Scripts shell

Sergiu IVANOV

sergiu.ivanov@u-pec.fr

Les diapos disponibles en ligne :

http://lacl.fr/~sivanov/doku.php?id=fr:cours_de_systemes_et_reseaux

Retour aux origines

Qu'est-ce que la ligne de commande ?

Plan du cours

1. Le shell : l'image générale
2. Scripts shell : les bases
3. Structures de contrôle
4. Tableaux
5. Fonctions

Plan du cours

1. Le shell : l'image générale
2. Scripts shell : les bases
3. Structures de contrôle
4. Tableaux
5. Fonctions

Le shell : l'interface en ligne de commande

Le shell = la couche logicielle qui constitue l'interface utilisateur d'un système d'exploitation

Le shell : l'interface en ligne de commande

Le shell = la couche logicielle qui constitue l'interface utilisateur d'un système d'exploitation

Très souvent le mot « shell » fait référence à une interface en ligne de commande.

Le shell : l'interface en ligne de commande

Le shell = la couche logicielle qui constitue l'interface utilisateur d'un système d'exploitation

Très souvent le mot « shell » fait référence à une interface en ligne de commande.

Des fois on entend dire « shell graphique ».

- l'interface de Windows
- l'interface d'Eclipse

L'interface texte : avantages et désavantages

- L'interface consiste juste en l'affichage de la lecture du texte.
- Peut être utilisé sur des systèmes avec peu de ressources ou par réseau.
- Intégration facile de nouvelles fonctions.
- Automatisation facile.

- Représentation non intuitive d'objets.
  - tout est texte
- Aide contextuelle relativement pauvre.
  - l'utilisateur est censé savoir auparavant ce qu'il faut faire

Les shells : quelques titres

`sh` : Bourne Shell

- était le shell UNIX par défaut
- beaucoup de shells modernes sont compatibles

Les shells : quelques titres

`sh` : Bourne Shell

- était le shell UNIX par défaut
- beaucoup de shells modernes sont compatibles

`bash` : Bourne Again Shell

- le shell par défaut sous la plupart de distributions Linux et sous MacOS

Les shells : quelques titres

`sh` : Bourne Shell

- était le shell UNIX par défaut
- beaucoup de shells modernes sont compatibles

`bash` : Bourne Again Shell

- le shell par défaut sous la plupart de distributions Linux et sous MacOS

`zsh` : Z Shell

- un shell avec des fonctionnalités avancées

Les shells : quelques titres

`sh` : Bourne Shell

- était le shell UNIX par défaut
- beaucoup de shells modernes sont compatibles

`bash` : Bourne Again Shell

- le shell par défaut sous la plupart de distributions Linux et sous MacOS

`zsh` : Z Shell

- un shell avec des fonctionnalités avancées

`cmd` : la ligne de commande Windows

- offre des fonctionnalités pareils aux autres shells, mais est moins utilisé

Les shells : quelques titres

`sh` : Bourne Shell

- était le shell UNIX par défaut
- beaucoup de shells modernes sont compatibles

`bash` : Bourne Again Shell

- le shell par défaut sous la plupart de distributions Linux et sous MacOS

`zsh` : Z Shell

- un shell avec des fonctionnalités avancées

`cmd` : la ligne de commande Windows

- offre des fonctionnalités pareils aux autres shells, mais est moins utilisé

PowerShell : un shell Windows qui se fonde sur .NET

Scripts shell

Script shell $=$ un programme pour un shell

Scripts shell

Script shell $=$ un programme pour un shell

- permet de vérifier des conditions, de faire des boucles et des fonctions

Scripts shell

Script shell = un programme pour un shell

- permet de vérifier des conditions, de faire des boucles et des fonctions
- est exécuté (interprété) directement par le shell (n'est pas compilé)

Réfléchissons-y

Pourquoi faire des scripts ?

- encore un langage de programmation ?!
- on pourrait écrire des programmes en Java (ou en un autre, vrai langage)

Réfléchissons-y

Pourquoi faire des scripts ?

- encore un langage de programmation ?!
- on pourrait écrire des programmes en Java (ou en un autre, vrai langage)

Les langages de scripts shell sont beaucoup mieux adaptés aux tâches de la gestion d'un système.

Réfléchissons-y

Pourquoi faire des scripts ?

- encore un langage de programmation ?!
- on pourrait écrire des programmes en Java (ou en un autre, vrai langage)

Les langages de scripts shell sont beaucoup mieux adaptés aux tâches de la gestion d'un système.

Les scripts bien faits sont donc plus clairs et plus faciles à maintenir.

Plan du cours

1. Le shell : l'image générale
2. Scripts shell : les bases
3. Structures de contrôle
4. Tableaux
5. Fonctions

Une référence

BASH Programming — Introduction HOW-TO

`http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html`

Le Hello World

```
#!/bin/bash

echo Hello World
```

Le Hello World

```
#!/bin/bash

echo Hello World
```

Afficher `Hello World`

Le Hello World

Le fichier sera interprété par `/bin/bash`

```
#!/bin/bash

echo Hello World
```

Afficher `Hello World`

Le Hello World

Le fichier sera interprété par `/bin/bash`

Le shebang ⟶

```
#!/bin/bash

echo Hello World
```

Afficher `Hello World`

Le Hello World

Le fichier sera interprété par `/bin/bash`

Le shebang ⟶ `#!/bin/bash`

```
echo Hello World
```


Afficher `Hello World`

Pour lancer :

- sauvegarder dans `script.sh`
- donner les droits à l'exécution : `chmod +x script.sh`
- lancer : `./script.sh`

Variables

```
#!/bin/bash
message="Hello World"
echo $message
```

Variables

```
#!/bin/bash
message="Hello World"
echo $message
```

- `$` pour utiliser la variable, pas de `$` à la définition

Variables

```
#!/bin/bash
message="Hello World"
echo $message
```

- $ pour utiliser la variable, pas de $ à la définition
- toutes les variables sont des chaînes de caractères
  - on fera des calculs numériques plus tard

Variables

```
#!/bin/bash
message="Hello World"
echo $message
```

- $ pour utiliser la variable, pas de $ à la définition
- toutes les variables sont des chaînes de caractères
  - on fera des calculs numériques plus tard
- les noms de variable sont sensibles à la casse

Variables

```
#!/bin/bash
message="Hello World"
echo $message
```

- $ pour utiliser la variable, pas de $ à la définition
- toutes les variables sont des chaînes de caractères
  - on fera des calculs numériques plus tard
- les noms de variable sont sensibles à la casse

Pas d'espaces autour de = à la définition !

Variables d'environnement

Variables partagées par tout le système.

- par toutes les instances de shell

La commande `env` affiche la liste de toutes les variables d'environnement.

Variables d'environnement

Variables partagées par tout le système.

- par toutes les instances de shell

La commande `env` affiche la liste de toutes les variables d'environnement.

Pour définir une variable d'environnement :

```
variable=valeur
export variable
```

ou bien

```
export variable=valeur
```

La variable d'environnement PATH

Donne la liste de dossiers où se trouvent les fichiers exécutables.

- `ls`, `cp`, `mv`, etc.

Les noms de dossier sont séparés par « : » :

```
echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/bin
```

Calculs numériques

```
variable=$((10 * 20 - 1))
```

ou bien

```
let "variable = 10 * 20 - 1"
```

```
echo $variable
```

```
-4
```

Calculs numériques

```
variable=$((10 * 20 - 1))
```

ou bien

```
let "variable = 10 * 20 - 1"
```

```
echo $variable
```

```
-4
```

Seulement les calculs avec les nombres entiers sont possibles :

```
echo $((3/2))
```

```
1
```

Plan du cours

1. Le shell : l'image générale
2. Scripts shell : les bases
3. Structures de contrôle
4. Tableaux
5. Fonctions

`if` : l'instruction conditionnelle

```
x="hello"
if [ $x == "hello" ]
then
   echo True
else
   echo False
fi
```

Les retours à la ligne sont importants.

On utilise `!=` pour l'inégalité.

Ces opérateurs ne marchent que pour les chaînes de caractères.

`if` : comparer les nombres

`if [ $value -eq 1 ]`	value $=1$?	(equals)
`if [ $value -ne 1 ]`	value $\neq 1$?	(not equals)
`if [ $value -gt 1 ]`	value >1?	(greater than)
`if [ $value -ge 1 ]`	value ≥ 1?	(greater or equal)
`if [ $value -lt 1 ]`	value <1?	(less than)
`if [ $value -le 1 ]`	value ≤ 1?	(less or equal)

`if` : vérifier plusieurs alternatives

```
x=4
if [ $x -eq 3 ]
then
  echo Three
elif [ $x -eq 4 ]
then
  echo Four
else
  echo Something else
fi
```

case : vérifier plusieurs alternatives

```
x=3
case $x in
  3 )
    echo Three
    ;;
  4 )
    echo Four
    ;;
  * )
    echo Something else
esac
```

`if` : vérifier si une variable est définie

```
if [ -z $x ]
then
  echo x n\'ést pas encore défini
fi

x=3

if [ -n $x ]
then
  echo x est défini maintenant
fi
```

`if` : vérifier qu'une condition n'est pas vraie

```
if [ ! -n $x ]
then
  echo x n\'ést pas encore défini
fi

x=3

if [ ! -z $x ]
then
  echo x est défini maintenant
fi
```

`if` : vérifier qu'une condition n'est pas vraie

```
if [ ! -n "$x" ]
then
  echo x n\'ést pas encore défini
fi

x=3

if [ ! -z "$x" ]
then
  echo x est défini maintenant
fi
```

Souci : vaut toujours mieux mettre des guillemets !

`if` : connecteurs logiques (à l'ancienne)

```
x=4
y=5

if [ "$x" -eq 4 -a "$y" -eq 5 ]
then
  echo '$x est 4 et $y est 5'
fi

if [ "$x" -eq 5 -o "$y" -eq 5 ]
then
  echo '$x est 5 ou $y est 5'
fi
```

if : connecteurs logiques : syntaxe plus claire

```
x=4
y=5

if [[ "$x" -eq 4 ]] && [[ "$y" -eq 5 ]]
then
  echo '$x est 4 et $y est 5'
fi

if [[ "$x" -eq 5 ]] || [[ "$y" -eq 5 ]]
then
  echo '$x est 5 ou $y est 5'
fi
```

`if` : `[ ... ]` contre `[[ ... ]]`

`[ ... ]` est un synonyme de la commande `test`.

- `if [ $x -gt 4 ]`

  =

  `if test $x -gt 4`

`if : [ ... ]` contre `[[ ... ]]`

`[ ... ]` est un synonyme de la commande `test`.

- `if [ $x -gt 4 ]`

  =

  `if test $x -gt 4`

`[[ ... ]]` est plus sûr et a plus de fonctionnalités.

- `if [[ $x -gt 4 ]]`

`if` : `[ ... ]` contre `[[ ... ]]`

`[ ... ]` est un synonyme de la commande `test`.

- `if [ $x -gt 4 ]`

  =

  `if test $x -gt 4`

`[[ ... ]]` est plus sûr et a plus de fonctionnalités.

- `if [[ $x -gt 4 ]]`

`[ ... ]` existe dans plus de shells que `[[ ... ]]`

- presque tout le monde utilise `bash` ou des shells bien compatibles (donc pas grave).

`if [[ ... ]]` : fonctionnalités en plus

Utiliser les connecteurs logiques `&&` et `||`.

Ne pas trop se soucier des guillemets.

- `if [ -n "$x" ]` = `if [[ -n $x ]]`

`if [[ ... ]]` : fonctionnalités en plus

Utiliser les connecteurs logiques `&&` et `||`.

Ne pas trop se soucier des guillemets.

- `if [ -n "$x" ]` = `if [[ -n $x ]]`

Utiliser les expressions rationnelles.

```
x="hello@world"
if [[ $x =~ .*@.* ]]
then
  echo "$x est une adresse de courriel."
fi
```

`if [[ ... ]]` : fonctionnalités en plus

Utiliser les connecteurs logiques `&&` et `||`.

Ne pas trop se soucier des guillemets.

- `if [ -n "$x" ]` = `if [[ -n $x ]]`

Utiliser les expressions rationnelles.

```
x="hello@world"
if [[ $x =~ .*@.* ]]
then
  echo "$x est une adresse de courriel."
fi
```

- pas très bien documenté
- des comportement non intuitifs

`for` : traverser une liste (à la `foreach`)

Traiter chaque ligne de la sortie d'une commande :

```
for i in $(ls)
do
  echo "J'ai vu $i !"
done
```

`for` : traverser une liste (à la `foreach`)

Traiter chaque ligne de la sortie d'une commande :

```
for i in $(ls)
do
  echo "J'ai vu $i !"
done
```

Faire une boucle `for` à la Java :

```
for i in $(seq 1 10)
do
  echo "Et $i !"
done
```

Même chose que `for(i = 1; i <= 10; i++)`

`while` : boucle while

```
i=0
while [[ $i -lt 10 ]]
do
   echo "et $i"
   i=$(($i + 1))
done
```

La syntaxe pour les conditions est la même que pour `if`.

`until` : boucle while avec la condition inverse

```
i=0
until [[ $i -ge 10 ]]
do
   echo "et $i"
   i=$(($i + 1))
done
```

`while` : lire un fichier ligne par ligne

`while` : lire un fichier ligne par ligne

La commande `read` renvoie les lignes de l'entrée standard une par une.

`while` : lire un fichier ligne par ligne

La commande `read` renvoie les lignes de l'entrée standard une par une.

```
while read line
do
  echo $line
done < script1
```

`while` : lire un fichier ligne par ligne

La commande `read` renvoie les lignes de l'entrée standard une par une.

```
while read line
do
  echo $line
done < script1
```

On peut aussi utiliser les pipes :

```
cat script1 | while read line
do
  echo $line
done
```

Plan du cours

1. Le shell : l'image générale
2. Scripts shell : les bases
3. Structures de contrôle
4. Tableaux
5. Fonctions

Tableaux : déclaration et indexation

Créer un tableau en utilisant la syntaxe liste :

```
tableau=( "salade" "tomate" "oignon" )
```

Accéder aux éléments d'après leurs indexes :

```
echo ${tableau[1]}
```

Tableaux : déclaration et indexation

Créer un tableau en utilisant la syntaxe liste :

```
tableau=( "salade" "tomate" "oignon" )
```

Accéder aux éléments d'après leurs indexes :

```
echo ${tableau[1]}
```

Donner les valeurs aux éléments directement :

```
tableau[0]="salade"
tableau[1]="tomate"
tableau[2]="oignon"
```

Tableaux : longueur et traversée

Pour savoir la longueur d'un tableau :

```
echo ${#tableau[@]}
```

ou bien

```
echo ${#tableau[*]}
```

Tableaux : longueur et traversée

Pour savoir la longueur d'un tableau :

```
echo ${#tableau[@]}
```

ou bien

```
echo ${#tableau[*]}
```

Pour traverser un tableau :

```
for i in ${tableau[@]}
```

ou bien

```
for i in ${tableau[*]}
```

Plan du cours

1. Le shell : l'image générale
2. Scripts shell : les bases
3. Structures de contrôle
4. Tableaux
5. Fonctions

Fonctions : syntaxe de base

Pour déclarer une fonction :

```
function helloWorld {
  echo Hello World
}
```

Pour appeler une fonction :

```
helloWorld
```

Fonctions : syntaxe de base

Pour déclarer une fonction :

```
function helloWorld {
  echo Hello World
}
```

Pour appeler une fonction :

```
helloWorld
```

Où sont les paramètres ?

Fonctions : les paramètres

On accède aux paramètres par leur numéro :

```
function premier {
  echo "premier paramètre = $1"
}
```

Fonctions : les paramètres

On accède aux paramètres par leur numéro :

```
function premier {
  echo "premier paramètre = $1"
}
```

La commande `shift` oublie le premier argument et décale tous les numéros des autres arguments.

```
function premier {
  echo "premier = $1"
  shift
  echo "deuxième = $1"
}
```

Fonctions : accéder à tous les paramètres

`$@`	le tableau de tous les paramètres
`$*`	une chaîne de caractères qui contient tous les paramètres (aplatis)
`$#`	le nombre de paramètres

Fonctions : accéder à tous les paramètres

`$@`	le tableau de tous les paramètres
`$*`	une chaîne de caractères qui contient tous les paramètres (aplatis)
`$#`	le nombre de paramètres

Pour énumérer tous les paramètres :

```
function tous {
  for i in "$@"
  do
    echo $i
  done
}
```

Fonctions : variables locales et globales

Par défaut, toutes les variables sont globales.

Pour déclarer une variable locale, utiliser `local`.

```
x=1
y="hello"

function modif {
  x=2
  local y="bonjour"
}

modif

echo $x
echo $y
```

Fonctions : variables locales et globales

Par défaut, toutes les variables sont globales.

Pour déclarer une variable locale, utiliser `local`.

```
x=1
y="hello"

function modif {
  x=2
  local y="bonjour"
}

modif

echo $x      # affiche 2
echo $y      # affiche "hello"
```

Fonctions : valeurs de retour

Les fonctions n'ont pas de valeur de retour, mais elles ont une sortie standard.

```
function func {
  echo "hello"
}
x=$(func)
echo $x
```

On peut aussi communiquer via les variables globales.

Fonctions : codes de retour

Une fonction peut renvoyer un code de retour.

```
function plusgrand {
  if [[ $1 -gt $2 ]]
  then
    return 1      # code de retour
  else
    return 0
  fi
}
```

La variable spéciale $? contient le code de retour du dernier appel.

Un code de retour est toujours une valeur numérique.

Fonctions = scripts (presque)

Une fonction se comporte comme un script.

Les deux ont une entrée standard et une sortie standard.

Les façons d'accéder aux paramètres sont les mêmes.

Pour sortir d'un script, utiliser `exit`

- peut prendre en argument le code de retour.