

Scripts avec bash

Guillaume Chanel - guillaume.chanel@hesge.ch

Florent Glück - florent.gluck@hesge.ch

March 27, 2025

ISC - HEPIA

Remerciements

Remerciements à Jean-Luc Falcone

Scripts

Problème réel

- Aggréger des listes d'emails
- Supprimer les doublons
- Les adresses peuvent être écrites avec des casses différentes

Solution possible

À l'aide des commandes shell suivantes :

cat	Concatène plusieurs fichiers
tr	Transforme les caractères
sort	Trie les lignes
uniq	Rend unique les lignes répétées

Exemple :

```
$ cat mail1.txt mail2.txt | tr '[A-Z]' '[a-z]' | sort | uniq > result.txt
```

Qu'est ce qu'un script ?

- Un script est un fichier texte contenant une séquence de commandes shell
- Un script peut être exécuté comme une commande ou un programme

Exécuter un script

Deux manières d'exécuter un script :

- Dans le shell **actuel**, avec la commande **source** :

```
$ source backup.sh
```

- Dans un **nouveau** shell (processus)

- passé en argument à l'exécution d'un shell :

```
$ bash backup.sh
```

- en exécutant le script si celui-ci possède le bit d'exécution :

```
$ ./backup.sh
```

#! Hash bang - spécifier l'interpréteur

- On peut préciser l'interpréteur du script à exécuter avec les caractères `#!` au début du script (prononcer *hash-bang*, *she-bang* ou *sha-bang*)
- Ceci s'applique :
 - à **tout** interpréteur, e.g. bash, zsh, python, perl, lua, ruby, etc.
 - **uniquement** lorsque le script est exécuté directement, e.g. :

```
$ ./backup.sh
```


#! Hash bang : exemples

Script bash, `uid.sh` :

```
#!/bin/bash  
echo $UID
```

Script python, `uid.py` :

```
#!/usr/bin/python3  
import os  
print(os.getuid())
```

Script ruby, `uid.rb` :

```
#!/usr/bin/ruby  
puts Process.uid
```

Exercice : premier script

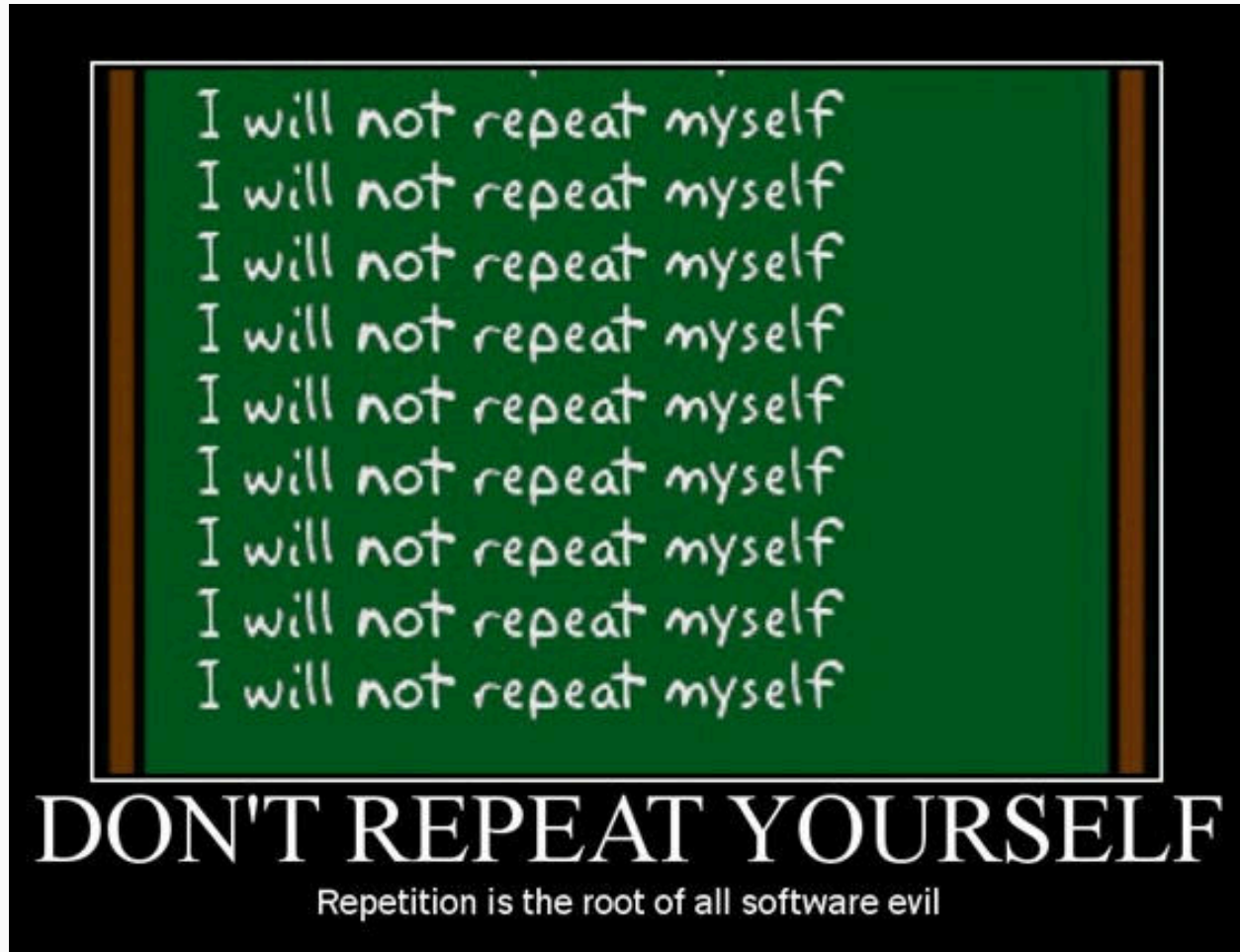
Écrire un script qui :

- Supprime le contenu du dossier `data/backup`
- Copie tous les fichiers `*.txt` et `*.dat` du dossier `data` vers le dossier de backup (voir architecture `git`)
- Change la permission des fichiers copiés pour ne donner QUE les droits de lecture à l'utilisateur et au groupe

Solution

backup1.sh :

```
#!/bin/bash
# Efface les anciens backups
rm -f ~/data/backup/*
# Backup les fichiers .txt et .dat
cp ~/data/*.txt ~/data/*.dat ~/data/backup/
chmod 440 ~/data/backup/*
```



Exécuter un script (2/2)

- Marquer `backup1.sh` comme **exécutable** pour l'utilisateur :

```
$ chmod u+x backup1.sh
```

- Exécuter `backup1.sh` dans un nouveau processus :

```
$ ./backup1.sh
```

Variables

- On assigne des variables avec le symbole =
- On accède à la valeur de la variable avec le symbole \$
- Exemple :

```
$ message="Je suis ... "  
$ echo message  
$ echo $message  
$ echo ${message}  
$ echo "${message}Luke"
```

Attention : les variables déclarées ainsi sont **globales**, même si elles sont déclarées dans le corps d'une fonction ou d'une boucle !

Exercice : variables

Modifier le script précédent pour éviter les répétitions

Solution

backup2.sh :

```
#!/bin/bash
DATA=~/.data
BACKUP=$DATA/backup
#Efface les anciens backups
rm -f $BACKUP/*
#Backup les fichiers .txt et .dat
cp $DATA/*.txt $DATA/*.dat $BACKUP
chmod 440 $BACKUP/*
```



Code de retour

Toute commande retourne un code entier positif avec pour valeur :

- **0** : si la commande c'est terminée avec succès (équivalent à VRAI)
- **code d'erreur** > 0 : si la commande a rencontré une erreur (équivalent à FAUX)

La variable `$?` contient le code de retour de la dernière commande :

```
$ ls *.sh  
$ RESULT=$?
```

 Ne pas **confondre** le **code de retour** d'une commande avec sa **sortie standard**, par exemple dans les substitutions !

Code de retour et enchaînement de commandes

Les listes ET/OU permettent d'enchaîner les commandes :

- Ici, `process` est exécuté, uniquement si `ls` retourne un code de 0 :

```
$ ls fichierquiexiste && process fichierquiexiste
```

- Ici, `echo` est exécuté, uniquement si `ls` retourne un code différent de 0 (donc une erreur) :

```
$ ls fichierquiNexistePAS || echo "Erreur le fichier n'existe pas" 1>&2
```

Conditions (if then)

Il existe des structures de contrôle `if` comme dans la plupart des langages :

```
if [ EXPRESSION ]; then  
    cmd1  
    cmd2  
    ...  
fi
```

Conditions (if then else)

```
if [ EXPRESSION ]; then  
    cmd1  
    cmd2  
    ...  
else  
    cmd10  
    cmd11  
    ...  
fi
```

Conditions (if then elif else)

```
if [ EXPRESSION ]; then
    cmd1
    cmd2
    ...
elif [ EXPRESSION ]; then
    cmd5
    cmd6
    ...
else
    cmd10
    cmd11
    ...
fi
```

Conditions (case)

```
case $f in
  *.tar.gz) tar xvzf $f ;;
  *.bz2)    bunzip2 $f ;;
  *.rar)    unrar x $f ;;
  *.gz)     gunzip $f ;;
  *.tgz)    tar xvzf $f ;;
  *.zip)    unzip $f ;;
  *)        echo "'$f': unsupported format" ;;
esac
```

- On peut tester une expression avec la commande `test`, généralement utilisée avec des crochets
- Les deux expressions ci-dessous sont équivalentes :

```
test EXPRESSION  
[ EXPRESSION ]
```

Tests sur les fichiers

-e FILE	Le fichier FILE existe
-f FILE	Le fichier FILE existe et est un fichier régulier
-d FILE	Le fichier FILE existe et est un répertoire
-x FILE	Le fichier FILE existe et est exécutable
-w FILE	Le fichier FILE existe et on peut y écrire
FILE1 -nt FILE2	Le fichier FILE1 est plus récent que le fichier FILE2
FILE1 -ot FILE2	Le fichier FILE1 est plus ancien que le fichier FILE2

Exercice : if

Modifiez le script précédent pour que :

- Si le répertoire `backup` existe, son contenu soit effacé (comme avant)
- Si le répertoire `backup` n'existe pas, il soit créé

Assurez-vous que votre script fonctionne correctement dans le cas où le nom de répertoire est le même que celui d'un fichier

Solution

backup3.sh :

```
...  
DATA=~ /data  
BACKUP=$DATA/backup  
# Si le repertoire existe  
if [ -d $BACKUP ]; then  
    # Efface les anciens backups  
    rm -f $BACKUP/*  
else  
    # Cree le repertoire  
    mkdir -p $BACKUP  
fi  
...
```

Autres expressions de test

! EXPRESSION	Retourne vrai si EXPRESSION est fausse
EXPR1 -a EXPR2	Retourne vrai si EXPR1 et EXPR2 sont vraies
EXPR1 -o EXPR2	Retourne vrai si EXPR1 ou EXPR2 sont vraies
STRING1 = STRING2	Retourne vrai si STRING1 est égal à STRING2
STRING1 != STRING2	Retourne vrai si STRING1 n'est pas égal à STRING2

Boucles (for) (1/2)

- Itère sur chaque éléments d'une liste
- Les éléments sont séparés par des espaces ou des retours à la ligne
- Assigne une variable utilisable à l'intérieur de la boucle

```
for VAR in LIST; do
    CMD1 $VAR
    CMD2 $VAR
    ...
done
```

Boucles (for) (2/2)

Une liste peut être :

- Des éléments séparés par des espaces :

```
for ANIMAL in chat chien oiseau
```

- Un intervalle :

```
for I in {5..42}
```

- Une expression représentant un ensemble de fichiers :

```
for FILE in *.txt
```

Exercice : for

Modifiez le script précédent avec une boucle `for` qui pour chaque fichier :

- Copie le fichier
- Affiche un message d'information de copie

Le mode des fichiers doit ensuite être changé pour tous les fichiers en une fois

Solution

backup4.sh :

```
...  
# Backup les fichiers .txt et .dat  
for f in $DATA/*.txt $DATA/*.dat; do  
    echo "Copie de $f"  
    cp $f $BACKUP  
done  
chmod 440 $BACKUP/*  
...
```

Comment effectuer le changement de mode dans la boucle ? (voir commande `basename`)

Substitution de commande

- Il est souvent intéressant de récupérer le résultat d'une commande
- Deux formes de syntaxes existent pour cela :
 - `CMD OPTIONS ARGUMENTS`
 - `$(CMD OPTIONS ARGUMENTS)`
- Exemple :

```
$ echo "Mon nom d'utilisateur est: `whoami`"  
$ echo "Mon nom d'utilisateur est: $(whoami)"  
$ a=`ls *.sh`  
$ a=$(ls *.sh)
```


Arguments

Il est possible de récupérer les arguments passés à un script en utilisant des **variables prédéfinies** :

\$0	Contient le nom du script tel qu'il a été appelé (e.g. /home/jane/script.sh ou ./script.sh)
\$1, \$2, \$3, etc.	\$1 indique le premier argument passé au script, \$2 le deuxième, etc.
\$@	La liste des arguments (sans le nom du script)
\$#	Le nombre d'arguments passés au script

Exercice : arguments

Modifiez le script précédent pour que les répertoires source et destination puissent être passés en argument au script

Solution

backup5.sh :

```
#!/bin/bash  
DATA=$1  
BACKUP=$2  
...
```

Utilisation :

```
$ ./backup5.sh ~/data ~/data/backup
```

Fonction (1/2)

Une fonction se définit de la manière suivante :

```
function show {  
    # $1 et $2 sont les 1er et 2eme arguments passés à la fonction  
    echo "$1 ----- $2"  
}  
show foo bar # appel à la fonction
```

On peut accéder aux arguments d'une fonction avec \$1, \$2, \$@, etc.
Ceux-ci *override* les arguments du script

Fonction (2/2)

- Une fonction n'a **pas de valeur de retour** !
- Cependant, il est possible de retourner des valeurs en utilisant :
 - des variables globales
 - la sortie standard en combinaison avec la substitution de commande :

```
function show {  
    # $1 et $2 sont les 1er et 2eme arguments passés à la fonction  
    echo "$1 ----- $2"  
}  
  
RET=$(show foo bar) # appel à la fonction  
echo $RET
```

Exercice : fonctions

- Dans le script précédent, remplacez les trois opérations du backup par un appel à une fonction avec arguments
- Incluez également le changement de mode dans la boucle

Solution

backup6.sh :

```
function copy {  
    echo "Copie de $1"  
    cp $1 $2  
    chmod 440 $2/${basename $1}  
}  
...  
for f in $DATA/*.txt $DATA/*.dat; do  
    copy $f $BACKUP # appel à la fonction  
done  
...
```

- Opérations arithmétiques
- Tableaux
- Variables définies par défaut
- Règles pour les guillemets (*quotation*)

Configurer bash

Fichiers de configurations

bash utilise plusieurs fichiers cachés à la racine du *home* pour obtenir une configuration :

.bashrc	Exécuté à chaque nouveau shell (non-login)
----------------	--

.bash_profile	Shell de login uniquement (<i>login shell</i>)
----------------------	--

bash_logout	Exécuté lorsqu'on quitte un shell de login
--------------------	--

Shell de login (.bash_profile)

- Les shells de login sont ceux pour lesquels **il faut s'authentifier** :
 - connexion à distance
 - console virtuelle (tty)
- Un terminal lancé depuis l'interface graphique n'est pas un shell de login
- La pratique actuelle est d'appeler¹ .bashrc depuis .bash_profile

.bash_profile :

```
...  
if [ -f ~/.bashrc ]; then  
    source ~/.bashrc  
fi  
...
```

¹automatique sous Mac

Configuration du shell (.bashrc)

Habituellement, on utilise `.bashrc` pour définir les éléments suivants :

- Variables d'environnement
- Alias
- Permissions par défaut
- Comportement de bash

Variables d'environnement (export)

- Variables accessibles aux processus
- Tous les langages de programmation (sérieux) permettent d'y accéder
- Les processus enfants héritent des variables d'environnement du parent
- On accède à leur valeur avec un `$` (comme une variable normale)
- On les définit avec le mot clé `export` :

```
$ export VAR1="hello world"  
$ echo $VAR
```

Exercice : variables d'environnement

1. Créez un variable normale
2. Exécutez un nouveau shell
3. Vérifiez l'existence de la variable créée précédemment
4. Effectuez les trois opérations ci-dessus mais avec une variable d'environnement
5. Qu'en conclure ?

Variables d'environnement standard

\$PATH	Chemin où trouver les fichiers exécutables
\$HOME	Répertoire <i>home</i> de l'utilisateur
\$LANG	Langue par défaut
\$DISPLAY	Serveur graphique
\$PS1	<i>Prompt</i> (invite) du shell
\$HOSTNAME	Nom de la machine

Chemin (PATH)

- Le chemin où trouver les fichiers exécutables (programmes/scripts)
- La variable d'environnement `$PATH` définit le chemin sous forme d'un ensemble de répertoires
- Si l'exécutable ne se trouve pas dans un des répertoires du `$PATH`, alors le chemin complet doit être précisé
- Exemple de `$PATH` :

```
/home/jane/bin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
```

- Ligne typique dans la config de bash :

```
export PATH=~/.local/bin:$PATH
```


Exercice PATH

1. Dans quel fichier écrire la ligne ci-dessous, `.bashrc` ou `.bash_profile` ?

```
export PATH=~/.local/bin:$PATH
```

2. Créez l'arborescence `.local/bin/` dans votre *home* et ajoutez-y un script exécutable
3. Tentez d'exécuter ce script sans indiquer son chemin
4. Ajoutez ce chemin à votre `PATH`
5. Vérifiez que cela fonctionne en exécutant le script, sans indiquer son chemin et depuis un autre répertoire

Alias

- On peut définir des alias pour raccourcir l'invocation de commandes et de leurs arguments
- On peut accéder à la commande originale en utilisant un *backslash* : \
- La commande `alias` exécutée seule affiche tous les alias
- On trouve typiquement ceci dans `.bashrc` :

```
alias rm='rm -i'
alias cp='cp -i'
alias mv='mv -i'
alias ls='ls --color=auto'
alias ll='ls -lh'
alias la='ll -a'
```

Exercice alias

Ajoutez un alias `findn` qui cherche dans le répertoire courant un fichier portant le nom passé en paramètre

Permissions par défaut (umask)

- Il est possible de spécifier un masque de permissions par défaut pour la création de fichiers et répertoires grâce à la commande `umask` :

```
umask [permissions]
```

- Ci-dessus, `permissions`, représente les permissions **à enlever** en octal
- Exécuté sans argument, `umask` affiche la valeur actuelle du masque
- Par exemple, on désire retirer au groupe le droit d'écrire et retirer aux autres (*other*) tous les droits :

```
umask 027
```

Exercice umask

- Observez le umask courant, puis créez un fichier et un répertoire pour vérifier la correspondance
- Changez la valeur du umask, puis créez un fichier pour vérifier la correspondance
- À l'aide d'un test empirique, déterminez si le umask est hérité d'une shell à l'autre

Commandes utiles

<code>awk -F":" '{print \$n}'</code>	Affiche la n ème colonne, où le séparateur de colonnes est :
<code>basename FILE</code>	Extrait le nom du fichier sans son chemin
<code>dirname FILE</code>	Extrait le chemin d'un fichier, sans le fichier
<code>sed s/JPEG/jpg/</code>	remplace 'JPEG' par 'jpeg'
<code>uname -a</code>	Affiche les informations du système (nom machine, version noyau, etc.)
<code>cut -c m-n</code>	Affiche les caractères de m à n de chaque ligne
<code>head -n2</code>	Affiche les 2 premières lignes
<code>tail -n2</code>	Affiche les 2 dernières lignes
<code>find pics -name "*.jpg" -exec convert {} {}.png \;</code>	Converti récursivement toutes les images se trouvant dans pics de jpg à png
<code>grep -rn main --include "*.go"</code>	Recherche le <i>pattern</i> main récursivement dans tous les fichiers se terminant par .go et affiche chaque ligne matchée

- Livre :
 - *Unix Power Tools, 2002, O'Reilly*
- Liens :
 - [Bash Guide for Beginners](#)
 - [Ryan's Bash Scripting Tutorial](#)
 - [Bash Scripting Tutorial: How to Write a Bash Script](#)
 - [Advanced Bash-Scripting Guide](#)