Scripts Shell

Laurent Tichit

6 avril 2011

Plan

Scripts

Commentaires

Les commentaires dans les scripts sont introduits par le symbole #.
Le Shebang est donc un commentaire spécial.
En effet, il est évalué par le shell appelant, qui va se charger d'appeler le bon interpréteur.

Shebang

Tout script doit commencer par une ligne commencant par `#!` suivi du résultat de la commande **which langage_de_script** :
`tichit@iml230:$` **which bash**

`/bin/bash`

Votre script *bash* doit commencer par :

`#! /bin/bash`

`tichit@iml230:$` **which perl**

`/usr/bin/perl`

Votre script *perl* doit commencer par :

`#! /usr/bin/perl`

Pour exécuter un script

- Soit on indique de facon explicite l'interpréteur :
  perl script.pl
  python script.py
  bash script.sh
- Soit on laisse le shell appeler le bon interpréteur.
  Dans ce cas, il faut donner les droits en exécution :
  chmod +x script.pl
  On pourra donc appeler le script comme n'importe quelle autre commande (à condition que le répertoire où se trouve le script soit contenu dans le PATH).
  C'est dans ce cas là que le Shebang est nécessaire ! !

Shell : variables

Utilisation des variables

`HOME=/home/tichit`

- nom de variable : `HOME`
- valeur de la variable : `$HOME`
- afficher la valeur de la variable : **echo $HOME**
- créer et affecter une variable : **VAR=toto**
- supprimer une variable : **unset VAR**
  Attention : pas d'espace autour de l'opérateur "="

Le PATH

```
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/
```

- Cette variable contient une *liste* de *chemins* de *répertoires* dans lesquelles le Shell ira chercher les exécutables.
- Si un exécutable donné ne se trouve pas dans le PATH, le Shell ne pourra pas la trouver, et nous donnera un message d'erreur : `bash: cmd: command not found`

Affectation des variables

- créer une variable avec l'affectation : **VAR=toto**
  Attention : pas d'espace autour de l'opérateur "="
- commande **read variables...** affecte à chaque *variable* un mot de la première ligne lue sur l'entrée standard.

  `read var1 var2 var3 var4`

  Fonctionnement :

  1. **read** attend qu'une ligne arrive sur l'entrée standard (**n**).
  2. Cette ligne est découpée en mots (séparateurs : espace, tabulation).
  3. Le $n^i eme$ mot est affecté à la $n^i eme$ variable.
  4. S'il y a plus de mots que de variables, la dernière variable contient toute la fin de la ligne.

  La liste des séparateurs se trouve dans la variable **IFS**. On peut la modifier.

Lister les variables

- lister toutes les variables : **set**
- lister toutes les variables d'environnement (celles qui sont visibles partout) : **printenv**

Bash : arguments d'un script

arguments génériques

- $# est équivalent à *argc* (-1) en C : c'est le nombre d'arguments que l'on a passé au script lors de l'appel.
- "$@" est équivalent à *argv* en C : c'est la liste des arguments que l'on a passé au script.

accès direct

- $0 est le nom complet du script (pour avoir le nom court : **basename $0**)
- $1 est le premier argument passé au script.
- ...
- $9 est le neuvième.

Impossible d'accéder directement aux suivants ($10 n'existe pas).

Commande interne *shift*

shift permet de **décaler** la liste des arguments :

- On perd le premier.
- Le second devient le premier et se trouve donc dans $1
- ...
- Le dixième devient le neuvième et se trouve donc dans $9

set et IFS

La commande **set chaine_de_caractères** permet d'affecter la liste d'arguments ($1, ..., $9, "$@", $#)
Elle se sert aussi des séparateurs, et donc de la variable d'environnement *IFS*. Ex :
set salut tout le monde, modifiera $1, $2, $3, $4 et $# vaudra 4.

Exercices

- Ecrivez le script *s1.sh* qui affiche le nombre et la valeur des arguments la ligne de commande.
- Ecrivez le script *s2.sh* qui affiche le nom du script, le 1er et le 10eme argument.

Erreur et code retour

Erreur de commande

Si la commande est erronnée, un message d'erreur apparait :

```
tichit@iml230:~$ heure

bash: heure : commande introuvable
```

Les messages d'erreur apparaissent sur la **sortie standard d'erreur**. Cette sortie est redirigée sur le terminal courant.

```
tichit@iml230:~$ rep -

bash: rep: commande introuvable

tichit@iml230:~$ cat toto

cat: toto: Aucun fichier ou répertoire de ce type

tichit@iml230:~$ rm .

rm: ne peut enlever « . » or « .. »

tichit@iml230:~$ rm public_html/

rm: ne peut enlever ‘public\_html/’: est un répertoire
```

Le code de retour

Il s'agit d'un **entier** permettant de savoir si une commande a réussi ou échoué.

- Déroulement normal : statut 0.
- Déroulement anormal : statut différent de 0.

Le code de retour sera rendu dans la variable **"?"**, ce qui permettra de réaliser des opérations conditionnelles lors de l'écriture de **scripts** shell.

```
tichit@iml230:~$ whoami

tichit

tichit@iml230:~$ echo $?

0

tichit@iml230:~$ rm public_html/

rm: ne peut enlever ‘public\_html/’: est un répertoire

tichit@iml230:~$ echo $?

1
```

La commande **exit**

Tout script doit indiquer au système s'il s'est déroulé avec succès ou s'est terminé sur un échec.
Pour celà, on se sert de la commande **exit entier** qui termine l'exécution du script.
exit 0 indique un succès, **exit n** (avec $n \neq 0$) indique un échec.
Tous vos scripts doivent se terminer par un **exit**.

Bash : structures de contrôle

Commande

```
if commande1 ; then
 commande2
 commande3
 ...
elif commande4 ; then
 commande5
 commande6
 ...
else
 commande7
 commande8
 ...
fi
```

Commande **if**

Les blocs `elif` et `else` sont facultatifs.
commande1 peut réussir ou échouer.
Si elle réussi, le bloc *then* est effectué.
Si elle échoue, le bloc *else* est effectué.

Si on redirige l'entrée standard du **if** (ou plutôt du **fi**) vers un fichier, **commande1** aura aussi son entrée standard redirigée vers ce fichier.

Commande **test**

La commande **test** est une commande qui ne fait rien d'autre que réussir ou échouer.
Elle sert très souvent d'argument à la commande **if**.
Elle possède un grand nombre d'options.
Elle peut aussi s'écrire [**...**]
Ex :

- **test -f /home/test** ou [**-f /home/test**] permet de savoir si le fichier */home/test* est un fichier ordinaire.
- **test $1** \> **a** permet de savoir si le premier argument est supérieur (lexicographiquement) à "a".
- **test $1 = help** permet de savoir si le premier argument est "help".
- **test $# -gt 3** permet de savoir si le nombre d'arguments est supérieur ou égal (numériquement) à 3.
- ...

Exercices

- Ecrivez le script *s3.sh* qui réussit s'il a au moins un argument et si la valeur du premier argument est comprise entre 'b' et 'd'.
- Ecrivez le script *s4.sh* qui appelle *s3.sh* avec son premier argument et qui affiche (**echo)** si *s3.sh* a réussi.

true et **false**

Ces deux commandes ne font rien.

- **true** se termine sur un succès.
- **false** se termine sur un échec.

Boucle **while**

```
while commande1 ; do
 commande2
 commande3
done
```

Si on redirige l'entrée standard du **while** (ou plutôt du **done**) vers un fichier, **commande1** aura aussi son entrée standard redirigée vers ce fichier.

Boucle **for**

```
for variable in liste ; do
 commande1
 commande2
done
```

variable est un nom de variable libre. *liste* est en fait une chaine de caractères qui devrait contenir des caractères de séparation (tabulations, espaces).

Boucle **for** sans arguments

```
for variable ; do
 commande1
 commande2
done
```

idem à

```
for variable in "$@" ; do
 commande1
 commande2
done
```

Exercices

- Ecrivez le script *s5.sh* qui affiche chaque argument sur une ligne en vous servant de **while** et **shift**.
- Même question mais avec une boucle **for** (script *s6.sh*).
- Enfin, écrivez le script *s7.sh* qui indique si les fichiers passés en paramètre existent et sont exécutables.

Exercice (avec while)

Exécutez le script suivant :

```
#!/bin/bash
x=1
while [ $x -le 5 ]
do
  echo "Hello World $x"
  x=$(( $x + 1 ))
done
```

Exercice (avec while)

Exécutez le script suivant :

```
#!/bin/bash
counter=$1
factorial=1
while test $counter -gt 0
do
    factorial=$((factorial * counter ))
    counter=$((counter - 1 ))
done
echo $factorial
```

Exercice (avec while)

Exécutez le script suivant :

```
#!/bin/bash
while :
do
echo "boucle infinie [ CTRL-C pour arreter ]"
done
```

Exercice (avec for)

Exécutez le script suivant :

```
#!/bin/bash
for i in 1 2 3 4 5
do
   echo "Hello World $i"
done
```

Exercice (avec for)

Exécutez le script suivant :

```
#!/bin/bash
for i in {1..5}
do
   echo "Hello World $i"
done
```

Exercice (avec for)

Exécutez le script suivant :

```
#!/bin/bash
for (( c=1; c<=5; c++ ))
do
echo "Hello World $i"
done
```

Exercice (avec for)

Exécutez le script suivant :

```
#!/bin/bash
for (( ; ; ))
do
   echo "boucle infinie [ CTRL-C pour arreter ]"
done
```

switch

```
case variable in
 expReg1)
   cmd1
   cmd2
   ;;
 expReg2)
   cmd3
   cmd4
   ;;
 *)
   cmd5
   cmd6
esac
```

variable est un nom de variable libre. *expReg1* et *expReg2* sont des expressions régulières (*, ?, [^-], |)

Exercice

- Créez une commande que vous appelerez **rm**, et qui remplacera la commande **rm** du système.
  Votre script aura pour but de déplacer tous les fichiers en argument dans le répertoire ~/Trash (la corbeille).
  Si ce répertoire n'existe pas, le créer.
  Si l'option **-f** est passé à la commande, dans ce cas, appeler la commande système **/bin/rm**.
- placer ce script dans votre répertoire ~**/bin** (créez-le si ce répertoire n'existe pas), et vérifiez (**which**) que vous appelez désormais votre **rm**.
- Dans une seconde version, faites en sorte que si la corbeille contient déjà un fichier de même nom, il ne soit pas écrasé (**date, tr**).

Fonctions

- Pour définir une fonction en bash :

```
ma_fonction () {
commandes unix
...
}
```

- Appel de fonction : une fonction est vue (donc appelée) comme une commande : **ma_fonction f1.txt**
- Les arguments passés à la fonction sont accédés par le tableau d'arguments ($1, ..., $9, $#, "$@", shift)
- Pour sortir (avant la dernière commande) : **return n** avec n = 0 : succès, n > 0 : échec.
- déclaration de variable locale : **local i=0**
- une fonction ne doit afficher sur sa sortie standard QUE ce qu'elle est censée retourner.

Terminaisons

Pour terminer :

- un script : **exit n** (n = code de retour indiquant succès ou échec).
- une fonction : **return n** (n = code de retour indiquant succès ou échec).
- une boucle : **break n** (n = nombre de boucles imbriquées dont on sort).
- le tour de boucle en cours : **continue**.

Exercice (fonctions)

Exécutez le script suivant :

```
#!/bin/bash
usage() {
  echo "Usage: $0 filename"
  exit 1
}
file_exits() {
  test -f "$1" && return 0 || return 1
}
test $# -eq 0 && usage
if file_exits "$1"
then
  echo "File found"
else
  echo "File not found"
fi
```

Bash : fichiers de configuration

$HOME/.bash_profile

C'est un fichier contenant des commandes *bash*. Le contenu de ce fichier est évalué une seule fois, lors d'un lancement d'un shell de connexion (malheusement si l'on se connecte en mode graphique, le contenu de ce fichier là n'est pas évalué).

$HOME/.bashrc

C'est un fichier contenant des commandes *bash*. Le contenu de ce fichier est évalué à chaque ouverture de nouveau *bash*.

Exercice

Ouvrez votre fichier de configuration `~/.bashrc` :

- décommentez les 3 lignes `alias ll=...`. Un alias est un *surnom*, un raccourci. Au passage, à quoi servent les options de *ls* figurant dans ces 3 lignes ?
- rajoutez en fin de fichier la ligne :
  `PATH=$PATH:.`
  Ceci vous permet de rajouter le répertoire de travail à la liste des répertoires contenant vos exécutable. De cette façon, vous pourrez exécuter vos scripts en tapant leur `nom`, au lieu de `./nom`.