Tirer davantage du shell Unix avec les fonctions

Stéphane Bortzmeyer

<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 23 janvier 2006

<https://www.bortzmeyer.org/shell-functions.html>

Je suis d'accord que le shell Unix est plus dur à apprendre, pour le débutant complet ou pour l'utilisateur occasionnel, que l'interface graphique, dite GUI. Mais, pour l'utilisateur quotidien, c'est un formidable outil de productivité. Comme le décrit bien Neal Stephenson dans son essai "*In the beginning was the command line*" <<https://www.bortzmeyer.org/command-line.html>>, le shell est un **langage**. Apprendre un langage est plus lent que de simplement faire des gestes et prononcer quelques onomatopées. Mais cela permet aussi bien plus de choses, et de manière bien plus efficace.

Un exemple qui est très difficile à faire avec les GUI : le shell permet de définir ses propres fonctions (une forme simplifiée des fonctions, les *alias*, est décrite dans un autre article <<https://www.bortzmeyer.org/shell-alias.html>>). Comme tout langage, il peut servir à définir de nouveaux mots ou concepts, qui permettront d'aller plus vite la prochaine fois.

Voyons quelques exemples concrets. Ils sont tirés de mon `.zshrc`, le fichier d'initialisation du shell que j'utilise, `zsh` (deux notes pour les gens peu familiarisés avec Unix : chaque utilisateur peut choisir son shell, et le fichier d'initialisation est lu au démarrage du shell, ce qui évite de retaper des définitions à chaque fois). On peut trouver bien d'autres fichiers d'initialisation sur l'excellent site <<http://www.dotfiles.com/>>.

Voici une première fonction qui simplifie l'envoi d'un formulaire signé au RIPE-NCC. Comme beaucoup de registres, le RIPE-NCC accepte des demandes de modification par des formulaires texte signés avec un outil comme GnuPG. Pour éviter d'oublier de signer et pour éviter d'envoyer le formulaire à la mauvaise adresse, une petite fonction est bien pratique :

```
ripe-sign-and-send() {
    if [ "$1" = "" ]; then
    echo "Usage: $0 form-file"
    return
    fi
    if [ ! -e $1 ]; then
    echo "$1 does not exist"
    return
    fi
    gpg --clearsign --armor --output - --default-key 'NIC France' $1 | \
    command mutt -s MODIFY \
    -e 'set from=bortzmeyer@nic.fr' \
    auto-dbm@ripe.net
}
```

Comment se lit-elle? On définit une fonction nommée `ripe-sign-and-send` qui commence par tester si elle a bien un argument (noté `$1`), si le fichier de ce nom existe (le shell est un langage de programmation complet, avec tests, boucles, etc) et elle appelle ensuite GnuPG puis passe le résultat à mutt pour envoi au RIPE-NCC. On note que je préfère, dans une fonction, employer les noms longs des options (`--armor` au lieu de `-a`) car on en tape le corps de la fonction qu'une fois et cela la rend plus lisible. En général, dans un script, il faut toujours utiliser les noms longs des options.

Une fois cette fonction définie, typiquement dans le fichier d'initialisation (`/.zshrc` pour mon shell), on peut l'utiliser facilement :

```
ripe-sign-and-send mnter-frnic.txt
```

et tout est fait automatiquement. Je dois dire que je me suis toujours demandé comment les gens qui ne font pas de fonctions shell pouvaient supporter des tâches répétitives qu'il faut documenter dans une procédure. Cela me semble un incroyable gaspillage de compétences.

Voyons une deuxième fonction. Elle me sert sur les machines NetBSD à trouver un paquetage déjà compilé en donnant une partie de son nom :

```
export NETBSD_SERVER=ftp.fr.netbsd.org
export PKG_PATH=ftp://$NETBSD_SERVER/pub/NetBSD/packages/'uname -r|cut -d _ -f 1'/'uname -p'/All
findpkg() {
    index=/tmp/NetBSD-'uname -m'-INDEX
    if [ ! -e $index ]; then
echo "Retrieving to $index..."
        wget --quiet -O $index ftp://$NETBSD_SERVER/pub/NetBSD/packages/pkgsrc/INDEX
    fi
    grep -i "^[^\|]*$1" $index | \
        awk -F\| '{print $1 " (" $4 ")"}'
}

```

On commence par déclarer, en dehors de la fonction, deux variables d'environnement, `NETBSD_SERVER` et `PKG_PATH`. Cette dernière est calculée à partir des résultats de la commande `uname`, qui affiche de l'information sur le système (NetBSD a des paquetages différents par architecture, et il tourne sur beaucoup d'architectures). Ensuite, on définit la fonction `findpkg` qui teste si un fichier d'index est déjà présent sur le disque (le shell a aussi des variables, ici, `$index`) et le télécharge sinon. Elle appelle ensuite `grep` pour fouiller dans le fichier `index` puis `awk` pour formater le résultat.

Les fonctions shell permettent d'enrichir les capacités d'un programme existant. Par exemple, l'excellent logiciel de gestion de versions darcs, que j'utilise pour ce blog, permet d'envoyer par courrier électronique les changements réalisés dans un dépôt. Le courrier électronique, qui marche même lorsque la connexion Internet n'est pas permanente, ne devrait pas être affecté si on est à un endroit perdu et déconnecté. Par défaut, darcs nécessite d'avoir une connexion Internet avec l'autre dépôt avec lequel on le compare. darcs n'a pas de commande pour envoyer les changements locaux, sans comparer avec un autre dépôt. Mais il est possible d'écrire cette commande :

```
# Send (by email) all the changes since a given synch point
darcsdiff() {
    if [ "$1" = "" ]; then
echo "Usage: $0 patch-ID"
return
    fi
    file=`mktemp`
    darcs changes --to-patch "$1" --context > ${file}
    darcs send --all --context ${file} .
    rm ${file}
}

```

Elle crée un fichier temporaire (la commande située entre les apostrophes inverses est exécutée et son résultat va dans la variable \$file), récupère le contexte des "patches" depuis un certain "patch" (donné en argument) avec `darcs changes` et les met dans le fichier temporaire. On envoie alors ces "patches" par courrier (avec `darcs send`). Et on détruit le fichier temporaire. On peut utiliser cette fonction avec `darcsdiff "ID of the patch BEFORE the first patch I want"`, par exemple :

```
% darcsdiff 'BIXI a Montreal'
What is the target email address? bortzmeyer@nic.fr
Successfully sent patch bundle to: bortzmeyer@nic.fr.
```

(Des détails sur l'utilisation de cette fonction se trouvent dans mon article sur la synchronisation de dépôts darcs par courrier <<https://www.bortzmeyer.org/synchroniser-depot-darcs-par-courrier.html>>.)

Une dernière fonction est plus complexe : une des grandes forces de l'outil ssh est sa capacité à faire suivre les messages du protocole X11 à travers le canal chiffré, ce qui augmente la sécurité, bien sûr, mais rend aussi l'utilisation de X11 bien plus facile (avant, il fallait définir une variable DISPLAY sans se tromper). Mais X11 consomme pas mal de bande passante et, si j'apprécie cette possibilité lorsque je me connecte sur une autre machine du même réseau local, j'en ai moins envie lorsque je travaille à grande distance, via des réseaux lents et peu fiables.

Comment déterminer automatiquement si le serveur est loin ou pas ? Une même machine (par exemple mon PC portable) peut se connecter à beaucoup d'endroits dans le monde et je ne peux donc pas mettre en dur la liste des machines proches ou lointaines. Une idée imparfaite mais qui marche suffisamment est de tester la durée d'aller-retour d'une demande d'écho, envoyée avec la commande ping.

Cela donne :

```
# Test the RTT (round-trip time) to a remote host with ping and then
# slogin to it, enabling or disabling the X11 forwarding depending on
# the value of the RTT
slogin () {

    if [ "$DISPLAY" = "" ]; then
command slogin -x $*
return
    fi
    threshold=35 # Milli-seconds of RTT
    if [ "$1" = "" ]; then
        echo "Usage: $0 host"
return
    fi
    # A better way to get the last argument?
    for arg in $*; do
last=$arg
    done
    host=$last
    # TODO: handle the user@host syntax
    if [ -z `which ping` ]; then
echo "$0 requires ping"
return
    fi
    # Some pings (Netkit ping on some Linuces) do not like fractional intervals
    ping -c 1 -q -n -i 0.3 $host > /dev/null 2>&1
    if [ $? ]; then
        ping_options="-c 3 -i 0.2"
    else # Less tests, to be faster
```

```
        ping_options="-c 2"
    fi
    average_rtt=`ping $ping_options -n -q $host | \
egrep '^(rtt|round-trip)' | cut -d= -f2 | cut -d/ -f2`
    if [ -z "$average_rtt" ]; then
echo "Cannot ping $host, assuming slow link, DISablingX forwarding"
command slogin -x $*
    else
if [ `expr $average_rtt "<=" $threshold` = 1 ]; then
echo "Fast link to $host, ENabling X forwarding"
command slogin -X $*
    else
echo "Slow link to $host, DISablingX forwarding"
command slogin -x $*
    fi
    fi
    title
}
```

Comme souvent en programmation, l'essentiel du code est occupé par des détails pour couvrir les 5 % de cas vicieux. Ici, il faut tenir compte du fait qu'il existe plusieurs programmes ping, avec des options différentes (d'où le test du code de retour de ping, \$!), que certains coupe-feu bloquent les demandes d'écho, etc.

En résumé, les fonctions sont une technique qui réalise le rêve de l'informatique : automatiser des tâches pénibles pour laisser les humains écrire pour leur blog.

Si vous souhaitez plus de détails en français sur le shell, vous pouvez consulter le cours de Philippe Dax <<http://www-igm.univ-mlv.fr/~dr/COURS/shell.html>> ou bien celui de Linux-Nantes <http://www.linux-nantes.fr.eu.org/old/docs/xavier/prog_sh/prog_sh.html>.