Architecture et système

Script shell

C. Nguyen, J. Razik

Département d'informatique
Université de Toulon

Le shell s'utilise en mode interactif, mais on peut également l'utiliser en mode script.

Dans ce mode la suite de lignes de commande est écrite dans un fichier texte (un simple éditeur de texte suffit) qu'on appelle un script. Le nom de ce fichier devient alors une nouvelle commande.

Exemple : hello.sh

```
#!/bin/bash
echo "Hello World"
```

Le « shebang » (*sharp bang*), placé au tout début du fichier, indique au système l'interpréte à utiliser pour traiter le fchier.

On peut faire d'un script un programme exécutable en modifiant ses droits par `chmod` et le rendre accessible en intégrant son chemin dans la variable d'environnement `PATH` (⚠ •).

Les variables dans un script

La syntaxe de manipulation des variables est la même que celle du mode interactif, pour rappel :

- elles n'ont pas besoin d'être déclarées et ne sont pas typées,
- on doit préfixer leur nom par le caractère $ pour accéder à leur contenu

Différence avec le mode interactif : les variables sont locales au script, elles ne vont exister que pendant son exécution.

Exemple : calcul.sh

```
$ cat calcul.sh
x=40 ; ((x+=2))
$ bash calcul.sh
$ echo $x
```

Les variables dans un script

Les commandes d'un script sont traitées dans un contexte d'exécution différent de celui du shell du terminal sur lequel il a été lancé.

La commande `source` (abréviation : •) : change ce comportement de localité des variables manipulées au sein d'un script en lançant son exécution dans le même contexte d'exécution que le shell du terminal.

Quand il n'est pas « sourcé », le lancement d'un script crée un *nouveau* processus (un sous-shell) qui va se charger de réaliser l'exécution du code du script.

Utiliser des variables externes

Pour les mêmes raison que précédemment [1], les variables du shell ne sont pas connues par défaut par un script shell.

Si l'on veut utiliser dans un script des variables qui sont fixées avant de lancer son exécution, on peut « sourcer » le script, mais il est préférable d'utiliser la commande `export`.

La commande `export` indique que la variable en argument sera *copiée* dans tous les contextes d'exécution des processus qui seront créés par la suite.

Exemple : affiche.sh

```
$ cat affiche.sh
echo $x
$ x=1
$ ./affiche.sh
```

1. le shell et un script sont exécutés dans deux processus différents.

Exemple d'utilisation de variables

```
$ cat visibilite.sh
echo $x ; ((x+=1)) ; echo $x
$ x=1
$ ./visibilite.sh
💬   ?
$ export x=1
$ ./visibilite.sh
💬   ?
$ unset x
$ x=1
$ source visibilite.sh
💬   ?
```

Les arguments d'un script

Comme pour n'importe quelle commande, des arguments peuvent être passés à un script par la ligne de commande.

Dans le code du script on peut exploiter ces arguments à l'aide de variables particulières qui sont automatiquement renseignées au début de l'exécution du script.

Ces variables sont les suivantes :

- $# : le nombre d'arguments passés,
- $@ : la liste de tous les arguments,
- $n : la valeur de l'argument à la position *n* lors de l'appel du script.

⚠ substitution de variables (exemple : t='a b c' ; combien.sh $t).

Manipulation des arguments

Pour exploiter au mieux les arguments d'un script, on peut mettre en œuvre deux mécanismes :

- la substitution de variable par une valeur par défaut : **:-**, syntaxe $variable :-valeur,
- le contrôle de la présence d'un argument obligatoire : **:?**, syntaxe $variable :?message.

```
ARG1=${1:-aucun}
ARG1=${1:?"fournissez un argument"}
```

Code retour d'un script

Une commande renvoie un code retour (0 si la commande s'est terminée normalement, une valeur entre 1 et 255 sinon) pour indiquer au processus parent si l'exécution s'est bien passée.

Pour un script, le code retour est soit celui de la dernière commande exécutée, soit celui associé à la commande `exit` (0 par défaut).

Le code retour se récupère par l'appelant du script en utilisant `$?`.

```
$ rm
rm : opérande manquant
$ echo $?
1
```

Plan

La commande test

Toute commande Unix a un code retour, ce qui permet de la voir comme une expression conditionnelle dont le code retour est assimilé à vrai s'il vaut 0 et à faux dans les autres cas.

Pour exprimer la condition $x > 0$ par exemple, le bash a deux syntaxes :

- `test $x -gt 0`
- `[␣$x -gt 0␣]`

La commande `test` dispose des principaux opérateurs de comparaison arithmétique sur les entiers : `-eq`, `-ne`, `-gt`, `-lt`, `-ge`, `-le`.

La commande test

`test` permet aussi de comparer deux chaînes de caractères grâce aux opérateurs : `-n`, `-z`, `=`, `==`, `!=`, `\<`, `\>`.

Remarque : < et > étant des caractères spéciaux du shell, ils doivent être « dépersonnalisés ».

La comparaison des chaînes de caractères se base sur le code ASCII des caractères. Ainsi, les chiffres sont « inférieures » aux majuscules qui sont elles-mêmes « inférieures » aux minuscules.

Exemple : `[ alice \> Alice ] ; echo $?`

```
0
```

⚠ ne pas utiliser < et > sur des nombres.
exemple : `[ 32 \> 200 ]`.

La commande test

`test` permet également de réaliser des tests sur les fichiers. Les opérateurs les plus utilisés sont :

- `-e` vrai si le fichier existe (pour les autres opérateurs, le fichier doit exister),
- `-s` vrai si le fichier n'est pas vide,
- `-f` vrai si le fichier est ordinaire,
- `-d` vrai si le fichier est un répertoire,
- `-r` vrai si le fichier est accessible en lecture,
- `-w` vrai si le fichier est accessible en écriture,
- `-x` vrai si le fichier possède le droit exécutable,

Opérateurs logiques

On peut combiner les tests à l'aide d'opérateurs logiques :

- ! est l'opérateur logique unaire NON,
- -a est l'opérateur logique binaire ET,
- -o est l'opérateur logique binaire OU.

Ces opérateurs respectent les règles de priorité habituelles de la logique.

Exemple : `[ -e confidentiel -a ! -r confidentiel ]`

La commande de test étendu

Il existe une version étendue de la commande `test` qui utilise des crochets doubles : [[...]].

Cette commande apporte quelques facilités :

- utilisation directe des opérateurs < et > [2],
- utilisation des opérateurs logique && (ET) et || (OU),
- les expressions arithmétiques sont évaluées,
- la chaîne de caractères à droite des opérateurs == ou != est traitée comme un motif (*pattern matching*) grâce aux caractères spéciaux ?, * et [ ],
  exemple : `[[ taratata == t*a ]]`.

2. mais **pas** les opérateurs <= et >=

Plan

Enchaînement conditionnel

Il existe trois caractères spéciaux permettant l'enchaînement de commandes : `;`, `||` et `&&`.

Le point-virgule permet d'enchaîner une suite d'instructions sur une même ligne de commandes, l'échec d'une instruction n'interrompt pas les autres.

Pour les deux autres, on conditionne l'exécution d'une commande au résultat d'une autre.

Avec l'enchaînement `cmd1 && cmd2`, `cmd2` n'est exécutée que si le code retour de `cmd1` est 0.

À l'inverse, pour l'enchaînement `cmd1 || cmd2`, `cmd2` ne sera exécutée que si `cmd1` échoue.

Structure de contrôle conditionnel `if`

La commande `if` permet de conditionner l'exécution d'une suite de commandes au résultat d'une condition.

Condition simple : la structure conditionnelle permet d'effectuer une opération si une condition est réalisée, syntaxe :

```
if condition
then
        commandes
fi
```

⚠ respecter les passages à la ligne.

Remarque : la condition peut être la commande `test` ou n'importe quelle autre commande puisque la commande `if` se base sur le code retour.

Structure de contrôle conditionnel `if`

Condition avec alternative : la structure conditionnelle peut inclure une partie `else` qui permet d'exécuter une suite de commandes alternatives au cas où la condition n'est pas vérifiée, syntaxe :

```
if condition
then
        commandes
else
        commandes
fi
```

Structure de contrôle conditionnel `if`

Conditions imbriquées : il est possible d'imbriquer des structures conditionnelles à l'intérieur d'autres structures conditionnelles, syntaxe :

```
if condition
then
        commandes
elif condition
then
        commandes
else
        commandes
fi
```

Structure de branchement conditionnel case

L'instruction case est une bonne alternative à l'utilisation de `if` imbriqués, syntaxe :

```
case expression in
motif1 )
        commandes ; ;
motif2 )
        commandes ; ;
...
esac
```

Au premier motif qui correspond, les commandes sont exécutées jusqu'au ; ; puis la commande case s'arrête en retournant le code retour de la dernière commande exécutée.

Structure de branchement conditionnel case

Les motifs avec laquelle la valeur est comparée sont des chaînes de caractères pouvant contenir les caractères spéciaux *, ? et [ ].

Si aucun des motifs ne correspond à la valeur de l'expression alors aucune commande n'est exécutée et le code retour de l'instruction case est 0.

S'il y a besoin d'exécuter des commandes dans le cas où aucun des motifs ne correspond on peut utiliser le motif * comme cas par défaut.

Si plusieurs cas doivent mener à l'exécution d'une même liste de commandes on peut écrire plusieurs motifs séparés par une barre verticale |.

Structure de branchement conditionnel case

Exemple : type.sh

```
#!/bin/bash
if [ $# -ne 1 ]
then
    echo "Erreur : nombre d'argument incorrect" >&2
    echo "Usage : $0 fichier" >&2
else
    case $1 in
    *.py)
        echo "code source en Python" ;;
    *.sh)
        echo "script bash" ;;
    *.jpeg | *.jpg | *.png)
        echo "image" ;;
    *)
        echo "type de fichier non reconnu" ;;
    esac
fi
```

Plan

Introduction

Les deux types de boucles les plus courantes sont :

- l'itération conditionnelle, qui repose sur une expression,
- l'itération bornée (par le nombre d'éléments d'une liste par exemple).

Itération conditionnelle

La commande `while` répéte l'exécution d'une suite de commandes tant qu'un test est évalué à vrai, syntaxe :

```
while condition
do
        commandes
done
```

Exemple : repeter.sh

```
#!/bin/bash
MOT=$1 :?"paramètre mot manquant"
NB=$2 :-10
i=0
while [ $i -lt $NB ]; do
      echo $MOT
      i=$(( $i + 1 ))
done
```

Lecture de fichier avec `while` et `read`

Une utilisation classique de la boucle `while` est la lecture de fichier ligne à ligne.

Exemple : lecture.sh

```
#!/bin/bash
while read li ; do
        echo -n "ligne : "
        echo $li
done < $1
```

La condition d'arrêt de cette boucle est définie par la commande `read` quand la commande arrive à la fin du fichier à lire.

Contrôle de la boucle while

La commande `continue` saute directement à l'itération suivante dans la boucle en n'effectuant pas les autres commandes restantes.

La commande `break` termine la boucle en sortant directement de celle-ci.

Exemple : compteur.sh

```
#!/bin/bash
a=0
while [ "$a" -le 10 ]; do
    a=$((a+1))
    if [ "$a" -eq 2 ] || [ "$a" -eq 8 ]
    then
        continue   # ou break
    fi
    echo -n "$a "
done
```

Itération bornée

La commande `for` répéte l'exécution d'une suite de commandes pour chacun des éléments d'une liste, syntaxe :

`for` *condition* `in` *liste* ; `do`
    commandes
`done`

Exemple : repeter.sh

```
#!/bin/bash
n=1
for arg in $@ ; do
    echo "argument $n = $arg"
    n=$(( $n + 1 ))
done
```

Utilisation d'un compteur

La commande seq prend 2 nombres en argument, et produit les valeurs successives entre ces 2 nombres.

Exemple

```
$ seq 1 5
1 2 3 4 5
```

On peut ainsi revenir au fonctionnement classique d'une boucle à base de compteur.

Exemple

```
...
for i in $(seq 1 $n) ; do
...
```

Autre structure de la boucle for

La boucle for peut aussi prendre une syntaxe héritée du langage de programmation C.

```
for (( EXP1 ; EXP2 ; EXP3 )) ; do
        commandes
done
```

Cette variante est caractérisée par trois paramètres qui contrôlent la boucle :

- EXP1 : initialisation de la boucle,
- EXP2 : test qui conditionne l'exécution,
- EXP3 : expression de comptage.

Exemple

```
...
for i ((i=1 ; i<=$n ; i++)) ; do
...
```

Plan

Définition

La notion de fonction sert avant tout à factoriser un ensemble de commandes.

Une fonction est définie par un nom suivi d'un couple de parenthèses et une paire d'accolades (corps de la fonction), syntaxe :

```
nom () {
        commandes
}
```

Une fonction doit être déclarée avant d'en faire l'appel et elle n'existe que pour le fichier script dans lequel elle a été définie.

Pour utiliser la fonction il suffit simplement de taper son nom, comme si c'était une commande shell classique.

Exemple de fonction

```
#!/bin/bash

# premier exemple de fonction
say_hello() {
    echo "Hello World"
}

# appel de la fonction
say_hello
```

⚠ l'exécution de la fonction s'effectue dans le même contexte que l'appelant, à la différence d'un script qui s'exécute par un sous-shell et qui possède son propre contexte.

Paramètres d'une fonction

La syntaxe de passage des arguments à une fonction est similaire à celle d'une commande classique, syntaxe :

$$nom\ arg_1\ arg_2\ \ldots arg_n$$

L'utilisation des paramètres dans le corps d'une fonction se fait de la même manière que pour un script (voir $1, $2, ..., $#, $@, ...).

```
say_hello() {
    echo "Hello $1"
}

say_hello Alice
```

Variables locales à une fonction

Quand on manipule une variable dans un script, les changements subis par cette variable valent pour l'ensemble du script.

Il en va de même au sein d'une fonction : la portée d'une variable est globale par défaut.

Il est cependant possible de limiter la portée d'une variable dont l'usage ne concerne que l'espace du corps d'une fonction grâce à la commande `local`, à placer au début du code de la fonction, syntaxe :

`local` *nom=valeur*

Retours des fonctions

Le code de retour d'une fonction est une valeur numérique (0 à 255) qui sera :

- le code retourné par la dernière commande exécutée dans le corps de la fonction,
- ou celui de la commande `return` qui prend en argument la valeur du code à renvoyer (0 par défaut) et qui sort.

```
max() {
    if [ $1 -gt $2 ] ; then
        return $1
    else
        return $2
    fi
}
```

Problème : max 580 127 retourne 68.

Retours des fonctions

Pour pouvoir retourner un entier quelconque ou une chaîne de caractères, il existe deux possibilités :

- utiliser une variable globale, qui sera mise à jour par la fonction,
- envoyer le résultat sur la sortie standard par la commande `echo` puis récupérer cette valeur en utilisant la syntaxe de substitution de commande : `$(...)`.

```
max() {
    if [ $1 -gt $2 ] ; then
        echo $1
    else
        echo $2
    fi
}
```

res=$(max 580 127) permet d'obtenir le bon résultat.

Scripts, fonctions et alias

Les alias sont des moyens de raccourcir la saisie de commande.

La commande `alias`, sans paramètre permet de lister tous les alias actuellement définit dans le shell.

La définition d'un alias suit la syntaxe suivante :

`alias` *nom*=*'commande'*

Par exemple, pour demander une confirmation à chaque suppression de fichier, on peut utiliser l'alias suivant : `alias rm='rm -i'`.

Pour inhiber un alias, il faut utiliser la commande avec le préfixe \, par exemple : `\rm`.

La commande `unalias` supprime un alias existant.

Scripts, fonctions et alias

Les fonctions définies dans les fichiers `.bashrc` ou `.profile` sont des commandes situées entre la simplicité d'un alias et la complexité d'un script.

Une fonction, ainsi définie, ne peut pas s'utiliser dans un script alors qu'un script pourrait très bien être utilisé dans un autre script.

Bonne pratique : commencer par créer un alias, si le traitement est trop complexe pour un alias, faire une fonction et si une fonction dépasse une quinzaine de lignes, faire un script.