Maîtriser les redirections shell

Écrit par madx, on Jul 16, 2014. Temps de lecture : environ 5 minutes

shell

Le shell (ou ligne de commande) est l'outil de prédilection pour bon nombre d'entre nous qui utilisons un OS de la famille UNIX. Nous sommes persuadés que bien utilisé, il remplace avantageusement un IDE complexe et gourmand.

Chez *Putain de code !*, on a une certaine tendance à utiliser Zsh, mais pour ma part, j'utilise encore et toujours le vénérable Bash, non pas pour faire mon barbu mais simplement car il me suffit amplement.

La suite de l'article est donc basée sur celui-ci mais tout devrait fonctionner à l'identique sur Zsh.

Venons en au sujet même de cet article, l'un des éléments essentiels de l'utilisation du shell : l'utilisation des redirections d'entrée/sortie de base.

Si ça vous paraît barbare, tenez-vous bien, on attaque tout de suite !

Une entrée, deux sorties

Un des principes de base sous UNIX est que tout est fichier et que l'activité du système est rendue par l'interaction de programmes (ou processus) à l'aide des dits fichiers.

Pour pouvoir collaborer avec ses congénères, chaque processus peut accéder par défaut à trois fichiers bien particuliers : l'entrée, la sortie, et la sortie d'erreur standards.

- L'entrée standard (`stdin`)

- La sortie standard (`stdout`)
- La sortie d'erreur (`stderr`)

Habituellement, l'entrée standard est liée au clavier de l'utilisateur *via* son émulateur de terminal, et les deux sorties sont liées à l'affichage dans ce même émulateur.

L'idée pour pouvoir faire collaborer les processus est donc de *brancher* les entrées et sorties de différents programmes afin d'obtenir un résultat.

Les redirections de base

Commençons par quelques redirections de base, celles qu'il faut connaître et qui sont employées systématiquement.

Pour nous exercer, nous allons utiliser principalement deux commandes :

- `echo` : permet d'écrire un message spécifié en tant qu'argument sur sa sortie standard.
- `cat` : permet d'afficher le contenu d'un fichier passé en argument sur sa sortie standard ou de répéter son entrée standard sur sa sortie standard.

Rappelez vous que par défaut, la sortie standard est affichée dans le terminal et l'entrée standard est le clavier de l'utilisateur.

Redirection de sortie : `>`

Ce type de redirection permet d'indiquer à un processus que tout ce qui devrait aller sur la sortie standard (par défaut, le terminal), doit plutôt être stocké dans un fichier.

Pour ça, on utilise la syntaxe `> [fichier]` :

```
$ echo "Hello" > message
$ cat message
Hello
```

Redirection d'entrée : `<`

À l'inverse, on peut aussi spécifier à un programme qu'il doit utiliser un fichier comme son entrée standard, à la place du clavier de l'utilisateur.

En réutilisant notre fichier `message`, on peut par exemple faire :

```
$ cat < message
Hello
```

Notez qu'on utilise bien `cat` *sans argument*, il utilise donc l'entrée standard.

Connecter deux processus : `|`

Dernier des connecteurs de base, l'opérateur `|`, aussi appelé *pipe* (et qui se prononce *paillepe*, avé l'accent).

Il permet tout simplement d'utiliser la sortie d'un programme comme entrée d'un autre.

Pour montrer ça, introduisons la commande `tr`, qui permet de remplacer des caractères dans l'entrée par d'autres.

```
$ echo "Hello!" | tr "[:lower:]" "[:upper:]"
HELLO!
```

Descripteurs de fichiers et redirections avancées

Pour aller plus loin, on va maintenant voir que chaque descripteur de fichier possède son propre identifiant numérique.

Pour `stdin`, `stdout` et `stderr` ce sera respectivement 0, 1 et 2.

On peut se servir de ces identifiants pour faire des redirections plus poussées !

Rediriger les autres descripteur

Par défaut, `>` ne redirige que la sortie standard vers un fichier. Pour rediriger un autre descripteur vers un fichier on emploiera la notation `X>` où *X* prendra la valeur de l'identifiant du descripteur.

Si vous avez bien suivi, on redirigera donc la sortie d'erreur vers un fichier à l'aide de la syntaxe `2>`.

Rediriger vers un autre descripteur : `>&`

On peut connecter les descripteurs d'un processus entre eux. C'est un mécanisme que l'on utilise très souvent quand on écrit des scripts Shell, notamment pour écrire sur la sortie d'erreur

d'erreur.

Pour celà, on utilise la redirection `X>&Y` avec *X* étant l'identifiant descripteur source et *Y* l'identifiant du descripteur sur lequel on veut rediriger.

L'exemple suivant montre comment rediriger la sortie standard sur la sortie d'erreur (c'est comme ça qu'on écrit sur la sortie d'erreur dans un script).

```
$ echo "Error" 1>&2
```

Plus généralement, on peut omettre la partie avant le `>&`, elle prendra la valeur 1 par défaut. On aura donc le script suivant :

```
$ echo "Error" >&2
```

Ajouter en fin de fichier : `>>`

Par défaut, `>` écrase le fichier dans lequel la redirection va s'effectuer. Pour éviter ce problème, on peut utiliser à la place `>>` qui va ajouter en fin de fichier.

```
# Sans >>
$ echo "1" > file
$ echo "2" > file
$ cat file
2
```

```
# Avec >>
$ echo "1" > file
$ echo "2" >> file
$ cat file
1
2
```

`/dev/null`, le trou noir

Il existe un fichier un peu particulier sur tout bon UNIX qui se respecte : `/dev/null`.

Ce dernier est en fait une sorte de trou noir : Si on affiche son contenu on n'obtient rien, et si on écrit quelque chose dedans elle disparait.

On se sert souvent de ce fichier pour supprimer la sortie d'un programme :

```
$ echo "Hello" >/dev/null
$ cat /dev/null
$
```

Ordre des redirections

Un dernier mot avant la fin, l'ordre des redirections est très important !

Imaginez la fonction suivante qui affiche un message sur la sortie standard et un sur la sortie d'erreur :

```
function programme() {
  echo "Message"
  echo "Error" >&2
}
```

Si on veut rediriger toutes les sorties vers `/dev/null` on peut utiliser la syntaxe suivante :

```
$ programme 2>/dev/null >/dev/null
```

Vous remarquerez que c'est assez verbeux, et qu'on pourrait simplifier celà en utilisant `>&` comme vu précédemment.

On essaie donc naïvement :

```
$ programme 2>&1 >/dev/null
Error
```

Hmm, pourtant on a tout branché ensemble non ? En fait non, on a redirigé la sortie sur le fichier pointé par `1` au moment où on écrit la redirection, donc avant qu'il soit redirigé sur `/dev/null`.

Pour bien faire, il faut écrire les redirections dans l'autre ordre :

```
$ programme >/dev/null 2>&1
```

On lira ceci : Rediriger la sortie standard vers `/dev/null`, puis rediriger la sortie d'erreur vers la sortie standard.

Pour résumer

On a en fait que deux redirections de sortie, `>` et `>>`, les symboles qui les entourent sont souvent optionnels.

Si on veut spécifier un fichier on donnera son nom, si on veut spécifier un descripteur on donnera son identifiant préfixé de `&`.

Pour aller plus loin

Le manuel de bash contient une section complète sur les redirections. Elle va beaucoup plus loin que cet article et je vous invite à la lire pour voir tout ce qu'il est possible de faire.

Il y a aussi une section dédiée à ce sujet dans l'Advanced Bash-Scripting Guide.

Écrit par madx

Développeur Ruby et JavaScript, avec un goût prononcé pour la simplicité, le soleil, le pastis. Il n'aime pas OS X car, objectivement, c'est nul. Quand il n'est pas devant son ordi, il y a de grandes chances qu'il compense avec son téléphone. Spécialiste en développement d'applications inutiles.

Matti Schneider • 4 years ago

> j'utilise encore et toujours le vénérable Bash, non pas pour faire mon barbu mais simplement car il me suffit amplement

+1
zsh c'est un shell de hipster :D

1 • Reply • Share ›

xavier nayrac • 4 years ago

Très bonne introduction. C'est simple et clair.
Ça manque peut-être un peu de `ls` par-ci par-là pour les grands débutants, mais je pinaille là ;)

Bash pour moi aussi, au fait, jamais vraiment eu besoin de plus…

• Reply • Share ›

lionelB Admin • 4 years ago

Merci madx, bien intéressant de quoi partir sur de bonnes bases !

• Reply • Share ›

English | Français