2012

Rapport

Un projet pour créer un shell virtuel

Un rapport de script qui réalise l'archive, parcourir et extraire du répertoire.

Programmeur :
Co é Hu
Rapport écrire par :
Co é Hu et Claude Liu
UTSEUS
05/03/2012

Contenu

1. Remercier

Je remercie à monsieur Florent Retraint, Professeur de LO03, il m'a donné des grandes aides sur le cours et le projet.
Je remercie à Cheng Liang pour des guides d'archive.
Permettez-moi de vous présenter mes remerciements sincères.
Remercier à tous les personnes dans notre groupe :
Coé Hu - Kewei Hu – 09125694
Claude Liu – Chenya Liu - 09125695

2. Instructions pour l'utilisation

La nouvelle commande shell, nommévsh, qui fonctionne selon les trois modes suivants :

1er mode : c'est le mode archive .Ce mode s'invoque par :

vsh [–archive] [nom_archive] [nom_répertoire].

Cette commande crée une archive nommée nom_archive qui contient tout le répertoire nom_répertoire.

2ème mode : c'est le mode extract :

vsh [–extract] [nom-archive]

Cette commande permet d'extraire le contenu de l'archive qui s'appelle nom-archive dans le répertoire courant.

3ème mode : c'est le mode browse :

vsh [–archive] [nom-archive]

Il contient six commandes:

pwd: vsh :> pwd

ls: vsh:> ls [vide] ou [chemin relative] ou [chemin absolu] du répertoire.

cd : vsh :> cd [nom] ou [chemin relative] ou [chemin absolu] ou [/] ou [..] du répertoire.

cat : vsh :> cat [nom] ou [chemin relative] ou [chemin absolu] du fichier.

extrait : vsh:> extract [nom] ou [chemin relative] ou [chemin absolu] du fichier.

exit : vsh :> exit

3. Structure du script

3.1 Général

La structure de la main programme est comme suivant :

```
301 case $1 in
302 -archive)
303     /*les instructions*/
304 ;;
305 -extract)
306     /*les instructions*/
307 ;;
308 -browse)
309     /*les instructions*/
310 ;;
311 *)
312     /*les instructions*/
313 ;;
314 esac
```

On utilise ‘case’ pour faire les choisies, pour le fonction [-browse], on utilise ‘case’ aussi, c’est facile pour le fonction qui a beaucoup de choix.

3.2 Fonction archive

On utilise la fonction file_archive() pour créer un archive. Mais dans la fonction, on utilise une sub-fonction nomme list_alldir() pour créer le header et body. Maintenant, on vous présente le structure de la fonction list_alldir().

```
3 list_alldir(){
4       /*instruction ici*/
5       for file in $1/*
6       do
7          if [ -d $file ]; then
8          list_alldir $file
9          fi
10      done
11 }
```

On utilise un mécanisme qui s'appelle **récursif** pour traverser tous les répertoires. Ici, si on trouve un fichier, on affiche ses contenus, si on trouve un répertoire, On utilise la fonction lui-même pour traverser les fichiers et les répertoires dans le répertoire.

3.3 Fonction extract

Dans ce cas, on n'utilise plus le mécanisme récursif, mais on utilise une **boucle** pour chercher tous les fichiers.

3.4 Fonction browse

On utilise 'case' pour chercher le choix. Tous les instructions sont dans le boucle '**while read $LINE in**' pour réaliser l'interface.

```
98 browse(){
99 /*instructions ici*/
100 while read LINE in
101 do
102    /*instructions ici*/
103 case $premier in
104 pwd)
105    /*instructions ici*/
106 ;;
107 exit)
108    /*instructions ici*/
109 ;;
110 ls)
111    /*instructions ici*/
112 ;;
113 cd)
114    /*instructions ici*/
115 ;;
116 cat | extract)
117    /*instructions ici*/
118 ;;
119 *)
120    /*instructions ici*/
121 ;;
122 esac
123 done
124 }
```

4. Idée de la plan

4.1. Archive

Pour montrer le répertoire, simplement on peut utiliser une fonction *list_alldir()* **récursif**. Pour distinguer si une ligne est fichier ou répertoire on vérifie le premier bit de partie droit, s'il est '-', c'est un fichier, s'il est 'd', c'est un répertoire. Dans le chemin présent, le moment on trouve un fichier ou un classeur, s'il est un répertoire, on prend directement la huitième(nom) , la première(droit) et la cinquième(taille) partie au fichier provisoire qui s'appelle ***header.tmp***, en le même temps, quand on trouve un fichier, on y prend les trois partie et on calcule premièrement combien de lignes il comporte et le stocke en utilisant un variable ***long,*** et deuxièmement ou on doit calculer le première ligne il doit placer dans le partie body. On copie le contenu dans ce fichier et le passe à un autre fichier provisoire qui s'appelle ***body.tmp*** en utilisant un variable ***debut.*** On calcule le variable par calculer la taille du fichier body.tmp, et *début* égale à la taille plus un.

```
 3 list_alldir(){
 4       echo directory" "$1   >>header.tmp
 5       ls -l $1 | tr -s ' '|grep "^d"|awk '{ print $8,$1,$5 }' >>header.tmp
 6       ls -l $1 | tr -s ' '|grep "^-"|awk '{ print $8 }' | while read LINE in
 7       do
 8          long=`expr $( cat $1/$LINE | wc -l )`
 9          debut=`expr $(cat body.tmp | wc -l ) + 1`
10          cat $1/$LINE >>body.tmp
11          ls -l $1/$LINE | tr -s ' '|awk '{ print nom,$1,$5,debut,long }'
   nom=$LINE debut=$debut long=$long >>header.tmp
12       done
```

Quand tout ci-dessus est fini, on va marcher la fonction ***list_alldir()*** encore une fois récursivement jusqu'au dernier répertoire.

```
15      for file in $1/*
16      do
17          if [ -d $file ]; then
18          list_alldir $file
19          fi
20      done
```

Quand on aura fini la fonction ***list_alldir()***, tous les contenus dans ***head.tmp*** et ***body.tmp*** sont dont on a besoin.

Pour afficher, on écrit un autre fonction ***file_archive().*** A la fin de la fonction [-archive], on a des choses ce que on a besoin, mais tous les choses sont dans les fichiers provisoires, ils ne sont pas affichés. Cette fonction nous aide à afficher. On prend le fichier header.tmp et body.tmp dans le fichier datif, et on note le début de header et body dans la première ligne du fichier. Et si après on finit l'exécution, on conserver des fichiers provisoires, ça va avoir trop de fichiers qui sont inutile, donc on archive des contenus et après on élimine des fichiers provisoires.

```
43 file_archive(){
44    list_alldir "$1"
45    first=`expr $( echo | wc -l header.tmp| cut -d " " -f1 ) + 3`
46    echo "3:"${first} >> $2
47    echo "" >> $2
48    cat header.tmp >> $2
49    cat body.tmp >> $2
50    rm *.tmp
51 }
```

4.2. Extract

Pour extraire, on a besoin de juger basé sur chaque ligne, on va décider quelles lignes représente le nom d'un fichier ou d'un classeur, quelles lignes représente le contenu d'un fichier, donc on doit discuter tous les conditions. Il y a 4 conditions différentes :

a) le symbole '@'

b) le contenu

c) le nom d'un fichier ou classeur

d) le répertoire.

Comment on distingue ? Premièrement, c'est facile de distinguer le symbole '@', donc on peut le déterminer d'abord. Après, on sait que la liste a le format suivant : <nom> <droits d'accès> <taille> (informations complémentaires). Il est séparé par ' ', et il y a$1 $2 $3 $4 et $5, s'il n'y pas de $3, ça peut être directory, donc on crée un chemin basé sur cette ligne.

```
53 extrait(){
54    body_debut=`expr $( cat $1 | awk -F ":" ' NR==1 { print $2 }' ) - 1`
55    IFS="\n"
56    cat $1 | awk 'NR==3,NR==line' line=$body_debut | while read LINE in
57    do
58       if [ $LINE != "@" ]
59       then
60          tmp1=$( echo $LINE | cut -d " " -f1 )
61          tmp=$( echo $LINE | cut -d " " -f3 )
62          if [ -z "$tmp" ]
63          then
64             if [ $tmp1 = "directory" ]
65             then
66             route=$( echo $LINE | cut -d " " -f2 )
67             mkdir -p ./$route
68             fi
```

S'il n'est pas '@' ni directory, ça peut être un classeur ou fichier, on peut le déterminer par la première lettre, 'd' représente un classeur et '-' représente un fichier. Si c'est un classeur on le crée basé sur le chemin, si c'est un fichier, on peut rendre son contenu basé sur $4 et $5 qui représentent le contenu de ce fichier. Particulièrement, si $5 est 0, ça veut dire le fichier est vide, on ne prend pas de contenue sur le body, si le fichier n'est pas vide, on prend les contenue entre $4 et $4+$5.

```
76            if [ $is_dir = "d" ]
77            then
78               mkdir -p ./$route/$dir_file_nom
79            elif [ $is_dir = "-" ]
80            then
81               touch ./$route/$dir_file_nom
82               tmp4=$( echo $LINE | cut -d " " -f5 )
83               if [ $tmp4 -gt 0 ]
84               then
85                  tmp3=$( echo $LINE | cut -d " " -f4 )
86                  line_debut=`expr $body_debut + $tmp3 `
87                  line_fin=`expr $line_debut + $tmp4 - 1`
88                  cat $1 | awk  'NR==debut,NR==fin' debut=$line_debut fin=
   $line_fin >> ./$route/$dir_file_nom
89               fi
90            fi
```

Particulièrement, on doit autoriser des fichiers et classeurs. On fait le tronçonnage de la partie droite, les premier trois bit présentent le droit d'user, la quatrième à sixième bit présentent le droit de groupe et les dernier trois bit présentent le droit d'autre.

```
71            droit=$( echo $LINE | cut -d " " -f2 )
72            is_dir=$( echo ${droit:0:1} )
73            u=$( echo ${droit:1:3} | sed -e 's/-//g' )
74            g=$( echo ${droit:4:3} | sed -e 's/-//g' )
75            o=$( echo ${droit:7:3} | sed -e 's/-//g' )

91            chmod o=$o  ./$route/$dir_file_nom
92            chmod u=$u  ./$route/$dir_file_nom
93            chmod g=$g  ./$route/$dir_file_nom
```

Tout ci-dessus explique comment la fonction ***extrait()*** fonctionne.

Explications additionnel :

Quand on utilise la fonction ***list_alldir()***, on utilise ***head.tmp*** et ***body.tmp***, donc avant d'exécuter, on a besoin de éliminer les fichiers s'ils existent car ça peut-être pas la première fois on l'exécute. Et en le même temps, ***archive*** doit être éliminé aussi. Donc on écrit la fonction ***init_del().***

4.3. Browse

Tout d'abord, on cherche le début du body et la racine du répertoire. Spécifiquement, on prend le chemin présent dans le fichier nommé path_now.tmp.

```
browse(){
IFS="\n"
body_debut=`expr $( cat $1 | awk -F ":" ' NR==1 { print $2 }' ) - 1`
path_racine=`expr $( cat $1 | awk 'NR==3' | awk ' { print $2 }' )`
cat $1 | awk 'NR==3' | awk ' { print $2 }' > path_now.tmp
echo -n "vsh:>"
```

Pour réaliser l'interface entre script et utilisateur, on toujours lit les commandes tapées par utilisateur, donc on utilise le boucle 'while read LINE in'. Pour distinguer les commandes, on utilise ***'case $paramètre in *)'***.

4.3.1. Paramètres

Dans le boucle 'while', on dispose les paramètres. Le premier présent le commande et la deuxième présent le chemin et le deuxième peut-être vide ou inutile. Donc il faut classer les paramètres.

Pour disposer le chemin, on classe dans le type suivant :

1. Chemins relatifs,
2. Chemins absolus,
3. Fichier ou répertoire dans le répertoire actuel.

Pour le chemin relatif, on pense qu'il ne commence pas par '/' mais il comporte '/' et il peut terminer par '/' ou pas.

Pour le chemin absolu, on pense qu'il commence par '/', il peut comporter plusieurs '/' et terminer par '/' ou pas.

Pour le fichier ou répertoire dans le répertoire actuel, il ne peut pas comporte '/'.

On supprime le dernier '/' de deuxième paramètre si il termine avec '/', et on ne supprime pas si le deuxième paramètre est '/'.

```
106 premier=$( echo $LINE | awk '{ print $1 }' )
107 deux=$( echo $LINE | awk '{ print $2 }' )
108 if [ "$deux" != "/" ]
109 then
110    si_fin_s=$( echo $deux | grep "/$" )
111    if [ -n $si_fin_s ]
112    then
113       deux=$( echo $deux | sed -e 's!/$!!' )
114    fi
115 fi
```

4.3.2. La commande pwd

D'abord, on lit le chemin actuel dans le fichier ***path_now.tmp***, s'il égale au chemin racine, on affiche '/', si non, on affiche le chemin actuel sauf la partie racine.

```
117 pwd)
118    path_now=`expr $( cat path_now.tmp )`
119    if [ $path_racine = $path_now ]
120    then
121       echo "/"
122    else
123       path_dis=$( cat path_now.tmp | sed -e "s/^$path_racine//" )
124       echo $path_dis
125    fi
```

4.3.3. La command ls

D'abord, on lit le chemin actuel dans le fichier ***path_now.tmp*** et le prend dans le variable nommé ***path_now***. On prend le premier bit pour déterminer s'il est un chemin relatif ou absolu. Ensuite, on teste le deuxième paramètre, s'il vide, on prend le chemin qui on cherche égale a chemin actuel, s'il est un chemin absolu, on ajoute chemin racine avant, s'il est chemin relatif, on ajoute chemin racine et ajoute '/' aussi.

```
132    line_now=3
133    path_now=`expr $( cat path_now.tmp )`
134    si_abs_path=$( echo ${deux:0:1} )
135    if [ -z $deux ]
136    then
137       path_search=$path_now
138    elif [ "$si_abs_path" = "/" ]
139    then
140       path_search=$path_racine$deux
141    else
142       path_search=$path_now/$deux
143    fi
```

Pour chercher le chemin datif, on utilise deux boucle ***'while',*** le premier boucle cherche le début de répertoire datif et le deuxième boucle affiche tous les fichiers et les répertoires dans le répertoire datif. Pour distinguer si le répertoire existe, on créer un fichier nommé ***dir_find.tmp,*** si les deux boucle a fini et le fichier ***dir_fin.tmp*** n'existe pas, on pense que le chemin de répertoire est faux.

```
144    cat $1 | awk 'NR==3,NR==line' line=$body_debut | cut -d " " -f2 |
    while read FILE in
145    do
146       line_now=`expr $line_now + 1`
147       if [ $FILE = $path_search ]
148       then
149          echo "1" > dir_find.tmp
150          cat $1 | awk 'NR==line,NR==line_fin' line=$line_now line_fin=
    $body_debut | while read DIR in
151          do
```

Pour noter s'il un répertoire, fichier ou fichier exécutable, on cherche le partie droite.

```
154             droit=$( echo $DIR | cut -d " " -f2 )
155             is_dir=$( echo ${droit:0:1} )
156             is_excu=$( echo $droit |  grep "x" )
157             if [ $is_dir = "d" ]
158             then
159                echo $DIR | awk ' { ORS=" ";print $1 "/" } '
160             else
161                if [ -n $is_excu  ]
162                then
163                   echo $DIR | awk ' { ORS=" ";print $1 "*" } '
164                else
165                   echo $DIR | awk ' { ORS=" ";print $1 } '
166             fi
```

4.3.4. Le command cd

Pour le deuxième paramètre, il y a trois types suivant :

a) Pour entrer le répertoire racine : [/]

b) Pour remonter d'un niveau dans la hiérarchie : [..]

c) Pour entrer le répertoire avec le chemin relatif ou absolu : [chemin]

a) le répertoire racine

On prend directement le chemin racine dans le fichier path_now.

b) remonter d'un niveau dans la hiérarchie

D'abord, on regard si on a déjà dans le répertoire racine, si oui, on ne fait pas, si non on diviser la dernière partie par '/' et on prend la première partie dans le fichier path_now.

c) le répertoire avec le chemin

D'abord, on teste si le chemin est relatif ou absolu, la méthode est comme le command ls.

Apres le processus, on prend le chemin dans le variable nommé path_entre.

Ensuite, on filtre les lignes commencées par 'directory', et on cherche les lignes filtrées s'il égale au variable path_entre, on créer un fichier nommé file_find.tmp et on prend le chemin datif dans le fichier ***path_now.tmp***.

```
205       cat $1 | awk 'NR==3,NR==line' line=$body_debut | grep
    "^directory " |awk '{ print $2 }' | while read dire in
206       do
207          if [ $dire = $path_entre ]
208          then
209             echo $dire > path_now.tmp
210             echo "1" > file_find.tmp
211             break
212          fi
213       done
```

4.3.5 Le command cat

Pour le command cat, la méthode pour disposer le chemin est un peu différent. On crée deux variables nommées ***file_path, file_nom*** et ***path_search.***

File_nom indique le nom de fichier sauf le chemin, ***file_path*** indique le chemin de fichier sauf le nom et ***path_search*** est égale au chemin ***file_path*** ajoute avec le chemin racine. On utilise le variable ***path_search*** pour localiser la ligne dans le fichier archive datif.

D'abord, si le deuxième paramètre est vide, on demande l'utilisateur de taper le chemin du fichier.si le chemin est absolu, on prend le variable égale au chemin racine avec ***file_path***, si le chemin n'est pas absolu ou relatif, on prend le variable path_search égale au chemin actuel, si le chemin est relatif, on le ***path_search*** égale au chemin racine plus '/' plus variable ***file_path.***

```
227   if [ -z $deux ]
228   then
229      echo "vous devez appliquer un nom de fichier"
230   elif [ $si_abs_path = "/" ]
231   then
232      file_path=$( echo ${deux%/*} )
233      path_search=$path_racine$file_path
234      file_nom=$( echo ${deux##*/} )
235   elif [ -z $si_rel_path ]
236   then
237      path_search=$path_now
238      file_nom=$deux
239   elif [ -n $si_rel_path ]
240   then
241      file_path=$( echo ${deux%/*} )
242      path_search=$path_now/$file_path
243      file_nom=$( echo ${deux##*/} )
244   fi
```

Ensuite, on utilise deux boucles 'while' pour chercher le fichier datif et afficher les contenus à l'écran.

Apres on a cherché la ligne de header qui présente le fichier datif on doit localiser les contenus dans le fichier archive, on lit la quatrième et cinquième partie de la ligne et on prend la quatrième partie dans le variable nommé '***debut***' et cinquième dans le variable nommé 'long'. On calcule le numéro de la ligne début par ajouter ***$debut*** et le variable enivrement nommé ***body_debut*** qui est créé tout d'abord, il indique le début de body dans le fichier archive.

```
258                nom=$( echo $DIR | awk ' { print $1 } ' )
259                if [ $file_nom = $nom ]
260                then
261                   debut=$( echo $DIR | cut -d " " -f4 )
262                   debut_abs=`expr $debut + $body_debut `
263                   long=$( echo $DIR | cut -d " " -f5 )
264                   fin_abs=`expr $debut_abs + $long - 1 `
```

On aussi créer un fichier nommé ***file_nom.tmp*** pour déterminer si le fichier existe ou pas.

4.3.6 Le command extract

C’est comme le command cat, on ne modifie que la ligne pour afficher les contenus, on prend les contenus dans le fichier datif et on n’afficher pas à l’écran.

```
265                     #echo "le contenu du fichier $file_nom sont:"
266                     if [ $premier = "cat" ]
267                     then
268                         cat $1 | awk 'NR==de,NR==fi' de=$debut_abs fi=
    $fin_abs
269                     else
270                         cat $1 | awk 'NR==de,NR==fi' de=$debut_abs fi=
    $fin_abs > $file_nom
271                     fi
```

5. Le Main Fonction

Quand on va exécuter un programme, la fonction doit être choisie par le premier paramètre $1. Si c’est ‘archive’ la fonction ***file_archive()*** est choisi, en même temps on doit confirmer l’instruction est correct parce que l’instruction correct a besoin de 3 paramètres. Et c’est la même chose pour ‘extrait’ sauf qu’il a besoin de 2 paramètres et il doit vérifier s’il existe le fichier avec lequel on extrait.

Pour les gens qui ne savent pas comment écrire l’instruction, il peut input ’-help’ pour l’aider.

6. Les commandes systèmes

En faisant le projet, l’instruction ‘sed’ et ‘awk’ et d’autre sont utilisé souvent et le plus important.

6.1 Le command awk

On utilise ‘awk’ pour séparer l’information et ne trouver que des choses dont on a besoin, il est efficace, d’ailleurs c’est facile de l’utiliser, il épargne beaucoup de temps.

```
body_debut=`expr $( cat $1 | awk -F ":" ' NR==1 { print $2 }' ) - 1`
```

Dans cette instruction, on sépare une information par ‘ : ’ et seulement le premier phrase est séparé, et après on affiche le deuxième élément.

6.2 Le command sed

L’instruction ‘sed’ est utilisée plutôt pour remplacer.

```
u=$( echo ${droit:1:3} | sed -e 's/-//g' )
g=$( echo ${droit:4:3} | sed -e 's/-//g' )
o=$( echo ${droit:7:3} | sed -e 's/-//g' )
```

Dans l’instruction, on sépare le variable droit et choisit $1~$3, après on utilise sed pour éliminer le symbole ’ / ’, ou on peut dire le remplacer par espace.

6.3 Les d’autres commandes

a)Cut pour cribler

```
tmp1=$( echo $LINE | cut -d " " -f1 )
```

Dans ce cas, on crible le variable $LINE par ‘(space)‘ et on obtient le premier partie.

b) Grep pour filtrer des caractères

```
ls -l $1 | tr -s ' '|grep "^d"|awk '{ print $8,$1,$5 }' >>header.tmp
```

Dans ce cas, on ne obtient que les expression commence par ‘d’.

c) Touch pour créer un fichier

```
touch ./$route/$dir_file_nom
```

d) Echo pour tronçonnage

```
file_path=$( echo ${deux%/*} )
file_nom=$( echo ${deux##*/} )
```

Dans le premier cas, on a obtenu tous les caractères à gauche de dernier ‘/’.

Dans le deuxième cas, on a obtenu tous les caractères à droite de dernier ‘/’.

Toutes sont importantes pour traiter un texte.

En un point de vue, sed et awk peuvent réaliser le même résultat, mais awk est souvent utilisé pour séparer et trouver, sed est utilisé plutôt pour remplacer. Toutes les instructions sont élémentaires et importants, ils nous bénéficient beaucoup.

7. Références

[W0] http://www.ibm.com/developerworks/cn/linux/l-bash-test.html

[W1] http://www.360doc.com/content/10/0928/16/3234041_57087955.shtml

[w2] http://hi.baidu.com/kaleidoscope88/blog/item/84a080d9835959f376c6389a.html

[W3]Linux Shell Scripting Cookbook de Sarah Lakshman

8. Annexes

Le script du projet : vsh.sh et vsh.pdf