Système

Introduction aux scripts

Ivan Canet · 17 déc. 2017 (14 janv. 2018)

TABLE DES MATIÈRES

I. VARIABLES

Toutes les variables sont de type chaîne de caractères.

1. Utilisation

1.1. Déclaration

Globale

```
NOM=VALEUR
```

Locale

Les variables locales sont utilisées dans le contexte des fonctions *(cf. Fonctions, page 3)*.

```
local NOM=VALEUR
```

1.2. Lecture

L'expansion des variables se fait avec le caractère `$` :

```
… $NOM
```

Attention, si le nom de variable comporte plusieurs mots *(eg. cf Tableaux, page 2)*, il faut utiliser la syntaxe suivante ;

```
… ${NOM}
```

2. Opérations décimales

2.1. Syntaxe

Pour créer une nouvelle variable en lui assignant le résultat d'un calcul, on utilise le mot-clef `let`.

```
let NOM=VALEUR
```

Dans ce contexte, il n'est plus nécessaire d'utiliser l'opérateur d'expansion `$`, par exemple :

```
let six=2*3
```

On peut aussi faire un calcul directement dans une ligne, sans passer par une variable supplémentaire :

```
… $((CALCUL))
```

De même, on peut exécuter une commande et utiliser son résultat :

```
… $(COMMANDE)
```

2.2. Opérateurs

Incrément

On peut incrémenter une variable avec ++.

```
let NOM++
```

Opérateurs arithmétiques

Somme	+
Différence	-
Produit	*
Quotient	/

3. Tableaux

Déclaration	`declare -a NOM`
Modification	`NOM[INDICE]=VALEUR`
Accès	`${NOM[INDICE]}`

4. Paramètres

Accès aux paramètres

Les paramètres d'un script (et d'une fonction, cf. page 3) sont accessibles grâce aux variables 1, 2, … 9.

Premier	$1
Deuxième	$2
…	
Neuvième	$9

Décalage avec shift

Avec cette méthode, il n'est pas possible d'accéder aux paramètres à partir du neuvième. On utilise pour cela la commande `shift`, qui a pour effet de remplacer le premier par le deuxième, le deuxième par le troisième, etc. jusqu'au neuvième qui prend la valeur du dixième.

Variables spéciales

$#	Nombre de paramètres
$*	Tous les paramètres
$0	Nom du script

II. Structures de contrôle

1. Tests

Effectuer un test

On peut effectuer les tests de deux manières ; avec la commande `test` ou avec la syntaxe de Bash.

```
test TEST
[ TEST ]
```

Ces deux syntaxes revoient `0` si et seulement si le test est validé.

Codes de retour et scripts

Les tests sont fondés sur le code de retour de chaque commande. Le code de retour par défaut d'un script est celui de sa dernière commande effectuée. Pour changer ce comportement, on utilise la commande `exit` :

```
exit CODE
```

Par convention, `0` signifie réussite et n'importe quel autre code signifie erreur.

Tests sur les fichiers

On souhaite obtenir des informations sur un document, appelé `NOM`.

Est-ce un fichier ?	`-f NOM`
Est-ce un répertoire ?	`-d NOM`

Tests sur les chaînes

Attention, les tests ne portent pas sur la valeur numérique des chaînes !

Égalité	`a = b`
Différence	`a != b`
Plus petite	`a \< b`
Plus grande	`a \> b`

Tests arithmétiques

Égalité	`a -eq b`
Différence	`a -ne b`
Inférieur ou égal	`a -le b`
Strictement inférieur	`a -lt b`
Supérieur ou égal	`a -ge b`
Strictement supérieur	`a -gt b`

Agir selon le résultat d'un test

Effectuer une commande uniquement si la précédente a réussi :

```
COMMANDE_1 && COMMANDE_2
```

Effectuer une commande uniquement si la précédente a échoué :

```
COMMANDE_1 || COMMANDE_2
```

On retrouve d'autres utilisations des codes de retour dans le `if` (page 3) et le `while` (page 4).

2. Fonctions

Syntaxe

```
function NOM {
      …
}
```

On peut utiliser les variables locales *(cf. Déclaration locale, page 1)* et les paramètres *(cf Paramètres, page 2)* comme si la fonction était un script à part entière.

3. Conditions

3.1. If

Syntaxe simple

```
if TEST
then
      …
fi
```

Utilisation d'un sinon

```
if TEST
then
      …
elif TEST
then
      …
else
      …
fi
```

Dans les deux exemples précédents, `TEST` pouvait faire référence à un test *(cf. Tests, page 2)*, ou à n'importe qu'elle autre commande *(cf. Codes de retour et scripts, page 3)*.

3.2. Switch

Le `switch` est très particulier en Bash.

Syntaxe

```
case VARIABLE in
CAS)
      … ;;
AUTRE_CAS)
      … ;;
*)
      … ;;
esac
```

Les deux `;;` sont l'équivalent du mot-clef « break » en Java.

On peut effectuer plusieurs commandes par cas en les séparant par un point-virgule ;.

Les cas

Si on veut faire la même action pour deux cas distincts, on peut utiliser le symbole pipe ;

```
CAS1 | CAS2)
```

On peut aussi utiliser des jokers pour faire des cas plus généraux, par exemple on peut sélectionner `oui`, `OUI`, `OuI`, etc. en utilisant :

```
[oO][uU][iI])
```

4. Boucles

4.1. For each

Les boucles ont pour utilité d'effectuer le même code plusieurs fois.

On peut utiliser le mot-clef `break` pour sortir d'une boucle prématurément.

Syntaxe

```
for VARIABLE in LISTE
do
    …
done
```

Ici, `LISTE` peut être un fichier ou une commande, dans ce cas `VARIABLE` parcours alors les lignes de sa sortie standard.

4.2. While

Syntaxe

```
while TEST
do
    …
done
```

`TEST` peut faire référence à un test *(cf. Tests, page 2)*, ou à n'importe qu'elle autre commande *(cf. Codes de retour et scripts, page 3)*.

III. PROCESSUS

Avant et arrière plan

Arrêter	`CTRL+C`
Mettre en pause	`CTRL+Z`
Relancer en avant-plan	`fg`
Relancer en arrière-plan	`bg`
Lancer en arrière-plan	`COMMANDE &`
Liste des commandes en cours	`jobs` `ps`

Envoyer des signaux

Il est possible d'envoyer des signaux aux processus avec la commande `kill` :

```
kill -SIGNAL PID
```

Les principaux signaux sont `SIGTERM` (`15`) et `SIGKILL` (`9`) ; la liste complète peut être trouvée avec :

```
kill -l
```

Plus d'options

ID du dernier processus lancé	`$!`
ID du shell courant	`$$`
Attendre la fin d'un processus en arrière-plan	`wait PID`
Code de retour de la dernière commande	`$?`