

12 - Stream IO non predefinite

Uno script `bash` potrebbe necessitare di effettuare letture e scritture su file, anche se non vengono passate come argomento.

È possibile ottenere un **file descriptor** sulla base di uno specifico file che si trova sul disco, ed utilizzarlo per effettuare operazioni di input/output. È possibile **specificare uno specifico file descriptor** tramite il comando `exec`. È possibile specificare uno *specifico* file descriptor, oppure lasciare che il sistema operativo **scelga un file descriptor libero** (quest'ultima scelta è preferibile per evitare conflitti). Si usano i seguenti **metacaratteri di redirectione**:

- `<` : solo lettura
- `>` : solo scrittura
- `>>` : append mode
- `<>` : lettura e scrittura

Alcuni esempi:

```
exec {FD}< /home/vulpi/ciao.txt
# In questo caso la variabile FD contiene il file descriptor che rappresenta
il file /home/vulpi/ciao.txt; viene considerata la sola lettura.
while read -u ${FD} STRINGA ; do
    echo "Ho letto: ${STRINGA}"
done
# Legge riga per riga!
#####
exec {FD}> /home/vulpi/out.txt
for name in CIA01 CIA02 CIA03 ; do
    echo "Inserisco ${name}" 1>&${FD}
done
# Vedremo dopo cosa fa quella cosa alla fine di echo.
```

I file descriptor aperti da una specifica `bash` si trovano nella directory speciale `/proc/$$`, dove `$$` è una variabile che restituisce il **Process ID della shell**. In ogni sottodirectory di `/proc` è presente la directory `fd`, che contiene i file descriptor utilizzati da tale programma in quel momento.

È possibile chiudere un qualsiasi stream i/o tramite la strana sintassi:

```
exec n>&-
```

con `n` il numero di file descriptor.

Non è possibile utilizzare uno stesso file descriptor una volta che viene chiuso.

Ridirezionamento

Quando una shell genera un processo figlio, tale processo figlio ottiene **UNA COPIA** dei file descriptor in quel momento aperti dal padre. È possibile però modificare tali file descriptor, associandoli a stream i/o diversi. In tal caso i file descriptor del figlio avranno gli stessi valori di quelli del padre, ma che si riferiscono a stream differenti.

Una shell padre può aprire e chiudere file, in base alle operazioni che deve compiere. Può tuttavia anche **modificare lo stream di alcuni dei suoi file descriptor**; in questo modo, utilizzando i vecchi file descriptor, può accedere **ad altri stream di dati**. Anche qui si hanno alcuni **operatori**:

- `<` : permette di ricevere un input da un file specificato successivamente.
- `>` : lo *standard output* del programma che precede viene inserito nel file successivo a questo carattere, **eliminando il contenuto di quest'ultimo**.
- `>>` : lo *standard output* del programma che precede viene inserito nel file successivo a questo carattere, **inserendolo in coda**.
- `|` (detto *pipe*): lo *standard output* del programma che precede viene usato come input del programma successivo al carattere.

Input da file

```
programma < input_file
```

`programma` vedrà il contenuto di `input_file` come se gli venisse digitato da tastiera. Si noti che, a differenza dello *stdout*, il programma si accorge che le cose da leggere sono terminate quando il file stesso è terminato. Nel caso fossimo da tastiera, al fine di segnalare la fine dell'inserimento, usiamo la combo *CTRL+D*.

Output su file

```
programma >> out_file # Append
programma > out_file # Overwrite
```

Ipotizziamo che l'output di `programma` sia `cane gatto cane` e che `out_file` contenga `animali animalosi`.

Dopo la prima operazione, l'output di `programma` verrà **aggiunto in coda** a `out_file`; ne consegue che `out_file` contenga `animali animalosi cane gatto cane`.

Dopo la seconda operazione, l'output di `programma` **sovrascriverà** il contenuto di `out_file`; ne consegue che `out_file` contenga `cane gatto cane`.

I ridirezionamenti di input e di output possono essere fatti contemporaneamente:

```
program < file_input > file_output
```

È possibile ridirezionare *standard output* e *standard error* contemporaneamente tramite `&>` e separatamente specificando `>` per lo *standard output* e `2>` per lo *standard error*.

NB: I ridirezionamenti non cambiano il valore del file descriptor che viene rediretto e nemmeno il numero dei file descriptor aperti fino a quel momento; è in tutto e per tutto **una redirezione**.

Generalizzazione operatori

Possiamo generalizzare gli operatori `>` e `<` per specificare da quale file descriptor considerare la redirezione.

- `N> Target` redireziona la stream con file descriptor `N` sul file `Target`. Omesso `N` si intende *standard output*.
- `<N Source` redireziona il file `Source` **sul** file descriptor `N` del programma specificato a sinistra dell'operatore. Omesso `N` si intende standard input.

Redirezionamento tramite pipe |

È possibile eseguire due comandi contemporaneamente, inviando lo *standard output* del primo come input nel secondo, tramite la *pipe* `|`.

```
program1 | program2
```

Cosa **fondamentale** della pipe è che se a sinistra o a destra della pipe sono presenti **sequenze di comandi** o **cicli iterativi** tali istruzioni sono ***eseguire in una shell figlia***.

Ridirezionamento di stream già definite tramite file descriptor

Se una `bash` ha due stream di dati aperti (entrambi di *output* o di *input*), ciascuno ha un valore di **file descriptor** `N` e `M`, è possibile redirezionare lo stream di file descriptor `N` in quello con file descriptor `M` tramite la sintassi

```
N>&M
```

Quando il comando precedente a questa operazione viene eseguito, ciò che normalmente verrebbe scritto in N **verrà scritto in M**.

Si faccia attenzione all'uso di `>` e `>&`:

- `>` redirige a un **file**, non a una **stream di dati**: il nome posto dopo a `>` definisce il **nome di un file**.
- `>&` redirige a una **stream di dati**, non a un file: ciò che è posto dopo a `>&` definisce il **file descriptor della stream**.

L'ordine di ridirezionamento è **fondamentale**:

```
ls . 2>error.txt 1>&2
```

```
ls . 1>&2 2>error.txt
```

- Primo comando: lo standard error di `ls .` viene rediretto al file `error.txt`; dopodiché lo stream di dati dello standard output viene rediretto allo stream di dati con file descriptor 2, il quale ora punta a `error.txt`: risulta **che `stdout` e `stderr` puntano e scrivono entrambi su `output.txt`**.
- Secondo comando: lo standard output di `ls .` viene rediretto allo stream di dati con file descriptor 2, ossia lo standard error; dopodiché lo stream di dati dello standard error viene rediretto al file `error.txt`: risulta che **lo standard output scriverà su `video`** (in particolare sulla stream di dati dello standard error) **mentre lo standard error viene scritto su `output.txt`**.

È possibile utilizzare l'operatore `>&` per redirezionare degli stream di dati allo standard input, in modo che **siano utilizzabili tramite pipe `|`**.

```
ls filechenonesiste.txt 2>&1 | grep such
```

```
# Lo stderr di ls viene rediretto allo stdout: in questo modo gli errori  
possono essere passati al comando successivo alla pipe!
```

L'operatore composto `|&` rende possibile passare al comando successivo ad esso sia lo **standard output che lo standard error**.

È bene notare i tempi di esecuzione di una sequenza di comandi:

- Se separati da `;` vengono eseguiti uno dopo l'altro - si aspetta che il primo finisca, poi si esegue il successivo
- Se separati da `|` vengono eseguiti **contemporaneamente**, redirigendo lo standard output del primo come standard input del secondo

- Se separati da `|&` vengono eseguiti **contemporaneamente** redirigendo sia standard output che standard error del primo come standard input del secondo.

È possibile effettuare ridirezionamenti per **blocchi di comando** quali `for`, `while`, `if-else`, i quali eseguiranno il ridirezionamento **per ogni istruzione presente all'interno del blocco**.

```
NUM=1
echo "${NUM}"
while (( "${NUM}" <= "3" )) ; do
    echo "${NUM}"
    ((NUM=${NUM}+1))
done > pippo.txt
echo "${NUM}"
```

In questo caso `pippo.txt` sarà lo standard output per le operazioni di `echo` interne al `while`.

Redirect "Here documents" e "Here strings"

Utilizzando l'operatore `<<` seguito da una parola, allora si possono specificare **una serie di stringhe** che verranno usate come input per una specifica istruzione o un blocco di istruzioni. La parola successiva a `<<` viene usata come **parola terminatrice** per tali stringhe.

```
while read A B C ; do
    echo $B
done << FINE
uno due tre quattro
alfa beta gamma
gatto cane
FINE
echo ciao

# Output: due beta cane ciao
```

Questo è un ridirezionamento di tipo "Here documents", ed è usato per **redirezionare stringhe senza specificare un file**. È possibile anche espandere variabili tra le stringhe disponibili.

Tramite l'operatore `<<<` si specifica invece **un unico argomento** che deve essere utilizzato come input per una specifica istruzione o blocco di istruzioni.

```
read A B C <<< alfa
echo 1 $A 2 $B 3 $C
# Output: 1 alfa 2 3

read A B C <<< "alfa beta gamma"
echo 1 $A 2 $B 3 $C
# Output 1 alfa 2 beta 3 gamma

read A B C <<< alfa beta gamma
echo 1 $A 2 $B 3 $C
# Output 1 alfa 2 3 - occhio agli argomenti!
```

Anche in questo caso è possibile espandere variaibli.