Scripts shell

Patrick MARCEL, Université d'Orléans

L2 Outils du développeur — S3

Scripts élémentaires

Un **script shell** est un fichier contenant une séquence de **commandes**.

```
$ bash monscript
```

La commande **bash** accepte un nom de script en paramètre qui est alors exécuté au lieu de passer en mode interactif.

```
$ . monscript
```

La **commande interne** `.` exécute les commandes du script dans le shell courant.

Scripts exécutables

Le **système d'exploitation** peut exécuter :

- des programmes au format binaire idoine (ELF)
- des scripts commençant par les caractères `#!`

```
#!program arg1 arg2 ... argN\n
```

Le programme `program` est exécuté avec en arguments : `arg1`, `arg2`, ..., `argN` suivi du chemin d'accès au script suivi des arguments passés au script.

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

Paramètres

`$1`, `$2`, ..., `$9` : arguments passés au script

`${10}`, `${11}`, ... : arguments suivants

`$@`, `$*` : liste de tous les arguments

`$#` : nombre total d'arguments

`$0` : nom de la commande invoquée

```
for i; do ...; done
```

Différence entre $@ et $*

En dehors de guillemets doubles, $@ = $*.

L'expansion de "$@" place chaque paramètre dans un champ distinct.

```
for i in "$@"; do ...; done
```

L'expansion de "$*" sépare les paramètres par $IFS.

```
IFS='!'; echo "$*"
```

Décalage avec shift

`shift` décale les paramètres : `$1` reçoit l'ancien `$2`, `$2` reçoit l'ancien `$3`, *etc* L'ancienne valeur `$1` est perdue.

```
while [ $# -ne 0 ]; do
 echo $1 $2
 shift 2
done
```

`shift n` décale les paramètres de n cases à la fois.

Affectation avec `set`

```
set -- the cake is a lie
```

La commande interne `set` permet de redéfinir les paramètres de position en les remplaçant par ses arguments.

La variable spéciale `$#` est modifiée en conséquence.

Options avec `set`

La commande interne `set` est aussi utilisée pour activer ou désactiver des **options** dont :

- `-e` errexit
- `-n` noexec
- `-v` verbose
- `-x` xtrace

Lecture avec `read`

```
read -p 'login: ' login
read -s -p 'passwd: ' pass
```

La commande interne `read` lit une ligne sur l'entrée standard, la découpe selon `IFS` et stocke le résultat dans les variables nommées en arguments.

```
while IFS=: read user pass uid gid id home
shell; do
 echo "$uid => $user ($home)"
done < /etc/passwd
```

Définition de fonctions

```
somme() {
 echo $(($1+$2));
}
```

Une **fonction shell** s'exécute comme une commande interne, ses arguments remplacent temporairement les variables positionnelles.

```
return 54
```

La commande interne `return` permet de quitter immédiatement une fonction et de positionner la valeur de retour `$?`.

Écrasement avec `exec`

```
#!/bin/bash
SPELL=ispell
exec nano "$@"
```

La commande interne `exec` exécute la commande passée en arguments sans créer de nouveau processus mais en remplaçant le shell courant.

Encore un peu de variables

`$$` PID du processus courant (le shell)

```
ls ${DATA:-/usr/share/default}
```

`${variable:-mot}` valeur de la variable ou bien `mot` si elle est non définie ou nulle.

```
MAN=${DATA:=$(pwd)}/man
```

`${variable:=mot}` avant de l'évaluer, affecte `mot` à `variable` si celle-ci est non définie ou nulle.

Bonnes pratiques getopts

```
aflag=; bflag=; OPTERR=0
while getopts ab: name
do
    case $name in
    a)    aflag=1;;
    b)    bflag=1
          bval="$OPTARG";;
    *)   printf "Usage: %s: [-a] [-b value] args\n" $0
          exit 2;;
    esac
done
if [ ! -z "$aflag" ]; then
    printf "Option -a specified\n"
fi
if [ ! -z "$bflag" ]; then
    printf "Option -b '%s' specified\n" "$bval"
fi
shift $(($OPTIND - 1))
printf "Remaining arguments are: %s\n" "$*"
```