

Introduction au Système UNIX

Séance 3 : Shell bash

Pourquoi utiliser bash ?

Bash est une version évoluée du shell sh (le "Bourne shell"). Le shell peut être utilisé comme un simple interpréteur de commande, mais il est aussi possible de l'utiliser comme langage de programmation interprété (scripts).

La connaissance du shell est indispensable au travail de l'administrateur unix :

- le travail en "ligne de commande" est souvent beaucoup plus efficace qu'à travers une interface graphique;
- dans de nombreux contextes (serveurs, systèmes embarqués, liaisons distantes lentes) on ne dispose pas d'interface graphique;
- le shell permet l'automatisation aisée des tâches répétitives (scripts);
- de très nombreuses parties du système UNIX sont écrites en shell, il faut être capable de les lire pour comprendre et éventuellement modifier leur fonctionnement.

Autres versions de shell

Il existe plusieurs versions de shell : sh (ancêtre de bash), csh (C shell), ksh (Korn shell), zsh, etc. Nous avons choisi d'enseigner bash car il s'agit d'un logiciel libre, utilisé sur toutes les distributions récentes de Linux et de nombreuses autres variantes d'UNIX.

Connaissant bash, l'apprentissage d'un autre shell sur le terrain ne devrait pas poser de difficultés.

Les mauvais côtés des shell

Le shell possède quelques inconvénients :

- documentation difficile d'accès pour le débutant (la page de manuel "man bash" est très longue et technique);
- messages d'erreurs parfois difficiles à exploiter, ce qui rend la mise au point des scripts fastidieuse;
- syntaxe cohérente, mais ardue (on privilégie la concision sur la clarté ;
- relative lenteur (langage interprétésans pré-compilation).

Ces mauvais côtés sont compensés par la facilité de mise en oeuvre (pas besoin d'installer un autre langage sur votre système).

Variables et évaluation

Les variables sont stockées comme des chaînes de caractères.

Les variables *d'environnement* sont des variables exportées aux processus (programmes) lancés par le shell. Les variables d'environnement sont gérées par UNIX, elles sont donc accessibles dans tous les langages de programmation (voir plus loin).

Pour définir une variable :

```
$ var='ceci est une variable'
```

Attention : pas d'espaces autour du signe égal. Les quotes (apostrophes) sont nécessaires si la valeur contient des espaces. C'est une bonne habitude de toujours entourer les chaînes de caractères de quotes.

Pour utiliser une variable, on fait précéder son nom du caractère "\$" :

```
$ echo $var
```

ou encore :

```
$ echo 'bonjour, ' $var
```

Dans certains cas, on doit entourer le nom de la variable par des accolades :

```
$ X=22  
$ echo Le prix est ${X}0 euros
```

affiche "Le prix est de 220 euros" (sans accolades autour de X, le shell ne pourrait pas savoir que l'on désigne la variable X et non la variable X0).

Lorsqu'on utilise une variable qui n'existe pas, le bash renvoie une chaîne de caractère vide, et n'affiche pas de message d'erreur (contrairement à la plupart des langages de programmation, y compris csh) :

```
$ echo Voici${UNE_DROLE_DE_VARIABLE}!
```

affiche "Voici!".

Pour indiquer qu'une variable doit être exportée dans l'environnement (pour la passer aux commandes lancées depuis ce shell), on utilise la commande **export** :

```
$ export X
```

ou directement :

```
$ export X=22
```

évaluation, guillemets et quotes

Avant évaluation (interprétation) d'un texte, le shell substitue les valeurs des variables. On peut utiliser les guillemets (") et les quotes (') pour modifier l'évaluation :

- les guillemets permettent de grouper des mots, sans supprimer le remplacement des variables. Par exemple, la commande suivante ne fonctionne pas :

```
$ x=Hauru no  
bash: no: command not found
```

Avec des guillemets c'est bon:
\$ x="Hauru no"

On peut utiliser une variable entre guillemets :

```
$ y="Titre: $x Ugoku Shiro"  
$ echo $x  
Titre: Hauru no Ugoku Shiro
```

- les quotes (apostrophes) groupent les mots et suppriment toute évaluation :

```
$ z='Titre: $x Ugoku Shiro'  
$ echo $z  
Titre: $x Ugoku Shiro
```

Découpage des chemins

Les scripts shell manipulent souvent chemins (*pathnames*) et noms de fichiers. Les commandes `basename` et `dirname` sont très commodes pour découper un chemin en deux parties (répertoires, nom de fichier) :

```
$ dirname /un/long/chemin/vers/toto.txt  
/un/long/chemin/vers  
$ basename /un/long/chemin/vers/toto.txt  
toto.txt
```

évaluation de commandes

Il est courant de stocker le résultat d'une commande dans une variable. Nous entendons ici par "résultat" la chaîne affichée par la commande, et non son code de retour.

Bash utilise plusieurs notations pour cela : les *back quotes* (```) ou les parenthèses :

```
$ REP=`dirname /un/long/chemin/vers/toto.txt`  
$ echo $REP  
/un/long/chemin/vers
```

ou, de manière équivalente :

```
$ REP=$(dirname /un/long/chemin/vers/toto.txt)  
$ echo $REP  
/un/long/chemin/vers
```

(attention encore une fois, pas d'espaces autour du signe égal).

La commande peut être compliquée, par exemple avec un tube :

```
$ Fichiers=$(ls /usr/include/*.h | grep std)  
$ echo $Fichiers  
/usr/include/stdint.h /usr/include/stdio_ext.h /usr/include/stdio.h  
/usr/include/stdlib.h /usr/include/unistd.h
```

Découpage de chaînes

Bash possède de nombreuses fonctionnalités pour découper des chaînes de caractères. L'une des plus pratiques est basée sur des motifs.

La notation `##` permet d'éliminer la *plus longue* chaîne en correspondance avec le motif :

```
$ Var='tonari no totoro'  
$ echo ${Var##*to}  
ro
```

ici le motif est `*to`, et la plus longue correspondance "tonari no toto"¹. Cette forme est utile pour récupérer l'extension (suffixe) d'un nom de fichier :

```
$ F='rep/bidule.tgz'  
$ echo ${F##*.}  
tgz
```

La notation `#` (un seul `#`) est similaire mais élimine la *plus courte* chaîne en correspondance :

```
$ Var='tonari no totoro'  
$ echo ${Var#*to}  
nari no totoro
```

De façon similaire, on peut éliminer la fin d'une chaîne :

```
$ Var='tonari no totoro'  
$ echo ${Var%no*}  
tonari
```

Ce qui permet de supprimer l'extension d'un nom de fichier :

```
$ F='rep/bidule.tgz'  
$ echo ${F%.*}  
rep/bidule
```

`%` prend la plus courte correspondance, et `%%` prend la plus longue :

```
$ Y='archive.tar.gz'  
$ echo ${Y%.*}  
archive.tar  
$ echo ${Y%%.*}  
archive
```

Exécution conditionnelle

L'instruction `if` permet d'exécuter des instructions si une condition est vraie. Sa syntaxe est la suivante :

```
if [ condition ]  
then  
    action  
fi
```

`action` est une suite de commandes quelconques. L'indentation n'est pas obligatoire mais très fortement recommandée pour la lisibilité du code. On peut aussi utiliser la forme complète :

```
if [ condition ]  
then  
    action1  
else  
    action2  
fi
```

ou encore enchaîner plusieurs conditions :

```
if [ condition1 ]
```

```

then
    action1
elif [ condition2 ]
    action2
elif [ condition3 ]
    action3
else
    action4
fi

```

Opérateurs de comparaison

Le shell étant souvent utilisé pour manipuler des fichiers, il offre plusieurs opérateurs permettant de vérifier diverses conditions sur ceux-ci : existence, dates, droits. D'autres opérateurs permettent de tester des valeurs, chaînes ou numériques. Le tableau ci-dessous donne un aperçu des principaux opérateurs :

Opérateur	Description	Exemple
Opérateurs sur des fichiers		
<code>-e filename</code>	vrai si <i>filename</i> existe	[<code>-e /etc/shadow</code>]
<code>-d filename</code>	vrai si <i>filename</i> est un répertoire	[<code>-d /tmp/trash</code>]
<code>-f filename</code>	vrai si <i>filename</i> est un fichier ordinaire	[<code>-f /tmp/glop</code>]
<code>-L filename</code>	vrai si <i>filename</i> est un lien symbolique	[<code>-L /home</code>]
<code>-r filename</code>	vrai si <i>filename</i> est lisible (r)	[<code>-r /boot/vmlinuz</code>]
<code>-w filename</code>	vrai si <i>filename</i> est modifiable (w)	[<code>-w /var/log</code>]
<code>-x filename</code>	vrai si <i>filename</i> est exécutable (x)	[<code>-x /sbin/halt</code>]
<code>file1 -nt file2</code>	vrai si <i>file1</i> plus récent que <i>file2</i>	[<code>/tmp/foo -nt /tmp/bar</code>]
<code>file1 -ot file2</code>	vrai si <i>file1</i> plus ancien que <i>file2</i>	[<code>/tmp/foo -ot /tmp/bar</code>]
Opérateurs sur les chaînes		
<code>-z chaine</code>	vrai si la <i>chaine</i> est vide	[<code>-z "\$VAR"</code>]
<code>-n chaine</code>	vrai si la <i>chaine</i> est non vide	[<code>-n "\$VAR"</code>]
<code>chaine1 = chaine2</code>	vrai si les deux chaînes sont égales	[<code>"\$VAR" = "totoro"</code>]
<code>chaine1 != chaine2</code>	vrai si les deux chaînes sont différentes	[<code>"\$VAR" != "tonari"</code>]
Opérateurs de comparaison numérique		
<code>num1 -eq num2</code>	égalité	[<code>\$nombre -eq 27</code>]
<code>num1 -ne num2</code>	inégalité	[<code>\$nombre -ne 27</code>]
<code>num1 -lt num2</code>	inférieur (<)	[<code>\$nombre -lt 27</code>]
<code>num1 -le num2</code>	inférieur ou égal (<=)	[<code>\$nombre -le 27</code>]
<code>num1 -gt num2</code>	supérieur (>)	[<code>\$nombre -gt 27</code>]
<code>num1 -ge num2</code>	supérieur ou égal (>=)	[<code>\$nombre -ge 27</code>]

Quelques points délicats doivent être soulignés :

- Toutes les variables sont de type chaîne de caractères. La valeur est juste convertie en nombre pour les opérateurs de conversion numérique.
- Il est nécessaire d'entourer les variables de guillemets (") dans les comparaisons. Le code suivant affiche "OK" si \$var est égale à "tonari no totoro" :

```
if [ "$myvar" = "tonari no totoro" ]
then
    echo "OK"
fi
```

Par contre, si on écrit la comparaison comme `if [$myvar = "tonari no totoro"]` le shell déclenche une erreur si \$myvar contient plusieurs mots. En effet, la substitution des variables a lieu avant l'interprétation de la condition.

Scripts shell

Un script bash est un simple fichier texte exécutable (droit x) commençant par les caractères `#!/bin/bash` (doivent être les premiers caractères du fichier).

Voici un exemple de script :

```
#!/bin/bash

if [ "${1##*.}" = "tar" ]
then
    echo $1 est une archive tar
else
    echo $1 n'est pas une archive tar
fi
```

Ce script utilise la variable \$1, qui est le premier argument passé sur la ligne de commande.

Arguments de la ligne de commande

Lorsqu'on entre une commande dans un shell, ce dernier sépare le nom de la commande (fichier exécutable ou commande interne au shell) des arguments (tout ce qui suit le nom de la commande, séparé par un ou plusieurs espaces). Les programmes peuvent utiliser les arguments (options, noms de fichiers à traiter, etc).

En bash, les arguments de la ligne de commande sont stockés dans des variables spéciales :

\$0	Le nom du programme
\$1, \$2, ...	Les paramètres
\$#	le nombre de paramètres
\$*	tous les paramètres

Le programme suivant illustre l'utilisation de ces variables :

```
#!/bin/bash

echo 'programme :' $0
echo 'argument 1 :' $1
echo 'argument 2 :' $2
echo 'argument 3 :' $3
echo 'argument 4 :' $4
echo "nombre d'arguments :" $#
echo "tous:" $*
```

Exemple d'utilisation, si le script s'appelle "myargs.sh" :

```
$ ./myargs.sh un deux trois
programme : ./myargs.sh
argument 1 : un
argument 2 : deux
argument 3 : trois
argument 4 :
nombre d'arguments : 3
tous: un deux trois
```

Autres structures de contrôle

Nous avons déjà évoqué l'instruction `if` et les conditions. On utilise souvent des répétitions (`for`) et des choix multiples (`case`).

Boucle for

Comme dans d'autres langages (par exemple python), la boucle `for` permet d'exécuter une suite d'instructions avec une variable parcourant une suite de valeurs. Exemple :

```
for x in un deux trois quatre
do
    echo x= $x
done
```

affichera :

```
x= un
x= deux
x= trois
x= quatre
```

On utilise fréquemment `for` pour énumérer des noms de fichiers, comme dans cet exemple :

```
for fichier in /etc/rc*
do
    if [ -d "$fichier" ]
    then
        echo "$fichier (repertoire)"
    else
        echo "$fichier"
    fi
done
```

Ou encore, pour traiter les arguments passés sur la ligne de commande :

```
#!/bin/bash

for arg in $*
do
    echo $arg
done
```

Instruction case

L'instruction case permet de choisir une suite d'instruction suivant la valeur d'une expression :

```
case "$x" in
    go)
        echo "demarrage"
        ;;
    stop)
        echo "arret"
        ;;
    *)
        echo "valeur invalide de x ($x)"
esac
```

Noter les deux ; pour signaler la fin de chaque séquence d'instructions.

Définition de fonctions

Il est souvent utile de définir des fonctions. La syntaxe est simple :

```
mafonction() {
    echo "appel de mafonction..."
}
```

```
mafonction
mafonction
```

qui donne :

```
appel de mafonction...
appel de mafonction...
```

Voici pour terminer un exemple de fonction plus intéressant :

```
tarview() {
    echo -n "Affichage du contenu de l'archive $1 "
    case "${1##*.}" in
        tar)
            echo "(tar compressé)"
            tar tvf $1
            ;;
        tgz)
            echo "(tar compressé gzip)"
            tar tzvf $1
            ;;
        bz2)
            echo "(tar compressé bzip2)"
            cat $1 | bzip2 -d | tar tvf -
    esac
}
```

```
    ;;  
    *)  
    echo "Erreur, ce n'est pas une archive"  
    ;;  
esac  
}
```

Plusieurs points sont à noter :

- `echo -n` permet d'éviter le passage à la ligne;
- La fonction s'appelle avec un argument (**\$1**)
`tarview toto.tar`