim. Vous êtes donc en mode interactif. Dans ce mode, vous ne pouvez pas écrire de texte.

Ce mode vous permet de : copier/coller/supprimer/séléctionner …

Pour se déplacer sous vi vous utiliserez les touches HJKL qui correspondent réspectivement à Gauche Bas Haut Droite

Pour se déplacer sous vim vous utiliserez les flêches Gauche Bas Haut Droite

---

Pour les commandes indispensable d'edition du mode interactif :

- v : Permet la sélection d'une partie du texte
- x : Efface le caractère courant (couper)
- y : Copie une ligne ou un mot
- d : Couper une ligne ou un mot
- p : coller

Les combos :

- yw/dw : copie/coupe le mot
- yy/dd : copie/coupe la ligne courante

Mode insertion

C'est le mode le plus 'classique'. Vous tapez du texte et ce dernier s'insère à l'endroit où se trouve le curseur.

Pour entrer dans ce mode, il existe plusieurs possibilités. L'une des plus courantes est d'appuyer sur la touche i (insertion).

Très important : Pour sortir de n'importe quel mode il faut appuyer sur la touche Echap.

Mode commande

En mode commande vous pouvez aussi

Ce mode est le plus complet et surtout celui qui donne toute la puissance à vim

Il permet de lancer des commandes telles que « quitter », « enregistrer », etc. Vous pouvez aussi l'utiliser pour activer des options de Vim :

```
* coloration syntaxique (:syntax on )

* affichage du numéro des lignes (:set number)

* indenter intelligemment (:set smarindent)
```

Execuction d'une commande sous VIM

Tout en étant en mode commande vous pouvez récupérer des informations du shell (la console) telles que ls, locate, cp, etc.

```
* soit en tapant certaines commandes comme :

    ":ls"

    ":locate"

* soit en lançant un sous shell :

    ":sh"
    (pour ce mode une fois la session finit
    avec exit ou CTRL+d le shell rend la main à vim)
```

Pour activer ce mode, vous devez être en mode interactif et appuyer sur la touche deux points « : ». Vous validerez la commande avec la touche Entrée et reviendrez alors au mode interactif.

action sur le fichier

En mode commande vous pouvez aussi

- remplacer du texte :

  :s/vieux/nouveau/g

- Trouver du texte :

  /chaine

  pour trouver l'occurrence 'chaine' suivante appuyez sur "n". Quand le document ne trouve plus rien en fin de document il vous indique qu'il va reprendre depuis le début.

- Enregistrer :

  :w

- Quiter :

  :q