

# Redirezione I/O

## Sommario

19.1. [Uso di `exec`](#)

19.2. [Redirigere blocchi di codice](#)

19.3. [Applicazioni](#)

In modo predefinito, ci sono sempre tre "file" aperti: lo `stdin` (la tastiera), lo `stdout` (lo schermo) e lo `stderr` (la visualizzazione a schermo dei messaggi d'errore). Questi, e qualsiasi altro file aperto, possono essere rediretti. Redirezione significa semplicemente catturare l'output di un file, di un comando, di un programma, di uno script e persino di un blocco di codice presente in uno script (vedi [Esempio 3-1](#) e [Esempio 3-2](#)), ed inviarlo come input ad un altro file, comando, programma o script.

### Esempio 3-1. Blocchi di codice e redirezione I/O

```
#!/bin/bash
# Legge le righe del file /etc/fstab.

File=/etc/fstab

{
read riga1
read riga2
} < $File

echo "La prima riga di $File è:"
echo "$riga1"
echo
echo "La seconda riga di $File è:"
echo "$riga2"

exit 0

# Ora, come sarebbe possibile verificare i diversi campi di ciascuna
# riga?
# Suggerimento: usate awk.
```

### Esempio 3-2. Salvare i risultati di un blocco di codice in un file

```
#!/bin/bash
# rpm-check.sh

# Interroga un file rpm per visualizzarne la descrizione ed il
#+contenuto, verifica anche se può essere installato.
# Salva l'output in un file.
#
# Lo script illustra l'utilizzo del blocco di codice.

SUCCESSO=0
E__ERR_ARG=65

if [ -z "$1" ]
then
    echo "Utilizzo: `basename $0` file-rpm"
    exit $E__ERR_ARG
```

```

fi

{
    echo
    echo "Descrizione Archivio:"
    rpm -qpi $1          # Richiede la descrizione.
    echo
    echo "Contenuto dell'archivio:"
    rpm -qpl $1          # Richiede il contenuto.
    echo
    rpm -i --test $1      # Verifica se il file rpm può essere installato.
    if [ "$?" -eq $SUCCESO ]
    then
        echo "$1 può essere installato."
    else
        echo "$1 non può essere installato."
    fi
    echo
} > "$1.test"           # Redirige l'output di tutte le istruzioni del
blocco                   #+ in un file.

echo "I risultati della verifica rpm si trovano nel file $1.test"

# Vedere la pagina di manuale di rpm per la spiegazione delle opzioni.

exit 0

```

Ad ogni file aperto viene assegnato un descrittore di file. [\[1\]](#) I descrittori di file per `stdin`, `stdout` e `stderr` sono, rispettivamente, 0, 1 e 2. Per aprire ulteriori file rimangono i descrittori dal 3 al 9. Talvolta è utile assegnare uno di questi descrittori di file addizionali allo `stdin`, `stdout` o `stderr` come duplicato temporaneo. [\[2\]](#) Questo semplifica il ripristino ai valori normali dopo una redirectione complessa (vedi [Esempio 19-1](#)).

```

OUTPUT_COMANDO >
    # Redirige lo stdout in un file.
    # Crea il file se non è presente, in caso contrario lo sovrascrive.

    ls -lR > dir-albero.list
    # Crea un file contenente l'elenco dell'albero delle directory.

: > nomefile
    # Il > svuota il file "nomefile", lo riduce a dimensione zero.
    # Se il file non è presente, ne crea uno vuoto (come con 'touch').
    # I : servono da segnaposto e non producono alcun output.

> nomefile
    # Il > svuota il file "nomefile", lo riduce a dimensione zero.
    # Se il file non è presente, ne crea uno vuoto (come con 'touch').
    # (Stesso risultato di ": >", visto prima, ma questo, con alcune
    #+ shell, non funziona.)

OUTPUT_COMANDO >>
    # Redirige lo stdout in un file.
    # Crea il file se non è presente, in caso contrario accoda l'output.

```

```

# Comandi di redirectione di riga singola
#+ (hanno effetto solo sulla riga in cui si trovano):
#-----

1>nomefile
# Redirige lo stdout nel file "nomefile".
1>>nomefile
# Redirige e accoda lo stdout nel file "nomefile".
2>nomefile
# Redirige lo stderr nel file "nomefile".
2>>nomefile
# Redirige e accoda lo stderr nel file "nomefile".
&>nomefile
# Redirige sia lo stdout che lo stderr nel file "nomefile".

M>N
# "M" è un descrittore di file, preimpostato a 1 se non impostato
esplicitamente.
# "N" è un nome di file.
# Il descrittore di file "M" è rediretto nel file "N."
M>&N
# "M" è un descrittore di file, se non impostato corrisponde a 1.
# "N" è un altro descrittore di file.

#=====

# Redirigere lo stdout una riga alla volta.
FILELOG=script.log

echo "Questo enunciato viene inviato al file di log, \
\"$FILELOG\"." 1>$FILELOG
echo "Questo enunciato viene accodato in \"$FILELOG\"." 1>>$FILELOG
echo "Anche questo enunciato viene accodato in \"$FILELOG\"." 1>>$FILELOG
echo "Questo enunciato viene visualizzato allo \
stdout e non comparirà in \"$FILELOG\"."
# Questi comandi di redirectione vengono automaticamente "annullati"
#+ dopo ogni riga.

# Redirigere lo stderr una riga alla volta.
FILEERRORI=script.err

comando_errato1 2>$FILEERRORI          # Il messaggio d'errore viene
                                         #+ inviato in $FILEERRORI.
comando_errato2 2>>$FILEERRORI          # Il messaggio d'errore viene
                                         #+ accodato in $FILEERRORI.
comando_errato3                          # Il messaggio d'errore viene
                                         #+ visualizzato allo stderr,
                                         #+ e non comparirà in $FILEERRORI.
# Questi comandi di redirectione vengono automaticamente "annullati"
#+ dopo ogni riga.
#=====

2>&1
# Redirige lo stderr allo stdout.
# I messaggi d'errore vengono visualizzati a video, ma come stdout.

```

```

i>&j
# Redirige il descrittore di file i in j.
# Tutti gli output del file puntato da i vengono inviati al file
#+ puntato da j.

>&j
# Redirige, per default, il descrittore di file 1 (lo stdout) in j.
# Tutti gli stdout vengono inviati al file puntato da j.

0< NOMEFILE
< NOMEFILE
# Riceve l'input da un file.
# È il compagno di ">" e vengono spesso usati insieme.
#
# grep parola-da-cercare <nomefile

[j]<>nomefile
# Apre il file "nomefile" in lettura e scrittura, e gli assegna il
#+ descrittore di file "j".
# Se il file "nomefile" non esiste, lo crea.
# Se il descrittore di file "j" non viene specificato, si assume per
#+ default il df 0, lo stdin.
#
# Una sua applicazione è quella di scrivere in un punto specifico
#+ all'interno di un file.
echo 1234567890 > File      # Scrive la stringa in "File".
exec 3<> File              # Apre "File" e gli assegna il df 3.
read -n 4 <&3              # Legge solo 4 caratteri.
echo -n . >&3              # Scrive il punto decimale dopo i
                        #+ caratteri letti.

exec 3>&-                  # Chiude il df 3.
cat File                  # ==> 1234.67890
# È l'accesso casuale, diamine.

|
# Pipe.
# Strumento generico per concatenare processi e comandi.
# Simile a ">", ma con effetti più generali.
# Utile per concatenare comandi, script, file e programmi.
cat *.txt | sort | uniq > file-finale
# Ordina gli output di tutti i file .txt e cancella le righe doppie,
# infine salva il risultato in "file-finale".

```

È possibile combinare molteplici istanze di redirezione input e output e/o pipe in un'unica linea di comando.

```

comando < file-input > file-output

comando1 | comando2 | comando3 > file-output

```

È possibile redirigere più flussi di output in un unico file.

```

ls -yz >> comandi.log 2>&1
# Invia il risultato delle opzioni errate "yz" di "ls" nel file "comandi.log"
# Poiché lo stderr è stato rediretto nel file, in esso si troveranno anche
#+ tutti i messaggi d'errore.

# Notate, però, che la riga seguente *non* dà lo stesso risultato.
ls -yz 2>&1 >> comandi.log

```

```
# Visualizza un messaggio d'errore e non scrive niente nel file.

# Nella redirectione congiunta di stdout e stderr,
#+ non è indifferente l'ordine dei comandi.
```

## Chiusura dei descrittori di file

`n<&-`

Chiude il descrittore del file di input *n*.

`0<&-`, `<&-`

Chiude lo `stdin`.

`n>&-`

Chiude il descrittore del file di output *n*.

`1>&-`, `>&-`

Chiude lo `stdout`.

I processi figli ereditano dai processi genitore i descrittori dei file aperti. Questo è il motivo per cui le pipe funzionano. Per evitare che un descrittore di file venga ereditato è necessario chiuderlo.

```
# Redirige solo lo stderr alla pipe.

exec 3>&1                                # Salva il "valore" corrente dello stdout.
ls -l 2>&1 >&3 3>&- | grep bad 3>&-      # Chiude il df 3 per 'grep' (ma
                                     #+ non per 'ls').

#           ^^^^      ^^^^
exec 3>&-                                # Ora è chiuso anche per la parte
                                     #+ restante dello script.

# Grazie, S.C.
```

Per una più dettagliata introduzione alla redirectione I/O vedi [Appendice E](#).

## 19.1. Uso di `exec`

Il comando **`exec <nomefile`** redirige lo `stdin` nel file indicato. Da quel punto in avanti tutto lo `stdin` proverrà da quel file, invece che dalla normale origine (solitamente l'input da tastiera). In questo modo si dispone di un mezzo per leggere un file riga per riga con la possibilità di verificare ogni riga di input usando [sed](#) e/o [awk](#).

### Esempio 19-1. Redirigere lo `stdin` usando `exec`

```
#!/bin/bash
# Redirigere lo stdin usando 'exec'.
```

```

exec 6<&0          # Collega il descrittore di file nr.6 allo stdin.
                  # Salva lo stdin.

exec < file-dati   # lo stdin viene sostituito dal file "file-dati"

read a1           # Legge la prima riga del file "file-dati".
read a2           # Legge la seconda riga del file "file-dati."

echo
echo "Le righe lette dal file."
echo "-----"
echo $a1
echo $a2

echo; echo; echo

exec 0<&6 6<&-
# È stato ripristinato lo stdin dal df nr.6, dov'era stato salvato,
#+ e chiuso il df nr.6 ( 6<&- ) per renderlo disponibile per un altro processo.
#
# <&6 6<&-      anche questo va bene.


echo -n "Immetti dei dati  "
read b1 # Ora "read" funziona come al solito, leggendo dallo stdin.
echo "Input letto dallo stdin."
echo "-----"
echo "b1 = $b1"

echo

exit 0

```

In modo simile, il comando **exec >nomefile** redirige lo `stdout` nel file indicato. In questo modo, tutti i risultati dei comandi, che normalmente verrebbero visualizzati allo `stdout`, vengono inviati in quel file.

 **exec N > nomefile** ha effetti sull'intero script o *shell corrente*. La redirectione nel [PID](#) dello script o della shell viene modificata da quel punto in avanti. Tuttavia . . .

**N > nomefile** ha effetti solamente sui nuovi processi generati, non sull'intero script o shell.

Grazie a Ahmed Darwish per la precisazione.

### Esempio 19-2. Redirigere lo `stdout` utilizzando *exec*

```

#!/bin/bash
# reassign-stdout.sh

FILELOG=filelog.txt

exec 6>&1          # Collega il descrittore di file nr.6 allo stdout.
                  # Salva lo stdout.

exec > $FILELOG   # stdout sostituito dal file "filelog.txt".

# ----- #
# Tutti i risultati dei comandi inclusi in questo blocco di codice vengono
#+ inviati al file $FILELOG.

```

```

echo -n "File di log: "
date
echo "-----"
echo

echo "Output del comando \"ls -al\""
echo
ls -al
echo; echo
echo "Output del comando \"df\""
echo
df

# ----- #

exec 1>&6 6>&-      # Ripristina lo stdout e chiude il descrittore di file nr.6.

echo
echo "== ripristinato lo stdout alla funzionalità di default == "
echo
ls -al
echo

exit 0

```

### Esempio 19-3. Redirigere con *exec*, nello stesso script, sia *stdin* che lo *stdout*

```

#!/bin/bash
# upperconv.sh
# Converte in lettere maiuscole il testo del file di input specificato.

E_ACCESSO_FILE=70
E_ERR_ARG=71

if [ ! -r "$1" ]      # Il file specificato ha i permessi in lettura?
then
    echo "Non riesco a leggere il file di input!"
    echo "Utilizzo: $0 file-input file-output"
    exit $E_ACCESSO_FILE
fi
                        # Esce con lo stesso errore anche quando non viene
                        #+ specificato il file di input $1 (perché?).

if [ -z "$2" ]
then
    echo "Occorre specificare un file di output."
    echo "Utilizzo: $0 file-input file-output"
    exit $E_ERR_ARG
fi

exec 4<&0
exec < $1              # Per leggere dal file di input.

exec 7>&1
exec > $2              # Per scrivere nel file di output.
                        # Nell'ipotesi che il file di output abbia i permessi
                        #+ di scrittura (aggiungiamo una verifica?).

# -----

```

```

    cat - | tr a-z A-Z      # Conversione in lettere maiuscole.
#   ^^^^^                # Legge dallo stdin.
#   ^^^^^^^^^^^          # Scrive sullo stdout.
# Comunque, sono stati rediretti sia lo stdin che lo stdout.
# -----

exec 1>&7 7>&-              # Ripristina lo stdout.
exec 0<&4 4<&-              # Ripristina lo stdin.

# Dopo il ripristino, la riga seguente viene visualizzata allo stdout
#+ come ci aspettiamo.
echo "Il file \"$1\" è stato scritto in \"$2\" in lettere maiuscole."

exit 0

```

La redirectione I/O è un modo intelligente per evitare il problema delle temute [variabili inaccessibili all'interno di una subshell](#).

### Esempio 19-4. Evitare una subshell

```

#!/bin/bash
# avoid-subshell.sh
# Suggestito da Matthew Walker.

Righe=0

echo

cat miofile.txt | while read riga;
do {
    echo $riga
    (( Righe++ )); # I valori assunti da questa variabile non
                  #+ sono accessibili al di fuori del ciclo.
                  # Problema di subshell.
}
done

echo "Numero di righe lette = $Righe"      # 0
                                           # Sbagliato!

echo "-----"

exec 3<> miofile.txt
while read riga <&3
do {
    echo "$riga"
    (( Righe++ )); # I valori assunti da questa variabile
                  #+ sono accessibili al di fuori del ciclo.
                  # Niente subshell, nessun problema.
}
done
exec 3>&-

echo "Numero di righe lette = $Righe"      # 8

echo

exit 0

```



```
# Le righe seguenti non vengono elaborate dallo script.

$ cat miofile.txt

Riga 1.
Riga 2.
Riga 3.
Riga 4.
Riga 5.
Riga 6.
Riga 7.
Riga 8.
```

## Note

- [1] Un *descrittore di file* è semplicemente un numero che il sistema operativo assegna ad un file aperto per tenerne traccia. Lo si consideri una versione semplificata di un puntatore ad un file. È analogo ad un *gestore di file* del C.
- [2] L'uso del *descrittore di file* 5 potrebbe causare problemi. Quando Bash crea un processo figlio, come con [exec](#), il figlio eredita il fd 5 (vedi la e-mail di Chet Ramey in archivio, [SUBJECT: RE: File descriptor 5 is held open](#)). Meglio non utilizzare questo particolare descrittore di file.

## 19.2. Redirigere blocchi di codice

Anche i blocchi di codice, come i cicli [while](#), [until](#) e [for](#), nonché i costrutti di verifica [if/then](#), possono prevedere la redirezione dello `stdin`. Persino una funzione può usare questa forma di redirezione **Esempio 19-5. Ciclo *while* rediretto**

```
#!/bin/bash
# redir2.sh

if [ -z "$1" ]
then
    Nomefile=nomi.data          # È il file predefinito, se non ne viene
                                #+ specificato alcuno.
else
    Nomefile=$1
fi
#+ Nomefile=${1:-nomi.data}
# può sostituire la verifica precedente (sostituzione di parametro).

conto=0

echo

while [ "$nome" != Smith ]     # Perché la variabile $nome è stata usata
                                #+ con il quoting?
do
```

```

    read nome                # Legge da $Nomefile invece che dallo stdin.
    echo $nome
    let "conto += 1"
done <"$Nomefile"          # Redirige lo stdin nel file $Nomefile.
#      ^^^^^^^^^^^^^^^

echo; echo "$conto nomi letti"; echo

exit 0

# È da notare che, in alcuni linguaggi di scripting di shell più vecchi, il
#+ ciclo rediretto viene eseguito come una subshell.
# Di conseguenza $conto restituirebbe 0, il valore di inizializzazione prima
#+ del ciclo.
# Bash e ksh evitano l'esecuzione di una subshell "ogni qual volta questo sia
#+ possibile", cosicché questo script, ad esempio, funziona correttamente.
# (Grazie a Heiner Steven per la precisazione.)

# Tuttavia . . .
# Bash, talvolta, *può* eseguire una subshell per un ciclo "while-read"
#+ posto dopo una PIPE, diversamente da un ciclo "while" REDIRETTO.

abc=hi
echo -e "1\n2\n3" | while read l
do abc="$l"
  echo $abc
done
echo $abc

# Grazie a Bruno de Oliveira Schneider per averlo dimostrato
#+ con il precedente frammento di codice.
# Grazie anche a Brian Onn per la correzione di un errore di notazione.

```

## Esempio 19-6. Una forma alternativa di ciclo *while* rediretto

```

#!/bin/bash

# Questa è una forma alternativa dello script precedente.

# Suggesto da Heiner Steven
#+ come espediente in quelle situazioni in cui un ciclo rediretto
#+ viene eseguito come subshell e, quindi, le variabili all'interno del ciclo
#+ non conservano i loro valori dopo che lo stesso è terminato.

if [ -z "$1" ]
then
    Nomefile=nomi.data        # È il file predefinito, se non ne viene
                              #+ specificato alcuno.
else
    Nomefile=$1
fi

exec 3<&0                      # Salva lo stdin nel descrittore di file 3.
exec 0<"$Nomefile"           # Redirige lo standard input.

conto=0
echo

```

```

while [ "$nome" != Smith ]
do
    read nome                # Legge dallo stdin rediretto ($Nomefile).
    echo $nome
    let "conto += 1"
done                          # Il ciclo legge dal file $Nomefile.
                              #+ a seguito dell'istruzione alla riga 21.

# La versione originaria di questo script terminava il ciclo "while" con
#+     done <"$Nomefile"
# Esercizio:
# Perché questo non è più necessario?

exec 0<&3                     # Ripristina il precedente stdin.
exec 3<&-                     # Chiude il temporaneo df 3.

echo; echo "$conto nomi letti"; echo

exit 0

```

### Esempio 19-7. Ciclo *until* rediretto

```

#!/bin/bash
# Uguale all'esempio precedente, ma con il ciclo "until".

if [ -z "$1" ]
then
    Nomefile=nomi.data        # È il file predefinito, se non ne viene
                              #+ specificato alcuno.
else
    Nomefile=$1
fi

# while [ "$nome" != Smith ]
until [ "$nome" = Smith ]    # Il != è cambiato in =.
do
    read nome                # Legge da $Nomefile, invece che dallo stdin.
    echo $nome
done <"$Nomefile"           # Redirige lo stdin nel file $Nomefile.
#     ^^^^^^^^^^^^^^^

# Stessi risultati del ciclo "while" dell'esempio precedente.

exit 0

```

### Esempio 19-8. Ciclo *for* rediretto

```

#!/bin/bash

if [ -z "$1" ]
then
    Nomefile=nomi.data        # È il file predefinito, se non ne viene
                              #+ specificato alcuno.
else
    Nomefile=$1
fi

conta righe=`wc $Nomefile | awk '{ print $1 }'`
#           Numero di righe del file indicato.

```

```
#
# Elaborato e con diversi espedienti, ciò nonostante mostra che
#+ è possibile redirigere lo stdin in un ciclo "for" ...
#+ se si è abbastanza abili.
#
# Più conciso      conta_righe=$(wc -l < "$Nomefile")

for nome in `seq $conta_righe` # Ricordo che "seq" genera una sequenza
                               #+ di numeri.
# while [ "$nome" != Smith ]   -- più complicato di un ciclo "while" --
do
    read nome                  # Legge da $Nomefile, invece che dallo stdin.
    echo $nome
    if [ "$nome" = Smith ]     # Sono necessarie tutte queste istruzioni
                               #+ aggiuntive.
    then
        break
    fi
done <"$Nomefile"             # Redirige lo stdin nel file $Nomefile.
#      ^^^^^^^^^^^^^^^

exit 0
```

Il precedente esempio può essere modificato per redirigere anche l'output del ciclo.

### **Esempio 19-9. Ciclo *for* rediretto (rediretti sia lo *stdin* che lo *stdout*)**

```
#!/bin/bash

if [ -z "$1" ]
then
    Nomefile=nomi.data          # È il file predefinito, se non ne viene
                               #+ specificato alcuno.
else
    Nomefile=$1
fi

Filereg=$Nomefile.nuovo        # Nome del file in cui vengono salvati i
                               #+ risultati.
NomeFinale=Jonah               # Nome per terminare la "lettura".

conta_righe=`wc $Nomefile | awk '{ print $1 }'` # Numero di righe del file
                                               #+ indicato.

for nome in `seq $conta_righe`
do
    read nome
    echo "$nome"
    if [ "$nome" = "$NomeFinale" ]
    then
        break
    fi
done < "$Nomefile" > "$Filereg" # Redirige lo stdin nel file $Nomefile,
#      ^^^^^^^^^^^^^^^^^^^^^ e lo salva nel file di backup $Filereg.

exit 0
```

### **Esempio 19-10. Costrutto *if/then* rediretto**

```
#!/bin/bash

if [ -z "$1" ]
then
    Nomefile=nomi.data      # È il file predefinito, se non ne viene
                           #+ specificato alcuno.
else
    Nomefile=$1
fi

TRUE=1

if [ "$TRUE" ]             # vanno bene anche  if true    e    if :
then
    read nome
    echo $nome
fi <"$Nomefile"
#  ^^^^^^^^^^^^^

# Legge solo la prima riga del file.
# Il costrutto "if/then" non possiede alcuna modalità di iterazione
#+ se non inserendolo in un ciclo.

exit 0
```

### Esempio 19-11. File dati *nomi.data* usato negli esempi precedenti

```
Aristotile
Belisario
Capablanca
Eulero
Goethe
Hamurabi
Jonah
Laplace
Maroczy
Purcell
Schmidt
Simmelweiss
Smith
Turing
Venn
Wilson
Znosko-Borowski

# Questo è il file dati per
#+ "redir2.sh", "redir3.sh", "redir4.sh", "redir4a.sh", "redir5.sh".
```

Redirigere lo `stdout` di un blocco di codice ha l'effetto di salvare il suo output in un file. Vedi [Esempio 3-2](#).

Gli [here document](#) rappresentano casi particolari di blocchi di codice rediretti. Stando così le cose, è possibile redirigere l'output di un *here document* nello `stdin` per un *ciclo while*.

```
# Esempio fornito da Albert Siersema
# Usato con il suo permesso (grazie!).

function creaOutput()
# Naturalmente, potrebbe anche essere un comando esterno.
```

```

# Qui viene mostrato come si possa usare una funzione allo stesso modo.
{
  ls -al *.jpg | awk '{print $5,$9}'
}

nr=0          # Vogliamo che il ciclo while sia in grado di manipolare queste
dimensTotale=0 #+ variabili e visualizzare i cambiamenti al termine del ciclo.

while read dimensFile Nomefile ; do
  echo "$Nomefile è grande $dimensFile byte"
  let nr++
  dimensTotale=$((dimensTotale+dimensFile))
  # O: "let dimensTotale+=dimensFile"
done<<EOF
$(creaOutput)
EOF

echo "$nr file per un totale di $totalSize byte"

```

## 19.3. Applicazioni

Un uso intelligente della redirectione I/O consente di mettere insieme, e verificare, frammenti di output dei comandi. Questo permette di generare dei rapporti e dei file di log.

### Esempio 19-12. Eventi da registrare in un file di log

```

#!/bin/bash
# logevents.sh, di Stephane Chazelas.

# Evento da registrare in un file.
# Deve essere eseguito da root (per l'accesso in scrittura a /var/log).

UID_ROOT=0      # Solo gli utenti con $UID 0 hanno i privilegi di root.
E_NONROOT=67    # Errore di uscita non root.

if [ "$UID" -ne "$UID_ROOT" ]
then
  echo "Bisogna essere root per eseguire lo script."
  exit $E_NONROOT
fi

DF_DEBUG1=3
DF_DEBUG2=4
DF_DEBUG3=5

# Decommentate una delle due righe seguenti per attivare lo script.
# LOG_EVENTI=1
# LOG_VAR=1

log() # Scrive la data e l'ora nel file di log.
{
  echo "$(date)  $" >&7      # *Accoda* la data e l'ora nel file.
                           # Vedi oltre.
}

```

```

}

case $LIVELLO_LOG in
  1) exec 3>&2          4> /dev/null 5> /dev/null;;
  2) exec 3>&2          4>&2        5> /dev/null;;
  3) exec 3>&2          4>&2        5>&2;;
  *) exec 3> /dev/null 4> /dev/null 5> /dev/null;;
esac

DF_LOGVAR=6
if [[ $LOG_VAR ]]
then exec 6>> /var/log/vars.log
else exec 6> /dev/null          # Sopprime l'output.
fi

DF_LOGEVENTI=7
if [[ $LOG_EVENTI ]]
then
  # then exec 7 >(exec gawk '{print strftime(), $0}' >> /var/log/event.log)
  # La riga precedente non funziona nella versione Bash 2.04.
  exec 7>> /var/log/event.log    # Accoda in "event.log".
  log                          # Scrive la data e l'ora.
else exec 7> /dev/null          # Sopprime l'output.
fi

echo "DEBUG3: inizio" >&${DF_DEBUG3}

ls -l >&5 2>&4                  # comando1 >&5 2>&4

echo "Fatto"                   # comando2

echo "invio mail" >&${DF_LOGEVENTI} # Scrive "invio mail" nel df nr.7.

exit 0

```