Programmazione Batch

Bash Shell

Roberto Sassi

http://www.dti.unimi.it/~sassi/

DTI Crema - Università degli Studi di Milano

Outline

- Programmazione Batch e Linguaggio di Script
- Scripts e shells
- Esecuzione di uno script
- Elementi di uno script:
  - variabili e parametri
  - quoting e reindirizzamento
  - comandi esterni
  - strutture condizionali e test (`case`)
  - strutture cicliche
  - debugging
  - interfaccia utente
- CGI con BASH

Linguaggio di Script/I

- Un linguaggio di script è in generale un linguaggio di programmazione interpretato e utilizzato per automatizzare operazioni di routine.
- In questa carrellata ci riferiamo in particolare ai linguaggi di script nativi alla shell di un sistema operativo.
- Shell $\equiv$ Interprete dei comandi. COMMAND.COM in DOS, CMD.exe in Windows 2000/XP e Bourne/C/Korn/... Shell in UNIX/Linux.

Linguaggio di Script/II

- Il “programma” è contenuto in un file di testo che l’interprete (la shell) processa (“‘parsing”) ed esegue riga per riga. Il linguaggio di script dipende dal tipo di interprete utilizzato.
- Ottimo per ottenere risutati velocemente.
- Ad esempio in DOS/Windows (file “leggi.BAT”):

```
@ECHO OFF
IF !%1==! GOTO MESSAGGIO
REM MOSTRA IL FILE PAGINA PER PAGINA
TYPE %1 | MORE
GOTO EXIT0
:MESSAGGIO
ECHO Non hai specificato il file!
:EXIT0
```

Linguaggio di Script/III

- In Unix/Linux, l'utente disponde nativamente di vari *script languages* per realizzare piccoli programmi costituiti da semplici file di testo
- Per specificare quale interprete deve eseguire lo script, la prima riga contiene sempre il nome e il percorso del programma a cui ci si riferisce. Ad es:
  `#!/bin/sh`
  `#!/bin/perl`
  Il carattere `#!` si chiama "Magic Number" o anche "sha-bang"

Tipi di Script

- In genere lo script language contiene la maggiorparte degli costrutti tipici di un linguaggio di programmazione
- Oltre agli script di shell, sono disponibili anche script di altri programmi, tra cui awk, perl e tcl (questo script, mediante il front-end grafico tk, consente di realizzare velocemente applicazioni per X)
- La shell principalmente utilizzata in Linux è `bash` (Bourne Again Shell) che offre numerose funzionalità (quella di riferimento nel seguito)
- **`#!/bin/sh`** è link simbolico a bash (POSIX)
- Storicamente un'altra shell molto famosa è la csh/tcsh, che prevede una sintassi degli script molto simile alla sintassi del C

Esecuzione di uno script/I

- Esistono tre modi per eseguire uno script
- La più semplice è passare lo script come argomento all'interprete, ad esempio gli script di shell possono essere eseguiti con `/bin/sh script`
- La soluzione più sofisticata è specificare il tipo di interprete all'interno dello script, rendere eseguibile lo script con `chmod +x script`, chiamare lo script con il suo nome, come un qualsiasi altro comando

Esecuzione di uno script/II

- ATTENZIONE: La shell quando si esegue uno script con i 2 metodi precedenti, fa una FORK e crea un processo shell CHILD a cui affida il compito di eseguire lo script (il processo PARENT entra in stato di WAIT). Se si vuole che lo script sia eseguito dal processo padre si deve invocare lo script diversamente:

```
.  script
source script
```

Utile per modicare le variabili di ambiente!

Sintassi - intro

- Uno script di shell = un file di testo che contiene una serie di comandi e di costrutti di shell
- Per ulteriori informazioni sulla sintassi e sui comandi di shell riferirsi alla man-page di bash
- Convenzione: prima riga inizia con **`#!/bin/sh`**
- # carattere usato per i commenti
- Utilizzo: 1) macro per lanciare successione di comandi 2) veri e propri programmi con parametri da linea di comando e variabili interne

Esempio di confronto

- Il file batch DOS/Windows “leggi.BAT” diventerebbe “leggi.sh”:

```
#!/bin/bash
if [ $# ! -ne 0 ]
then
# MOSTRA IL FILE PAGINA PER PAGINA
cat $1 | more
else
echo Non hai specificato il file!
fi
exit 0
```

Sintassi - variabili

- variabili = stringhe alfanumeriche il cui valore è identificato con `$var`
- variabili *case sensitive*. `var1` diversa da `Var1`
- `var` è il nome della variabile, `$var` il suo contenuto!!!
- inizializzazione: `var=stringa`, `var=$altravar` [senza spazi tra le stringhe e il carattere '=']
- la stringa deve essere racchiusa tra virgolette (o apostrofi) se contiene spazi; i caratteri speciali vanno preceduti da \
- È possibile accedere alle variabili d'ambiente (le stesse che vengono visualizzate dal comando **env**); **`PATH`**, **`PWD`**

Sintassi - variabili (Bash 2.0)

Dalla versione 2 di Bash è possibile dichiarare le variabili per tipo attraverso il comando `declare`.

- `-i integer`, `-a array`, `-r readonly`
- `-f functions`: dichiarazioni di nomi di funzioni
- `-x export`: la variabile dichiarata con l'opzione **x** sarà esportata all'esterno dell'ambiente dello script
- **Esempio:**
  `declare -i counter` è una variabile intera
- Se non dichiarato diversamente le variabili sono considerate come stringhe

Esempio

```
#!/bin/bash
# ATTENZIONE: NO spazi intorno a = !
a=375
hello=$a
echo hello
# Stampa: la stringa "hello".
echo "$hello"
# Stampa: 375 (uguale a echo "${hello}")
# Assegna il valore nullo
hello=
# Disalloca la variabile
unset hello
```

Sintassi - parametri

- I parametri dello script possono essere letti mediante i simboli $\$n$ (con n intero positivo)
- $0 contiene il nome dello script, $1 contiene il primo parametro, $# contiene il numero di parametri e $* contiene tutta la riga dei parametri
- Ad esempio, con la riga di comando **`script par1 par2`**, i parametri assumeranno i seguenti valori: 0=script, 1=par1, 2=par2, #=2 e *="par1 par2"

Sintassi - quoting - 1

- *quoting* = l'uso di apici e virgolette per racchiudere stringhe
- Per visualizzare una frase si possono usare indistintamente due comandi:
  - `echo "frase"`
  - `echo 'frase'`
- si usano le virgolette se la frase contiene degli apostrofi; si usa l'apostrofo se la frase contiene le virgolette

Sintassi - quoting - 2

- **NB**: tutto ciò racchiuso da una coppia di accenti gravi (“‘”) viene interpretato come un comando; in questo modo è possibile assegnare ad una variabile il risultato di un comando
- per inserire caratteri speciali in stringhe o espressioni bisogna utilizzare il carattere di escape "\", usato anche per andare **a capo** su una nuova riga senza interrompere il comando
- Esempio:

```
#!/bin/bash
# Calcola la somma di tre numeri
echo "La somma e’ ‘expr $1 + $2 + $3‘."
```

Sintassi - reindirizzamento - 1

- Ad ogni comando che viene eseguito dallo script, vengono *aperti* tre flussi (`stdin`, `stdout` e `stderr`)
- la shell consente un facile reindirizzamento degli stessi verso altri comandi e verso file
- Nella bash i tre flussi vengono numerati rispettivamente con 0, 1 e 2
- Esempi
  - `comando 1> output`
  - `comando 2> errore`
  - `comando 0< input`

Sintassi - reindirizzamento - 2

- per utilizzare sia il flusso 1 che il flusso 2 si utilizza "&>"
- per aggiungere in coda ad un file si utilizza il ">>"
- la *pipe* (indicata con il carattere "|") concatena lo standard output di un programma con lo standard input di un altro programma

Sintassi - comandi esterni - 1

- È possibile usare comandi di sistema (o altri script) o direttamente o mediante il quoting (con il carattere "`")
- Quando un comando chiamato direttamente termina la sua esecuzione restituisce il controllo allo script; per sapere se il comando è stato eseguito correttamente, è disponibile una variabile particolare (indicata con **?**) che contiene il codice di uscita dell'ultimo comando
- il test **`[$? = 0]`** verifica che non vi siano stati errori (per convenzione i comandi restituiscono un valore diverso da 0 quando si è riscontrato un qualche problema)

Sintassi - comandi esterni - 2

- Per fornire condizioni d'uscita diverse da 0 si può usare il comando **`exit num`**, che sospende l'esecuzione dello script e restituisce il valore num (nella variabile **`?`**).
- I comandi richiamati mediante quoting, vengono impiegati per ottenere informazioni a run-time
- A esempio, per visualizzare la versione del kernel è possibile utilizzare il seguente comando:

```
echo "La versione del kernel è
‘/bin/uname -a | /bin/cut -d ’ ’ -f3‘."
```

Sintassi - comandi esterni - 3

- Notare l'utilizzo del comando **`cut`**, come filtro (di colonna) per selezionare solo i valore desiderato
- Per consentire la comunicazione tra script diversi (o tra script e comandi) è possibile utilizzare i comandi **`trap`** (per accettare un segnale) e **`kill`** (per inviare un segnale).
- Specificando in uno script l'istruzione

  `trap "comando" segnale`

  lo script eseguirà **`comando`** non appena **`segnale`** viene ricevuto dallo script

Sintassi - strutture condizionali - 1

- costrutto **`if then else fi`**
- nidificazione **`elif`**, per evitare di generare troppe coppie **`if fi`**
- per uscire direttamente dal costrutto si utilizza il **`break`**
- l'argomento di **`if`** è un "test" ossia un'espressione che restituisca un valore vero o falso
- nella bash si utilizzano le parentesi quadre per racchiudere l'espressione del test (altre shell utilizzano il comando esterno test). Esistono numerosi tipi di test, negli esempi vedremo alcuni esempi di condizioni su file e su variabili

Sintassi - strutture condizionali - 2

- eseguire il "demone del kernel" (per la gestione automatica dei moduli), solo se il comando è presente

```
if [ -x /sbin/kerneld ] # -x vero solo se il file esiste
then
   /sbin/kerneld
else
   echo "Non e’ possibile eseguire kerneld"
fi
```

Sintassi - Test su file, numeri e stringhe - 1

- **Documentazione**: **`man test`**
- Esempi:
  - **`-x`** vero solo se il file esiste
  - **`-s`** vero solo se il file esiste e non è vuoto
  - **`-d`** vero solo se il file è una directory
  - **`-f`** vero solo se il file è realmente un file
  - **`-r`** vero solo se si hanno i diritti in lettura al file
  - **`-w`** vero solo se si hanno i diritti in scrittura al file

Sintassi - Test su file, numeri e stringhe - 2

- sono disponibili test specifici per variabili numeriche
  - **`[$numero -eq 0]`** vero se uguale (si può usare anche **`=`**)
  - **`-gt`** vero se maggiore
  - **`-lt`** vero se minore
  - **`-ge`** vero se maggiore o uguale
  - **`-le`** vero se minore o uguale

Sintassi - Test su file, numeri e stringhe - 3

- test specifici per le variabili stringa sono invece:
  - l'uguaglianza (**=**)
  - la disuguaglianza (**!=**)
  - verifica sulla lunghezza della stringa (**-z** per verificare la lunghezza nulla e **-n** per verificare che la stringa sia non vuota)

Sintassi - Operazioni numeriche - 1

Il comando base per effettuare le operazioni su numeri è il seguente:

- comando esterno **`expr`**. Esempio:
  - **`count=‘expr $count + 2‘`**
- **NB:** *rispettare gli spazi prima e dopo il segno* +

Sintassi - Operazioni numeriche - 2

Con Bash 2.0 esiste il comando `let` per svolgere operazioni matematiche. **Esempi:**

- `let a=11`, assegnamento
- `let a = a + 5` oppure `let a += 5`
- `let "a = a / 4"` oppure `let "a /= 4"`
- `let "a -= 5"` oppure `let "a = a - 5"`
- `let "a *= 10"` oppure `let "a = a * 10"`
- `let "a %= 8"` oppure `let "a = a % 8"`

NB: l'operatore **%** è il modulo

Sintassi - Operazioni numeriche - 3

Altro operatore che può essere estremamente utile è il **Double Parentheses Construct** secondo lo **C-style**. Esempi:

- `$(( a = 23 ))`, assegnamento
  NB: spazio prima e dopo =
- `$(( a++ ))`, Post-incremento della variabile
- `$(( a- ))`, Post-decremento della variabile
- `$(( ++a ))`, Pre-incremento della variabile
- `$(( -a ))`, Pre-decremento della variabile
- `$(( t = a<45?7:11 ))`, assegnamento condizionato

Esercizio/I

Esempio:

- `max`: script che riceve **due** numeri e restituisce a schermo il valore massimo

```
#!/bin/bash
if [ $1 -gt $2 ] #Spazi prima e dopo!
then
    echo $1
else
    echo $2
fi
```

Oppure

```
#!/bin/bash
echo $((risultato = $1>$2 ? $1 : $2))
```

Esercizio/II

Esempio:

- `sum`: script che riceve **tre** numeri e ne restituisce la somma. Avevamo già visto

```
#!/bin/bash
# Calcola la somma di tre numeri
echo "La somma e' 'expr $1 + $2 + $3'."
```

Oppure

```
#!/bin/bash
echo "La somma dei tre numeri e" \
    $(($1 + $2 + $3)) "."
```

Sintassi - strutture condizionali - 3

- altro costrutto condizionale: **`case esac`**,
- utile nel caso in cui la struttura **`if then elif then else fi`** diventi troppo sviluppata

```
case stringa in
stringa1)
   se stringa = stringa1 esegui comandi fino al doppio ";"
   ed ignora tutti gli altri casi;;
stringa2)
   se stringa = stringa2 esegui comandi fino al doppio ";"
   ed ignora tutti gli altri casi;;
stringa3)
   se stringa = stringa3 esegui comandi fino al doppio ";"
   ed ignora tutti gli altri casi;;
esac
```

Esercizio/III

Esempio: Utilizzo di TRAP:

```
#!/bin/bash
trap 'echo Thank you for playing' EXIT
# RANDOM: numero casuale tra 0 32767
# >&2 redirige lo stdout allo stderr
magicnum=$(($RANDOM%10+1))
echo 'Guess a number between 1 and 10:'
while echo -n 'Guess: ' >&2 ; read guess; do
    sleep 0.2
    if [ "$guess" = $magicnum ]; then
        echo 'Right!'
        exit
    fi
    echo 'Wrong!'
done
```

Sintassi - strutture cicliche - 1

- per realizzare dei cicli all'interno di script di shell si utilizza principalmente il costrutto **`for do done`**
- **Esempio**: abilitare la tastiera numerica (BLOCK NUM) su tutte le console virtuali

```
for t in 1 2 3 4 5 6
  do
  setleds +num < /dev/tty$t > /dev/null
done
echo $PWD
```

Sintassi - strutture cicliche - 2

Nella Bash 2.0 ci sono ulteriori modi di scrivere un ciclo `for` usando il C-style. Esempi:

- `for ((a=1; a <= LIMIT ; a++))`
- `for ((a=1, b=1; a <= LIMIT ; a++, b++))`

Sintassi - strutture cicliche - 3

- Un altro costrutto per realizzare i cicli è **`while do done`**
- In questo caso l'argomento di **`while`** è un "test" tipico dei costrutti **`if`**
- Il costrutto analogo, ma con il test invertito è **`until do done`**
- in entrambi i casi il test è sempre all'inizio del ciclo **`do done`**

Sintassi - strutture cicliche - 4

Esempio until do:

```
if [ $# -eq 0]; then echo "Mancano argomenti..."; fi
until [ $# -eq 0]; do      # "#" contiene numero di parametri passati
  if [ ! -s $1]
     then echo "File $1 non esistente"
  else
     comandi_specifici
  fi
  shift                    # sposta var a sinistra (1=$2, 2=$3, ...)
done
```

Sintassi - strutture cicliche - 5

Esempio while:

```
#!/bin/bash

var0=0
LIMIT=10

while [ "$var0" -lt "$LIMIT" ]
do
  echo -n "$var0 "        # -n suppresses newline.
  var0=‘expr $var0 + 1‘   # var0=$(($var0+1)) also works.
done

echo

exit 0
```

Esempio: for e case

Esempio:

```
for file in ‘ls . ‘ ; do  # per ogni elemento della dir corrente
  case $file in           # visualizza un messaggio appropriato
    *.gif|*.jpg) echo "$file: file grafico" ;;
    *.txt|*.text) echo "$file: file di testo" ;;
    *.c|*.f|*.for) echo "$file: file sorgente" ;;
    *) echo "$file: file generico" ;;
  esac
  echo $PWD
done
```

Esercizio: testo

Scrivere uno script che compie le seguenti operazioni:

- riceve due parametri, `$1` e `$2` che corrispondono rispettivamente all'operatore in modulo e a un numero qualsiasi minore di `$1`
- lo script esce con errore se `$1` minore o uguale di `$2`
- calcola il numero `num` tra 0 e `$1` la cui seguente operazione in modulo (`num % $1`) è uguale a `$2` e lo stampa a schermo

Esercizio: Soluzione

```
#!/bin/bash
if [ $1 -le $2 ]
then
    echo "Il secondo parametro deve essere minore del primo\n"
    exit 1
fi
num=0
while [ ‘expr $num % "$1"‘ -ne "$2" ]
do
    let num++
    if [ "$num" -gt "$1" ]
    then
    echo "Numero non trovato"
    exit 2
    fi
done
echo "Il numero che cercavi {\‘e} $num"
exit 0
```

Debugging

- Per **collaudare** gli script di shell sono disponibili due opzioni
- **`-v`**, modalità verbose: visualizza ogni comando prima di eseguirlo
- **`-x`**, modalità execute: visualizza ogni comando eseguito con le relative variabili ad esso associate
- In entrambi i casi per eseguire lo script è necessario utilizzare la forma:
  - **`sh -v script`**
  - **`sh -x script`**

Interfaccia Utente

- Benché non sia prevista nell'ambiente di shell, è possibile realizzare un'interfaccia utente mediante il comando esterno **`dialog`** (non molto veloce ma il risultato è più che soddisfacente)
- per visualizzare il contenuto di un file in una finestra in attesta di una conferma: **`dialog -title "Finestra informativa" -textbox "file-name" 22 77`**
- mostra una finestra di conferma (yes/no), la variabile **`$?`** vale 0 solo in caso di yes: **`dialog -title "Confermare" -clear -yesno "Siete sicuri?" 10 30`**

Per saperne di più

- il filesystem di Linux è pieno di esempi di script di shell
- i file rc (eseguiti durante l'inizializzazione del sistema in **/etc/rc.d/**)
- i file profile (che contengono la configurazione della bash e si trovano in **/etc/profile** e **~/.profile**)
- esistono numerosi testi su Unix che contengono informazioni utili (ed esempi) per la programmazione di shell; ad esempio un testo per chi vuole iniziare è il libro di Henry McGilton e Rachel Morgan "Il sistema operativo Unix", edito dalla McGraw-Hill
- riviste specializzate

Documentazione on line:

- **Articolo su Pluto Linux Journal**:
  www.pluto.linux.it/journal/pj9705/script.html
- **Bash Guide for Beginners**:
  http://www.tldp.org/LDP/Bash-Beginners-Guide/Bash-Beginners-Guide.pdf
- **Advanced Bash-Scripting Guide**:
  http://www.tldp.org/LDP/abs/abs-guide.pdf
- **BASH reference manual (CHM)**:
  http://htmlhelp.berlios.de/books/download/bashref-2.05b.chm

Gli script di shell sono stati una delle prime forme di programmi CGI. Ad esempio un semplice contatore:

```
#!/bin/bash
data="6 Novembre 2003"
test -f contatore.txt || echo ’0’ > contatore.txt
visitatori=‘cat contatore.txt‘
visitatori=‘expr $visitatori + 1‘
echo "$visitatori" > contatore.txt

echo "Content-type: text/html"
echo ""
echo "<br />$visitatori visitatori dal $data.<br />"
# per includerlo nel file shtml:
# <!--#include virtual="/cgibin/conta.cgi" -->
```

CGI/lla

Risposta a form HTML:

```
<html>
<head>
<title>Usiamo una CGI BASH</title>
<body>
Come ti chiami?
<form action="/cgi-bin/nome.cgi" method="GET">
<input type="text" name="nome">
<input type="submit" value="invia">
</form>
</body>
```

Lo script nome.cgi:

```
#!/bin/bash
nome=${QUERY_STRING:6}
echo "Content-type: text/html"
echo ""
echo -e "<html>\n<head>"
echo -e "<title>Saluto</title>\n</head><body>"
echo "<br />Ciao $nome, Benvenuto!<br />"
echo -e "</body>\n</html>"
```

CGI/llc

Lo script nome.cgi, produce sullo stdout (supponiamo che l'utente abbia digitato "Mario"):

```
Content-type: text/html

<html>
<head>
<title>Saluto</title>
</head><body>
<br />Ciao Mario, Benvenuto!<br />
</body>
</html>
```

Esercizio: testo

- scrivere lo script **`clean-core`** che compia le seguenti operazioni:
  - cerca tutti i file che iniziano per **`core`** nella propria home
  - cancellarli uno ad uno chiedendo conferma all'utente
  - alla fine lo script deve produrre il numero di file trovati
- Hints:
  - usare comando **`find`**
  - usare comando **`grep`**
  - usare apici gravi per assegnare risultato ricerca

Esercizio: soluzione 1

```
#!/bin/sh

home=/home/roberto/

echo "Numero core trovati ‘find $home -name "core*" | grep -c core‘"

for file in ‘find /home/roberto/ -name "core*"‘
do
    if [ -n file ]; then
        /bin/rm -iv $file
    fi
done
```

Esercizio: soluzione 2

```
#!/bin/sh

# $HOME variabile globale (vedi comando env)

counter=0

for file in ‘find $HOME -name "core*"‘
do
    if [ -n file ]; then
        /bin/rm -iv $file
    fi
    counter=‘expr $counter + 1‘
done

echo "Numero core trovati " $counter
```

Esercizio: soluzione parametrica

- Soluzione precedente per ricercare i file in modo parametrico (tutti i file che contengono una certa stringa data in input)
- **NB:** lo script è stato modificato (mostra il file trovato anziché cancellare) per evitare di cancellare file importanti

```
#!/bin/sh
stringa=$1
counter=0
for file in ‘find $HOME -name "*$stringa*"‘
do
    if [ -n file ]; then
        echo "Trovato file " $file
    fi
    counter=‘expr $counter + 1‘
done
echo "Numero file contenenti " $stringa " trovati " $counter
```