

Systemes UNIX. Shell comme langage de programmation

Sergey Kirgizov

Plan de cours

CM 1 — Introduction, Histoire d'UNIX

CM 2 — Architecture

CM 3 — Fichiers, arbres, inodes

CM 4-5 — Écosystème du shell, redirections, processus

CM 6 — Droits

CM 7 — Expressions régulières, grep et sed avancées

CM 8 — Shell comme langage de programmation, ⇐ on est là

CM 9 — Traitement de texte : AWK et sed

CM 10 — Réseau et UNIX

CM 11 – Administration : utilisateurs, logiciels, processus

CM 12 — Surprise !

Merci Benoît !

Ce cours est fait en partie à partir du cours de Benoît Darties.
<https://benoit.darties.fr/>

Shell et programmation

Le shell peut jouer le rôle d'interface de programmation

Elements usuels de la programmation impérative

- Variables : Affectation et lecture
- Opérations : arithmétiques, concaténations, ...
- Tests conditionnels (if...then...else)
- Boucles déterministes (for)
- Boucles non déterministes (while)

Ecriture des instructions

- Les instructions d'un programme shell peuvent être entrées directement dans l'interface du shell
- Ou peuvent être écrites dans un fichier, qui sera ensuite lu et exécuté : fichier script

Exécution des commandes dans un fichier script

Fichier script

Objet et intérêt d'un fichier script :

- Stocker les commandes à exécuter dans un fichier
- Commandes exécutées séquentiellement
- Pouvoir exécuter une ou plusieurs fois une suite d'instructions qui peut s'avérer longue
- Ne pas avoir à retaper à chaque fois les commandes
- Continuer à bénéficier de la puissance/caractéristiques du shell
- Pouvoir programmer l'exécution de ces commandes à différents moments de la journée, ou selon certains événements

Exécution des commandes dans un fichier script

Quelques éléments préliminaires

- généralement : extension `.sh`
- Lignes précédées d'un `#` (hash / croisillon) : commentaires
- Exécution comme un programme classique
- Fichier exécutable : s'assurer des droits d'exécution (+x)
- Commandes externes (programmes) et internes (du shell)
- Différents shell = différentes syntaxes
 - Besoin de spécifier quel shell utiliser
 - La première ligne du fichier doit préciser le shell
Notation : `#!<chemin du shell>`
exemple : `#!/bin/bash`

Un premier script hello world

Fichier helloworld.sh

```
1 #!/bin/bash
2
3 # author : Benoit Darties
4 # date : 2015-08-05
5 # description : simple hello world script
6
7 echo "hello world !!!"
```

Ajout des droits d'exécution s'ils n'y étaient pas déjà

```
1 benoit$ chmod +x helloworld.sh
```

Exécution :

```
1 benoit$ ./helloworld.sh
2 hello world !!!
```

Itérations déterministes

Boucles `for ... do... done`

- Répétition (itérations) d'une ou plusieurs d'instructions
- A chaque itération : mêmes instructions
- Nécessité de déclarer un compteur d'itération
- Seule la valeur du compteur d'itérations va changer à chaque tour (généralement incrémentation de 1 en 1)
- Itérations déterministes : on sait combien de tours on va faire
 - Nécessité de préciser un domaine : ensemble des valeurs que va prendre successivement le compteur de boucle à chaque tour
- Délimiteurs de début (`do`) et de fin (`done`) de boucle

Itérations déterministes

Utilisation de `for ... do ... done`

- syntaxe :

```
for (( init; condition; incrementation ))  
do  
    instruction1  
    instruction2  
    ...  
done
```

- les espaces sont importants !
- *init* : initiation du compteur d'itération
- *condition* : on boucle tant que la condition est vérifiée
- *incrémentation* : comment le compteur d'itération évolue
- Autre notation : `for variable in {1..10} do`

Itérations déterministes

Contenu du fichier script boucleFor.sh

```
1  #!/bin/bash
2
3  for (( i=0; i<10; i++ ))
4  do
5      echo ceci est le $i tour de boucle
6  done
```

Exécution du fichier script boucleFor.sh

```
1  Galactica:script benoit$ ./boucleFor.sh
2  ceci est le 0 eme tour de boucle
3  fin de ce tour de boucle
4  ceci est le 1 eme tour de boucle
5  fin de ce tour de boucle
6  ceci est le 2 eme tour de boucle
7  fin de ce tour de boucle
8  ceci est le 3 eme tour de boucle
9  fin de ce tour de boucle
10 ceci est le 4 eme tour de boucle
11 fin de ce tour de boucle
```

Itérations déterministes

For peut également itérer sur autre chose que des nombres !

- A la place de nombres, itérations sur un ensemble de mots
- syntaxe :

```
for compteur in mot1 mot2 mot3 ...
do
    instruction1
    instruction2
    ...
done
```

- Utilisation quasi-similaire
- Mais à chaque tour de boucle, le compteur prend la valeur du mot suivant dans la liste fournie en paramètres

Itérations déterministes

Contenu du fichier script boucleFor2.sh

```
1  #!/bin/bash
2
3  for i in Bulbizarre Salameche Carapuce Pikachu Raichu Ronflex
4  do
5      echo "dans ce tour de boucle, i vaut $i"
6  done
```

Exécution du fichier script boucleFor2.sh

```
1  Galactica:script benoit$ ./boucleFor2.sh
2  dans ce tour de boucle, i vaut Bulbizarre
3  dans ce tour de boucle, i vaut Salameche
4  dans ce tour de boucle, i vaut Carapuce
5  dans ce tour de boucle, i vaut Pikachu
6  dans ce tour de boucle, i vaut Raichu
7  dans ce tour de boucle, i vaut Ronflex
```

Rappels sur les variables

- Comment créer une variable : lui affecter une valeur
- Comment lui affecter une valeur : opérateur =
- Comment lire une valeur : \$NomVariable

Question

Quel est le résultat de la suite d'instructions suivantes :

```
$ A=5  
$ B=6  
$ C=$A  
$ D=$C+$B+1  
$ echo $D
```

Rappels sur les variables

- Comment créer une variable : lui affecter une valeur
- Comment lui affecter une valeur : opérateur =
- Comment lire une valeur : \$NomVariable

Question

Quel est le résultat de la suite d'instructions suivantes :

```
$ A=5  
$ B=6  
$ C=$A  
$ D=$C+$B+1  
$ echo $D
```

Réponse

```
$ echo $D  
5+6+1
```

Operatiions sur les variables

Le shell n'interprete pas directement les operations mathematique

- Comportement "bete" : pas de calculs effectues
- Les operateurs sont traites comme des caracteres standards

Solution 1 (commune) : utiliser la commande `let` :

```
1 benoit$ A=5
2 benoit$ B=6
3 benoit$ let C=2*A+B-2
4 benoit$ echo $C
5 14
```

Solution 2 (specifique bash) : utiliser les doubles parentheses

```
1 benoit$ A=5
2 benoit$ B=6
3 benoit$ (( C=2*A+B-2 ))
4 benoit$ echo $C
5 14
```

Opération sur les variables

Liste des principaux opérateurs acceptés par `let` ou `((...))`

Signe	Opération	Signe	Opération	Signe	Opération
+	Addition	<=	Inférieur ou égal	&&	ET logique
-	Soustraction	>=	Supérieur ou égal		OU logique
*	Multiplication	>	Strictement supérieur	&	Et bit à bit
/	Quotient	<	Strictement inférieur		OU bit à bit
%	Modulo	==	Egalité	^	OU exclusif bit à bit
=	Affectation	!=	différent de	<<	Décalage de bits à gauche
(.)	Priorité des opérations	!	Négation	>>	Décalage de bits à droite

Opérateurs arithmétiques
standard

Opérateurs de comparaison
(retourne un booléen)

Opérateurs booléens
et bit à bit

Exécution conditionnelles

Condition `if ... then... [else ...] fi`

- exécute une ou plusieurs instructions selon une condition
- syntaxe :

```
if condition-a-tester
then
  instruction1 si condition vraie
  instruction2 si condition vraie
  ...
else
  instruction1 si condition fausse
  instruction2 si condition fausse
  ...
fi
```

- Partie `else` optionnelle

Exécution conditionnelles

La définition d'une condition n'est pas si simple !

- *condition-a-tester* n'est pas une condition classique
- Il s'agit du **code retour** d'une commande, d'une expression
- Rappel : code retour de la dernière commande visible dans \$?
- Si le code retour est 0, la condition est considérée vraie
- Si le code retour est différent 0, la condition est fausse

Multiples solutions pour les conditions

- On peut réutiliser la commande `let`
- On peut réutiliser les doubles parenthèses (en bash)
- On peut utiliser une commande spécifique : `test`
 - Commande la plus appropriée
 - teste une expression arithmétique, logique, un fichier, ...
 - renvoie 0 si le test est ok, et 1 sinon

Exécution conditionnelles : exemple avec `let`

Pour être sûr, on vérifie simplement la valeur du code retour :

```
1 Galactica:script benoit$ let 5*6==12
2 Galactica:script benoit$ echo $?
3 1
4
5 Galactica:script benoit$ let 5*6>12
6 Galactica:script benoit$ echo $?
7 0
```

Au sein du fichier script `testIf_bracket.sh`

```
1 #!/bin/bash
2
3 A=5
4 if (( $A==6 ))
5 then
6     echo "A est bien egal à 6"
7 else
8     echo "A ne vaut pas 6. Il est égal à $A"
9 fi
```

Exécution du fichier script `testIf_bracket.sh`

```
1 Galactica:script benoit$ ./testIf_let.sh
2 A ne vaut pas 6. Il est égal à 5
```

Exécution conditionnelles : exemple avec ((...))

Pour être sûr, on vérifie simplement la valeur du code retour :

```
1 Galactica:script benoit$ (( 5+6 == 12 ))
2 Galactica:script benoit$ echo $?
3 1
4
5 Galactica:script benoit$ (( 5*6 > 12 ))
6 Galactica:script benoit$ echo $?
7 0
```

Au sein du fichier script testIf_bracket.sh

```
1 #!/bin/bash
2
3 A=5
4 if (( $A==6 ))
5 then
6     echo "A est bien egal à 6"
7 else
8     echo "A ne vaut pas 6. Il est egal à $A"
9 fi
```

Exécution du fichier script testIf_bracket.sh

```
1 Galactica:script benoit$ ./testIf_bracket.sh
2 A ne vaut pas 6. Il est egal à 5
```

Exécution conditionnelles : la commande `test`

la commande `test`

- Permet de faire tout un tas de tests différents :
 - Vérifie si des (in)équations sont justes
 - Fait des tests sur le status ou l'existence d'un fichier
 - Analyse et études de chaînes de caractères
- Ces tests peuvent être utilisés comme condition dans un `if`
- Syntaxe :
 - 1 seule opérande (exemple : est-ce qu'un fichier existe ?)
`test operateur operande`
 - 2 opérandes (exemple : est-ce que A est supérieur à 50 ?)
`test operande1 operateur operande2`
- Les opérandes sont les éléments à tester
- Difficulté : trouver le bon opérateur par rapport au test à faire

Exécution conditionnelles : la commande `test`

Comparaisons arithmétiques : opérateurs à 2 opérandes

Opérateur	Signification
<code>-eq</code>	Valeurs algébriquement identiques
<code>-ne</code>	Valeurs algébriquement différentes
<code>-gt</code>	La première valeur est strictement supérieure
<code>-ge</code>	La première valeur est supérieure ou égale
<code>-lt</code>	La première valeur est strictement inférieure
<code>-le</code>	La première valeur est inférieure ou égale

Tests sur fichiers : opérateurs à 1 opérande

Opérateur	Signification
-b	Fichier spécial bloc
-c	Fichier spécial caractères
-d	Répertoire
-e	Existe
-f	Fichier régulier
-g	Drapeau group ID actif
-h	Lien symbolique

Opérateur	Signification
-k	Drapeau Sticky Bit actif
-p	Tube nommé
-r	Droits de lecture
-s	Taille supérieure à 0
-x	Droits d'exécution
-z	Taille égale à 0
-O	Propriétaire = utilisateur

Tests sur fichiers : opérateurs à 2 opérandes

Opérateur	Signification
-nt	Le premier fichier existe et est plus récent que le second
-ot	Le premier fichier existe et est plus ancien que le second
-ef	Les deux entrées réfèrent au même fichier (même inode)

Tests sur les chaînes de caractères : opérateurs à 1 opérande

Opérateur	Signification
-z	Chaîne de longueur nulle
-n	Chaîne de longueur non nulle

Tests sur les chaînes de caractères : opérateurs à 2 opérandes

Opérateur	Signification
=	Chaînes identiques
!=	Chaînes différentes
'>'	La première chaîne a une valeur supérieure (valeur ASCII)
'<'	La première chaîne a une valeur inférieure (valeur ASCII)

Exemples d'utilisation de `test` dans une condition

```
1 A=7
2 if test $A -gt 6
3 then
4     echo "A est strictement supérieur à 6"
5 fi
6
7 if test $A -eq 6
8 then
9     echo "A est égal à 6"
10 fi
11
12 if test -e fichier.txt
13 then
14     echo le fichier fichier.txt existe
15 fi
16
17 if test -d fichierTest
18 then
19     echo le fichier fichierTest existe
20 fi
21
22 if test "bonjour" '<' "bonsoir"
23 then
24     echo "bonjour" précède "bonsoir" \ (ordre lexicographique\ )
25 fi
```

Effectuer plusieurs tests d'affilée

Utilisation des conjonctions

- La conjonction `&&` : se traduit par 'et'
- Se place entre deux commandes
- Impose que le code retour de la première commande doit être 0 pour que la seconde soit exécutée
- L'expression entière retournera 0 si les deux commandes ont été correctement exécutées, et 0 sinon
- Permet de faire deux tests consécutifs

```
1 if test $A -gt 6 && test $A -lt 10 && test $A -ne 8
2 then
3     echo "A est compris entre 6 et 10 mais n'est pas 8"
4 fi
```

Exemple sur une lcd : protection des commandes

```
1 test -d monRep && mv fichier.txt monRep
```

Effectuer plusieurs tests d'affilée

d'autres moyens d'exprimer la conjonction

Certaines options de commandes permettent d'intégrer plusieurs expressions conjonctives à la suite

Au sein des doubles parenthèses :

```
1 (( A < 30 )) && (( A > 15 ))  
2 # est la meme chose que  
3 (( (A < 30 ) && ( A > 15) ))
```

Avec la commande `test` : option `-a` (and)

```
1 test $A -lt 30 && test $A -gt 15  
2 # est la meme chose que  
3 test $A -lt 30 -a $A -gt 15
```

Effectuer plusieurs tests d'affilée

Utilisation des disjonctions

- La conjonction `||` : se traduit par 'ou'
- Se place entre deux commandes
- Impose que le code retour de la première commande doit être 1 pour que la seconde soit exécutée
- L'expression entière retournera 0 si au moins une des deux commandes a été correctement exécutée, et 0 sinon
- Permet de faire deux tests consécutifs

```
1 if test $A -gt 6 || test $A -lt 10 || test $A -ne 8
2 then
3     echo "A est plus grand que 6, ou plus petit que 10, ou n'est pas égale à 8"
4 fi
```

Exemple sur une lcd : protection des commandes

```
1 test -e monRep || mkdir monRep
```

Effectuer plusieurs tests d'affilée

d'autres moyens d'exprimer la disjonction

Certaines options de commandes permettent d'intégrer plusieurs expressions disjonctives à la suite

Au sein des doubles parenthèses :

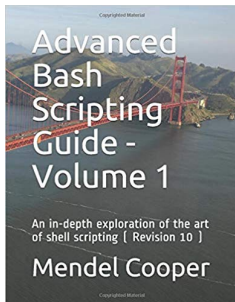
```
1 (( A < 30 )) || (( A > 15 ))
2 # est la meme chose que
3 (( (A < 30) || (A > 15) ))
```

Avec la commande `test` : option `-o` (or)

```
1 test $A -lt 30 || test $A -gt 15
2 # est la meme chose que
3 test $A -lt 30 -o $A -gt 15
```

Remarques

- ▶ Les redirections s'appliquent aux blocs : for, while, ..., etc...
- ▶ Plus d'info dans le livre "ABS" :
 - ▶ tableau,
 - ▶ fonctions
 - ▶ etc...



Advanced Bash-Scripting Guide par Mendel Cooper
<https://www.tldp.org/LDP/abs/html/>

Questions ?