Cours GNU/Linux

Scripting shell

Plan

1 - Le scripting

2 - Le scripting avancé

1 – le scripting

Présentation

Les expressions arithmétiques

La gestion des chaînes de caractères

Les structures de contrôles

presentation

Un script est une macro opération sur le système, il effectue une série d'opérations unitaires cohérentes, **Une suite de lignes de commande.**

Le script est un fichier texte, contenant un programme codé dans un langage interprété.

Les shell scripts, script en langage shell, sont présents dans presque tous les systèmes Unix.

Comme tout programme, il devra être versionné et documenté.

Le shebang

Le caractère # est utilisé pour commenter tous les caractères qui le suivent.

La chaîne **#!/un/chemin/vers/un/executable** en début de script définit la ligne de commande d'interprétation du script.

Si le fichier premiershell.sh commence par la ligne

```
#!/bin/bash
```

Alors il serait exécuté via la commande :

```
/bin/bash premiershell.sh
```

Le shebang serait alors ignoré pat le bash car en commentaire

Méthodologie

Presque toutes les opérations unitaires d'un script shell peuvent être testées en ligne de commande.

La méthode : On exécute pas à pas le script tout en le développant.

Nous utiliserons 2 terminaux, un dans lequel nous exécutons les commandes, le second dans lequel nous écrivons le script.

Appel du script

Le script doit être exécutable pour l'utilisateur l'appelant. La commande suivante donne le droit d'exécution pour tout les utilisateurs.

```
chmod +x script
```

Appel du script : le script n'étant pas dans un répertoire du PATH il faut spécifier son chemin pour l’exécuter

```
/chemin/vers/le/script
```

L'option du bash -x permet de débugger le traitement des commandes

1 – le scripting

Présentation

Les expressions arithmétiques

La gestion des chaînes de caractères

Les structures de contrôles

La commande expr

La commande expr retourne le résultat d'une expression arithmétique ou logique

- Syntaxe :
  - expr nb1 operateur nb2
  - expr chaine : expression-régulière
- Opérateurs : (les caractères spéciaux doivent être protégé par \)
  - opérateurs simple : + - * / %
  - opérateurs de comparaison : \> \>= \< \<= = !=
  - opérateurs logique : \| \&
- Opérations complexes :
  - expr operation operateur operation
  - Avec pour operation : \(nb1 operateur nb2 \)

La commande (( ))

Le commande (( )) est plus simple.

- Elle ne nécessite pas d'espace entre les arguments et l'utilisation du caractère d’échappement \
- Elle permet d'assigner une valeur à une variable en effectuant l'opération
- Elle permet la substitution d'une opération arithmétique par son résultat
- Syntaxe :
  - ((var=val1OPval2)) assigne le résultat de l'opération OP entre val1 et val2
  - $((val1OPval2)) : sera substitué à l'évaluation de la ligne de commande par le résultat de l'opération
- Opérateurs :
  - opérateurs simple : + - * / %
  - opérateurs de comparaison : > >= < <= == !=
  - opérateurs logique : ! || &&
  - regroupement d'opérations : (expression)

1 – le scripting

Présentation

Les expressions arithmétiques

La gestion des chaînes de caractères

Les structures de contrôles

Commande echo 1/2

Nous avons vue la commande echo qui affiche le reste de la ligne de commande

cette commande affiche une ligne complète jusqu'au retour à la ligne.

Elle dispose d'options permettant de formater l'affichage

- -n : supprime le retour à la ligne
- -e : permet l'utilisation des séquences d’échappement
- -E : supprime l'utilisation des séquences d’échappement

Commande echo 2/2

Séquences d'échappement : Attention le \ est interprété par le shell avant d'être transmis à la commande

\e caractère Échap.

\f saut de page

\n saut de ligne

\r retour chariot

\t tabulation horizontale

\v tabulation verticale

\\ barre contre-oblique

\0nnn le caractère dont le code ASCII est NNN (en octal)

Traitement des chaînes par le shell

Le shell lors de l'évaluation de la ligne de commande procède aux substitutions ($xxxxx) et au traitement des protections \ ' ' “ ”

Le shell n'évalue pas le contenu des chaînes de caractères entre simples quottes.

Le shell ne traite pas les protections entre double quottes. En revanche il traite les substitutions.

Les quottes les plus externes prévalent sur les autres

```
alan@laptop:~& echo “'$var'”
'val'
alan@laptop:~&
```

La concaténation est implicite :

```
alan@laptop:~& echo $var$var
valval
```

Commande read 1/2

La commande read permet d'attendre une donnée sur l'entrée standard

```
alan@laptop:~/test$ read a
aaa
alan@laptop:~/test$ echo $a
```

elle peut être utilisée avec une série de valeurs

```
alan@laptop:~/test$ read a b c
aaa bbb ccc
alan@laptop:~/test$ echo “$a - $b - $c”
```

La variable interne du shell IFS définit le séparateur de champs

Commande read 2/2

```
alan@laptop:~/test$ oldIFS=$IFS
alan@laptop:~/test$ IFS=":"
alan@laptop:~/test$ read a b c
aaa bbb:ccc
alan@laptop:~/test$ echo $a $b $c
aaa bbb ccc
alan@laptop:~/test$ echo $a
aaa bbb
alan@laptop:~/test$
```

1 – le scripting

Présentation

Les expressions arithmétiques

La gestion des chaînes de caractères

Les structures de contrôles

Les expressions conditionnelles

Les tests logiques

Sous Unix le résultat d'un test est le code retour d'une commande.

Si la commande est exécutée avec succès, le code retour est 0 et le test retourne vrai, si la commande est exécutée avec une erreur, le code retour est différent de 0 et le test retourne faux.

De ce fait toute commande est un test.

La commande test

La commande interne test effectue une évaluation logique d'une expression (Voir : help test)

Elle est équivalente à la syntaxe [[expression]]

Elle intègre un grand nombre de tests sur les fichiers, sur les chaînes de caractères et sur des numériques.

Syntaxe :

```
test OP objet
test objet1 OPB Objet2
[[ OP objet ]]
[[ objet1 OPB objet2 ]]
```

Test sur les fichiers

Opérateurs sur les fichiers : voici les principaux

```
-d FILE        True if file is a directory.
-e FILE        True if file exists.
-f FILE        True if file exists and is a regular file.
-h FILE        True if file is a symbolic link.
-L FILE        True if file is a symbolic link.
-p FILE        True if file is a named pipe.
-r FILE        True if file is readable by you.
-s FILE        True if file exists and is not empty.
-w FILE        True if the file is writable by you.
-x FILE        True if the file is executable by you.
-O FILE        True if the file is effectively owned by you.
-G FILE        True if the file is effectively owned by your group.
-N FILE        True if the file has been modified since it was last read.
FILE1 -nt FILE2  True if file1 is newer than file2 (according to
modification date).
FILE1 -ot FILE2  True if file1 is older than file2.
FILE1 -ef FILE2  True if file1 is a hard link to file2.
```

Test sur les chaînes

Opérateurs de la commande test sur les chaînes de caractères :

```
-z STRING      True if string is empty.
-n STRING      True if string is not empty.
STRING1 = STRING2    True if the strings are equal.
STRING1 != STRING2   True if the strings are not equal.
STRING1 < STRING2    True if STRING1 sorts before STRING2
lexicographically.
STRING1 > STRING2    True if STRING1 sorts after STRING2
lexicographically.
```

Test numériques

Opérateurs de la commande test sur des numériques :

```
num1 -eq num2 : EQual
num1 -ne num2 : Not Equal
num1 -lt num2 : Less Than
num1 -le num2 : Less or Equal
num1 -gt num2 : Greater Than
num1 -ge num2 : Greater or Equal
```

Autres tests

Opérateur sur les variables :

```
-v VAR        True if VAR exist.
```

Les opérateurs logiques sur les expressions de test :

```
expr -a expr : le “et” logique (and)
expr -o expr : le “ou” logique (or)
!expr : la négation logique
```

if then elif else fi

C'est l'expression conditionnelle la plus explicite.

syntaxe :

```
if cmd # la commande est un test
  then commands # les commandes exécutées si
le test est valide
elif cmd
  then commands # les commandes si le premier
test est invalide mais le second est valide
else commands # les commandes si aucun test
n'est valide
fi
```

Chaînage conditionnel de commande

Les tests étant des commandes, les chaînage de commande conditionnel && et || permettent de traiter les expressions conditionnelles simples.

`test && cmd` : la commande cmd est traitée si test renvoie vrai

Si test est faux, la proposition test && test2 est forcément fausse, le test2 ne sera donc pas évalué, donc la commande non traitée

`test || cmd` : la commande cmd est traitée si test renvoie faux

Si test est vrai, la proposition test ou test2 est forcément vrai, le test2 ne sera donc pas évalué et donc la commande non traitée

Pour les actions conditionnelles simples, nous préfèrerons cette écriture.

Case in esac

Case va exécuter une série de commande (commands) sur une correspondance de motifs. Les commandes correspondantes au **premier** motif reconnu seront exécutées, les autres seront ignorées.

Syntaxe :

```
case $var in
    motif1|motif2)
        commands
        ;;
    motif3)
        commands
        ;;
    *)
        commands
        ;;
esac
```

Les boucles

for in do done

Permet de répéter un traitement sur une série de valeurs

Syntaxe :

for var in liste de valeur

do commands # on utilisera $var pour instancier la valeur courante

done

Exemple :

alan@laptop:~$ for var in liste de valeurs

> do echo $var

> done

liste

de

valeurs

alan@laptop:~$

While do done

Répète un traitement tant qu'un test retourne vrai.

Syntaxe :

```
while cmd # cmd est le test
do commands
done
```

Exemple :

```
alan@laptop:~$ while [[ "XX$var" != "XXquite" ]]; do echo "hein?"; read var; done
hein?
rien
hein?
quite
alan@laptop:~$
```

until

Répète un traitement tant qu'un test retourne faux.

Syntaxe :

```
until cmd # cmd est le test
do commands
done
```

Exemple :

```
alan@laptop:~$ until [[ "XX$var" == "XXquite" ]]; do echo "hein?"; read var; done
hein?
rien j'ai dit
hein?
quite
alan@laptop:~$
```

break continue 1/2

break et continue sont des opérateurs de boucles, ils s'utilisent sur les boucles for while until et select (voir plus loin).

break casse la boucle et force la sortie

continue termine l'exécution des commandes de la boucle et reviens en début de boucle

en cas de boucles imbriquées il est possible de spécifier un niveau de boucle à interrompre :

- break 2 : interrompt 2 niveau de boucle
- continue 2 : reviens au début de la boucle parente

Break continue 2/2

Exemple :

```
alan@laptop:~$ for u in 1 2 3
do for v in 1 2 3
   do echo $u $v
      [[ $v -eq 2 && $u -eq 1 ]] && echo continue  && continue
      [[ $v -eq 2 && $u -eq 2 ]] && echo continue 2 && continue 2
      [[ $u -eq 3 ]] && echo break && break
      echo \-
   done
done
1 1
-
1 2
continue
1 3
-
2 1
-
2 2
continue 2
3 1
break
alan@laptop:~$
```

2 - scripting avancé

Gestion de variables avancée

Les substitutions

Les fonctions

Les interactions avec l'utilisateurs

Les entrées sorties

Commande declare

La commande interne declare permet de gérer les variables et les déclarer de façon explicite

declare -p : retourne un script déclarant les variables présentes dans l'environnement

declare -i var=val : définit la variable var en tant qu'un entier

declare -a var : déclare la variable var en tant que liste de valeurs indexées

declare -A var : déclare la variable var en tant que tableau associatif

Les listes

Une liste est une variable contenant plusieurs valeurs indexées.

Affectation : **variable[index]=valeur** ou **variable=(valeurX valeurY ...)**

```
VAR[0]=val0
VAR[1]=val1
VAR=(val0 val1 val2 val3 val4)
```

A l'utilisation les accolades sont nécessaires : **${variable[index]}**

```
echo ${VAR[1]}
```

Toute variable est en fait une liste avec une seule entrée: $VAR est équivalent à ${VAR[0]}

```
echo ${VAR}
```

Les tableaux associatifs

Disponible uniquement sur certains shell, c'est une liste dont l'index est une chaîne de caractères.

Affectation : **variable[index]=valeur** ou **variable=([index1]=valeur1 [index2]valeur2 ...)**

```
var[root]=toor
var=([hello]=world [foo]=bar)
```

A l'utilisation les accolades sont nécessaires : **${variable[index]}**

```
echo ${var[hello]}
echo ${var[foo]}
```

Gestion des listes et tableaux

Il existe une syntaxe pour traiter les listes :

Tous les éléments de la liste : **${variable[@|*]}**

```
echo ${VAR[@]}
echo ${VAR[*]}
```

Toutes les clés d'un tableau : ${!variable[@]}

```
echo ${!VAR[@]}
```

Le nombre d'éléments définis : **${#variable[@|*]}**

```
echo ${#VAR[@]}
echo ${#VAR[*]}
```

La taille d'un élément : **${#variable[index]}**

```
echo ${#VAR[2]}
```

La variable $IFS 1/2

$IFS pour Internal Fields Separator

- Ce sont les séparateurs de champs du shell
- Cette variable a pour valeur : <espace><tab><return>

En changeant cette variable on peut changer le comportement de certaines commandes internes afin qu'elles traitent des lignes complètes.

Modification :

```
oldIFS=$IFS   # on sauvegarde la valeur actuelle
IFS="[Ctrl]+v[enter]" # on redéfinit la variable, les touches [Ctrl]+v permettent de protéger la touche suivante [enter] de son interprétation par le bash.
```

La variable $IFS 2/2

Test sur la boucle for :

```
alan@laptop:~$ echo *
Bureau Documents Images work
alan@laptop:~$ for u in `echo *`
> do
> echo $u
> done
Bureau Documents Images work
alan@laptop:~$ IFS=$oldIFS # on restore la variable IFS
alan@laptop:~$ for u in `echo *`; do echo $u; done
Bureau
Documents
Images
work
alan@laptop:~$
```

2 - scripting avancé

Gestion de variables avancée

Les substitutions

Les fonctions

Les interactions avec l'utilisateurs

Les entrées sorties

Substitutions de variables 1/2

Il est possible de substituer une valeur aux variables **non définies**

on substitue l'expression par une valeur : **$(variable:-valeur)**

```
echo ${home:-/home/alan}
echo $home
home=toto
echo ${home:-/home/alan}
```

on substitue l'expression et on définit la variable : **$(variable:=valeur)**

```
echo ${home:=/home/alan}
echo $home
```

Substitutions de variables 2/2

Il est possible de substituer une valeur aux variables **définies**

on substitue l'expression par une valeur : **$(variable:+valeur)**

```
echo $home
home=toto
echo ${home:+/home/alan}
```

il est possible d'interrompre l'exécution si la variable n'est pas définie en affichant un message d'erreur

syntaxe : **$(variable:?erreur)**

```
unset $home
echo ${home:?"variable non définie"}
```

Substitution de chaînes

Les substitutions permettent d'extraire une partie de chaîne de caractères d'une variable, la variable n'est pas modifiée.

Syntaxe : ${var[#|##|%|%%]modele}

- var est une variable
- # et ## signifie par la gauche
- % et %% signifie par la droite
  - 1 caractère pour la plus petite chaîne 2 caractères pour la plus grande.
- modele est le modèle de type wildcard de ce qui est supprimé de la chaîne.

Exemple :

```
ligne="val1:val2:val3"
echo ${ligne#*:}   #retourne val2:val3
echo ${ligne##*:}  #retourne val3
echo ${ligne%:*}   #retourne val1:val2
echo ${ligne%%:*}  #retourne val1
```

2 - scripting avancé

Gestion de variables avancée

Les substitutions

Les fonctions

Les interactions avec l'utilisateurs

Les entrées sorties

Présentation

Une fonction est un ensemble cohérent de commandes pouvant être utilisé plusieurs fois

Une fonction doit être définie avant d'être appelée. Elle fait partie de l'environnement.

Tout comme les scripts, les fonctions sont des commandes, elles ont donc un code retour. Celui-ci est transmis par la commande interne “return”

Création de fonctions

Syntaxes :

```
function foo {
  echo “bar ?”
  return $?
}
bar() {
  echo “foo ?”
  return $?
}
```

Arguments de fonctions

Tout comme les scripts les fonctions acceptent des arguments, il sont adressés de la même façon que pour les scripts : $1 $2 $3 ….

$0 n'est pas modifié, cela reste le nom du script

$# contient le nombre d'arguments

$* contient tous les arguments

Gestion de variables

Afin d'éviter les écrasements de variables, les variables doivent être définie localement aux fonctions.

```
function foo {
local var=$1
echo $var
}
```

Fonctions usuelles d'un script

usage : retourner la syntaxe de la commande et sortir

end : fonction de sortie du script

log : gestion de l'écriture des logs

Création d'une bibliothèque

Un simple fichier texte contenant des déclaration de fonction

Chargement de la bibliothèque dans le script

. lib.sh

1 - scripting avancé

Gestion de variables avancée

Les substitutions

Les fonctions

Les interactions avec l'utilisateurs

Les entrées sorties

Gestion de la ligne de commande

Nous avons vu les paramètres positionnels :

$0 $1 $2 $* $@ $#

Nous pouvons alors traiter les arguments du script via ces variables.

La commande interne shift permet de déplacer de droite à gauche les paramètres de la ligne de commande. Ainsi après avoir passé la commande shift le paramètre 1 est perdu puis les suivants sont déplacés vers la gauche $1 contient alors $2, $2 contient $3 etc...

La commande set permet de les redéfinir

set val vale valeu valeur

La commande getopt offre une solution complète de traitement de la ligne de commande

Commande getopt

La commande getopt lit le premier argument et shift les suivants.

Syntaxe : getopts: getopts chaine-options var [arg]

chaine-options contient les lettres d'options qui devront être reconnues précédées par “:” (erreurs silencieuses) ; si une lettre est suivie par un deux-points, elle devra posséder un argument.

Exemple : :a:bc pour les options -a argument -b -c

var est une variable alimentée par getopt de la façon suivante

l'option en cours de traitement si connue par la chaine-options

« ? » si l'option est inconnue par la chaine-options (la variable $OPTARG contiendra alors l'option en cours de traitement)

« : » si l'option en cours de traitement nécessite un argument non présent (la variable $OPTARG contiendra alors l'option en cours de traitement)

Exemple de traitement d'options

```
#!/bin/bash
while getopts ":a:bc" opt; do
  case $opt in
    a)
      echo "-a checked, Parameter: $OPTARG" >&2
      optiona=1
      file=$OPTARG
      ;;
    b)
      echo "-b checked" >&2
      optionb=1
      ;;
    c)
      echo "-c checked" >&2
      optionc=1
      ;;
    \?)
      echo "Unknown option: -$OPTARG" >&2
      ;;
    :)
      echo "Option -$OPTARG requires an argument."
>&2
      exit 1
      ;;
  esac
done
```

```
# getopt retourne 0 tant qu'il y as des option le
test while est vrai tant qu'il y a des options.
# chaque variable optionX exist le test revois vrai

[[ $optionc ]] || echo option c required >&2
[[ $optionc ]] || exit 1
[[ $optiona ]] && echo action for option a on $file
[[ $optionb ]] && echo action for option b
[[ $optionc ]] && echo action for option c
[[ $optionf ]] && echo action for option f
```

Lecture de l'entrée standard

En utilisant la commande read dans une boucle while il est possible de lire en continu l'entrée standard.

```
#!/bin/bash
oldIFS=$IFS
IFS=:
while read a b c d e f
do
    echo traitement de la ligne $a  $b  $c  $d  $f
done
alan@laptop:~/test$ cat /etc/passwd | ./lectent
traitement de la ligne root x 0 0 /root /bin/bash
…
```

select in do done

Outil d'interaction avec l'utilisateur, il permet de proposer des choix à l'utilisateur et exécute une série de commandes pour chaque choix effectué

Syntaxe :

select $var in liste de valeurs

do commands

done

Exemple :

```
alan@laptop:~$ select var in liste de valeurs et quite; do echo $var; [[ $var ==
"quite" ]] && break ; done
1) liste
2) de
3) valeurs
4) et
5) quite
#? 2
de
#? 5
quite
alan@laptop:~$
```

1 - scripting avancé

Gestion de variables avancée

Les substitutions

Les fonctions

Les interactions avec l'utilisateurs

Les entrées sorties

Les descripteurs de fichiers 1/2

Un descripteur de fichier est identifié par un chiffre, en ligne de commande les entrées sorties standards sont en fait les descripteurs de fichier 0 et 1 qui renvoient vers les entrées sorties du terminal.

Il est possible de créer d'autres descripteurs de fichiers avec la commande exec.

```
exec 3<./input 4>./output
```

La commande read peut alors lire sur le descripteur de fichier 3

```
read -u3 ligne
```

et il est possible d'écrire sur le fichier de sortie via le descripteur de fichier 4

```
echo test >&4
```

Il faudra fermer correctement les descripteurs de fichiers :

```
exec 3<&-
exec 4>&-
```

Les descripteurs de fichiers 2/2

Il est possible de redéfinir les descripteurs de fichiers toujours avec la commande exec :

```
exec 0<$file 1>$output.txt
```

il est alors possible de faire un script traitant l'entrée standard qui suivant ses paramètres de ligne de commande basculera sur le traitement d'un fichier.

```
[[ -v $file ]] && exec 0<$file
while read ligne
do
...
```

Génération d'une entrée standard

Certaines commandes fonctionnent en mode interactif, il peut être utile alors de générer, dans un script qui les utilise, leur entrée standard.

Syntaxe : EOF est ici une chaîne choisie pour délimiter l'entrée générée.

```
cmd <<!EOF
texte …
… texte
!EOF
```

3 - Conclusions : Les bonnes pratiques

Objectifs

Lisibilité : c'est trivial il paraît, mais après ce que j'ai vu cela me semble important de le préciser.

La robustesse : on n'aime pas chercher et corriger des bugs, surtout les chercher (rapport au point 1).

La performance : on voit de ces trucs ! J'vous jure.

Le style : parce qu'il faut avoir la classe en toute circonstance

Documentez bien sur!

Faites un entête de script, même minime.

Exemple de cartouche de script :

```
#!/bin/bash
#############################################################################
# Description   : Brève description de l'objectif et des paramètres d'entree   #
#                 et de sortie
#
# Usage         : <syntax script>
#
# Suivi version :
#  V   | Date     | Auteur            | Description des modifications
# -----|----------|-------------------|---------------------------------------
#  0.9 | 20130312 | asi               | Creation
#  1.0 | 20130315 | asi               | Correction après tests
#############################################################################
```

Indentez le code (2 ou 4 espaces) (sous vi : :autoind)

Commentez les commandes complexes, et expliquez les astuces,

Utiliser des noms de variable explicites

Commentez aussi les fonctions

Soyez pertinents

Utilisez un shebang explicite : /bin/bash si le script est testé en bash, re-testez tout le script si vous changez de shell.

Renvoyez des codes retour non nul en cas d'erreurs.

Validez vos variables et testez-les avant de les utiliser.

Evitez les risques (rm -rf $var = DANGER!)

Traitement de flux

Lorsqu'un script doit traiter un fichier, traitez si possible le cas ou le fichier est lu sur l'entrée standard

si le fichier est produit par une autre commande, on évite alors l'écriture sur disque, puis la lecture sur disque en chaînant les commande au travers d'un pipe.

Les performances 1/2

Privilégiez les commandes internes au shell, vous utiliserez alors moins de ressources. Toute commande externe est un nouveau processus avec allocation de mémoire et prise de temps cpu.

Exemple :

```
ligne=val1:val2:val3
echo val1:val2:val3 | cut -d: -f1
echo ${ligne%%:*}
```

Les performances 2/2

Evitez les écritures sur disque.

Utilisez le flux de données et les pipes tant que possible.

L'idempotence

Un script idempotent est un script qui donne le même résultat quelque soit le nombre de lancements.

- On évite donc :
  - Les surcharges (l'action est déjà faite pas besoin de la refaire)
  - Les erreurs liées à un rappel de commande accidentel

Tester l'idempotence permet de s'assurer que tous les tests devant être effectués l'ont bien été.

Un script idempotent est robuste

Conservez et revendiquez votre style de code