Archiviare e comprimere i file

gzip *<file>* comprime il file e produce *<file.gz>*

gunzip *<file.gz>* decomprime un file compresso con `gzip` e restituisce il file originario

gzip è compatibile con zip/pkzip/winzip, ma comprime un solo file alla volta

bzip2 *<file>* comprime il file e produce *<file.bz2>*

bunzip2 *<file.bz2>* decomprime un file compreso con `bzip2` e restituisce il file originario

bzip2 come `gzip` comprime un solo file alla volta; `bzip2` *non* è compatibile con gli altri formati ed è più lento di `gzip`, ma molto spesso comprime meglio (es.: sul computer su cui sto scrivendo `/var/log/warn` è lungo 607629 byte; comprimendolo con `gzip` ottengo `warn.gz` di 45229 byte, mentre usando `bzip2` ottengo `warn.bz2` di 32809 byte (circa il 27% in meno)

tar cvfz *<archivio.tar.gz>* *<file ...>* crea l'archivio *<archivio.tar.gz>*, compresso con `gzip` contenente i file e/o le directory indicate

```
ls
schede2.aux schede2.pdf schede.aux schede.pdf
schede2.log schede2.tex schede.log schede.tex
tar cvfz schede.tar.gz schede*tex schede*pdf
schede2.tex
schede.tex
schede2.pdf
schede.pdf
ls
schede2.aux schede2.pdf schede.aux schede.pdf schede2.log
schede2.tex schede.log schede.tex schede.tar.gz
```

tar xvfz *<archivio.tar.gz>* inverte l'operazione precedente, ripristinando la struttura di file e directory contenuta nell'archivio

il significato dei comandi dati a `tar`:

- **c** create: crea un nuovo archivio
- **x** extract: estrae i file da un archivio
- **t** test: controlla il contenuto dell'archivio (in pratica si usa per sapere quali file sono contenuti nell'archivio senza decomprimerlo)
- **v** verbose: racconta quello che sta facendo (su quali file sta lavorando)
- **f** file name: il prossimo parametro è il nome dell'archivio su cui operare[1]
- **z** gzip: usa **gzip** per (de)comprimere l'archivio
- **j** (al posto di **z**): usa **bzip2** per (de)comprimere l'archivio; gli archivi compressi con `bzip2` di solito prendono nomi del tipo *<archivio.tar.bz2>*

note

1. spesso gli archivi compressi con `gzip` prendono nomi del tipo *<archivio.tgz>*
2. differenza fondamentale rispetto a `zip`: i file vengono prima riuniti in un archivio e poi l'archivio viene compresso; `zip` invece comprime ciascun file separatamente e poi riunisce i file compressi in un archivio; anche se gli algoritmi sono esattamente gli stessi, il metodo usato da `tar` di solito dà risultati migliori
3. un archivio prodotto da `tar` viene spesso chiamato *tarball*

[1] **tar** può lavorare anche senza conoscere il nome dell'archivio su cui lavora...

Shell script

Tutte le operazioni che abbiamo visto fin qui possono essere eseguite anche dall'ambiente grafico. Perché allora usare la shell?

1. perché è più *veloce*: quando si impara a usarla è molto più rapido e semplice scrivere un comando, magari formato da 2 caratteri, che selezionare una finestra, poi aprire un menu, poi trovare il comando...
2. perché è più *leggera*: un comando come **`ls`** eseguito in una directory con alcune migliaia di file è ragionevolmente veloce anche su un computer di 10 anni fa; una finestra con migliaia di icone, magari ciascuna con il *preview* del file, può mettere in crisi una workstation moderna
3. perché è uno standard: i principali comandi della shell sono (quasi) gli stessi su qualunque computer (dal palmare al supercomputer) con qualunque variante di Unix e affini
4. perché è **molto più potente**: i comandi possono essere combinati tra loro per ottenere nuove funzionalità

Grazie alla shell possiamo di fatto creare nuovi comandi, senza la necessità di scrivere nuovi programmi ma semplicemente combinando quelli esistenti grazie al «linguaggio» (semplice e minimale) messo a disposizione dalla shell (*shell script*).

Uno script è semplicemente un file di testo contenente i comandi che devono essere eseguiti. La prima riga è speciale: inizia sempre con i caratteri **`#!`** seguiti dal nome dell'*interprete* che eseguirà i comandi. Nel nostro caso l'interprete sarà sempre la shell che abbiamo imparato a usare, quindi la prima riga sarà sempre[2]

```
#!/bin/sh
```

Per eseguire uno script dobbiamo

1. renderlo eseguibile, cosa che si ottiene con il comando **`chmod +x <nome dello script>`**
2. copiarlo in una directory in cui la shell «sa» che ci sono file eseguibili: a questo punto possiamo eseguirlo semplicemente scrivendo il suo nome, come facciamo per qualsiasi altro comando. Alternativamente possiamo chiamarlo con il *percorso completo*; ad es. se lo script è nella directory corrente, potremo chiamarlo con **`./<nome dello script>`**

Un semplicissimo script

Abbiamo visto che **`rm`** cancella i file senza possibilità di recuperarli. Vediamo un piccolo script che invece li sposta in una directory «cestino» da cui poterli recuperare.

Sono 50 righe, delle quali 8 sono commenti e 6 sono bianche; restano 36 righe di «codice», che tra l'altro compila un elenco dei file cancellati, li rinomina in modo da poter tenere nel cestino più versioni dello stesso file (o più file con lo stesso nome). Inoltre segnala un eventuale uso scorretto dello script e fornisce una riga di *help* e legge (ed eventualmente crea) un file di configurazione che ci permette di scegliere dove collocare il nostro «cestino». Senza questi *optional* avremmo potuto ridurlo a 20 righe, addirittura a 17 accettando di spostare nel cestino un solo file alla volta. Quante righe di C sarebbero necessarie per ottenere lo stesso risultato?

(Nota: nella prima riga abbiamo scritto **`#!/bin/sh -x`**. Il parametro **`-x`** costringe `/bin/sh` a stampare i comandi che sta per eseguire: è molto utile per capire cosa sta succedendo)

[2] Una convenzione dei sistemi Unix è che quando un file eseguibile inizia con **`#!`**, la prima riga è il nome di un programma che viene eseguito e a cui viene fornito in input il testo completo del file. Quindi se all'inizio di un file di testo aggiungiamo **`#!/usr/bin/less`** e poi lo rendiamo eseguibile, otteniamo...

```
#!/bin/sh -x
#
# trash.sh
#
# moves file(s) to trash directory
# DEMO/TUTORIAL ONLY -- DO NOT USE IN ANY OTHER ENVIRONMENT
#
# pc20050217

if [ $# -eq 0 ] ; then
 self=$(basename $0)
 echo "Usage: $self filename [filename...]"
 exit 1
fi

trashrc="${HOME}/.trashrc"
if [ -r $trashrc ] ; then
 . $trashrc
else
 cat > $trashrc <<-EOF
        TRASHDIR="${HOME}/.trash.d"
        TRASHINFO="${HOME}/.trash.d/.trashinfo"
        EOF
 . $trashrc
fi

if [ ! -d ${TRASHDIR} ] ; then
 mkdir ${TRASHDIR}
 if [ $? -ne 0 ] ; then
  echo "mkdir ${TRASHDIR} failed!"
  exit 2
 fi
fi

until [ -z $1 ] ; do
 file=$1
 if [ ! -f ${file} ] ; then
  echo "${file} is not a regular file!"
  exit 3
 fi
 filename=$(basename $1)
 junkfile=$(mktemp ${TRASHDIR}/${filename}_XXXXXX)
 junkname=$(basename $junkfile)
 echo -ne "$(pwd)\t${file}\t${junkname}\n" >> ${TRASHINFO}

 mv ${file} ${junkfile}
 shift
done

#EOF
```