Introduction à l'écriture de scripts shell — Partie 2

Gazette Linux n°112 — Mars 2005

Ben Okopnik

Article paru dans le n°112 de la Gazette Linux de mars 2005.
Traduction française par Deny <deny CHEZ monaco POINT net>.
Relecture de la traduction française par Joëlle Cornavin <jcornavi CHEZ club TIRET internet POINT fr>.

Table des matières

Le mois dernier, nous avons jeté un coup d'½il à quelques notions de base concernant la création d'un script shell, ainsi qu'à quelques-uns des mécanismes sous-jacents qui font que tout cela fonctionne. Cette fois-ci, nous allons découvrir comment boucles et exécution conditionnelle permettent de diriger les flux de programme dans les scripts données, et d'acquérir quelques bonnes habitudes en matière d'écriture de scripts.

1. Conventions

La seule chose à noter dans cet article est l'utilisation des points de suspension (...). Je les emploie pour indiquer que le code affiché n'est qu'un fragment, non un script entier en soi. Pour vous aider, imaginez chaque point de suspension comme une ou plusieurs lignes de code qui n'ont pas été réellement écrites.

2. Boucles et exécution conditionnelle

2.1. `for;, do;, done`

On écrit le plus souvent un script pour automatiser une tâche répétitive ; par exemple, si vous devez modifier répétitivement une série de fichiers dans un répertoire donné, vous pourriez avoir un script de ce type :

ou de celui-ci :

Dans ces scripts, le code fait exactement la même chose, mais la première version est beaucoup plus lisible, en particulier si vous construisez de gros scripts avec plusieurs niveaux. Il est souhaitable, lorsqu'on écrit du code, d'indenter chaque niveau (les commandes à l'intérieur de la boucle) ; cela facilite considérablement la relecture et le suivi de votre code.

La structure de contrôle au-dessus est appelée boucle `for` : elle cherche des éléments restants dans une liste (c'est-à-dire « y a-t-il encore des fichiers, en plus de ceux que nous avons déjà lus, qui correspondent au motif *`~/hebdomadaire/*.txt`* ? »). Si le test renvoie `true`, il affecte le nom de l'élément actuel dans la liste à la variable de boucle (`n` ici) et exécute le corps de la boucle (la partie entre `do` et `done`), puis vérifie à nouveau. Une fois que la liste est vide, la boucle `for` arrête de boucler et passe le contrôle à la ligne suivant le mot-clé `done` — dans notre exemple, l'instruction `echo`.

> Si vous voulez que `for` boucle un certain nombre de fois, la syntaxe du script shell peut s'avérer quelque peu fastidieuse :

Très pénible ! Pour itérer par exemple 250 fois, vous devez saisir tout cela au clavier ! Heureusement, il y a un « raccourci », la commande **seq**, qui affiche une séquence de nombres allant de 1 jusqu'au maximum indiqué, par exemple :

En pratique, il fonctionne comme le précédent script. **seq** fait partie du paquetage shellutils de GNU et il est probablement déjà installé sur votre système. Vous pouvez également la possibilité de faire ce genre d'itération avec une boucle `while`, mais c'est un peu plus compliqué.

2.2. `while;, do,; done`

Souvent, nous avons besoin d'un mécanisme de commande agissant d'après une condition spécifiée, plutôt qu'en itérant dans toute une liste. La boucle `while` remplit cette exigence :

Le flux général de ce script est : nous invoquons **pppd**, puis continuons à boucler jusqu'à ce qu'une connexion véritable soit établie (pour utiliser ce script, remplacez *`192.168.0.1`* par l'adresse IP attribuée par votre FAI). Voici les détails :

1. La commande **ping -c 1 xxx.xxx.xxx.xxx** envoie un seul **ping** à l'adresse IP fournie. Notez qu'elle se doit d'être une adresse IP, non une URL. Autrement, **ping** échouera immédiatement en raison de l'absence de DNS. S'il n'y a pas de réponse dans les 10 secondes, il affiche un message comme celui-ci :
2. La seule ligne qui nous intéresse ici est celle qui indique le pourcentage de paquets perdus ; avec un seul paquet, ce ne peut être que 0% (c'est-à-dire un **ping** réussi) ou 100%. En redirigeant la sortie de **ping** par un tube via la commande **grep 100%**, nous nous restreignons à cette ligne, si la perte est en effet de 100%. Une perte de 0% ne produira aucune sortie. Notez que La chaîne `100%` n'a rien de particulier : nous aurions pu utiliser `ret=-1`, `unreachable` ou n'importe quoi d'autre qui est spécifique à une réponse sur erreur.
3. Les crochets qui contiennent l'instruction sont un synonyme de la commande **test**, qui renvoie `0` ou `1` (`true` ou `false`) en fonction de l'évaluation de ce qu'il y a à l'intérieur des crochets. L'opérateur `-n` retourne `true` si la longueur d'une chaîne donnée est supérieure à 0. Puisque la chaîne est supposée être contiguë (pas d'espaces) et que la ligne à vérifier ne l'est pas, nous devons mettre la sortie entre guillemets. C'est une technique que vous utiliserez encore et encore pour l'écriture de scripts. Notez bien qu'il faut un espace autour des crochets. Par exemple, `[-n $STRING]` n'est pas valide, mais `[ -n $STRING ]` est correct. Pour plus d'informations sur les opérateurs utilisés avec **test**, saisissez **help test** ; de nombreuses options très utiles sont disponibles.
4. Tant que le test ci-dessus retourne `true` (par exemple tant que le **ping** échoue), la boucle `while` s'exécute — en affichant la chaîne *`Connexion en cours...`* toutes les dix secondes. Dés qu'un simple **ping** réussit (par exemple, le test retourne `false`), la boucle `while` s'arrête et passe le contrôle à l'instruction après `done`.

2.3. `until; do; done`

La boucle `until` est l'inverse de `while` : elle continue tant que le test est faux, et s'arrête quand il devient vrai. J'ai rarement eu l'occasion de l'employer ; la boucle `while` et la souplesse des tests disponibles suffisent habituellement. Cette construction est juste du « sucre syntaxique » pour ceux qui préfèrent éviter les inversions logiques.

2.4. `if;, then;, [else]; fi`

Nous devons à de nombreuses reprises vérifier l'existence d'une condition et exécuter l'instruction selon le résultat. Pour ces occasions, nous avons l'instruction `if` :

Si une variable appelée `PATRON` a été définie en tant qu'« idiot » (les programmeurs de C prennnent note : `=` et `==` sont équivalents dans une instruction `test` — aucune affectation ne se produit), alors la première instruction `echo` sera exécutée. Dans tous les autres cas, ce sera la seconde instruction `echo` qui s'exécutera (*`if $PATRON ="idiot"`*, vous travaillerez toujours là. Dommage !). Notez que l'instruction `else` est optionnelle, comme dans cet extrait de script :

Ce sous-programme se terminera à l'évidence si la variable `ERREUR` est tout sauf vide, mais cela n'affectera pas autrement le déroulement du programme.

2.5. `case`, `in`, `esac`

L'outil restant que nous pouvons utiliser pour le branchement conditionnel est essentiellement une instruction `if` multiple, basée sur l'évaluation d'un test. Si, par exemple, nous savons que les seules sorties possibles d'un programme imagianire appelé `intel_cpu_test` sont 4, 8, 16, 32, ou 64, alors nous pouvons écrire le code suivant :

Avant que vous m'inondiez de courrier pour savoir comment faire tourner Linux sur un 8088... vous ne pouvez pas le faire, ni sur une calculatrice :).

Évidemment, le `*` est un « fourre-tout » : si quelqu'un, dans les laboratoires secrets d'Intel® fait tourner ce programme sur son nouveau processeur (nom de code « UltraSuperPerfectionné »), il faut que le script renvoie une réponse appropriée plutôt qu'un message d'erreur. Notez les points-virgules doubles — ils « ferment » chacun des ensembles « motif/commande » et sont (pour certaines raisons) une source d'erreur communes dans les *constructs* `case/esac`. Prêtez une attention spéciale aux vôtres !

2.6. `break` et `continue`

Ces instructions interrompent le flux de programme d'une manière particulière. Le `break`, une fois exécuté, sort immédiatement de la boucle enveloppante ; l'instruction `continue` ignore l'itération actuelle de la boucle. Ce comportement est utile dans un certain nombre de situations, en particulier dans les longues boucles où l'existence d'une condition donnée rend tous les tests suivants inutiles. Voici un long (mais je l'espère compréhensible) exemple :

Deux points importants : notez que lors de la vérification de l'état des différentes provisions de la surprise-partie, il aurait été plus judicieux d'écrire plusieurs instructions `if` — les pommes-chips et les bretzels peuvent manquer au même moment (c'est-à-dire qu'ils ne sont pas mutuellement exclusifs). Dans notre exemple, les pommes-chips ont la plus priorité la plus élevée ; si les deux produits viennent à manquer simultanément, il faudra deux itérations de boucle pour les remplacer.

Nous pouvons continuer à vérifier l'état des provsions en tentant de convaincre les policiers que nous tenons en fait une réunion philatélique (en fait, garder les cacahuètes à flot se révèle le facteur crucial à ce stade), mais nous risquons de faire l'impasse sur l'état des boissons — juste à temps pour voir Jacques ronfler sur le canapé.

L'instruction `continue` saute la dernière partie de la boucle `while` tant que la fonction `police_sur_place` retourne `true` ;pour l'essentiel, le corps de la boucle est tronqué à ce point. Notez que même si elle est effectivement à l'intérieur du *construct* `if`, elle affecte la boucle qui l'entoure : `continue` et `break` ne s'appliquent l'une et l'autre qu'aux boucles, c'est-à-dire les *constructs* `for`, `while` et `until`.

3. Retour vers le futur

Voici le script du mois dernier :

Chose intéressante, peu de temps après avoir terminé l'article du mois dernier, je me suis livré à un bout de code en C sur une machine qui n'avait pas de rcs (l'outil Système de contrôle de versions de GNU) installé. Ce script m'a été très utile comme « micro-RCS » : je m'en suis servi pour prendre des

instantanés de l'état du projet. Simples, les scripts généralisés de ce genre deviennent très pratiques à certains moments...

4. Vérification des erreurs

Le script ci-dessus est fonctionnel, pour vous ou quiconque se donne la peine de le lire et de le comprendre. Examinons-le de plus près, cependant : ce que nous demandons à un programme ou à un script est de saisir le nom et qu'il fonctionne, n'est-ce pas ? Cela, ou bien nous indiquer précisément pourquoi il n'a pas fonctionné. Dans ce cas, nous obtenons ce message quelque peu hermétique :

Pour toute autre personne, et pour nous-mêmes, quand nous arrivons à oublier précisément comment utiliser ce script extrêmement complexe, avec d'innombrables options, nous devons introduire un contrôle d'erreurs, en particulier les informations de syntaxe et/ou d'utilisation. Voyons comment appliquer ce que nous venons d'apprendre :

L'opérateur `-z` de `test` retourne '0' (vrai) pour une chaîne de longueur nulle ; nous faisons un test pour savoir si `bkup` est lancé sans nom de fichier. Le tout début est à mon avis le meilleur endroit pour insérer des informations d'aide et/ou d'utilisation. Si vous oubliez quelles sont les options, lancez simplement le script sans options et vous obtiendrez un « cours de rappel » instantané des options disponibles. Vous n'avez même pas à mettre les commentaires originaux, à présent — notez que nous avons incorporé en substance nos commentaires préalables dans les informations d'utilisation. Il est toujours judicieux d'introduire des commentaires à des endroits délicats ou astucieux du script — cette ruse brillante vous évitera des problèmes par la suite.

Avant de faire tourner ce script, dotons-le de quelques fonctions supplémentaires. Et si nous lui permettions de copier différents types de fichiers dans différents répertoires ? Jetons un coup d'½il, d'après ce que nous avons appris :

Voici le résumé des changements :

1. La section « Commentaires » de l'aide mentionne maintenant « ...arborescence du répertoire » plutôt que simplement « répertoire », indiquant le changement que nous avons effectué.
2. La ligne « Usage » a été allongée pour représenter l'argument optionnel (comme l'indiquent les crochets) ; nous avons également ajouté une explication sur la manière d'utiliser cet argument, puisqu'il n'est peut-être pas évidente pour tout le monde.
3. Un *construct* `if` a été jouté, qui vérifie si `'$2'` (un second argument à « bkup ») existe ; si tel est le cas, il vérifie l'existence d'un répertoire du nom spécifié sous `~/Backup` et en crée un s'il n'y en a pas (l'opérateur `-d` teste si un fichier existe et qu'il s'agit d'un répertoire).
4. La commande **cp** a dorénavant une variable `sous-répertoire` insérée entre `~/Backup` et `$1`.

Désormais vous pouvez saisir des commandes comme :

et trier le tout dans les catégories que vous souhaitez. De plus, l'ancien comportement de « bkup » est toujours disponible.

bkup fichier.xyz

enverra une sauvegarde de « fichier.xyz » dans le répertoire `~/Backup` lui-même ; c'est pratique pour les fichiers inclassables.

D'ailleurs, pourquoi ajoutons-nous un `/` à `$2` dans la boucle `if` qui précéde la ligne **cp** ? Eh bien, si `$2` n'existe pas, alors nous voulons que `bkup` agisse comme il le faisait à l'origine, c'est-à-dire copie le fichier dans le répertoire `Backup`. Si nous écrivons ceci :

Notez le `/` supplémentaire entre `$sous-répertoire` et `$1`) et, comme `$2` n'est pas spécifié, alors `$sous-répertoire` devient vierge et la ligne au-dessus devient :

Ce n'est pas forcément gênant, mais nous voulons respecter la pratique syntaxique standard du shell dans la mesure du possible (puisque les particularités du shell, telles que le double `/` ignoré, ne seront pas nécessairement préservées).

En fait, c'est vraiment une bonne idée d'examiner toutes les possibilités chaque fois que construisez des variables dans une chaîne ; une erreur classique de ce genre est présentée dans le programme suivant :

NE PAS UTILISER CE SCRIPT !

NE PAS UTILISER CE PROGRAMME !

Bien, au moins ils l'ont commenté !

Que se passe-t-il si on lance ce script sans saisir de paramètre ? La ligne active dans le script devient :

En supposant que vous êtes l'utilisateur Joseph dans votre répertoire personnel, le résultat est plutôt désastreux : il supprime la totalité de vos fichiers personnels. Cela devient catastrophique si vous êtes l'utilisateur *root* dans le répertoire *root* — le système entier est inutilisable !

Les virus ressemblent parfois à ce genre de programme convivial et inoffensif...

Soyez prudent lorsque vous écrivez des scripts. Comme vous venez de le voir, vous avez le pouvoir de détruire votre système entier en un clin d'½il.

Quand vous êtes connecté en tant que *root*, ne lancez aucun script shell dont vous n'êtes pas certain de l'innocuité.

5. Récapitulation

Les boucles et l'exécution conditionnelle sont une partie très importante de la plupart des scripts. Au fur et à mesure que nous analyserons d'autres scripts shell dans des articles à venir, vous verrez quelques-unes des innombrables manières dont on peut les utiliser — un script d'une complexité même moyenne ne peut exister sans cela.

Le mois prochain, nous jetterons un coup d'½il à quelques outils couramment utilisés dans les scripts shell — outils qui peuvent vous être familiers en tant qu'outils en ligne de commande — et nous découvrirons comment les combiner pour produire les résultats souhaités. Nous disséquerons aussi deux scripts de mon cru, à moins que quelqu'un d'autre soit assez courageux pour envoyer le résultats de ses réflexions.

Tous commentaires et corrections concernant cette série d'articles sont les bienvenus, ainsi que l'envoi de tout script intéressant.

Au mois prochain — Bon Linux !

6. Références

Les pages de manuel de bash, builtins, sed, mutt.

Ben est l’éditeur en chef de la Gazette Linux et est membre de l’« Answer Gang ».

Ben est né à Moscou en Russie en 1962. Il a commencé à s’intéresser à l’électricité dès l’age de 6 ans en enfonçant une fourchette dans une prise et déclenchant ainsi un incendie, depuis il n’a jamais cessé de s’intéresser à la technologie. Il travaille avec les ordinateurs depuis les Temps Anciens où on devait souder soi-même les composants sur les cartes à circuits imprimés et où les programmes devaient tenir dans 4 Ko de mémoire. Il serait heureux de payer tout psychologue capable de le guérir des cauchemars récurrents qu’il a gardés de cette époque.

Ses expériences suivantes comprennent la création de programmes dans pratiquement une douzaine de langages, la maintenance de réseaux et de bases de données pendant l’approche d’un ouragan et l’écriture d’articles pour des publications allant des magazines de voile aux journaux technologiques. Après une croisière de 7 ans dans l’Atlantique et les Caraïbes et des passages sur la côte Est des Etats-Unis, il a désormais posé l’ancre à St. Augustine en Floride. Il est instructeur technique chez Sun Microsystems et travaille également à titre privé comme consultant Open Source et développeur Web. Ses passe-temps actuels incluent notamment l’aviation, le yoga, les arts martiaux, la moto, l’écriture et l’histoire romaine. Son Palm Pilot est saturé d’alarmes dont la plupart contiennent des points d’exclamation.

Il travaille avec Linux depuis 1997 et lui doit d’avoir perdu tout intérêt sur les retombées d’une guerre nucléaire dans le Nord-est du Pacifique.