Christophe Blaess

Conseils pour l'écriture d'un script

Ecriture de scripts shell

Ce document a été réalisé sur système Linux avec les logiciels libres :

- **Formation+** pour préparer le cours complet,
- **LaTeX** pour produire les transparents et le support de cours imprimé,
- **Xfig** pour les graphiques vectoriels,
- **The Gimp** pour les images et les photos.

Support de cours : version 15

Ecriture de scripts shell

Conseils pour l'écriture d'un script

Christophe Blaess

Introduction

Règle d'or : l'écriture d'un script doit privilégier la lisibilité.

Exprimer clairement le travail du script évite les erreurs lors de la programmation initiale.

Plus un script est facile à lire, plus simples sont les opérations de maintenance et de portage.

On peut aisément réutiliser des portions d'un script bien écrit pour une tâche légèrement différente.

La taille du code, l'occupation mémoire ou la rapidité d'exécution sont des éléments secondaires dans la programmation par scripts.

Présentation générale du script

Insérer une ligne shebang

Le plus souvent : `#! /bin/sh` mais parfois : `#! /bin/ksh`

Ajouter un en-tête au fichier

Comportant :
- le nom du fichier et une description succinte,
- le nom et l'adresse e-mail de l'auteur,
- un numéro de version, et/ou une date de modification.

Utiliser des commentaires

Pour décrire **le but** des portions les plus complexes.

```
#! /bin/sh
################################################
# de_double.sh
#
# Elimine les doublons de la base de données,
# le fichier initial est toujours sauvegardé.
# Des options configurent le comportement.
#
# Auteur : Prenom NOM <prenom.nom@hote.com>
#
# Historique :
#    18/01/2005 (1.3) :
#      correction bug sur longueur des champs
#    14/01/2005 (1.2) :
#      ajout option -f
#    09/01/2005 (1.1) :
#      inversion de l'option -i
#    07/01/2005 (1.0) :
#      écriture initiale
################################################

        VERSION=1.3
```

Choisir avec soin les noms de variables

`fichier`, `repertoire`, `reponse`, `nb_lignes`, sont préférables à `f1`, `r`, `r1` ou `nl`.

Utiliser des noms détaillés pour les variables globales, utilisées à différents endroits du script.

Utiliser des noms brefs pour des variables employées localement dans des portions très courtes.

Habituellement, sous Unix, les variables contenant des paramètres fixes (chemin d'accès à l'application, nom de l'utilisateur, etc.) sont écrites en majuscules.

A la lecture, encadrer les variables par des guillemets

Pour protéger les éventuels espaces (noms de fichier, etc.)

Utiliser l'option `set -u` du shell

Pour détecter les variables non initialisées.

```
VERSION=1.3
REPERTOIRE_APPLI=/opt/lib/appli-1.0

set -u

[...]

for arg in "$@"
do
        if [ "$arg" = "-v" ]; then
                echo "$0 - $VERSION" >&2
                exit 0
        fi
        [...]
done

[...]

for fic in "${REPERTOIRE_APPLI}"/*.dat
do
        traitement_fichier "$fic"
done
```

Indenter le code

L'organisation logique du code doit être visible au premier coup d'oeil.

Utiliser plutôt des tabulations que des espaces.

Les blocs de code dépendant de structures de contrôle (boucles, tests...) doivent être clairement identifiés.

Mettre les fonctions en évidence

Les définitions de fonctions se trouvent en début de script.

Chaque fonction doit commencer par un commentaire décrivant son rôle, ses arguments et sa valeur de retour.

Limiter avec le mot-clé `local` la portée des variables utilisées dans les fonctions.

```
oui_ou_non()
{
        # Affiche la question passée en argument. Attend une réponse.
        # Renvoie Vrai si c'est Oui, Faux si c'est Non, et recommence sinon.

        local rep

        while true
        do
                echo "$@"
                printf "O/N ? "
                read rep
                case "$rep" in
                        [Oo]* ) return 0 ;;
                        [Nn]* ) return 1 ;;
                esac
        done
}

[...]

if oui_ou_non "Voulez-vous quitter le programme"; then exit 0; fi
```

Gestion des erreurs

Vérifier les arguments en ligne de commande

Avant d'entreprendre toute action, s'assurer que les valeurs numériques sont correctes, que les fichiers indiqués existent, que les répertoires peuvent être parcourus, etc.

Tester les codes de retour

Vérifier la réussite ou l'échec de **toutes** les commandes invoquées.

Afficher tous les messages interactifs sur la sortie d'erreur

Rediriger les messages d'information et d'erreur avec `>&2`

Utiliser un code de retour significatif

Si le script se termine normalement, renvoyer (avec `exit`) zéro (signifiant vrai). Sinon, renvoyer une valeur non-nulle variant suivant le type d'erreur et documentée avec le script.

```
if [ $# -ne 2 ]
then
        echo "usage: $0 source destination" >&2
        exit 1
fi

if [ ! -r "$1" ]
then
        echo "Impossible de lire $1" >&2
        exit 2
fi

if [ -e "$2" ] && [ ! -w "$2" ]
then
        echo "Impossible d'écrire dans $2" >&2
        exit 2
fi


[...]

exit 0
```

Prévoir un message d'aide

Une option en ligne de commande (`-h` par convention) affichera une aide détaillée.

Insérer des messages de diagnostic

Définir une variable au début du script `DEBUG=2`

Puis truffer le script de messages de diagnostic :

```
[ $DEBUG -gt 0 ] && echo "Fichier = $fichier" >&2
```

Configuration par des variables d'environnement

Laisser l'utilisateur configurer une partie du comportement du script par des variables d'environnement.

En cas d'absence, adopter un comportement par défaut :

```
mkdir -p ${INSTALL_DIR:-/usr/local/lib/appli-0.2}
```

Bibliothèques de fonctions

Regrouper dans un fichier particulier les fonctions utilitaires susceptibles d'êtres employées dans plusieurs scripts.

Utiliser le point (`.`) en début de script pour « sourcer » le contenu du fichier bibliothèque.

```
#! /bin/sh

. /usr/local/lib/application/mes_fonctions.sh

# les fonctions communes sont maintenant disponibles

...
```

Cette méthode peut servir pour charger des fichiers de constantes (chemins d'accès, répertoires, versions...)