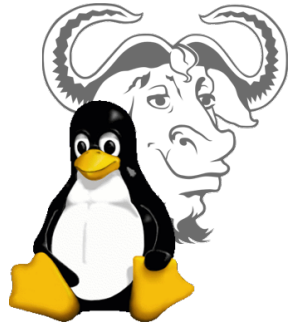
Administration d'un système GNU / Linux

06 — Scripts bash (1/2)

Anthony Labarre

上海师范大学

28 octobre 2024



Plan d'aujourd'hui

1. Alias
2. Format des scripts
3. Variables
4. Conditions
5. Boucles
6. Paramètres de scripts

Alias

Commandes plus complexes

- Il arrive que l'on écrive des combinaisons de commandes que l'on a envie de réutiliser;
- On n'a pas envie de les apprendre par cœur, et elles sont parfois longues à écrire;
- On peut définir des "synonymes" pour ces commandes;

Exemple

```
$ mise_a_jour
```

au lieu de

```
$ sudo apt update && sudo apt upgrade
```

Définition de synonymes

Ces synonymes se définissent avec la commande `alias`.

Syntaxe de `alias`

```
alias nouvelle_commande='autre(s) commande(s)'
```

Pas d'espace autour de = !

Exemple

Pour l'exemple précédent:

```
alias mise_a_jour='sudo apt update && sudo apt upgrade'
```

Ou si on a du mal à se rappeler de la commande pour changer le mot de passe:

```
alias 密码=passwd
```

`alias` et `unalias`

- `alias` sans paramètre donne la liste des synonymes;
- `unalias` supprime des synonymes:
  - `unalias` `commande` supprime `commande`;
  - `unalias -a` supprime tous les synonymes;

Les changements ne sont valables que pour la session actuelle!

- En particulier:
  - quitter la session efface les synonymes;
  - ouvrir une nouvelle session efface les synonymes (même si l'utilisateur ne change pas);
  - si vous envoyez à quelqu'un un script utilisant vos synonymes, vous devez aussi lui communiquer les synonymes;

Sauvegarde et réutilisation

- On doit enregistrer les synonymes dans un fichier pour pouvoir les réutiliser;
- Nous les placerons dans le fichier `~/.bashrc`, qui est chargé automatiquement au début de chaque session;
- Pour recharger `~/.bashrc` sans devoir se reconnecter, on utilise la commande `source ~/.bashrc`;

Commandes plus complexes

> *For almost every purpose, shell functions are preferred over aliases.*
> *http://www.gnu.org/software/bash/manual/html_node/Aliases.html#Aliases*

- Les `alias` sont utiles mais limités: on ne peut pas les "paramétrer";
- Il est donc souvent utile de pouvoir écrire des fonctions, ou des programmes plus complexes qu'on réutilisera;
- Pour y arriver, il faut connaître le langage du *shell* qu'on utilise;

Intérêt des scripts *shell*

- Les scripts *shell* servent surtout pour les tâches d'administration;
- Ils sont particulièrement utiles si l'on tire parti des commandes et programmes du sytème;
- Mais les "vrais" langages de programmation restent plus puissants;

Intérêt des scripts `bash`

- Il existe **beaucoup** d'autres *shells*;
- Chaque *shell* a son propre dialecte, mais tous ces langages sont très ressemblants;
- On n'utilisera que `bash`: ce n'est pas le "meilleur", mais c'est le plus répandu;

Wikipedia propose une comparaison des *shells* les plus utilisés.

Format des scripts

Premier script bash: Hello World!

Voici un premier script très basique affichant simplement “Hello, SHNU!”:

./src/hello_shnu.sh

```
1 #!/usr/bin/env bash
2 echo "Hello, SHNU!"  # affiche "Hello, SHNU!"
```

On nomme les fichiers avec l’extension `.sh` — ici: `hello_shnu.sh`
Pour lancer le script:

1. `hello_shnu.sh` doit être exécutable;
2. on tape `./hello_shnu.sh` dans le terminal;

La première ligne *#!/usr/bin/env bash* dit à GNU qu’il faut utiliser bash pour exécuter le script.

Commentaires

- Les commentaires s'écrivent comme en Python: tout ce qui suit un `#` est ignoré (sauf la première ligne spéciale);
- On ne peut pas commenter plusieurs lignes à la fois (pas d'équivalent de `'''Très \n longue \n chaîne.'''` ou des `/* commentaires en C(++) */`);

Variables

Deuxième script bash: Hello l'utilisateur!

Personnalisons notre premier script pour qu'il utilise notre nom d'utilisateur:

./src/hello_shnu_2.sh

```
1 #!/usr/bin/env bash
2 echo "Hello, $USER"
```

- La *variable* `USER` est une **variable d'environnement** de `bash`: elle contient le nom de l'utilisateur actuellement connecté;
- Le symbole `$` permet de récupérer la valeur de cette variable pour l'afficher;
- `bash` possède de nombreuses autres variables d'environnement, dont `printenv` fournit la liste (avec leurs valeurs);

Variables et affectations

- Pour affecter une valeur à une variable, on écrit: `nomvariable=valeur`

N'écrivez pas d'espaces autour du =! Sinon, vous aurez l'erreur "command not found".

- Pour obtenir la **valeur** d'une variable, on utilise l'opérateur $ (exemple: `$nomvariable`).
- On peut supprimer une variable avec la commande `unset nomvariable;`

Manipuler le résultat d'une commande

Modifions encore notre script pour qu'il enregistre et affiche la date d'aujourd'hui:

./src/hello_shnu_3.sh

```
1 #!/usr/bin/env bash
2 echo "Hello, $USER"
3 aujourdhui=$(date)
4 echo "La date d'aujourd'hui est $aujourdhui."
```

- L'opérateur $ permet aussi d'évaluer le résultat d'une commande.
- Attention aux parenthèses!
  - `$date` est la valeur de la variable `date`;
  - `$(date)` est le résultat de la commande `date`;

Affichage

Deux commandes permettent d'afficher des valeurs: `echo` et `printf`.

- ☺ `echo` est plus intuitive;
- ☹ `printf` fonctionne comme le `printf` du C;

Pas de types

Les variables en bash n'ont pas de type; on doit donc faire attention quand on effectue des calculs:

./src/hello_shnu_4.sh

```
1  #!/usr/bin/env bash
2  echo "Hello, $USER"
3  aujourdhui=$(date)
4  echo "La date d'aujourd'hui est $aujourdhui."
5  x=1+1
6  echo "Le résultat de 1+1 est $x, si l'on déclare x=1+1"
7  y=$((1+1))
8  echo "Le résultat de 1+1 est $y, si l'on déclare y=\$((1+1))"
9  z=$[1+1]
10 echo "Et si l'on effectue \$[1+1]? on obtient $z"
11 printf "Et si l'on effectue \$(1+1)? Le résultat est "
12 echo "$(1+1)"
```

Récapitulatif sur les $

Attention aux trois utilisations possibles de $:

1. `$variable`: donne la valeur d'une variable;
2. `$(commande)`: donne le résultat d'une commande;
3. `$[calcul]` ou `$((calcul))`: donne le résultat d'un calcul **sur des entiers**;

`bash` ne gère pas les nombres réels: `x=$((1+1.5))` provoque une erreur! Il faut utiliser les programmes `bc` ou `dc`.

Pas de types

Attention aux pièges causés par l'absence de types:

```
$ x=1
$ x=$((x+1))
$ echo $x
2   # ok, normal
$ x+=1
$ echo $x
21  # hein?
```

Par défaut, les variables de `bash` sont des chaînes, sur lesquelles on peut parfois faire des calculs.

Pas de types . . . mais . . . (declare)

On peut quand même tricher un peu pour les types:

```
$ declare -i x=1  # force x à être entier
$ x+=1
$ echo $x
2  # ouf, les maths fonctionnent
$ declare -r pi=3.14159  # force pi à être une constante
$ pi+=5
bash: pi: readonly variable
```

La commande `typeset` est un synonyme de `declare` et fonctionne aussi sous d'autres *shells* (par exemple `ksh`).

Opérateurs et formes condensées

- Les opérations que vous connaissez sont disponibles (+, -, *, /, %, ...);
- Les formes condensées aussi (+=, -=, *=, /=, ...);
- Attention encore une fois aux espaces!
  - ✓ `x+=1` fonctionne (attention au résultat ...);
  - × `x += 1` déclenche une erreur;

Les autres formes condensées ne fonctionnent pas toujours ... pour être sûr de ne pas avoir de problème, utilisez `let`. Par exemple:

```
let x*=2
```

Conditions

Instructions conditionnelles (`if`, `elif`, `else`)

Très semblables à Python, mais:

1. les tests sont encadrés par des `[[ ]]` **avec des espaces**;
2. les blocs `if` et `elif` commencent par `then`;
3. la fin d'un `if` doit être marquée par un `fi`;

En Python

```
if test_1:
    # bloc if
elif test_2:
    # bloc elif
else:
    # bloc else
```

En bash

```
if [[ test_1 ]]
then
    # bloc if
elif [[ test_2 ]]
then
    # bloc elif
else
    # bloc else
fi
```

Comparaisons

Les comparaisons renvoient 0 ou 1 (au lieu de `False` ou `True`) et se font à l'aide des opérateurs suivants:

Comparaison	Opérateur
égalité	`$a == $b`
différence	`$a != $b`
est inférieur	`$a < $b`
est inférieur ou égal	`$a <= $b`
est supérieur	`$a > $b`
est supérieur ou égal	`$a >= $b`

On peut combiner les conditions avec les opérateurs logiques:

- `and`, qu'on écrit `&&`;
- `or`, qu'on écrit `||`;
- `not`, qu'on écrit `!`;

N'oubliez pas d'encadrer les espaces autour des opérateurs!

Boucles

Boucles `for`

Les `for` fonctionnent comme en C et en Python, mais il faut préciser le début du bloc (et la fin) avec `do` (et `done`).

En Python

```
for x in iterable:
    # bloc for
# suite
```

En Python

```
for i in range(1, n+1):
    # bloc for
# suite
```

En C

```
for (int i=0; i<n; i++) {
    // bloc for
}
```

En bash

```
for x in iterable
do
    # bloc for
done
```

En bash

```
for i in $(seq n)
do
    # bloc for
done
```

En bash

```
for ((i=0; i<n; i++))
do
    # bloc for
done
```

Boucles `for`

La syntaxe des `for` est très souple:

```
$ for lettre in A B C D E; do printf $lettre" "; done; echo
A B C D E
```

```
$ for lettre in {A..E}; do printf $lettre" "; done; echo
A B C D E
```

```
$ for chiffre in {1..5}; do printf $chiffre" "; done; echo
1 2 3 4 5
```

```
$ for texte in *.txt; do echo $texte; done;
# équivalent de ls *.txt
```

Boucles `while`

Les `while` fonctionnent comme en C et en Python, mais il faut préciser le début du bloc (et la fin) avec `do` (et `done`).

En Python

```
while condition:
    # bloc while
# suite
```

En bash

```
while [[ condition ]]
do
    # bloc while
done
```

Paramètres de scripts

Paramètres de scripts

- Les scripts peuvent prendre des paramètres, accessibles par leur position et $:
  - `$0` est le nom du script;
  - `$1` est le premier paramètre;
  - `$2` est le deuxième paramètre;
  - ...
- Attention: après `$9`, on doit utiliser des accolades (donc `${10}`, `${11}`, etc.)
- Il y a aussi quelques variables spéciales:
  - `$#` est le nombre de paramètres du script;
  - `$*` et `$@` contiennent tous les paramètres du script;

Exemple: saluer un utilisateur existant (ou non)

Le script ci-dessous prend comme seul paramètre un nom d'utilisateur:

./src/param_script.sh

```
1 #!/usr/bin/env bash
2 user=$1
3 echo "Bonjour $user!"
4 if [[ $(grep ^$user /etc/passwd | cut -f1 -d:) == $user ]]
5 then
6     echo "Je vous ai trouvé dans /etc/passwd ."
7 else
8     echo "Je ne vous ai pas trouvé dans /etc/passwd ."
9 fi
```