Scripting

Struttura
Variabili
Interazione con l'utente
Controllo di flusso
Controllo dei processi
Utilità varie

Script

Uno shell script è un file di testo con flag di eseguibilità impostato e, che inizia con la stringa #!/bin/sh o #!/bin/bash.

Tale stringa viene vista dalla shell stessa come un commento (come ogni stringa preceduta da **#**), mentre per il s.o. #! rappresenta il *magic number* che identifica gli *script*, cioè quei file comandi non in linguaggio macchina che possono essere compresi solo con l'ausilio di un *interprete*.

/bin/bash è appunto l'interprete da invocare per gli script shell, ma il meccanismo vale anche per altri linguaggi (ad es. #!/usr/bin/perl)

Struttura degli script

I caratteri contenuti in uno script vengono interpretati dalla shell come se fossero digitati dall'utente. Ovviamente la possibilità di codificare in uno script comandi anche complessi richiede la disponibilità di strutture di controllo per eseguire test o per iterare azioni ripetitive. Tutte le strutture accettate da uno script possono comunque essere utilizzate sulla linea di comando.

In questo contesto è particolarmente utile sapere come includere script in script, per riusare il codice e realizzare file di configurazione che, anziché inventare formati esotici, semplicemente definiscono variabili della shell. Per questo si può usare il comando **source**

File /etc/data.conf:

```
TDIR=/tmp
ELEMENTS=$(seq 5 45)
```

File loop.sh:

```
#!/bin/bash
source /etc/data.conf
for i in $ELEMENTS ; do
   rm -f $TDIR/$i
done
```

Variabili posizionali

Ogni script può accedere agli argomenti indicati sulla propria linea di comando, utilizzando le variabili:

`$0, $1, $2, ...`	comando, primo arg, secondo arg, …
`$*` e `$@`	vengono espanse in $1 $2 … ma questo può causare problemi se gli argomenti contengono caratteri speciali, invece...
`"$*"` e `"$@"`	vengono espansi in "$1 $2 …" e "$1" "$2" …
`$#`	contiene il numero di argomenti

Il comando **shift** riduce per scorrimento l'elenco delle variabili posizionali, assegnando a $N il contenuto di $N+1 (il valore di $1 verrà quindi perso e $# sarà decrementato di 1)

```
echo $# $1 $2 $3     → 3 pippo pluto paperino
shift
echo $# $1 $2        → 2 pluto paperino
```

Assegnamento condizionale a variabili

Quando si lavora con le variabili fornite sulla linea di comando, più spesso che in altri casi, è utile poter gestire facilmente l'assegnazione di valori di default

```
FILEDIR=${1:-"/tmp"}

   # usa il default se $1 è null o not set

cd ${HOME:=/tmp}

   # usa il default se $HOME è null o not set, nel qual caso
viene inizializzata al default

FILESRC=${2:?"Error. You must supply a source file."}

   # stampa l'errore ed esce se $2  è null o not set
```

Nota: omettendo il "**:**", la stringa vuota viene considerata valida ed il valore di default è usato solo se la variabile fornita è not set

Interagire con l'utente – read

Si può usare read per leggere stringhe dal terminale, a patto che lo stdin dello script non sia ridiretto:

```
cat file | while read riga; do
   elabora "$riga"
   echo -n 'premi invio per la prossima riga'
   read risposta … legge un'ulteriore riga di file!
done
```

Soluzione possibile: leggere veramente dal terminale

```
read < $(tty)
```

Problema ulteriore: la subshell non è associata ad un terminale... come risolverlo?

Interagire con l'utente – read

read può effettuare il parsing della riga letta

```
$ read FIRST SECOND THIRD REST ; echo   "$FIRST /
$SECOND / $THIRD / $REST"
Nel mezzo del cammin di nostra vita
Nel / mezzo / del / cammin di nostra vita
$
```

La variabile d'ambiente IFS (input field separator) decreta quale carattere rappresenta il separatore di campo (di default qualsiasi blank)

```
$ IFS='"' read HTML1 TYPE HTML2 NAME HTML3 VALUE
HTML4 ; echo "$NAME = $VALUE"
<input type="hidden" name="page" value="login">
page = login
$
```

Interagire con l'utente – select

Si può automatizzare la selezione di alternative con il built-in select.

```
directorylist="Finished $(for i in /*;do [ -d "$i" ] && echo $i; done)"
PS3='Directory to process? ' # Set a useful select prompt
until [ "$directory" == "Finished" ]; do
    printf "%b" "\a\n\nSelect a directory to process:\n" >&2
    select directory in $directorylist; do
        # User types a number which is stored in $REPLY, but select
        # returns the value of the entry
        if [ "$directory" = "Finished" ]; then
            echo "Finished processing directories."
            break
        elif [ -n "$directory" ]; then
            echo "You chose number $REPLY, processing $directory..."
            # Do something here
            break
        else
            echo "Invalid selection!"
        fi # end of handle user's selection
    done # end of select a directory
done # end of while not finished
```

Parsing di opzioni sulla riga di comando

Per costruire script che supportino la classica sintassi
comando opzioni_precedute_dal_trattino argomenti
è utile il builtin getopts.
Sintassi:
getopts optstring NAME [arg]

Stringa che definisce i caratteri da riconoscere come opzioni. Se un carattere è seguito da : significa che è atteso un parametro per quell'opzione

Nome di variabile in cui collocare il parametro correntemente analizzato

Funzionamento: ad ogni invocazione, getopts esamina una variabile posizionale, assegnando il suo indice ad **OPTIND** ed il suo contenuto a **NAME.** Se è un'opzione che richiede un argomento nella successiva variabile posizionale, questo viene letto (incrementando **OPTIND)** ed assegnato a **OPTARG.**

getopt – una "ricetta"

```
#!/usr/bin/env bash
aflag=
bflag=
while getopts 'ab:' OPTION ; do
     case $OPTION in
         a)    aflag=1
                       ;;
         b)    bflag=1
               bval="$OPTARG"
                       ;;
         ?)    printf "Usage: %s: [-a] [-b value] args\n" $(basename $0) >&2
               exit 2
                       ;;
     esac
done
shift $(($OPTIND – 1))    # getopts cycles over $*, doesn't shift it

if [ "$aflag" ] ; then
     printf "Option -a specified\n"
fi
if [ "$bflag" ] ; then
      printf 'Option -b "%s" specified\n' "$bval"
fi
printf "Remaining arguments are: %s\n" "$*"
```

Array

bash supporta gli array monodimensionali:

```
$ declare -a MYVECTOR
$ MYVECTOR=(un elenco di elementi)
$ echo ${MYVECTOR[2]}
di
$ STRINGA="effettua.il.parsing.come.read"
$ IFS='.' MYVECTOR=($STRINGA)
$ echo ${MYVECTOR[2]}
parsing
$
```

Per visualizzare tutti gli elementi dell'array si può usare l'indice * o @
La differenza di comportamento è la stessa che c'è tra $* e $@

```
$ echo ${MYVECTOR[*]}
effettua il parsing come read
```

Array

Non è obbligatorio usare indici consecutivi.
Se si vuole conoscere il set di indici corrispondenti a celle assegnate dell'array, si utilizza **${!name[@]}**

```
$ A[0]="un elemento"
$ A[2]="un altro elemento"
$ echo ${!A[@]}
0 2
$
```

Array + expansion ≈ indirezione ed array associativi

Un'espansione particolare permette di ottenere il nome di una variabile, che poi può essere usato in qualsiasi altra espressione, ottenendo un riferimenti indiretto ai valori:

```
$ CHIAVE=PIPPO
$ PIPPO=VALORE
$ echo ${!CHIAVE}
VALORE
```

In Bash 4 e superiori sono supportati i veri array associativi (in cui l'indice può essere una stringa, non solo un numero)

```
$ declare -A ASAR
$ ASAR[chiaveuno]=valoreuno
$ echo ${ASAR[chiaveuno]}
valoreuno
$ KEY=chiaveuno
$ echo ${ASAR[$KEY]}
valoreuno
```

Notare la maiuscola

Verifica di condizioni

La verifica di una condizione e la specifica di una o più azioni da eseguire in modo condizionale si ottiene con il costrutto:

if <comando test 1>
then
    <comandi se test 1 ok>
[**elif** <comando test 2>
then
    <comandi se test 2 ok>]
[**else**
    <comandi se nessun test ok>]
fi

se il "comando test" restituisce un valore di ritorno =0 (OK), allora vengono eseguiti i comandi indicati nella clausola **then**.

Il case

La verifica di condizioni multiple è realizzabile più leggibilmente che con una catena di **elif** con il costrutto **case**:

```
case "$variabile" in
    nome1        ) echo vale nome1 ;;
    nome?        ) echo vale nome2, nomea, nomez ;;
    nome*        ) echo vale nome11, nome, nomepippo ;;
    [1-9]nome    ) echo vale 1nome, 2nome, …, 9nome ;;
    *            ) echo non cade in nessuna delle precedenti
;;
esac
```

Test

Il comando che viene eseguito per verificare una condizione if è spesso **test**. L'invocazione del test può essere abbreviata con il simbolo **[**

```
if test -e $filename
then
    rm $filename
fi
```

```
if [ -e $filename ]
then
    rm $filename
fi
```

Test unari

-d file	true se file esiste ed è una directory
-e file	true se file esiste
-f file	true se file esiste ed è un file regolare
-r file	true se file esiste ed è leggibile
-s file	true se file esiste ed ha dimensione >0
-w file	true se file esiste ed è scrivibile
-x file	true se file esiste ed è eseguibile
-z string	true se la stringa è vuota
-n string	true se string è non vuota

Test binari

file1 -nt file2	true se file1 è più 'nuovo' di file2
file1 -ot file2	true se file1 è più vecchio di file2
string1=string2	true se le stringhe sono uguali
string1 != string2	true se sono diverse
arg1 OP arg2	verifiche aritmetiche sugli operandi arg1 e arg2 - OP può valere:

-eq	uguaglianza numerica
-ne	diversità numerica
-lt	arg1 minore (strettamente) di arg2
-le	arg1 minore o uguale ad arg2
-gt	arg1 maggiore (strettamente) di arg2
-ge	arg2 maggiore o uguale ad arg2

Cicli definiti

E' possibile eseguire ripetutamente un insieme di comandi assegnando, volta per volta, ad una variabile un valore prelevato da una lista:

```
for numero in 1 2 3 4 5
do
    echo $numero
done
```

La lista di valori può essere anche omessa, in tal caso la variabile assume a turno il valore degli argomenti passati all'invocazione dello script.

Cicli definiti

Ricordiamo che la shell espande ogni stringa con wildcard nell'elenco dei nomi di file che concordano con tale schema. Con il **for** questo è molto utile:

```
for filename in *        # ←   esegue il ciclo per ogni file della
do                       #     directory corrente
    echo $filename       # ←   cosa succede se un file ha un
done                     #     carattere speciale nel nome?
```

Cicli definiti

Il comando seq è molto utile per realizzare cicli definiti che iterano su di un contatore:

```
for fp in $(seq 1.0  .01  1.1)...
```

start — incremento — end

Nelle versioni recenti di bash inoltre è disponibile una sintassi C-like del costrutto for – le espressioni utilizzabili sono più generali ma operano solo su numeri interi

```
for (( i=0, j=0 ; i+j < 10 ; i++, j++ )) …
```

Espressioni di inizializzazione — Test di terminazione* — Espressioni eseguite ad ogni iterazione

* attenzione, se più d'uno, i primi sono eseguiti ma solo l'ultimo determina se il ciclo prosegue (true) o si interrompe (false)

Cicli indefiniti

La shell consente anche l'uso di cicli non enumerativi che, al contrario del for, eseguono l'iterazione fino all'avverarsi di una condizione.
I cicli non enumerativi sono **while** ed **until**.

```
while [ $i -le 10 ]
do
    echo $i
    i=`expr $i + 1`
done
```

oppure

```
until [ $i -gt 10 ]
```

L'unica differenza tra i due è la negazione della condizione di uscita.

Interruzione forzata dei cicli

E' possibile uscire anticipatamente da un ciclo utilizzando la keyword **break**.

```
while true
do
    if [ $i -gt 10 ]
    then
        break
    fi
    echo $i
done
```

All'interno di un ciclo si puo' trovare anche **continue** per forzare l'esecuzione immediata dell'iterazione successiva.

Le funzioni

Il linguaggio di script della shell consente anche le definizione di funzioni per la modularizzazione delle attività.

Le funzioni sono semplicemente salti di esecuzione nello stesso spazio di memoria dello script chiamante, per avere variabili locali devono esplicitamente dichiararle.

Le funzioni si invocano come gli script, con gli argomenti sulla riga di comando. Gli argomenti saranno disponibili nelle versioni locali delle variabili posizionali, senza intaccare quelle dello script chiamante, con l'eccezione di $0 che è immutato (mentre il nome della funzione invocata è in $FUNCNAME)

```
#!/bin/bash
f1 () {
   local a=10
   echo $0 $a $1 $FUNCNAME
}
f1 pippo                    →  func.sh 10 pippo f1
echo $a                     →  (riga vuota)
```

Gestione dei processi

Ogni comando lanciato da shell o dal sistema diviene un ***processo***. I processi sono identificati univocamente da un numero chiamato Process ID (PID)

Sorvegliare i processi e poterne bloccare, ripristinare, o alterare l'esecuzione (ad esempio alzandone o abbassandone la priorità) è uno dei compiti fondamentali dell'amministrazione del sistema.

Un processo svolge le proprie azioni a nome dell'utente che lo ha lanciato (i processi lanciati da root hanno il potere di assumere l'identità di altri utenti, così facendo si "declassano" e perdono il potere di tornare indietro)

Gestione dei processi

Il comando **ps** è il modo più semplice di ottenere l'elenco dei processi attivi.

I parametri più utili di **ps** sono:

ax visualizza tutti i processi, anche non propri
u mostra gli utenti proprietari del processo
w visualizza la riga di comando completa che ha originato il processo
f visualizza i rapporti di discendenza tra processi

Segnali

Uno dei meccanismi con cui i processi possono comunicare tra loro e con il s.o. è quello dei segnali.

Ogni processo può “registrare” presso il sistema operativo una routine di gestione (**handler**) per un segnale. Quando un altro processo invia il corrispondente segnale, il processo a cui è destinato viene sospeso e viene eseguito l’handler.

Diversi tipi di segnali hanno caratteristiche quali:

- comportamento predefinito in assenza di handler
- possibilità di essere ignorati
- comportamento imposto

Segnali

I segnali possono essere inviati dal s.o. per indicare situazioni d’errore o comunque degne di attenzione, ad esempio:

7	SIGBUS	Bus error
8	SIGFPE	Floating Point Exception
11	SIGSEGV	Segmentation Violation
13	SIGPIPE	Broken Pipe
14	SIGALRM	Alarm expired
17	SIGCHLD	Child ended

Segnali

… oppure possono essere originati da altri processi per comunicare al destinatario una richiesta (solo il proprietario di un processo e root possono inviargli segnali), ad esempio:

1	SIGHUP	Hangup
9	SIGKILL	Kill
15	SIGTERM	Terminate
18	SIGCONT	Continue if stopped
19	SIGSTOP	Stop process

Segnali

Normalmente l'invocazione di un comando da shell blocca la shell per tutto il tempo di esecuzione. In tale intervallo la shell intercetta la pressione di alcune combinazioni di tasti e invia al processo un segnale:

Ctrl + Z → SIGSTOP
Ctrl + C → SIGKILL

Per inviare esplicitamente un segnale ad un processo si usa il comando **kill** con primo argomento il segnale e secondo il PID

kill –SIGKILL 150

Job control

Si può usare un'unica shell per l'esecuzione contemporanea di più comandi che non abbiano necessità di accedere al terminale, lanciandoli in background (sullo sfondo).

Questo si ottiene postponendo il carattere **&** alla command line. La shell risponde comunicando un numero tra parentesi quadre (job id) che identifica il job localmente a questa shell.

Se si lancia una command line senza **&**, e si vuole rimediare, si può dare un segnale di **STOP** con Ctrl+Z. Anche in questo caso si riceve un job id. Con il comando **bg %***job_id*, si invia un segnale CONT che riavvia il processo e contemporaneamente lo si mette in background.

Job control

Un processo in background non riceve più comandi dal terminale, poiché la tastiera torna ad agire sulla shell; se è necessario riportare in foreground (primo piano) un processo ricollegandolo così al terminale, si usa il comando **fg %***job_id*.

Il comando **jobs** mostra l'elenco dei job, cioè di tutti i processi avviati dalla shell corrente, indicando il loro stato (attivo o stoppato).

nohup

Ogni comando lanciato in una shell da luogo ad un processo che, normalmente, viene terminato dall'invio del segnale SIGHUP al termine della sessione di lavoro (chiusura della shell). Se si desidera creare un processo che 'sopravviva' al termine della sessione (es. un download lungo) bisogna renderlo immune dall'hangup.

Questo si ottiene anteponendo il comando **nohup** che provvede, inoltre, a scollegare l'output del processo dal terminale se non fatto esplicitamente nell'invocazione.

Di default, nohup dirige l'output sul file 'nohup.out'

Gestori di segnale

Il builtin **trap** permette di definire un'azione personalizzata da eseguire alla ricezione di un segnale. Si noti che:

- alcuni segnali hanno handler non ridefinibili: STOP, CONT, KILL
- alcuni segnali hanno handler di default (tipicamente causano la terminazione): ABRT, HUP, INT, USR1, TERM
- esistono pseudo-segnali per il debugging degli script:
  - DEBUG è lanciato dalla shell prima di eseguire ogni comando
  - RETURN è lanciato dalla shell dopo il rientro da una chiamata a funzione o dopo l'inclusione di un file con **`source`**
  - ERR è lanciato dalla shell ad ogni comando che fallisce
  - EXIT è lancato dalla shell in uscita (sia causata da exit, fine script, o qualsiasi segnale di terminazione - tranne ovviamente KILL)

Sintassi:

```
trap [-lp] [codice da eseguire] [segnale [segnale]]
```

Gestori di segnale

Esempio: chiusura pulita di gerarchie di processi, supporto alla riapertura di file descriptor

```
#!/bin/bash
LOGFILE=/var/log/custom.messages.log
function logging () {
    case “$1” in
      start)       receive_messages > “$LOGFILE” & LOGGERPID=$! ;;
      stop)        test -n “$LOGGERPID” && kill “$LOGGERPID” ;;
      restart)     logging stop && logging start ;;
    esac
}

trap 'logging restart' USR1
trap 'logging stop' EXIT
logging start
while :
do
    something_interesting
    generate_lots_of_messages
done
```

Command substitution

È possibile valutare 'al volo' l'output di un comando racchiudendolo tra *backtick* (virgolette a rovescio `)
oppure tra $(e) - che a differenza del backtick sono annidabili

```
for filename in `ls p*`
do
    echo $filename
done
```

```
dottori=$(cat $(/usr/local/bin/getnames) | grep Dr. | sort -u)
```

Process substitution

La command substitution è utilizzabile solo quando i comandi producono output

- di dimensione limitata
- da processare al posto di parametri della riga di comando

Quando invece si vuole utilizzare l'output di un comando sotto forma di stream, ma in modo più flessibile rispetto alla pipeline lineare, si può ricorrere alla ***process substitution***:

```
cmp <(prog1) <(prog2)
```

Comando che si aspetta due file come parametri

La shell fa trovare a cmp due fd aperti, su cui riversa l'output di prog1 e prog2

```
ls | tee >(grep foo | wc >foo.count) \
         >(grep bar | wc >bar.count) \
         | grep baz | wc >baz.count
```

al posto dei file la shell apre dei fd che vanno ad alimentare i processi indicati

tee copia stdin su stdout e su ogni file dato come argomento

Eval

Il builtin **`eval`** permette di processare un file come se fosse uno script, sottoponendolo ai 12 passi di valutazione elencati in precedenza. Questo permette ad uno script di generare altri script ed eseguirli correttamente.

Es:

```
$ listpage="ls | more"
$ $listpage
ls: cannot access |: No such file or directory
ls: cannot access more: No such file or directory
$ eval $listpage
```

… elenco file paginato …

La parameter expansion avviene dopo la tokenization... "|" e "more" appaiono quindi troppo tardi per essere interpretati come token generici e non solo come argomenti di ls

Eval

Una delle funzionalità più utili, dopo la ri-valutazione dei separatori, è far comparire variabili e forzarne l'espansione.
Bisogna, come sempre, porre molta attenzione all'escaping dei metacaratteri per inibire la loro interpretazione al primo passaggio di espansione ed eventualmente abilitarli al secondo.
Esempio:

```
$ A=ciao
$ eval "P=$A ; echo $P"

$ unset P
$ eval "P=$A ; echo \$P"
ciao
```