# Script bash : enchaînement de commandes et redirection

- Objet : Script bash : enchaînement de commandes et redirection
- Niveau requis :
  débutant, avisé
- Commentaires : Fix Me!
- Débutant, à savoir : Utiliser GNU/Linux en ligne de commande, tout commence là !.
- Suivi :
  - Création par Hypathie le 18/03/2014
  - Testé par Hypathie Juin 2014
- Commentaires sur le forum : Lien vers le forum concernant ce tuto [1)]

Contributeurs, les Fix Me! sont là pour vous aider, supprimez-les une fois le problème corrigé ou le champ rempli !

**Nota : Les autres wiki :**

- debuter-avec-les-scripts-shell-bash
- script-bash-variables-arguments-parametres
- modification de variable et de paramètre
- 
- script-bash-etat-de-sortie-et-les-tests
- script-bash-les-tableaux
- script-bash-les-fonctions

## Enchaînements de commandes dans les scripts

Parmi les opérateurs d'enchaînement de commandes (voir les opérateurs de contrôle)

- considérons maintenant :

```
&    &&   ;   ( )    { }
```

> Pour ce qui concerne les opérateurs de contrôle du point de vue de **l'enchaînement conditionné** (avec && et || )
> voir : état de sortie et code de retour.

### Parallélisme et succession & , && et ;

- Avec l'opérateur de contrôle & : parallélisme.

> Toutes les commandes sont exécutées parallèlement.
> Dans l'exemple ci-dessus, chaque commande étant traitée par un sous-shell,
> le résultat d'une commande ne peut pas être conservée dans un même processus,
> afin que chaque commande puisse “travailler” en rapport au résultat de la
> commande précédente.

> Par exemple, il faut que le fichier créé soit “connu” du shell pour qu'il puise être ouvert par “gedit” dans ce même shell.)

- Avec l'opérateur de contrôle && : succession conditionnée.

> La commande suivante est exécutée successivement à la précédente, seulement si cette précédente commande a fonctionné.
> Voir code de retour.

- Avec l'opérateur `;` : succession non conditionnée

> Chaque commande est exécuté l'une après l'autre même si l'une d'elle a mal fonctionné.

**Exemple**

script

```
#!/bin/bash
set -o posix
printf "Un nouveau script utilisateur : son nom ? "
{ read nom && echo "#!/bin/bash" >> $nom && chmod u+x $nom && mv ~/$nom
~/MesScripts && /usr/bin/gedit ~/MesScripts/$nom ;}
```

> En mettant && entre les commandes, ce script fonctionne aussi bien qu'avec les `;`.
> Mais avec &, on obtiendrait un message d'erreur.
>
> Dans cet exemple, la valeur d'une commande est nécessaire à la commande suivante.
> On n'a pas chercher à rediriger le résultat d'une commande vers un fichier ou le contenu d'un fichier vers une commande (excepté pour concaténer dans le fichier créé la première ligne du futur script).
> Pour le faire, il faut utiliser les redirections (voir plus bas).
> On n'a pas cherché non plus à transmettre le résultat d'une commande
> à une autre commande (tube `|`)

## Regroupement de commandes, parenthèses ou accolades ?

Le shell bash fournit deux mécanismes pour regrouper les commandes:

1. l'insertion de la suite de commandes entre accolades ;
2. l'insertion de cette suite de commandes entre une paire de parenthèses.

### { suite-de-commandes ;}

Entre accolades, la valeur change commande après commande et le changement est conservé jusqu'à la dernière commande parce que toutes la série appartient au même processus.

Les accolades sont des mots-clé de bash.
Il ne faut donc pas oublier de mettre un espace entre l'accolade ouvrante et la première commande de la liste.

script

```
#!/bin/bash
{ pwd ; cd ~/Documents ; echo $(pwd) ;}
```

```
/home/hypathie
/home/hypathie/Documents
```

Pour se servir des accolades pour conserver la valeur d'une variable et la faire changer de commande en commande, il ne faut pas terminer le regroupement de commandes par &.
Car cela a pour effet de ne pas exécuter chaque commande dans le shell courant mais dans un sous-shell.

script

```
#!/bin/bash
{ pwd ; cd ~/Documents ; echo $(pwd) ;} &
```

```
/home/hypathie
```

> Le prompt ne revient pas il faut faire Ctrl+C !

> L'utilisation du groupement de commandes sert souvent à la redirection globale de l'entrée du groupe de commande ou à sa sortie.
> On le verra plus loin.

**(suite-de-commandes ;)**

Les parenthèses sont des opérateurs.
Il n'y a donc pas besoin d'espace entre la parenthèse ouvrante et la première commande.
Insérée dans une parenthèse, la suite de commandes est exécutée dans un sous-shell.

script

```
#!/bin/bash
nom=nenette
(prenom=hypathie ; echo $prenom )
echo $nom
```

```
hypathie
nenette
```

Voir : Guide avancé d'écriture des scripts Bash: 21. Sous-shells

On ne confondra plus !

script

```
#!/bin/bash
{ var1=yep ;}
echo $var1
var3=coucou
{ var4=yep ; $var4 ;}
echo $var3
```

```
yep
essai.sh: ligne5: yep : commande introuvable
coucou
```

Mais :

script

```
#!/bin/bash
var1=coucou
(var2=yep )
echo $var1 $var2
( var3="au revoir" ; echo $var3 )
```

```
coucou
au revoir
```

## Notion de sous-shell

Les variables comprises dans ces parenthèses, à l'intérieur du sous-shell, ne sont pas visibles par le reste du script. Le processus parent ou père ne peut pas accéder aux variables créées dans le processus fils, le sous-shell.

Dans le script ci-dessous, on voit que le terminal “reçoit le retour du shell père et celui du shell fils ( “`echo $prenom`” ).

### Le processus père ne peut pas récupérer les variables d'un processus fils

script

```
#!/bin/bash
nom=nenette
( prenom=hypathie )
echo $nom $prenom
```

```
nenette
```

> Créer un sous-shell permet ainsi de protéger de ce qui se passe dans le processus fils.
> Créer un processus fil permet aussi au processus père de continuer son programme “pendant” l'exécution du processus fils, ce qui évite de ralentir l'ensemble du programme.

En graphique, un sous-shell pour récupérer la main sur le terminal, c'est bien pratique 😎
Comparez :

```
iceweasel
```

avec :

```
iceweasel &
[1] 4245
```

> Ici `[1]` est le jobID et 4245 est le PID, l'identifiant du processus.
> Pour aller plus loin voir la notion de processus et les commandes : `ps` ; `top` ; `nice` et `renice`.
>
> Dans le premier cas, si l'on ferme le terminal, on ferme aussi “iceweasel”, processus fils du shell.
> Mais dans le deuxième cas (en mettant & après la commande) le processus fils est en arrière plan et l'on peut donc utiliser le shell, ou fermer le terminal sans tuer en même temps le processus fils (iceweasel).

### Lancer deux sous-shell en parallèle

Il est possible de lancer deux processus en parallèle. Comparez le retour des lignes n°2 et n°3 avec celles des lignes n°5 et n°6 du code ci-dessous.

script

```
#!/bin/bash
( echo "bonjour" ) & ( echo "au revoir" )
```

```
( cd /etc/apt ; ls ) & ( cd /etc/calendar ; ls )
echo " "
cd /etc/apt ; ls
cd /etc/calendar ; ls
```

### Le shell restreint

#### Passer en mode restreint

```
set --restricted
```

ou

```
set -r
```

#### Usage dans les scripts

On passe en mode restreint pour diminuer les risques.
En mode restreint, certaines commandes sont désactivées :

- cd
- exec (commande qui permet des substitution de processus
- empêche de sortir du mode restreint depuis le script qui la mis en place
- empêche les redirections de sortie (écrire dans des fichiers)
- empêche l'utilisation de commandes contenant / (pour éviter des modification à la racine)
- empêche de modifier les valeurs des variables d'environnement
  - $PATH : en le modifiant on peut changer l'utilisation de certaines commandes.
  - $SHELL : en le modifiant on peut utiliser un autre interpréteur, en cours de programme.
  - $BASH_ENV et $ENV : les modifications de l'environnement ne se font pas sans connaissances.
  - $SHELLOPTS : où on peut changer les options d'environnement du shell
- et les droits sont limités.

## Redirections et le pipe

- Liste des opératieurs : un tableau des opérateurs de redirection
- Exercices dans le terminal, tout est là : les chevrons

### Redirections : quelques points importants pour les scripts

#### Rappels sur les flux

Les redirections permettent de travailler non pas en se servant du code de retour (qui indique la réussite ou l'échec de l'exécution d'une commande) mais sur les flux.
Un processus unix possède (par défaut) trois voies d'interaction entre le système et l'utilisateur. Une entrée et deux sorties. Chacun de ces “lieux” sont identifiés par un descripteur de fichier.

1. une entrée standard (par défaut le clavier stdin), de descripteur 0 (nom de l'entrée du processus, ne pas confondre avec le code de retour !);
2. une sortie standard (par défaut l'écran stdout), de descripteur 1 ;
3. une sortie standard pour les message d'erreur (stderr) de descripteur 2.

Pour chaque programme lancé, un flux est créé. Ce flux est une sorte de canal par lequel transite les données entre les espaces, entrée et sortie.
On peut imaginer un terminal, comme la réunion virtuelle d'un clavier et d'un écran. **Merci à captnfab pour cette comparaison sur IRC** 😀

```
<&-   <&-  # Permettent la fermeture de l'entrée standard et de la sortie standard.
 >|        # Force une redirection vers un fichier.txt pour pouvoir écraser
             le fichier quand il existe et que l'option noclobber (-c) est activée.
```

### Utilisation de la sortie d'erreur dans les scripts

- La sortie standard d'erreur peut être dirigée vers un fichier en le créant ou en l'écrasant :

```
ls vi 2>err   # retour du prompt : le message d'erreur a été inscrit
                dans le fichier "erreurs" qui s'est créé s'il n'existait pas.
```

- On peut aussi concaténer :

```
2>>fichier    # si "fichier" n'existait pas le message
                d'erreur aurait été ajouté en dernière ligne.
```

- Elle peut aussi être dirigée vers le fichier “poubelle” ou “puits” :

```
2>/dev/null
```

(Tout ce qui y est dirigé est perdu, inutile de concaténer !) On s'en sert souvent lorsqu'on est intéressé par le fait de récupérer le code de retour.

- redirection de la sortie d'une commande vers un autre canal :

```
 >&
```

```
ls -l 1>&2  la sortie du répertoire courant et envoyé sur le canal de sortie d'erreur ;
            cela à pour effet, de lister le contenu, mais le terminal affiche alors le canal
            de sortie d'erreur. Relancer la dernière commande est impossible.
```

```
            On peut lancer une autre commande, ou faire ctrl+c. Ooufffff
```

- Pour diriger la sortie standard et la sortie d'erreur à la fin d'un fichier :

```
 >>&
```

- Pour rediriger la sortie standard des messages d'erreur (stderr) vers la sortie standard #(stdout), on utilise la syntaxe :

```
2>&1
```

Par exemple :

```
##vi: /usr/bin/vi
ls vi 2>&1 2>erreurs  # retour du prompt on retrouve le message d'erreurs
                        dans le fichier "erreurs" qui s'est créé.
                      Cela un équivalent de ls vi 2>err
```

```
ls vi erreur>errrrr 2>&1  #retour du prompt
                             ls ne peut lister le fichier vi ; le message
d'erreur
                             est envoyé dans le fichier "errrr" qui est
nouvellement créé
                             et qui est la sortie standard (1),
                             puis ls ne peut lister le fichier "erreur", le
message est
                             envoyé vers la sortie d'erreur qui est redirigé
vers (1)
                             c'est-à-dire le fichier "erreur".
```

#### Utilisation du "text processing"

- Tout est là : http://wiki.debian-facile.org/atelier:chantier:les-commandes-join-paste-split-et-nl

## Le pipe

Tout est là : le pipe

## Un petit exercice sur opérateurs d'enchaînement et de redirection

#### énoncé

Écrire un script qui crée le dossier “ABCD” et 4 fichiers vides (nommés a b c d) ; qui liste le contenu de “ABCD” et qui inscrit le résultat dans un fichier nommé “ls1” qui sera placé dans “ABCD” ; qui depuis le répertoire personnel crée le fichier vide nommé “fichier.txt”,
liste à nouveau ABCD, inscrit le résultat dans le fichier “ls2”, rangé dans “ABCD”; qui permet

d'inscrire depuis le terminal une ligne de texte dans le fichier nommé “fichier.txt” ; puis une deuxième ligne de texte dans “fichier.txt”, en affichant dans le terminal,
le nombre de lignes, de mots et d'octets que possède le fichier “fichier.txt” ; se servir de différentes méthodes tout au long du script pour vérifier
au niveau du terminal que chaque commande s'est bien déroulée.

Attention: Créer un fichier avec > est un bashisme. La méthode universelle est d'utiliser “touch”. Amateurs de magie blanche et noire voir ce fil : http://debian-facile.org/viewtopic.php?pid=86634#p86634

**une solution**

Bonne lecture 

script

```
#!/bin/bash
set -o posix
{ mkdir ~/ABCD 2>>/dev/null ;\
 echo $? ;\
 cd ABCD && touch a b c d ;\
 echo $? ;\
 ls -l >> ~/ABCD/ls1 ;\
 echo $? ;\
 cd ~ ;\
 pwd ;\
 touch ~/ABCD/fichier.txt ;\
 echo $? ;\
 pwd && ls -l ~/ABCD >> ~/ABCD/ls2 ;\
 echo $? ;\
 read phrase1 && echo ${phrase1} >> ~/ABCD/fichier.txt && echo $? ;\
 read phrase2 ;\
 cat >> ~/ABCD/fichier.txt << EOF
$phrase2
EOF
echo $?
cat < ~/ABCD/fichier.txt | wc
echo $? ;}
```

- **2»/dev/null** : permet ici de relancer le script autant de fois qu'on veut, sans voir apparaître de message d'erreur : mkdir fichier-existant ne réinitialise pas un fichier de type dossier en le vidant.
- **$?** : permet ici de vérifier que la commande précédente s'est déroulée avec succès quand “**;**” a été utilisé, inutile de vérifier avec **&&**.

# La suite c'est ici

script-bash-etat-de-sorie-et-les-tests

1)

N'hésitez pas à y faire part de vos remarques, succès, améliorations ou échecs !