eBook Gratuit

APPRENEZ Bash

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#bash

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: bash

It is an unofficial and free Bash ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Bash.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec Bash

Versions

Version	Date de sortie
0,99	1989-06-08
1,01	1989-06-23
2.0	1996-12-31
2,02	1998-04-20
2,03	1999-02-19
2,04	2001-03-21
2.05b	2002-07-17
3.0	2004-08-03
3.1	2005-12-08
3.2	2006-10-11
4.0	2009-02-20
4.1	2009-12-31
4.2	2011-02-13
4.3	2014-02-26
4.4	2016-09-15

Examples

Bonjour tout le monde en utilisant des variables

Créez un nouveau fichier appelé `hello.sh` avec le contenu suivant et attribuez-lui des droits exécutables avec `chmod +x hello.sh` .

> Exécuter / Exécuter via: `./hello.sh`

```
#!/usr/bin/env bash
```

```
# Note that spaces cannot be used around the `=` assignment operator
whom_variable="World"

# Use printf to safely output the data
printf "Hello, %s\n" "$whom_variable"
#> Hello, World
```

Cela affichera `Hello, World` sur la sortie standard lorsqu'elle sera exécutée.

Pour dire à bash où le script est, vous devez être très précis, en le pointant vers le répertoire contenant, normalement avec `./` s'il s'agit de votre répertoire de travail, où `.` est un alias du répertoire en cours. Si vous ne spécifiez pas le répertoire, `bash` essaie de localiser le script dans l'un des répertoires contenus dans la variable d'environnement `$PATH` .

Le code suivant accepte un argument `$1` , qui est le premier argument de la ligne de commande, et le renvoie dans une chaîne formatée, après `Hello,` .

> Exécuter / Exécuter via: `./hello.sh World`

```
#!/usr/bin/env bash
printf "Hello, %s\n" "$1"
#> Hello, World
```

Il est important de noter que `$1` doit être cité en double citation, et non en guichet unique. `"$1"` développe comme le premier argument de la ligne de commande, tandis que `'$1'` évalué à la chaîne littérale `$1` .

> **Note de sécurité:**
> Lisez **les implications de la sécurité en oubliant de citer une variable dans les shells bash** pour comprendre l'importance de placer le texte variable entre guillemets.

Bonjour le monde

Shell interactif

Le shell Bash est couramment utilisé de manière **interactive:** il vous permet d'entrer et de modifier des commandes, puis de les exécuter lorsque vous appuyez sur la touche `Retour` . De nombreux systèmes d'exploitation basés sur Unix ou sur Unix utilisent Bash comme shell par défaut (notamment Linux et macOS). Le terminal entre automatiquement dans un processus interactif de shell Bash au démarrage.

Sortez `Hello World` en tapant ce qui suit:

```
echo "Hello World"
#> Hello World  # Output Example
```

Remarques

- Vous pouvez changer le shell en tapant simplement le nom du shell dans le terminal. Par exemple: `sh` , `bash` , etc.
- `echo` est une commande intégrée Bash qui écrit les arguments reçus sur la sortie standard. Il ajoute une nouvelle ligne à la sortie, par défaut.

Shell non interactif

Le shell Bash peut également être exécuté de manière **non interactive à** partir d'un script, ce qui ne nécessite aucune interaction humaine. Le comportement interactif et le comportement par script doivent être identiques - une considération importante pour la conception du shell Unix V7 Bourne et de Bash de manière transitoire. Par conséquent, tout ce qui peut être fait sur la ligne de commande peut être placé dans un fichier script pour être réutilisé.

Suivez ces étapes pour créer un script `Hello World` :

1. Créez un nouveau fichier appelé `hello-world.sh`

   ```
   touch hello-world.sh
   ```

2. Rendre le script exécutable en exécutant `chmod +x hello-world.sh`

3. Ajoutez ce code:

   ```
   #!/bin/bash
   echo "Hello World"
   ```

   Ligne 1 : La première ligne du script doit commencer par la séquence de caractères `#!` , appelé *shebang* [1] . Le shebang demande au système d'exploitation d'exécuter `/bin/bash` , le shell Bash, en lui transmettant le chemin du script comme argument.

   Par exemple `/bin/bash hello-world.sh`

   Ligne 2 : Utilise la commande `echo` pour écrire `Hello World` sur la sortie standard.

4. Exécutez le script `hello-world.sh` partir de la ligne de commande en utilisant l'un des éléments suivants:

   - `./hello-world.sh` - le plus couramment utilisé et recommandé
   - `/bin/bash hello-world.sh`
   - `bash hello-world.sh` - en supposant que `/bin` soit dans votre `$PATH`
   - `sh hello-world.sh`

Pour une utilisation en production réelle, vous devez omettre l'extension `.sh` (ce qui est trompeur, car il s'agit d'un script Bash, et non d'un script `sh`) et peut-être déplacer le fichier dans un répertoire de votre `PATH` votre répertoire de travail actuel, tout comme une commande système telle que `cat` ou `ls` .

Les erreurs courantes incluent:

1. Oublier d'appliquer les droits d'exécution sur le fichier, c'est-à-dire `chmod +x hello-world.sh` , ce qui entraîne la sortie de `./hello-world.sh: Permission denied` .

2. Modifier le script sous Windows, ce qui produit des caractères de fin de ligne incorrects que Bash ne peut pas gérer.

   Un symptôme courant est `: command not found` lorsque le retour chariot a forcé le curseur au début de la ligne, remplaçant le texte précédant les deux points dans le message d'erreur.

   Le script peut être corrigé à l'aide du programme `dos2unix` .

   Un exemple d'utilisation: `dos2unix hello-world.sh`

   `dos2unix` édite le fichier en ligne.

3. En utilisant `sh ./hello-world.sh` , ne réalisant pas que `bash` et `sh` sont des shells distincts avec des fonctionnalités distinctes (bien que Bash est rétro-compatible, l'erreur inverse est inoffensive).

   Quoi qu'il en soit, il est préférable d'utiliser la ligne shebang du script plutôt que d'écrire explicitement `bash` ou `sh` (ou `python` ou `perl` ou `awk` ou `ruby` ou ...) avant le nom de fichier de chaque script.

   Une ligne de shebang commune à utiliser pour rendre votre script plus portable consiste à utiliser `#!/usr/bin/env bash` au lieu de coder en dur un chemin vers Bash. De cette façon, `/usr/bin/env` doit exister, mais au-delà, `bash` doit simplement être sur votre `PATH` . Sur de nombreux systèmes, `/bin/bash` n'existe pas et vous devez utiliser `/usr/local/bin/bash` ou un autre chemin absolu; Ce changement évite de devoir en comprendre les détails.

---

[1] *Également appelé sha-bang, hashbang, pound-bang, hash-pling.*

Affichage des informations sur les éléments intégrés Bash

```
help <command>
```

Cela affichera la page d'aide Bash (manuelle) pour le intégré spécifié.

Par exemple, `help unset` affichera:

```
unset: unset [-f] [-v] [-n] [name ...]
   Unset values and attributes of shell variables and functions.

   For each NAME, remove the corresponding variable or function.

   Options:
     -f    treat each NAME as a shell function
     -v    treat each NAME as a shell variable
     -n    treat each NAME as a name reference and unset the variable itself
```

```
       rather than the variable it references

   Without options, unset first tries to unset a variable, and if that fails,
   tries to unset a function.

   Some variables cannot be unset; also see `readonly'.

   Exit Status:
   Returns success unless an invalid option is given or a NAME is read-only.
```

Pour voir une liste de tous les éléments intégrés avec une courte description, utilisez

```
help -d
```

Bonjour tout le monde avec entrée utilisateur

Ce qui suit invitera l'utilisateur à saisir des données, puis stockera cette entrée sous forme de chaîne (texte) dans une variable. La variable est ensuite utilisée pour donner un message à l'utilisateur.

```
#!/usr/bin/env bash
echo  "Who are you?"
read name
echo "Hello, $name."
```

La commande `read` here lit une ligne de données de l'entrée standard dans le `name` la variable. Ceci est ensuite référencé en utilisant `$name` et imprimé en standard avec `echo` .

Exemple de sortie:

```
$ ./hello_world.sh
Who are you?
Matt
Hello, Matt.
```

Ici, l'utilisateur a entré le nom "Matt", et ce code a été utilisé pour dire `Hello, Matt.` .

Et si vous souhaitez ajouter quelque chose à la valeur de la variable lors de son impression, utilisez des accolades autour du nom de la variable, comme illustré dans l'exemple suivant:

```
#!/usr/bin/env bash
echo  "What are you doing?"
read action
echo "You are ${action}ing."
```

Exemple de sortie:

```
$ ./hello_world.sh
What are you doing?
Sleep
You are Sleeping.
```

Ici, lorsque l'utilisateur entre une action, "ing" est ajouté à cette action lors de l'impression.

Gestion des arguments nommés

```
#!/bin/bash

deploy=false
uglify=false

while (( $# > 1 )); do case $1 in
   --deploy) deploy="$2";;
   --uglify) uglify="$2";;
   *) break;
 esac; shift 2
done

$deploy && echo "will deploy... deploy = $deploy"
$uglify && echo "will uglify... uglify = $uglify"

# how to run
# chmod +x script.sh
# ./script.sh --deploy true --uglify false
```

Bonjour tout le monde en mode "Debug"

```
$ cat hello.sh
#!/bin/bash
echo "Hello World"
$ bash -x hello.sh
+ echo Hello World
Hello World
```

L'argument `-x` vous permet de parcourir chaque ligne du script. Un bon exemple est ici:

```
$ cat hello.sh
#!/bin/bash
echo "Hello World\n"
adding_string_to_number="s"
v=$(expr 5 + $adding_string_to_number)

$ ./hello.sh
Hello World

expr: non-integer argument
```

L'erreur suggérée ci-dessus n'est pas suffisante pour tracer le script; Cependant, l'utilisation de la méthode suivante vous permet de mieux détecter l'erreur dans le script.

```
$ bash -x hello.sh
+ echo Hello World\n
Hello World

+ adding_string_to_number=s
+ expr 5 + s
expr: non-integer argument
```

```
+ v=
```

Importance de citer dans les chaînes

La citation est importante pour l'expansion des chaînes en bash. Avec ceux-ci, vous pouvez contrôler la façon dont le bash analyse et développe vos chaînes.

Il existe deux types de devis:

- **Faible** : *utilise des guillemets doubles: "*
- **Fort** : *utilise des guillemets simples: '*

Si vous voulez vous baser pour développer votre argument, vous pouvez utiliser **Weak Quoting** :

```
#!/usr/bin/env bash
world="World"
echo "Hello $world"
#> Hello World
```

Si vous ne voulez pas étendre votre argument, vous pouvez utiliser **Strong Quoting** :

```
#!/usr/bin/env bash
world="World"
echo 'Hello $world'
#> Hello $world
```

Vous pouvez également utiliser la commande escape pour empêcher l'expansion:

```
#!/usr/bin/env bash
world="World"
echo "Hello \$world"
#> Hello $world
```

Pour des informations plus détaillées que les détails du débutant, vous pouvez continuer à le lire ici .

Lire Démarrer avec Bash en ligne: https://riptutorial.com/fr/bash/topic/300/demarrer-avec-bash

Chapitre 2: Achèvement programmable

Examples

Achèvement simple en utilisant la fonction

```
_mycompletion() {
    local command_name="$1" # not used in this example
    local current_word="$2"
    local previous_word="$3" # not used in this example
    # COMPREPLY is an array which has to be filled with the possible completions
    # compgen is used to filter matching completions
    COMPREPLY=( $(compgen -W 'hello world' -- "$current_word") )
}
complete -F _mycompletion mycommand
```

Exemple d'utilisation:

```
$ mycommand [TAB][TAB]
hello world
$ mycommand h[TAB][TAB]
$ mycommand hello
```

Achèvement simple pour les options et les noms de fichiers

```
# The following shell function will be used to generate completions for
# the "nuance_tune" command.
_nuance_tune_opts ()
{
  local curr_arg prev_arg
  curr_arg=${COMP_WORDS[COMP_CWORD]}
  prev_arg=${COMP_WORDS[COMP_CWORD-1]}

  # The "config" option takes a file arg, so get a list of the files in the
  # current dir.  A case statement is probably unnecessary here, but leaves
  # room to customize the parameters for other flags.
  case "$prev_arg" in
    -config)
      COMPREPLY=( $( /bin/ls -1 ) )
      return 0
      ;;
    esac

    # Use compgen to provide completions for all known options.
    COMPREPLY=( $(compgen -W '-analyze -experiment -generate_groups -compute_thresh -config -
output -help -usage -force -lang -grammar_overrides -begin_date -end_date -group -dataset -
multiparses -dump_records -no_index -confidencelevel -nrecs -dry_run -rec_scripts_only -
save_temp -full_trc -single_session -verbose -ep -unsupervised -write_manifest -remap -
noreparse -upload -reference -target -use_only_matching -histogram -stepsize' -- $curr_arg )
);
}

# The -o parameter tells Bash to process completions as filenames, where applicable.
```

```
complete -o filenames -F _nuance_tune_opts nuance_tune
```

Lire Achèvement programmable en ligne: https://riptutorial.com/fr/bash/topic/3162/achevement-programmable

Chapitre 3: Aliasing

Introduction

Les alias de shell sont un moyen simple de créer de nouvelles commandes ou d'emballer des commandes existantes avec du code de votre choix. Ils se chevauchent quelque peu avec les fonctions de la coque, qui sont cependant plus polyvalentes et devraient donc souvent être préférées.

Remarques

L'alias ne sera disponible que dans le shell où la commande alias a été lancée.

Pour conserver l'alias, envisagez de le placer dans votre `.bashrc`

Examples

Créer un alias

```
alias word='command'
```

Invoquer un `word` exécutera la `command` . Tous les arguments fournis à l'alias sont simplement ajoutés à la cible de l'alias:

```
alias myAlias='some command --with --options'
myAlias foo bar baz
```

Le shell va alors exécuter:

```
some command --with --options foo bar baz
```

Pour inclure plusieurs commandes dans le même alias, vous pouvez les `&&` à `&&` . Par exemple:

```
alias print_things='echo "foo" && echo "bar" && echo "baz"'
```

Liste de tous les alias

```
alias -p
```

listera tous les alias actuels.

Développer alias

En supposant que cette `bar` est un alias pour `someCommand -flag1` .

Tapez `bar` sur la ligne de commande, puis appuyez sur `Ctrl` + `alt` + `e`

vous obtiendrez `someCommand -flag1` où la `bar` était debout.

Supprimer un alias

Pour supprimer un alias existant, utilisez:

```
unalias {alias_name}
```

Exemple:

```
# create an alias
$ alias now='date'

# preview the alias
$ now
Thu Jul 21 17:11:25 CEST 2016

# remove the alias
$ unalias now

# test if removed
$ now
-bash: now: command not found
```

Contourner un alias

Parfois, vous pouvez vouloir ignorer un alias temporairement, sans le désactiver. Pour travailler avec un exemple concret, considérez cet alias:

```
alias ls='ls --color=auto'
```

Et disons que vous voulez utiliser la commande `ls` sans désactiver l'alias. Vous avez plusieurs options:

- Utilisez la `command` incorporée: `command ls`
- Utilisez le chemin complet de la commande: `/bin/ls`
- Ajoutez un `\` n'importe où dans le nom de la commande, par exemple: `\ls` ou `l\s`
- Citez la commande: `"ls"` ou `'ls'`

BASH_ALIASES est un tableau d'assoc bash interne

Les alias sont nommés raccourcis de commandes, on peut définir et utiliser dans des instances de bash interactives. Ils sont contenus dans un tableau associatif nommé BASH_ALIASES. Pour utiliser cette variable dans un script, il doit être exécuté dans un shell interactif

```
#!/bin/bash -li
# note the -li above! -l makes this behave like a login shell
# -i makes it behave like an interactive shell
```

```
#
# shopt -s expand_aliases will not work in most cases

echo There are ${#BASH_ALIASES[*]} aliases defined.

for ali in "${!BASH_ALIASES[@]}"; do
   printf "alias: %-10s triggers: %s\n" "$ali" "${BASH_ALIASES[$ali]}"
done
```

Lire Aliasing en ligne: https://riptutorial.com/fr/bash/topic/368/aliasing

Chapitre 4: Arithmétique Bash

Syntaxe

- $ ((EXPRESSION)) - Evalue l'expression et renvoie son résultat.
- expr EXPRESSION - Imprime le résultat de l'EXPRESSION sur la sortie standard.

Paramètres

Paramètre	Détails
EXPRESSION	Expression à évaluer

Remarques

Un espace ("") est requis entre chaque terme (ou signe) de l'expression. "1 + 2" ne fonctionnera pas, mais "1 + 2" fonctionnera.

Examples

Commande arithmétique

- `let`

```
let num=1+2
let num="1+2"
let 'num= 1 + 2'
let num=1 num+=2
```

Vous avez besoin de citations s'il y a des espaces ou des caractères de globalisation. Donc, ceux-ci auront une erreur:

```
let num = 1 + 2    #wrong
let 'num = 1 + 2'  #right
let a[1] = 1 + 1   #wrong
let 'a[1] = 1 + 1' #right
```

- `(( ))`

```
((a=$a+1))     #add 1 to a
((a = a + 1))  #like above
((a += 1))     #like above
```

Nous pouvons utiliser `(())` dans `if` . Quelques exemples:

```
if (( a > 1 )); then echo "a is greater than 1"; fi
```

La sortie de `(())` peut être affectée à une variable:

```
result=$((a + 1))
```

Ou utilisé directement en sortie:

```
echo "The result of a + 1 is $((a + 1))"
```

Arithmétique simple avec (())

```
#!/bin/bash
echo $(( 1 + 2 ))
```

Sortie: 3

```
# Using variables
#!/bin/bash
var1=4
var2=5
((output=$var1 * $var2))
printf "%d\n" "$output"
```

Sortie: 20

Arithmétique simple avec expr

```
#!/bin/bash
expr 1 + 2
```

Sortie: 3

Lire Arithmétique Bash en ligne: https://riptutorial.com/fr/bash/topic/3652/arithmetique-bash

Chapitre 5: Bash sur Windows 10

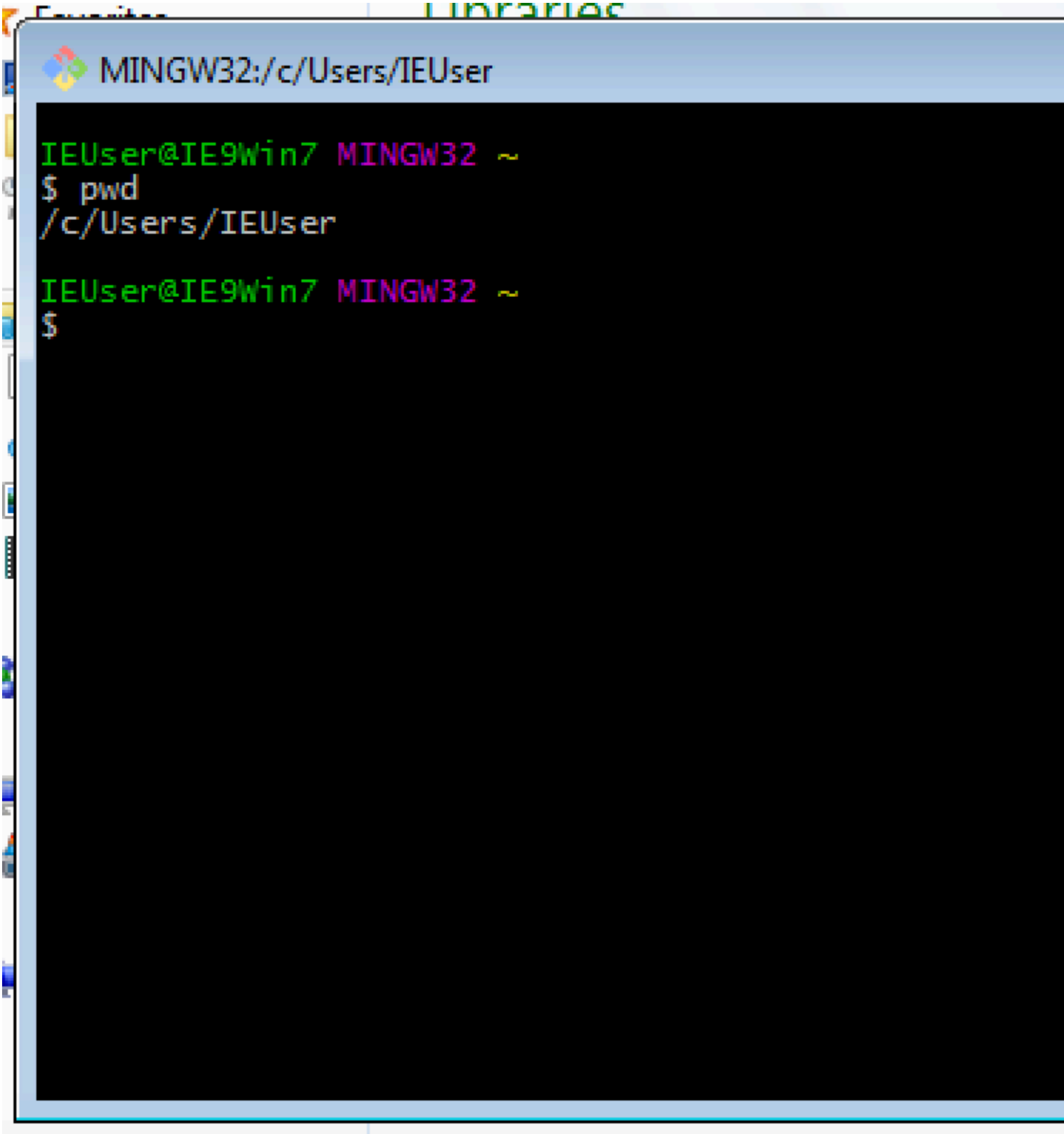
Examples

Readme

La manière la plus simple d'utiliser Bash sous Windows consiste à installer Git pour Windows. Il est livré avec Git Bash qui est un véritable Bash. Vous pouvez y accéder avec un raccourci dans:

```
Start > All Programs > Git > Git Bash
```

Des commandes telles que `grep` , `ls` , `find` , `sed` , `vi` etc fonctionnent.



Lire Bash sur Windows 10 en ligne: https://riptutorial.com/fr/bash/topic/9114/bash-sur-windows-10

Chapitre 6: Chaîne de commandes et d'opérations

Introduction

Il existe des moyens de chaîner les commandes ensemble. Des simples comme juste un; ou plus complexes comme les chaînes logiques qui fonctionnent selon certaines conditions. Le troisième est les commandes de canalisation, qui transfèrent efficacement les données de sortie à la commande suivante de la chaîne.

Examples

Compter une occurrence de modèle de texte

L'utilisation d'un tube fait que la sortie d'une commande est l'entrée du prochain.

```
ls -1 | grep -c ".conf"
```

Dans ce cas, la sortie de la commande ls est utilisée comme entrée de la commande grep. Le résultat sera le nombre de fichiers qui incluent ".conf" dans leur nom.

Cela peut être utilisé pour construire des chaînes de commandes ultérieures aussi longtemps que nécessaire:

```
ls -1 | grep ".conf" | grep -c .
```

transférer la sortie racine cmd au fichier utilisateur

Souvent, on veut montrer le résultat d'une commande exécutée par root à d'autres utilisateurs. La commande **tee** permet d'écrire facilement un fichier avec les perms utilisateurs d'une commande exécutée en tant que root:

```
su -c ifconfig | tee ~/results-of-ifconfig.txt
```

Seul **ifconfig** s'exécute en tant que root.

chaînage logique des commandes avec && et ||

&& enchaîne deux commandes. Le second ne fonctionne que si le premier sort avec succès. **||** enchaîne deux commandes. Mais le second ne fonctionne que si le premier sort avec échec.

```
[ a = b  ] && echo "yes" || echo "no"

# if you want to run more commands within a logical chain, use curly braces
```

```
# which designate a block of commands
# They do need a ; before closing bracket so bash can diffentiate from other uses
# of curly braces
[ a = b ] && { echo "let me see."
               echo "hmmm, yes, i think it is true" ; } \
          || { echo "as i am in the negation i think "
               echo "this is false. a is a not b."  ; }
# mind the use of line continuation sign \
# only needed to chain yes block with || ....
```

chaînage en série des commandes avec un point-virgule

Un point-virgule ne sépare que deux commandes.

```
echo "i am first" ; echo "i am second" ; echo " i am third"
```

chaîner des commandes avec |

Le **|** prend la sortie de la commande de gauche et la canalise en entrée de la bonne commande. Attention, cela se fait dans un sous-shell. Par conséquent, vous ne pouvez pas définir les valeurs des vars du processus appelant dans un canal.

```
find . -type f -a -iname '*.mp3'  | \
       while read filename; do
             mute --noise "$filename"
       done
```

Lire Chaîne de commandes et d'opérations en ligne:
https://riptutorial.com/fr/bash/topic/5589/chaine-de-commandes-et-d-operations

Chapitre 7: Changer de coque

Syntaxe

- echo $ 0
- ps -p $$
- echo $ SHELL
- export SHELL = / bin / bash
- exec / bin / bash
- chat / etc / shells

Examples

Trouver le shell actuel

Il y a plusieurs façons de déterminer le shell actuel

```
echo $0
ps –p $$
echo $SHELL
```

Changer la coquille

Pour changer le bash actuel, exécutez ces commandes

```
export SHELL=/bin/bash
exec /bin/bash
```

pour changer le bash qui s'ouvre au démarrage, éditez `.profile` et ajoutez ces lignes

Liste des coques disponibles

Pour répertorier les shells de connexion disponibles:

```
cat /etc/shells
```

Exemple:

```
$ cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/dash
/bin/bash
/bin/rbash
```

Lire Changer de coque en ligne: https://riptutorial.com/fr/bash/topic/3951/changer-de-coque

Chapitre 8: Citant

Syntaxe

- \ C (n'importe quel caractère sauf newline)
- «Tout littéral sauf les guillemets simples»; 'ceci:' \ '' est une citation simple '
- $ 'only \\ et \' sont spéciaux; \ n = nouvelle ligne, etc. '
- "$ variable et autre texte; \" \\\ $ \ `sont spéciaux"

Examples

Newlines et caractères de contrôle

Une nouvelle ligne peut être incluse dans une chaîne simple ou entre guillemets. Notez que backslash-newline n'aboutit pas à une nouvelle ligne, le saut de ligne est ignoré.

```
newline1='
'
newline2="
"
newline3=$'\n'
empty=\

echo "Line${newline1}break"
echo "Line${newline2}break"
echo "Line${newline3}break"
echo "No line break${empty} here"
```

Vous pouvez utiliser des chaînes, des barres obliques inverses ou des barres obliques inverses pour insérer des caractères de contrôle, comme dans de nombreux autres langages de programmation.

```
echo $'Tab: [\t]'
echo $'Tab again: [\009]'
echo $'Form feed: [\f]'
echo $'Line\nbreak'
```

Double guillemets pour la substitution de variables et de commandes

Les substitutions de variables doivent uniquement être utilisées entre guillemets.

```
calculation='2 * 3'
echo "$calculation"         # prints 2 * 3
echo $calculation           # prints 2, the list of files in the current directory, and 3
echo "$(($calculation))"    # prints 6
```

En dehors des guillemets, `$var` prend la valeur de `var` , la divise en parties délimitées par des espaces et interprète chaque partie comme un motif glob (wildcard). À moins que vous ne vouliez

ce comportement, placez toujours `$var` guillemets: `"$var"` .

La même chose s'applique aux substitutions de commandes: `"$(mycommand)"` est le résultat de `mycommand` , `$(mycommand)` est le résultat de split + glob sur la sortie.

```
echo "$var"             # good
echo "$(mycommand)"     # good
another=$var            # also works, assignment is implicitly double-quoted
make -D THING=$var      # BAD! This is not a bash assignment.
make -D THING="$var"    # good
make -D "THING=$var"    # also good
```

Les substitutions de commandes ont leurs propres contextes de cotation. Ecrire des substitutions arbitrairement imbriquées est facile car l'analyseur gardera une trace de la profondeur d'imbrication au lieu de rechercher avec avidité le premier `"` caractère. Le surligneur de la syntaxe StackOverflow analyse cependant cette erreur. Par exemple:

```
echo "formatted text: $(printf "a + b = %04d" "${c}")" # "formatted text: a + b = 0000"
```

Les arguments variables d'une substitution de commande doivent également être entre guillemets à l'intérieur des extensions:

```
echo "$(mycommand "$arg1" "$arg2")"
```

Citant le texte littéral

Tous les exemples de ce paragraphe impriment la ligne

```
!"#$&'()*;<=>?  @[\]^`{|}~
```

Une barre oblique inverse cite le caractère suivant, c'est-à-dire que le caractère suivant est interprété littéralement. La seule exception est une nouvelle ligne: backslash-newline se développe en une chaîne vide.

```
echo \!\"\#\$\&\'\(\)\*\;\<\=\>\?\ \ \@\[\\\]\^\`\{\|\}\~
```

Tout le texte entre guillemets simples (guillemets `'` , également appelé apostrophe, est imprimé littéralement. Même les barres obliques inverses se représentent, et il est impossible d'inclure un seul devis. Au lieu de cela, vous pouvez arrêter la chaîne littérale, inclure un guillemet simple littéral avec une barre oblique inverse et redémarrer la chaîne littérale. Ainsi, la séquence de 4 caractères `'\''` permet effectivement d'inclure une seule citation dans une chaîne littérale.

```
echo '!"#$&'\''()*;<=>?  @[\]^`{|}~'
#           ^^^^
```

Dollar-single-quote lance une chaîne littérale `$'…'` comme beaucoup d'autres langages de programmation, où la barre oblique inverse cite le caractère suivant.

```
echo $'!"#$&\'()*;<=>?  @[\\]^`{|}~'
#           ^^            ^^
```

Les guillemets doubles `"` délimitent des chaînes semi-littérales où seuls les caractères `" \ $` et `` ` `` conservent leur signification particulière. Ces caractères ont besoin d'une barre oblique inverse avant (notez que si la barre oblique inverse est suivie par un autre caractère, la barre oblique inverse reste). Les guillemets doubles sont surtout utiles pour inclure une variable ou une substitution de commande.

```
echo "!\"#\$&'()*;<=>?  @[\\]^\`{|}~"
#      ^^                  ^^  ^^
echo "!\"#\$&'()*;<=>?  @[\]^\`{|}~"
#      ^^                  ^  ^^      \[ prints \[
```

Interactivement, méfiez-vous de cela `!` déclenche l'expansion de l'historique entre guillemets: `"!oops"` recherche une ancienne commande contenant `oops` ; `"\!oops"` ne permet pas l'expansion de l'historique mais conserve la barre oblique inverse. Cela ne se produit pas dans les scripts.

Différence entre guillemet double et devis unique

Double citation	Simple citation
Permet l'expansion des variables	Empêche l'expansion variable
Permet l'expansion de l'historique si activé	Empêche l'expansion de l'histoire
Autorise la substitution de commandes	Empêche la substitution de commande
`*` et `@` peuvent avoir une signification particulière	`*` et `@` sont toujours des littéraux
Peut contenir à la fois une citation simple ou double	Le devis unique n'est pas autorisé dans un devis unique
`$` , `` ` `` , `"` , `\` peut être échappé avec `\` pour empêcher leur signification particulière	Tous sont littéraux

Propriétés communes aux deux:

- Empêche la globulation
- Empêche le fractionnement de mots

Exemples:

```
$ echo "!cat"
echo "cat file"
cat file
$ echo '!cat'
!cat
echo "\"'\""
```

```
"'"
$ a='var'
$ echo '$a'
$a
$ echo "$a"
var
```

Lire Citant en ligne: https://riptutorial.com/fr/bash/topic/729/citant

Chapitre 9: Contrôle de l'emploi

Syntaxe

- long_cmd &
- emplois
- fg% JOB_ID
- fg%? MOTIF
- fg% JOB_ID

Examples

Exécuter la commande en arrière-plan

```
$ sleep 500 &
[1] 7582
```

Met la commande de veille en arrière-plan. 7582 est l'identifiant du processus en arrière-plan.

Liste des processus d'arrière-plan

```
$ jobs
[1]   Running                 sleep 500 &   (wd: ~)
[2]-  Running                 sleep 600 &   (wd: ~)
[3]+  Running                 ./Fritzing &
```

Le premier champ indique les identifiants de travail. Le signe + et - qui suit l'identifiant du travail pour deux travaux indique le travail par défaut et le travail par défaut candidat suivant lorsque le travail par défaut en cours se termine respectivement. Le travail par défaut est utilisé lorsque les commandes `fg` ou `bg` sont utilisées sans aucun argument.

Le deuxième champ indique le statut du travail. Le troisième champ est la commande utilisée pour démarrer le processus.

Le dernier champ (wd: ~) indique que les commandes de veille ont été lancées depuis le répertoire de travail ~ (Home).

Apporter un processus d'arrière-plan au premier plan

```
$ fg %2
sleep 600
```

% 2 spécifie le travail no. 2. Si fg est utilisé sans aucun argument si le dernier processus est placé en arrière-plan au premier plan.

```
$ fg %?sle
sleep 500
```

`?sle` réfère à la commande de processus baground contenant "sle". Si plusieurs commandes d'arrière-plan contiennent la chaîne, une erreur est générée.

Arrêter un processus de premier plan

Appuyez sur Ctrl + Z pour arrêter un processus de premier plan et le placer en arrière-plan

```
$ sleep 600
^Z
[8]+  Stopped                 sleep 600
```

Redémarrez le processus d'arrière-plan arrêté

```
$ bg
[8]+ sleep 600 &
```

Lire Contrôle de l'emploi en ligne: https://riptutorial.com/fr/bash/topic/5193/controle-de-l-emploi

Chapitre 10: Copier (cp)

Syntaxe

- cp [options] destination source

Paramètres

Option	La description
`-a` , `-archive`	Combine les options `d` , `p` et `r`
`-b` , `-backup`	Avant le retrait, effectue une sauvegarde
`-d` , `--no-deference`	Préserve les liens
`-f` , `--force`	Supprimer les destinations existantes sans demander à l'utilisateur
`-i` , `--interactive`	Afficher l'invite avant de remplacer
`-l` , `--link`	Au lieu de copier, liez plutôt des fichiers
`-p` , `--preserve`	Préserver les attributs du fichier lorsque cela est possible
`-R` , `--recursive`	Copier récursivement des répertoires

Examples

Copier un seul fichier

Copiez `foo.txt` de `/path/to/source/` vers `/path/to/target/folder/`

```
cp /path/to/source/foo.txt /path/to/target/folder/
```

Copiez `foo.txt` de `/path/to/source/` vers `/path/to/target/folder/` dans un fichier appelé `bar.txt`

```
cp /path/to/source/foo.txt /path/to/target/folder/bar.txt
```

Copier des dossiers

copier le dossier `foo` dans la `bar` dossiers

```
cp -r /path/to/foo /path/to/bar
```

Si la barre de dossiers existe avant de lancer la commande, alors `foo` et son contenu seront copiés dans la `bar` dossiers. Cependant, si la `bar` n'existe pas avant de lancer la commande, la `bar` dossiers sera créée et le contenu de `foo` sera placé dans la `bar`

Lire Copier (cp) en ligne: https://riptutorial.com/fr/bash/topic/4030/copier--cp-

Chapitre 11: co-processus

Examples

Bonjour le monde

```
# create the co-process
coproc bash

# send a command to it (echo a)
echo 'echo Hello World' >&"${COPROC[1]}"

# read a line from its output
read line <&"${COPROC[0]}"

# show the line
echo "$line"
```

La sortie est "Hello World".

Lire co-processus en ligne: https://riptutorial.com/fr/bash/topic/6933/co-processus

Chapitre 12: Correspondance de motif et expressions régulières

Syntaxe

- $ shopt -u option # Désactiver l'option intégrée de Bash
- $ shopt -s option # Activer l'option intégrée de Bash

Remarques

Classes de personnages

Les classes de caractères valides pour le glob `[]` sont définies par le standard POSIX:

> alnum alpha ascii blank cntrl digit graphe inférieur imprimer espace ponctuel mot supérieur xdigit

À l'intérieur de `[]` plus d'une classe ou plage de caractères peut être utilisée, par exemple,

```
$ echo a[a-z[:blank:]0-9]*
```

correspondra à tout fichier commençant par `a` et suivi d'une lettre minuscule ou d'un blanc ou d'un chiffre.

Il faut toutefois garder à l'esprit qu'un `[]` glob ne peut être totalement nié et pas seulement des parties de celui-ci. Le caractère négatif *doit* être le premier caractère après l'ouverture `[` , par exemple, cette expression correspond à tous les fichiers qui ne commencent **pas** par `a`

```
$ echo [^a]*
```

Ce qui suit correspond à tous les fichiers qui commencent par un chiffre ou un `^`

```
$ echo [[:alpha:]^a]*
```

Il **ne** correspond à aucun fichier ou un dossier qui commence par la lettre , sauf un `a` parce que le `^` est interprété comme un littéral `^` .

Caractères de globes échappant

Il est possible qu'un fichier ou un dossier contienne un caractère glob dans le cadre de son nom. Dans ce cas, un glob peut être échappé avec un `\` précédent pour une correspondance littérale. Une autre approche consiste à utiliser `''` guillemets doubles `""` ou simples `''` pour traiter le fichier. Bash ne traite pas les globes entourés par `""` ou `''` .

Différence avec les expressions régulières

La différence la plus significative entre les globes et les expressions régulières est qu'une expression régulière valide nécessite un qualificatif ainsi qu'un quantificateur. Un qualificatif identifie *ce qui* doit correspondre et un quantificateur indique à quelle fréquence il doit correspondre au qualificatif. Le RegEx équivalent au `*` glob est `.*` Où `.` signifie n'importe quel caractère et `*` signifie zéro ou plusieurs correspondances du caractère précédent. L'équivalent RegEx pour le `?` glob est `.{1}` . Comme avant, le qualificatif `.` correspond à n'importe quel caractère et le `{1}` indique qu'il doit correspondre exactement au qualificateur précédent. Cela ne doit pas être confondu avec le `?` quantifier, qui correspond à zéro ou une fois dans un RegEx. Le `[]` glob est utilisable de la même manière dans un RegEx, à condition qu'il soit suivi d'un quantificateur obligatoire.

Expressions régulières équivalentes

Glob	RegEx
*	.*
?	.
[]	[]

Examples

Vérifier si une chaîne correspond à une expression régulière

3.0

Vérifiez si une chaîne contient exactement 8 chiffres:

```
$ date=20150624
$ [[ $date =~ ^[0-9]{8}$ ]] && echo "yes" || echo "no"
yes
$ date=hello
$ [[ $date =~ ^[0-9]{8}$ ]] && echo "yes" || echo "no"
no
```

Le * glob

Préparation

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

L'astérisque * est probablement le glob le plus utilisé. Il correspond simplement à n'importe quelle chaîne

```
$ echo *acy
macy stacy tracy
```

Un seul * ne correspondra pas aux fichiers et dossiers qui résident dans les sous-dossiers

```
$ echo *
emptyfolder folder macy stacy tracy
$ echo folder/*
folder/anotherfolder folder/subfolder
```

Le ** glob

4.0

Préparation

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -s globstar
```

Bash est capable d'interpréter deux astérisques adjacents en un seul glob. Avec l'option `globstar` activée, cela peut être utilisé pour faire correspondre les dossiers qui résident plus profondément dans la structure des répertoires.

```
echo **
emptyfolder folder folder/anotherfolder folder/anotherfolder/content
folder/anotherfolder/content/deepfolder folder/anotherfolder/content/deepfolder/file
folder/subfolder folder/subfolder/content folder/subfolder/content/deepfolder
folder/subfolder/content/deepfolder/file macy stacy tracy
```

Le `**` peut être considéré comme une extension de chemin, quelle que soit la profondeur du chemin. Cet exemple correspond à tout fichier ou dossier qui commence par `deep` , quelle que soit sa profondeur d'imbrication:

```
$ echo **/deep*
folder/anotherfolder/content/deepfolder folder/subfolder/content/deepfolder
```

Le ? glob

Préparation

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

Le `?` correspond simplement à un caractère

```
$ echo ?acy
macy
$ echo ??acy
stacy tracy
```

Le [] glob

Préparation

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

S'il est nécessaire de faire correspondre des caractères spécifiques, «[]» peut être utilisé. Tout caractère à l'intérieur de '[]' sera comparé exactement une fois.

```
$ echo [m]acy
macy
$ echo [st][tr]acy
stacy tracy
```

Le `[]` glob, cependant, est plus polyvalent que cela. Il permet également une correspondance négative et même des plages de caractères et de classes de caractères correspondantes. Une correspondance négative est obtenue en utilisant `!` ou `^` comme premier caractère suivant `[` . Nous pouvons faire correspondre `stacy` par

```
$ echo [!t][^r]acy
stacy
```

Nous disons ici bash que nous voulons faire correspondre uniquement les fichiers qui ne commencent pas par un `t` et que la deuxième lettre n'est pas un `r` et que le fichier se termine par

`acy` .

Les plages peuvent être comparées en séparant une paire de caractères avec un tiret (`-`). Tout caractère compris entre ces deux caractères - inclus - sera mis en correspondance. Par exemple, `[rt]` est équivalent à `[rst]`

```
$ echo [r-t][r-t]acy
stacy tracy
```

Les classes de caractères peuvent être appariées par `[:class:]` , par exemple, pour correspondre à des fichiers contenant un espace

```
$ echo *[[:blank:]]*
file with space
```

Faire correspondre les fichiers cachés

Préparation

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

L'option *intégrée* Bash *dotglob* permet de faire correspondre les fichiers et dossiers cachés, c'est-à-dire les fichiers et les dossiers qui commencent par un `.`

```
$ shopt -s dotglob
$ echo *
file with space folder .hiddenfile macy stacy tracy
```

Correspondance insensible à la casse

Préparation

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
```

```
$ shopt -u globstar
```

La définition de l'option `nocaseglob` correspondra au glob dans une `nocaseglob` insensible à la casse

```
$ echo M*
M*
$ shopt -s nocaseglob
$ echo M*
macy
```

Comportement lorsqu'un glob ne correspond à rien

Préparation

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

Si le glob ne correspond à rien, le résultat est déterminé par les options `nullglob` et `failglob` . Si aucun d'eux n'est défini, Bash retournera le glob lui-même si rien ne correspond

```
$ echo no*match
no*match
```

Si `nullglob` est activé, rien (`null`) n'est renvoyé:

```
$ shopt -s nullglob
$ echo no*match

$
```

Si `failglob` est activé, un message d'erreur est renvoyé:

```
$ shopt -s failglob
$ echo no*match
bash: no match: no*match
$
```

Notez que l'option `failglob` remplace l'option `nullglob` , c'est-à-dire que si `nullglob` et `failglob` sont tous deux définis, une erreur est renvoyée en cas de non-correspondance.

Globe étendu

2,02

Préparation

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

L'option `extglob` de `extglob` peut étendre les capacités de correspondance d'un glob

```
shopt -s extglob
```

Les sous-modèles suivants comprennent des globes étendus valides:

- `?(pattern-list)` - Correspond à zéro ou à une occurrence des motifs donnés
- `*(pattern-list)` - Correspond à zéro ou plusieurs occurrences des patterns donnés
- `+(pattern-list)` - Correspond à une ou plusieurs occurrences des patterns donnés
- `@(pattern-list)` - Correspond à l'un des motifs donnés
- `!(pattern-list)` - Correspond à tout sauf un des motifs donnés

La `pattern-list` est une liste de globes séparés par `|` .

```
$ echo *([r-t])acy
stacy tracy

$ echo *([r-t]|m)acy
macy stacy tracy

$ echo ?([a-z])acy
macy
```

La `pattern-list` même peut être un autre glob étendu imbriqué. Dans l'exemple ci-dessus, nous avons vu que nous pouvons faire correspondre `tracy` et `stacy` avec `*(rt)` . Ce glob étendu lui-même peut être utilisé à l'intérieur du glob étendu négatif `!(pattern-list)` afin de correspondre à `macy`

```
$ echo !(*([r-t]))acy
macy
```

Il correspond à tout ce qui ne commence **pas** par zéro ou plusieurs occurrences des lettres `r` , `s` et `t` , ce qui ne laisse que la correspondance possible entre `macy` .

Regex correspondant

```
pat='[^0-9]+([0-9]+)'
s='I am a string with some digits 1024'
```

```
[[ $s =~ $pat ]] # $pat must be unquoted
echo "${BASH_REMATCH[0]}"
echo "${BASH_REMATCH[1]}"
```

Sortie:

```
I am a string with some digits 1024
1024
```

Au lieu d'affecter la regex à une variable (`$pat`), nous pourrions aussi faire:

```
[[ $s =~ [^0-9]+([0-9]+) ]]
```

Explication

- La construction `[[ $s =~ $pat ]]` effectue la correspondance de regex
- Les groupes capturés, c'est-à-dire les résultats de la correspondance, sont disponibles dans un tableau nommé *BASH_REMATCH.*
- Le 0ème index du tableau *BASH_REMATCH* correspond à la correspondance totale
- Le *i´eme* index du tableau *BASH_REMATCH* est le *i´eme* groupe capturé, où *i* = 1, 2, 3 ...

Obtenir des groupes capturés à partir d'une correspondance regex avec une chaîne

```
a='I am a simple string with digits 1234'
pat='(.*) ([0-9]+)'
[[ "$a" =~ $pat ]]
echo "${BASH_REMATCH[0]}"
echo "${BASH_REMATCH[1]}"
echo "${BASH_REMATCH[2]}"
```

Sortie:

```
I am a simple string with digits 1234
I am a simple string with digits
1234
```

Lire Correspondance de motif et expressions régulières en ligne:
https://riptutorial.com/fr/bash/topic/3795/correspondance-de-motif-et-expressions-regulieres

Chapitre 13: Couper la commande

Introduction

Dans Bash, la commande `cut` est utile pour diviser un fichier en plusieurs parties plus petites.

Syntaxe

- couper le fichier [option]

Paramètres

Option	La description
`-b LIST , --bytes=LIST`	Imprimer les octets répertoriés dans le paramètre LIST
`-c LIST , --characters=LIST`	Imprimer les caractères dans les positions spécifiées dans le paramètre LIST
`-f LIST , --fields=LIST`	Champs d'impression ou colonnes
`-d DELIMITER`	Utilisé pour séparer les colonnes ou les champs

Examples

Afficher la première colonne d'un fichier

Supposons que vous ayez un fichier qui ressemble à ceci

```
John Smith 31
Robert Jones 27
...
```

Ce fichier a 3 colonnes séparées par des espaces. Pour sélectionner uniquement la première colonne, procédez comme suit.

```
cut -d ' ' -f1 filename
```

Ici, le drapeau `-d` spécifie le délimiteur ou ce qui sépare les enregistrements. L'indicateur `-f` spécifie le numéro de champ ou de colonne. Cela affichera la sortie suivante

```
John
Robert
...
```

Afficher les colonnes x à y d'un fichier

Parfois, il est utile d'afficher une série de colonnes dans un fichier. Supposons que vous ayez ce fichier

```
Apple California 2017 1.00 47
Mango Oregon 2015 2.30 33
```

Pour sélectionner les 3 premières colonnes à faire

```
cut -d ' ' -f1-3 filename
```

Cela affichera la sortie suivante

```
Apple California 2017
Mango Oregon 2015
```

Lire Couper la commande en ligne: https://riptutorial.com/fr/bash/topic/9138/couper-la-commande

Chapitre 14: Création de répertoires

Introduction

Manipulation des répertoires à partir de la ligne de commande

Examples

Déplacer tous les fichiers qui ne figurent pas déjà dans un répertoire dans un répertoire auto-nommé

ll | grep ^ - | awk -F "." '{imprimer $ 2 "." $ 3} '| awk -F ":" '{print $ 2}' | awk '{$ 1 = ""; imprimer $ 0} '| couper -c2- | awk -F "." '{print "mkdir" "$ 1" "; mv" "$ 1". "$ 2" "" "$ 1" ""}'> tmp; source tmp

Lire Création de répertoires en ligne: https://riptutorial.com/fr/bash/topic/8168/creation-de-repertoires

Chapitre 15: Déclaration de cas

Examples

Déclaration de cas simple

Dans sa forme la plus simple prise en charge par toutes les versions de bash, case statement exécute la casse correspondant au modèle. `;;` l'opérateur casse après le premier match, le cas échéant.

```
#!/bin/bash

var=1
case $var in
1)
  echo "Antartica"
 ;;
2)
  echo "Brazil"
 ;;
3)
  echo "Cat"
 ;;
esac
```

Les sorties:

```
Antartica
```

Déclaration de cas avec chute

4.0

Depuis bash 4.0, un nouvel opérateur `;&` été introduit qui fournit un mécanisme de chute .

#! / bin / bash

```
var=1
case $var in
1)
  echo "Antartica"
  ;&
2)
  echo "Brazil"
  ;&
3)
  echo "Cat"
  ;&
esac
```

Les sorties:

```
Antartica
Brazil
Cat
```

Ne tomber que si les modèles suivants correspondent

4.0

Depuis Bash 4.0, un autre opérateur `;;&` été introduit, qui fournit également la possibilité de passer à travers *si* les modèles dans les déclarations de cas suivantes, le cas échéant, correspondent.

```
#!/bin/bash

var=abc
case $var in
a*)
  echo "Antartica"
  ;;&
xyz)
  echo "Brazil"
  ;;&
*b*)
  echo "Cat"
  ;;&
esac
```

Les sorties:

```
Antartica
Cat
```

Dans l'exemple ci-dessous, l' `abc` correspond au premier et au troisième cas, mais pas au second. Donc, le deuxième cas n'est pas exécuté.

Lire Déclaration de cas en ligne: https://riptutorial.com/fr/bash/topic/5237/declaration-de-cas

Chapitre 16: Emplois à des moments précis

Examples

Exécuter le travail une fois à une heure précise

Remarque: **at** n'est pas installé par défaut sur la plupart des distributions modernes.

Pour exécuter un travail une fois à un autre moment que maintenant, dans cet exemple, vous pouvez utiliser

```
echo "somecommand &" | at 5pm
```

Si vous voulez attraper la sortie, vous pouvez le faire de la manière habituelle:

```
echo "somecommand > out.txt 2>err.txt &" | at 5pm
```

`at` comprendre de nombreux formats de temps, de sorte que vous pouvez également dire

```
echo "somecommand &" | at now + 2 minutes
echo "somecommand &" | at 17:00
echo "somecommand &" | at 17:00 Jul 7
echo "somecommand &" | at 4pm 12.03.17
```

Si aucune année ou date n'est donnée, cela suppose la prochaine fois que l'heure spécifiée se produit. Donc, si vous donnez une heure qui a déjà passé aujourd'hui, cela prendra demain, et si vous donnez un mois qui a déjà passé cette année, cela prendra l'année prochaine.

Cela fonctionne également avec Nohup comme vous vous en doutez.

```
echo "nohup somecommand > out.txt 2>err.txt &" | at 5pm
```

Il existe d'autres commandes pour contrôler les tâches temporisées:

- **atq** liste tous les travaux temporisés (**atq** ueue)
- **atrm** supprime un travail chronométré (**atr** e **m** ove)
- **batch** fait pratiquement la même chose qu'à, mais exécute les travaux uniquement lorsque la charge du système est inférieure à 0,8

Toutes les commandes s'appliquent aux tâches de l'utilisateur connecté. Si vous êtes connecté en tant que root, les tâches système sont bien entendu traitées.

Faire des travaux à des moments spécifiés à plusieurs reprises en utilisant systemd.timer

systemd fournit une implémentation moderne de **cron** . Pour exécuter un script périodique, un

service et un fichier de minuterie sont nécessaires. Les fichiers de service et de temporisation doivent être placés dans / etc / systemd / {system, user}. Le fichier de service:

```
[Unit]
Description=my script or programm does the very best and this is the description

[Service]
# type is important!
Type=simple
# program|script to call. Always use absolute pathes
# and redirect STDIN and STDERR as there is no terminal while being executed
ExecStart=/absolute/path/to/someCommand >>/path/to/output 2>/path/to/STDERRoutput
#NO install section!!!! Is handled by the timer facitlities itself.
#[Install]
#WantedBy=multi-user.target
```

Suivant le fichier de minuterie:

```
[Unit]
Description=my very first systemd timer
[Timer]
# Syntax for date/time specifications is  Y-m-d H:M:S
# a * means "each", and a comma separated list of items can be given too
# *-*-* *,15,30,45:00  says every year, every month, every day, each hour,
# at minute 15,30,45 and zero seconds

OnCalendar=*-*-* *:01:00
# this one runs each hour at one minute zero second e.g. 13:01:00
```

Lire Emplois à des moments précis en ligne: https://riptutorial.com/fr/bash/topic/7283/emplois-a-des-moments-precis

Chapitre 17: Emplois et processus

Examples

Liste des travaux en cours

```
$ tail -f /var/log/syslog > log.txt
[1]+  Stopped                 tail -f /var/log/syslog > log.txt

$ sleep 10 &

$ jobs
[1]+  Stopped                 tail -f /var/log/syslog > log.txt
[2]-  Running                 sleep 10 &
```

Gestion des travaux

Créer des emplois

Pour créer un travail, ajoutez juste un seul `&` après la commande:

```
$ sleep 10 &
[1] 20024
```

Vous pouvez également effectuer un traitement en cours en appuyant sur `Ctrl` + `z` :

```
$ sleep 10
^Z
[1]+  Stopped                 sleep 10
```

Arrière-plan et avant-plan d'un processus

Pour amener le processus au premier plan, la commande `fg` est utilisée avec `%`

```
$ sleep 10 &
[1] 20024

$ fg %1
sleep 10
```

Maintenant, vous pouvez interagir avec le processus. Pour le ramener en arrière-plan, vous pouvez utiliser la commande `bg` . En raison de la session de terminal occupée, vous devez d'abord arrêter le processus en appuyant sur `Ctrl` + `z` .

```
$ sleep 10
```

```
^Z
[1]+  Stopped                 sleep 10

$ bg %1
[1]+ sleep 10 &
```

En raison de la paresse de certains programmeurs, toutes ces commandes fonctionnent également avec un seul `%` s'il n'y a qu'un processus ou pour le premier processus de la liste. Par exemple:

```
$ sleep 10 &
[1] 20024

$ fg %        # to bring a process to foreground 'fg %' is also working.
sleep 10
```

ou juste

```
$ %           # laziness knows no boundaries, '%' is also working.
sleep 10
```

De plus, il suffit de taper `fg` ou `bg` sans aucun argument pour gérer le dernier travail:

```
$ sleep 20 &
$ sleep 10 &
$ fg
sleep 10
^C
$ fg
sleep 20
```

Tuer des emplois en cours d'exécution

```
$ sleep 10 &
[1] 20024

$ kill %1
[1]+  Terminated              sleep 10
```

Le processus de veille s'exécute en arrière-plan avec l'ID de processus (pid) `20024` et le travail numéro `1` . Pour référencer le processus, vous pouvez utiliser soit le pid, soit le numéro de travail. Si vous utilisez le numéro de travail, vous devez le préfixer avec `%` . Le signal de suppression par défaut envoyé par `kill` est `SIGTERM` , ce qui permet au processus cible de sortir normalement.

Certains signaux de suppression courants sont présentés ci-dessous. Pour voir une liste complète, exécutez `kill -l` .

Nom du signal	Valeur du signal	Effet
`SIGHUP`	`1`	Raccrocher

Nom du signal	Valeur du signal	Effet
SIGINT	2	Interrompre depuis le clavier
SIGKILL	9	Tuer le signal
SIGTERM	15	Signal de terminaison

Démarrer et tuer des processus spécifiques

Le moyen le plus simple de tuer un processus en cours est de le sélectionner via le nom du processus, comme dans l'exemple suivant, en utilisant la commande `pkill` comme

```
pkill -f test.py
```

(ou) un moyen plus `pgrep` utiliser `pgrep` pour rechercher l'ID de processus réel

```
kill $(pgrep -f 'python test.py')
```

Le même résultat peut être obtenu en utilisant `grep` sur `ps -ef | grep name_of_process` puis en `ps -ef | grep name_of_process` le processus associé au pid résultant (identificateur de processus). La sélection d'un processus utilisant son nom est pratique dans un environnement de test mais peut être très dangereuse lorsque le script est utilisé en production: il est pratiquement impossible d'être sûr que le nom correspond au processus que vous souhaitez réellement tuer. Dans ces cas, l'approche suivante est en réalité beaucoup plus sûre.

Démarrez le script qui sera finalement tué avec l'approche suivante. Supposons que la commande que vous voulez exécuter et éventuellement tuer soit `python test.py`

```
#!/bin/bash

if [[ ! -e /tmp/test.py.pid ]]; then   # Check if the file already exists
    python test.py &                   #+and if so do not run another process.
    echo $! > /tmp/test.py.pid
else
    echo -n "ERROR: The process is already running with pid "
    cat /tmp/test.py.pid
    echo
fi
```

Cela créera un fichier dans le `/tmp` contenant le pid du processus `python test.py` Si le fichier existe déjà, nous supposons que la commande est déjà en cours d'exécution et que le script renvoie une erreur.

Ensuite, lorsque vous voulez le tuer, utilisez le script suivant:

```
#!/bin/bash

if [[ -e /tmp/test.py.pid ]]; then   # If the file do not exists, then the
```

```
    kill `cat /tmp/test.py.pid`      #+the process is not running. Useless
    rm /tmp/test.py.pid              #+trying to kill it.
else
    echo "test.py is not running"
fi
```

cela tuera exactement le processus associé à votre commande, sans compter sur aucune information volatile (comme la chaîne utilisée pour exécuter la commande). Même dans ce cas, si le fichier n'existe pas, le script suppose que vous souhaitez supprimer un processus non en cours d'exécution.

Ce dernier exemple peut être facilement amélioré pour exécuter la même commande plusieurs fois (en ajoutant le fichier pid au lieu de le remplacer, par exemple) et pour gérer les cas où le processus meurt avant d'être tué.

Liste tous les processus

Il existe deux méthodes courantes pour répertorier tous les processus sur un système. Tous deux listent tous les processus en cours d'exécution par tous les utilisateurs, bien qu'ils diffèrent par le format qu'ils produisent (la raison de ces différences est historique).

```
ps -ef   # lists all processes
ps aux   # lists all processes in alternative format (BSD)
```

Cela peut être utilisé pour vérifier si une application donnée est en cours d'exécution. Par exemple, pour vérifier si le serveur SSH (sshd) est en cours d'exécution:

```
ps -ef | grep sshd
```

Vérifier quel processus s'exécute sur un port spécifique

Pour vérifier quel processus s'exécute sur le port 8080

```
lsof -i :8080
```

Recherche d'informations sur un processus en cours d'exécution

`ps aux | grep <search-term>` affiche les processus correspondant au *terme de recherche*

Exemple:

```
root@server7:~# ps aux | grep nginx
root       315  0.0  0.3 144392  1020 ?        Ss   May28   0:00 nginx: master process
/usr/sbin/nginx
www-data  5647  0.0  1.1 145124  3048 ?        S    Jul18   2:53 nginx: worker process
www-data  5648  0.0  0.1 144392   376 ?        S    Jul18   0:00 nginx: cache manager process
root     13134  0.0  0.3   4960   920 pts/0    S+   14:33   0:00 grep --color=auto nginx
root@server7:~#
```

Ici, la deuxième colonne est l'identifiant du processus. Par exemple, si vous voulez tuer le processus nginx, vous pouvez utiliser la commande `kill 5647` . Il est toujours conseillé d'utiliser la commande `kill` avec `SIGTERM` plutôt que `SIGKILL` .

Job de fond

```
$ gzip extremelylargefile.txt &
$ bg
$ disown %1
```

Cela permet à un processus de longue durée de continuer une fois que votre shell (terminal, ssh, etc.) est fermé.

Lire Emplois et processus en ligne: https://riptutorial.com/fr/bash/topic/398/emplois-et-processus

Chapitre 18: En utilisant chat

Syntaxe

- chat [OPTIONS] ... [FICHIER] ...

Paramètres

Option	Détails
-n	Imprimer les numéros de ligne
-v	Afficher les caractères non imprimables à l'aide de la notation ^ et M, sauf LFD et TAB
-T	Affiche les caractères de tabulation comme ^ I
-E	Afficher les caractères de saut de ligne (LF) comme $
-e	Identique à -vE
-b	Nombre de lignes de sortie non vides, remplace -n
-UNE	équivalent à -vET
-s	supprimer les lignes de sortie vides répétées, s se réfère à squeeze

Remarques

`cat` peut lire à la fois des fichiers et des entrées standard et les concaténer à la sortie standard

Examples

Impression du contenu d'un fichier

```
cat file.txt
```

imprimera le contenu d'un fichier.

Si le fichier contient des caractères non-ASCII, vous pouvez afficher ces caractères symboliquement avec `cat -v` . Cela peut être très utile dans les situations où des caractères de contrôle seraient autrement invisibles.

```
cat -v unicode.txt
```

Très souvent, pour un usage interactif, il est préférable d'utiliser un téléavertisseur interactif `less` ou `more` . (`less` est beaucoup plus puissant que `more` et il est conseillé d'utiliser `less` souvent que `more` .)

```
less file.txt
```

Transmettre le contenu d'un fichier en entrée à une commande. Une approche généralement considérée comme meilleure (UUOC) consiste à utiliser la redirection.

```
tr A-Z a-z <file.txt   # as an alternative to cat file.txt | tr A-Z a-z
```

Si le contenu doit être répertorié en arrière à partir de sa fin, la commande `tac` peut être utilisée:

```
tac file.txt
```

Si vous souhaitez imprimer le contenu avec des numéros de ligne, utilisez `-n` avec `cat` :

```
cat -n file.txt
```

Pour afficher le contenu d'un fichier sous une forme octet par octet sans ambiguïté, un vidage hexadécimal est la solution standard. Ceci est utile pour des extraits très courts d'un fichier, par exemple lorsque vous ne connaissez pas le codage précis. L'utilitaire de vidage hexadécimal standard est `od -cH` , bien que la représentation soit légèrement lourde; les remplacements courants incluent `xxd` et `hexdump` .

```
$ printf 'Hëllö wörld' | xxd
0000000: 48c3 ab6c 6cc3 b620 77c3 b672 6c64       H..ll.. w..rld
```

Afficher les numéros de ligne avec sortie

Utilisez le drapeau `--number` pour imprimer les numéros de ligne avant chaque ligne. Sinon, `-n` fait la même chose.

```
$ cat --number file

 1  line 1
 2  line 2
 3
 4  line 4
 5  line 5
```

Pour ignorer les lignes vides lors du comptage de lignes, utilisez `--number-nonblank` , ou simplement `-b` .

```
$ cat -b file

 1  line 1
 2  line 2
```

```
 3  line 4
 4  line 5
```

Lire depuis l'entrée standard

```
cat < file.txt
```

La sortie est identique à celle de `cat file.txt` , mais elle lit le contenu du fichier à partir de l'entrée standard et non directement à partir du fichier.

```
printf "first line\nSecond line\n" | cat -n
```

La commande echo avant le `|` génère deux lignes. La commande cat agit sur la sortie pour ajouter des numéros de ligne.

Concaténer des fichiers

C'est le but principal du `cat` .

```
cat file1 file2 file3 > file_all
```

`cat` peut également être utilisé de manière similaire pour concaténer des fichiers dans le cadre d'un pipeline, par exemple

```
cat file1 file2 file3 | grep foo
```

Ecrire dans un fichier

```
cat >file
```

Il vous permettra d'écrire le texte sur le terminal qui sera enregistré dans un fichier nommé *fichier* .

```
cat >>file
```

fera de même, sauf qu'il ajoutera le texte à la fin du fichier.

NB: `Ctrl + D` pour terminer l'écriture du texte sur le terminal (Linux)

Un document ici peut être utilisé pour incorporer le contenu d'un fichier dans une ligne de commande ou un script:

```
cat <<END >file
Hello, World.
END
```

Le jeton après le symbole `<<` redirection est une chaîne arbitraire qui doit apparaître seule sur une

ligne (sans espace avant ou arrière) pour indiquer la fin du document ici. Vous pouvez ajouter des guillemets pour empêcher le shell d'effectuer une substitution de commande et une interpolation de variable:

```
cat <<'fnord'
Nothing in `here` will be $changed
fnord
```

(Sans les guillemets, `here` serait exécuté comme une commande, et `$changed` serait remplacé par la valeur de la variable `changed` - ou rien si elle était indéfinie.)

Afficher les caractères non imprimables

Ceci est utile pour voir s'il existe des caractères non imprimables ou des caractères non-ASCII.

Par exemple, si vous avez copié le code depuis le Web, vous pouvez avoir des guillemets comme ″ au lieu de standard " .

```
$ cat -v file.txt
$ cat -vE file.txt # Useful in detecting trailing spaces.
```

par exemple

```
$ echo '″     ' | cat -vE # echo | will be replaced by actual file.
M-bM-^@M-^]     $
```

Vous pouvez également utiliser le `cat -A` (A pour Tous) équivalent à `cat -vET` . Il affichera les caractères TAB (affichés comme `^I`), les caractères non imprimables et la fin de chaque ligne:

```
$ echo '″ `' | cat -A
M-bM-^@M-^]^I`$
```

Concaténer des fichiers compressés

Les fichiers compressés par `gzip` peuvent être directement concaténés en plus gros fichiers compressés.

```
cat file1.gz file2.gz file3.gz > combined.gz
```

C'est une propriété de `gzip` moins efficace que la concaténation des fichiers d'entrée et la compression du résultat:

```
cat file1 file2 file3 | gzip > combined.gz
```

Une démonstration complète:

```
echo 'Hello world!' > hello.txt
echo 'Howdy world!' > howdy.txt
```

```
gzip hello.txt
gzip howdy.txt

cat hello.txt.gz howdy.txt.gz > greetings.txt.gz

gunzip greetings.txt.gz

cat greetings.txt
```

Qui se traduit par

```
Hello world!
Howdy world!
```

Notez que `greetings.txt.gz` est un ***fichier unique*** et est décompressé en tant que ***fichier unique*** `greeting.txt` . Comparez ceci avec `tar -czf hello.txt howdy.txt > greetings.tar.gz` , qui garde les fichiers séparés à l'intérieur de l'archive.

Lire En utilisant chat en ligne: https://riptutorial.com/fr/bash/topic/441/en-utilisant-chat

Chapitre 19: En utilisant le tri

Introduction

sort est une commande Unix permettant de classer les données dans un ou plusieurs fichiers dans une séquence.

Syntaxe

- sort [option] filename

Paramètres

Option	Sens
-u	Rendre chaque ligne de sortie unique

Remarques

Manuel d'utilisation complet du `sort` ligne

Examples

Sorte de la commande de tri

`sort` commande de `sort` est utilisée pour trier une liste de lignes.

Entrée d'un fichier

```
sort file.txt
```

Entrée d'une commande

Vous pouvez trier toute commande de sortie. Dans l'exemple une liste de fichiers suivant un pattern.

```
find * -name pattern | sort
```

Rendre la sortie unique

Si chaque ligne de la sortie doit être unique, ajoutez l'option `-u` .

Pour afficher le propriétaire des fichiers dans le dossier

```
ls -l | awk '{print $3}' | sort -u
```

Tri numérique

Supposons que nous ayons ce fichier:

```
test>>cat file
10.Gryffindor
4.Hogwarts
2.Harry
3.Dumbledore
1.The sorting hat
```

Pour trier ce fichier numériquement, utilisez l'option de tri avec -n:

```
test>>sort -n file
```

Cela devrait trier le fichier comme ci-dessous:

```
1.The sorting hat
2.Harry
3.Dumbledore
4.Hogwarts
10.Gryffindor
```

Inverser l'ordre de tri: Pour inverser l'ordre du tri, utilisez l'option -r

Pour inverser l'ordre de tri du fichier ci-dessus, utilisez:

```
sort -rn file
```

Cela devrait trier le fichier comme ci-dessous:

```
10.Gryffindor
4.Hogwarts
3.Dumbledore
2.Harry
1.The sorting hat
```

Trier par clés

Supposons que nous ayons ce fichier:

```
test>>cat Hogwarts
  Harry        Malfoy        Rowena        Helga
  Gryffindor   Slytherin     Ravenclaw     Hufflepuff
  Hermione     Goyle         Lockhart      Tonks
  Ron          Snape         Olivander     Newt
  Ron          Goyle         Flitwick      Sprout
```

Pour trier ce fichier en utilisant une colonne comme clé, utilisez l'option k:

```
test>>sort -k 2 Hogwarts
```

Cela va trier le fichier avec la colonne 2 comme clé:

```
  Ron         Goyle         Flitwick         Sprout
  Hermione    Goyle         Lockhart         Tonks
  Harry       Malfoy        Rowena           Helga
  Gryffindor  Slytherin     Ravenclaw        Hufflepuff
  Ron         Snape         Olivander        Newt
```

Maintenant, si nous devons trier le fichier avec une clé secondaire avec l'utilisation de la clé primaire:

```
sort -k 2,2 -k 1,1 Hogwarts
```

Cela va d'abord trier le fichier avec la colonne 2 comme clé primaire, puis trier le fichier avec la colonne 1 comme clé secondaire:

```
  Hermione         Goyle         Lockhart         Tonks
  Ron              Goyle         Flitwick         Sprout
  Harry            Malfoy        Rowena           Helga
  Gryffindor       Slytherin     Ravenclaw        Hufflepuff
  Ron              Snape         Olivander        Newt
```

Si nous avons besoin de trier un fichier avec plus d'une clé, nous devons spécifier pour chaque option -k où le tri se termine. Donc -k1,1 signifie que vous devez commencer le tri dans la première colonne et terminer le tri dans la première colonne.

option -t

Dans l'exemple précédent, le fichier avait le séparateur par défaut - tab. En cas de tri d'un fichier comportant un délimiteur autre que celui par défaut, l'option -t doit être utilisée pour spécifier le séparateur. Supposons que nous ayons le fichier ci-dessous:

```
test>>cat file
5.|Gryffindor
4.|Hogwarts
2.|Harry
3.|Dumbledore
1.|The sorting hat
```

Pour trier ce fichier selon la deuxième colonne, utilisez:

```
test>>sort -t "|" -k 2 file
```

Cela va trier le fichier comme ci-dessous:

```
3.|Dumbledore
5.|Gryffindor
2.|Harry
4.|Hogwarts
```

```
1.|The sorting hat
```

Lire En utilisant le tri en ligne: https://riptutorial.com/fr/bash/topic/6834/en-utilisant-le-tri

Chapitre 20: Espace de noms

Examples

Il n'y a pas de choses comme les espaces de noms

```
myfunc(){
    echo "I will never be executed."
}
another_func(){
    # this "redeclare" overwrites original function
    myfunc(){ echo "I am the one and only"; }
}
# myfunc will print "I will never be executed"
myfunc
# but if we call another_func first
another_func
# it gets overwritten and
myfunc
# no prints "I am the one and only"
```

La dernière déclaration gagne. Il n'y a pas de choses comme les espaces de noms! Cependant, les fonctions peuvent contenir d'autres fonctions.

Lire Espace de noms en ligne: https://riptutorial.com/fr/bash/topic/6835/espace-de-noms

Chapitre 21: Éviter la date en utilisant printf

Introduction

Dans Bash 4.2, une conversion temporelle intégrée de shell pour `printf` été introduite: la spécification de format `%(datefmt)T` permet à `printf` sortir la chaîne date-heure correspondant à la chaîne de format `datefmt` telle que comprise par `strftime` .

Syntaxe

- printf '% (dateFmt) T' # dateFmt peut être n'importe quelle chaîne de format reconnue par strftime
- printf '% (dateFmt) T' -1 # -1 représente l'heure actuelle (par défaut pour aucun argument)
- printf '% (dateFmt) T' -2 # -2 représente l'heure à laquelle le shell a été appelé

Remarques

Utiliser `printf -v foo '%(...)T'` est identique à `foo=$(date +'...')` et enregistre une fourchette pour l'appel à la `date` programme externe.

Examples

Obtenez la date actuelle

```
$ printf '%(%F)T\n'
2016-08-17
```

Définir la variable à l'heure actuelle

```
$ printf -v now '%(%T)T'
$ echo "$now"
12:42:47
```

Lire Éviter la date en utilisant printf en ligne: https://riptutorial.com/fr/bash/topic/5522/eviter-la-date-en-utilisant-printf

Chapitre 22: Expansion Brace

Remarques

Manuel de référence Bash: Extension Brace

Examples

Créer des répertoires pour regrouper les fichiers par mois et par année

```
$ mkdir 20{09..11}-{01..12}
```

La saisie de la commande `ls` que les répertoires suivants ont été créés:

```
2009-01 2009-04 2009-07 2009-10 2010-01 2010-04 2010-07 2010-10 2011-01 2011-04 2011-07 2011-
10
2009-02 2009-05 2009-08 2009-11 2010-02 2010-05 2010-08 2010-11 2011-02 2011-05 2011-08 2011-
11
2009-03 2009-06 2009-09 2009-12 2010-03 2010-06 2010-09 2010-12 2011-03 2011-06 2011-09 2011-
12
```

Mettre un `0` devant `9` dans l'exemple garantit que les nombres sont remplis avec un seul `0` . Vous pouvez également composer des numéros avec plusieurs zéros, par exemple:

```
$ echo {001..10}
001 002 003 004 005 006 007 008 009 010
```

Créer une sauvegarde des fichiers dot

```
$ cp .vimrc{,.bak}
```

Cela se développe dans la commande `cp .vimrc .vimrc.bak` .

Modification de l'extension du nom de fichier

```
$ mv filename.{jar,zip}
```

Cela se développe en `mv filename.jar filename.zip` .

Utiliser des incréments

```
$ echo {0..10..2}
0 2 4 6 8 10
```

Un troisième paramètre pour spécifier un incrément, à savoir `{start..end..increment}`

L'utilisation d'incréments n'est pas limitée aux nombres

```
$ for c in {a..z..5}; do echo -n $c; done
afkpuz
```

Utilisation de l'extension d'accolades pour créer des listes

Bash peut facilement créer des listes à partir de caractères alphanumériques.

```
# list from a to z
$ echo {a..z}
a b c d e f g h i j k l m n o p q r s t u v w x y z

# reverse from z to a
$ echo {z..a}
z y x w v u t s r q p o n m l k j i h g f e d c b a

# digits
$ echo {1..20}
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

# with leading zeros
$ echo {01..20}
01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20

# reverse digit
$ echo {20..1}
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

# reversed with leading zeros
$ echo {20..01}
20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01

# combining multiple braces
$ echo {a..d}{1..3}
a1 a2 a3 b1 b2 b3 c1 c2 c3 d1 d2 d3
```

L'extension d'accolade est la toute première extension qui a lieu, elle ne peut donc être combinée à aucune autre extension.

Seuls les caractères et les chiffres peuvent être utilisés.

Cela ne fonctionnera pas: `echo {$(date +$H)..24}`

Faire plusieurs répertoires avec des sous-répertoires

```
mkdir -p toplevel/sublevel_{01..09}/{child1,child2,child3}
```

Cela créera un dossier de niveau supérieur appelé `toplevel` , neuf dossiers à l'intérieur de `toplevel` nommés `sublevel_01` , `sublevel_02` , etc. Puis à l'intérieur de ces sous-niveaux: `child1` , `child2` , `child3` , vous donnant:

```
toplevel/sublevel_01/child1
```

```
toplevel/sublevel_01/child2
toplevel/sublevel_01/child3
toplevel/sublevel_02/child1
```

etc. Je trouve cela très utile pour créer plusieurs dossiers et sous-dossiers à des fins spécifiques, avec une commande bash. Remplacez les variables pour aider à automatiser / analyser les informations fournies au script.

Lire Expansion Brace en ligne: https://riptutorial.com/fr/bash/topic/3351/expansion-brace

Chapitre 23: Expressions conditionnelles

Syntaxe

- [[-OP $ filename]]
- [[$ file1 -OP $ file2]]
- [[-z $ string]]
- [[-n $ string]]
- [["$ string1" == "$ string2"]]
- [["$ string1" == $ pattern]]

Remarques

La syntaxe `[[ … ]]` entoure les expressions conditionnelles intégrées bash. Notez que des espaces sont requis de chaque côté des crochets.

Les expressions conditionnelles peuvent utiliser des opérateurs unaires et binaires pour tester les propriétés des chaînes, des entiers et des fichiers. Ils peuvent également utiliser les opérateurs logiques `&&` , `||` et `!` .

Examples

Comparaison de fichier

```
if [[ $file1 -ef $file2 ]]; then
  echo "$file1 and $file2 are the same file"
fi
```

"Même fichier" signifie que la modification de l'un des fichiers en place affecte l'autre. Deux fichiers peuvent être identiques même s'ils ont des noms différents, par exemple s'il s'agit de liens matériels ou de liens symboliques avec la même cible, ou s'il s'agit d'un lien symbolique pointant vers l'autre.

Si deux fichiers ont le même contenu, mais qu'il s'agit de fichiers distincts (de sorte que la modification de l'un n'affecte pas l'autre), alors `-ef` signale comme étant différents. Si vous souhaitez comparer deux fichiers octet par octet, utilisez l'utilitaire `cmp` .

```
if cmp -s -- "$file1" "$file2"; then
  echo "$file1 and $file2 have identical contents"
else
  echo "$file1 and $file2 differ"
fi
```

Pour produire une liste lisible par l'homme des différences entre les fichiers texte, utilisez l'utilitaire `diff` .

```
if diff -u "$file1" "$file2"; then
  echo "$file1 and $file2 have identical contents"
else
  : # the differences between the files have been listed
fi
```

Tests d'accès aux fichiers

```
if [[ -r $filename ]]; then
  echo "$filename is a readable file"
fi
if [[ -w $filename ]]; then
  echo "$filename is a writable file"
fi
if [[ -x $filename ]]; then
  echo "$filename is an executable file"
fi
```

Ces tests prennent en compte les autorisations et la propriété pour déterminer si le script (ou les programmes lancés à partir du script) peuvent accéder au fichier.

Méfiez-vous des conditions de course (TOCTOU) : le fait que le test réussisse maintenant ne signifie pas qu'il est toujours valide sur la ligne suivante. Il est généralement préférable d'essayer d'accéder à un fichier et de traiter l'erreur plutôt que d'essayer d'abord et de gérer ensuite l'erreur au cas où le fichier aurait changé entre-temps.

Comparaisons numériques

Les comparaisons numériques utilisent les opérateurs et amis `-eq`

```
if [[ $num1 -eq $num2 ]]; then
  echo "$num1 == $num2"
fi
if [[ $num1 -le $num2 ]]; then
  echo "$num1 <= $num2"
fi
```

Il y a six opérateurs numériques:

- `-eq` égal
- `-ne` pas égal
- `-le` moins ou égal
- `-lt` moins que
- `-ge` plus grand ou égal
- `-gt` supérieur à

Notez que les opérateurs `<` et `>` intérieur de `[[ … ]]` comparent des chaînes, pas des nombres.

```
if [[ 9 -lt 10 ]]; then
  echo "9 is before 10 in numeric order"
fi
if [[ 9 > 10 ]]; then
```

```
  echo "9 is after 10 in lexicographic order"
fi
```

Les deux côtés doivent être des nombres écrits en décimal (ou en octal avec un zéro non significatif). Vous pouvez également utiliser la syntaxe d'expression arithmétique `((…))` , qui effectue **des** calculs d' **entiers** dans une syntaxe similaire à C / Java /….

```
x=2
if ((2*x == 4)); then
  echo "2 times 2 is 4"
fi
((x += 1))
echo "2 plus 1 is $x"
```

Comparaison de chaînes et correspondance

La comparaison de chaînes utilise l'opérateur `==` entre des chaînes entre *guillemets* . L'opérateur `!=` Annule la comparaison.

```
if [[ "$string1" == "$string2" ]]; then
  echo "\$string1 and \$string2 are identical"
fi
if [[ "$string1" != "$string2" ]]; then
  echo "\$string1 and \$string2 are not identical"
fi
```

Si le membre de droite n'est pas cité, c'est un modèle générique auquel `$string1` est associé.

```
string='abc'
pattern1='a*'
pattern2='x*'
if [[ "$string" == $pattern1 ]]; then
  # the test is true
  echo "The string $string matches the pattern $pattern"
fi
if [[ "$string" != $pattern2 ]]; then
  # the test is false
  echo "The string $string does not match the pattern $pattern"
fi
```

Les opérateurs `<` et `>` comparent les chaînes dans l'ordre lexicographique (il n'y a pas d'opérateur moins ou moins égal ou plus grand que les chaînes).

Il y a des tests unaires pour la chaîne vide.

```
if [[ -n "$string" ]]; then
  echo "$string is non-empty"
fi
if [[ -z "${string// }" ]]; then
  echo "$string is empty or contains only spaces"
fi
if [[ -z "$string" ]]; then
  echo "$string is empty"
```

```
fi
```

Au-dessus, la vérification `-z` peut signifier que `$string` n'est pas défini ou qu'elle est définie sur une chaîne vide. Pour distinguer entre vide et non défini, utilisez:

```
if [[ -n "${string+x}" ]]; then
    echo "$string is set, possibly to the empty string"
fi
if [[ -n "${string-x}" ]]; then
    echo "$string is either unset or set to a non-empty string"
fi
if [[ -z "${string+x}" ]]; then
    echo "$string is unset"
fi
if [[ -z "${string-x}" ]]; then
    echo "$string is set to an empty string"
fi
```

où `x` est arbitraire. Ou sous forme de tableau :

```
                       +-------+-------+-----------+
            $string is: | unset | empty | non-empty |
+----------------------+-------+-------+-----------+
| [[ -z ${string} ]]   | true  | true  | false     |
| [[ -z ${string+x} ]] | true  | false | false     |
| [[ -z ${string-x} ]] | false | true  | false     |
| [[ -n ${string} ]]   | false | false | true      |
| [[ -n ${string+x} ]] | false | true  | true      |
| [[ -n ${string-x} ]] | true  | false | true      |
+----------------------+-------+-------+-----------+
```

Sinon , l'état peut être vérifié dans une déclaration de cas:

```
case ${var+x$var} in
  (x) echo empty;;
  ("") echo unset;;
  (x*[![:blank:]]*) echo non-blank;;
  (*) echo blank
esac
```

Où `[:blank:]` est un caractère d'espacement horizontal spécifique aux paramètres régionaux (tabulation, espace, etc.).

Tests de type de fichier

L'opérateur conditionnel `-e` teste si un fichier existe (y compris tous les types de fichiers: répertoires, etc.).

```
if [[ -e $filename ]]; then
  echo "$filename exists"
fi
```

Il existe également des tests pour des types de fichiers spécifiques.

```
if [[ -f $filename ]]; then
  echo "$filename is a regular file"
elif [[ -d $filename ]]; then
  echo "$filename is a directory"
elif [[ -p $filename ]]; then
  echo "$filename is a named pipe"
elif [[ -S $filename ]]; then
  echo "$filename is a named socket"
elif [[ -b $filename ]]; then
  echo "$filename is a block device"
elif [[ -c $filename ]]; then
  echo "$filename is a character device"
fi
if [[ -L $filename ]]; then
  echo "$filename is a symbolic link (to any file type)"
fi
```

Pour un lien symbolique, hormis `-L` , ces tests s'appliquent à la cible et renvoient false pour un lien rompu.

```
if [[ -L $filename || -e $filename ]]; then
  echo "$filename exists (but may be a broken symbolic link)"
fi

if [[ -L $filename && ! -e $filename ]]; then
  echo "$filename is a broken symbolic link"
fi
```

Test sur l'état de sortie d'une commande

Statut de sortie 0: succès
Etat de sortie autre que 0: échec

Pour tester le statut de sortie d'une commande:

```
if command;then
    echo 'success'
else
    echo 'failure'
fi
```

Un test de ligne

Vous pouvez faire des choses comme ceci:

```
[[ $s = 'something' ]] && echo 'matched' || echo "didn't match"
[[ $s == 'something' ]] && echo 'matched' || echo "didn't match"
[[ $s != 'something' ]] && echo "didn't match" || echo "matched"
[[ $s -eq 10 ]] && echo 'equal' || echo "not equal"
(( $s == 10 )) && echo 'equal' || echo 'not equal'
```

Un test de ligne pour le statut de sortie:

```
command && echo 'exited with 0' || echo 'non 0 exit'
cmd && cmd1 && echo 'previous cmds were successful' || echo 'one of them failed'
cmd || cmd1 #If cmd fails try cmd1
```

Lire Expressions conditionnelles en ligne: https://riptutorial.com/fr/bash/topic/731/expressions-conditionnelles

Chapitre 24: Extension des paramètres Bash

Introduction

Le caractère `$` introduit une extension de paramètre, une substitution de commande ou une expansion arithmétique. Le nom ou le symbole du paramètre à développer peut être placé entre accolades, qui sont facultatifs mais servent à protéger la variable à étendre des caractères qui la suivent, ce qui pourrait être interprété comme faisant partie du nom.

En savoir plus dans le manuel de l'utilisateur de Bash .

Syntaxe

- $ {paramètre: offset} # Sous-chaîne commençant au décalage
- $ {paramètre: offset: longueur} # Sous-chaîne de longueur "longueur" commençant à l'offset
- $ {# paramètre} # Longueur du paramètre
- $ {parameter / pattern / string} # Remplace la première occurrence du motif par une chaîne
- $ {parameter // pattern / string} # Remplace toutes les occurrences du motif par une chaîne
- $ {paramètre / # modèle / chaîne} # Remplace le motif par une chaîne si le motif est au début
- $ {paramètre /% modèle / chaîne} # Remplace le motif par la chaîne si le motif est à la fin
- $ {parameter # pattern} # Supprime la correspondance la plus courte du pattern depuis le début du paramètre
- $ {parameter ## pattern} # Supprime la correspondance la plus longue du pattern depuis le début du paramètre
- $ {paramètre% pattern} # Supprime la correspondance la plus courte du motif de la fin du paramètre
- $ {paramètre %% pattern} # Supprime la plus longue correspondance du motif de la fin du paramètre
- $ {parameter: -word} # Développez le mot si le paramètre unset / undefined
- $ {parametre: = mot} # Etendre au mot si le paramètre unset / undefined et définir le paramètre
- $ {paramètre: + mot} # Développez en mot si jeu de paramètres / défini

Examples

Substrings et sous-réseaux

```
var='0123456789abcdef'

# Define a zero-based offset
$ printf '%s\n' "${var:3}"
3456789abcdef

# Offset and length of substring
```

```
$ printf '%s\n' "${var:3:4}"
3456
```

4.2

```
# Negative length counts from the end of the string
$ printf '%s\n' "${var:3:-5}"
3456789a

# Negative offset counts from the end
# Needs a space to avoid confusion with ${var:-6}
$ printf '%s\n' "${var: -6}"
abcdef

# Alternative: parentheses
$ printf '%s\n' "${var:(-6)}"
abcdef

# Negative offset and negative length
$ printf '%s\n' "${var: -6:-5}"
a
```

Les mêmes extensions s'appliquent si le paramètre est un **paramètre positionnel** ou l' **élément d'un tableau en indice** :

```
# Set positional parameter $1
set -- 0123456789abcdef

# Define offset
$ printf '%s\n' "${1:5}"
56789abcdef

# Assign to array element
myarr[0]='0123456789abcdef'

# Define offset and length
$ printf '%s\n' "${myarr[0]:7:3}"
789
```

Les expansions analogues s'appliquent aux **paramètres de position** , les décalages étant basés sur une base:

```
# Set positional parameters $1, $2, ...
$ set -- 1 2 3 4 5 6 7 8 9 0 a b c d e f

# Define an offset (beware $0 (not a positional parameter)
# is being considered here as well)
$ printf '%s\n' "${@:10}"
0
a
b
c
d
e
f

# Define an offset and a length
```

```
$ printf '%s\n' "${@:10:3}"
0
a
b

# No negative lengths allowed for positional parameters
$ printf '%s\n' "${@:10:-2}"
bash: -2: substring expression < 0

# Negative offset counts from the end
# Needs a space to avoid confusion with ${@:-10:2}
$ printf '%s\n' "${@: -10:2}"
7
8

# ${@:0} is $0 which is not otherwise a positional parameters or part
# of $@
$ printf '%s\n' "${@:0:2}"
/usr/bin/bash
1
```

L'extension de sous-chaîne peut être utilisée avec **les tableaux indexés** :

```
# Create array (zero-based indices)
$ myarr=(0 1 2 3 4 5 6 7 8 9 a b c d e f)

# Elements with index 5 and higher
$ printf '%s\n' "${myarr[@]:12}"
c
d
e
f

# 3 elements, starting with index 5
$ printf '%s\n' "${myarr[@]:5:3}"
5
6
7

# The last element of the array
$ printf '%s\n' "${myarr[@]: -1}"
f
```

Longueur du paramètre

```
# Length of a string
$ var='12345'
$ echo "${#var}"
5
```

Notez que c'est la longueur en nombre de *caractères* qui n'est pas nécessairement la même que le nombre d' *octets* (comme dans UTF-8 où la plupart des caractères sont encodés dans plus d'un octet), ni le nombre de *glyphes / graphèmes* combinaisons de caractères), ni nécessairement la même largeur d'affichage.

```
# Number of array elements
```

```
$ myarr=(1 2 3)
$ echo "${#myarr[@]}"
3

# Works for positional parameters as well
$ set -- 1 2 3 4
$ echo "${#@}"
4

# But more commonly (and portably to other shells), one would use
$ echo "$#"
4
```

Modifier la casse des caractères alphabétiques

4.0

En majuscule

```
$ v="hello"
# Just the first character
$ printf '%s\n' "${v^}"
Hello
# All characters
$ printf '%s\n' "${v^^}"
HELLO
# Alternative
$ v="hello world"
$ declare -u string="$v"
$ echo "$string"
HELLO WORLD
```

Pour minuscule

```
$ v="BYE"
# Just the first character
$ printf '%s\n' "${v,}"
bYE
# All characters
$ printf '%s\n' "${v,,}"
bye
# Alternative
$ v="HELLO WORLD"
$ declare -l string="$v"
$ echo "$string"
hello world
```

Toggle Case

```
$ v="Hello World"
# All chars
$ echo "${v~~}"
hELLO wORLD
$ echo "${v~}"
# Just the first char
hello World
```

Paramètre d'indirection

`Bash` indirection `Bash` permet d'obtenir la valeur d'une variable dont le nom est contenu dans une autre variable. Exemple de variables:

```
$ red="the color red"
$ green="the color green"

$ color=red
$ echo "${!color}"
the color red
$ color=green
$ echo "${!color}"
the color green
```

Quelques exemples supplémentaires illustrant l'utilisation de l'expansion indirecte:

```
 $ foo=10
 $ x=foo
 $ echo ${x}      #Classic variable print
 foo

 $ foo=10
 $ x=foo
 $ echo ${!x}     #Indirect expansion
 10
```

Un autre exemple:

```
$ argtester () { for (( i=1; i<="$#"; i++ )); do echo "${i}";done; }; argtester -ab -cd -ef
1   #i expanded to 1
2   #i expanded to 2
3   #i expanded to 3

$ argtester () { for (( i=1; i<="$#"; i++ )); do echo "${!i}";done; }; argtester -ab -cd -ef
-ab     # i=1 --> expanded to $1 ---> expanded to first argument sent to function
-cd     # i=2 --> expanded to $2 ---> expanded to second argument sent to function
-ef     # i=3 --> expanded to $3 ---> expanded to third argument sent to function
```

Substitution de valeur par défaut

```
${parameter:-word}
```

Si le paramètre n'est pas défini ou nul, l'expansion du mot est substituée. Sinon, la valeur du paramètre est substituée.

```
$ unset var
$ echo "${var:-XX}"     # Parameter is unset -> expansion XX occurs
XX
$ var=""                # Parameter is null -> expansion XX occurs
$ echo "${var:-XX}"
XX
$ var=23                # Parameter is not null -> original expansion occurs
```

```
$ echo "${var:-XX}"
23
```

```
${parameter:=word}
```

Si le paramètre est non défini ou nul, l'expansion du mot est affectée au paramètre. La valeur du paramètre est alors substituée. Les paramètres de position et les paramètres spéciaux ne peuvent pas être affectés de cette manière.

```
$ unset var
$ echo "${var:=XX}"     # Parameter is unset -> word is assigned to XX
XX
$ echo "$var"
XX
$ var=""                # Parameter is null -> word is assigned to XX
$ echo "${var:=XX}"
XX
$ echo "$var"
XX
$ var=23                # Parameter is not null -> no assignment occurs
$ echo "${var:=XX}"
23
$ echo "$var"
23
```

Erreur si la variable est vide ou non définie

La sémantique de cette méthode est similaire à celle de la substitution de valeur par défaut, mais au lieu de remplacer une valeur par défaut, elle génère une erreur avec le message d'erreur fourni. Les formulaires sont `${VARNAME?ERRMSG}` et `${VARNAME:?ERRMSG}` . La forme avec `:` va erreur notre si la variable est **non définie** *ou* **vide** , alors que la forme sans sera seulement une erreur si la variable n'est pas *définie* . Si une erreur est `ERRMSG` , le `ERRMSG` est `ERRMSG` et le code de sortie est défini sur `1` .

```
#!/bin/bash
FOO=
# ./script.sh: line 4: FOO: EMPTY
echo "FOO is ${FOO:?EMPTY}"
# FOO is
echo "FOO is ${FOO?UNSET}"
# ./script.sh: line 8: BAR: EMPTY
echo "BAR is ${BAR:?EMPTY}"
# ./script.sh: line 10: BAR: UNSET
echo "BAR is ${BAR?UNSET}"
```

L'exécution de l'exemple complet au-dessus de chacune des instructions d'écho erronées doit être commentée pour continuer.

Supprimer un motif du début d'une chaîne

Match le plus court:

```
$ a='I am a string'
$ echo "${a#*a}"
m a string
```

Match le plus long:

```
$ echo "${a##*a}"
 string
```

Supprimer un motif à la fin d'une chaîne

Match le plus court:

```
$ a='I am a string'
$ echo "${a%a*}"
I am
```

Match le plus long:

```
$ echo "${a%%a*}"
I
```

Remplacer le motif dans la chaîne

Premier match:

```
$ a='I am a string'
$ echo "${a/a/A}"
I Am a string
```

Tous les matches:

```
$ echo "${a//a/A}"
I Am A string
```

Match au début:

```
$ echo "${a/#I/y}"
y am a string
```

Match à la fin:

```
$ echo "${a/%g/N}"
I am a strinN
```

Remplacez un motif par rien:

```
$ echo "${a/g/}"
I am a strin
```

Ajouter un préfixe aux éléments du tableau:

```
$ A=(hello world)
$ echo "${A[@]/#/R}"
Rhello Rworld
```

Munging pendant l'expansion

Les variables ne doivent pas nécessairement être étendues à leurs valeurs - les sous-chaînes peuvent être extraites lors de l'extension, ce qui peut être utile pour extraire des extensions de fichiers ou des parties de chemins. Les personnages qui retentissent gardent leurs significations habituelles, donc `.*` Se réfère à un point littéral, suivi de toute séquence de caractères; ce n'est pas une expression régulière.

```
$ v=foo-bar-baz
$ echo ${v%%-*}
foo
$ echo ${v%-*}
foo-bar
$ echo ${v##*-}
baz
$ echo ${v#*-}
bar-baz
```

Il est également possible de développer une variable en utilisant une valeur par défaut - disons que je veux appeler l'éditeur de l'utilisateur, mais si je n'en ai pas défini un, je voudrais lui donner `vim` .

```
$ EDITOR=nano
$ ${EDITOR:-vim} /tmp/some_file
# opens nano
$ unset EDITOR
$ $ ${EDITOR:-vim} /tmp/some_file
# opens vim
```

Il existe deux manières différentes d'effectuer cette expansion, qui diffèrent selon que la variable concernée est vide ou non définie. Utiliser `:-` utilisera la valeur par défaut si la variable est non définie ou vide, alors que `-` utilise uniquement la valeur par défaut si la variable est désactivée, mais utilisera la variable si elle est définie sur la chaîne vide:

```
$ a="set"
$ b=""
$ unset c
$ echo ${a:-default_a} ${b:-default_b} ${c:-default_c}
set default_b default_c
$ echo ${a-default_a} ${b-default_b} ${c-default_c}
set default_c
```

Semblable aux défauts, des alternatives peuvent être données; Lorsqu'une valeur par défaut est utilisée si une variable particulière n'est pas disponible, une alternative est utilisée si la variable est disponible.

```
$ a="set"
$ b=""
$ echo ${a:+alternative_a} ${b:+alternative_b}
alternative_a
```

Notant que ces extensions peuvent être imbriquées, l'utilisation d'alternatives devient particulièrement utile lorsque vous fournissez des arguments aux indicateurs de ligne de commande;

```
$ output_file=/tmp/foo
$ wget ${output_file:+"-o ${output_file}"} www.stackexchange.com
# expands to wget -o /tmp/foo www.stackexchange.com
$ unset output_file
$ wget ${output_file:+"-o ${output_file}"} www.stackexchange.com
# expands to wget  www.stackexchange.com
```

Extension de paramètres et noms de fichiers

Vous pouvez utiliser Bash Parameter Expansion pour émuler des opérations courantes de traitement de noms de fichiers telles que `basename` et `dirname` .

Nous utiliserons ceci comme notre exemple de chemin:

```
FILENAME="/tmp/example/myfile.txt"
```

Pour émuler `dirname` et renvoyer le nom de répertoire d'un chemin de fichier:

```
echo "${FILENAME%/*}"
#Out: /tmp/example
```

Pour émuler le `basename $FILENAME` et renvoyer le nom de fichier d'un chemin de fichier:

```
echo "${FILENAME##*/}"
#Out: myfile.txt
```

Pour émuler le `basename $FILENAME .txt` et renvoyer le nom de fichier sans le `.txt.` extension:

```
BASENAME="${FILENAME##*/}"
echo "${BASENAME%%.txt}"
#Out: myfile
```

Lire Extension des paramètres Bash en ligne: https://riptutorial.com/fr/bash/topic/502/extension-des-parametres-bash

Chapitre 25: Fractionnement de fichiers

Introduction

Parfois, il est utile de diviser un fichier en plusieurs fichiers distincts. Si vous avez des fichiers volumineux, il peut être judicieux de les diviser en morceaux plus petits.

Examples

Diviser un fichier

L'exécution de la commande split sans aucune option divisera un fichier en 1 ou plusieurs fichiers séparés contenant chacun jusqu'à 1000 lignes.

```
split file
```

Cela créera des fichiers nommés `xaa` , `xab` , `xac` , etc., contenant chacun jusqu'à 1000 lignes. Comme vous pouvez le voir, tous sont préfixés par la lettre `x` par défaut. Si le fichier initial était inférieur à 1000 lignes, un seul de ces fichiers serait créé.

Pour modifier le préfixe, ajoutez le préfixe souhaité à la fin de la ligne de commande

```
split file customprefix
```

Maintenant, les fichiers nommés `customprefixaa` , `customprefixab` , `customprefixac` etc. seront créés

Pour spécifier le nombre de lignes à générer par fichier, utilisez l'option `-l` . Ce qui suit divisera un fichier en un maximum de 5000 lignes

```
split -l5000 file
```

OU

```
split --lines=5000 file
```

Vous pouvez également spécifier un nombre maximal d'octets au lieu de lignes. Cela se fait en utilisant les options `-b` ou `--bytes` . Par exemple, pour autoriser un maximum de 1 Mo

```
split --bytes=1MB file
```

Nous pouvons utiliser sed avec l'option w pour diviser un fichier en plusieurs fichiers. Les fichiers peuvent être divisés en spécifiant une adresse de ligne ou un motif.

Supposons que nous ayons ce fichier source que nous voudrions séparer:

```
cat –n sourcefile
```

1 sur le Ning Nang Nong
2 Où les vaches vont Bong!
3 et les singes disent tous BOO!
4 Il y a un Nong Nang Ning
5 Où vont les arbres Ping!
6 Et les théières jibber jabber joo.
7 sur le Nong Ning Nang

Commande pour diviser le fichier par numéro de ligne:

```
sed '1,3w f1
> 4,7w f2' sourcefile
```

Cela écrit line1 à line3 dans le fichier f1 et line4 à line7 dans le fichier f2, à partir du fichier source.

```
cat –n f1
```

1 sur le Ning Nang Nong
2 Où les vaches vont Bong!
3 et les singes disent tous BOO!

```
cat –n f2
```

1 Il y a un Nong Nang Ning
2 Où vont les arbres Ping!
3 Et les théières jibber jabber joo.
4 sur le Nong Ning Nang

Commande pour diviser le fichier par contexte / modèle:

```
sed '/Ning/w file1
> /Ping/w file2' sourcefile
```

Cela divise le fichier source en fichier1 et fichier2. file1 contient toutes les lignes correspondant à Ning, file2 contient les lignes correspondant à Ping.

```
cat file1
```

Sur le Ning Nang Nong
Il y a un Nong Nang Ning
Sur le Nong Ning Nang

```
cat file2
```

Où vont les arbres Ping!

Lire Fractionnement de fichiers en ligne: https://riptutorial.com/fr/bash/topic/9151/fractionnement-de-fichiers

Chapitre 26: Fractionnement de mots

Syntaxe

- Définissez IFS sur newline: IFS = $ '\ n'
- Définissez IFS sur nullstring: IFS =
- Définissez IFS sur / character: IFS = /

Paramètres

Paramètre	Détails
IFS	Séparateur de champ interne
-X	Les commandes d'impression et leurs arguments au fur et à mesure de leur exécution (option Shell)

Remarques

- Le fractionnement de mots n'est pas effectué pendant les affectations, par exemple `newvar=$var`
- Le fractionnement de mots n'est pas effectué dans la construction `[[ ... ]]`
- Utilisez des guillemets doubles sur les variables pour empêcher le fractionnement des mots

Examples

Fractionnement avec IFS

Pour être plus clair, créons un script nommé `showarg` :

```
#!/usr/bin/env bash
printf "%d args:" $#
printf " <%s>" "$@"
echo
```

Voyons maintenant les différences:

```
$ var="This is an example"
$ showarg $var
4 args: <This> <is> <an> <example>
```

> `$var` est divisé en 4 arguments. `IFS` est un espace blanc et le fractionnement des mots se produit donc dans les espaces

```
$ var="This/is/an/example"
$ showarg $var
1 args: <This/is/an/example>
```

Dans ce qui précède, le fractionnement des mots n'a pas eu lieu car les caractères `IFS` n'ont pas été trouvés.

Maintenant, définissons `IFS=/`

```
$ IFS=/
$ var="This/is/an/example"
$ showarg $var
4 args: <This> <is> <an> <example>
```

Le `$var` divise en 4 arguments pas un seul argument.

Quoi, quand et pourquoi

Lorsque le shell effectue une *expansion des paramètres* , une *substitution de commande* , *une expansion variable ou arithmétique* , il recherche les limites de mots dans le résultat. Si une limite de mot est trouvée, le résultat est divisé en plusieurs mots à cette position. La limite du mot est définie par une variable shell `IFS` (Internal Field Separator). La valeur par défaut pour IFS est l'espace, la tabulation et la nouvelle ligne, c'est-à-dire que le fractionnement des mots se produira sur ces trois caractères d'espace blanc si cela n'est pas empêché explicitement.

```
set -x
var='I am
a
multiline string'
fun() {
    echo "-$1-"
    echo "*$2*"
    echo ".$3."
}
fun $var
```

Dans l'exemple ci-dessus, c'est ainsi que la fonction `fun` est exécutée:

```
fun I am a multiline string
```

`$var` est divisé en 5 arguments, seuls `I` , `am` et `a` seront imprimés.

IFS et fractionnement de mots

Voir quoi, quand et pourquoi si vous ne connaissez pas l'affiliation d'IFS au fractionnement de mots

définissons l'IFS sur un caractère d'espace uniquement:

```
set -x
var='I am
```

```
a
multiline string'
IFS=' '
fun() {
    echo "-$1-"
    echo "*$2*"
    echo ".$3."
}
fun $var
```

Ce fractionnement ne fonctionnera que sur les espaces. La fonction `fun` sera exécutée comme ceci:

```
fun I 'am
a
multiline' string
```

`$var` est divisé en 3 arguments. `I am\na\nmultiline` et la `string` sera imprimée

Définissons l'IFS sur newline uniquement:

```
IFS=$'\n'
...
```

Maintenant, le `fun` sera exécuté comme:

```
fun 'I am' a 'multiline string'
```

`$var` est divisé en 3 arguments. `I am` , `a` `multiline string` sera imprimée

Voyons ce qui se passe si nous définissons IFS sur nullstring:

```
IFS=
...
```

Cette fois, le `fun` sera exécuté comme ceci:

```
fun 'I am
a
multiline string'
```

`$var` n'est pas divisé, c'est-à-dire qu'il reste un seul argument.

Vous pouvez empêcher le fractionnement de mots en définissant l'IFS sur nullstring

Une manière générale d'empêcher le fractionnement des mots est d'utiliser un double guillemet:

```
fun "$var"
```

empêchera le fractionnement des mots dans tous les cas décrits ci-dessus, c'est-à-dire que la

fonction `fun` sera exécutée avec un seul argument.

Mauvais effets du fractionnement de mots

```
$ a='I am a string with spaces'
$ [ $a = $a ] || echo "didn't match"
bash: [: too many arguments
didn't match
```

> `[ $a = $a ]` été interprété comme `[ I am a string with spaces = I am a string with spaces ]` . `[` est la commande de `test` pour laquelle `I am a string with spaces` n'est pas un seul argument, c'est plutôt **6** arguments !!

```
$ [ $a = something ] || echo "didn't match"
bash: [: too many arguments
didn't match
```

> `[ $a = something ]` été interprété comme `[ I am a string with spaces = something ]`

```
$ [ $(grep . file) = 'something' ]
bash: [: too many arguments
```

> La commande `grep` renvoie une chaîne multiligne avec des espaces, vous pouvez donc imaginer combien d'arguments sont disponibles: D

Voir quoi, quand et pourquoi pour l'essentiel.

Utilité du fractionnement de mots

Dans certains cas, le fractionnement des mots peut être utile:

Remplissage du tableau:

```
arr=($(grep -o '[0-9]\+' file))
```

> Cela remplira `arr` avec toutes les valeurs numériques trouvées dans le *fichier*

En boucle à travers des mots séparés par des espaces:

```
words='foo bar baz'
for w in $words;do
    echo "W: $w"
done
```

Sortie:

```
W: foo
W: bar
W: baz
```

Paramètres séparés par des espaces qui ne contiennent pas d'espaces blancs:

```
packs='apache2 php php-mbstring php-mysql'
sudo apt-get install $packs
```

ou

```
packs='
apache2
php
php-mbstring
php-mysql
'
sudo apt-get install $packs
```

> Cela va installer les paquets. Si vous doublez les `$packs` cela générera une erreur.
>
> Unquoetd `$packs` envoie tous les noms de paquets séparés par des espaces comme arguments à `apt-get` , en les citant, il enverra la chaîne `$packs` en un seul argument, puis `apt-get` tentera d'installer un paquet nommé `apache2 php php-mbstring php-mysql` (pour le premier) qui n'existe évidemment pas

Voir quoi, quand et pourquoi pour l'essentiel.

Fractionnement par séparateur

Nous pouvons simplement effectuer un remplacement simple des séparateurs de l'espace vers la nouvelle ligne, comme dans l'exemple suivant.

```
echo $sentence | tr " " "\n"
```

Il divisera la valeur de la `sentence` variable et la montrera ligne par ligne respectivement.

Lire Fractionnement de mots en ligne: https://riptutorial.com/fr/bash/topic/5472/fractionnement-de-mots

Chapitre 27: Gestion de l'invite du système

Syntaxe

- export PS1 = "quelque chose" # s'affiche lorsque bash attend une commande à saisir
- export PS2 = "anotherthing" # dsplayed quand l'instruction s'étend à plus de lignes
- export PS3 = "Invite de questions pour une instruction select" # Invite rarement à utiliser select. Commencez par définir PS3 selon vos besoins, puis **sélectionnez** Appel. Voir **aide sélectionner**
- export PS4 = "surtout utile pour le débogage; numéro de ligne, etc." # utilisé pour le débogage des scripts bash.

Paramètres

Échapper	Détails
\une	Un personnage de cloche.
\ré	La date, en format "Jour Mois Mois" (par exemple, "Mardi 26 Mai").
\ D {FORMAT}	Le FORMAT est passé à `strftime '(3) et le résultat est inséré dans la chaîne d'invite. un FORMAT vide produit une représentation temporelle spécifique à l'environnement local. Les accolades sont obligatoires
\ e	Un caractère d'évasion. \ 033 fonctionne bien sûr aussi.
\ h	Le nom d'hôte, jusqu'au premier `. ' (c.-à-d. pas de partie de domaine)
\ H	Le nom d'hôte éventuellement avec une partie de domaine
\ j	Le nombre d'emplois actuellement gérés par le shell.
\ l	Le nom de base du nom du terminal du shell.
\ n	Une nouvelle ligne.
\ r	Un retour de chariot.
\ s	Le nom du shell, le nom de base de `$ 0 '(la partie qui suit la barre oblique finale).
\ t	L'heure, au format 24 heures HH: MM: SS.
\ T	L'heure, au format HH: MM: SS sur 12 heures.
@	L'heure, au format 12 heures / heure.

Échapper	Détails
\UNE	L'heure, au format HH: MM 24 heures.
\ u	Le nom d'utilisateur de l'utilisateur actuel.
\ v	La version de Bash (par exemple, 2.00)
\ V	La sortie de Bash, version + patchlevel (par exemple, 2.00.0)
\ w	Le répertoire de travail actuel, avec $ HOME abrégé avec un tilde (utilise la variable $ PROMPT_DIRTRIM).
\ W	Le nom de base de $ PWD, avec $ HOME abrégé avec un tilde.
!	Le numéro d'historique de cette commande.
#	Le numéro de commande de cette commande.
$	Si l'ID effectif est 0, # sinon **$** .
\ NNN	Le caractère dont le code ASCII est la valeur octale NNN.
\	Une barre oblique inverse.
\ [	Commencez une séquence de caractères non imprimables. Cela pourrait être utilisé pour incorporer une séquence de contrôle de terminal dans l'invite.
\]	Terminez une séquence de caractères non imprimables.

Examples

Utilisation de la variable d'environnement PROMPT_COMMAND

Lorsque la dernière commande d'une instance de bash interactive est terminée, la variable PS1 évaluée est displayes. Avant d'afficher PS1, bash vérifie si PROMPT_COMMAND est défini. Cette valeur de cette variable doit être un programme ou un script appelable. Si cette variable est définie, ce programme / script est appelé AVANT que l'invite PS1 ne soit affichée.

```
# just a stupid function, we will use to demonstrate
# we check the date if Hour is 12 and Minute is lower than 59
lunchbreak(){
   if (( $(date +%H) == 12 && $(date +%M) < 59 )); then
      # and print colored \033[ starts the escape sequence
      # 5; is blinking attribute
      # 2; means bold
      # 31 says red
      printf "\033[5;1;31mmind the lunch break\033[0m\n";
   else
      printf "\033[33mstill working...\033[0m\n";
   fi;
```

```
}

# activating it
export PROMPT_COMMAND=lunchbreak
```

Utiliser PS2

PS2 est affiché lorsqu'une commande s'étend sur plusieurs lignes et que bash attend plus de frappes. Il s'affiche également lorsqu'une commande composée comme **lorsque ... do..done** et identique est entrée.

```
export PS2="would you please complete this command?\n"
# now enter a command extending to at least two lines to see PS2
```

Utiliser PS3

Lorsque l'instruction select est exécutée, elle affiche les éléments donnés précédés d'un numéro, puis affiche l'invite PS3:

```
export PS3="  To choose your language type the preceding number : "
select lang in EN CA FR DE; do
   # check input here until valid.
   break
done
```

Utiliser PS4

PS4 est affiché lorsque bash est en mode débogage.

```
#!/usr/bin/env bash

# switch on debugging
set -x

# define a stupid_func
stupid_func(){
   echo I am line 1 of stupid_func
   echo I am line 2 of stupid_func
}

# setting the PS4 "DEBUG" prompt
export PS4='\nDEBUG level:$SHLVL subshell-level: $BASH_SUBSHELL \nsource-file:${BASH_SOURCE}
line#:${LINENO} function:${FUNCNAME[0]:+${FUNCNAME[0]}(): }\nstatement: '

# a normal statement
echo something

# function call
stupid_func

# a pipeline of commands running in a subshell
( ls -l | grep 'x' )
```

Utiliser PS1

PS1 est l'invite système normale indiquant que bash attend les commandes en cours de saisie. Il comprend certaines séquences d'échappement et peut exécuter des fonctions ou des programmes. Comme bash doit positionner le curseur après l'invite d'affichage, il doit savoir comment calculer la longueur effective de la chaîne d'invite. Pour indiquer des séquences de caractères non imprimables dans la variable PS1, des accolades sont utilisées: **\ [** *une séquence de caractères non imprimable* **\]** . Tout ce qui est dit est vrai pour tous les PS PS.

(Le caret noir indique le curseur)

```
#everything not being an escape sequence will be literally printed
export PS1="literal sequence "  # Prompt is now:
literal sequence █

# \u == user \h == host \w == actual working directory
# mind the single quotes avoiding interpretation by shell
export PS1='\u@\h:\w > ' # \u == user, \h == host, \w actual working dir
looser@host:/some/path > █

# executing some commands within PS1
# following line will set foreground color to red, if user==root,
# else it resets attributes to default
# $( (($EUID == 0)) &&  tput setaf 1)
# later we do reset attributes to default with
# $(  tput sgr0 )
# assuming being root:
PS1="\[$( (($EUID == 0)) &&  tput setaf 1 \]\u\[$(tput sgr0)\]@\w:\w \$ "
looser@host:/some/path > █  # if not root else <red>root<default>@host....
```

Lire Gestion de l'invite du système en ligne: https://riptutorial.com/fr/bash/topic/7541/gestion-de-l-invite-du-systeme

Chapitre 28: Gestion de la variable d'environnement PATH

Syntaxe

- Ajouter un chemin: PATH = $ PATH: / new / path
- Ajouter un chemin: PATH = / new / path: $ PATH

Paramètres

Paramètre	Détails
CHEMIN	Variable d'environnement de chemin

Remarques

Fichier de configuration Bash:

Ce fichier provient d'un nouveau shell Bash interactif.

Dans les systèmes GNU / Linux, c'est généralement le fichier ~ / .bashrc; dans Mac, c'est ~ / .bash_profile ou ~ / .profile

Exportation:

La variable PATH doit être exportée une fois (c'est fait par défaut). Une fois exporté, il restera exporté et les modifications apportées seront immédiatement appliquées.

Appliquer les modifications:

Pour appliquer les modifications à un fichier de configuration Bash, vous devez recharger ce fichier dans un terminal (`source /path/to/bash_config_file`)

Examples

Ajouter un chemin à la variable d'environnement PATH

La variable d'environnement PATH est généralement définie dans ~ / .bashrc ou ~ / .bash_profile ou / etc / profile ou ~ / .profile ou /etc/bash.bashrc (fichier de configuration Bash spécifique à la distribution)

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

Maintenant, si nous voulons ajouter un chemin (par exemple `~/bin`) à la variable PATH:

```
PATH=~/bin:$PATH
# or
PATH=$PATH:~/bin
```

Mais cela ne modifiera le PATH que dans le shell actuel (et son sous-shell). Une fois que vous quittez le shell, cette modification aura disparu.

Pour le rendre permanent, nous devons ajouter ce bit de code au fichier ~ / .bashrc (ou autre) et recharger le fichier.

Si vous exécutez le code suivant (dans le terminal), il ajoutera définitivement `~/bin` au PATH:

```
echo 'PATH=~/bin:$PATH' >> ~/.bashrc && source ~/.bashrc
```

Explication:

- `echo 'PATH=~/bin:$PATH' >> ~/.bashrc` ajoute la ligne `PATH=~/bin:$PATH` à la fin du fichier ~ / .bashrc (vous pouvez le faire avec un éditeur de texte)
- `source ~/.bashrc` recharge le fichier ~ / .bashrc

Ceci est un bit de code (exécuté dans le terminal) qui vérifiera si un chemin est déjà inclus et ajoute le chemin seulement s'il n'est pas:

```
path=~/bin            # path to be included
bashrc=~/.bashrc      # bash file to be written and reloaded
# run the following code unmodified
echo $PATH | grep -q "\(^\|:\)$path\(:\|/\{0,1\}$\)" || echo "PATH=\$PATH:$path" >> "$bashrc";
source "$bashrc"
```

Supprimer un chemin de la variable d'environnement PATH

Pour supprimer un PATH d'une variable d'environnement PATH, vous devez éditer le fichier ~ / .bashrc ou ~ / .bash_profile ou / etc / profile ou ~ / .profile ou /etc/bash.bashrc (spécifique à la distribution) et supprimer l'affectation pour ce chemin particulier.

Au lieu de trouver l'assignation exacte, vous pourriez simplement faire un remplacement dans `$PATH` dans sa phase finale.

Ce qui suit va supprimer en toute sécurité `$path` de `$PATH` :

```
path=~/bin
PATH="$(echo "$PATH" |sed -e "s#\(^\|:\)$(echo "$path" |sed -e 's/[^^]/[&]/g' -e
's/\^/\\^/g')\(:\|/\{0,1\}$\)#\1\2#" -e 's#:\+#:#g' -e 's#^:\|:$##g')"
```

Pour le rendre permanent, vous devrez l'ajouter à la fin de votre fichier de configuration bash.

Vous pouvez le faire de manière fonctionnelle:

```
rpath(){
    for path in "$@";do
        PATH="$(echo "$PATH" |sed -e "s#\(^\|:\)$(echo "$path" |sed -e 's/[^^]/[&]/g' -e
's/\^/\\^/g')\(:\|/\{0,1\}$\)#\1\2#" -e 's#:\+#:#g' -e 's#^:\|:$##g')"
    done
    echo "$PATH"
}

PATH="$(rpath ~/bin /usr/local/sbin /usr/local/bin)"
PATH="$(rpath /usr/games)"
# etc ...
```

Cela facilitera le traitement de plusieurs chemins.

Remarques:

- Vous devrez ajouter ces codes dans le fichier de configuration Bash (~ / .bashrc ou autre).
- Exécutez le `source ~/.bashrc` pour recharger le fichier de configuration Bash (~ / .bashrc).

Lire Gestion de la variable d'environnement PATH en ligne:
https://riptutorial.com/fr/bash/topic/5515/gestion-de-la-variable-d-environnement-path

Chapitre 29: getopts: analyse intelligente des paramètres positionnels

Syntaxe

- getopts optstring name [args]

Paramètres

Paramètre	Détail
optstring	Les caractères d'option à reconnaître
prénom	Puis le nom où l'option analysée est stockée

Remarques

Les options

`optstring` : les caractères d'option à reconnaître

> Si un caractère est suivi de deux points, l'option doit avoir un argument, qui doit être séparé par un espace blanc. Les deux points (`:`) (point d'interrogation `?`) Ne peuvent pas être utilisés comme caractères d'option.

À chaque appel, `getopts` place l'option suivante dans le nom de la variable shell, en initialisant le nom s'il n'existe pas, et l'index du prochain argument à traiter dans la variable `OPTIND` . `OPTIND` est initialisé à `1` chaque fois que le shell ou un script shell est appelé.

Lorsqu'une option nécessite un argument, `getopts` place cet argument dans la variable `OPTARG` . Le shell ne réinitialise pas automatiquement `OPTIND` ; il doit être réinitialisé manuellement entre plusieurs appels à `getopts` dans la même invocation de shell si un nouvel ensemble de paramètres doit être utilisé.

Lorsque la fin des options est rencontrée, `getopts` avec une valeur de retour supérieure à zéro.

`OPTIND` est défini sur l'index du premier argument non-option et name est défini sur `?` . `getopts` analyse normalement les paramètres de position, mais si plus d'arguments sont donnés dans `args` , `getopts` analyse à la place.

`getopts` peuvent signaler des erreurs de deux manières. Si le premier caractère de `optstring` est deux points (`:`), les rapports d'erreur silencieuse est utilisée. En fonctionnement normal, les messages de diagnostic sont imprimés lorsque des options non valides ou des arguments d'option

manquants sont rencontrés.

Si la variable `OPTERR` est définie sur `0` , aucun message d'erreur ne sera affiché, même si le premier caractère de `optstring` n'est pas un deux-points.

Si une option non valide est vu, `getopts` places `?` dans le `name` et, si non silencieux, imprime un message d'erreur et désélectionne `OPTARG` . Si `getopts` est silencieux, le caractère d'option trouvé est placé dans `OPTARG` et aucun message de diagnostic n'est imprimé.

Si un argument requis n'est pas trouvé et que `getopts` n'est pas silencieux, un point d'interrogation (`?`) Est placé dans le `name` , `OPTARG` n'est pas `OPTARG` et un message de diagnostic est imprimé. Si `getopts` est silencieux, deux points (`:`) est placé dans le nom et `OPTARG` est réglé sur le caractère d'option.

Examples

pingnmap

```
#!/bin/bash
# Script name : pingnmap
# Scenario : The systems admin in company X is tired of the monotonous job
# of pinging and nmapping, so he decided to simplify the job using a script.
# The tasks he wish to achieve is
# 1. Ping - with a max count of 5 -the given IP address/domain. AND/OR
# 2. Check if a particular port is open with a given IP address/domain.
# And getopts is for her rescue.
# A brief overview of the options
# n : meant for nmap
# t : meant for ping
# i : The option to enter the IP address
# p : The option to enter the port
# v : The option to get the script version


while getopts ':nti:p:v' opt
#putting : in the beginnnig suppresses the errors for invalid options
do
case "$opt" in
   'i')ip="${OPTARG}"
       ;;
   'p')port="${OPTARG}"
       ;;
   'n')nmap_yes=1;
       ;;
   't')ping_yes=1;
       ;;
   'v')echo "pingnmap version 1.0.0"
       ;;
    *) echo "Invalid option $opt"
       echo "Usage : "
       echo "pingmap -[n|t[i|p]|v]"
       ;;
esac
done
if  [ ! -z "$nmap_yes" ] && [ "$nmap_yes" -eq "1" ]
```

```
then
   if [ ! -z "$ip" ] && [ ! -z "$port" ]
   then
     nmap -p "$port" "$ip"
   fi
fi

if  [ ! -z "$ping_yes" ] && [ "$ping_yes" -eq "1" ]
then
   if [ ! -z "$ip" ]
   then
     ping -c 5 "$ip"
   fi
fi
shift $(( OPTIND - 1 )) # Processing additional arguments
if [ ! -z "$@" ]
then
  echo "Bogus arguments at the end : $@"
fi
```

Sortie

```
$ ./pingnmap -nt -i google.com -p 80

Starting Nmap 6.40 ( http://nmap.org ) at 2016-07-23 14:31 IST
Nmap scan report for google.com (216.58.197.78)
Host is up (0.034s latency).
rDNS record for 216.58.197.78: maa03s21-in-f14.1e100.net
PORT   STATE SERVICE
80/tcp open  http

Nmap done: 1 IP address (1 host up) scanned in 0.22 seconds
PING google.com (216.58.197.78) 56(84) bytes of data.
64 bytes from maa03s21-in-f14.1e100.net (216.58.197.78): icmp_seq=1 ttl=57 time=29.3 ms
64 bytes from maa03s21-in-f14.1e100.net (216.58.197.78): icmp_seq=2 ttl=57 time=30.9 ms
64 bytes from maa03s21-in-f14.1e100.net (216.58.197.78): icmp_seq=3 ttl=57 time=34.7 ms
64 bytes from maa03s21-in-f14.1e100.net (216.58.197.78): icmp_seq=4 ttl=57 time=39.6 ms
64 bytes from maa03s21-in-f14.1e100.net (216.58.197.78): icmp_seq=5 ttl=57 time=32.7 ms

--- google.com ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4007ms
rtt min/avg/max/mdev = 29.342/33.481/39.631/3.576 ms
$ ./pingnmap -v
pingnmap version 1.0.0
$ ./pingnmap -h
Invalid option ?
Usage :
pingmap -[n|t[i|p]|v]
$ ./pingnmap -v
pingnmap version 1.0.0
$ ./pingnmap -h
Invalid option ?
Usage :
pingmap -[n|t[i|p]|v]
```

Lire getopts: analyse intelligente des paramètres positionnels en ligne: https://riptutorial.com/fr/bash/topic/3654/getopts--analyse-intelligente-des-parametres-positionnels

Chapitre 30: Grep

Syntaxe

- grep [OPTIONS] PATTERN [FILE ...]

Examples

Comment rechercher un motif dans un fichier

Pour trouver le mot **foo** dans la *barre de* fichiers:

```
grep foo ~/Desktop/bar
```

Pour rechercher toutes les lignes **ne** contenant **pas** foo dans la *barre de* fichiers:

```
grep -v foo ~/Desktop/bar
```

Pour trouver tous les mots contenant foo (WIldcard Expansion):

```
grep "*foo" ~/Desktop/bar
```

Lire Grep en ligne: https://riptutorial.com/fr/bash/topic/10852/grep

Chapitre 31: Ici des documents et ici des cordes

Examples

En retrait ici des documents

Vous pouvez indenter le texte à l'intérieur des documents avec des tabulations, vous devez utiliser l'opérateur `<<-` redirection au lieu de `<<` :

```
$ cat <<- EOF
    This is some content indented with tabs `\t`.
    You cannot indent with spaces you __have__ to use tabs.
    Bash will remove empty space before these lines.
    __Note__: Be sure to replace spaces with tabs when copying this example.
EOF

This is some content indented with tabs _\t_.
You cannot indent with spaces you __have__ to use tabs.
Bash will remove empty space before these lines.
__Note__: Be sure to replace spaces with tabs when copying this example.
```

Un cas d'utilisation pratique (comme mentionné dans `man bash`) est dans les scripts shell, par exemple:

```
if cond; then
    cat <<- EOF
    hello
    there
    EOF
fi
```

Il est habituel d'indenter les lignes dans les blocs de code, comme dans cette instruction `if` , pour une meilleure lisibilité. Sans la syntaxe de l'opérateur `<<-` , nous serions obligés d'écrire le code ci-dessus comme ceci:

```
if cond; then
    cat << EOF
hello
there
EOF
fi
```

C'est très désagréable à lire, et ça empire dans un scénario réaliste plus complexe.

Ici des cordes

2.05b

Vous pouvez alimenter une commande en utilisant les chaînes comme ceci:

```
$ awk '{print $2}' <<< "hello world - how are you?"
world

$ awk '{print $1}' <<< "hello how are you
> she is fine"
hello
she
```

Vous pouvez également alimenter une `while` boucle avec une chaîne ici:

```
$ while IFS=" " read -r word1 word2 rest
> do
> echo "$word1"
> done <<< "hello how are you - i am fine"
hello
```

Limite des cordes

Un heredoc utilise la *chaîne* de *limitation* pour déterminer quand arrêter de consommer les entrées. La terminaison limite **doit**

- Être au début d'une ligne.
- Soyez le seul texte sur la ligne **Note:** Si vous utilisez `<<-` la chaîne de limitation peut être préfixée avec des tabulations `\t`

Correct:

```
cat <<limitstring
line 1
line 2
limitstring
```

Cela va sortir:

```
line 1
line 2
```

Utilisation incorrecte:

```
cat <<limitstring
line 1
line 2
 limitstring
```

Comme la `limitstring` sur la dernière ligne n'est pas exactement au début de la ligne, le shell continuera d'attendre d'autres entrées, jusqu'à ce qu'il voie une ligne qui commence par `limitstring` et ne contienne rien d'autre. Ce n'est qu'à ce moment-là qu'il arrêtera d'attendre la saisie, et passera ensuite le document ici à la commande `cat` .

Notez que lorsque vous préfixez la chaîne de limitation initiale avec un trait d'union, tous les onglets au début de la ligne sont supprimés avant l'analyse, ainsi les données et la chaîne de limite peuvent être indentées avec des tabulations

```
cat <<-limitstring
        line 1    has a tab each before the words line and has
            line 2 has two leading tabs
        limitstring
```

produira

```
line 1    has a tab each before the words line and has
line 2 has two leading tabs
```

avec les onglets principaux (mais pas les onglets internes) supprimés.

Créer un fichier

Une utilisation classique des documents ici est de créer un fichier en tapant son contenu:

```
cat > fruits.txt << EOF
apple
orange
lemon
EOF
```

Le document ici est les lignes entre les `<< EOF` et `EOF` .

Ce document ici devient l'entrée de la commande `cat` . La commande `cat` sort simplement son entrée, et en utilisant l'opérateur de redirection de sortie `>` nous redirigeons vers un fichier `fruits.txt` .

Par conséquent, le fichier `fruits.txt` contiendra les lignes:

```
apple
orange
lemon
```

Les règles habituelles de redirection de sortie s'appliquent: si le `fruits.txt` n'existait pas auparavant, il sera créé. S'il existait auparavant, il sera tronqué.

Exécuter la commande avec ici le document

```
ssh -p 21 example@example.com <<EOF
  echo 'printing pwd'
  echo "\$(pwd)"
  ls -a
  find '*.txt'
EOF
```

```
$
```

est échappé car nous ne voulons pas qu'il soit développé par le shell courant, c'est- `$(pwd)` dire que `$(pwd)` doit être exécuté sur le shell distant.

Autrement:

```
ssh -p 21 example@example.com <<'EOF'
  echo 'printing pwd'
  echo "$(pwd)"
  ls -a
  find '*.txt'
EOF
```

Note : La fermeture EOF **devrait** être au début de la ligne (Pas d'espaces blancs auparavant). Si l'indentation est requise, des onglets peuvent être utilisés si vous démarrez votre heredoc avec `<<-` . Reportez-vous aux exemples Indenting here documents et Limit Strings pour plus d'informations.

Exécuter plusieurs commandes avec sudo

```
sudo -s <<EOF
  a='var'
  echo 'Running serveral commands with sudo'
  mktemp -d
  echo "\$a"
EOF
```

- `$a` doit être échappé pour empêcher son extension par le shell actuel

Ou

```
sudo -s <<'EOF'
  a='var'
  echo 'Running serveral commands with sudo'
  mktemp -d
  echo "$a"
EOF
```

Lire Ici des documents et ici des cordes en ligne: https://riptutorial.com/fr/bash/topic/655/ici-des-documents-et-ici-des-cordes

Chapitre 32: La commande de coupe

Introduction

La commande `cut` est un moyen rapide d'extraire des parties de lignes de fichiers texte. Il appartient aux plus anciennes commandes Unix. Ses implémentations les plus populaires sont la version GNU trouvée sur Linux et la version FreeBSD disponible sur MacOS, mais chaque version d'Unix a la sienne. Voir ci-dessous pour les différences. Les lignes d'entrée sont lues à partir de `stdin` ou de fichiers répertoriés comme arguments sur la ligne de commande.

Syntaxe

- cut -f1,3 # extrait le premier *et le* troisième **champ** *délimité par des tabulations* (à partir de stdin)
- cut -f1-3 # extrait *du* premier *au* troisième champ (extrémités incluses)
- cut -f-3 # -3 est interprété comme 1-3
- cut -f2- # 2 est interprété *de la seconde au dernier*
- cut -c1-5,10 # extrait de stdin les **personnages** dans les positions 1,2,3,4,5,10
- cut -s -f1 # supprime les lignes ne contenant pas de délimiteurs
- cut --complement -f3 # (coupe GNU uniquement) extrait *tous les* champs *sauf* le troisième

Paramètres

Paramètre	Détails
-f, --fields	Sélection par champs
-d, --delimiter	Délimiteur pour la sélection par champs
-c, --characters	Sélection basée sur les caractères, délimiteur ignoré ou erreur
-s, - uniquement délimité	Supprimer les lignes sans caractère de délimitation (imprimé tel quel)
--complément	Sélection inversée (extraire tout *sauf les* champs / caractères spécifiés
--distributeur de sortie	Indiquez quand il doit être différent du délimiteur d'entrée

Remarques

1. Différences de syntaxe

Les options longues du tableau ci-dessus ne sont prises en charge que par la version GNU.

2. Aucun personnage ne bénéficie d'un traitement spécial

FreeBSD `cut` (fourni avec MacOS, par exemple) ne possède pas le commutateur `--complement` et, dans le cas des plages de caractères, on peut utiliser la commande `colrm` place:

```
$ cut --complement -c3-5 <<<"123456789"
126789

$ colrm 3 5 <<<"123456789"
126789
```

Cependant, il y a une grande différence, car `colrm` traite les caractères TAB (ASCII 9) comme des tabulations réelles jusqu'au multiple de huit suivant, et les backspaces (ASCII 8) de -1; au contraire, `cut` traite tous les caractères comme une colonne de large.

```
$ colrm  3 8 <<<$'12\tABCDEF' # Input string has an embedded TAB
12ABCDEF

$ cut --complement -c3-8 <<<$'12\tABCDEF'
12F
```

3. (Toujours pas) l'internationalisation

Lorsque la `cut` été conçue, tous les caractères duraient un octet et l'internationalisation n'était pas un problème. Lorsque les systèmes d'écriture avec des caractères plus larges sont devenus populaires, la solution adoptée par POSIX consistait à distinguer l'ancien commutateur `-c` , qui devrait conserver son sens de la sélection des *caractères,* peu importe le nombre d'octets, et introduire un nouveau commutateur `-b` sélectionnez des *octets,* quel que soit le codage de caractères actuel. Dans les implémentations les plus courantes, `-b` été introduit et fonctionne, mais `-c` fonctionne toujours exactement comme `-b` et pas comme il le devrait. Par exemple avec GNU `cut` :

Il semble que le filtre anti-spam de SE liste les textes anglais avec des caractères kanji isolés. Je ne pouvais pas surmonter cette limitation, alors les exemples suivants sont moins expressifs qu'ils pourraient l'être.

```
# In an encoding where each character in the input string is three bytes wide,
# Selecting bytes 1-6 yields the first two characters (correct)
$ LC_ALL=ja_JP.UTF-8 cut -b1-6 kanji.utf-8.txt
...first two characters of each line...


# Selecting all three characters with the -c switch doesn't work.
# It behaves like -b, contrary to documentation.
$ LC_ALL=ja_JP.UTF-8 cut -c1-3 kanji.utf-8.txt
...first character of each line...
```

```
# In this case, an illegal UTF-8 string is produced.
# The -n switch would prevent this, if implemented.
$ LC_ALL=ja_JP.UTF-8 cut -n -c2 kanji.utf-8.txt
...second byte, which is an illegal UTF-8 sequence...
```

Si vos personnages sont en dehors de la plage ASCII et que vous voulez utiliser `cut` , vous devez toujours être au courant de la largeur des caractères dans l' encodage et utilisez `-b` en conséquence. Si et quand `-c` commence à fonctionner comme documenté, vous n'aurez pas à modifier vos scripts.

4. Comparaisons de vitesse

Les restrictions de `cut` ont des doutes sur son utilité. En fait, les mêmes fonctionnalités peuvent être obtenues par des utilitaires plus puissants et plus populaires. Cependant, l'avantage de la `cut` est sa *performance* . Voir ci-dessous pour des comparaisons de vitesse. `test.txt` a trois millions de lignes, chacune contenant cinq champs séparés par des espaces. Pour le test `awk` , `mawk` été utilisé, car il est plus rapide que GNU `awk` . Le shell lui-même (dernière ligne) est de loin le moins performant. Les temps donnés (en secondes) sont ce que la commande de `time` donne en *temps réel* .

(Juste pour éviter les malentendus: toutes les commandes testées donnaient la même sortie avec l'entrée donnée, mais elles ne sont bien sûr pas équivalentes et donneraient des sorties différentes dans des situations différentes, notamment si les champs étaient délimités par un nombre variable d'espaces)

Commander	Temps
`cut -d ' ' -f1,2 test.txt`	1.138s
`awk '{print $1 $2}' test.txt`	1.688s
`join -a1 -o1.1,1.2 test.txt /dev/null`	1.767s
`perl -lane 'print "@F[1,2]"' test.txt`	11.390s
`grep -o '^\([^ ]*\) \([^ ]*\)' test.txt`	22.925s
`sed -e 's/^\([^ ]*\) \([^ ]*\).*$/\1 \2/' test.txt`	52.122s
`while read ab _; do echo $a $b; done <test.txt`	55.582s

5. Pages de manuel référentielles

- Groupe ouvert
- GNOU
- FreeBSD

Examples

Utilisation de base

L'utilisation typique est avec les fichiers de type CSV, où chaque ligne est composée de champs séparés par un délimiteur, spécifié par l'option `-d` . Le délimiteur par défaut est le caractère de tabulation. Supposons que vous ayez un fichier de données `data.txt` avec des lignes comme

```
0 0 755 1482941948.8024
102 33 4755 1240562224.3205
1003 1 644 1219943831.2367
```

alors

```
# extract the third space-delimited field
$ cut -d ' ' -f3 data.txt
755
4755
644

# extract the second dot-delimited field
$ cut -d. -f2 data.txt
8024
3205
2367

# extract the character range from the 20th through the 25th character
$ cut -c20-25 data.txt
948.80
056222
943831
```

Comme d'habitude, il peut y avoir des espaces optionnels entre un switch et son paramètre: `-d`, est le même que `-d ` ,

GNU `cut` permet de spécifier une option `--output-delimiter` : (une caractéristique indépendante de cet exemple est qu'un point-virgule en tant que délimiteur d'entrée doit être échappé pour éviter son traitement spécial par le shell)

```
$ cut --output-delimiter=, -d\; -f1,2 <<<"a;b;c;d"
a,b
```

Un seul caractère délimiteur

Vous ne pouvez pas avoir plus d'un delimiter: si vous spécifiez quelque chose comme `-d ",;:"` , certaines implémentations utiliseront uniquement le premier caractère comme séparateur (. Dans ce cas, la virgule) Autres mises en œuvre (par exemple GNU `cut`) donnera vous un message d'erreur.

```
$ cut -d ",;:" -f2 <<<"J.Smith,1 Main Road,cell:1234567890;land:4081234567"
cut: the delimiter must be a single character
Try `cut --help' for more information.
```

Les délimiteurs répétés sont interprétés comme des champs vides

```
$ cut -d, -f1,3 <<<"a,,b,c,d,e"
a,b
```

est assez évident, mais avec des chaînes délimitées par des espaces, il pourrait être moins évident pour certains

```
$ cut -d ' ' -f1,3 <<<"a  b c d e"
a b
```

`cut` ne peut pas être utilisé pour analyser les arguments comme le font le shell et les autres programmes.

Aucun devis

Il n'y a aucun moyen de protéger le délimiteur. Les feuilles de calcul et les logiciels de gestion de CSV similaires peuvent généralement reconnaître un caractère de citation de texte qui permet de définir des chaînes contenant un délimiteur. Avec la `cut` vous ne pouvez pas.

```
$ cut -d, -f3 <<<'John,Smith,"1, Main Street"'
"1
```

Extraire, ne pas manipuler

Vous ne pouvez extraire que des parties de lignes, pas réorganiser ni répéter des champs.

```
$ cut -d, -f2,1 <<<'John,Smith,USA' ## Just like -f1,2
John,Smith
$ cut -d, -f2,2 <<<'John,Smith,USA' ## Just like -f2
Smith
```

Lire La commande de coupe en ligne: https://riptutorial.com/fr/bash/topic/8762/la-commande-de-coupe

Chapitre 33: La portée

Examples

Portée dynamique en action

La portée dynamique signifie que les recherches de variables ont lieu dans la portée où une fonction est *appelée* , et non là où elle est *définie* .

```
$ x=3
$ func1 () { echo "in func1: $x"; }
$ func2 () { local x=9; func1; }
$ func2
in func1: 9
$ func1
in func1: 3
```

Dans un langage de portée lexicale, `func1` recherche *toujours* la valeur de `x` dans la portée globale, car `func1` est *défini* dans la portée locale.

Dans un langage à portée dynamique, `func1` regarde dans la portée où il est *appelé* . Lorsqu'il est appelé depuis `func2` , il cherche d'abord dans le corps de `func2` une valeur de `x` . Si elle n'y était pas définie, elle se situerait dans la portée globale, où `func2` était appelé.

Lire La portée en ligne: https://riptutorial.com/fr/bash/topic/2452/la-portee

Chapitre 34: Le débogage

Examples

Déboguer un script bash avec "-x"

Utilisez "-x" pour activer la sortie de débogage des lignes exécutées. Il peut être exécuté sur une session ou un script entier ou activé par programmation dans un script.

Exécutez un script avec la sortie de débogage activée:

```
$ bash -x myscript.sh
```

Ou

```
$ bash --debug myscript.sh
```

Activez le débogage dans un script bash. Il peut éventuellement être réactivé, bien que la sortie de débogage soit automatiquement réinitialisée à la fermeture du script.

```
#!/bin/bash
set -x    # Enable debugging
# some code here
set +x    # Disable debugging output.
```

Vérification de la syntaxe d'un script avec "-n"

L'option -n vous permet de vérifier la syntaxe d'un script sans avoir à l'exécuter:

```
~> $ bash -n testscript.sh
testscript.sh: line 128: unexpected EOF while looking for matching `"'
testscript.sh: line 130: syntax error: unexpected end of file
```

Déboguer usigh bashdb

Bashdb est un utilitaire similaire à gdb, dans la mesure où vous pouvez définir des points d'arrêt sur une ligne ou sur une fonction, imprimer du contenu de variables, relancer l'exécution du script, etc.

Vous pouvez normalement l'installer via votre gestionnaire de paquets, par exemple sur Fedora:

```
sudo dnf install bashdb
```

Ou l'obtenir de la page d'accueil . Ensuite, vous pouvez l'exécuter avec votre script en tant que paramètre:

```
bashdb <YOUR SCRIPT>
```

Voici quelques commandes pour vous aider à démarrer:

```
l - show local lines, press l again to scroll down
s - step to next line
print $VAR - echo out content of variable
restart - reruns bashscript, it re-loads it prior to execution.
eval - evaluate some custom command, ex: eval echo hi

b <line num> set breakpoint on some line
c - continue till some breakpoint
i b - info on break points
d <line #> - delete breakpoint at line #

shell - launch a sub-shell in the middle of execution, this is handy for manipulating
variables
```

Pour plus d'informations, je vous recommande de consulter le manuel:
http://www.rodericksmith.plus.com/outlines/manuals/bashdbOutline.html

Voir aussi la page d'accueil:
http://bashdb.sourceforge.net/

Lire Le débogage en ligne: https://riptutorial.com/fr/bash/topic/3655/le-debogage

Chapitre 35: Les fonctions

Syntaxe

- Définissez une fonction avec le mot-clé `function` :

```
function f {
```

}

- Définir une fonction avec `()` :

```
f(){
```

}

- Définissez une fonction avec le mot-clé `function` et `()` :

```
function f(){
```

}

Examples

Fonction simple

Dans `helloWorld.sh`

```
#!/bin/bash

# Define a function greet
greet ()
{
    echo "Hello World!"
}

# Call the function greet
greet
```

En exécutant le script, nous voyons notre message

```
$ bash helloWorld.sh
Hello World!
```

Notez que l' approvisionnement d' un fichier avec les fonctions qui les rend disponibles dans votre session actuelle de bash.

```
$ source helloWorld.sh   # or, more portably, ". helloWorld.sh"
$ greet
Hello World!
```

Vous pouvez `export` une fonction dans certains shells afin de l'exposer aux processus enfants.

```
bash -c 'greet'  # fails
export -f greet  # export function; note -f
bash -c 'greet'  # success
```

Fonctions avec arguments

Dans `helloJohn.sh` :

```
#!/bin/bash

greet() {
  local name="$1"
  echo "Hello, $name"
}

greet "John Doe"
```

```
# running above script
$ bash helloJohn.sh
Hello, John Doe
```

1. Si vous ne modifiez pas l'argument de quelque manière que ce soit, il n'est pas nécessaire de le copier dans une variable `local` - simplement en `echo "Hello, $1"` .

2. Vous pouvez utiliser `$1` , `$2` , `$3` , etc. pour accéder aux arguments de la fonction.

   > **Note:** pour les arguments de plus de 9 `$10` ne fonctionnera pas (bash le lira comme **$ 1** 0), vous devez faire `${10}` , `${11}` , etc.

3. `$@` fait référence à tous les arguments d'une fonction:

   ```
   #!/bin/bash
   foo() {
     echo "$@"
   }

   foo 1 2 3 # output => 1 2 3
   ```

   > **Note:** Vous devriez pratiquement toujours utiliser des guillemets doubles autour de `"$@"` , comme ici.

   Si vous omettez les guillemets, le shell développera des caractères génériques (même si l'utilisateur les a spécifiquement cités pour éviter cela) et introduira généralement des comportements indésirables, voire des problèmes de sécurité.

```
foo "string with spaces;" '$HOME' "*"
# output => string with spaces; $HOME *
```

4. pour les arguments par défaut, utilisez `${1:-default_val}` . Par exemple:

```
#!/bin/bash
foo() {
  local val=${1:-25}
  echo "$val"
}

foo     # output => 25
foo 30  # output => 30
```

5. pour exiger un argument, utilisez `${var:?error message}`

```
foo() {
  local val=${1:?Must provide an argument}
  echo "$val"
}
```

Valeur de retour d'une fonction

L'instruction de `return` dans Bash ne renvoie pas de valeur comme les fonctions C, mais quitte la fonction avec un statut de retour. Vous pouvez le considérer comme le statut de sortie de cette fonction.

Si vous voulez retourner une valeur de la fonction, envoyez la valeur à `stdout` comme ceci:

```
fun() {
    local var="Sample value to be returned"
    echo "$var"
    #printf "%s\n" "$var"
}
```

Maintenant, si vous faites:

```
var="$(fun)"
```

la sortie de `fun` sera stockée dans `$var` .

Gestion des indicateurs et des paramètres facultatifs

Les fonctions *getopts* intégrées peuvent être utilisées à l'intérieur de fonctions pour écrire des fonctions adaptées aux indicateurs et aux paramètres facultatifs. Cela ne présente pas de difficulté particulière, mais il faut bien gérer les valeurs touchées par les *getopts* . Par exemple, nous définissons une fonction *failwith* qui écrit un message sur *stderr* et quitte avec le code 1 ou un code arbitraire fourni en paramètre à l'option `-x` :

```
# failwith [-x STATUS] PRINTF-LIKE-ARGV
```

```
#  Fail with the given diagnostic message
#
# The -x flag can be used to convey a custom exit status, instead of
# the value 1.  A newline is automatically added to the output.

failwith()
{
    local OPTIND OPTION OPTARG status

    status=1
    OPTIND=1

    while getopts 'x:' OPTION; do
        case ${OPTION} in
            x)    status="${OPTARG}";;
            *)    1>&2 printf 'failwith: %s: Unsupported option.\n' "${OPTION}";;
        esac
    done

    shift $(( OPTIND - 1 ))
    {
        printf 'Failure: '
        printf "$@"
        printf '\n'
    } 1>&2
    exit "${status}"
}
```

Cette fonction peut être utilisée comme suit:

```
failwith '%s: File not found.' "${filename}"
failwith -x 70 'General internal error.'
```

etc.

Notez que pour *printf* , les variables ne doivent pas être utilisées comme premier argument. Si le message à imprimer comprend le contenu d'une variable, il faut utiliser le spécificateur `%s` pour l'imprimer, comme dans

```
failwith '%s' "${message}"
```

Le code de sortie d'une fonction est le code de sortie de sa dernière commande

Prenons l'exemple de cette fonction pour vérifier si un hôte est actif:

```
is_alive() {
    ping -c1 "$1" &> /dev/null
}
```

Cette fonction envoie un ping unique à l'hôte spécifié par le premier paramètre de fonction. La sortie et l'erreur de sortie de `ping` sont toutes les deux redirigées vers `/dev/null` , donc la fonction ne générera jamais rien. Mais la commande `ping` aura le code de sortie 0 en cas de succès et non

nul en cas d'échec. Comme il s'agit de la dernière commande (et dans cet exemple, la seule) de la fonction, le code de sortie de `ping` sera réutilisé pour le code de sortie de la fonction elle-même.

Ce fait est très utile dans les instructions conditionnelles.

Par exemple, si l'hôte `graucho` est `graucho` , connectez-vous avec `ssh` :

```
if is_alive graucho; then
    ssh graucho
fi
```

Un autre exemple: vérifiez jusqu'à ce que l'hôte `graucho` soit `graucho` , puis connectez-vous avec `ssh` :

```
while ! is_alive graucho; do
    sleep 5
done
ssh graucho
```

Imprimer la définition de la fonction

```
getfunc() {
    declare -f "$@"
}

function func(){
    echo "I am a sample function"
}

funcd="$(getfunc func)"
getfunc func # or echo "$funcd"
```

Sortie:

```
func ()
{
    echo "I am a sample function"
}
```

Une fonction qui accepte les paramètres nommés

```
foo() {
  while [[ "$#" -gt 0 ]]
  do
    case $1 in
      -f|--follow)
        local FOLLOW="following"
        ;;
      -t|--tail)
        local TAIL="tail=$2"
        ;;
    esac
    shift
```

```
  done

  echo "FOLLOW: $FOLLOW"
  echo "TAIL: $TAIL"
}
```

Exemple d'utilisation:

```
foo -f
foo -t 10
foo -f --tail 10
foo --follow --tail 10
```

Lire Les fonctions en ligne: https://riptutorial.com/fr/bash/topic/472/les-fonctions

Chapitre 36: Lire un fichier (flux de données, variable) ligne par ligne (et / ou champ par champ)?

Paramètres

Paramètre	Détails
IFS	Séparateur de champ interne
fichier	Un nom de fichier / chemin
`-r`	Empêche l'interprétation de la barre oblique inverse lorsqu'elle est utilisée avec read
`-t`	Supprime une nouvelle ligne de chaque ligne lue par `readarray`
`-d DELIM`	Continuer jusqu'à ce que le premier caractère de DELIM soit lu (avec `read`), plutôt que newline

Examples

Lit le fichier (/ etc / passwd) ligne par ligne et champ par champ

```
#!/bin/bash
FILENAME="/etc/passwd"
while IFS=: read -r username password userid groupid comment homedir cmdshell
do
  echo "$username, $userid, $comment $homedir"
done < $FILENAME
```

Dans le fichier de mots de passe Unix, les informations utilisateur sont stockées ligne par ligne, chaque ligne étant constituée d'informations pour un utilisateur séparé par deux points (:). Dans cet exemple, lors de la lecture ligne par ligne du fichier, la ligne est également divisée en champs utilisant le caractère deux-points comme délimiteur indiqué par la valeur donnée pour IFS.

Entrée de l'échantillon

```
mysql:x:27:27:MySQL Server:/var/lib/mysql:/bin/bash
pulse:x:497:495:PulseAudio System Daemon:/var/run/pulse:/sbin/nologin
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
tomcat:x:91:91:Apache Tomcat:/usr/share/tomcat6:/sbin/nologin
webalizer:x:67:67:Webalizer:/var/www/usage:/sbin/nologin
```

Échantillon sortie

```
mysql, 27, MySQL Server /var/lib/mysql
pulse, 497, PulseAudio System Daemon /var/run/pulse
sshd, 74, Privilege-separated SSH /var/empty/sshd
tomcat, 91, Apache Tomcat /usr/share/tomcat6
webalizer, 67, Webalizer /var/www/usage
```

Pour lire ligne par ligne et avoir la ligne entière affectée à la variable, voici une version modifiée de l'exemple. Notez que nous avons une seule variable par nom de ligne mentionnée ici.

```
#!/bin/bash
FILENAME="/etc/passwd"
while IFS= read -r line
do
  echo "$line"
done < $FILENAME
```

Entrée d'échantillon

```
mysql:x:27:27:MySQL Server:/var/lib/mysql:/bin/bash
pulse:x:497:495:PulseAudio System Daemon:/var/run/pulse:/sbin/nologin
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
tomcat:x:91:91:Apache Tomcat:/usr/share/tomcat6:/sbin/nologin
webalizer:x:67:67:Webalizer:/var/www/usage:/sbin/nologin
```

Échantillon sortie

```
mysql:x:27:27:MySQL Server:/var/lib/mysql:/bin/bash
pulse:x:497:495:PulseAudio System Daemon:/var/run/pulse:/sbin/nologin
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
tomcat:x:91:91:Apache Tomcat:/usr/share/tomcat6:/sbin/nologin
webalizer:x:67:67:Webalizer:/var/www/usage:/sbin/nologin
```

Lire les lignes d'un fichier dans un tableau

```
readarray -t arr <file
```

Ou avec une boucle:

```
arr=()
while IFS= read -r line; do
   arr+=("$line")
done <file
```

Traverser un fichier ligne par ligne

```
while IFS= read -r line; do
   echo "$line"
done <file
```

Si le fichier ne contient pas de nouvelle ligne à la fin, alors:

```
while IFS= read -r line || [ -n "$line" ]; do
   echo "$line"
done <file
```

Lire les lignes d'une chaîne dans un tableau

```
var='line 1
line 2
line3'
readarray -t arr <<< "$var"
```

ou avec une boucle:

```
arr=()
while IFS= read -r line; do
   arr+=("$line")
done <<< "$var"
```

En boucle à travers une chaîne ligne par ligne

```
var='line 1
line 2
line3'
while IFS= read -r line; do
   echo "-$line-"
done <<< "$var"
```

ou

```
readarray -t arr <<< "$var"
for i in "${arr[@]}";do
    echo "-$i-"
done
```

Faire défiler la sortie d'une ligne de commande ligne par ligne

```
while IFS= read -r line;do
    echo "**$line**"
done < <(ping google.com)
```

ou avec une pipe:

```
ping google.com |
while IFS= read -r line;do
    echo "**$line**"
done
```

Lire un champ de fichier par champ

Supposons que le séparateur de champs soit : (deux-points) dans le *fichier* .

```
while IFS= read -d : -r field || [ -n "$field" ]; do
   echo "$field"
done <file
```

Pour un contenu:

```
first : se
con
d:
    Thi rd:
    Fourth
```

La sortie est la suivante:

```
**first **
** se
con
d**
**
    Thi rd**
**
    Fourth
**
```

Lire un champ de chaîne par champ

Supposons que le séparateur de champs est :

```
var='line: 1
line: 2
line3'
while IFS= read -d : -r field || [ -n "$field" ]; do
   echo "-$field-"
done <<< "$var"
```

Sortie:

```
-line-
- 1
line-
- 2
line3
-
```

Lire les champs d'un fichier dans un tableau

Supposons que le séparateur de champs est :

```
arr=()
while IFS= read -d : -r field || [ -n "$field" ]; do
```

```
   arr+=("$field")
done <file
```

Lire les champs d'une chaîne dans un tableau

Supposons que le séparateur de champs est :

```
var='1:2:3:4:
newline'
arr=()
while IFS= read -d : -r field || [ -n "$field" ]; do
   arr+=("$field")
done <<< "$var"
echo "${arr[4]}"
```

Sortie:

```
newline
```

En boucle à travers la sortie d'un champ de commande par champ

Supposons que le séparateur de champs est :

```
while IFS= read -d : -r field || [ -n "$field" ];do
    echo "**$field**"
done < <(ping google.com)
```

Ou avec une pipe:

```
ping google.com | while IFS= read -d : -r field || [ -n "$field" ];do
    echo "**$field**"
done
```

Lire Lire un fichier (flux de données, variable) ligne par ligne (et / ou champ par champ)? en ligne: https://riptutorial.com/fr/bash/topic/5473/lire-un-fichier--flux-de-donnees--variable--ligne-par-ligne--et---ou-champ-par-champ--

Chapitre 37: Liste des fichiers

Syntaxe

- ls [OPTION] ... [FICHIER] ...

Paramètres

Option	La description
`-a` , `--all`	Liste toutes les entrées, y compris celles qui commencent par un point
`-A` , `-A --almost-all`	Recenser toutes les entrées à l' exclusion `.` et `..`
`-c`	Trier les fichiers par heure de modification
`-d` , `--directory`	Lister les entrées du répertoire
`-h` , `--human-readable`	Afficher les tailles dans un format lisible par l'homme (c.-à-d. `K` , `M`)
`-H`	Même que ci-dessus seulement avec des puissances de 1000 au lieu de 1024
`-l`	Afficher le contenu au format de liste longue
`-o`	Format longue liste sans information de groupe
`-r` , `--reverse`	Afficher le contenu dans l'ordre inverse
`-s` , `--size`	Taille d'impression de chaque fichier en blocs
`-S`	Trier par taille de fichier
`--sort=WORD`	Trier le contenu par un mot. (c.-à-d. taille, version, statut)
`-t`	Trier par heure de modification
`-u`	Trier par dernière heure d'accès
`-v`	Trier par version
`-1`	Liste un fichier par ligne

Examples

Liste de fichiers

La commande `ls` répertorie le contenu d'un répertoire spécifié, à l' **exception des** fichiers dotfiles. Si aucun répertoire n'est spécifié, le contenu du répertoire actuel est répertorié par défaut.

Les fichiers répertoriés sont triés par ordre alphabétique, par défaut et alignés dans des colonnes s'ils ne tiennent pas sur une seule ligne.

```
$ ls
apt  configs  Documents  Fonts    Music      Programming  Templates  workspace
bin  Desktop  eclipse    git      Pictures   Public       Videos
```

Liste des fichiers dans un format de liste longue

L'option `-l` la commande `ls` imprime le contenu d'un répertoire spécifié dans un format de liste long. Si aucun répertoire n'est spécifié, le contenu du répertoire actuel est répertorié par défaut.

```
ls -l /etc
```

Exemple de sortie:

```
total 1204
drwxr-xr-x  3 root root    4096 Apr 21 03:44 acpi
-rw-r--r--  1 root root    3028 Apr 21 03:38 adduser.conf
drwxr-xr-x  2 root root    4096 Jun 11 20:42 alternatives
...
```

La sortie affiche d'abord le `total` , qui indique la taille totale en **blocs** de tous les fichiers du répertoire répertorié. Il affiche ensuite huit colonnes d'informations pour chaque fichier du répertoire répertorié. Voici les détails pour chaque colonne dans la sortie:

Colonne No.	Exemple	La description
1.1	`d`	Type de fichier (voir tableau ci-dessous)
1.2	`rwxr-xr-x`	Chaîne d'autorisation
2	`3`	Nombre de liens durs
3	`root`	Le nom du propriétaire
4	`root`	Groupe de propriétaires
5	`4096`	Taille du fichier en octets
6	`Apr 21 03:44`	Heure de modification
7	`acpi`	Nom de fichier

Type de fichier

Le type de fichier peut être l'un des caractères suivants.

Personnage	Type de fichier
`-`	Fichier régulier
`b`	Bloquer le fichier spécial
`c`	Fichier spécial de personnage
`C`	Fichier haute performance ("données contiguës")
`d`	Annuaire
`D`	Porte (fichier IPC spécial dans Solaris 2.5+ uniquement)
`l`	Lien symbolique
`M`	Fichier hors ligne ("migré") (Cray DMF)
`n`	Fichier spécial réseau (HP-UX)
`p`	FIFO (tuyau nommé)
`P`	Port (fichier système spécial dans Solaris 10+ uniquement)
`s`	Prise
`?`	Un autre type de fichier

Liste des fichiers triés par taille

L'option `-S` la commande `ls` trie les fichiers par ordre décroissant de taille de fichier.

```
$ ls -l -S ./Fruits
total 444
-rw-rw-rw- 1 root root 295303 Jul 28 19:19 apples.jpg
-rw-rw-rw- 1 root root 102283 Jul 28 19:19 kiwis.jpg
-rw-rw-rw- 1 root root  50197 Jul 28 19:19 bananas.jpg
```

Lorsqu'il est utilisé avec l'option `-r` , l'ordre de tri est inversé.

```
$ ls -l -S -r /Fruits
total 444
-rw-rw-rw- 1 root root  50197 Jul 28 19:19 bananas.jpg
-rw-rw-rw- 1 root root 102283 Jul 28 19:19 kiwis.jpg
-rw-rw-rw- 1 root root 295303 Jul 28 19:19 apples.jpg
```

Liste des fichiers sans utiliser `ls`

Utilisez les fonctions d' extension de nom de fichier du shell Bash et d' extension pour obtenir les noms de fichiers:

```
# display the files and directories that are in the current directory
printf "%s\n" *

# display only the directories in the current directory
printf "%s\n" */

# display only (some) image files
printf "%s\n" *.{gif,jpg,png}
```

Pour capturer une liste de fichiers dans une variable à traiter, il est généralement recommandé d'utiliser un tableau bash :

```
files=( * )

# iterate over them
for file in "${files[@]}"; do
    echo "$file"
done
```

Liste des dix fichiers les plus récemment modifiés

Le tableau suivant répertorie jusqu'à dix des fichiers les plus récemment modifiés dans le répertoire en cours, en utilisant un format de liste long (`-l`) et trié par heure (`-t`).

```
ls -lt | head
```

Liste tous les fichiers, y compris les fichiers Dotfiles

Un fichier de **points** est un fichier dont le nom commence par a `.` . Celles-ci sont normalement cachées par `ls` et ne sont pas listées, sauf si elles sont demandées.

Par exemple, la sortie suivante de `ls` :

```
$ ls
bin  pki
```

L'option `-a` ou `--all` listera tous les fichiers, y compris les fichiers dot.

```
$ ls -a
.   .ansible        .bash_logout   .bashrc  .lesshst   .puppetlabs  .viminfo
..  .bash_history   .bash_profile  bin      pki        .ssh
```

L'option `-A` ou `--almost-all` répertorie tous les fichiers, y compris les fichiers de points, mais n'indique pas de liste implicite `.` et `..` Notez cela `.` est le répertoire actuel et `..` est le répertoire parent.

```
$ ls -A
.ansible        .bash_logout    .bashrc  .lesshst  .puppetlabs  .viminfo
.bash_history   .bash_profile   bin      pki       .ssh
```

Liste des fichiers dans un format arborescent

La commande d' `tree` répertorie le contenu d'un répertoire spécifié dans un format arborescent. Si aucun répertoire n'est spécifié, le contenu du répertoire actuel est répertorié par défaut.

Exemple de sortie:

```
$ tree /tmp
/tmp
├── 5037
├── adb.log
└── evince-20965
    └── image.FPWTJY.png
```

Utilisez l'option `-L` la commande `tree` pour limiter la profondeur d'affichage et l'option `-d` pour ne lister que les répertoires.

Exemple de sortie:

```
$ tree -L 1 -d /tmp
/tmp
└── evince-20965
```

Lire Liste des fichiers en ligne: https://riptutorial.com/fr/bash/topic/366/liste-des-fichiers

Chapitre 38: Math

Examples

Math utilisant dc

`dc` est l'une des plus anciennes langues sur Unix.

Il utilise la notation polonaise inversée , ce qui signifie que vous empilez d'abord des nombres, puis des opérations. Par exemple `1+1` est écrit comme `1 1+` .

Pour imprimer un élément depuis le haut de la pile, utilisez la commande `p`

```
echo '2 3 + p' | dc
5

or

dc <<< '2 3 + p'
5
```

Vous pouvez imprimer l'élément supérieur plusieurs fois

```
dc <<< '1 1 + p 2 + p'
2
4
```

Pour les nombres négatifs, utilisez _ préfixe

```
dc <<< '_1 p'
-1
```

Vous pouvez également utiliser des lettres majuscules de `A to F` pour les nombres compris entre `10 and 15` et . comme point décimal

```
dc <<< 'A.4 p'
10.4
```

`dc` utilise une précision différente, ce qui signifie que la précision est limitée uniquement par la mémoire disponible. Par défaut, la précision est définie sur 0 décimale

```
dc <<< '4 3 / p'
1
```

Nous pouvons augmenter la précision en utilisant la commande `k` . `2k` utilisera

```
dc <<< '2k 4 3 / p'
1.33
```

```
dc <<< '4k 4 3 / p'
1.3333
```

Vous pouvez également l'utiliser sur plusieurs lignes

```
dc << EOF
1 1 +
3 *
p
EOF
6
```

`bc` est un préprocesseur pour `dc` .

Math utilisant bc

`bc` est un langage de calcul arbitraire de précision. Il pourrait être utilisé de manière interactive ou être exécuté à partir de la ligne de commande.

Par exemple, il peut imprimer le résultat d'une expression:

```
echo '2 + 3' | bc
5

echo '12 / 5' | bc
2
```

Pour l'arithmétique flottante, vous pouvez importer la bibliothèque standard `bc -l` :

```
echo '12 / 5' | bc -l
2.40000000000000000000
```

Il peut être utilisé pour comparer des expressions:

```
echo '8 > 5' | bc
1

echo '10 == 11' | bc
0

echo '10 == 10 && 8 > 3' | bc
1
```

Math utilisant les capacités de bash

Le calcul arithmétique peut également être effectué sans impliquer d’autres programmes comme celui-ci:

Multiplication:

```
echo $((5 * 2))
10
```

Division:

```
echo $((5 / 2))
2
```

Modulo:

```
echo $((5 % 2))
1
```

Exponentiation:

```
echo $((5 ** 2))
25
```

Math utilisant expr

`expr` ou `Evaluate expressions` évalue une expression et écrit le résultat sur une sortie standard

Arithmétique de base

```
expr 2 + 3
5
```

En multipliant, vous devez échapper au signe `*`

```
expr 2 \* 3
6
```

Vous pouvez également utiliser des variables

```
a=2
expr $a + 3
5
```

Gardez à l'esprit qu'il ne supporte que les entiers, donc l'expression comme ça

```
expr 3.0 / 2
```

va lancer une erreur `expr: not a decimal number: '3.0'`.

Il prend en charge les expressions régulières pour faire correspondre les modèles

```
expr 'Hello World' : 'Hell\(.*\)rld'
o Wo
```

Ou trouvez l'index du premier caractère dans la chaîne de recherche

> Cela produira `expr: syntax error` sous **Mac OS X** , car elle utilise **BSD expr** qui n'a pas la commande index, alors que expr sous Linux est généralement **GNU expr**

```
expr index hello l
3

expr index 'hello' 'lo'
3
```

Lire Math en ligne: https://riptutorial.com/fr/bash/topic/2086/math

Chapitre 39: Mise en réseau avec Bash

Introduction

Bash est souvent utilisé dans la gestion et la maintenance des serveurs et des clusters. Les informations relatives aux commandes typiques utilisées par les opérations réseau, quand utiliser quelle commande pour quel usage, et des exemples / exemples d'applications uniques et / ou intéressantes doivent être inclus

Examples

Commandes réseau

```
ifconfig
```

La commande ci-dessus affiche toutes les interfaces actives de la machine et donne également les informations de

1. Adresse IP attribuer à l'interface
2. Adresse MAC de l'interface
3. Adresse de diffusion
4. Transmettre et recevoir des octets

Un exemple

```
ifconfig -a
```

La commande ci-dessus montre également l'interface de désactivation

```
ifconfig eth0
```

La commande ci-dessus ne montrera que l'interface eth0

```
ifconfig eth0 192.168.1.100 netmask 255.255.255.0
```

La commande ci-dessus affectera l'adresse IP statique à l'interface eth0

```
ifup eth0
```

La commande ci-dessus activera l'interface eth0

```
ifdown eth0
```

La commande ci-dessous désactive l'interface eth0

```
ping
```

La commande ci-dessus (Packet Internet Grouper) permet de tester la connectivité entre les deux nœuds

```
ping -c2 8.8.8.8
```

La commande ci-dessus envoie une commande ping ou teste la connectivité avec le serveur Google pendant 2 secondes.

```
traceroute
```

La commande ci-dessus est à utiliser lors du dépannage pour déterminer le nombre de sauts effectués pour atteindre la destination.

```
netstat
```

La commande ci-dessus (statistiques du réseau) donne les informations de connexion et leur état

```
dig www.google.com
```

La commande ci-dessus (groupeur d'informations de domaine) interroge les informations relatives au DNS

```
nslookup www.google.com
```

La commande ci-dessus interroge le DNS et trouve l'adresse IP correspondant au nom du site Web.

```
route
```

La commande ci-dessus est utilisée pour vérifier les informations d'itinéraire NetWrok. Il vous montre essentiellement la table de routage

```
router add default gw 192.168.1.1 eth0
```

La commande ci-dessus ajoutera la route par défaut du réseau de l'interface eth0 à 192.168.1.1 dans la table de routage.

```
route del default
```

La commande ci-dessus supprime la route par défaut de la table de routage

Lire Mise en réseau avec Bash en ligne: https://riptutorial.com/fr/bash/topic/10737/mise-en-reseau-avec-bash

Chapitre 40: Modèles de conception

Introduction

Réaliser des modèles de conception courants dans Bash

Examples

Le modèle Publish / Subscribe (Pub / Sub)

Lorsqu'un projet Bash se transforme en bibliothèque, il peut s'avérer difficile d'ajouter de nouvelles fonctionnalités. Les noms de fonctions, les variables et les paramètres doivent généralement être modifiés dans les scripts qui les utilisent. Dans des scénarios comme celui-ci, il est utile de découpler le code et d'utiliser un modèle de conception piloté par les événements. Dans ce modèle, un script externe peut s'abonner à un événement. Lorsque cet événement est déclenché (publié), le script peut exécuter le code qu'il a enregistré avec l'événement.

pubsub.sh:

```
#!/usr/bin/env bash

#
# Save the path to this script's directory in a global env variable
#
DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"

#
# Array that will contain all registered events
#
EVENTS=()

function action1() {
    echo "Action #1 was performed ${2}"
}

function action2() {
    echo "Action #2 was performed"
}

#
# @desc   :: Registers an event
# @param  :: string $1 - The name of the event. Basically an alias for a function name
# @param  :: string $2 - The name of the function to be called
# @param  :: string $3 - Full path to script that includes the function being called
#
function subscribe() {
    EVENTS+=("${1};${2};${3}")
}

#
# @desc   :: Public an event
# @param  :: string $1 - The name of the event being published
```

```
    #
    function publish() {
        for event in ${EVENTS[@]}; do
            local IFS=";"
            read -r -a event <<< "$event"
            if [[  "${event[0]}" ==  "${1}" ]]; then
                ${event[1]} "$@"
            fi
        done
    }

    #
    # Register our events and the functions that handle them
    #
    subscribe "/do/work"           "action1" "${DIR}"
    subscribe "/do/more/work"      "action2" "${DIR}"
    subscribe "/do/even/more/work" "action1" "${DIR}"

    #
    # Execute our events
    #
    publish "/do/work"
    publish "/do/more/work"
    publish "/do/even/more/work" "again"
```

Courir:

```
chmod +x pubsub.sh
./pubsub.sh
```

Lire Modèles de conception en ligne: https://riptutorial.com/fr/bash/topic/9531/modeles-de-conception

Chapitre 41: Parallèle

Introduction

Les travaux dans GNU Linux peuvent être parallélisés en utilisant GNU Parallel. Un travail peut être une commande unique ou un petit script à exécuter pour chacune des lignes de l'entrée. L'entrée type est une liste de fichiers, une liste d'hôtes, une liste d'utilisateurs, une liste d'URL ou une liste de tables. Un travail peut également être une commande qui lit à partir d'un tube.

Syntaxe

1. parallel [options] [commande [arguments]] <list_of_arguments>

Paramètres

Option	La description
`-jn`	Exécutez n jobs en parallèle
`-k`	Garder le même ordre
`-X`	Plusieurs arguments avec remplacement du contexte
`--colsep regexp`	Fractionner l'entrée sur regexp pour les remplacements de position
`{} {.} {/} {/.} {#}`	Cordes de remplacement
`{3} {3.} {3/} {3/.}`	Chaînes de remplacement positionnelles
`-S sshlogin`	`Example: foo@server.example.com`
`--trc {}.bar`	Abréviation de --transfer --return {} .bar --cleanup
`--onall`	Exécuter la commande donnée avec un argument sur tous les sshlogins
`--nonall`	Exécuter la commande donnée sans arguments sur tous les sshlogins
`--pipe`	Split stdin (entrée standard) à plusieurs tâches.
`--recend str`	Séparateur de fin d'enregistrement pour --pipe.
`--recstart str`	Séparateur de début d'enregistrement pour --pipe.

Examples

Paralléliser les tâches répétitives sur la liste des fichiers

De nombreuses tâches répétitives peuvent être effectuées plus efficacement si vous utilisez davantage de ressources de votre ordinateur (c.-à-d. Les processeurs et la RAM). Vous trouverez ci-dessous un exemple d'exécution de plusieurs tâches en parallèle.

Supposons que vous ayez une `< list of files >` , disons la sortie de `ls` . De plus, laissez ces fichiers bz2 compressés et réglez-les dans l'ordre suivant.

1. Décompressez les fichiers bz2 en utilisant `bzcat` pour stdout
2. Lignes Grep (par exemple, filtre) avec des mots-clés spécifiques utilisant `grep <some key word>`
3. Pipe la sortie à concaténer en un seul fichier gzippé en utilisant `gzip`

Exécuter ceci en utilisant une boucle while peut ressembler à ceci

```
filenames="file_list.txt"
while read -r line
do
name="$line"
     ## grab lines with puppies in them
     bzcat $line | grep puppies | gzip >> output.gz
done < "$filenames"
```

En utilisant GNU Parallel, nous pouvons exécuter 3 travaux parallèles en même temps en faisant simplement

```
parallel -j 3 "bzcat {} | grep puppies" ::: $( cat filelist.txt ) | gzip > output.gz
```

Cette commande est simple, concise et plus efficace lorsque le nombre de fichiers et la taille du fichier sont importants. Les jobs sont initiés en `parallel` , l'option `-j 3` lance 3 jobs parallèles et l'entrée dans les jobs parallèles est effectuée par `:::` . La sortie est finalement acheminée vers `gzip > output.gz`

Paralléliser STDIN

Maintenant, imaginons que nous avons un fichier volumineux (par exemple 30 Go) qui doit être converti ligne par ligne. Disons que nous avons un script, `convert.sh` , qui fait cela `<task>` . Nous pouvons canaliser le contenu de ce fichier en stdin pour que parallel soit utilisé et avec des *morceaux* tels que

```
<stdin> | parallel --pipe --block <block size> -k <task> > output.txt
```

où `<stdin>` peut provenir de quelque chose comme `cat <file>` .

Comme exemple reproductible, notre tâche sera `nl -n rz` . Prenez n'importe quel fichier, le mien sera `data.bz2` et passez-le à `<stdin>`

```
bzcat data.bz2 | nl | parallel --pipe --block 10M -k nl -n rz | gzip > ouptput.gz
```

L'exemple ci-dessus prend `<stdin>` partir de `bzcat data.bz2 | nl` , où `output.gz nl` juste comme preuve de concept que la sortie finale `output.gz` sera enregistrée dans l'ordre de réception. Ensuite, `parallel` divise le `<stdin>` en morceaux de taille 10 Mo, et pour chaque segment, il passe par `nl -n rz` où il ajoute simplement un nombre justifié (voir `nl --help` pour plus de détails). Les options `--pipe` indique à `parallel` de séparer `<stdin>` en plusieurs tâches et `-- block` spécifie la taille des blocs. L'option `-k` spécifie que le classement doit être maintenu.

Votre résultat final devrait ressembler à quelque chose comme

```
000001       1  <data>
000002       2  <data>
000003       3  <data>
000004       4  <data>
000005       5  <data>
 ...
000587  552409  <data>
000588  552410  <data>
000589  552411  <data>
000590  552412  <data>
000591  552413  <data>
```

Mon fichier original comportait 552 413 lignes. La première colonne représente les travaux parallèles et la deuxième colonne représente la numérotation de la ligne d'origine transmise en `parallel` par blocs. Vous devez noter que l'ordre dans la deuxième colonne (et le reste du fichier) est conservé.

Lire Parallèle en ligne: https://riptutorial.com/fr/bash/topic/10778/parallele

Chapitre 42: Personnalisation de PS1

Examples

Modifier l'invite PS1

Pour changer PS1, il suffit de changer la valeur de la variable shell PS1. La valeur peut être définie dans le fichier `~/.bashrc` ou `/etc/bashrc` , en fonction de la distribution. PS1 peut être changé en n'importe quel texte simple comme:

```
PS1="hello "
```

Outre le texte en clair, un certain nombre de caractères spéciaux avec antislash sont pris en charge:

Format	action
`\a`	un caractère de sonnerie ASCII (07)
`\d`	la date au format «Jour de la semaine» (par exemple, «Mardi 26 mai»)
`\D{format}`	le format est passé à strftime (3) et le résultat est inséré dans la chaîne d'invite; un format vide entraîne une représentation temporelle spécifique à l'environnement local. Les accolades sont obligatoires
`\e`	un caractère d'échappement ASCII (033)
`\h`	le nom d'hôte jusqu'au premier '.'
`\H`	le nom d'hôte
`\j`	le nombre d'emplois actuellement gérés par le shell
`\l`	le nom de base du nom de périphérique du terminal du shell
`\n`	nouvelle ligne
`\r`	retour de chariot
`\s`	le nom du shell, le nom de base de 0 $ (la partie qui suit la barre oblique finale)
`\t`	l'heure actuelle en 24 heures Format HH: MM: SS
`\T`	l'heure actuelle en 12 heures HH: MM: SS format
`\@`	l'heure actuelle au format 12 heures / heure
`\A`	l'heure actuelle en format 24 heures HH: MM

Format	action
`\u`	le nom d'utilisateur de l'utilisateur actuel
`\v`	la version de bash (par exemple, 2.00)
`\V`	la sortie de bash, version + niveau du patch (par exemple, 2.00.0)
`\w`	le répertoire de travail actuel, avec $ HOME abrégé avec un tilde
`\W`	le nom de base du répertoire de travail en cours, avec $ HOME abrégé avec un tilde
`\!`	le numéro d'historique de cette commande
`\#`	le numéro de commande de cette commande
`\$`	si l'UID effectif est 0, un #, sinon un $
`\nnn*`	le caractère correspondant au nombre octal nnn
`\`	une barre oblique inverse
`\[`	commencer une séquence de caractères non imprimables, qui pourrait être utilisée pour incorporer une séquence de contrôle de terminal dans l'invite
`\]`	terminer une séquence de caractères non imprimables

Ainsi, par exemple, nous pouvons configurer PS1 pour:

```
PS1="\u@\h:\w\$ "
```

Et ça va sortir:

> utilisateur @ machine: ~ $

Afficher une branche git en utilisant PROMPT_COMMAND

Si vous vous trouvez dans un dossier d'un dépôt git, il peut être intéressant de montrer la branche actuelle sur laquelle vous vous trouvez. Dans `~/.bashrc` ou `/etc/bashrc` ajoutez ce qui suit (git est requis pour que cela fonctionne):

```
function prompt_command {
    # Check if we are inside a git repository
    if git status > /dev/null 2>&1; then
        # Only get the name of the branch
        export GIT_STATUS=$(git status | grep 'On branch' | cut -b 10-)
    else
        export GIT_STATUS=""
    fi
}
```

```
# This function gets called every time PS1 is shown
PROMPT_COMMAND=prompt_command

PS1="\$GIT_STATUS \u@\h:\w\$ "
```

Si nous sommes dans un dossier dans un dépôt git, cela va générer:

    utilisateur de la branche @ machine: ~ $

Et si nous sommes dans un dossier normal:

    utilisateur @ machine: ~ $

Afficher le nom de la branche git à l'invite du terminal

Vous pouvez avoir des fonctions dans la variable PS1, assurez-vous simplement de la citer seule ou utilisez la commande escape pour les caractères spéciaux:

```
gitPS1(){
    gitps1=$(git branch 2>/dev/null | grep '*')
    gitps1="${gitps1:+ (${gitps1/#\* /})}"
    echo "$gitps1"
}

PS1='\u@\h:\w$(gitPS1)$ '
```

Il vous donnera une invite comme celle-ci:

```
user@Host:/path (master)$
```

Remarques:

- Apportez les modifications dans le fichier `~/.bashrc` ou `/etc/bashrc` ou `~/.bash_profile` ou `~./profile` (selon le système d'exploitation) et enregistrez-le.
- Exécutez le `source ~/.bashrc` (spécifique à la distribution) après avoir enregistré le fichier.

Afficher l'heure dans l'invite du terminal

```
timeNow(){
    echo "$(date +%r)"
}
PS1='[$(timeNow)] \u@\h:\w$ '
```

Il vous donnera une invite comme celle-ci:

```
[05:34:37 PM] user@Host:/path$
```

Remarques:

- Apportez les modifications dans le fichier `~/.bashrc` ou `/etc/bashrc` ou `~/.bash_profile` ou `~./profile`

(selon le système d'exploitation) et enregistrez-le.

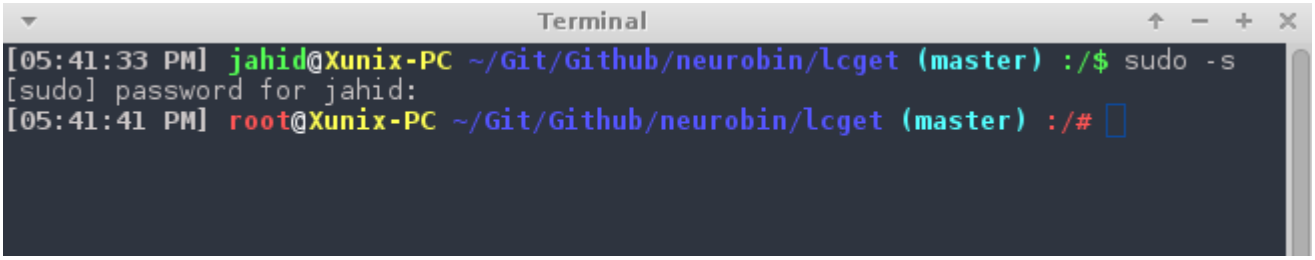
- Exécutez le `source ~/.bashrc` (spécifique à la distribution) après avoir enregistré le fichier.

Coloriser et personnaliser l'invite du terminal

Voici comment l'auteur définit sa variable `PS1` personnelle:

```
gitPS1(){
    gitps1=$(git branch 2>/dev/null | grep '*')
    gitps1="${gitps1:+ (${gitps1/#\* /})}"
    echo "$gitps1"
}
#Please use the below function if you are a mac user
gitPS1ForMac(){
    git branch 2> /dev/null | sed -e '/^[^*]/d' -e 's/* \(.*\)/ (\1)/'
}
timeNow(){
    echo "$(date +%r)"
}
if [ "$color_prompt" = yes ]; then
  if [ x$EUID = x0 ]; then
    PS1='\[\033[1;38m\][$(timeNow)]\[\033[00m\]
\[\033[1;31m\]\u\[\033[00m\]\[\033[1;37m\]@\[\033[00m\]\[\033[1;33m\]\h\[\033[00m\]
\[\033[1;34m\]\w\[\033[00m\]\[\033[1;36m\]$(gitPS1)\[\033[00m\] \[\033[1;31m\]:/#\[\033[00m\]
'
  else
    PS1='\[\033[1;38m\][$(timeNow)]\[\033[00m\]
\[\033[1;32m\]\u\[\033[00m\]\[\033[1;37m\]@\[\033[00m\]\[\033[1;33m\]\h\[\033[00m\]
\[\033[1;34m\]\w\[\033[00m\]\[\033[1;36m\]$(gitPS1)\[\033[00m\] \[\033[1;32m\]:/$\[\033[00m\]
'
  fi
else
    PS1='[$(timeNow)] \u@\h \w$(gitPS1) :/$ '
fi
```

Et voici à quoi ressemble mon invite:



Référence de couleur:

```
# Colors
txtblk='\e[0;30m' # Black - Regular
txtred='\e[0;31m' # Red
txtgrn='\e[0;32m' # Green
txtylw='\e[0;33m' # Yellow
txtblu='\e[0;34m' # Blue
txtpur='\e[0;35m' # Purple
txtcyn='\e[0;36m' # Cyan
txtwht='\e[0;37m' # White
bldblk='\e[1;30m' # Black - Bold
```

```
bldred='\e[1;31m' # Red
bldgrn='\e[1;32m' # Green
bldylw='\e[1;33m' # Yellow
bldblu='\e[1;34m' # Blue
bldpur='\e[1;35m' # Purple
bldcyn='\e[1;36m' # Cyan
bldwht='\e[1;37m' # White
unkblk='\e[4;30m' # Black - Underline
undred='\e[4;31m' # Red
undgrn='\e[4;32m' # Green
undylw='\e[4;33m' # Yellow
undblu='\e[4;34m' # Blue
undpur='\e[4;35m' # Purple
undcyn='\e[4;36m' # Cyan
undwht='\e[4;37m' # White
bakblk='\e[40m'   # Black - Background
bakred='\e[41m'   # Red
badgrn='\e[42m'   # Green
bakylw='\e[43m'   # Yellow
bakblu='\e[44m'   # Blue
bakpur='\e[45m'   # Purple
bakcyn='\e[46m'   # Cyan
bakwht='\e[47m'   # White
txtrst='\e[0m'    # Text Reset
```

Remarques:

- Apportez les modifications dans le fichier `~/.bashrc` ou `/etc/bashrc` ou `~/.bash_profile` ou `~./profile` (selon le système d'exploitation) et enregistrez-le.
- Pour `root` vous devrez peut-être également modifier le fichier `/etc/bash.bashrc` ou `/root/.bashrc`
- Exécutez le `source ~/.bashrc` (spécifique à la distribution) après avoir enregistré le fichier.
- Remarque: si vous avez enregistré les modifications dans `~/.bashrc` , n'oubliez pas d'ajouter la `source ~/.bashrc` dans votre `~/.bash_profile` pour que cette modification de `PS1` soit enregistrée à chaque démarrage de l'application Terminal.

Afficher le statut et l'heure de retour de la commande précédente

Parfois, nous avons besoin d'un indice visuel pour indiquer le statut de retour de la commande précédente. L'extrait suivant le met en tête de la PS1.

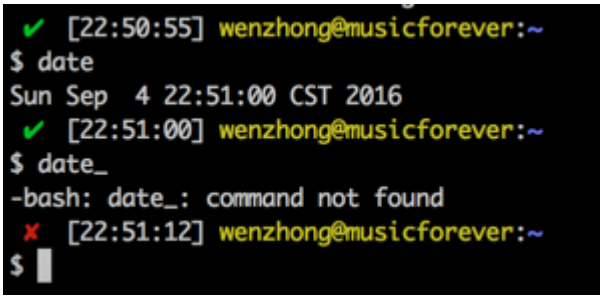
Notez que la fonction __stat () devrait être appelée à chaque fois qu'une nouvelle PS1 est générée, sinon elle resterait liée au statut de retour de la dernière commande de votre fichier .bashrc ou .bash_profile.

```
# -ANSI-COLOR-CODES- #
Color_Off="\033[0m"
###-Regular-###
Red="\033[0;31m"
Green="\033[0;32m"
Yellow="\033[0;33m"
####-Bold-####
```

```
function __stat() {
    if [ $? -eq 0 ]; then
        echo -en "$Green ✔ $Color_Off "
  else
     echo -en "$Red ✘ $Color_Off "
  fi
}

PS1='$(__stat)'
PS1+="[\t] "
PS1+="\e[0;33m\u@\h\e[0m:\e[1;34m\w\e[0m \n$ "

export PS1
```



Lire Personnalisation de PS1 en ligne: https://riptutorial.com/fr/bash/topic/3340/personnalisation-de-ps1

Chapitre 43: Pièges

Examples

Espaces blancs lors de l'attribution de variables

Les espaces ont leur importance lors de l'attribution des variables.

```
foo = 'bar' # incorrect
foo= 'bar'  # incorrect
foo='bar'   # correct
```

Les deux premières entraîneront des erreurs de syntaxe (ou pire, l'exécution d'une commande incorrecte). Le dernier exemple définira correctement la variable `$foo` sur le texte "bar".

Manquer la dernière ligne dans un fichier

La norme C stipule que les fichiers doivent se terminer par une nouvelle ligne. Par conséquent, si EOF arrive en fin de ligne, certaines commandes risquent de ne pas manquer cette ligne. Par exemple:

```
$ echo 'one\ntwo\nthree\c' > file.txt

$ cat file.txt
one
two
three

$ while read line ; do echo "line $line" ; done < file.txt
one
two
```

Pour vous assurer que cela fonctionne correctement dans l'exemple ci-dessus, ajoutez un test pour qu'il continue la boucle si la dernière ligne n'est pas vide.

```
$ while read line || [ -n "$line" ] ; do echo "line $line" ; done < file.txt
one
two
three
```

Les commandes en échec n'arrêtent pas l'exécution du script

Dans la plupart des langages de script, si un appel de fonction échoue, il peut déclencher une exception et arrêter l'exécution du programme. Les commandes Bash n'ont pas d'exceptions, mais elles ont des codes de sortie. Un code de sortie non nul signale un échec, cependant, un code de sortie différent de zéro n'arrêtera pas l'exécution du programme.

Cela peut conduire à des situations dangereuses (bien que forcément artificielles) comme ceci:

```
#!/bin/bash
cd ~/non/existent/directory
rm -rf *
```

Si `cd` -ing dans ce répertoire échoue, Bash ignorera l'échec et passera à la commande suivante, en nettoyant le répertoire à partir duquel vous avez exécuté le script.

La meilleure façon de résoudre ce problème consiste à utiliser la commande set :

```
#!/bin/bash
set -e
cd ~/non/existent/directory
rm -rf *
```

`set -e` indique à Bash de quitter le script immédiatement si une commande renvoie un statut différent de zéro.

Lire Pièges en ligne: https://riptutorial.com/fr/bash/topic/3656/pieges

Chapitre 44: Pipelines

Syntaxe

- [heure [-p]] [!] commande1 [| ou | & command2]…

Remarques

Un pipeline est une séquence de commandes simples séparées par l'un des opérateurs de contrôle `|` ou `|&` (source).

`|` connecte la sortie de `command1` à l'entrée de `command2` .

`|&` connecte la sortie standard et l'erreur standard de `command1` à l'entrée standard de `command2` .

Examples

Afficher tous les processus paginés

```
ps -e | less
```

`ps -e` montre tous les processus, sa sortie est connectée à l'entrée de plus via `|` , `less` pagine les résultats.

Utiliser | &

`|&` relie la sortie standard et erreur standard de la première commande à la seconde tout `|` connecte uniquement la sortie standard de la première commande à la deuxième commande.

Dans cet exemple, la page est téléchargée via `curl` . Avec l'option `-v` , `curl` écrit des informations sur `stderr` , la page téléchargée est écrite sur `stdout` . Le titre de la page peut être trouvé entre `<title>` et `</title>` .

```
curl -vs 'http://www.google.com/' |& awk '/Host:/{print}
/<title>/{match($0,/<title>(.*)<\/title>/,a);print a[1]}'
```

Le résultat est:

```
> Host: www.google.com
Google
```

Mais avec `|` beaucoup plus d'informations seront imprimées, c'est-à-dire celles qui sont envoyées à `stderr` car seule la `stdout` est acheminée vers la commande suivante. Dans cet exemple, toutes les lignes sauf la dernière ligne (Google) ont été envoyées à `stderr` par `curl` :

```
* Hostname was NOT found in DNS cache
*   Trying 172.217.20.228...
* Connected to www.google.com (172.217.20.228) port 80 (#0)
> GET / HTTP/1.1
> User-Agent: curl/7.35.0
> Host: www.google.com
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Date: Sun, 24 Jul 2016 19:04:59 GMT
< Expires: -1
< Cache-Control: private, max-age=0
< Content-Type: text/html; charset=ISO-8859-1
< P3P: CP="This is not a P3P policy! See
https://www.google.com/support/accounts/answer/151657?hl=en for more info."
< Server: gws
< X-XSS-Protection: 1; mode=block
< X-Frame-Options: SAMEORIGIN
< Set-Cookie: NID=82=jX0yZLPPUE7u13kKNevUCDg8yG9Ze_C03o0IM-
EopOSKL0mMITEagIE816G55L2wrTlQwgXkhq4ApFvvYEoaWF-
oEoq2T0sBTuQVdsIFULj9b2O8X35O0sAgUnc3a3JnTRBqelMcuS9QkQA; expires=Mon, 23-Jan-2017 19:04:59
GMT; path=/; domain=.google.com; HttpOnly
< Accept-Ranges: none
< Vary: Accept-Encoding
< X-Cache: MISS from jetsib_appliance
< X-Loop-Control: 5.202.190.157 81E4F9836653D5812995BA53992F8065
< Connection: close
<
{ [data not shown]
* Closing connection 0
Google
```

Modifier la sortie continue d'une commande

```
~$ ping -c 1 google.com # unmodified output
PING google.com (16.58.209.174) 56(84) bytes of data.
64 bytes from wk-in-f100.1e100.net (16.58.209.174): icmp_seq=1 ttl=53 time=47.4 ms
~$ ping google.com | grep -o '^[0-9]\+[^()]\+' # modified output
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
...
```

Le tube (`|`) connecte la `stdout` de `ping` à `stdin` de `grep` , qui le traite immédiatement. Certaines autres commandes, telles que `sed` par défaut en mémoire tampon leur `stdin` , ce qui signifie qu'il doit recevoir suffisamment de données avant d'imprimer quoi que ce soit, entraînant potentiellement des retards dans le traitement ultérieur.

Lire Pipelines en ligne: https://riptutorial.com/fr/bash/topic/5485/pipelines

Chapitre 45: Quand utiliser eval

Introduction

Avant tout: sachez ce que vous faites! Deuxièmement, alors que vous devriez éviter d'utiliser `eval` , si son utilisation crée un code plus propre, continuez.

Examples

En utilisant Eval

Par exemple, considérez ce qui suit qui définit le contenu de `$@` au contenu d'une variable donnée:

```
a=(1 2 3)
eval set -- "${a[@]}"
```

Ce code est souvent accompagné de `getopt` ou de `getopts` pour définir `$@` à la sortie des analyseurs d'option mentionnés ci-dessus. Cependant, vous pouvez également l'utiliser pour créer une fonction `pop` simple qui peut fonctionner directement sur des variables sans avoir à stocker le résultat. la variable d'origine:

```
isnum()
{
    # is argument an integer?
    local re='^[0-9]+$'
    if [[ -n $1 ]]; then
        [[ $1 =~ $re ]] && return 0
        return 1
    else
        return 2
    fi
}

isvar()
{
    if isnum "$1"; then
        return 1
    fi
    local arr="$(eval eval -- echo -n "\$$1")"
    if [[ -n ${arr[@]} ]]; then
        return 0
    fi
    return 1
}

pop()
{
    if [[ -z $@ ]]; then
        return 1
    fi

    local var=
```

```
    local isvar=0
    local arr=()

    if isvar "$1"; then # let's check to see if this is a variable or just a bare array
        var="$1"
        isvar=1
        arr=($(eval eval -- echo -n "\${$1[@]}")) # if it is a var, get its contents
    else
        arr=($@)
    fi

    # we need to reverse the contents of $@ so that we can shift
    # the last element into nothingness
    arr=($(awk <<<"${arr[@]}" '{ for (i=NF; i>1; --i) printf("%s ",$i); print $1; }'

    # set $@ to ${arr[@]} so that we can run shift against it.
    eval set -- "${arr[@]}"

    shift # remove the last element

    # put the array back to its original order
    arr=($(awk <<<"$@" '{ for (i=NF; i>1; --i) printf("%s ",$i); print $1; }'

    # echo the contents for the benefit of users and for bare arrays
    echo "${arr[@]}"

    if ((isvar)); then
        # set the contents of the original var to the new modified array
        eval -- "$var=(${arr[@]})"
    fi
}
```

Utiliser Eval avec Getopt

Même si eval n'est peut-être pas nécessaire pour une fonction de type `pop` , elle est toutefois requise lorsque vous utilisez `getopt` :

Considérons la fonction suivante qui accepte l'option `-h` :

```
f()
{
    local __me__="${FUNCNAME[0]}"
    local argv="$(getopt -o 'h' -n $__me__ -- "$@")"

    eval set -- "$argv"

    while :; do
        case "$1" in
            -h)
              echo "LOLOLOLOL"
              return 0
              ;;
            --)
              shift
              break
              ;;
    done
```

```
    echo "$@"
}
```

Sans `eval set -- "$argv"` génère `-h --` au lieu de celui désiré `(-h --)` et entre ensuite dans une boucle infinie car `-h --` ne correspond pas `--` ou `-h` .

Lire Quand utiliser eval en ligne: https://riptutorial.com/fr/bash/topic/10113/quand-utiliser-eval

Chapitre 46: Raccourcis clavier

Remarques

`bind -P` affiche tous les raccourcis configurés.

Examples

Rappel de raccourcis

Raccourci	La description
`Ctrl` + `r`	rechercher l'histoire en arrière
`Ctrl` + `p`	commande précédente dans l'histoire
`Ctrl` + `n`	prochaine commande dans l'histoire
`Ctrl` + `g`	quitter le mode de recherche d'historique
`Alt` + `.`	utiliser le dernier mot de la commande précédente
	Répéter pour obtenir le dernier mot de la précédente + 1 commande
`Alt` + `n` `Alt` + `.`	utiliser le nième mot de la commande précédente
`!!` + `Retour`	exécuter à nouveau la dernière commande (utile lorsque vous avez oublié sudo: `sudo !!`)

Modification des raccourcis

Raccourci	La description
`Ctrl` + `a`	se déplacer au début de la ligne
`Ctrl` + `e`	passer à la fin de la ligne
`Ctrl` + `k`	Tuez le texte de la position actuelle du curseur à la fin de la ligne.
`Ctrl` + `u`	Tuer le texte de la position actuelle du curseur au début de la ligne
`Ctrl` + `w`	Tuez le mot derrière la position actuelle du curseur
`Alt` + `b`	reculer d'un mot
`Alt` + `f`	avancer d'un mot

Raccourci	La description
`Ctrl` + `Alt` + `e`	shell développer ligne
`Ctrl` + `y`	Ramenez le dernier texte tué dans le tampon au niveau du curseur.
`Alt` + `y`	Tournez à travers le texte tué. Vous ne pouvez le faire que si la commande précédente est `Ctrl` + `y` ou `Alt` + `y` .

Tuer du texte supprime le texte, mais l'enregistre pour que l'utilisateur puisse le réinsérer en tirant. Semblable à couper et coller, sauf que le texte est placé sur un anneau mortel, ce qui permet de stocker plus d'un ensemble de texte pour être redirigé vers la ligne de commande.

Vous pouvez en savoir plus dans le manuel emacs .

Contrôle de l'emploi

Raccourci	La description
`Ctrl` + `c`	Arrête le travail en cours
`Ctrl` + `z`	Suspendre le travail en cours (envoyer un signal SIGTSTP)

Macros

Raccourci	La description
`Ctrl` + `x` , `(`	commencer à enregistrer une macro
`Ctrl` + `x` , `)`	arrêter d'enregistrer une macro
`Ctrl` + `x` , `e`	exécuter la dernière macro enregistrée

Custome Key Bindings

Avec la commande de `bind` il est possible de définir des liaisons de clé personnalisées.

L'exemple suivant lie un `Alt` + `w` à `>/dev/null 2>&1` :

```
bind '"\ew":"\" >/dev/null 2>&1\""
```

Si vous voulez exécuter la ligne immédiatement, ajoutez-lui `\Cm` (`Enter`):

```
bind '"\ew":"\" >/dev/null 2>&1\C-m\""
```

Lire Raccourcis clavier en ligne: https://riptutorial.com/fr/bash/topic/3949/raccourcis-clavier

Chapitre 47: Redirection

Syntaxe

- commande </ path / to / file # Redirection de l'entrée standard dans le fichier
- command> / path / to / file # Redirige la sortie standard vers flie
- commande descripteur_fichier> / chemin / vers / fichier # Redirige la sortie de descripteur_fichier vers le fichier
- commande> & file_descriptor # Redirige la sortie vers file_descriptor
- commande file_descriptor> & another_file_descriptor # Redirige file_descriptor vers another_file_descriptor
- commande <& file_descriptor # Redirige file_descriptor vers l'entrée standard
- commande &> / path / to / file # Redirige la sortie standard et l'erreur standard vers le fichier

Paramètres

Paramètre	Détails
descripteur de fichier interne	Un nombre entier.
direction	Un de `>` , `<` ou `<>`
descripteur de fichier externe ou chemin d'accès	`&` suivi d'un entier pour le descripteur de fichier ou un chemin.

Remarques

Les programmes de la console UNIX ont un fichier d'entrée et deux fichiers de sortie (les flux d'entrée et de sortie, ainsi que les périphériques, sont traités comme des fichiers par le système d'exploitation). pour venir - ou aller à - un fichier ou un autre programme.

`STDIN` est une entrée standard et est la manière dont le programme reçoit une saisie interactive. `STDIN` est généralement `STDIN` descripteur de fichier 0.

`STDOUT` est une sortie standard. Tout ce qui est émis sur `STDOUT` est considéré comme le "résultat" du programme. `STDOUT` est généralement associé au descripteur de fichier 1.

`STDERR` est l'endroit où les messages d'erreur sont affichés. En règle générale, lors de l'exécution d'un programme à partir de la console, `STDERR` est `STDERR` à l'écran et ne peut pas être distingué de `STDOUT` . `STDERR` est généralement associé au descripteur de fichier 2.

L'ordre de redirection est important

```
command > file 2>&1
```

Redirige les deux (`STDOUT` et `STDERR`) vers le fichier.

```
command 2>&1 > file
```

Seulement redirections `STDOUT` , parce que le descripteur de fichier 2 est redirigé vers le fichier pointé par le descripteur de fichier 1 (ce qui est le fichier `file` encore lorsque l'instruction est évaluée).

Chaque commande dans un pipeline a ses propres `STDERR` (et `STDOUT`) car chacun est un nouveau processus. Cela peut créer des résultats surprenants si vous vous attendez à ce qu'une redirection affecte l'ensemble du pipeline. Par exemple cette commande (enveloppée pour la lisibilité):

```
$ python -c 'import sys;print >> sys.stderr, "Python error!"' \
| cut -f1 2>> error.log
```

va imprimer "erreur Python!" à la console plutôt que le fichier journal. Au lieu de cela, attachez l'erreur à la commande que vous souhaitez capturer:

```
$ python -c 'import sys;print >> sys.stderr, "Python error!"' 2>> error.log \
| cut -f1
```

Examples

Redirection de la sortie standard

`>` redirige la sortie standard (aka `STDOUT`) de la commande en cours dans un fichier ou un autre descripteur.

Ces exemples écrivent la sortie de la commande `ls` dans le fichier `file.txt`

```
ls >file.txt
> file.txt ls
```

Le fichier cible est créé s'il n'existe pas, sinon ce fichier est tronqué.

Le descripteur de redirection par défaut est la sortie standard ou `1` si aucun n'est spécifié. Cette commande est équivalente aux exemples précédents avec la sortie standard explicitement indiquée:

```
ls 1>file.txt
```

Remarque: la redirection est initialisée par le shell exécuté et non par la commande exécutée, elle est donc effectuée avant l'exécution de la commande.

Redirection de STDIN

`<` lit à partir de son argument correct et écrit dans son argument de gauche.

Pour écrire un fichier dans `STDIN` il faut *lire* `/tmp/a_file` et *écrire* dans `STDIN` c'est-à-dire `0</tmp/a_file`

Remarque: le descripteur de fichier interne est `STDIN` par défaut sur `0` (`STDIN`) pour `<`

```
$ echo "b" > /tmp/list.txt
$ echo "a" >> /tmp/list.txt
$ echo "c" >> /tmp/list.txt
$ sort < /tmp/list.txt
a
b
c
```

Redirection à la fois STDOUT et STDERR

Les descripteurs de fichiers tels que `0` et `1` sont des pointeurs. Nous changeons ce que les descripteurs de fichiers indiquent avec la redirection. `>/dev/null` signifie que `1` pointe vers `/dev/null` .

D'abord, nous `STDOUT` `1` (`STDOUT`) sur `/dev/null` puis le point `2` (`STDERR`) sur tout ce que `1` pointe vers.

```
# STDERR is redirect to STDOUT: redirected to /dev/null,
# effectually redirecting both STDERR and STDOUT to /dev/null
echo 'hello' > /dev/null 2>&1
```

4.0

Cela *peut* être réduit à ce qui suit:

```
echo 'hello' &> /dev/null
```

Toutefois, cette forme peut être indésirable en production si la compatibilité du shell est un problème car elle entre en conflit avec POSIX, introduit une ambiguïté d'analyse et que les shells sans cette fonctionnalité l'interprètent de manière erronée:

```
# Actual code
echo 'hello' &> /dev/null
echo 'hello' &> /dev/null 'goodbye'

# Desired behavior
echo 'hello' > /dev/null 2>&1
echo 'hello' 'goodbye' > /dev/null 2>&1

# Actual behavior
echo 'hello' &
echo 'hello' & goodbye > /dev/null
```

REMARQUE: `&>` est connu pour fonctionner comme souhaité dans Bash et Zsh.

Redirection de STDERR

`2` est `STDERR` .

```
$ echo_to_stderr 2>/dev/null # echos nothing
```

Définitions:

`echo_to_stderr` est une commande qui écrit `"stderr"` dans `STDERR`

```
echo_to_stderr () {
    echo stderr >&2
}

$ echo_to_stderr
stderr
```

Ajouter vs tronquer

Tronquer `>`

1. Créez le fichier spécifié s'il n'existe pas.
2. Tronquer (supprimer le contenu du fichier)
3. Ecrire dans un fichier

```
$ echo "first line" > /tmp/lines
$ echo "second line" > /tmp/lines

$ cat /tmp/lines
second line
```

Ajouter `>>`

1. Créez le fichier spécifié s'il n'existe pas.
2. Ajouter un fichier (écriture à la fin du fichier).

```
# Overwrite existing file
$ echo "first line" > /tmp/lines

# Append a second line
$ echo "second line" >> /tmp/lines

$ cat /tmp/lines
first line
second line
```

STDIN, STDOUT et STDERR expliqués

Les commandes ont une entrée (STDIN) et deux types de sorties, la sortie standard (STDOUT) et l’erreur standard (STDERR).

Par exemple:

STDIN

```
root@server~# read
Type some text here
```

L'entrée standard est utilisée pour fournir une entrée à un programme. (Ici, nous utilisons la `read` intégrée pour lire une ligne de STDIN.)

STDOUT

```
root@server~# ls file
file
```

La sortie standard est généralement utilisée pour une sortie "normale" à partir d'une commande. Par exemple, `ls` répertorie les fichiers, de sorte que les fichiers sont envoyés à STDOUT.

STDERR

```
root@server~# ls anotherfile
ls: cannot access 'anotherfile': No such file or directory
```

L'erreur standard est (comme son nom l'indique) utilisée pour les messages d'erreur. Ce message n'étant pas une liste de fichiers, il est envoyé à STDERR.

STDIN, STDOUT et STDERR sont les trois *flux standard.* Ils sont identifiés à la coquille par un nombre plutôt que par un nom:

0 = standard dans
1 = Sortie standard
2 = erreur standard

Par défaut, STDIN est attaché au clavier et STDOUT et STDERR apparaissent dans le terminal. Cependant, nous pouvons rediriger STDOUT ou STDERR à tout ce dont nous avons besoin. Par exemple, supposons que vous ayez uniquement besoin de la sortie standard et que tous les messages d'erreur imprimés sur une erreur standard soient supprimés. C'est à ce moment que nous utilisons les descripteurs `1` et `2` .

Redirection de STDERR vers / dev / null
Prenant l'exemple précédent,

```
root@server~# ls anotherfile 2>/dev/null
root@server~#
```

Dans ce cas, s'il y a un STDERR, il sera redirigé vers / dev / null (un fichier spécial qui ignore tout ce qui y est placé), vous n'aurez donc aucune erreur sur le shell.

Redirection de plusieurs commandes vers le même fichier

```
{
  echo "contents of home directory"
  ls ~
} > output.txt
```

Utiliser des canaux nommés

Parfois, vous voudrez peut-être sortir quelque chose par un programme et l'entrer dans un autre programme, mais vous ne pouvez pas utiliser un canal standard.

```
ls -l | grep ".log"
```

Vous pouvez simplement écrire dans un fichier temporaire:

```
touch tempFile.txt
ls -l > tempFile.txt
grep ".log" < tempFile.txt
```

Cela fonctionne très bien pour la plupart des applications, cependant, personne ne saura ce `tempFile` fait `tempFile` et quelqu'un pourrait le supprimer s'il contient la sortie de `ls -l` dans ce répertoire. C'est là qu'entre en jeu un tube nommé:

```
mkfifo myPipe
ls -l > myPipe
grep ".log" < myPipe
```

`myPipe` est techniquement un fichier (tout est sous Linux), faisons donc `ls -l` dans un répertoire vide que nous venons de créer dans un tube:

```
mkdir pipeFolder
cd pipeFolder
mkfifo myPipe
ls -l
```

La sortie est la suivante:

```
prw-r--r-- 1 root root 0 Jul 25 11:20 myPipe
```

Notez le premier caractère dans les autorisations, il est répertorié comme un tube, pas un fichier.

Maintenant, faisons quelque chose de cool.

Ouvrez un terminal et notez le répertoire (ou créez-en un pour faciliter le nettoyage) et créez un canal.

```
mkfifo myPipe
```

Maintenant, mettons quelque chose dans le tuyau.

```
echo "Hello from the other side" > myPipe
```

Vous remarquerez que cela se bloque, l'autre côté du tuyau est toujours fermé. Ouvrons l'autre côté du tuyau et passons à travers.

Ouvrez un autre terminal et accédez au répertoire dans lequel se trouve le tube (ou, si vous le connaissez, ajoutez-le au tube):

```
cat < myPipe
```

Vous remarquerez qu'après la sortie de `hello from the other side` , le programme du premier terminal se termine, de même que celui du deuxième terminal.

Maintenant, lancez les commandes en sens inverse. Commencez avec le `cat < myPipe` , puis `cat < myPipe` écho à quelque chose. Cela fonctionne toujours, car un programme attendra que quelque chose soit mis dans le tube avant de se terminer, car il sait qu'il doit obtenir quelque chose.

Les canaux nommés peuvent être utiles pour déplacer des informations entre terminaux ou entre programmes.

Les pipes sont petites. Une fois rempli, le graveur se bloque jusqu'à ce qu'un lecteur lise le contenu. Vous devez donc exécuter le lecteur et le graveur sur différents terminaux ou exécuter l'un ou l'autre en arrière-plan:

```
ls -l /tmp > myPipe &
cat < myPipe
```

Plus d'exemples utilisant des canaux nommés:

- Exemple 1 - Toutes les commandes sur le même terminal / même shell

  ```
  $ { ls -l && cat file3; } >mypipe &
  $ cat <mypipe
  # Output: Prints ls -l data and then prints file3 contents on screen
  ```

- Exemple 2 - Toutes les commandes sur le même terminal / même shell

  ```
  $ ls -l >mypipe &
  $ cat file3 >mypipe &
  $ cat <mypipe
  #Output: This prints on screen the contents of mypipe.
  ```

  `file3` vous que le premier contenu de `file3` est affiché, puis les données `ls -l` sont affichées (configuration LIFO).

- Exemple 3 - Toutes les commandes sur le même terminal / même shell

  ```
  $ { pipedata=$(<mypipe) && echo "$pipedata"; } &
  $ ls >mypipe
  ```

```
# Output: Prints the output of ls directly on screen
```

`$pipedata` vous que la variable `$pipedata` n'est pas utilisable dans le terminal principal / shell principal car l'utilisation de `&` invoque un sous-shell et que `$pipedata` n'est disponible que dans ce sous-shell.

- Exemple 4 - Toutes les commandes sur le même terminal / même shell

```
$ export pipedata
$ pipedata=$(<mypipe) &
$ ls –l *.sh >mypipe
$ echo "$pipedata"
#Output : Prints correctly the contents of mypipe
```

Cela imprime correctement la valeur de la variable `$pipedata` dans le shell principal en raison de la déclaration d'exportation de la variable. Le terminal principal / shell principal n'est pas bloqué en raison de l'invocation d'un shell d'arrière-plan (`&`).

Imprimer les messages d'erreur sur stderr

Les messages d'erreur sont généralement inclus dans un script à des fins de débogage ou pour fournir une expérience utilisateur enrichie. Il suffit d'écrire un message d'erreur comme ceci:

```
cmd || echo 'cmd failed'
```

peut fonctionner pour des cas simples mais ce n'est pas la manière habituelle. Dans cet exemple, le message d'erreur pollue la sortie réelle du script en mélangeant les erreurs et la sortie réussie dans `stdout` .

En bref, le message d'erreur devrait aller à `stderr` pas `stdout` . C'est assez simple:

```
cmd || echo 'cmd failed' >/dev/stderr
```

Un autre exemple:

```
if cmd; then
    echo 'success'
else
    echo 'cmd failed' >/dev/stderr
fi
```

Dans l'exemple ci-dessus, le message de réussite sera imprimé sur `stdout` tandis que le message d'erreur sera imprimé sur `stderr` .

Une meilleure façon d’imprimer un message d’erreur est de définir une fonction:

```
err(){
    echo "E: $*" >>/dev/stderr
}
```

Maintenant, quand vous devez imprimer une erreur:

```
err "My error message"
```

Redirection vers les adresses réseau

2,04

Bash considère certains chemins comme spéciaux et peut communiquer avec le réseau en écrivant dans `/dev/{udp|tcp}/host/port` . Bash ne peut pas configurer un serveur d'écoute, mais peut initier une connexion, et pour TCP, il peut au moins lire les résultats.

Par exemple, pour envoyer une simple requête Web, vous pouvez:

```
exec 3</dev/tcp/www.google.com/80
printf 'GET / HTTP/1.0\r\n\r\n' >&3
cat <&3
```

et les résultats de la page Web par défaut de `www.google.com` seront imprimés sur `stdout` .

De même

```
printf 'HI\n' >/dev/udp/192.168.1.1/6666
```

enverrait un message UDP contenant `HI\n` à un auditeur sur `192.168.1.1:6666`

Lire Redirection en ligne: https://riptutorial.com/fr/bash/topic/399/redirection

Chapitre 48: Répertoires de navigation

Examples

Passer au dernier répertoire

Pour le shell actuel, cela vous amène au répertoire précédent dans lequel vous vous trouviez, peu importe où il se trouvait.

```
cd -
```

Faire cela plusieurs fois efficacement "bascule" vous être dans le répertoire en cours ou le précédent.

Passer au répertoire de base

Le répertoire par défaut est le répertoire personnel (`$HOME` , généralement `/home/username`), donc `cd` sans aucun répertoire vous y emmène

```
cd
```

Ou vous pourriez être plus explicite:

```
cd $HOME
```

Un raccourci pour le répertoire de base est `~` , ce qui permet également de l'utiliser.

```
cd ~
```

Répertoires absolus vs relatifs

Pour passer à un répertoire absolument spécifié, utilisez le nom entier, en commençant par une barre oblique inverse `\` , donc:

```
cd /home/username/project/abc
```

Si vous souhaitez passer à un répertoire proche de votre mode actuel, vous pouvez spécifier un emplacement relatif. Par exemple, si vous êtes déjà dans `/home/username/project` , vous pouvez entrer le sous-répertoire `abc` :

```
cd abc
```

Si vous souhaitez accéder au répertoire situé au-dessus du répertoire en cours, vous pouvez utiliser l'alias `..` Par exemple, si vous étiez dans `/home/username/project/abc` et que vous souhaitiez

accéder à `/home/username/project` , vous devez procéder comme suit:

```
cd ..
```

Cela peut aussi s'appeler "monter" un répertoire.

Passer au répertoire du script

En général, il existe deux types de **scripts** Bash:

1. Outils système fonctionnant à partir du répertoire de travail en cours
2. Outils de projet qui modifient les fichiers par rapport à leur place dans le système de fichiers

Pour le deuxième type de scripts, il est utile de passer au répertoire dans lequel le script est stocké. Cela peut être fait avec la commande suivante:

```
cd "$(dirname "$(readlink -f "$0")")"
```

Cette commande exécute 3 commandes:

1. `readlink -f "$0"` détermine le chemin d'accès au script en cours (`$0`)
2. `dirname` convertit le chemin d'accès au script vers le chemin d'accès à son répertoire
3. `cd` change le répertoire de travail en cours dans le répertoire qu'il reçoit de `dirname`

Lire Répertoires de navigation en ligne: https://riptutorial.com/fr/bash/topic/6784/repertoires-de-navigation

Chapitre 49: Script avec des paramètres

Remarques

- `shift` décale les paramètres de position vers la gauche pour que `$2` devienne `$1` , `$3` devienne `$2` et ainsi de suite.
- `"$@"` est un tableau de tous les paramètres de position transmis au script / à la fonction.
- `"$*"` est une chaîne composée de tous les paramètres de position transmis au script / à la fonction.

Examples

Analyse de paramètres multiples

Pour analyser beaucoup de paramètres, la façon de le faire preferred utilise une boucle *while,* une déclaration *de cas,* et le *décalage.*

`shift` est utilisé pour faire apparaître le premier paramètre de la série, ce qui fait que ce qui était auparavant *2 $ est* maintenant *1 $* . Ceci est utile pour traiter les arguments un par un.

```
#!/bin/bash

# Load the user defined parameters
while [[ $# > 0 ]]
do
        case "$1" in

                -a|--valueA)
                        valA="$2"
                        shift
                        ;;

                -b|--valueB)
                        valB="$2"
                        shift
                        ;;

                --help|*)
                        echo "Usage:"
                        echo "    --valueA \"value\""
                        echo "    --valueB \"value\""
                        echo "    --help"
                        exit 1
                        ;;
        esac
        shift
done

echo "A: $valA"
echo "B: $valB"
```

Entrées et sorties

```
$ ./multipleParams.sh --help
Usage:
    --valueA "value"
    --valueB "value"
    --help

$ ./multipleParams.sh
A:
B:

$ ./multipleParams.sh --valueB 2
A:
B: 2

$ ./multipleParams.sh --valueB 2 --valueA "hello world"
A: hello world
B: 2
```

Accès aux paramètres

Lors de l'exécution d'un script Bash, les paramètres transmis au script sont nommés en fonction de leur position: `$1` est le nom du premier paramètre, `$2` est le nom du deuxième paramètre, etc.

Un paramètre manquant évalue simplement une chaîne vide. La vérification de l'existence d'un paramètre peut être effectuée comme suit:

```
if [ -z "$1" ]; then
    echo "No argument supplied"
fi
```

Obtenir tous les paramètres

`$@` et `$*` sont des moyens d'interagir avec tous les paramètres du script. En référence à la page de manuel Bash , nous voyons que:

- `$*` : Étend les paramètres de position à partir de un. Lorsque l'expansion se produit entre guillemets, elle se développe en un seul mot avec la valeur de chaque paramètre séparée par le premier caractère de la variable spéciale IFS.
- `$@` : Étend les paramètres de position à partir de un. Lorsque l'expansion se produit entre guillemets, chaque paramètre se développe en un mot distinct.

Obtenir le nombre de paramètres

`$#` obtient le nombre de paramètres passés dans un script. Un cas d'utilisation typique serait de vérifier si le nombre d'arguments approprié est transmis:

```
if [ $# -eq 0 ]; then
    echo "No arguments supplied"
fi
```

Exemple 1

Parcourez tous les arguments et vérifiez s'ils sont des fichiers:

```
for item in "$@"
do
    if [[ -f $item ]]; then
        echo "$item is a file"
    fi
done
```

Exemple 2

Parcourez tous les arguments et vérifiez s'ils sont des fichiers:

```
for (( i = 1; i <= $#; ++ i ))
do
    item=${@:$i:1}

    if [[ -f $item ]]; then
        echo "$item is a file"
    fi
done
```

Analyse d'argument utilisant une boucle for

Un exemple simple qui fournit les options:

Opter	Alt. Opter	Détails
`-h`	`--help`	Montrer de l'aide
`-v`	`--version`	Afficher les informations de version
`-dr path`	`--doc-root path`	Une option qui prend un paramètre secondaire (un chemin)
`-i`	`--install`	Une option booléenne (true / false)
`-*`	-	Option invalide

```
#!/bin/bash
dr=''
install=false

skip=false
for op in "$@";do
    if $skip;then skip=false;continue;fi
    case "$op" in
        -v|--version)
            echo "$ver_info"
            shift
```

```
            exit 0
            ;;
        -h|--help)
            echo "$help"
            shift
            exit 0
            ;;
        -dr|--doc-root)
            shift
            if [[ "$1" != "" ]]; then
                dr="${1/%\//}"
                shift
                skip=true
            else
                echo "E: Arg missing for -dr option"
                exit 1
            fi
            ;;
        -i|--install)
            install=true
            shift
            ;;
        -*)
            echo "E: Invalid option: $1"
            shift
            exit 1
            ;;
    esac
done
```

Script wrapper

Le script Wrapper est un script qui encapsule un autre script ou une autre commande pour fournir des fonctionnalités supplémentaires ou simplement pour rendre quelque chose de moins fastidieux.

Par exemple, le `egrep` actuel dans le nouveau système GNU / Linux est remplacé par un script wrapper nommé `egrep` . Voici à quoi ça ressemble:

```
#!/bin/sh
exec grep -E "$@"
```

Ainsi, lorsque vous exécutez `egrep` dans de tels systèmes, vous exécutez en fait `grep -E` avec tous les arguments transférés.

Dans le cas général, si vous voulez exécuter un exemple de script / commande `exmp` avec un autre script `mexmp` alors le script wrapper `mexmp` ressemblera à:

```
#!/bin/sh
exmp "$@" # Add other options before "$@"
# or
#full/path/to/exmp "$@"
```

Diviser la chaîne en un tableau dans Bash

Disons que nous avons un paramètre String et que nous voulons le diviser par une virgule

```
my_param="foo,bar,bash"
```

Pour diviser cette chaîne par une virgule, nous pouvons utiliser;

```
IFS=',' read -r -a array <<< "$my_param"
```

Ici, IFS est une variable spéciale appelée séparateur de champs internes qui définit le ou les caractères utilisés pour séparer un modèle en jetons pour certaines opérations.

Pour accéder à un élément individuel:

```
echo "${array[0]}"
```

Pour parcourir les éléments:

```
for element in "${array[@]}"
do
    echo "$element"
done
```

Pour obtenir à la fois l'index et la valeur:

```
for index in "${!array[@]}"
do
    echo "$index ${array[index]}"
done
```

Lire Script avec des paramètres en ligne: https://riptutorial.com/fr/bash/topic/746/script-avec-des-parametres

Chapitre 50: Script shebang

Syntaxe

- Utilisez `/bin/bash` comme interpréteur bash:

  #! / bin / bash

- Recherchez l'interpréteur bash dans la variable d'environnement `PATH` avec l'exécutable `env` :

  #! / usr / bin / env bash

Remarques

Une erreur commune est d'essayer d'exécuter de Windows fin en ligne au format `\r\n` fichiers de script sur les systèmes UNIX / Linux, dans ce cas , l'interpréteur de script utilisé dans le tralala est:

```
/bin/bash\r
```

Et est inconsciemment introuvable, mais peut être difficile à comprendre.

Examples

Shebang direct

Pour exécuter un fichier script avec l'interpréteur `bash` , la **première ligne** d'un fichier script doit indiquer le chemin absolu vers l'exécutable `bash` à utiliser:

```
#!/bin/bash
```

Le chemin `bash` dans le shebang est résolu et utilisé uniquement si un script est directement lancé comme ceci:

```
./script.sh
```

Le script doit avoir une autorisation d'exécution.

Le shebang est ignoré lorsqu'un interpréteur `bash` est explicitement indiqué pour exécuter un script:

```
bash script.sh
```

Env shebang

Pour exécuter un fichier script avec l'exécutable `bash` trouvé dans la variable d'environnement `PATH`

à l'aide de l'exécutable `env` , la **première ligne** d'un fichier script doit indiquer le chemin absolu vers l'exécutable `env` avec l'argument `bash` :

```
#!/usr/bin/env bash
```

Le chemin `env` dans le shebang est résolu et utilisé uniquement si un script est directement lancé comme ceci:

```
script.sh
```

Le script doit avoir une autorisation d'exécution.

Le shebang est ignoré lorsqu'un interpréteur `bash` est explicitement indiqué pour exécuter un script:

```
bash script.sh
```

Autres shebangs

Il existe deux types de programmes connus du noyau. Un programme binaire est identifié par son en-tête ELF (**E** xtenable **L** oadable **F** ormat), qui est généralement produit par un compilateur. Le second est des scripts de toutes sortes.

Si un fichier commence à la toute première ligne avec la séquence **#!** alors la chaîne suivante doit être un chemin d'accès d'un interprète. Si le noyau lit cette ligne, il appelle l'interpréteur nommé par ce chemin et donne tous les mots suivants dans cette ligne comme arguments à l'interpréteur. S'il n'y a pas de fichier nommé "quelque chose" ou "faux":

```
#!/bin/bash something wrong
echo "This line never gets printed"
```

bash essaie d'exécuter son argument "quelque chose de mal" qui n'existe pas. Le nom du fichier de script est également ajouté. Pour voir cela clairement utiliser un **écho** shebang:

```
#"/bin/echo something wrong
# and now call this script named "thisscript" like so:
# thisscript one two
# the output will be:
something wrong ./thisscript one two
```

Certains programmes comme **awk** utilisent cette technique pour exécuter des scripts plus longs résidant dans un fichier disque.

Lire Script shebang en ligne: https://riptutorial.com/fr/bash/topic/3658/script-shebang

Chapitre 51: Scripts CGI

Examples

Méthode de demande: GET

Il est assez facile d'appeler un script CGI via `GET` .
Tout d'abord, vous aurez besoin de l' `encoded url` du script.

Ensuite, vous ajoutez un point d'interrogation `?` suivi de variables.

- Chaque variable doit avoir deux sections séparées par **=** .
  La première section doit toujours être un nom unique pour chaque variable,
  alors que la seconde partie ne contient que des valeurs
- Les variables sont séparées par **&**
- La longueur totale de la chaîne ne doit pas dépasser **255** caractères
- Les noms et les valeurs doivent être encodés en HTML (remplacer: **</, /?:** @ **& = + $**)
  Allusion:
  Lorsque vous utilisez **des formulaires HTML,** la méthode de requête peut être générée par elle-même.
  Avec **Ajax,** vous pouvez encoder tout via **encodeURI** et **encodeURIComponent**

Exemple:

```
http://www.example.com/cgi-bin/script.sh?var1=Hello%20World!&var2=This%20is%20a%20Test.&
```

Le serveur doit communiquer via le **partage de ressources inter-origine** (CORS) uniquement, afin de sécuriser la demande. Dans cette présentation, nous utilisons **CORS** pour déterminer le type de `Data-Type` nous voulons utiliser.

Il y a beaucoup `Data-Types` nous pouvons choisir, les plus courants sont ...

- **text / html**
- **texte simple**
- **application / json**

Lors de l'envoi d'une demande, le serveur créera également de nombreuses variables d'environnement. Pour l'instant, les variables d'environnement les plus importantes sont `$REQUEST_METHOD` et `$QUERY_STRING` .

La **méthode de demande** doit être `GET` rien d' autre!
La **chaîne de requête** inclut toutes les `html-endoded data` .

Le script

```
#!/bin/bash
```

```
# CORS is the way to communicate, so lets response to the server first
echo "Content-type: text/html"    # set the data-type we want to use
echo ""    # we dont need more rules, the empty line initiate this.

# CORS are set in stone and any communication from now on will be like reading a html-
document.
# Therefor we need to create any stdout in html format!

# create html scructure and send it to stdout
echo "<!DOCTYPE html>"
echo "<html><head>"

# The content will be created depending on the Request Method
if [ "$REQUEST_METHOD" = "GET" ]; then

    # Note that the environment variables $REQUEST_METHOD and $QUERY_STRING can be processed
by the shell directly.
    # One must filter the input to avoid cross site scripting.

    Var1=$(echo "$QUERY_STRING" | sed -n 's/^.*var1=\([^&]*\).*$/\1/p')    # read value of
"var1"
    Var1_Dec=$(echo -e $(echo "$Var1" | sed 's/+/ /g;s/%\(..\)/\\x\1/g;'))    # html decode

    Var2=$(echo "$QUERY_STRING" | sed -n 's/^.*var2=\([^&]*\).*$/\1/p')
    Var2_Dec=$(echo -e $(echo "$Var2" | sed 's/+/ /g;s/%\(..\)/\\x\1/g;'))

    # create content for stdout
    echo "<title>Bash-CGI Example 1</title>"
    echo "</head><body>"
    echo "<h1>Bash-CGI Example 1</h1>"
    echo "<p>QUERY_STRING: ${QUERY_STRING}<br>var1=${Var1_Dec}<br>var2=${Var2_Dec}</p>"    #
print the values to stdout

else

    echo "<title>456 Wrong Request Method</title>"
    echo "</head><body>"
    echo "<h1>456</h1>"
    echo "<p>Requesting data went wrong.<br>The Request method has to be \"GET\" only!</p>"

fi

echo "<hr>"
echo "$SERVER_SIGNATURE"    # an other environment variable
echo "</body></html>"    # close html

exit 0
```

Le **document HTML** ressemblera à ceci ...

```
<html><head>
<title>Bash-CGI Example 1</title>
</head><body>
<h1>Bash-CGI Example 1</h1>
<p>QUERY_STRING: var1=Hello%20World!&amp;var2=This%20is%20a%20Test.&amp;<br>var1=Hello
World!<br>var2=This is a Test.</p>
<hr>
<address>Apache/2.4.10 (Debian) Server at example.com Port 80</address>
```

```
</body></html>
```

La **sortie** des variables ressemblera à ceci ...

```
var1=Hello%20World!&var2=This%20is%20a%20Test.&
Hello World!
This is a Test.
Apache/2.4.10 (Debian) Server at example.com Port 80
```

Effets secondaires négatifs ...

- Tout l'encodage et le décodage ne sont pas beaux, mais sont nécessaires
- La demande sera lisible par le public et laissera un plateau derrière
- La taille d'une demande est limitée
- A besoin d'une protection contre les scripts croisés (XSS)

Méthode de requête: POST / w JSON

L'utilisation de la méthode de requête `POST` en combinaison avec `SSL` rend le transfert de données plus sûr.

En outre...

- La plupart du codage et du décodage ne sont plus nécessaires
- L'URL sera visible par toute personne et doit être encodée en URL.
  Les données seront envoyées séparément et doivent donc être sécurisées via SSL
- La taille des données est presque illimitée
- Toujours besoin d'une protection contre les scripts croisés (XSS)

Pour garder cette vitrine simple, nous voulons recevoir **des données JSON**
et la communication devrait se faire par **-** dessus le **partage de ressources d'origine croisée** (CORS).

Le script suivant montrera également deux types de **contenu** différents.

```
#!/bin/bash

exec 2>/dev/null    # We dont want any error messages be printed to stdout
trap "response_with_html && exit 0" ERR    # response with an html message when an error
occurred and close the script

function response_with_html(){
    echo "Content-type: text/html"
    echo ""
    echo "<!DOCTYPE html>"
    echo "<html><head>"
    echo "<title>456</title>"
    echo "</head><body>"
    echo "<h1>456</h1>"
    echo "<p>Attempt to communicate with the server went wrong.</p>"
```

```
    echo "<hr>"
    echo "$SERVER_SIGNATURE"
    echo "</body></html>"
}

function response_with_json(){
    echo "Content-type: application/json"
    echo ""
    echo "{\"message\": \"Hello World!\"}"
}

if [ "$REQUEST_METHOD" = "POST" ]; then

    # The environment variabe $CONTENT_TYPE describes the data-type received
    case "$CONTENT_TYPE" in
    application/json)
        # The environment variabe $CONTENT_LENGTH describes the size of the data
        read -n "$CONTENT_LENGTH" QUERY_STRING_POST        # read datastream

        # The following lines will prevent XSS and check for valide JSON-Data.
        # But these Symbols need to be encoded somehow before sending to this script
        QUERY_STRING_POST=$(echo "$QUERY_STRING_POST" | sed "s/'//g" | sed
's/\$//g;s/`//g;s/\*//g;s/\\//g' )        # removes some symbols (like \ * ` $ ') to prevent
XSS with Bash and SQL.
        QUERY_STRING_POST=$(echo "$QUERY_STRING_POST" | sed -e :a -e 's/<[^>]*>//g;/</N;//ba')
# removes most html declarations to prevent XSS within documents
        JSON=$(echo "$QUERY_STRING_POST" | jq .)        # json encode - This is a pretty save
way to check for valide json code
    ;;
    *)
        response_with_html
        exit 0
    ;;
    esac

else
    response_with_html
    exit 0
fi

# Some Commands ...

response_with_json

exit 0
```

Vous obtiendrez `{"message":"Hello World!"}` Comme réponse lors de l'envoi **de données JSON** via `POST` à ce script. Tout le reste recevra le document HTML.

Important est également le varialbe `$JSON` . Cette variable est libre de XSS, mais peut toujours contenir des valeurs erronées et doit être vérifiée en premier. S'il vous plaît garder cela à l'esprit.

Ce code fonctionne de manière similaire sans JSON.
Vous pouvez obtenir toutes les données de cette façon.
Il vous suffit de modifier le type de `Content-Type` pour vos besoins.

Exemple:

```
if [ "$REQUEST_METHOD" = "POST" ]; then
    case "$CONTENT_TYPE" in
    application/x-www-form-urlencoded)
            read -n "$CONTENT_LENGTH" QUERY_STRING_POST
    text/plain)
            read -n "$CONTENT_LENGTH" QUERY_STRING_POST
    ;;
    esac
fi
```

Last but not least, n'oubliez pas de répondre à toutes les demandes, sinon les programmes tiers ne sauront pas s'ils ont réussi.

Lire Scripts CGI en ligne: https://riptutorial.com/fr/bash/topic/9603/scripts-cgi

Chapitre 52: Sélectionnez un mot clé

Introduction

Le mot-clé de sélection peut être utilisé pour obtenir un argument d'entrée dans un format de menu.

Examples

Le mot-clé de sélection peut être utilisé pour obtenir un argument d'entrée dans un format de menu

Supposons que vous souhaitiez que l' `user select` mots-clés dans un menu, nous pouvons créer un script similaire à

```
#!/usr/bin/env bash

select os in "linux" "windows" "mac"
do
    echo "${os}"
    break
done
```

Explication: Ici, le mot-clé `select` est utilisé pour parcourir une liste d'éléments qui seront présentés à l'invite de commande pour un utilisateur. Notez le mot-clé `break` pour sortir de la boucle une fois que l'utilisateur a fait son choix. Sinon, la boucle sera sans fin!

Résultats: Lors de l'exécution de ce script, un menu de ces éléments sera affiché et l'utilisateur sera invité à effectuer une sélection. Lors de la sélection, la valeur sera affichée et reviendra à l'invite de commande.

```
>bash select_menu.sh
1) linux
2) windows
3) mac
#? 3
mac
>
```

Lire Sélectionnez un mot clé en ligne: https://riptutorial.com/fr/bash/topic/10104/selectionnez-un-mot-cle

Chapitre 53: Séquence d'exécution du fichier

Introduction

`.bash_profile` , `.bash_login` , `.bashrc` et `.profile` font tous la même chose: configurez et définissez les fonctions, les variables et les tris.

La principale différence est que `.bashrc` est appelé à l'ouverture d'une fenêtre non connectée mais interactive, et que `.bash_profile` et les autres sont appelés pour un shell de connexion. Beaucoup de gens ont leur `.bash_profile` ou un appel similaire `.bashrc` toute façon.

Remarques

Les autres fichiers importants sont:

- `/etc/profile` , pour le code d'initialisation à l'échelle du système (non spécifique à l'utilisateur).
- `.bash_logout` , déclenché lors de la `.bash_logout` (pensez au nettoyage)
- `.inputrc` , similaire à `.bashrc` mais pour readline.

Examples

.profile vs .bash_profile (et .bash_login)

`.profile` est lu par la plupart des shells au démarrage, y compris bash. Cependant, `.bash_profile` est utilisé pour les configurations spécifiques à bash. Pour le code d'initialisation général, placez-le dans `.profile` . Si c'est spécifique à bash, utilisez `.bash_profile` .

`.profile` n'est pas conçu spécifiquement pour bash, mais `.bash_profile` est à la place. (`.profile` est pour Bourne et autres shells similaires, dont bash est basé) Bash retournera à `.profile` si `.bash_profile` n'est pas trouvé.

`.bash_login` est un repli pour `.bash_profile` , s'il n'est pas trouvé. Généralement, il `.bash_profile` mieux utiliser `.bash_profile` ou `.profile` place.

Lire Séquence d'exécution du fichier en ligne: https://riptutorial.com/fr/bash/topic/8626/sequence-d-execution-du-fichier

Chapitre 54: Sortie de script couleur (multiplate-forme)

Remarques

`tput` interroge la base de données terminfo pour obtenir des informations dépendantes du terminal.

De tput sur Wikipedia :

> En informatique, `tput` est une commande standard du système d'exploitation Unix qui utilise les capacités du terminal.
>
> Selon le système, `tput` utilise la base de données terminfo ou termcap, ainsi que le type de terminal dans l'environnement.

de l' invite Bash HOWTO: Chapitre 6. Séquences d'échappement ANSI: Couleurs et mouvement du curseur :

- **tput setab [1-7]**
  - Définir une couleur d'arrière-plan à l'aide de l'échappement ANSI
- **tput setb [1-7]**
  - Définir une couleur de fond
- **tput setaf [1-7]**
  - Définir une couleur de premier plan à l'aide de l'échappement ANSI
- **tput setf [1-7]**
  - Définir une couleur de premier plan
- **tput bold**
  - Définir le mode gras
- **tput sgr0**
  - Désactiver tous les attributs (ne fonctionne pas comme prévu)

Examples

color-output.sh

Dans la section d'ouverture d'un script bash, il est possible de définir certaines variables qui fonctionnent comme des aides pour colorer ou formater autrement la sortie du terminal pendant l'exécution du script.

Différentes plates-formes utilisent différentes séquences de caractères pour exprimer la couleur. Cependant, il existe un utilitaire appelé `tput` qui fonctionne sur tous les systèmes * nix et renvoie des chaînes de coloriage de terminaux spécifiques à la plate-forme via une API multi-plates-formes cohérente.

Par exemple, pour stocker la séquence de caractères qui transforme le texte du terminal en rouge ou en vert:

```
red=$(tput setaf 1)
green=$(tput setaf 2)
```

Ou, pour stocker la séquence de caractères qui réinitialise le texte à l'apparence par défaut:

```
reset=$(tput sgr0)
```

Ensuite, si le script BASH devait afficher des sorties de couleurs différentes, ceci peut être réalisé avec:

```
echo "${green}Success!${reset}"
echo "${red}Failure.${reset}"
```

Lire Sortie de script couleur (multiplate-forme) en ligne:
https://riptutorial.com/fr/bash/topic/6670/sortie-de-script-couleur--multiplate-forme-

Chapitre 55: Sourcing

Examples

Sourcing d'un fichier

La recherche d'un fichier est différente de l'exécution, dans la mesure où toutes les commandes sont évaluées dans le contexte de la session bash en cours. Cela signifie que toutes les variables, fonctions ou alias définis persisteront tout au long de votre session.

Créez le fichier que vous souhaitez `sourceme.sh`

```
#!/bin/bash

export A="hello_world"
alias sayHi="echo Hi"
sayHello() {
    echo Hello
}
```

A partir de votre session, source le fichier

```
$ source sourceme.sh
```

Dès lors, vous avez toutes les ressources du fichier source disponibles

```
$ echo $A
hello_world

$ sayHi
Hi

$ sayHello
Hello
```

Notez que la commande `.` est synonyme de `source` , de sorte que vous pouvez simplement utiliser

```
$ . sourceme.sh
```

Trouver un environnement virtuel

Lors du développement de plusieurs applications sur une machine, il devient utile de séparer les dépendances en environnements virtuels.

Avec l'utilisation de `virtualenv` , ces environnements proviennent de votre shell de sorte que lorsque vous exécutez une commande, elle provient de cet environnement virtuel.

Ceci est le plus souvent installé en utilisant `pip` .

```
pip install https://github.com/pypa/virtualenv/tarball/15.0.2
```

Créer un nouvel environnement

```
virtualenv --python=python3.5 my_env
```

Activer l'environnement

```
source my_env/bin/activate
```

Lire Sourcing en ligne: https://riptutorial.com/fr/bash/topic/564/sourcing

Chapitre 56: strace

Syntaxe

- strace -c [df] [-In] [-bexecve] [-eexpr] ... [-Ooverhead] [-Ssortby] -ppid ... / [-D] [-Evar [= val]] ... Commande [-uusername] [args]

Examples

Comment observer les appels système d'un programme

Pour un *fichier exécutable ou une commande* exec, l'exécution de cette liste répertoriera tous les appels système:

```
$ ptrace exec
```

Pour afficher des appels système spécifiques, utilisez l'option -e:

```
$ strace -e open exec
```

Pour enregistrer la sortie dans un fichier, utilisez l'option -o:

```
$ strace -o output exec
```

Pour trouver les appels système utilisés par un programme actif, utilisez l'option -p en spécifiant le pid [comment obtenir le pid] :

```
$ sudo strace -p 1115
```

Pour générer un rapport statistique de tous les appels système utilisés, utilisez l'option -c:

```
$ strace -c exec
```

Lire strace en ligne: https://riptutorial.com/fr/bash/topic/10855/strace

Chapitre 57: Structures de contrôle

Syntaxe

- ["$ 1" = "$ 2"] #Un "[" crochet est en fait une commande. Pour cela, il faut un espace avant et après.
- test "$ 1" = "$ 2" #Test un synonyme de la commande "["

Paramètres

Paramètre à [ou test	Détails
Opérateurs de fichiers	Détails
`-e "$file"`	Renvoie true si le fichier existe
`-d "$file"`	Renvoie true si le fichier existe et est un répertoire
`-f "$file"`	Renvoie true si le fichier existe et est un fichier normal
`-h "$file"`	Renvoie true si le fichier existe et constitue un lien symbolique
Comparateurs de chaînes	Détails
`-z "$str"`	Vrai si la longueur de la chaîne est zéro
`-n "$str`	Vrai si la longueur de la chaîne est différente de zéro
`"$str" = "$str2"`	True si string $ str est égal à string $ str2. Pas le meilleur pour les entiers. Cela peut fonctionner mais sera inconsistant
`"$str" != "$str2"`	Vrai si les chaînes ne sont pas égales
Comparateurs entiers	Détails
`"$int1" -eq "$int2"`	Vrai si les entiers sont égaux
`"$int1" -ne "$int2"`	Vrai si les entiers ne sont pas égaux
`"$int1" -gt "$int2"`	True si int1 est supérieur à int 2
`"$int1" -ge "$int2"`	True si int1 est supérieur ou égal à int2
`"$int1" -lt "$int2"`	True si int1 est inférieur à int 2

Paramètre à [ou test	Détails
`"$int1" -le "$int2"`	True si int1 est inférieur ou égal à int2

Remarques

De nombreux paramètres de comparaison sont disponibles dans bash. Tous ne sont pas encore listés ici.

Examples

Si déclaration

```
if [[ $1 -eq 1 ]]; then
    echo "1 was passed in the first parameter"
elif [[ $1 -gt 2 ]]; then
    echo "2 was not passed in the first parameter"
else
    echo "The first parameter was not 1 and is not more than 2."
fi
```

La fermeture `fi` est nécessaire, mais les clauses `elif` et / ou `else` peuvent être omises.

Les virgules avant `then` sont syntaxe standard pour combiner deux commandes sur une seule ligne; ils peuvent être omis que si `then` est déplacé à la ligne suivante.

Il est important de comprendre que les crochets `[[` ne font pas partie de la syntaxe, mais sont traités comme une commande; c'est le code de sortie de cette commande qui est en cours de test. Par conséquent, vous devez toujours inclure des espaces entre les crochets.

Cela signifie également que le résultat de toute commande peut être testé. Si le code de sortie de la commande est un zéro, l'instruction est considérée comme vraie.

```
if grep "foo" bar.txt; then
    echo "foo was found"
else
    echo "foo was not found"
fi
```

Les expressions mathématiques, lorsqu'elles sont placées à l'intérieur de doubles parenthèses, renvoient également 0 ou 1 de la même manière et peuvent également être testées:

```
if (( $1 + 5 > 91 )); then
    echo "$1 is greater than 86"
fi
```

Vous pouvez également rencontrer `if` les déclarations avec crochets simples. Ceux-ci sont définis dans la norme POSIX et sont garantis pour fonctionner dans tous les shells compatibles POSIX, y compris Bash. La syntaxe est très similaire à celle de Bash:

```
if [ "$1" -eq 1 ]; then
    echo "1 was passed in the first parameter"
elif [ "$1" -gt 2 ]; then
    echo "2 was not passed in the first parameter"
else
    echo "The first parameter was not 1 and is not more than 2."
fi
```

En boucle

```
#! /bin/bash

i=0

while [ $i -lt 5 ] #While i is less than 5
do
    echo "i is currently $i"
    i=$[$i+1] #Not the lack of spaces around the brackets. This makes it a not a test
expression
done #ends the loop
```

Veillez à ce qu'il y ait des espaces autour des parenthèses pendant le test (après la déclaration while). Ces espaces sont nécessaires.

Cette boucle de sortie:

```
i is currently 0
i is currently 1
i is currently 2
i is currently 3
i is currently 4
```

Pour boucle

```
#! /bin/bash

for i in 1 "test" 3; do #Each space separated statement is assigned to i
    echo $i
done
```

D'autres commandes peuvent générer des instructions à boucler. Voir "Utilisation de la boucle pour itérer les nombres".

Cela produit:

```
1
test
3
```

Utilisation de la boucle For pour répertorier les itérations sur les nombres

```
#! /bin/bash
```

```
for i in {1..10}; do # {1..10} expands to "1 2 3 4 5 6 7 8 9 10"
    echo $i
done
```

Cela génère les éléments suivants:

```
1
2
3
4
5
6
7
8
8
10
```

Pour une boucle avec une syntaxe de style C

Le format de base du style C `for` boucle est le suivant:

```
for (( variable assignment; condition; iteration process ))
```

Remarques:

- L'affectation de la variable à l'intérieur du style C `for` boucle peut contenir des espaces contrairement à l'attribution habituelle
- Les variables de style C `for` boucle ne sont pas précédées de `$` .

Exemple:

```
for (( i = 0; i < 10; i++ ))
do
    echo "The iteration number is $i"
done
```

Nous pouvons également traiter plusieurs variables dans le style C `for` boucle:

```
for (( i = 0, j = 0; i < 10; i++, j = i * i ))
do
    echo "The square of $i is equal to $j"
done
```

Jusqu'à la boucle

Jusqu'à ce que la boucle s'exécute jusqu'à ce que la condition soit vraie

```
i=5
until [[ i -eq 10 ]]; do #Checks if i=10
    echo "i=$i" #Print the value of i
```

```
    i=$((i+1)) #Increment i by 1
done
```

Sortie:

```
i=5
i=6
i=7
i=8
i=9
```

Quand `i` atteint 10 l'état jusqu'à ce que la boucle devient vraie et la fin de la boucle.

continuer et casser

Exemple pour continuer

```
for i in [series]
do
    command 1
    command 2
    if (condition) # Condition to jump over command 3
            continue # skip to the next value in "series"
    fi
    command 3
done
```

Exemple de pause

```
for i in [series]
do
    command 4
    if (condition) # Condition to break the loop
    then
            command 5 # Command if the loop needs to be broken
            break
    fi
    command 6 # Command to run if the "condition" is never true
done
```

En boucle sur un tableau

`for` boucle:

```
arr=(a b c d e f)
for i in "${arr[@]}";do
    echo "$i"
done
```

Ou

```
for ((i=0;i<${#arr[@]};i++));do
```

```
    echo "${arr[$i]}"
done
```

`while` boucle:

```
i=0
while [ $i -lt ${#arr[@]} ];do
    echo "${arr[$i]}"
    i=$(expr $i + 1)
done
```

Ou

```
i=0
while (( $i < ${#arr[@]} ));do
    echo "${arr[$i]}"
    ((i++))
done
```

Pause de boucle

Casser plusieurs boucles:

```
arr=(a b c d e f)
for i in "${arr[@]}";do
    echo "$i"
    for j in "${arr[@]}";do
        echo "$j"
        break 2
    done
done
```

Sortie:

```
a
a
```

Casser une seule boucle:

```
arr=(a b c d e f)
for i in "${arr[@]}";do
    echo "$i"
    for j in "${arr[@]}";do
        echo "$j"
        break
    done
done
```

Sortie:

```
a
a
```

```
b
a
c
a
d
a
e
a
f
a
```

Déclaration de changement de cas

Avec l'instruction `case` , vous pouvez associer des valeurs à une variable.

L'argument transmis à `case` est développé et tente de correspondre à chaque modèle.

Si une correspondance est trouvée, les commandes jusqu'à `;;` sont exécutés.

```
case "$BASH_VERSION" in
 [34]*)
    echo {1..4}
    ;;
  *)
    seq -s" " 1 4
esac
```

Les patterns ne sont pas des expressions régulières mais des correspondances de patterns de shell (aussi appelés globs).

Pour une boucle sans paramètre de liste de mots

```
for arg; do
    echo arg=$arg
done
```

Une boucle `for` sans paramètre de liste de mots parcourra les paramètres de position à la place. En d'autres termes, l'exemple ci-dessus est équivalent à ce code:

```
for arg in "$@"; do
    echo arg=$arg
done
```

En d'autres termes, si vous vous surprenez à écrire `for i in "$@"; do ...; done` , juste laisser tomber la `in` une partie, et il suffit d' écrire `for i; do ...; done` .

Exécution conditionnelle des listes de commandes

Comment utiliser l'exécution conditionnelle des listes de commandes

Toute commande, expression ou fonction intégrée, ainsi que toute commande ou script externe

peuvent être exécutés de manière conditionnelle à l'aide des fonctions `&&` *(and)* et `||` *(ou)* opérateurs.

Par exemple, cela imprimera uniquement le répertoire en cours si la commande `cd` a réussi.

```
cd my_directory && pwd
```

De même, cela se terminera si la commande `cd` échoue, empêchant une catastrophe:

```
cd my_directory || exit
rm -rf *
```

Lorsque vous combinez plusieurs instructions de cette manière, il est important de vous rappeler que (contrairement à de nombreux langages de style C) ces opérateurs n'ont pas de priorité et sont associés à gauche .

Ainsi, cette déclaration fonctionnera comme prévu ...

```
cd my_directory && pwd || echo "No such directory"
```

- Si le `cd` réussit, le `&& pwd` s'exécute et le nom du répertoire de travail actuel est imprimé. A moins que `pwd` échoue (une rareté) le `|| echo ...` ne sera pas exécuté.
- Si le `cd` échoue, le `&& pwd` sera ignoré et le `|| echo ...` va courir

Mais ce ne sera pas le cas (si vous pensez `if...then...else`) ...

```
cd my_directory && ls || echo "No such directory"
```

- Si le `cd` échoue, le `&& ls` est ignoré et le `|| echo ...` est exécuté.
- Si le `cd` réussit, le `&& ls` est exécuté.
    - Si le `ls` réussit, le `|| echo ...` est ignoré. *(jusqu'ici tout va bien)*
    - ***MAIS ... si le `ls` échoue, le `|| echo ...` sera également exécuté.***

        C'est le `ls` , pas le `cd` , ***c'est la commande précédente*** .

Pourquoi utiliser une exécution conditionnelle des listes de commandes

L'exécution conditionnelle est un cheveu plus rapide que `if...then` mais son principal avantage est de permettre aux fonctions et aux scripts de sortir tôt, ou "court-circuit".

Contrairement à de nombreux langages tels que `C` où la mémoire est explicitement allouée aux structs et aux variables, et donc (et doit donc être désallouée), `bash` gère cela sous les couvertures. Dans la plupart des cas, nous n'avons rien à nettoyer avant de quitter la fonction. Une déclaration de `return` désallouera tout ce qui est local à la fonction et l'exécution de la collecte à l'adresse de retour de la pile.

Le retour de fonctions ou la sortie de scripts dès que possible peut donc améliorer considérablement les performances et réduire la charge du système en évitant l'exécution inutile

du code. Par exemple...

```
my_function () {

    ### ALWAYS CHECK THE RETURN CODE

    # one argument required. "" evaluates to false(1)
    [[ "$1" ]]             || return 1

    # work with the argument. exit on failure
    do_something_with "$1" || return 1
    do_something_else      || return 1

    # Success! no failures detected, or we wouldn't be here
    return 0
}
```

Lire Structures de contrôle en ligne: https://riptutorial.com/fr/bash/topic/420/structures-de-controle

Chapitre 58: Substitution de processus

Remarques

La substitution de processus est une forme de redirection dans laquelle l'entrée ou la sortie d'un processus (une séquence de commandes) apparaît sous la forme d'un fichier temporaire.

Examples

Comparer deux fichiers du Web

Ce qui suit compare deux fichiers avec `diff` utilisant la substitution de processus au lieu de créer des fichiers temporaires.

```
diff <(curl http://www.example.com/page1) <(curl http://www.example.com/page2)
```

Alimenter une boucle while avec la sortie d'une commande

Cela alimente un `while` en boucle avec la sortie d'un `grep` commande:

```
while IFS=":" read -r user _
do
    # "$user" holds the username in /etc/passwd
done < <(grep "hello" /etc/passwd)
```

Avec la commande coller

```
# Process substitution with paste command is common
# To compare the contents of two directories
paste <( ls /path/to/directory1 ) <( ls /path/to/directory1 )
```

Fichiers concaténés

Il est bien connu que vous ne pouvez pas utiliser le même fichier pour l'entrée et la sortie dans la même commande. Par exemple,

```
$ cat header.txt body.txt >body.txt
```

ne fait pas ce que vous voulez Au moment où `cat` lit `body.txt` , il a déjà été tronqué par la redirection et il est vide. Le résultat final sera que `body.txt` contiendra `body.txt` le contenu de `header.txt` .

On pourrait penser à éviter cela avec la substitution de processus, c'est-à-dire que la commande

```
$ cat header.txt <(cat body.txt) > body.txt
```

forcera le contenu original de `body.txt` à être sauvegardé dans un tampon quelque part avant que le fichier ne soit tronqué par la redirection. Ça ne marche pas Le `cat` entre parenthèses commence à lire le fichier uniquement après que tous les descripteurs de fichiers ont été configurés, tout comme celui externe. Il est inutile d'essayer d'utiliser la substitution de processus dans ce cas.

La seule façon d'ajouter un fichier à un autre fichier consiste à créer un fichier intermédiaire:

```
$ cat header.txt body.txt >body.txt.new
$ mv body.txt.new body.txt
```

c'est ce que font les programmes `sed` ou `perl` ou similaires sous le tapis lorsqu'ils sont appelés avec une option *edit-in-place* (généralement `-i`).

Diffuser un fichier via plusieurs programmes à la fois

Cela compte le nombre de lignes dans un gros fichier avec `wc -l` tout en le compressant simultanément avec `gzip` . Les deux fonctionnent en même temps.

```
tee >(wc -l >&2) < bigfile | gzip > bigfile.gz
```

Normalement, `tee` écrit son entrée dans un ou plusieurs fichiers (et stdout). Nous pouvons écrire dans les commandes au lieu des fichiers avec `tee >(command)` .

Ici, la commande `wc -l >&2` compte les lignes lues depuis le `tee` (qui à son tour lit depuis le `bigfile`). (Le nombre de lignes est envoyé à stderr (`>&2`) pour éviter de mélanger avec l'entrée de `gzip` .) La sortie standard de `tee` est transmise simultanément à `gzip` .

Pour éviter l'utilisation d'un sous-shell

Un aspect majeur de la substitution de processus est qu'il nous permet d'éviter l'utilisation d'un sous-shell lorsqu'il commande des commandes depuis le shell.

Cela peut être démontré par un exemple simple ci-dessous. J'ai les fichiers suivants dans mon dossier actuel:

```
$ find . -maxdepth 1 -type f -print
foo bar zoo foobar foozoo barzoo
```

Si je conduis vers une boucle `while` / `read` qui incrémente un compteur comme suit:

```
count=0
find . -maxdepth 1 -type f -print | while IFS= read -r _; do
    ((count++))
done
```

`$count` maintenant ne contient *pas* `6` , car il a été modifié dans le contexte du sous-shell. Toutes les commandes présentées ci-dessous sont exécutées dans un contexte de sous-shell et la portée

des variables utilisées est perdue après la fin du sous-shell.

```
command &
command | command
( command )
```

La substitution de processus résoudra le problème en évitant d'utiliser le tuyau `|` opérateur comme dans

```
count=0
while IFS= read -r _; do
    ((count++))
done < <(find . -maxdepth 1 -type f -print)
```

Cela conservera la valeur de la variable `count` car aucun sous-shell n'est invoqué.

Lire Substitution de processus en ligne: https://riptutorial.com/fr/bash/topic/2647/substitution-de-processus

Chapitre 59: Substitutions d'histoire de Bash

Examples

En utilisant! $

Vous pouvez utiliser le `!$` Pour réduire la répétition lorsque vous utilisez la ligne de commande:

```
$ echo ping
ping
$ echo !$
ping
```

Vous pouvez également tirer parti de la répétition

```
$ echo !$ pong
ping pong
$ echo !$, a great game
pong, a great game
```

Notez que dans le dernier exemple, nous n'avons pas obtenu de `ping pong, a great game` car le dernier argument passé à la commande précédente était `pong` , nous pouvons éviter un tel problème en ajoutant des guillemets. En continuant avec l'exemple, notre dernier argument était le `game` :

```
$ echo "it is !$ time"
it is game time
$ echo "hooray, !$!"
hooray, it is game time!
```

Référence rapide

Interaction avec l'histoire

```
# List all previous commands
history

# Clear the history, useful if you entered a password by accident
history -c
```

Indicateurs d'événements

```
# Expands to line n of bash history
!n

# Expands to last command
!!
```

```
# Expands to last command starting with "text"
!text

# Expands to last command containing "text"
!?text

# Expands to command n lines ago
!-n

# Expands to last command with first occurrence of "foo" replaced by "bar"
^foo^bar^

# Expands to the current command
!#
```

Désignateurs de mots

Ceux-ci sont séparés par `:` de l'indicateur d'événement auquel ils se réfèrent. Les deux points peuvent être omis si le mot désignateur ne commence pas par un nombre `!^` Est le même que `!:^` .

```
# Expands to the first argument of the most recent command
!^

# Expands to the last argument of the most recent command (short for !!:$)
!$

# Expands to the third argument of the most recent command
!:3

# Expands to arguments x through y (inclusive) of the last command
# x and y can be numbers or the anchor characters ^ $
!:x-y

# Expands to all words of the last command except the 0th
# Equivalent to :^-$
!*
```

Modificateurs

Ceux-ci modifient l'événement précédent ou le désignateur de mot.

```
# Replacement in the expansion using sed syntax
# Allows flags before the s and alternate separators
:s/foo/bar/ #substitutes bar for first occurrence of foo
:gs|foo|bar| #substitutes bar for all foo

# Remove leading path from last argument ("tail")
:t

# Remove trailing path from last argument ("head")
:h

# Remove file extension from last argument
:r
```

Si la variable Bash `HISTCONTROL` contient `ignorespace` ou `ignoreboth` (ou, alternativement, `HISTIGNORE` contient le modèle `[ ]*`), vous pouvez empêcher que vos commandes soient stockées dans l'historique Bash en les ajoutant avec un espace:

```
# This command won't be saved in the history
 foo

# This command will be saved
bar
```

Rechercher dans l'historique des commandes par pattern

Appuyez sur la `commande r` et tapez un motif.

Par exemple, si vous avez récemment exécuté `man 5 crontab` , vous pouvez le trouver rapidement en *commençant à taper* "crontab". L'invite changera comme ceci:

```
(reverse-i-search)`cr': man 5 crontab
```

Le `` `cr' `` il y a la chaîne que j'ai tapé jusqu'ici. Il s'agit d'une recherche incrémentielle. Lorsque vous continuez à taper, le résultat de la recherche est mis à jour pour correspondre à la commande la plus récente contenant le modèle.

Appuyez sur les touches fléchées gauche ou droite pour modifier la commande correspondante avant de l'exécuter ou sur la touche `Entrée` pour exécuter la commande.

Par défaut, la recherche trouve la dernière commande exécutée correspondant au motif. Pour revenir plus loin dans l'historique, appuyez à nouveau sur la `commande r` . Vous pouvez appuyer dessus à plusieurs reprises jusqu'à ce que vous trouviez la commande souhaitée.

Passez au répertoire nouvellement créé avec! #: N

```
$ mkdir backup_download_directory && cd !#:1
mkdir backup_download_directory && cd backup_download_directory
```

Cela remplacera le Nième argument de la commande en cours. Dans l'exemple `!#:1` est remplacé par le premier argument, ie backup_download_directory.

Répéter la commande précédente avec une substitution

```
$ mplayer Lecture_video_part1.mkv
$ ^1^2^
mplayer Lecture_video_part2.mkv
```

Cette commande remplacera `1` par `2` dans la commande précédemment exécutée. Il ne remplacera que la première occurrence de la chaîne et équivaut à `!!:s/1/2/` .

Si vous voulez remplacer *toutes les* occurrences, vous devez utiliser `!!:gs/1/2/` ou `!!:as/1/2/` .

Répéter la commande précédente avec sudo

```
$ apt-get install r-base
E: Could not open lock file /var/lib/dpkg/lock - open (13: Permission denied)
E: Unable to lock the administration directory (/var/lib/dpkg/), are you root?
$ sudo !!
sudo apt-get install r-base
[sudo] password for <user>:
```

Lire Substitutions d'histoire de Bash en ligne:
https://riptutorial.com/fr/bash/topic/1519/substitutions-d-histoire-de-bash

Chapitre 60: Tableaux

Examples

Assignations de tableaux

Affectation de liste

Si vous êtes familier avec Perl, C ou Java, vous pourriez penser que Bash utilisera des virgules pour séparer les éléments du tableau, mais ce n'est pas le cas; Bash utilise plutôt des espaces:

```
# Array in Perl
my @array = (1, 2, 3, 4);
```

```
# Array in Bash
array=(1 2 3 4)
```

Créez un tableau avec de nouveaux éléments:

```
array=('first element' 'second element' 'third element')
```

Affectation des indices

Créez un tableau avec des index d'éléments explicites:

```
array=([3]='fourth element' [4]='fifth element')
```

Affectation par index

```
array[0]='first element'
array[1]='second element'
```

Affectation par nom (tableau associatif)

4.0

```
declare -A array
array[first]='First element'
array[second]='Second element'
```

Affectation dynamique

Créez un tableau à partir de la sortie d'une autre commande, par exemple, utilisez **seq** pour obtenir une plage de 1 à 10:

```
array=(`seq 1 10`)
```

Affectation à partir des arguments d'entrée du script:

```
array=("$@")
```

Affectation dans les boucles:

```
while read -r; do
    #array+=("$REPLY")      # Array append
    array[$i]="$REPLY"      # Assignment by index
    let i++                 # Increment index
done < <(seq 1 10)  # command substitution
echo ${array[@]}    # output: 1 2 3 4 5 6 7 8 9 10
```

où `$REPLY` est toujours l'entrée courante

Accès aux éléments du tableau

Imprimer l'élément à l'index 0

```
echo "${array[0]}"
```

4.3

Imprimer le dernier élément à l'aide de la syntaxe d'extension de sous-chaîne

```
echo "${arr[@]: -1 }"
```

4.3

Imprimer le dernier élément en utilisant la syntaxe de l'indice

```
echo "${array[-1]}"
```

Imprimer tous les éléments, chacun cité séparément

```
echo "${array[@]}"
```

Imprimer tous les éléments en une seule chaîne entre guillemets

```
echo "${array[*]}"
```

Imprimer tous les éléments de l'index 1, chacun cité séparément

```
echo "${array[@]:1}"
```

Imprimer 3 éléments de l'index 1, chacun cité séparément

```
echo "${array[@]:1:3}"
```

Opérations sur les cordes

Si vous vous référez à un seul élément, les opérations sur les chaînes sont autorisées:

```
array=(zero one two)
echo "${array[0]:0:3}" # gives out zer (chars at position 0, 1 and 2 in the string zero)
echo "${array[0]:1:3}" # gives out ero (chars at position 1, 2 and 3 in the string zero)
```

si `${array[$i]:N:M}` donne une chaîne de la `N` ième position (à partir de 0) dans la chaîne `${array[$i]}` avec `M` caractères suivants.

Longueur du tableau

`${#array[@]}` donne la longueur du tableau `${array[@]}` :

```
array=('first element' 'second element' 'third element')
echo "${#array[@]}" # gives out a length of 3
```

Cela fonctionne également avec des chaînes dans des éléments simples:

```
echo "${#array[0]}"    # gives out the lenght of the string at element 0: 13
```

Modification de tableau

Changer d'index

Initialiser ou mettre à jour un élément particulier du tableau

```
array[10]="elevenths element"    # because it's starting with 0
```

3.1

Ajouter

Modifiez le tableau en ajoutant des éléments à la fin si aucun indice n'est spécifié.

```
array+=('fourth element' 'fifth element')
```

Remplacez le tableau entier par une nouvelle liste de paramètres.

```
array=("${array[@]}" "fourth element" "fifth element")
```

Ajouter un élément au début:

```
array=("new element" "${array[@]}")
```

Insérer

Insérer un élément à un index donné:

```
arr=(a b c d)
# insert an element at index 2
i=2
arr=("${arr[@]:0:$i}" 'new' "${arr[@]:$i}")
echo "${arr[2]}" #output: new
```

Effacer

Supprimer les index de tableau à l'aide de la commande `unset` intégrée:

```
arr=(a b c)
echo "${arr[@]}"   # outputs: a b c
echo "${!arr[@]}"  # outputs: 0 1 2
unset -v 'arr[1]'
echo "${arr[@]}"   # outputs: a c
echo "${!arr[@]}"  # outputs: 0 2
```

Fusionner

```
array3=("${array1[@]}" "${array2[@]}")
```

Cela fonctionne également pour les tableaux épars.

Réindexer un tableau

Cela peut être utile si des éléments ont été supprimés d'un tableau ou si vous n'êtes pas certain qu'il existe des lacunes dans le tableau. Pour recréer les index sans lacunes:

```
array=("${array[@]}")
```

Itération du tableau

L'itération Array se décline en deux versions: foreach et le classique for-loop:

```
a=(1 2 3 4)
# foreach loop
for y in "${a[@]}"; do
    # act on $y
    echo "$y"
done
# classic for-loop
for ((idx=0; idx < ${#a[@]}; ++idx)); do
    # act on ${a[$idx]}
    echo "${a[$idx]}"
done
```

Vous pouvez également parcourir la sortie d'une commande:

```
a=($(tr ',' ' ' <<<"a,b,c,d")) # tr can transform one character to another
```

```
for y in "${a[@]}"; do
    echo "$y"
done
```

Détruire, supprimer ou supprimer un tableau

Pour détruire, supprimer ou supprimer un tableau:

```
unset array
```

Pour détruire, supprimer ou supprimer un seul élément de tableau:

```
unset array[10]
```

Tableaux associatifs

4.0

Déclarez un tableau associatif

```
declare -A aa
```

La déclaration d'un tableau associatif avant l'initialisation ou l'utilisation est obligatoire.

Initialiser des éléments

Vous pouvez initialiser les éléments un par un comme suit:

```
aa[hello]=world
aa[ab]=cd
aa["key with space"]="hello world"
```

Vous pouvez également initialiser un tableau associatif entier dans une seule instruction:

```
aa=([hello]=world [ab]=cd ["key with space"]="hello world")
```

Accéder à un élément de tableau associatif

```
echo ${aa[hello]}
# Out: world
```

Liste des clés de tableau associatif

```
echo "${!aa[@]}"
#Out: hello ab key with space
```

Liste des valeurs de tableau associatif

```
echo "${aa[@]}"
#Out: world cd hello world
```

Itérer sur des clés et des valeurs de tableau associatif

```
for key in "${!aa[@]}"; do
    echo "Key:   ${key}"
    echo "Value: ${array[$key]}"
done

# Out:
# Key:   hello
# Value: world
# Key:   ab
# Value: cd
# Key:   key with space
# Value: hello world
```

Compter les éléments de tableau associatif

```
echo "${#aa[@]}"
# Out: 3
```

Liste des index initialisés

Récupère la liste des index inialisés dans un tableau

```
$ arr[2]='second'
$ arr[10]='tenth'
$ arr[25]='twenty five'
$ echo ${!arr[@]}
2 10 25
```

En boucle à travers un tableau

Notre exemple de tableau:

```
arr=(a b c d e f)
```

Utiliser une boucle `for..in` :

```
for i in "${arr[@]}"; do
    echo "$i"
done
```

2,04

Utiliser le style C `for` boucle:

```
for ((i=0;i<${#arr[@]};i++)); do
    echo "${arr[$i]}"
```

```
done
```

L' utilisation `while` boucle:

```
i=0
while [ $i -lt ${#arr[@]} ]; do
    echo "${arr[$i]}"
    i=$((i + 1))
done
```

2,04

L' utilisation `while` boucle avec condition numérique:

```
i=0
while (( $i < ${#arr[@]} )); do
    echo "${arr[$i]}"
    ((i++))
done
```

Utiliser une boucle `until` :

```
i=0
until [ $i -ge ${#arr[@]} ]; do
    echo "${arr[$i]}"
    i=$((i + 1))
done
```

2,04

Utiliser une boucle `until` avec un conditionnel numérique:

```
i=0
until (( $i >= ${#arr[@]} )); do
    echo "${arr[$i]}"
    ((i++))
done
```

Tableau de la chaîne

```
stringVar="Apple Orange Banana Mango"
arrayVar=(${stringVar// / })
```

Chaque espace dans la chaîne indique un nouvel élément dans le tableau résultant.

```
echo ${arrayVar[0]} # will print Apple
echo ${arrayVar[3]} # will print Mango
```

De même, d'autres caractères peuvent être utilisés pour le délimiteur.

```
stringVar="Apple+Orange+Banana+Mango"
```

```
arrayVar=(${stringVar//+/ })
echo ${arrayVar[0]} # will print Apple
echo ${arrayVar[2]} # will print Banana
```

Fonction d'insertion de tableau

Cette fonction va insérer un élément dans un tableau à un index donné:

```
insert(){
    h='
################## insert ########################
# Usage:
#   insert arr_name index element
#
#   Parameters:
#       arr_name    : Name of the array variable
#       index       : Index to insert at
#       element     : Element to insert
##################################################
    '
    [[ $1 = -h ]] && { echo "$h" >/dev/stderr; return 1; }
    declare -n __arr__=$1   # reference to the array variable
    i=$2                    # index to insert at
    el="$3"                 # element to insert
    # handle errors
    [[ ! "$i" =~ ^[0-9]+$ ]] && { echo "E: insert: index must be a valid integer"
>/dev/stderr; return 1; }
    (( $1 < 0 )) && { echo "E: insert: index can not be negative" >/dev/stderr; return 1; }
    # Now insert $el at $i
    __arr__=("${__arr__[@]:0:$i}" "$el" "${__arr__[@]:$i}")
}
```

Usage:

```
insert array_variable_name index element
```

Exemple:

```
arr=(a b c d)
echo "${arr[2]}" # output: c
# Now call the insert function and pass the array variable name,
# index to insert at
# and the element to insert
insert arr 2 'New Element'
# 'New Element' was inserted at index 2 in arr, now print them
echo "${arr[2]}" # output: New Element
echo "${arr[3]}" # output: c
```

Lire un fichier entier dans un tableau

Lecture en une seule étape:

```
IFS=$'\n' read -r -a arr < file
```

Lecture en boucle:

```
arr=()
while IFS= read -r line; do
  arr+=("$line")
done
```

4.0

En utilisant `mapfile` ou `readarray` (qui sont synonymes):

```
mapfile -t arr < file
readarray -t arr < file
```

Lire Tableaux en ligne: https://riptutorial.com/fr/bash/topic/471/tableaux

Chapitre 61: Tableaux associatifs

Syntaxe

- déclarer -A assoc_array # sans initialiser
- declare -A assoc_array = ([key] = "value" [une autre clé] = "attention aux espaces" [trois espaces] = "tous les blancs résument")
- echo $ {assoc_array [@]} # les valeurs
- echo $ {! assoc_array [@]} # les clés

Examples

Examen des tableaux d'assoc

Toutes les utilisations nécessaires affichées avec cet extrait de code:

```
#!/usr/bin/env bash

declare -A assoc_array=([key_string]=value                                  \
                        [one]="something"                                   \
                        [two]="another thing"                               \
                        [ three ]='mind the blanks!'                        \
                        [ " four" ]='count the blanks of this key later!'   \
                                                [IMPORTANT]='SPACES DO ADD UP!!!'
\
                        [1]='there are no integers!'                        \
                        [info]="to avoid history expansion "                \
                        [info2]="quote exclamation mark with single quotes" \
                       )
echo # just a blank line
echo now here are the values of assoc_array:
echo ${assoc_array[@]}
echo not that useful,
echo # just a blank line
echo this is better:

declare -p assoc_array    # -p == print

echo have a close look at the spaces above\!\!\!
echo # just a blank line

echo accessing the keys
echo the keys in assoc_array are ${!assoc_array[*]}
echo mind the use of indirection operator \!
echo # just a blank line


echo now we loop over the assoc_array line by line
echo note the \! indirection operator which works differently,
echo if used with assoc_array.
echo # just a blank line

for key in "${!assoc_array[@]}"; do # accessing keys using ! indirection!!!!
```

```
   printf "key: \"%s\"\nvalue: \"%s\"\n\n" "$key" "${assoc_array[$key]}"
done


echo have a close look at the spaces in entries with keys two, three and four above\!\!\!
echo # just a blank line
echo # just another blank line

echo there is a difference using integers as keys\!\!\!
i=1
echo declaring an integer var i=1
echo # just a blank line
echo Within an integer_array bash recognizes artithmetic context.
echo Within an assoc_array bash DOES NOT recognize artithmetic context.
echo # just a blank line
echo this works: \${assoc_array[\$i]}:  ${assoc_array[$i]}
echo this NOT!!: \${assoc_array[i]}:  ${assoc_array[i]}
echo # just a blank line
echo # just a blank line
echo an \${assoc_array[i]} has a string context within braces in contrast to an integer_array
declare -i integer_array=( one two three )
echo "doing a: declare -i integer_array=( one two three )"
echo # just a blank line

echo both forms do work: \${integer_array[i]} : ${integer_array[i]}
echo and this too:  \${integer_array[\$i]} : ${integer_array[$i]}
```

Lire Tableaux associatifs en ligne: https://riptutorial.com/fr/bash/topic/7536/tableaux-associatifs

Chapitre 62: Transfert de fichier à l'aide de scp

Syntaxe

- scp / some / local / répertoire / nom_fichier nom_utilisateur @ nom_hôte: chemin_fichier_destination
- scp nom_utilisateur @ nom_hôte: chemin_fichier_origine / répertoire / local /

Examples

scp transférer le fichier

Pour transférer un fichier en toute sécurité sur une autre machine - tapez:

```
scp file1.txt tom@server2:$HOME
```

Cet exemple présente le transfert de `file1.txt` depuis notre hôte vers le `server2` de l'utilisateur `tom` de `server2` .

scp transférer plusieurs fichiers

`scp` peut également être utilisé pour transférer plusieurs fichiers d'un serveur à un autre. Vous trouverez ci-dessous un exemple de transfert de tous les fichiers du répertoire `my_folder` avec l'extension `.txt` vers `server2` . Dans l'exemple ci-dessous, tous les fichiers seront transférés vers le répertoire utilisateur de l'utilisateur `tom` .

```
scp /my_folder/*.txt tom@server2:$HOME
```

Télécharger le fichier en utilisant scp

Pour télécharger un fichier du serveur distant vers la machine locale - tapez:

```
scp tom@server2:$HOME/file.txt /local/machine/path/
```

Cet exemple montre comment télécharger le fichier nommé `file.txt` du répertoire personnel de l'utilisateur `tom` vers le répertoire actuel de notre machine locale.

Lire Transfert de fichier à l'aide de scp en ligne: https://riptutorial.com/fr/bash/topic/5484/transfert-de-fichier-a-l-aide-de-scp

Chapitre 63: Trouver

Introduction

find est une commande permettant de rechercher de manière récursive un répertoire pour des fichiers (ou répertoires) correspondant à un critère, puis d'effectuer une action sur les fichiers sélectionnés.

trouver search_path selection_criteria action

Syntaxe

- find [-H] [-L] [-P] [-D debugopts] [-Olevel] [chemin ...] [expression]

Examples

Recherche d'un fichier par nom ou par extension

Pour rechercher des fichiers / répertoires avec un nom spécifique, par rapport à `pwd` :

```
$ find . -name "myFile.txt"
./myFile.txt
```

Pour rechercher des fichiers / répertoires avec une extension spécifique, utilisez un caractère générique:

```
$ find . -name "*.txt"
./myFile.txt
./myFile2.txt
```

Pour rechercher des fichiers / répertoires correspondant à l'une des nombreuses extensions, utilisez le drapeau `or` :

```
$ find . -name "*.txt" -o -name "*.sh"
```

Pour trouver des fichiers / répertoires dont le nom commence par abc et se termine par un caractère alpha à la suite d'un chiffre:

```
$ find . -name "abc[a-z][0-9]"
```

Pour trouver tous les fichiers / répertoires situés dans un répertoire spécifique

```
$ find /opt
```

Pour rechercher uniquement des fichiers (pas des répertoires), utilisez `-type f` :

```
find /opt -type f
```

Pour rechercher uniquement des répertoires (pas des fichiers réguliers), utilisez `-type d` :

```
find /opt -type d
```

Recherche de fichiers par type

Pour rechercher des fichiers, utilisez le drapeau `-type f`

```
$ find . -type f
```

Pour trouver des répertoires, utilisez le drapeau `-type d`

```
$ find . -type d
```

Pour rechercher des périphériques de bloc, utilisez l'indicateur `-type b`

```
$ find /dev -type b
```

Pour trouver des liens symboliques, utilisez le drapeau `-type l`

```
$ find . -type l
```

Exécuter des commandes sur un fichier trouvé

Parfois, nous devrons exécuter des commandes sur beaucoup de fichiers. Cela peut être fait en utilisant `xargs` .

```
find . -type d -print | xargs -r chmod 770
```

La commande ci-dessus trouvera de manière récursive tous les répertoires (`-type d`) relatifs à `.` (qui est votre répertoire de travail actuel) et exécutez `chmod 770` sur eux. L'option `-r` indique à `xargs` de ne pas exécuter `chmod` si `find` n'a trouvé aucun fichier.

Si vos noms de fichiers ou répertoires contiennent un caractère d'espace, cette commande peut être bloquée; une solution consiste à utiliser les éléments suivants

```
find . -type d -print0 | xargs -r -0 chmod 770
```

Dans l'exemple ci-dessus, les `-print0` et `-0` spécifient que les noms de fichiers seront séparés à l'aide d'un octet `null` et permettent l'utilisation de caractères spéciaux, tels que des espaces, dans les noms de fichiers. Ceci est une extension GNU et peut ne pas fonctionner dans d'autres versions de `find` et `xargs` .

La méthode privilégiée consiste à ignorer la commande `xargs` et à `find` le sous-processus lui-même:

```
find . -type d -exec chmod 770 {} \;
```

Ici, le `{}` est un espace réservé indiquant que vous souhaitez utiliser le nom du fichier à ce stade. `find` exécutera `chmod` sur chaque fichier individuellement.

Vous pouvez également passer tous les noms de fichiers à un *seul* appel de `chmod` , en utilisant

```
find . -type d -exec chmod 770 {} +
```

C'est aussi le comportement des extraits de `xargs` ci-dessus. (Pour appeler chaque fichier individuellement, vous pouvez utiliser `xargs -n1`).

Une troisième option consiste à laisser bash la boucle sur la liste des noms de fichiers `find` résultats:

```
find . -type d | while read -r d; do chmod 770 "$d"; done
```

Ceci est syntaxiquement le plus lourd, mais pratique lorsque vous souhaitez exécuter plusieurs commandes sur chaque fichier trouvé. Cependant, cela est **dangereux** face aux noms de fichiers avec des noms impairs.

```
find . -type f | while read -r d; do mv "$d" "${d// /_}"; done
```

qui remplacera tous les espaces dans les noms de fichiers par des traits de soulignement. (Cet exemple ne fonctionnera pas non plus s'il y a des espaces dans les noms de *répertoires* principaux.)

Le problème avec ce qui précède est que `while read -r` attend une entrée par ligne, mais que les noms de fichiers peuvent contenir des nouvelles lignes (et aussi, `read -r` perdra tout espace vide). Vous pouvez résoudre ce problème en retournant les choses:

```
find . -type d -exec bash -c 'for f; do mv "$f" "${f// /_}"; done' _ {} +
```

De cette façon, `-exec` reçoit les noms de fichiers sous une forme complètement correcte et portable; le `bash -c` reçoit sous la forme d'un certain nombre d'arguments, qui seront trouvés dans `$@` , correctement cités, etc. citations.)

Le mystérieux _ est nécessaire car le premier argument de `bash -c 'script'` est utilisé pour remplir `$0` .

Recherche de fichier par temps d'accès / modification

Sur un système de fichiers `ext` , chaque fichier est associé à une heure d'accès, de modification et de changement (Status) associée. Pour afficher ces informations, vous pouvez utiliser `stat myFile.txt` ; En utilisant les drapeaux dans *find* , nous pouvons rechercher des fichiers qui ont été

modifiés dans un certain intervalle de temps.

Pour rechercher des fichiers qui *ont* été modifiés au cours des 2 dernières heures:

```
$ find . -mmin -120
```

Pour rechercher des fichiers qui *n'ont pas* été modifiés au cours des 2 dernières heures:

```
$ find . -mmin +120
```

L'exemple ci - dessus sont à la recherche que sur le temps *modifié* - à la recherche sur **un** temps de ccès, ou **c** pendu fois, utilisez `a` ou `c` en conséquence.

```
$ find . -amin -120
$ find . -cmin +120
```

Format général:

`-mmin n` : le fichier a été modifié *n* minutes auparavant
`-mmin -n` : Le fichier a été modifié il y a moins de *n* minutes
`-mmin +n` : le fichier a été modifié il y a plus de *n* minutes

Recherchez les fichiers qui *ont* été modifiés au cours des 2 derniers jours:

```
find . -mtime -2
```

Recherchez les fichiers qui *n'ont pas* été modifiés au cours des 2 derniers jours

```
find . -mtime +2
```

Utilisez `-atime` et `-ctime` pour le temps d'accès et l'heure de changement d'état respectivement.

Format général:

`-mtime n` : le fichier a été modifié il y a *24* heures
`-mtime -n` : le fichier a été modifié il y a moins de *24* heures
`-mtime +n` : le fichier a été modifié il y a plus de *24* heures

Recherchez les fichiers modifiés dans une ***plage de dates*** , du 2007-06-07 au 2007-06-08:

```
find . -type f -newermt 2007-06-07 ! -newermt 2007-06-08
```

Recherchez les fichiers accessibles dans une ***plage d'horodatages*** (en utilisant des fichiers comme horodatage), il y a 1 heure à 10 minutes:

```
touch -t $(date -d '1 HOUR AGO' +%Y%m%d%H%M.%S) start_date
touch -t $(date -d '10 MINUTE AGO' +%Y%m%d%H%M.%S) end_date
timeout 10 find "$LOCAL_FOLDER" -newerat "start_date" ! -newerat "end_date" -print
```

Format général:

`-newerXY reference` : Compare l'horodatage du fichier en cours avec la référence. `XY` peut avoir l'une des valeurs suivantes: `at` (heure d'accès), `mt` (heure de modification), `ct` (heure de modification) et plus. `reference` est le *nom d'un fichier que l'on* souhaite comparer à l'horodatage spécifié (accès, modification, modification) ou à une *chaîne* décrivant une heure absolue.

Recherche de fichiers par extension spécifique

Pour rechercher tous les fichiers d'une certaine extension dans le chemin d'accès actuel, vous pouvez utiliser la syntaxe de `find` suivante. Cela fonctionne en utilisant `bash's` construction de `glob` intégrée de `bash's` pour correspondre à tous les noms ayant le `.extension` .

```
find /directory/to/search -maxdepth 1 -type f -name "*.extension"
```

Pour trouver tous les fichiers de type `.txt` du répertoire actuel seul, faites

```
find . -maxdepth 1 -type f -name "*.txt"
```

Recherche de fichiers en fonction de leur taille

Trouvez des fichiers de plus de 15 Mo:

```
find -type f -size +15M
```

Trouver des fichiers moins de 12 Ko:

```
find -type f -size -12k
```

Trouvez des fichiers d'une taille de 12 Ko exactement:

```
find -type f -size 12k
```

Ou

```
find -type f -size 12288c
```

Ou

```
find -type f -size 24b
```

Ou

```
find -type f -size 24
```

Format général:

```
find [options] -size n[cwbkMG]
```

Trouver des fichiers de taille n-block, où + n signifie plus que n-block, -n signifie moins que n-block et n (sans aucun signe) signifie exactement n-block

Taille de bloc:

1. `c` : octets
2. `w` : 2 octets
3. `b` : 512 octets (par défaut)
4. `k` : 1 Ko
5. `M` : 1 Mo
6. `G` : 1 Go

Filtrer le chemin

Le paramètre `-path` permet de spécifier un motif correspondant au chemin du résultat. Le motif peut également correspondre au nom lui-même.

Pour rechercher uniquement les fichiers contenant un `log` n'importe où sur leur chemin (dossier ou nom):

```
find . -type f -path '*log*'
```

Pour rechercher uniquement les fichiers dans un dossier appelé `log` (à n'importe quel niveau):

```
find . -type f -path '*/log/*'
```

Pour rechercher uniquement les fichiers dans un dossier appelé `log` ou `data` :

```
find . -type f -path '*/log/*' -o -path '*/data/*'
```

Pour rechercher tous les fichiers **sauf** ceux contenus dans un dossier appelé `bin` :

```
find . -type f -not -path '*/bin/*'
```

Pour trouver tous les fichiers, **sauf** ceux contenus dans un dossier appelé `bin` ou fichiers journaux:

```
find . -type f -not -path '*log' -not -path '*/bin/*'
```

Lire Trouver en ligne: https://riptutorial.com/fr/bash/topic/566/trouver

Chapitre 64: true, false et: commandes

Syntaxe

- true,: - renvoie toujours 0 comme code de sortie.
- false - renvoie toujours 1 comme code de sortie.

Examples

Boucle infinie

```
while true; do
    echo ok
done
```

ou

```
while :; do
   echo ok
done
```

ou

```
until false; do
    echo ok
done
```

Fonction Retour

```
function positive() {
    return 0
}

function negative() {
    return 1
}
```

Code qui sera toujours / jamais exécuté

```
if true; then
    echo Always executed
fi
if false; then
    echo Never executed
fi
```

Lire true, false et: commandes en ligne: https://riptutorial.com/fr/bash/topic/6655/true--false-et--

commandes

Chapitre 65: Type de coquille

Remarques

Login Shell

Un shell de connexion est un shell dont le premier caractère de l'argument zéro est un - ou un qui a été lancé avec l'option –login. L'initialisation est plus complète que dans un (sous) shell interactif normal.

Shell interactif

Un shell interactif est un shell démarré sans arguments non optionnels et sans l'option -c dont l'entrée et l'erreur standard sont toutes deux connectées aux terminaux (comme déterminé par isatty (3)), ou une option démarrée avec l'option -i. PS1 est défini et $ - inclut i si bash est interactif, permettant à un script shell ou à un fichier de démarrage de tester cet état.

Shell non interactif

Un shell non interactif est un shell dans lequel l'utilisateur ne peut pas interagir avec le shell. Par exemple, un shell exécutant un script est toujours un shell non interactif. Le script peut quand même accéder à son tty.

Configuration d'un shell de connexion

En vous connectant:

```
If '/etc/profile' exists, then source it.
If '~/.bash_profile' exists, then source it,
else if '~/.bash_login' exists, then source it,
else if '~/.profile' exists, then source it.
```

Pour les shells interactifs non connectés

Au démarrage:

```
If `~/.bashrc' exists, then source it.
```

Pour les coques non interactives

Au démarrage: Si la variable d'environnement ENV est non nulle, développez la variable et sourcez le fichier nommé par la valeur. Si Bash n'est pas démarré en mode Posix, il recherche BASH_ENV avant ENV.

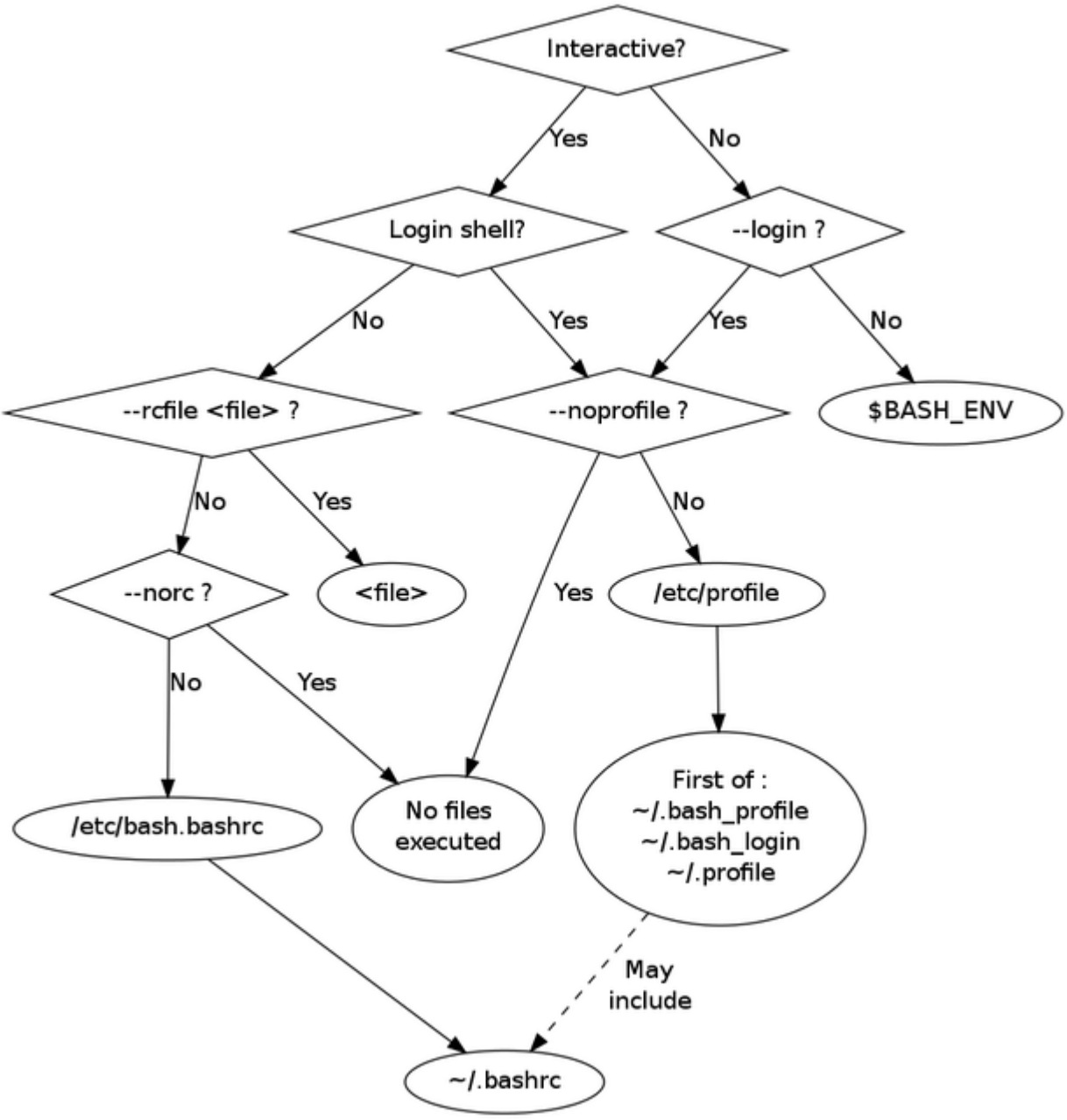
Examples

Introduction aux fichiers de points

Dans Unix, les fichiers et répertoires commençant par un point contiennent généralement des paramètres pour un programme / une série de programmes spécifiques. Les fichiers de points sont généralement cachés à l'utilisateur, vous devrez donc lancer `ls -a` pour les voir.

Un exemple de fichier de points est `.bash_history` , qui contient les dernières commandes exécutées, en supposant que l'utilisateur utilise Bash.

Il existe divers fichiers provenant de l'architecture Bash. L'image ci-dessous, extraite de ce site , montre le processus de décision derrière le choix des fichiers à générer au démarrage.



Démarrer un shell interactif

```
bash
```

Détecter le type de coque

```
shopt -q login_shell && echo 'login' || echo 'not-login'
```

Lire Type de coquille en ligne: https://riptutorial.com/fr/bash/topic/6517/type-de-coquille

Chapitre 66: URL de décodage

Examples

Exemple simple

URL codée

> http% 3A% 2F% 2Fwww.foo.com% 2Findex.php% 3Fid% 3Dqwerty

Utilisez cette commande pour décoder l'URL

```
echo "http%3A%2F%2Fwww.foo.com%2Findex.php%3Fid%3Dqwerty" | sed -e "s/%\([0-9A-F][0-9A-
F]\)/\\\\\x\1/g" | xargs -0 echo -e
```

URL décodée (résultat de la commande)

> http://www.foo.com/index.php?id=qwerty

Utiliser printf pour décoder une chaîne

```
#!bin/bash

$ string='Question%20-
%20%22how%20do%20I%20decode%20a%20percent%20encoded%20string%3F%22%0AAnswer%20%20%20-
%20Use%20printf%20%3A)'
$ printf '%b\n' "${string//%/\\x}"

# the result
Question - "how do I decode a percent encoded string?"
Answer   - Use printf :)
```

Lire URL de décodage en ligne: https://riptutorial.com/fr/bash/topic/10895/url-de-decodage

Chapitre 67: Utiliser "trap" pour réagir aux signaux et aux événements du système

Syntaxe

- action piège sigspec ... # Exécuter "action" sur une liste de signaux
- piège sigspec ... # Omettre l'action réinitialise les traps pour les signaux

Paramètres

Paramètre	Sens
-p	Liste des pièges actuellement installés
-l	Liste des noms de signaux et numéros correspondants

Remarques

Le `trap` utilitaire est une enveloppe spéciale intégrée. Il est défini dans POSIX , mais bash ajoute également des extensions utiles.

Les exemples compatibles avec POSIX commencent par `#!/bin/sh` et les exemples commençant par `#!/bin/bash` utilisent une extension bash.

Les signaux peuvent être soit un numéro de signal, un nom de signal (sans le préfixe SIG), soit le mot-clé spécial `EXIT` .

Ceux garantis par POSIX sont:

Nombre	prénom	Remarques
0	SORTIE	Toujours exécuter à la sortie du shell, quel que soit le code de sortie
1	SIGHUP	
2	SIGINT	C'est ce que `^C` envoie
3	SIGQUIT	
6	SIGABRT	
9	SIGKILL	
14	SIGALRM	

Nombre	prénom	Remarques
15	SIGTERM	C'est ce que `kill` envoie par défaut

Examples

SIGINT de capture ou Ctl + C

Le piège est réinitialisé pour les sous-couches, de sorte que le `sleep` agira toujours sur le signal `SIGINT` envoyé par `^C` (généralement en quittant), mais le processus parent (c'est-à-dire le script shell) ne le fera pas.

```
#!/bin/sh

# Run a command on signal 2 (SIGINT, which is what ^C sends)
sigint() {
    echo "Killed subshell!"
}
trap sigint INT

# Or use the no-op command for no output
#trap : INT

# This will be killed on the first ^C
echo "Sleeping..."
sleep 500

echo "Sleeping..."
sleep 500
```

Et une variante qui vous permet encore de quitter le programme principal en appuyant deux fois sur `^C` :

```
last=0
allow_quit() {
    [ $(date +%s) -lt $(( $last + 1 )) ] && exit
    echo "Press ^C twice in a row to quit"
    last=$(date +%s)
}
trap allow_quit INT
```

Introduction: nettoyer les fichiers temporaires

Vous pouvez utiliser la commande `trap` pour "piéger" les signaux; c'est l'équivalent shell de l'appel `signal()` ou `sigaction()` en C et de la plupart des autres langages de programmation pour capter les signaux.

L'une des utilisations les plus courantes de `trap` est de nettoyer les fichiers temporaires à la sortie attendue et inattendue.

Malheureusement, pas assez de scripts shell le font :-(

```
#!/bin/sh

# Make a cleanup function
cleanup() {
  rm --force -- "${tmp}"
}

# Trap the special "EXIT" group, which is always run when the shell exits.
trap cleanup EXIT

# Create a temporary file
tmp="$(mktemp -p /tmp tmpfileXXXXXXX)"

echo "Hello, world!" >> "${tmp}"

# No rm -f "$tmp" needed. The advantage of using EXIT is that it still works
# even if there was an error or if you used exit.
```

Cumulez une liste de trappes à exécuter à la sortie.

Avez-vous déjà oublié d'ajouter un `trap` pour nettoyer un fichier temporaire ou d'autres travaux à la sortie?

Avez-vous déjà mis un piège qui a annulé un autre?

Ce code facilite l'ajout d'éléments à la sortie d'un élément à la fois, plutôt que d'avoir une seule instruction `trap` quelque part dans votre code, ce qui peut être facile à oublier.

```
# on_exit and add_on_exit
# Usage:
#   add_on_exit rm -f /tmp/foo
#   add_on_exit echo "I am exiting"
#   tempfile=$(mktemp)
#   add_on_exit rm -f "$tempfile"
# Based on http://www.linuxjournal.com/content/use-bash-trap-statement-cleanup-temporary-files
function on_exit()
{
    for i in "${on_exit_items[@]}"
    do
        eval $i
    done
}
function add_on_exit()
{
    local n=${#on_exit_items[*]}
    on_exit_items[$n]="$*"
    if [[ $n -eq 0 ]]; then
        trap on_exit EXIT
    fi
}
```

Tuer les processus de l'enfant à la sortie

Les expressions pièges ne doivent pas nécessairement être des fonctions ou des programmes individuels, elles peuvent également être des expressions plus complexes.

En combinant `jobs -p` et `kill` , nous pouvons tuer tous les processus enfants générés par le shell à la sortie:

```
trap 'jobs -p | xargs kill' EXIT
```

réagir au changement de taille de la fenêtre des terminaux

Il y a un signal WINCH (WINdowCHange), qui est déclenché quand on redimensionne une fenêtre de terminal.

```
declare -x rows cols

update_size(){
  rows=$(tput lines) # get actual lines of term
  cols=$(tput cols)  # get actual columns of term
  echo DEBUG terminal window has no $rows lines and is $cols characters wide
}

trap update_size WINCH
```

Lire Utiliser "trap" pour réagir aux signaux et aux événements du système en ligne: https://riptutorial.com/fr/bash/topic/363/utiliser--trap--pour-reagir-aux-signaux-et-aux-evenements-du-systeme

Chapitre 68: Utilité du sommeil

Introduction

La commande de sommeil peut être utilisée pour faire une pause pendant un temps donné.

Si vous souhaitez utiliser des entrées différentes, utilisez comme ceci Secondes: $ sleep 1s (secondes par défaut) Minutes: $ sleep 1m Hours: $ sleep 1h jours: $ sleep 1d

Si vous voulez dormir moins d'une seconde, utilisez $ sleep 0.5 Vous pouvez utiliser comme ci-dessus selon vos besoins.

Examples

$ dormir 1

Ici, le processus initié cet appel dormira pendant 1 seconde.

Lire Utilité du sommeil en ligne: https://riptutorial.com/fr/bash/topic/10879/utilite-du-sommeil

Chapitre 69: Variables de typage

Examples

déclarer des variables faiblement typées

declare est une commande interne de bash. (commande interne utiliser l' **aide** pour afficher "manpage"). Il est utilisé pour afficher et définir des variables ou afficher des corps de fonctions.

Syntaxe: **declare [options] [nom [= valeur]] ...**

```
# options are used to define
# an integer
declare -i myInteger
declare -i anotherInt=10
# an array with values
declare -a anArray=( one two three)
# an assoc Array
declare -A assocArray=( [element1]="something" [second]=anotherthing )
# note that bash recognizes the string context within []

# some modifiers exist
# uppercase content
declare -u big='this will be uppercase'
# same for lower case
declare -l small='THIS WILL BE LOWERCASE'

# readonly array
declare -ra constarray=( eternal true and unchangeable )

# export integer to environment
declare -xi importantInt=42
```

Vous pouvez aussi utiliser le + qui supprime l'attribut donné. Surtout inutile, juste pour la complétude.

Pour afficher des variables et / ou des fonctions, il existe également des options

```
# printing definded vars and functions
declare -f
# restrict output to functions only
declare -F # if debugging prints line number and filename defined in too
```

Lire Variables de typage en ligne: https://riptutorial.com/fr/bash/topic/7195/variables-de-typage

Chapitre 70: variables globales et locales

Introduction

Par défaut, chaque variable dans bash est **globale** pour chaque fonction, script et même pour le shell externe si vous déclarez vos variables dans un script.

Si vous voulez que votre variable soit locale à une fonction, vous pouvez utiliser `local` pour que cette variable soit une nouvelle variable indépendante de la portée globale et dont la valeur ne sera accessible qu'à l'intérieur de cette fonction.

Examples

Variables globales

```
var="hello"

function foo(){
    echo $var
}

foo
```

Va évidemment sortir "bonjour", mais cela fonctionne aussi dans l'autre sens:

```
function foo()  {
    var="hello"
}

foo
echo $var
```

Va aussi sortir `"hello"`

Variables locales

```
function foo() {
    local var
    var="hello"
}

foo
echo $var
```

N'affichera rien, car var est une variable locale à la fonction foo, et sa valeur n'est pas visible de l'extérieur.

Mélanger les deux ensemble

```
var="hello"

function foo(){
    local var="sup?"
    echo "inside function, var=$var"
}

foo
echo "outside function, var=$var"
```

Va sortir

```
inside function, var=sup?
outside function, var=hello
```

Lire variables globales et locales en ligne: https://riptutorial.com/fr/bash/topic/9256/variables-globales-et-locales

Chapitre 71: Variables internes

Introduction

Un aperçu des variables internes de Bash, où, comment et quand les utiliser.

Examples

Bash variables internes en un coup d'œil

Variable	Détails
`$*` / `$@`	Paramètres de position / script (arguments). Développez comme suit: `$*` et `$@` sont les mêmes que `$1 $2 ...` (notez que cela n'a généralement aucun sens de laisser ceux qui ne sont pas cotés) `"$*"` est identique à `"$1 $2 ..."` [1] `"$@"` est identique à `"$1" "$2" ...` 1. Les arguments sont séparés par le premier caractère de $ IFS, qui ne doit pas nécessairement être un espace.
`$#`	Nombre de paramètres de position transmis au script ou à la fonction
`$!`	ID de processus de la dernière commande (la plus proche des pipelines) dans le dernier travail mis en arrière-plan (notez que ce n'est pas nécessairement la même que l'ID de groupe de processus du travail lorsque le contrôle des travaux est activé)
`$$`	ID du processus qui a exécuté `bash`
`$?`	Etat de sortie de la dernière commande
`$n`	Paramètres de position, où n = 1, 2, 3, ..., 9
`${n}`	Paramètres de position (comme ci-dessus), mais n peut être> 9
`$0`	Dans les scripts, chemin d'accès au script; avec `bash -c 'printf "%s\n" "$0"' name args` : `name` (le premier argument après le script en ligne), sinon, l' `argv[0]` reçu par `bash` .
`$_`	Dernier champ de la dernière commande
`$IFS`	Séparateur de champ interne
`$PATH`	Variable d'environnement PATH utilisée pour rechercher des exécutables
`$OLDPWD`	Répertoire de travail précédent

Variable	Détails
`$PWD`	Répertoire de travail actuel
`$FUNCNAME`	Tableau de noms de fonctions dans la pile des appels d'exécution
`$BASH_SOURCE`	Tableau contenant les chemins source pour les éléments du tableau `FUNCNAME` . Peut être utilisé pour obtenir le chemin du script.
`$BASH_ALIASES`	Tableau associatif contenant tous les alias actuellement définis
`$BASH_REMATCH`	Tableau de correspondances du dernier match de regex
`$BASH_VERSION`	Version de la version Bash
`$BASH_VERSINFO`	Un tableau de 6 éléments avec des informations sur la version de Bash
`$BASH`	Chemin absolu vers le shell Bash en cours d'exécution lui-même (déterminé `bash` par `bash` sur la base de `argv[0]` et la valeur de `$PATH` ; peut être erroné dans les cas de coin)
`$BASH_SUBSHELL`	Niveau Bash subshell
`$UID`	Réel (pas efficace si différent) ID utilisateur du processus exécutant `bash`
`$PS1`	Invite de ligne de commande principale; voir Utilisation des variables PS *
`$PS2`	Invite de ligne de commande secondaire (utilisée pour une entrée supplémentaire)
`$PS3`	Invite de ligne de commande tertiaire (utilisé dans la boucle de sélection)
`$PS4`	Invite de ligne de commande quaternaire (utilisée pour ajouter des informations avec une sortie détaillée)
`$RANDOM`	Un entier pseudo-aléatoire compris entre 0 et 32767
`$REPLY`	Variable utilisée par `read` par défaut lorsqu'aucune variable n'est spécifiée. Également utilisé par `select` pour renvoyer la valeur fournie par l'utilisateur
`$PIPESTATUS`	Variable de tableau contenant les valeurs de statut de sortie de chaque commande dans le pipeline de premier plan le plus récemment exécuté.

L'attribution de variables ne doit pas avoir d'espace avant et après. `a=123` pas `a = 123` . Ce dernier (un signe égal entouré par des espaces) de manière isolée signifie exécuter la commande `a` avec les arguments `=` et `123` , mais il est également visible dans l'opérateur de comparaison de chaîne (qui est syntaxiquement un argument `[` ou `[[` ou tout autre test que vous êtes en utilisant).

$ BASHPID

ID de processus (pid) de l'instance actuelle de Bash. Ce n'est pas la même que la variable `$$` , mais cela donne souvent le même résultat. Ceci est nouveau dans Bash 4 et ne fonctionne pas dans Bash 3.

```
~> $ echo "\$\$ pid = $$  BASHPID = $BASHPID"
$$ pid = 9265  BASHPID = 9265
```

$ BASH_ENV

Une variable d'environnement pointant vers le fichier de démarrage Bash qui est lu lors de l'appel d'un script.

$ BASH_VERSINFO

Un tableau contenant les informations complètes sur la version est divisé en éléments, bien plus pratique que `$BASH_VERSION` si vous recherchez uniquement la version principale:

```
~> $ for ((i=0; i<=5; i++)); do echo "BASH_VERSINFO[$i] = ${BASH_VERSINFO[$i]}"; done
BASH_VERSINFO[0] = 3
BASH_VERSINFO[1] = 2
BASH_VERSINFO[2] = 25
BASH_VERSINFO[3] = 1
BASH_VERSINFO[4] = release
BASH_VERSINFO[5] = x86_64-redhat-linux-gnu
```

$ BASH_VERSION

Affiche la version de bash en cours d'exécution, ce qui vous permet de décider si vous pouvez utiliser des fonctionnalités avancées:

```
~> $ echo $BASH_VERSION
4.1.2(1)-release
```

$ EDITEUR

L'éditeur par défaut qui sera impliqué par tous les scripts ou programmes, généralement vi ou emacs.

```
~> $ echo $EDITOR
vi
```

$ FUNCNAME

Pour obtenir le nom de la fonction en cours - tapez:

```
my_function()
{
    echo "This function is $FUNCNAME"    # This will output "This function is my_function"
}
```

Cette instruction ne renverra rien si vous le tapez en dehors de la fonction:

```
my_function

echo "This function is $FUNCNAME"    # This will output "This function is"
```

$ HOME

Le répertoire de base de l'utilisateur

```
~> $ echo $HOME
/home/user
```

$ HOSTNAME

Le nom d'hôte attribué au système lors du démarrage.

```
~> $ echo $HOSTNAME
mybox.mydomain.com
```

$ HOSTTYPE

Cette variable identifie le matériel, il peut être utile pour déterminer quels binaires exécuter:

```
~> $ echo $HOSTTYPE
x86_64
```

$ GROUPES

Un tableau contenant les nombres de groupes dans lesquels se trouve l'utilisateur:

```
#!/usr/bin/env bash
echo You are assigned to the following groups:
for group in ${GROUPS[@]}; do
   IFS=: read -r name dummy number members < <(getent group $group )
   printf "name: %-10s number: %-15s members: %s\n" "$name" "$number" "$members"
done
```

$ IFS

Contient la chaîne séparateur de champ interne que bash utilise pour diviser les chaînes lors de la boucle , etc. La valeur par défaut est les caractères blancs: `\n` (saut de ligne), `\t` (onglet) et dans l' espace. Changer cela en quelque chose d'autre vous permet de diviser les chaînes en utilisant des caractères différents:

```
IFS=","
INPUTSTR="a,b,c,d"
for field in ${INPUTSTR}; do
    echo $field
```

```
done
```

Le résultat de ce qui précède est:

```
a
b
c
d
```

Remarques:

- Ceci est responsable du phénomène connu sous le nom de fractionnement de mots .

$ LINENO

Affiche le numéro de ligne dans le script en cours. Surtout utile lors du débogage de scripts.

```
#!/bin/bash
# this is line 2
echo something  # this is line 3
echo $LINENO # Will output 4
```

$ MACHTYPE

Semblable à `$HOSTTYPE` ci-dessus, cela inclut également des informations sur le système d'exploitation ainsi que sur le matériel

```
~> $ echo $MACHTYPE
x86_64-redhat-linux-gnu
```

$ OLDPWD

OLDPWD (OLDP Rint **W** ravailler **D** irectory) contient le répertoire avant le dernier `cd` commande:

```
~> $ cd directory
directory> $ echo $OLDPWD
/home/user
```

$ OSTYPE

Renvoie des informations sur le type de système d'exploitation exécuté sur la machine, par exemple.

```
~> $ echo $OSTYPE
linux-gnu
```

$ PATH

Le chemin de recherche pour trouver des binaires pour les commandes. Les exemples courants incluent `/usr/bin` et `/usr/local/bin` .

Lorsqu'un utilisateur ou un script tente d'exécuter une commande, les chemins dans `$PATH` sont recherchés afin de trouver un fichier correspondant avec une autorisation d'exécution.

Les répertoires de `$PATH` sont séparés par un `:` caractère.

```
~> $ echo "$PATH"
/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin
```

Par exemple, si vous tapez `$PATH` ci-dessus, si vous tapez `lss` à l'invite, le shell recherchera `/usr/kerberos/bin/lss` , puis `/usr/local/bin/lss` , puis `/bin/lss` , puis `/usr/bin/lss` , dans cet ordre, avant de conclure qu'il n'ya pas une telle commande.

$ PPID

L'ID de processus (pid) du script ou du parent du shell, c'est-à-dire le processus qui a appelé le script ou le shell en cours.

```
~> $ echo $$
13016
~> $ echo $PPID
13015
```

$ PWD

PWD (P Rint **T** ravail **D** irectory) Le répertoire de travail actuel , vous êtes en ce moment:

```
~> $ echo $PWD
/home/user
~> $ cd directory
directory> $ echo $PWD
/home/user/directory
```

$ SECONDS

Nombre de secondes pendant lesquelles un script a été exécuté. Cela peut devenir assez grand si montré dans le shell:

```
~> $ echo $SECONDS
98834
```

$ SHELLOPTS

Une liste en lecture seule des options bash est fournie au démarrage pour contrôler son comportement:

```
~> $ echo $SHELLOPTS
braceexpand:emacs:hashall:histexpand:history:interactive-comments:monitor
```

$ SHLVL

Lorsque la commande bash est exécutée, un nouveau shell est ouvert. La variable d'environnement $ SHLVL contient le nombre de niveaux de shell exécutés par le shell *actuel* .

Dans une *nouvelle* fenêtre de terminal, l'exécution de la commande suivante produira des résultats différents en fonction de la distribution Linux utilisée.

```
echo $SHLVL
```

En utilisant *Fedora 25* , le résultat est "3". Cela indique que lors de l'ouverture d'un nouveau shell, une commande bash initiale s'exécute et exécute une tâche. La commande initiale bash exécute un processus enfant (une autre commande bash) qui, à son tour, exécute une dernière commande bash pour ouvrir le nouveau shell. Lorsque le nouveau shell s'ouvre, il s'exécute en tant que processus enfant de 2 autres processus shell, d'où la sortie de "3".

Dans l'exemple suivant (étant donné que l'utilisateur exécute Fedora 25), la sortie de $ SHLVL dans un nouveau shell sera définie sur "3". Comme chaque commande bash est exécutée, $ SHLVL s'incrémente d'une unité.

```
~> $ echo $SHLVL
3
~> $ bash
~> $ echo $SHLVL
4
~> $ bash
~> $ echo $SHLVL
5
```

On peut voir que l'exécution de la commande 'bash' (ou l'exécution d'un script bash) ouvre un nouveau shell. En comparaison, la recherche d'un script exécute le code dans le shell actuel.

test1.sh

```
#!/usr/bin/env bash
echo "Hello from test1.sh. My shell level is $SHLVL"
source "test2.sh"
```

test2.sh

```
#!/usr/bin/env bash
echo "Hello from test2.sh. My shell level is $SHLVL"
```

run.sh

```
#!/usr/bin/env bash
echo "Hello from run.sh. My shell level is $SHLVL"
```

```
./test1.sh
```

Exécuter:

```
chmod +x test1.sh && chmod +x run.sh
./run.sh
```

Sortie:

```
Hello from run.sh. My shell level is 4
Hello from test1.sh. My shell level is 5
Hello from test2.sh. My shell level is 5
```

$ UID

Une variable en lecture seule qui stocke le numéro d'identification de l'utilisateur:

```
~> $ echo $UID
12345
```

1 $ 2 $ 3 $ etc ...

Paramètres de position passés au script à partir de la ligne de commande ou d'une fonction:

```
#!/bin/bash
# $n is the n'th positional parameter
echo "$1"
echo "$2"
echo "$3"
```

Le résultat de ce qui précède est:

```
~> $ ./testscript.sh firstarg secondarg thirdarg
firstarg
secondarg
thirdarg
```

Si le nombre d'arguments positionnels est supérieur à neuf, des accolades doivent être utilisées.

```
#  "set -- " sets positional parameters
set -- 1 2 3 4 5 6 7 8 nine ten eleven twelve
# the following line will output 10 not 1 as the value of $1 the digit 1
# will be concatenated with the following 0
echo $10   # outputs 1
echo ${10} # outputs ten
# to show this clearly:
set -- arg{1..12}
echo $10
echo ${10}
```

$

Pour obtenir le nombre d’arguments en ligne de commande ou de paramètres de position, tapez:

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

Lorsqu'il est exécuté avec trois arguments, l'exemple ci-dessus aboutira à la sortie:

```
~> $ ./testscript.sh firstarg secondarg thirdarg
3
```

$ *

Renverra tous les paramètres de position dans une seule chaîne.

testscript.sh:

```
#!/bin/bash
echo "$*"
```

Exécutez le script avec plusieurs arguments:

```
./testscript.sh firstarg secondarg thirdarg
```

Sortie:

```
firstarg secondarg thirdarg
```

$!

L'ID de processus (pid) du dernier travail exécuté en arrière-plan:

```
~> $ ls &
testfile1 testfile2
[1]+  Done                    ls
~> $ echo $!
21715
```

$ _

Affiche le dernier champ de la dernière commande exécutée, utile pour faire passer quelque chose à une autre commande:

```
 ~> $ ls *.sh;echo $_
testscript1.sh  testscript2.sh
testscript2.sh
```

Il donne le chemin du script s'il est utilisé avant toute autre commande:

test.sh:

```
#!/bin/bash
echo "$_"
```

Sortie:

```
~> $ ./test.sh # running test.sh
./test.sh
```

Note: Ce n'est pas une manière infaillible d'obtenir le chemin du script

$?

Le statut de sortie de la dernière fonction ou commande exécutée. Habituellement, 0 signifie que tout autre élément indique un échec:

```
~> $ ls *.blah;echo $?
ls: cannot access *.blah: No such file or directory
2
~> $ ls;echo $?
testfile1 testfile2
0
```

$$

L'ID de processus (pid) du processus en cours:

```
~> $ echo $$
13246
```

$ @

`"$@"` développe en tant que mots séparés pour tous les arguments de la ligne de commande. Il est différent de `"$*"` , qui étend tous les arguments en un seul mot.

`"$@"` est particulièrement utile pour une boucle par des arguments et la manipulation des arguments avec des espaces.

Considérons que nous sommes dans un script que nous avons invoqué avec deux arguments, comme ceci:

```
$ ./script.sh "␣1␣2␣" "␣3␣␣4␣"
```

Les variables `$*` ou `$@` deviendront `$1␣$2` , qui à leur tour se développeront en 1 `1␣2␣3␣4` pour que la boucle ci-dessous:

```
for var in $*; do # same for var in $@; do
    echo \<"$var"\>
done
```

imprimera pour les deux

```
<1>
<2>
<3>
<4>
```

Alors que `"$*"` sera étendu à `"$1␣$2"` qui s'étendra à son tour en `"␣1␣2␣␣␣3␣␣4␣"` et donc la boucle:

```
for var in "$*"; do
    echo \<"$var"\>
done
```

invoquera seulement l' `echo` une fois et imprimera

```
<␣1␣2␣␣␣3␣␣4␣>
```

Et enfin `"$@"` se développera dans `"$1" "$2"` , qui se développera dans `"␣1␣2␣" "␣3␣␣4␣"` et donc la boucle

```
for var in "$@"; do
    echo \<"$var"\>
done
```

imprimera

```
<␣1␣2␣>
<␣3␣␣4␣>
```

préservant ainsi à la fois l'espacement interne dans les arguments et la séparation des arguments. Notez que la construction `for var in "$@"; do ...` est tellement commun et idiomatique que c'est la valeur par défaut pour une boucle for et peut être raccourci `for var; do ...`

$ HISTSIZE

Le nombre maximum de commandes mémorisées:

```
~> $ echo $HISTSIZE
1000
```

$ RANDOM

Chaque fois que ce paramètre est référencé, un entier aléatoire compris entre 0 et 32767 est

généré. Assigner une valeur à cette variable génère le générateur de nombres aléatoires (source).

```
~> $ echo $RANDOM
27119
~> $ echo $RANDOM
1349
```

Lire Variables internes en ligne: https://riptutorial.com/fr/bash/topic/4797/variables-internes

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec Bash	4444, Acey, Ajay Sangale, Alessandro Mascolo, Anil, Ashari, Benjamin W., Blachshma, Bob Bagwill, Bubblepop, Burkhard, Christopher Bottoms, Colin Yang, Community, CraftedCart, Danny, Diego Torres Milano, divyum, dotancohen, fedorqui, Franck Dernoncourt, Functino, Gavyn, glenn jackman, Inanc Gumus, Ingve, intboolstring, J F, Jahid, Jean-Baptiste Yunès, JHS, Jonny Henly, l0b0, lesmana, Marek Skiba, Matt, Matt Clark , mouviciel, Nathan Arthur, niglesias, rap-2-h, Richard Hamilton, Riker, satyanarayan rao, shloosh, Shubham, sjsam, Sundeep, Sylvain Bugat, Trevor Clarke, tripleee, user1336087, William Pursell, WMios, Yuki Inoue, Zaz,
2	Achèvement programmable	Benjamin W., jandob
3	Aliasing	Bostjan, Daniel Käfer, dingalapadum, glenn jackman, intboolstring, janos, JHS, Neui, tripleee, uhelp
4	Arithmétique Bash	Alessandro Mascolo, Ashkan, Gavyn, Jesse Chen, user1336087
5	Bash sur Windows 10	Thomas Champion
6	Chaîne de commandes et d'opérations	jordi, Mateusz Piotrowski, uhelp
7	Changer de coque	depperm, IamaTacos
8	Citant	Benjamin W., binki, Gilles, Jahid, Will
9	Contrôle de l'emploi	Samik
10	Copier (cp)	dingalapadum, Richard Hamilton
11	co-processus	Dunatotatos
12	Correspondance de motif et expressions régulières	Benjamin W., chepner, Chris Rasys, fedorqui, Grisha Levit, Jahid, nautical, RamenChef, suvayu

13	Couper la commande	Richard Hamilton
14	Création de répertoires	Brendan Kelly
15	Déclaration de cas	P.P.
16	Emplois à des moments précis	fifaltra, uhelp
17	Emplois et processus	Amir Rachum, Bubblepop, DonyorM, fifaltra, J F, Jouster500, Mike Metzger, Neui, Riccardo Petraglia, Root, Sameer Srivastava, Sk606, suvayu, u02sgb, WAF, WannaGetHigh, zarak
18	En utilisant chat	anishsane, Bubblepop, Christopher Bottoms, glenn jackman, intboolstring, Jahid, Matt Clark, Rafa Moyano, Root, Samik, Samuel, SLePort, tripleee, vielmetti, xhienne,
19	En utilisant le tri	Flows, Mohima Chaudhuri
20	Espace de noms	meatspace, uhelp
21	Éviter la date en utilisant printf	Benjamin W., chepner, kojiro, RamenChef
22	Expansion Brace	4444, Benjamin W., Jamie Metzger, mnoronha, Peter, uhelp, zarak
23	Expressions conditionnelles	BurnsBA, Gilles, hedgar2017, Jahid, Jonny Henly, mnoronha, RamenChef,
24	Extension des paramètres Bash	Benjamin W., chepner, codeforester, fedorqui, George Vasiliou, Grexis, hedgar2017, J F, Jahid, Jesse Chen, Stephane Chazelas, Sylvain Bugat, uhelp, Will, WMios, ymbirtt
25	Fractionnement de fichiers	Mohima Chaudhuri, Richard Hamilton
26	Fractionnement de mots	Jahid, Jesse Chen, RamenChef, Skynet
27	Gestion de l'invite du système	RamenChef, uhelp
28	Gestion de la variable d'environnement PATH	Jahid, kojiro, RamenChef

29	getopts: analyse intelligente des paramètres positionnels	pepoluan, sjsam
30	Grep	Chandrahas Aroori
31	Ici des documents et ici des cordes	Ajinkya, Benjamin W., Deepak K M, fedorqui, Iain, Jahid, janos, lanoxx, Stobor, uhelp
32	La commande de coupe	Dario
33	La portée	Benjamin W., chepner
34	Le débogage	Gavyn, Leo Ufimtsev, Sk606
35	Les fonctions	BrunoLM, Christopher Bottoms, dimo414, divyum, edi9999, Jahid, janos, Matt Clark, Michael Le Barbier Grünewald, Neui, Reboot, Samik, Sergey, Sylvain Bugat, tripleee,
36	Lire un fichier (flux de données, variable) ligne par ligne (et / ou champ par champ)?	Jahid, vmaroli
37	Liste des fichiers	Benjamin W., cswl, depperm, glenn jackman, Holt Johnson, Iain , intboolstring, J F, Jonny Henly, Markus V., Misa Lazovic, mpromonet, Neui, Osaka, Richard Hamilton, RJHunter, Samik, Sylvain Bugat, teksisto
38	Math	deepmax, Pavel Kazhevets, Tim Rijavec
39	Mise en réseau avec Bash	dhimanta
40	Modèles de conception	mattmc
41	Parallèle	Jon
42	Personnalisation de PS1	Blacksilver, Cows quack, Jahid, Jonny Henly, Josh de Kock, Kamal Soni, Misa Lazovic, tversteeg, Wenzhong
43	Pièges	Cody, Scroff
44	Pipelines	Ashkan, Jahid, liborm, lynxlynxlynx
45	Quand utiliser eval	Alexej Magura

46	Raccourcis clavier	Daniel Käfer, JHS, Judd Rogers, m02ph3u5, Saqib Rokadia
47	Redirection	Alexej Magura, Antoine Bolvy, Archemar, Benjamin W., Brydon Gibson, chaos, David Grayson, Eric Renouf, fedorqui, Gavyn, George Vasiliou, hedgar2017, Jahid, Jon Ericson, Judd Rogers, lesmana, liborm, Matt Clark, miken32, Neui, Pooyan Khosravi, RamenChef, Root, Stephane Chazelas, Sven Schoenung, Sylvain Bugat, Warren Harper, William Pursell, Wolfgang,
48	Répertoires de navigation	Christopher Bottoms, JepZ
49	Script avec des paramètres	Jahid, James Taylor, Kelum Senanayake, Matt Clark, RamenChef
50	Script shebang	DocSalvager, Sylvain Bugat, uhelp
51	Scripts CGI	suleiman
52	Sélectionnez un mot clé	UNagaswamy
53	Séquence d'exécution du fichier	Riker
54	Sortie de script couleur (multiplate-forme)	charneykaye
55	Sourcing	hgiesel, JHS, Matt Clark
56	strace	Chandrahas Aroori
57	Structures de contrôle	Dennis Williamson, DocSalvager, DonyorM, Edgar Rokyan, gzh, Jahid, janos, miken32, Samik, SLePort
58	Substitution de processus	Benjamin W., cb0, Dario, Doctor J, fedorqui, Inian, Martin Lange , Мона_Сах
59	Substitutions d'histoire de Bash	Benjamin W., Bubblepop, Grexis, janos, jimsug, kalimba, Will Barnwell, zarak
60	Tableaux	Alexej Magura, Arronical, Benjamin W., Bubblepop, chepner, Christopher Werby, codeforester, fedorqui, Grisha Levit, Jahid, janos, jerblack, John Kugelman, markjwill, Mateusz Piotrowski, NeilWang, ormaaj, RamenChef, Samik, Sk606, UNagaswamy, Will,
61	Tableaux associatifs	Benjamin W., uhelp, UNagaswamy

62	Transfert de fichier à l'aide de scp	Benjamin W., onur güngör, Pian0_M4n, Rafa Moyano, RamenChef, Reboot, Wojciech Kazior
63	Trouver	Batsu, Daniel Käfer, Echoes_86, fedorqui, GiannakopoulosJ, Inian, Jahid, John Bollinger, Laurel, leftaroundabout, Matt Clark, Michael Gorham, Mohima Chaudhuri, Peter, sjsam, tripleee,
64	true, false et: commandes	Alessandro Mascolo
65	Type de coquille	Jeffrey Lin, liborm, William Pursell
66	URL de décodage	Crazy, I0_ol
67	Utiliser "trap" pour réagir aux signaux et aux événements du système	Benjamin W., Carpetsmoker, dubek, jackhab, Jahid, jerblack, laconbass, Mike S, phs, Roman Piták, Sriharsha Kalluru, suvayu, TomOnTime, uhelp, Will, William Pursell
68	Utilité du sommeil	Ranjit Mane
69	Variables de typage	uhelp
70	variables globales et locales	George Vasiliou, Ocab19
71	Variables internes	Alexej Magura, Ashari, Ashkan, Benjamin W., codeforester, criw , Daniel Käfer, Dario, Dr Beco, fedorqui, Gavyn, Grisha Levit, Jahid, mattmc, Savan Morya, Stephane Chazelas, tripleee, uhelp, William Pursell, Wojciech Kazior