BASH, LES BONNES PRATIQUES

Romain Pelisse [Sustain Engineer @Red Hat]

Relecture Aurélie Garnier

À l'heure du DevOps et de l'« infrastructure as code », la conception de scripts Bash est loin d'avoir disparu, comme certains le pensaient il y a quelques années. Avec des besoins d'automatisation de plus en plus présents, mais aussi de plus en plus souvent placés entre les mains de développeurs plutôt que d'administrateurs système, de nombreux scripts sont produits, et jouent des rôles toujours plus critiques dans la maintenance des systèmes. Mais, paradoxalement, si ces développeurs sont souvent des experts sur leur langage de programmation, qu'il s'agisse de Java, C# ou autre Python, ils semblent souvent comme frappés d'amnésie quand ils implémentent leurs scripts Bash ! En fait, si la syntaxe de ce dernier est très permissive, et ses mécanismes d'interprétation très différents d'un langage de programmation dit « traditionnel », il n'en offre pas moins de nombreuses techniques et pratiques pour en assurer une exécution propre, saine et fiable – pourtant peu utilisées. Démonstration.

Mots-clés : Bash, Shell, DevOps, Cloud, Programmation, Linux

Résumé

Cet article couvre un ensemble de bonnes pratiques et de techniques à utiliser lors de la conception de scripts Bash, pour leur assurer une exécution fiable et robuste.

Les scripts shell (Bash ou autres d'ailleurs) sont partout. On en trouve un peu à toutes les sauces, et dans l'ensemble, ils ne sont que rarement irréprochables, même du point de vue strictement technique. Cependant, ils font la plupart du temps le travail qu'on leur demande, et, s'ils n'y parviennent pas, on se contente généralement de les exécuter une nouvelle fois plutôt que de chercher à les rendre plus « imperméables » aux erreurs.

Leur manque de robustesse, de propreté ou même de certaines fonctionnalités n'est jamais réellement un problème, car ils ne sont que rarement exécutés. Généralement uniquement au démarrage d'une machine ou d'un serveur, ou même parfois encore plus rarement – par exemple lors d'une installation.

Dans ce genre de contexte, s'ils ne s'exécutent pas correctement, ils sont simplement relancés et/ou rapidement corrigés à la main. On y perd un peu de temps, mais étant donné leur relative faible fréquence d'utilisation, peu de personnes reviennent dessus.

Avec l'émergence des environnements « cloud » et l'automatisation des déploiements de systèmes comme d'applications, ces mêmes scripts d'installation qui étaient exécutés « une fois de temps en temps » ou simplement qui « plantent parfois sans explication », vont désormais être exécutés presque tout le temps - lors du démarrage d'une nouvelle instance ou de la construction d'un environnement de tests par exemple.

Bref, dans ce contexte, ce comportement aléatoire va devenir une source de souffrance constante. Il est donc essentiel de désormais concevoir ces scripts de manière robuste, propre et structurée, pour leur permettre de supporter plus aisément aléas et cas limites.

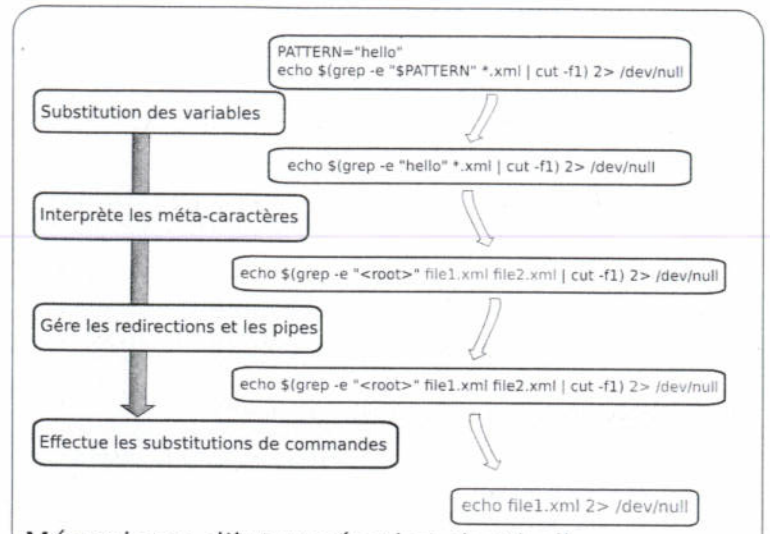
La suite de cet article va donc décrire un ensemble de bonnes pratiques et de conseils techniques visant à structurer le mieux possible nos scripts shell. Ce premier article se concentrera sur l'organisation des scripts et l'utilisation des fonctions. Un autre article sera dédié à des techniques de programmation.

1 En-tête du script

Tout script doit commencer par une ligne de commentaire indiquant au système d'exploitation quel interpréteur ce dernier doit utiliser pour l'exécuter. Ouvrons, à titre d'exemple, un script issu du répertoire `/etc/init.d/` :

Le mécanisme d'interprétation du Bash

Pour être sûr de bien comprendre l'exécution des nombreux extraits de scripts illustrant cet article, une petite piqûre de rappel – sous forme graphique, sur la manière dont le Bash interprète les lignes de commandes qui lui sont passées :



Mécanisme d'interprétation du Shell

```
$ head -3 /etc/init.d/ntpd
#!/bin/bash
#
# ntpd          This shell script takes care of starting and stopping
```

Lors de l'exécution de ce script, le système d'exploitation sait donc quel interpréteur invoquer (en l'occurrence `bash`). Notez aussi que cette pratique ne se limite bien évidemment pas au simple script Shell, puisqu'on la retrouve pour les scripts Ruby ou Python.

Attention, c'est une bonne pratique, mais comme toutes les pratiques, elle n'est en aucun cas obligatoire, et un script shell sans cet en-tête peut être exécuté sans erreur.

Ceci reste néanmoins très pertinent, car si, sur la plupart des systèmes, le raccourci `/bin/sh` pointe vers `/bin/bash`, comme on le voit ci-dessous, il est bien plus prudent d'utiliser directement `/bin/bash` et d'expliciter ainsi de manière très claire de quel interpréteur - et de quelles fonctionnalités associées, votre script a besoin pour s'exécuter sans erreur.

```
$ ls -l /bin/sh
lrwxrwxrwx. 1 root root 4 Aug 21 14:22 /bin/sh -> bash
```

En effet, en cas d'absence de bash sur le système, notre script ne sera pas exécuté, alors que si le raccourci pointe vers un autre interpréteur, il risque de s'exécuter partiellement ou, encore pire, sans problème - sauf dans certains cas limites, qui (loi de Murphy **[1]** oblige) se feront un plaisir de surgir au pire des moments.

Bref, ne jouons pas avec le feu, et prenons toujours soin de spécifier exactement l'interpréteur à utiliser. Cette information servira autant au système qui l'exécute, qu'au mainteneur du script, qui saura ainsi quelles fonctionnalités utiliser (lui permettant de savoir s'il peut, par exemple, utiliser ou non les tableaux associatifs introduits, depuis la version 4, dans la syntaxe du Bash **[2]**).

2 Fonction en Bash

Si un script shell est par nature un code procédural **[3]**, il n'en reste pas moins que l'on peut factoriser son code à l'aide de **fonctions**, comme tout autre langage de programmation. Ces dernières ne sont malheureusement que rarement utilisées par de nombreux concepteurs de scripts, qui se contentent généralement d'un « bon vieux » copier/coller de derrière les fagots, aboutissant rapidement à une kyrielle de scripts au contenu plus ou moins similaire...

Ce qu'il est intéressant de noter, c'est que ce genre de pratiques est fortement prohibé dans les autres domaines de la programmation. Néanmoins, cela ne semble poser aucun problème, par exemple, à un développeur Java chevronné, qui traque la moindre ligne de copier/coller dans son code à l'aide de CPD **[4]**, de créer trois scripts distincts - tous sur la même base, sans chercher à en factoriser quoi que ce soit...

Sans continuer plus longtemps sur les dangers liés au copier/coller de code (bug à corriger dans plusieurs endroits, code plus long à lire, etc.), qui ne sont plus à démontrer, on peut noter qu'en se privant de fonctions, le concepteur d'un script se prive non seulement d'une facilité certaine de réutilisation, mais aussi d'un mécanisme d'encapsulation fort pratique.

En effet, il est possible de déclarer au sein d'une fonction des variables qui lui seront locales. C'est non seulement très pratique, et améliore la lisibilité, mais cela permet aussi de réduire le nombre de variables utilisées et déclarées lors de l'exécution du script, évitant, par exemple, de réutiliser par inadvertance une variable dans une autre partie du script.

Par ailleurs, il est assez élégant, pour accroître la lisibilité et la propreté du script, d'utiliser ces préfixes pour renommer les arguments d'entrée de la fonction :

```
my_function() {
    local first_arg="${1}"
    local second_arg="${2}"
}
```

Vous noterez aussi, au passage, la syntaxe utilisée pour déclarer une fonction. En effet, bash en reconnaît plusieurs, dont la suivante, plus « verbeuse » :

```
function my_function() {
    local first_arg="${1}"
    local second_arg="${2}"
}
```

Cette dernière permet d'omettre les parenthèses lors de la déclaration de la fonction :

```
function my_function {
    local first_arg="${1}"
    local second_arg="${2}"
}
```

Néanmoins, la première syntaxe est à la fois plus expressive - la présence des parenthèses désignant clairement le nom qui les précède comme un nom de fonction, mais aussi plus succincte, et nous la privilégierons donc pour la suite.

Un dernier point sur les fonctions, avant de passer à la suite. Bien qu'il soit possible de déclarer une fonction à peu près n'importe où dans un script, il est de loin préférable de les regrouper au début du fichier, juste après les imports de fichiers annexes (voir plus loin) :

```
#!/bin/bash

./lib/commons.sh

my_function() {
    local first_arg="${1}"
    local second_arg="${2}"
}

...
# début du script en tant que tel
```

2.1 Variables internes dans les fonctions

Il est souvent méconnu que Bash dispose de nombreuses variables internes **[5]** qui sont fort pratiques dans la conception des scripts. Spécialement dans le cas des fonctions, car elles permettent, entre autres, de récupérer le nom de la fonction en cours d'exécution :

```
complex_fonction() {
    # ...
    local status="${?}"
    if [ ${status} -ne 0 ] ; then
        log "a command failed in $FUNCNAME with status
${status}"
    fi
    #...
}
```

3 Fonction usage()

La toute première chose à écrire dans un script shell est la fonction `usage()` qui décrira comment utiliser le script en question. C'est élémentaire, mais, dans les faits, si vous regardez la plupart des scripts shell qui vous sont livrés avec votre dernière distribution Linux (par exemple) vous constaterez, à votre grande surprise, que même les « grands noms » de notre industrie ne se tiennent que rarement à cette convention.

Loin de se limiter à constituer une bonne pratique, cette fonction va tout d'abord nous permettre de réfléchir, avant de rentrer tête baissée dans la conception, à l'interface que nous souhaitons exposer pour notre script. Quand nous parlons ici d'interface, il ne s'agit pas d'interface homme/machine, mais plutôt de l'API, en quelque sorte, que notre script va offrir.

Ce point est d'autant plus crucial que nos scripts, devenant plus robustes et plus utiles, vont rapidement se multiplier et, surtout, s'appeler mutuellement les uns les autres.

```
usage() {
  echo "$(basename ${0}) [-v] -n <name-of-env>
  echo ""
  echo "-n name of the environment"
  echo "-v verbose mode"
}
```

Lorsque votre script est invoqué sans aucun argument, il devrait, par défaut, appeler la fonction `usage()` et s'arrêter avec un statut « zéro ». Ce comportement est très appréciable car, tout d'abord, si, lors de l'invocation manuelle du script, une erreur de saisie aboutit à l'exécution du script sans argument, on est sûr de ne pas exécuter tout de même le script. C'est peu de chose, mais c'est déjà appréciable, car sur un système en production, ceci peut avoir de graves conséquences.

Ensuite, quand vous n'avez pas exécuté un script depuis longtemps, vous aurez vraisemblablement oublié quel paramètre lui passer et comment le valoriser. Il suffit alors d'exécuter le script sans argument pour obtenir la syntaxe appropriée. Si vous ajoutez systématiquement ce comportement à tous vos scripts, vous pourrez ainsi aisément obtenir de l'aide, sans prendre le risque de déclencher de manière involontaire une quelconque opération.

Cette pratique est nettement plus efficace et utile que la rédaction de documentation technique ou de guides d'utilisation ou d'exploitation, que beaucoup d'entreprises ou d'organisations semblent tant apprécier. En effet, ces documents, vivant dans un espace généralement complètement différent de celui des scripts qu'ils décrivent, ont tendance très très rapidement à se désynchroniser de l'évolution naturelle du script, et à se retrouver complètement obsolètes ou, pire, erronés.

À l'inverse, la fonction `usage()`, quant à elle, sera plus aisément mise à jour, vraisemblablement au moment même où le mainteneur du script optera pour ajouter ou modifier des arguments. En outre, les utilisateurs du script préféreront se référer plutôt à cette aide, directement accessible depuis la console dans laquelle ils travaillent qu'à un document, qu'il faudra d'abord localiser, avant de le télécharger pour, enfin, l'ouvrir avec un autre logiciel...

Tous ces détails peuvent sembler triviaux, mais vous verrez que, dans la pratique, on apprécie vite non seulement le confort qu'ils apportent, mais surtout la structure qu'ils confèrent. Nul besoin d'un document de plus de dix pages pour s'assurer qu'un processus est bien documenté si l'ensemble des scripts dispose d'une méthode `usage()`...

4 Gérer des librairies de fonctions

Comme évoqué dans la section précédente, le bash permet, comme tout autre langage de script, d'importer des fonctions, et même des variables situées dans un autre fichier. Cette capacité est fort pratique, et pourtant, malheureusement, fort peu utilisée.

4.1 La fonction unify_args()

Comme avec tout autre de langage de programmation, nous développons aussi en Bash des fonctions réutilisables qui peuvent donc être réutilisées dans l'ensemble de nos

scripts. Prenons l'exemple d'une fonction conçue par un collègue, il y a quelques années, que j'ai souvent réutilisée, `unify_args()` :

```
unify_args () {
    local tmp=$(mktemp)

    for i in "$@"; do
            echo "$i" >>"${tmp}"
    done

    RC=$(cat "${tmp}" | sort | uniq)
    clean "${tmp}"
}
```

Comme son nom l'indique, cette fonction permet d'unifier une série d'arguments en une seule chaîne. C'est un traitement relativement simple, mais que vous aurez probablement à réutiliser dans de nombreux endroits. La logique qu'elle encapsule n'est pas très élaborée, mais en la plaçant dans une fonction, on peut non seulement la réutiliser facilement, mais aussi réduire la taille du code là où elle est invoquée.

En outre, cette implémentation se charge aussi du « nettoyage », en effaçant le fichier temporaire qu'elle utilise. Cette plomberie n'ayant pas grand intérêt en tant que telle, il est appréciable de la voir « dissimulée » dans notre fonction.

On peut remarquer aussi que l'implémentation de la fonction présentée ici n'est pas très optimale. Elle utilise un fichier temporaire - soit une entrée/sortie, ce qui est peu performant. Mais l'avantage d'avoir regroupé ce code dans une fonction est que, si jamais quelqu'un venait à réimplémenter la fonction sous une forme plus efficace, l'ensemble de scripts l'utilisant bénéficierait immédiatement du gain de performances.

4.2 La fonction remote_open()

Étudions maintenant une autre fonction, `remote_open`, qui permet d'établir une connexion SSH vers un hôte distant :

```
remote_open () {
    if [ -z "${IDENTITY}" -o -z "${HOST}" ]; then
            log "remote_open needs IDENTITY (${IDENTITY}) and
HOST (${HOST}) to be set"
        exit 1
    fi
   log "Setting up shared ssh connection"
    if [ -z "${DRYRUN}" ]; then
            if [ ! -z "${CONTROLDIR}" -o ! -z "${CONTROLOPTS}"
]; then
            log "SSH MASTER CONNECTION ALREADY UP: controldir:
'${CONTROLDIR}' \
              controlopts: '${CONTROLOPTS}'"
                return
            fi

            log "ssh [...] -C -i ${IDENTITY} root@${HOST} $@"
            CONTROLDIR="$(mktemp -d)"
        ssh ${SSH_OPTIONS} -o ControlMaster=auto -o
"ControlPath=${CONTROLDIR}/%r-%h:%p"
            \-i "${IDENTITY}" "root@${HOST}" -Nf
            rc="$?"

            CONTROLOPTS="-o ControlMaster=auto -o
ControlPath=${CONTROLDIR}/%r-%h:%p"
            log "SSH MASTER CONNECTION UP: controldir:
'${CONTROLDIR}' controlopts: '${CONTROLOPTS}'"

            # are /not/ readonly by intend, otherwise we can not
unset them
            export CONTROLOPTS
            export CONTROLDIR

            return "$rc"
    else
        log "DRYRUN: ssh -q -o ControlMaster=auto -o
ControlPath=<tmp>/%r-%h:%p \
            -o ConnectTimeout=10 -o UserKnownHostsFile=/dev/null
-o StrictHostKeyChecking=no -C -i ${IDENTITY} root@${HOST} $@"
    fi
}
```

Si vous étudiez attentivement cette fonction, vous réaliserez qu'elle est assez longue, essentiellement pour deux raisons. Tout d'abord, à l'aide d'une variable globale, on peut la rendre extrêmement verbeuse, ce qui est très précieux pour analyser tout problème lors de l'exécution des scripts. En outre, elle prend aussi en charge la mise en place d'une connexion SSH « maître » [6] qui pourra être réutilisée par les commandes ultérieures.

Ainsi, si un script exécute plusieurs commandes sur un hôte distant, il va réutiliser la même connexion SSH, plutôt que d'en créer une nouvelle. Ceci va grandement améliorer les performances, en supprimant de multiples - et inutiles, établissements de connexion. Ce travail, qui fut assez pénible et un peu laborieux à réaliser, il faut le reconnaître, va donc profiter à l'ensemble des scripts et leur apporter un gain de performances réellement non négligeable.

Comme évoqué dans la précédente section, l'utilisation de fonctions a donc permis non seulement de clarifier le code, mais aussi de rendre nos scripts plus efficaces.

5 Import des librairies

Comment importer un script ? De façon évidente, importer un script à l'aide d'un chemin absolu est lié très étroitement à la manière dont il est installé sur le système, mais, à l'inverse, utiliser un chemin relatif, bien que plus portable pose aussi d'autres complications, si le script n'est pas invoqué dans le contexte attendu.

Il faut donc faire un choix sagement mûri entre l'utilisation de chemins relatifs ou absolus lors de l'import des librairies de fonctions. Soit vous optez pour utiliser un chemin absolu, il est fortement recommandé de réaliser un packaging de vos scripts sous forme de RPM, de paquet Debian, ou autre, de manière à garantir le chemin, lors de l'installation des scripts.

Soit vous optez pour l'utilisation d'un chemin relatif, qui sera plus souple, mais vous contraindra néanmoins à toujours exécuter les scripts depuis le même répertoire. L'extrait de code ci-dessous utilise ainsi un chemin relatif pour l'import, mais impose donc à l'utilisateur de ce dernier de l'exécuter depuis la racine du répertoire abritant les scripts dont il dépend :

```
#!/bin/sh

. "$(dirname "${0}")/common/base"
```

Conclusion : patron d'un script complet

Faisons un premier bilan des différentes bonnes pratiques évoquées dans cet article, sous la forme d'un script d'exemple qui en fera une première synthèse. Supposons, à titre d'exemple, que nous souhaitions réaliser un script chargé de détruire un environnement complet déployé Amazon.

Par environnement « complet », on entend ici un ensemble d'instances, sur lesquelles sont déployés nos applicatifs, associées aussi à un *bucket S3* et à une *elastic IP*, qui permettent de rendre le système accessible depuis le monde extérieur.

Ainsi, notre script doit s'occuper non seulement de détruire les instances, mais aussi de nettoyer le *bucket*, et enfin de libérer l'*elastic IP*.

```
#! /bin/sh

. "$(dirname $0)/common/ec2"

usage () {
    echo "${BASENAME} [-h] -B <bucketname> <instance-id>
[<instance-id>...]"
        log "- a script to terminate all provided instances and
the delete
        log "the associated bucket."
}

delete_instances () {
...
}

empty_bucket () {
...
}

free_ip() {
...
}
```

Comme décrit en amont, notre script commence par indiquer l'interpréteur à utiliser (dans notre cas `/bin/sh`, car on n'utilise ici aucune fonctionnalité spécifique à bash), suivi de l'import des librairies nécessaires (ici, la librairie `ec2`), puis la fonction `usage` qui documente enfin l'utilisation du script.

À la suite de la fonction `usage`, on trouve les différentes fonctions spécifiques à ce script. Par souci de concision, leur code n'est pas détaillé ici. On pourra noter que vraisemblablement ces fonctions utiliseront les primitives fournies par la librairie `ec2`.

La plupart des pratiques ou considérations évoquées tout au long de ce premier article tiennent plus de la convention ou de l'organisation, et ne sont pas, en tant que telles, spécifiques au Bash ou à d'autres langages de programmation, néanmoins, elles forment, de notre analyse, un prérequis nécessaire à la conception de scripts robustes, fiables et faciles à maintenir.

La suite au prochain article, c'est-à-dire dans les pages suivantes... ■

Références

[1] La Loi de Murphy : http://en.wikipedia.org/wiki/Murphy's_law

[2] Tableau associatif en Bash : http://www.linuxjournal.com/content/bash-associative-arrays

[3] Programmation procédurale : http://fr.wikipedia.org/wiki/Programmation_proc%C3%A9durale

[4] CPD : http://pmd.sourceforge.net/cpd.html

[5] Variables internes du Bash : http://tldp.org/LDP/abs/html/internalvariables.html

[6] Connexion SSH partagée : https://puppetlabs.com/blog/speed-up-ssh-by-reusing-connections

BASH : ALLER ENCORE PLUS LOIN AVEC LES BONNES PRATIQUES

Romain Pelisse [Sustain Engineer @Red Hat]

Dans le précédent article, nous avons déjà présenté un ensemble de conventions et d'astuces permettant d'améliorer grandement la lisibilité et la maintenance de scripts Bash. Dans ce second article, nous allons aller encore plus loin et discuter de nombreux aspects de l'interpréteur, et de la syntaxe qui lui est associée, qui assureront la robustesse de vos scripts.

Mots-clés : Bash, Shell, DevOps, Cloud, Programmation, Linux

Résumé

Cet article couvre un ensemble de techniques et de mécanismes spécifiques à Bash qui, utilisés systématiquement, apportent beaucoup de robustesse et de fiabilité à l'exécution des scripts.

Sans surprise, le premier point que nous allons évoquer dans cet article concerne la plus primitive des primitives de la programmation : les variables. Nous étudierons ensuite un ensemble de techniques permettant de gérer proprement les erreurs lors de l'exécution du script, avant de finir par une section « rubrique à brac », contenant une série d'astuces et de techniques en tout genre.

1 Protéger ses variables

1.1 Overquoting...

Vous en avez probablement déjà fait l'expérience, les scripts shell sont relativement fragiles. Il suffit parfois de passer un mauvais argument pour obtenir un comportement totalement différent de celui qu'on attendait. Un exemple rapide à l'aide de la commande `cp` :

```
function copy() {
    local src=${1}
    local dest=${2}

    cp ${src} ${dest}
}

$ copy a b
$ copy a "b d" # problème...
cp: target `d' is not a directory
```

À l'exécution de la dernière commande, une erreur se produit, car `cp` tente de copier le fichier `a` et le fichier `b` dans un répertoire `d` qui n'existe pas. Le comportement souhaité était en fait la copie du fichier `a` dans le fichier « `b d` ».

Cet exemple, profondément inutile, a pour seul mérite de bien illustrer le problème. Il est très aisé de se retrouver avec un espace ou un caractère spécial, inattendu, au sein d'une variable. C'est pour ceci qu'il est fondamental de bien protéger ses variables, afin de se prémunir de ce genre de dysfonctionnement.

Pour ce faire, vous avez déjà dû noter que nous privilégions sciemment l'utilisation d'accolades autour du nom des variables. C'est déjà une bonne pratique en soi, mais ce n'est pas suffisant.

Il est aussi important de systématiquement - sauf si c'est contraire au comportement désiré, utiliser des guillemets pour encadrer les variables. On s'assure ainsi que lors de l'interprétation de la variable, et donc de son remplacement par sa valeur, l'intégralité du contenu de cette dernière sera interprété ensuite correctement par le shell :

```
function copy() {
    local src="${1}"
    local dest="${2}"

    cp "${src}" "${dest}"
}
```

Cette pratique est qualifiée par l'un de mes collègues, qui en est pourtant un ardent défenseur, d' *overquoting*. Elle peut sembler fastidieuse pour un gain assez réduit, mais, dans les faits, elle vous prémunira de nombreuses erreurs lors de l'exécution, et, surtout, vous permettra d'obtenir des messages d'erreur beaucoup plus explicites et cohérents.

1.2 Utiliser des variables globales en lecture seule

Les variables globales sont un mécanisme généralement fort critiqué. Pour s'en convaincre, il suffit de se rappeler qu'un des objectifs avoués de la programmation orientée objet était de rendre ce genre de variables inaccessibles depuis l'extérieur en les plaçant dans des classes, et de n'autoriser que les seules fonctions qui nécessitent d'y accéder à les manipuler.

Leur usage est beaucoup plus admis dans la conception de scripts et, dans ce contexte, il n'est pas non plus facile de pouvoir entièrement s'en passer. Néanmoins, il faut s'assurer qu'elles ne soient pas détournées de leur usage initial, ou pire encore, surchargées à notre insu lors de l'import d'autres scripts, par exemple.

Heureusement, le Bash dispose de mots-clés permettant de s'en assurer. Ainsi, il est donc prudent de déclarer les variables globales qui ne sont pas destinées à être modifiées, ce qui est le cas de la plupart de ces dernières, comme étant en lecture seule :

```
$ readonly var="true"
$ var="false"
bash: var: readonly variable
```

Ainsi, dès la première modification maladroite ou involontaire, vous serez notifié très clairement de l'existence d'un problème ou d'un conflit. Sans l'utilisation de ce mot-clé, votre variable globale verra sa valeur modifiée, ce qui entraînera vraisemblablement, en amont du script, d'autres problèmes difficilement explicables (« Mais pourquoi le répertoire créé porte-t-il le nom de l'utilisateur exécutant le script ??? »).

1.3 Tester la nullité des variables et des arguments

Au sein d'un script, il est assez courant, à la suite d'une erreur d'exécution, de se retrouver avec une variable non définie - alors que le script attend d'elle d'avoir été associé à une valeur. Ce genre de problème est notamment très fréquent lors de l'utilisation de substitution de commandes :

```
$ files_to_edit=$(ls -l "${folder}")
# si 'folder n'a pas été défini, la liste contient le contenu du
répertoire courant !
```

En conséquence, il est prudent de toujours veiller à tester une variable, pour vérifier si elle est définie, avant de s'en servir. Cette remarque s'applique bien évidemment encore plus aux arguments des fonctions.

À titre d'illustration, reprenons notre exemple, vide de sens, mais très didactique, de la fonction `copy` :

```
copy() {
    local src="${1}"
    local dest="${2}"

    if [ -z ${src} ] ; then
        echo "le premier argument, le fichier source n'a pas été
fourni".
        exit 1
    fi

    if [ -z ${dest} ] ; then
        echo "le second argument, le répertoire ou fichier de
destination,
            n'a pas été fourni."
        exit 2
    fi

    cp "${src}" "${dest}"
}
```

Syntaxe des substitutions de commande

Une remarque importe sur les substitutions de commandes : il faut souligner que la syntaxe utilisant des *backquotes* se voit régulièrement préférer celle utilisée dans cet article et employant le symbole `$`, comme pour l'interprétation des variables, associée à des parenthèses.

```
# syntaxe 'historique'
$ hostname=`hostname`
# 'nouvelle' syntaxe:
$ hostname=$(hostname)
```

On notera au passage, toujours sur le sujet des substitutions de commandes, qu'il est rarement nécessaire d'entourer ces dernières de guillemets, puisque la nature même de leur exécution garantit que le résultat soit proprement associé à la variable.

2 Gérer les arguments : getopt

Comme nous l'avons décrit précédemment, il est crucial de doter ses scripts d'arguments appropriés à leur usage et les plus explicites possible. Mais, évidemment une fois les arguments transmis, il reste à les traiter. Ceci inclut, dans le cas du Bash, d'associer leurs valeurs aux bonnes variables, et de vérifier que ces valeurs soient cohérentes avec les attentes du script.

Ce genre de travail est fastidieux, et, heureusement, il existe différents mécanismes pour aider le développeur dans cette tâche. Nous avons retenu ici l'utilisation de la commande **`getopt`** qui, sans être la plus souple, ni la plus puissante des options, reste néanmoins un bon outil, aussi très structurant.

La commande **`getopt`** permet de déclarer, par une série de lettres, les options supportées par le script. Une option étant ici un caractère préfixé d'un **`-`**. Ainsi, en déclarant la chaîne **`pde`** à **`getopt`**, vous indiquez à ce dernier que votre script supporte l'utilisation d'options **`-p`**, **`-d`** et **`-e`**.

En outre, on peut indiquer à **`getopt`**, par l'ajout d'un **`:`** après la lettre, que l'argument est associé à une valeur. Ainsi **`p:`** indique que le script reçoit un argument de type **`-p valeur`**.

La commande **`getopt`** s'imbrique naturellement dans les structures **`while`** et **`case`** du Bash, et permet donc d'analyser des arguments aussi aisément que le montre le code suivant :

```
while getopts hB: OPT; do
    case "$OPT" in
        h)
            usage
            exit 0
            ;;
        b)
            readonly BATCH_NAME=${OPTARG}
            ;;
        *)
            echo "Unrecognized options:${OPTARG}"
            ;;
    esac
done
```

Ainsi, cette structure nous permet de gérer un premier cas d'erreur possible : l'argument non reconnu. En outre, **`getopt`** lors de son analyse des arguments reçus, va non seulement définir le nom de l'argument analysé dans la variable **`OPT`**, mais aussi placer la valeur associée, le cas échéant, dans la variable **`OPTARG`**.

Et il ne reste donc plus qu'à définir les variables associées aux options qui sont nécessaires pour le script. À la sortie de cette boucle **`while`**, on peut aisément tester si toutes les valeurs nécessaires ont été définies, et si ces dernières semblent pertinentes :

```
if [ -z "${BATCH_NAME}" ]; then
    echo "The batch name is not optionnal."
    usage
    exit 1
fi
```

Enfin, dernier avantage, **`getopt`** ne nous interdit pas d'ajouter une série d'arguments supplémentaires qu'il ne gérera donc pas lui-même. Supposons, par exemple, que nous souhaitions passer aux scripts toute une série de noms d'instance de machines virtuelles, on peut aisément extraire ces arguments, après le traitement effectué par **`getopt`**.

Et, comme Bash supporte désormais la notion de tableau **[1]**, la récupération de ces arguments peut se faire en quelques lignes :

```
shift $(expr $OPTIND - 1)
declare -a INSTANCES="$@"

if [ -z "${BATCH_NAME}" -a -z "${INSTANCES}" ]; then
    usage
    exit 1
fi
```

Maintenant, il faut aussi reconnaître que **`getopt`** a également quelques limites claires. Par exemple, il ne peut pas analyser d'arguments plus explicites tels que **`--help`**. Comme sous-entendu au début de cette section, ce n'est donc certainement pas le *framework* absolu pour l'analyse d'arguments, mais il offre un moyen rapide et simple aux développeurs de scripts pour analyser leurs arguments d'entrée.

3 Gestion des erreurs

Un autre point important, lors de la conception de scripts, spécialement de scripts d'infrastructure, est la **gestion des erreurs**. Malheureusement, le Bash a plutôt mauvaise réputation dans ce domaine. La plupart des scripts ne vérifient que rarement que le statut d'une commande est bien égal à `0` après son exécution, et, à l'inverse, bien souvent, des scripts continuent à s'exécuter « malgré tout ».

Heureusement, il ne s'agit que de mauvaises pratiques et, en aucun cas, de défaut irrémédiable lié à l'interpréteur en lui-même. En effet, il est possible de configurer, lors de l'exécution, ce dernier pour s'interrompre immédiatement à la première commande exécutée retournant une erreur (statut différent de zéro) :

```
set -e
```

Utiliser systématiquement cette option, au début de chaque script, dès les premières heures de sa conception, en facilite le développement, puisque son exécution s'arrête dès que la première commande retourne une valeur de statut différente de zéro, et non plusieurs lignes plus loin.

En effet, il est fréquent que les scripts continuent de s'exécuter malgré l'apparition d'une erreur, et que le « crash », en tant que tel, n'intervienne donc que bien plus tard. Ce phénomène, fort désagréable, rend souvent difficile l'identification de la cause racine du problème, en plus d'entraîner des effets de bords (ex : création d'un répertoire utilisateur alors que l'utilisateur n'a pas été créé).

Ainsi, utiliser systématiquement ce mécanisme évite que vos systèmes ne se retrouvent dans un état incohérent, à la suite d'une exécution d'une série d'opérations qui n'auraient jamais dû avoir lieu, puisqu'une commande exécutée en amont a échoué.

Notez aussi que cette commande est réversible au cours de l'exécution du script, ce qui peut parfois se révéler fort pratique - voire nécessaire, si votre script doit néanmoins continuer à s'exécuter malgré une erreur, ou s'il est simplement en mesure de gérer l'erreur (par exemple, en réessayant simplement d'exécuter la commande une nouvelle fois).

La fonction suivante illustre ce dernier point :

```
transfer_file() {
    local source="${1}"
    local target="${2}"

    local timeout=120
    local pace=5
    local wait_since=0

    while [ ${wait_since} -le ${timeout} ] ; do
        set +e
        scp "${source}" "${target}"
        if [ ${?} -ne 0 ] ;
            let wait_since=${wait_since}+${pace}
        else
            return
        fi
        set -e
    done
    exit 1
}
```

3.1 Gérer les erreurs dans les commandes imbriquées

Une des fonctionnalités internes du Bash qui gagne réellement à être connue est très certainement la variable **`$PIPESTATUS`**. Cette variable interne permet en effet de vérifier que chacune des commandes d'une série de commandes imbriquées a été exécutée sans erreur !

Pour s'en convaincre, étudions l'exemple ci-dessous :

```
$ ls -l | mail | cat | cut -f1
No mail for rpelisse
$ echo "${?}"
0
$ ls -l | mail | cat | cut -f1
No mail for rpelisse
$ echo "0:${PIPESTATUS[0]} 1:${PIPESTATUS[1]} 2:${PIPESTATUS[2]}
3:${PIPESTATUS[3]}"
0:0 1:1 2:0 3:0
```

Dans cet extrait de code, l'appel à la commande `mail`, seconde commande invoquée au sein de cette série de commandes imbriquées, échoue, car l'utilisateur ne dispose pas de compte de messagerie électronique locale. Néanmoins, si l'on se contente de tester la valeur de la variable `${?}`, en bout de chaîne d'exécution, cette anomalie n'apparaît pas, puisqu'elle est, pour ainsi dire, masquée par l'exécution avec succès de la dernière commande, `cut` en l'occurrence, qui, elle, s'est exécutée avec succès.

```
var=$(ls -l | mail | cat | cut -f1=
for status in "${PIPESTATUS[@]}"
do
  if [  ${status} -ne 0 ] ; then
    exit ${status}
  fi
done
```

Heureusement, à l'aide de cette variable interne, `PIPESTATUS`, ou plutôt en fait de ce tableau, il est possible de vérifier, a posteriori, que, à chaque étape de l'exécution d'une série de commandes imbriquées, tout s'est déroulé sans erreur.

`PIPESTATUS` permet donc de rendre l'utilisation des commandes imbriquées beaucoup plus sûres, ce qui n'est pas négligeable, car l'utilisation de *pipes* apporte de nombreux avantages dans un script Bash.

Non seulement elle réduit la taille du code shell à une seule ligne, au lieu de plusieurs, mais elle permet surtout de passer les informations d'une commande à l'autre, directement en mémoire sans passer par des fichiers temporaires.

Ce dernier mécanisme conclut donc l'ensemble des éléments évoqués pour rendre nos scripts plus robustes. Une fois toutes ces techniques mises en place dans vos propres scripts, vous constaterez avec plaisir qu'ils sont désormais plus fiables, et que, par conséquent, vous vous en servirez non seulement plus souvent, mais que vous n'hésiterez plus à leur confier même des tâches plus critiques, car vous aurez désormais les bonnes pratiques pour leur garantir un comportement correct.

4 Rubrique à brac

4.1 Écrire un fichier temporaire

Lors de la conception d'un script Bash, on se retrouve, tôt ou tard, à devoir utiliser des fichiers temporaires. Et comme avec tous les langages de programmation du monde, on se retrouve dans la position désagréable de devoir « choisir » où placer ce fichier dans l'arborescence du système.

Ceci est toujours désagréable, car on ne sait jamais vraiment où placer ce fichier. Après tout, peut-on vraiment garantir qu'un emplacement existera toujours sur l'ensemble des systèmes où le script s'exécutera ? On peut opter pour un choix raisonnable, en prenant par exemple, le répertoire `/tmp`. Il est vraisemblable que ce dernier existera toujours, quel que soit le système, néanmoins ceci ne résout pas tout le problème pour autant.

Reste le nom du fichier. Comment s'assurer qu'il n'existe pas de fichier portant le même nom dans le répertoire ? Et si c'est le cas, comment faire ? Ajouter un suffixe ? Effacer le fichier existant ? Rapidement, cette simple opération d'écriture dans un fichier pose un grand nombre de questions...

Heureusement, il existe une commande fort utile nommée `mktemp`. Cette dernière permet de créer un fichier (ou un répertoire) dans `/tmp`. La commande garantit donc que le nom généré ne corresponde à aucun autre fichier déjà existant dans ce répertoire, vous libérant de facto de la gestion de potentiels conflits de noms.

L'exemple ci-dessous, bien que présentant très peu d'intérêt en termes fonctionnels, illustre bien l'utilisation de `mktemp` :

```
copy_init_scripts_without_comments() {
    local target_host="${1}"

    # copy script in temporay directory, and removes comments
    directory=$(mktemp -d)
    for file in /etc/init.d/
    do
        sed -e '/#/d' "${file}" > \
            "${directory}/$(basename ${file})"
    done

    # create an archive and push it to the targeted host
    archive=$(mktemp)
    tar czf "${archive}" "${directory}"
    scp "${archive}" \
        "${target_host}":$(date +%Y%m%d)-initscripts-backup.tgz
    rm -rf "${archive}" "${directory}"
}
```

4.2 Processus et PID

Au sein d'un script, on a souvent besoin d'une sorte d'identifiant unique pour indiquer, par exemple, comment aisément retrouver, au sein des fichiers de journalisation, l'instance du script qui a été responsable de telle ou telle action. Cet identifiant unique existe en fait déjà naturellement au sein du système, puisque tout script Bash qui s'exécute est, en fait, un processus, et qu'il possède donc, en toute logique, un PID.

Et ce PID est, de manière fort pratique, très aisé à récupérer au sein d'un script :

```
$ echo $$
14092
```

Mais ceci va plus loin, car on peut aussi lancer, au sein d'un script, des commandes en tâches de fond. Pour ce faire, il suffit simplement d'ajouter le symbole & à la fin de la ligne de commandes. Ce mécanisme est en général bien connu de la plupart des utilisateurs du shell, mais le fait qu'on puisse récupérer le PID du processus mis en tâche l'est souvent moins :

```
$ dd if=/dev/zero of=/dev/null &
$ pid=$(${!})
$ sleep 120
$ kill "${pid}"
```

Ce mécanisme se révèle très utile pour coordonner un ensemble de scripts s'exécutant en parallèle. Le script ci-dessous illustre bien ceci.

```
...
run() {
  local client="${1}"
  local nb_threads="${2}"
  local logfile="${3}"

  echo -n "starting ${nb_threads} with ${client}..."
  ./run.sh -n 10 ${client} -u "${url}" -t "${topic}" &>
"${logfile}" &
  export LAST_PID="$(echo $!)"
  echo "started (pid:${LAST_PID})"
}


export LAST_PID=""


run -c 10 "${log_dir}/10_consumers.log"
readonly CONSUMERS_10_PID="${LAST_PID}"
run -p 60 "${log_dir}/60_producers.log"
readonly PRODUCERS_60_PID="${LAST_PID}"
run -c 60 "${log_dir}/60_consumers.log"
readonly CONSUMERS_69_PID="${LAST_PID}"
echo "logs redirected into ${log_dir}"
echo -n "waits for 3 minutes before killing 10 consumers..."
if [ "${log}" = "true" ]; then
        tail -f ${log_dir}/*.log &
fi


sleep 180
echo "kill 10 consumers (${CONSUMERS_10_PID})"
kill "${CONSUMERS_10_PID}"
echo  "Done."
```

Note

Ce script a été conçu pour reproduire un problème d'accès concurrent par des processus « consommateurs » et « producteurs » [2]. Ce script utilise donc jusqu'à trois sous-processus pour démarrer deux groupes de consommateurs et un groupe de producteurs.

Pour reproduire le problème, il était nécessaire, après un certain temps de démarrage, d'interrompre abruptement un groupe de consommateurs. Ceci a aussi été automatisé dans le script, en utilisant le PID du processus lancé en tâche de fond précédemment.

Conclusion

Ces quelques pratiques et astuces sont loin d'être exhaustives, mais vous auront, espérons-le, donné déjà quelques éléments concrets pour améliorer vos scripts, changer vos habitudes et explorer plus en profondeur les nombreuses options mises à votre disposition par Bash. ■

Références

[1] Tableau associatif en Bash : http://www.linuxjournal.com/content/bash-associative-arrays

[2] Producteur / Consommateur : http://en.wikipedia.org/wiki/Producer-consumer_problem