git4LaTeX
Una guida introduttiva a Git per progetti LaTeX

Pietro Giuffrida

29 gennaio 2011

Indice

1 Scopo della guida **3**

2 Introduzione a Git **4**

3 Un semplice progetto LaTeX **6**

3.1 Creazione e inizializzazione del repository 6

3.2 Aggiungere altri file al progetto: il file `.gitignore` 7

3.3 Consultazione dei log e ripristino di file 9

3.4 Gestione dei *branch* 12

3.4.1 Perché creare un *branch*? 12

3.4.2 Creare e cancellare un *branch* 13

3.4.3 Effettuare un *merge* 15

3.4.4 Copiare un singolo *commit* da un *branch* a un altro 16

3.5 Configurazioni basilari di Git 17

4 Backup **17**

4.1 Backup su periferica esterna (usb) 17

4.2 Backup on-line 18

5 commit automatico: inotifywait **18**

6 Installazione per Windows **19**

1 Scopo della guida

Git è un *revision control system* (o *version control system*, spesso abbreviato in VCS), vale a dire un programma che permette di tener traccia di tutte le modifiche e le evoluzione effettuate nel corso della stesura di un codice o di un qualsiasi progetto su supporto digitale. Git è rilasciato sotto licenza GPL, ed è disponibile, oltre che nei repository delle varie distribuzioni GNU-Linux, all'indirizzo `http://git-scm.com/`.

LaTeX è un programma di composizione tipografica di alta qualità. LaTeX è software libero. Generalmente fa parte dei programmi fondamentali preinstallati in ogni distribuzione GNU-Linux, ed è comunque disponibile a partire dall'indirizzo `http://www.latex-project.org/` per i principali sistemi operativi.

In questo documento desidero mostrare come utilizzare Git per tener traccia delle modifiche rilevanti e delle versioni elaborate nel corso dell'elaborazione di un documento scritto con LaTeX. Tramite Git è infatti possibile mantenere un backup incrementale del proprio codice sorgente, sia in locale che in remoto, senza incorrere in un eccessivo dispendio di energie e senza deconcentrarsi eccessivamente dalla stesure del proprio testo. I passaggi descritti relativamente all'uso di Git dovrebbero essere validi in linea di principio per l'elaborazione di qualsiasi documento (foto, file `.doc`, `.xls` e quant'altro), ma Git dà il meglio di sé con i file di testo puro, poiché permette di vedere tutte le modifiche apportate al file di volta in volta. Proprio per questo l'uso principale di Git, e dei VCS in generale, è quello di controllare lo sviluppo dei software.[1] Anche il codice dei documenti LaTeX si scrive su file di testo puro e quindi Git è particolarmente adatto a essere utilizzato in accoppiata con LaTeX. Questa guida è proprio strutturata in modo specifico per illustrare la loro interazione, almeno a livello base.

Solo i primi due paragrafo sono indispensabili. In essi vengono tratte le basi del funzionamento di Git e mostrati i passaggi fondamentali per la creazione di un repository locale del proprio progetto, per il salvataggio progressivo delle versioni, e quindi per svolgere eventuali operazioni di ripristino.

I paragrafi successivi sono invece dedicate alla creazione e alla sincronizzazione di un repository remoto (che si tratti di un servizio on-line o semplicemente di una memoria esterna), e a uno script bash che permette di eseguire dei commit automatici a ogni salvataggio dei file.

La guida non si prefigge alcun compito specifico per quanto riguarda LaTeX, a proposito del quale esistono numerose guide.[2]

[1] Git, infatti, è stato creato da Linus Torvalds, inventore del kernel Linux, perché aveva bisogno di un buon VCS per gestire lo sviluppo di Linux.

[2] Si veda a questo proposito, oltre alla documentazione reperibile sul proprio computer

La guida non contempla l'uso di una qualunque interfaccia grafica (GUI). Tutti i comandi e i passaggi illustrati sono eseguiti tramite una shell Linux. Non ho esperienza né di Git né di LaTeX su sistemi operativi non Unix-like. Un minimo di compatibilità con altri sistemi è garantita dal fatto che i comandi tipici della shell Linux sono evidenziati e commentati, in modo che l'utente di altri sistemi operativi o abituato all'uso di GUI possa sostituirli svolgendo altrimenti le medesime attività. I comandi di Git dovrebbero invece restare i medesimi in ogni OS, per quanto anche in questo caso le medesime attività possano essere svolte mediante GUI.

Non sono un esperto di informatica, ma trovo bello far le cose, per quanto possibile, con le mie mani, sapere cosa fa la macchina e avere l'illusione che nel rapporto quasi-simbiotico con il computer sia io a decidere.

Dato il carattere ludico della presente guida, scelgo come licenza la Creative Commons Attribution-NonCommercial-ShareAlike 2.5 Italy. Gli unici vincoli sono quindi quelli di riconoscere la paternità dell'opera originale, di non lucrare sulla sua distribuzione, e di distribuire l'opera e i suoi eventuali derivati con la medesima licenza.[3]

2 Introduzione a Git

Git permette di tenere traccia delle modifiche apportate a un progetto registrando una copia dei file del progetto in un database. Sostanzialmente Git scatta una foto dei file presenti nella cartella nel momento in cui si effettua il *commit*.

Per controllare continuamente l'integrità dei dati Git utilizza un sistema, detto *checksum*, che associa a ogni stato di un progetto una sequenza di bit che la identifica. Nella fattispecie Git utilizza l'algoritmo SHA-1 che restituisce una stringa esadecimale (detta *hash*) composta da 40 caratteri alfanumerici (numeri da `0` a `9` e lettere da `a` a `f`) che può apparire così:

```
43c5858e91a7090b834dd9f09ddfae1061901ee4
```

In questo modo è impossibile modificare un file del progetto senza che Git se ne renda conto. Per fare riferimento a una particolare versione del progetto si indica il corrispondente *hash* o, a volte, solo i primi 7 caratteri.

---

mediante il comando `texdoc nomepacchetto`, la documentazione in italiano reperibile a partire dal sito del GuIT (`http://www.guit.sssup.it`), oltre all'ottima guida di L. Pantieri disponibile sul sito `http://www.lorenzopantieri.net/LaTeX.html`.

[3] Quest'opera è stata rilasciata sotto la licenza Creative Commons Attribution-NonCommercial-ShareAlike 2.5 Italy. Per leggere una copia della licenza visita il sito web `http://creativecommons.org/licenses/by-nc-sa/2.5/it/` o spedisci una lettera a Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

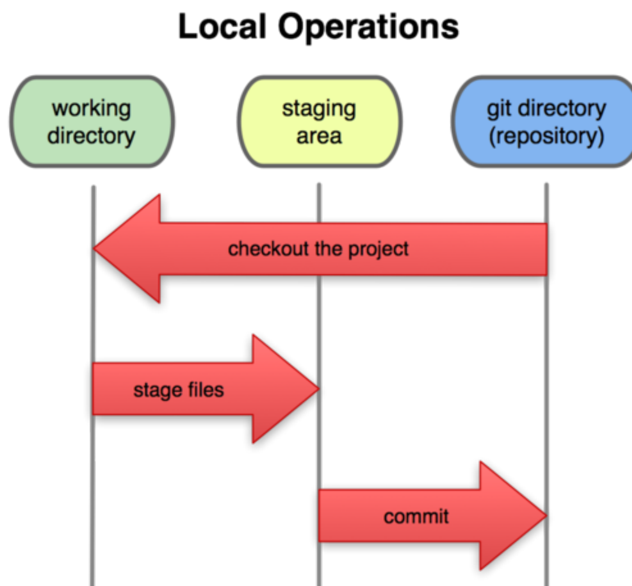


Figura 1: Working directory, staging area e git directory. Immagine presa da: http://progit.org/book/ch1-3.html.

Prima di iniziare a metterci al lavoro c'è un'ultima cosa da sapere su Git. I file possono trovarsi in tre stati chiamati, in inglese, *committed*, *modified* e *staged*. *Commited* significa che il file è stato salvato nel proprio database locale; *modified* indica che il file è stato modificato ma non ancora salvato nel database con un *commit*; *staged* significa che il file è stato modificato e la sua versione attuale verrà salvata nel database con il *commit* successivo. Un progetto Git può quindi essere suddiviso in tre sezioni principali: la *git directory*, la *working directory* e la *staging area*. La prima è dove Git conserva i metadati e gli oggetti del database del proprio progetto. La *working directory* è, come dice il nome stesso, la "cartella di lavoro", ossia una copia di una versione del progetto a nostra disposizione per l'uso e la modifica dei file. L'ultima sezione, la *staging area*, è un semplice file, generalmente contenuto nella cartella Git, che conserva le informazioni su ciò che dovrà entrare nel *commit* successivo.

Con Git si lavora più o meno così:

1. si modifica un file presente nella *working directory*;

2. si aggiunge un suo *snapshot*, cioè una sua copia, nella *staging area*;

3. si esegue un *commit*, cioè l'operazione con la quale i file vengono copiati così come sono presenti alla *staging area* all'interno della git directory in maniera definitiva. Al *commit* viene associato l'*hash* che identifica univocamente la versione del progetto così salvata.

Se una particolare versione di un file si trova nella cartella git è considerato *committed*. Se è stato modificato e aggiunto alla *staging area* esso è detto *staged*. Se è stato modificato da quando è stata aperta la cartella di lavoro ma non ancora aggiunto alla *staging area* allora il file è detto *modified*.[4]

3 Un semplice progetto LaTeX

Vogliamo sviluppare un documento con LaTeX, utilizzando Git come VCS. Git non funziona in modo particolarmente esotico. Si tratta semplicemente di creare una directory, di posizionare in essa i nostri file `.tex` e di dire a Git di considerare tale directory come un repository.

3.1 Creazione e inizializzazione del repository

```
~$ mkdir progetto
~$ cd progetto
~/progetto$ touch np_main.tex
~/progetto$ git init
~/progetto$ git add .
~/progetto$ git commit -am "Inizializzazione del nuovo
    progetto"
```

Vediamo con calma i singoli passaggi. I primi tre comandi servono rispettivamente per creare la directory (`mkdir nome_directory`), per spostarsi al suo interno (`cd nome_directory`) e per creare un file vuoto chiamato `np_main.tex` (`touch nome_file`).

I passaggi successivi, eseguiti sempre dati dall'interno della directory, sono quelli specifici di Git:

- con `git init` creiamo una cartella nascosta chiamata `.git/` che contiene il nostro repository Git;
- `git add .` aggiunge l'argomento (in questo caso '.', ossia la cartella in cui siamo posizionati e tutti i file al suo interno) alla *staging area* del repository appena creato;
- con col comando `commit -am "nota di versione"` effettuiamo il commit che, come detto, salva una copia dei file contenuti nella *staging area* all'interno del database. Nel registro delle operazioni effettuate a questa operazione risulterà associato il messaggio `"nota di versione"`.

[4] Questo paragrafo è stato tradotto in parte da `http://progit.org/book/ch1-3.html`.

Così facendo saremo già pronti a lavorare con il nostro editor di fiducia sul file `.tex` appena creato. Non resta che eseguire un *commit* al termine di ogni modifica rilevante, in modo da salvare una determinata versione del progetto.

```
~/progetto$ echo "una modifica rilevante" >> np_main.tex
~/progetto$ git commit -am "ulteriore sviluppo"
```

Finora abbiamo eseguito il *commit* passando a Git le due opzioni `-am`. Con l'opzione `-m` diciamo a Git di utilizzare il testo tra virgolette che segue il comando come nota di versione del *commit* che stiamo eseguendo. Con l'opzione `-a` chiediamo a Git di mettere nella *staging area* tutti i file del progetto prima di eseguire il *commit*. Se non specifichiamo un messaggio con l'opzione `-m` si aprirà l'editor di testo associato di default a Git in cui potremo scrivere il messaggio del *commit*. Vedremo più avanti come impostare l'editor.

3.2 Aggiungere altri file al progetto: il file `.gitignore`

Si presenta ora uno scenario tra i più comuni: dobbiamo aggiungere un ulteriore file al progetto, per esempio un file `.tex` contenente un ulteriore capitolo, o un file `.bib` contenente la bibliografia, o un qualsiasi altro file. Quando eseguiamo il *commit* successivo, potremmo aspettarci che Git si accorga del nuovo arrivato, e in effetti dovrebbe restituire un messaggio di questo tipo:

```
~/progetto$ touch np_secondary.tex
~/progetto$ git commit -am "aggiunto file np_secondary.
    tex"
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be
    committed)
#
#       np_secondary.tex
nothing added to commit but untracked files present (use
    "git add" to track)
```

Git si accorge quindi della presenza di un nuovo file, ma non lo aggiunge automaticamente al progetto. D'altra parte, se avesse rilevato delle modifiche ai file precedentemente aggiunti al progetto, non si sarebbe nemmeno curato di comunicarci che il nuovo file non è ancora stato aggiunto al progetto. Si sarebbe infatti limitato a salvare le modifiche ai file che gli abbiamo precedentemente detto di gestire. Solo con il comando `git status` Git ci comunica con certezza l'anomalia di un nuovo file non ancora caricato nel repository.

Git è un programma molto gentile perché ci dà spesso dei suggerimenti molto utili. Infatti se osserviamo attentamente l'output precedente possiamo vedere che Git ci sta dicendo che possiamo includere i nuovi file con il comando

```
git add <file>...
```

Si può scegliere di procedere in due modi: aggiungere indiscriminatamente tutti i file della directory al progetto che stiamo sviluppando; oppure aggiungere un singolo file. Nel primo caso dovremmo ripetere il comando già usato in fase di inizializzazione:

```
~/progetto$ git add .
```

Nel secondo caso aggiungiamo un singolo file:

```
~/progetto$ git add np_secondary.tex
```

Per quanto possa sembrare eccessivo, io preferisco usare il secondo comando. Mi accade con estrema frequenza di dover aggiungere dei file a un progetto, e spesso sono fin troppo distratto da quel che sto scrivendo per occuparmi di quel Git si aspetta da me L'aggiunta indiscriminata di ogni file nella directory al repository presenta però delle controindicazioni. La più ovvia per chi lavora con LaTeX è la seguente: nel corso dell'elaborazione di un testo inevitabilmente si procederà alla generazione del documento in pdf a partire dai sorgenti; LaTeX provvederà quindi alla generazioni di una serie di file secondary (`.toc`, `.out`, ...), nonché di un pdf più o meno inutile. Se eseguissimo il comando `git add .` subito dopo una compilazione, evidentemente Git aggiungerebbe alla *staging area* anche i vari file di lavoro, dal pdf, ai file di log. Per ovviare a questo inconveniente, e continuare pigramente a eseguire `git add .`, è utile creare nella nostra directory di un file denominato `.gitignore`.

```
~/progetto$ echo "*.log" >> .gitignore
~/progetto$ echo "*.pdf" >> .gitignore
~/progetto$ echo "*.blg" >> .gitignore
~/progetto$ echo "*.bbl" >> .gitignore
~/progetto$ echo "*.aux" >> .gitignore
~/progetto$ echo "*-blx.bib" >> .gitignore
~/progetto$ echo "*.out" >> .gitignore
~/progetto$ echo "*~" >> .gitignore
~/progetto$ git add .gitignore
~/progetto$ git commit -am "Aggiunto il file .gitignore"
```

Mediante questo file, istruiamo Git a proposito di tutti i file che non fanno effettivamente parte del progetto, pur essendo presenti nella cartella. Da questo momento in poi Git si limiterà a ignorarli, il che ci permette di eseguire prudenzialmente il comando `git add .` prima di ogni *commit*.

3.3 Consultazione dei log e ripristino di file

Ipotizziamo a titolo esemplificativo il seguente scenario: abbiamo accidentalmente eliminato un file del nostro progetto, e abbiamo anche eseguito un *commit*.

```
~/progetto$ echo "pippo" >> np_secondary.tex
~/progetto$ git add .
~/progetto$ git commit -am "Ho solo iniziato a lavorare"
~/progetto$ rm np_secondary.tex
~/progetto$ git commit -am "Ma ho gia' perso tutto"
```

Ora chiediamo conto a Git della sua capacità di ripristinare una versione precedentemente salvata. La prima cosa da fare è consultare i log dei commit precedentemente effettuati, per decidere quale di essi ripristinare. Il comando appropriato sarebbe quindi `git log`, che contempla anche un'ampia serie di interessanti opzioni. Rimandando al manuale per la maggior parte di tali opzioni, segnalo solo che è possibile:

- selezionare esclusivamente i log relativi a un dato file, con la sintassi `git log nome_file`;
- chiedere a Git di stampare solo gli ultimi *n* log, con la sintassi `git log -n`.

Trattandosi di un file eliminato, non possiamo chiedere a Git di stampare solo i log a esso relativi. Dobbiamo quindi cercare tra i log più recenti fino a identificare uno stadio del progetto in cui il file è ancora presente.

```
~/progetto$ git log -2
commit 380228784f4095efdbce9d1d3408bcbdf01548cb
Author: Pietro Giuffrida <pietro.giuffri@gmail.com>
Date:   Sat Oct 2 18:58:44 2010 +0200

    Ma ho gia' perso tutto

commit c21d825df42f55ba127487fd93acf1c13c41b028
Author: Pietro Giuffrida <pietro.giuffri@gmail.com>
Date:   Sat Oct 2 18:55:07 2010 +0200

    Ho solo iniziato a lavorare
```

Nel nostro caso, l'eliminazione del file è ben documentata dalle note. La lunga stringa che compare dopo `commit` è l'*hash* che, come anticipato prima, è la stringa che identifica univocamente una particolare versione del progetto. Possiamo subito verificare il contenuto del *commit* facendo riferimento al corrispettivo *hash* con il comando `git show <hash>`:

```
~/progetto$ git show
    c21d825df42f55ba127487fd93acf1c13c41b028
commit c21d825df42f55ba127487fd93acf1c13c41b028
Author: Pietro Giuffrida <pietro.giuffri@gmail.com>
Date:   Sat Oct 2 18:58:29 2010 +0200

    aggiunto file np_secondary.tex

diff --git a/np_secondary.tex b/np_secondary.tex
new file mode 100644
index 0000000..bfa5424
--- /dev/null
+++ b/np_secondary.tex
@@ -0,0 +1 @@
+pippo
```

Il file `np_secondary.tex` fa effettivamente parte del commit da noi individuato. Esso è identificato da un'ulteriore stringa alfanumerica denominata *index*. Git, tramite il comando `diff`, ci comunica inoltre che esso contiene la stringa `pippo`. Possiamo quindi procedere a ricrearne una copia facendo riferimento all'*index*:

```
~/progetto$ git show bfa5424 > file_ripristinato
```

La procedura, con le complicazioni del caso, dovrebbe essere valida anche per circostanze concrete. Vi consiglio comunque di fare delle prove simulando per esempio *commit* di più file contenenti varie modifiche, per far pratica con gli indici e la sintassi di Git. Possiamo ipotizzare in questo caso uno scenario di questo tipo:

```
~/progetto$ touch topolino pippo
~/progetto$ echo qualcosa >> topolino
~/progetto$ echo qualcosaltro >> pippo
~/progetto$ git add .
~/progetto$ git commit -am "creati e modificati topolino
    e pippo"
~/progetto$ echo "ancora qualcosa" >> topolino
~/progetto$ git commit -am "ulteriormente elaborato
    topolino"
~/progetto$ git log topolino
commit 87548d4cd43cda917cf788866abd76f8399b639b
Author: Pietro Giuffrida <pietro.giuffri@gmail.com>
Date:   Sat Oct 2 19:27:23 2010 +0200

    ulteriormente elaborato topolino
```

```
commit f7a696e891bb81446f56adc996bc0a20a23ae9b5
Author: Pietro Giuffrida <pietro.giuffri@gmail.com>
Date:   Sat Oct 2 19:27:01 2010 +0200

    creati e modificati topolino e pippo

~/progetto$ git show
    f7a696e891bb81446f56adc996bc0a20a23ae9b5
commit f7a696e891bb81446f56adc996bc0a20a23ae9b5
Author: Pietro Giuffrida <pietro.giuffri@gmail.com>
Date:   Sat Oct 2 19:27:01 2010 +0200

    creati e modificati topolino e pippo

diff --git a/pippo b/pippo
new file mode 100644
index 0000000..a09d8a1
--- /dev/null
+++ b/pippo
@@ -0,0 +1 @@
+qualcosa
diff --git a/topolino b/topolino
new file mode 100644
index 0000000..a09d8a1
--- /dev/null
+++ b/topolino
@@ -0,0 +1 @@
+qualcosaltro

~/progetto$ git show a09d8a1 > topolino_ripristinato
```

Chiaramente, Git permette il ripristino esclusivamente dei file aggiunti al repository ed esclusivamente di versioni esplicitamente salvate mediante il comando `commit`. I salvataggi effettuati tra un *commit* e l'altro non sono quindi ricostruibili.

Nel caso in cui stiamo modificando un file, per esempio `topolino`, ma ci rendiamo conto che le modifiche apportate non ci soddisfano e vogliamo ripristinarlo alla versione del file salvata nell'ultimo *commit*, se il file è *modified* ma non *staged*, usando la terminologia vista all'inizio, possiamo usare il comando `checkout`:

```
git checkout topolino
```

3.4 Gestione dei *branch*

Con il comando `git status` possiamo controllare lo stato del progetto, per esempio se ci sono dei file che sono stati modificati ma non ancora salvati nel *commit* e così via. Un tipico output di questo comando è:

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

All'ultimo rigo leggiamo `nothing to commit (working directory clean)`, cioè non sono presenti file modificati (a parte eventualmente quelli ignorati con il file `.gitignore`) dopo l'ultimo *commit*. Ma cosa significa `On branch master`? *Branch* in inglese significa "ramo" e Git ci dà la possibilità di creare una linea di sviluppo per il nostro progetto parallela a quella iniziale (chiamata *master* in Git) ma con delle modifiche che divergono da questa proprio come il ramo di un albero diverge dal tronco centrale (e per completare questa analogia botanica la linea di sviluppo centrale è chiamata a volte *trunk*, cioè "tronco").

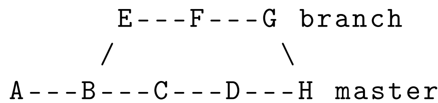
3.4.1 Perché creare un *branch*?

Le risposte a questa domanda possono essere diverse. Una potrebbe essere, per esempio perché abbiamo intenzione di scrivere un capitolo per il nostro documento LaTeX ma non siamo sicuri che sia veramente necessario: possiamo allora creare un *branch* nel quale scriveremo solo questo capitolo mentre nel ramo principale *master* continueremo normalmente a modificare il documento. Quando il capitolo sarà pronto, se il risultato sarà soddisfacente potremo procedere alla funsione del ramo "sperimentale" in quello principale, operazione chiamata *merge* in gergo, e nel nostro documento "comparirà magicamente" il capitolo che abbiamo sviluppato nel *branch*, in caso contrario possiamo semplicemente cancellare il *branch*, come se avessimo potato il ramo indesiderato dell'albero.

Un altro caso in cui può essere utile creare un *branch* è quello in cui si lavori a più mani su un documento: mentre voi scrivete il vostro documento un vostro amico vi chiede di collaborarvi (e anch'egli userà Git per tenere traccia delle sue modifiche). Lui si creerà un suo ramo di sviluppo in modo che voi potrete continuare a redigere il vostro documento senza che le sue modifiche vadano a confliggere con le vostre. Alla fine il vostro amico vi farà vedere il risultato delle sue modifiche: se le apprezzate potete effettuare il *merge* del suo ramo nel vostro, altrimenti potete dirgli che le sue modifiche non vi piacciono e il suo *branch* sarà cancellato.[5]

[5] Il vostro (forse ormai ex) amico potrebbe anche decidere di continuare a sviluppare il suo

Un semplice grafico dello sviluppo di un progetto in cui è stato effettuato un *branching* potrebbe avere questo aspetto:



Nel ramo `master` abbiamo effettuato le modifiche indicate con `A` e `B`. A questo punto creiamo il ramo `branch`. D'ora in poi le modifiche effettuate su un ramo avranno valore solo su quello: così continuiamo a lavorare normalmente sul ramo `master` apportando le modifiche indicate con `C` e `D`, mentre in `branch` avremo effettuato le modifiche `E`, `F` e `G` che apparterranno solo a questo ramo. Decidiamo quindi di fondere i due rami perché le modifiche fatte in `branch` ci soddisfano: con la modifica `H` tutte le modifiche effettuate in `branch` verranno importate nel ramo principale e potremo continuare lì il nostro lavoro.

3.4.2 Creare e cancellare un *branch*

Per creare un *branch* dobbiamo usare il comando `git branch <nome_branch>`. Se il comando va a buon fine non ci sarà nessun output, possiamo però controllare l'elenco dei rami esistenti con il comando `git branch` senza alcun ulteriore argomento:

```
$ git branch foo
$ git branch
  foo
* master
```

L'asterisco vicino a `master` sta a indicare che al momento ci troviamo nel *branch* `master`. Per spostarci in un altro *branch* usiamo il comando `git checkout <nome_branch>`:

```
$ git checkout foo
Switched to branch 'foo'
$ git branch
* foo
  master
```

Che significa che ci siamo spostati in un altro *branch*? Abbiamo cambiato cartella? No, siamo sempre nella stessa cartella (possiamo controllarlo con il comand `pwd`), ma è stato Git che ha modificato la nostra cartella di lavoro, la *working directory*, andando a prendere dal suo database l'ultimo *snapshot*, l'ul-

branch in maniera completamente indipendente da voi: questa situazione, nel mondo della programmazione, è detta *fork*.

tima "fotografia", del *branch* che gli abbiamo richiesto. In alternativa a `git checkout <nome_branch>`, per creare un nuovo ramo si può usare il comando `git checkout -b <nome_branch>` che in più effettua direttamente il *checkout* del nuovo *branch*:

```
$ git checkout -b foo
Switched to a new branch 'foo'
```

Un *branch* appena creato e non modificato è uguale al ramo da cui è stato creato, contiene i suoi stessi file e la stessa cronologia. Lo sviluppo del progetto può continuare in maniera identica a quella vista prima (cioè si usano i soliti comandi `git add` e `git commit`), però le modifiche apportate in questo *branch* non compariranno nel registro di quello principale. Possiamo verificarlo controllando i rispettivi log:

```
$ git branch
* foo
  master
$ git log
commit c71da4fe9bf764274e67a2326ec1dc3911691928
Author: Pietro Giuffrida <pietro.giuffri@gmail.com>
Date:   Sat Oct 9 19:09:56 2010 +0200

    primo commit nel branch 'foo'

commit e17384278acadbfefefebf0ced4e7103005a597c
Author: Pietro Giuffrida <pietro.giuffri@gmail.com>
Date:   Sat Oct 9 19:08:46 2010 +0200

    iniziamo il nuovo progetto
$ git checkout master
Switched to branch 'master'
$ git log
commit e17384278acadbfefefebf0ced4e7103005a597c
Author: Pietro Giuffrida <pietro.giuffri@gmail.com>
Date:   Sat Oct 9 19:08:46 2010 +0200

    iniziamo il nuovo progetto
```

Per cancellare un ramo di cui si è già effettuato il *merge* con il ramo principale (vedremo più avanti come si fa) si usa l'opzione `-d` del comando `git branch`:

```
$ git branch -d foo
Deleted branch foo (was c71da4f).
```

Se proviamo a usare l'opzione `-d` con un ramo di cui non è stato ancora effettuato il *merge* Git ci avverte:

```
$ git branch -d foo
error: The branch foo' is not fully merged.
If you are sure you want to delete it, run 'git branch -D
     foo'.
```

Come al solito Git ci suggerisce i comandi che ci possono tornare utili: se vogliamo ugualmente cancellare il *branch* che non è stato fuso in quello principale dobbiamo usare l'opzione `-D` al posto di `-d`:

```
$ git branch -D foo
Deleted branch foo (was c71da4f).
```

3.4.3 Effettuare un *merge*

Per effettuare la fusione, *merge*, di due rami si usa il comando `git merge <nome_branch>` da dare nel ramo in cui si desidera importare il ramo chiamato `<nome_branch>`. Se tutto va bene otterremo un output simile al seguente (supponiamo, per esempio, che il ramo da importare si chiami `foo`):

```
$ git merge foo
Updating da45d11..729129c
Fast-forward
 capitolo5.tex |   18 +++++++++++++++++
 1 files changed, 18 insertions(+), 0 deletions(-)
 create mode 100644 capitolo5.tex
```

Dopo di ciò possiamo normalmente fare il *commit* con `git add .` e `git commit -am "messaggio di commit"`.

Può verificarsi un problema se uno o più file sono stati modificati nello stesso punto in tutti e due i rami che si vogliono fondere, dopo che è stato creato il *branch*. Git, infatti, non è in grado di gestire automaticamente questa situazione (è uno strumento potente ma non può certo leggere nel nostro pensiero, non può sapere perché il file è stato modificato diversamente nei due rami) e ci avverte con un messaggio di questo tipo:

```
$ git merge foo
Auto-merging main.tex
CONFLICT (content): Merge conflict in main.tex
Automatic merge failed; fix conflicts and then commit the
     result.
```

In questo caso il file in cui si sono verificati i conflitti si chiama `main.tex`. Aprendo con il nostro editor di testo potremo leggere a un certo punto qualcosa di questo tipo:

```
<<<<<<< HEAD
\usepackage{hyperref}
=======
\usepackage[bookmarks=false,colorlinks=true]{hyperref}
>>>>>>> foo
```

La zona compresa fra `<<<<<<< HEAD` e `=======` indica il contenuto attuale del ramo, mentre ciò che è scritto fra `=======` e `>>>>>>> foo` è ciò che si trova nel corrispondente punto dello stesso file `main.tex` ma nel ramo `foo`. Dopo aver corretto il file come desideriamo che risulti, possiamo finalmente salvarlo ed effettuare come al solito il *commit*.

3.4.4 Copiare un singolo *commit* da un *branch* a un altro

Mentre sviluppiamo un *branch* potremmo accorgerci che il *commit* appena effettuato apporta delle modifiche che possono essere utili anche in un altro *branch* (come per esempio la correzione di un errore di ortografia o una modifica al preambolo del documento). Non è necessario effettuare il *merge* solo per un importare un *commit* o modificare manualmente i file dell'altro *branch* perché è possibile effettuare un'operazione detta *cherry-picking* (che significa "raccolta di ciligie"[6]) che nell'importare nell'altro *branch* il singolo *commit* che indicheremo. La sintassi è semplice (ovviamente prima di dare il seguente comando bisogna posizionarsi nel *branch* in cui si intende importare il *commit*): `git cherry-pick <hash_del_commit>`. Inoltre l'oggetto, l'autore (nel caso in cui ci siano più collaboratori che partecipano al progetto) e la data e orario del *commit* saranno gli stessi di quello importato. Al posto dell'intero *hash* del *commit* si possono inserire anche solo i primi 7 caratteri.

Se dovesse verificarsi qualche problema durante il *cherry-picking*, come può succedere durante il *merge*, Git ci avverte, ci consiglia di risolvere i conflitti, usare il solito `git add` come opportuno e poi, per effettuare il *commit*, di dare il comando `git commit -c <hash_del_commit>`, in modo da utilizzare le stesse informazioni del *commit* a cui ci riferiamo (autore, data e orario, oggetto):

```
$ git cherry-pick 97b8c9b
Automatic cherry-pick failed.  After resolving the
    conflicts,
mark the corrected paths with 'git add <paths>' or 'git
    rm <paths>'
```

[6]Chi ha inventato tutta questa terminologia doveva essere un amante della natura.

```
and commit the result with:

        git commit -c 97b8c9b
```

3.5 Configurazioni basilari di Git

Per quanto non strettamente indispensabile per un uso individuale di Git sul proprio pc, segnalo alcune configurazioni elementari necessarie per un uso ottimale di Git.

Le impostazioni di Git possono essere impostate con il comando `config` di Git. Usando l'opzione `--system` le configurazioni così impostate saranno valide per tutti gli utenti che hanno accesso al sistema e verranno generalmente salvate nel file di configurazione `$(prefix)/etc/gitconfig`. Con l'opzione `--global` le impostazioni saranno valide solo per il proprio utente e verranno salvate nel file `~/.gitconfig`. Infine non usando alcuna opzione le configurazioni avranno valore solo per il repository in cui vengono impostate.

In primo luogo occorre passare a Git qualche informazione circa l'utente:

```
git config --global user.name "Pietro Giuffrida"
git config --global user.email pietro.giuffri@gmail.com
```

In secondo luogo è utile dire a Git di colorare i log, in modo da renderli più leggibili:

```
git config --global color.branch auto
git config --global color.diff auto
git config --global color.interactive auto
git config --global color.status auto
```

Possiamo poi impostare l'editor di testo predefinito da associre a Git con l'opzione `core.editor`. Se per esempio vogliamo usare Gedit daremo il comando

```
git config --global core.editor gedit
```

4 Backup

4.1 Backup su periferica esterna (usb)

```
~/progetto$ git clone -l file:///home/pietro/elab/
    git4latex/ /media/usb/gigt/
```

4.2 Backup on-line

1. registrarsi su gitorius (richiede ssh key e passphrase!)
2. `git remote add origin git@gitorious.org:git4latex/git4latex.git` (che chiederà la passphrase!)
3. `git push origin master` per sincronizzare il tutto di volta in volta a fine giornata

Prima sincronizzazione da locale verso gitorius:

```
$ git remote add origin git@gitorious.org:progetto/
    progetto.git
```

A fine giornata di lavoro:

```
$ git push origin master
```

Se volete ricreare il repository in locale:

```
$ mkdir riprogetto
$ cd riprogetto
$ git clone git://gitorious.org/git4latex/git4latex.git
```

5 commit automatico: inotifywait

```
inotifywait -q -m -e CLOSE_WRITE --format="git commit -m
    'autocommit on change' %w" file.txt | sh
```

```
#!/bin/bash
#
# gitwait - watch file and git commit all changes as they
     happen
#

while true; do

    inotifywait -qq -e CLOSE_WRITE ~/.calendar/calendar

    cd ~/.calendar; git commit -a -m 'autocommit on
        change'

done
```

ma è quasi pericoloso se salvi automaticamente ogni tot minuti!!

6 Installazione per Windows

Installare Git su un sistema operativo Windows è davvero molto semplice. Il progetto msysGit ha una delle più semplici procedure di installazione.

Bisogna semplicemente scaricare ed installare l'installer .exe dalla pagina Google Code.

Dopo aver terminato l'installazione, si ha la possibilità di usare Git sia tramite interfaccia grafica, sia tramite una shell unix-like emulata o integrata nel più noto Prompt dei Comandi di Windows.