Git

Stefano Entatti

1 Cos'è un sistema di controllo di versione

Nella realizzazione di un progetto software potresti esserti accorto di voler tenere traccia dei tuoi progressi, magari registrandoli in modo da sapere cosa hai fatto e quando, o di voler tornare al momento esattamente prima a quella modifica che non fa funzionare più nulla.

Potresti provare, ogni volta che apporti modifiche importanti, a creare una nuova copia del progetto, numerarla e magari annotarti quali cambiamenti ha portato. Man mano che il progetto avanza, le copie possono diventare moltissime e serve comunque confrontare i file riga per riga per capire cosa cambia da una copia all'altra.

Se al progetto lavorano più persone, (e quindi carichi il progetto su drive) non è comunque possibile che più persone modifichino il codice nello stesso momento, e se ognuno modifica una copia diversa, queste vanno poi unite sempre a mano con dei copia e incolla.

Un sistema di controllo di versione tiene traccia di tutte le modifiche effettuate su un insieme di file (che compongono il progetto), ovvero di chi le ha fatte, quando, quali righe sono state tolte e quali aggiunte. Ogni registrazione di modifica riporta l'autore, un titolo e un descrizione che spiega in dettaglio il perche' e' stata necessaria o qualche peculiarita' della modifica stessa in modo da facilitare la comprensione ad altri sviluppatore o ad una rilettura in un futuro. Inoltre in questo modo e' possibile navigare e riportarsi in qualsiasi punto della storia del progetto.

Per permettere a più persone di lavorare nello stesso momento vengono ognuno copia in locale il progetto preso da un punto di riferimento (server comune ad esempio) e dopo aver apportato le modifiche necessarie e' possibile unirle tenendo traccia di chi ha modificato cose ed in quale ordine.

Esistono molti software di controllo versione, tra i più famosi ci sono Git, CVS, Subversion, Mercurial, Bazaar e BitKeeper.

2 Il sistema di controllo versione Git

Git è il sistema di controllo versione più utilizzato, ed è anche uno dei più semplici da usare. é nato nel 2005 ed è stato ideato da Linus Torvalds (il creatore di Linux) per facilitare lo sviluppo del kernel di Linux, uno dei progetti opensource più grandi, a cui lavorano moltissime persone. Anche git è opensource.

Git è distribuito, ciò significa che l'intero **repository**, ovvero il tutti i file che compongono un progetto, tra cui sorgenti, readme e file creati da git per tenere traccia dello storico delle modifiche può essere memorizzato su un server a cui possono accedere tutti i membri del progetto. Allo stesso tempo ognuno può tenere una copia completa del del repository in locale e utilizzare tutte le funzionalità di git anche offline. L'interazione col server è limiata a quando lo sviluppatore desidera scaricare le modifiche effettuate da altri o vuole caricare le proprie sul server.

Git è nato come un programma a riga di comando, e anche se viene considerato più efficiente se utilizzato in questo modo, oggi esistono molti client grafici e estensioni che permettono di usufruire della maggior parte delle funzionalità di git direttamente dall'IDE.

3 Branching

Git e' in grado di gestire contemporaneamente piu' sviluppi all'interno della stessa copia del progetto, questi si chiamano branch. Con piu' sviluppi si intendono piu' tipologie di modifiche differenti anche all'interno dello stesso file. In questo si puo' saltare da uno sviluppo all'altro senza perdere nessuna modifica e senza avere piu' copie del progetto.

Ogni repository git possiede almeno un branch, chiamato **master**. Possono essere creati infiniti branch, ognuno dei quali può essere utilizzato per la realizzazione di una particolare funzionalita', in modo che modifiche su parti molto diverse del codice non possano causare conflitti. Spesso un singolo sviluppatore lavora su un proprio branch, in modo da non dover gestire anche i cambiamenti fatti da altri mentre svolge il suo compito.

Oppure il branch master può essere quello che viene distribuito agli utilizzatori del software perchè considerato stabile, mentre gli altri possono essere sperimentali e dunque contenere bug, o addirittura essere utilizzati per apportare modifiche sperimentali che possono essere scartate senza compromettere tutto il resto. A volte vengono utilizzati branch diversi per separare versioni con funzionalità leggermente diverse dello stesso software.

Un branch viene creato a partire da un altro. Dopo la creazione di un branch figlio le modifiche che nel tempo vengono apportate su questo rimangono completamente separate da quelle che eventualmente continuano ad essere apportate sul padre. Se vengono creati più branch, almeno uno di questi discende da master.

Ovviamente, due branch possono essere uniti tramite un **merge**. Gli algoritmi di merge confrontano due branch riga per riga, manentendo quella appartenente al branch in cui è stata modificata. Se però la stessa riga è stata cambiata in entrambi i branch si crea un conflitto e git richiede all'utente di risolverlo a mano.

Si può passare in qualsiasi momento da un branch all'altro.

4 Comandi principali

Git offre moltissimi comandi, ognuno dei quali svolge un piccolo insieme di funzionalità. Ogni comando (di seguito i principali) è sempre preceduto da "git", ognuno può accettare un diverso numero di argomenti e di opzioni (tratteremo le principali per ogni comando), spesso precedute da un singolo o da un doppio meno.

4.1 init, clone

Per creare un nuovo repository locale si utilizza init (stando all'interno della cartella del progetto del quale si vuole tenere traccia):

```
$ git init prova
Inizializzato repository Git vuoto in /home/user/prova/.git/
```

Per copiare un intero repository remoto si utilizza git clone. Viene scaricato solo il branch master, ma una volta clonato si può accedere a tutti gli altri

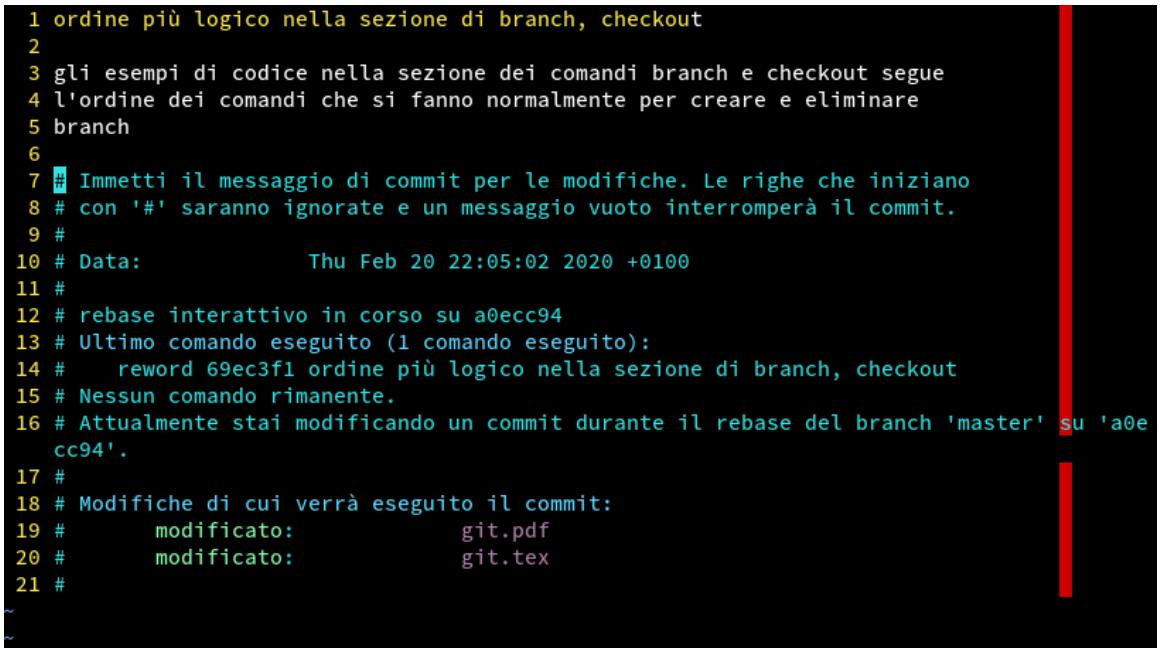

Figure 1: scrittura del testo di un commit in vim

```
$ git clone https://github.com/torvalds/linux.git
```

Per utilizzare tutti gli altri comandi, occorre posizionarsi all' interno della cartella del repository, che viene creata in automatico da clone e init.

4.2 Commit

Ogni volta che si fa una certa quantità di cambiamenti è utile fare un commit, ovvero segnare un punto nello sviluppo a cui sarà sempre possibile tornare.

```
$ git commit
```

Questo comando aprirà l'editor di default di git. Nella prima riga va scritto il titolo del commit. è una buona norma che il titolo mantenga una lunghezza massima di 72 o 80 caratteri e contenga solo nomi e verbi al presente. Dalla seconda riga inizia la descrizione, che può avere una lunghezza molto più ampia per spiegare in modo discorsivo cosa si è fatto, perchè e se eventualmente ha causato dei problemi.

se non si necessita di una descrizione si può utilizzare l'opzione -m ("message"):

```
$ git commit -m "add options page"
```

Ci sono varie teorie sulla lunghezza e il contenuto dei messaggi e delle descrizioni dei commit e su ogni *quanto* si debba committare. In genere un commit deve essere relativo ad un solo argomento e non comprendere modifiche totalmente indipendenti tra di loro. I messaggi di commit non devono essere generici (come "fix crash") altrimenti col passare del tempo sarà impossibile capire cosa si era fatto senza controlare il codice.

L'opzione -s ("signed") aggiunge la firma dell'autore nella descrizione:

```
Signed-off-by: user1 <user1@gmail.com>
```

4.3 Add, rm, reset

I file coinvolti dal commit devono essere prima selezionati con add. In questo modo si entra nella *staging area*. è una copia intermedia tra la cartella locale e il repository che contiene tutte le modifiche che verranno comprese nel prossimo commit.

Se ad esempio si modificano functions.cpp, functions.hpp e main.cpp:

```
$ git add functions.cpp functions.hpp
$ git commit -m "added get function"
```

In questo caso le moficiche di main.cpp non verranno aggiunte al commit "added get function".

Qualsiasi modifica effettuata su functions.cpp o functions.hpp dopo l'utilizzo di add verrebbe anch'essa esclusa dal commit.

Git add si comporta in modo indifferente sia per file appena creati che per le modifiche di file già esistenti.

L'opzione -a ("all") di commit permette di saltare questo passaggio.

Alcune opzioni utili per add:

- `-A` aggiunge qualsiasi modifica all'area di staging
- `.` come -A ma non aggiunge la rimozione dei file
- `-u` non aggiunge i nuovi file

Il comando git rm fa il contrario di add, mentre reset pulisce completamente la staging area.

4.4 Push

Permette di caricare un numero illimitato di commit su un branch di un repository remoto. Le modifiche remote vengono unite a quelle locali.

Git non permette di effettuare un push se il repository remoto contiene commit che non sono presenti su quello locale, perchè sul repository remoto andrebbero persi. Se ci si trova in questa situazione occorre effettuare prima un pull.

```
$ git push
Username for 'https://github.com': Stivvo
Password for 'https://Stivvo@github.com':
To https://github.com/Stivvo/msTest.git
```

```
 ! [rejected]        master -> master (fetch first)
error: push di alcuni riferimenti su 'https://github.com/Stivvo/msTest.git' non riuscito
Gli aggiornamenti sono stati rifiutati perché il remoto contiene delle
modifiche che non hai localmente. Ciò solitamente è causato da un push
da un altro repository allo stesso riferimento. Potresti voler integrare
le modifiche remote (ad es. con 'git pull ...') prima di eseguire
nuovamente il push.
Vedi la 'Nota sui fast forward' in 'git push --help' per ulteriori
dettagli
```

4.5 Merge, Pull, fetch

Merge permette di fondere le modifiche di due branch, locali o remoti (o anche commit diversi dello stesso branch). Se si effettua un pull come suggerito sopra, avverrà infatti un merge. Se git rivela dei conflitti segnala all'utente di risolverli manualmente.

Fetch permette di aggiornare le copie locali ad un repository remoto. In questo modo si creeranno come due copie dello stesso repository, una su cui si sta lavorando offline e una aggiornata alla versione remota. Questo permette effettuare il merge tra la versione remota e quella locale anche in assenza di connessione.

Pull è in sostanza un fetch seguito da un merge, ed è quello che capita di utilzzare puù spesso. In generale pull fonnde ciò che si trova al momento sul repository remoto con quello locale.

Pull e fetch chiamati senza argomenti vanno a prelevare la versione remota del branch su cui si è localmente.

```
<<<<<<< HEAD
class FirstClass {
=======
class SecondClass {
>>>>>>> 4ceb8e7c4fe78b59c00be99418f54280df19078c
```

Questo è il caso in cui mentre il repository locale è rimasto indietro di alcuni commit la classe è stata rinominata in FirstClass. Nel frattempo un altro sviluppatore ha fatto un push di un commit in cui ha rinominato la stessa classe in SecondClass. Quando il primo si trova a dover fare il pull delle modifiche del secondo, per risovere i conflitti di merge deve scegliere tra la versione locale (HEAD) e quella dell'altro, identificata dal codice hash del commit che ha rinominato la classe in SecondClass. Il prossimo push sarà quello del commit di merge, automaticamente creato da git.

4.6 Checkout, Branch, merge

Git branch senza opzioni viene utilizzato per creare un nuovo branch locale. La creazione del branch develop:

```
$ git branch develop
```

Per posizionarsi su develop:

```
$ git checkout develop
M git.pdf
M git.tex
Si è passati al branch 'develop'
```

Se si prova a eseguire un push dal branch appena creato occorre aggiungerlo alla lista dei branch remoti:

```
fatal: Il branch corrente develop non ha alcun branch upstream.
Per eseguire il push del branch corrente ed impostare il remoto come upstream, usa

    git push --set-upstream origin develop
```

L'opzione –all mostra tutti i branch locali e remoti. Il branch seguito dall'asterisco è quello su cui si è posizionati correntemtente.

```
$ git branch --all
* develop
  master
  temp
  remotes/origin/HEAD -> origin/master
  remotes/origin/develop
  remotes/origin/master
```

In questo caso, temp è solamente locale.
L'opzione -d invece elimina un branch. Non è possibile eliminare il branch corrente:

```
$ git branch -d develop
error: Impossibile eliminare il branch 'develop' di cui è stato eseguito
il checkout in '/home/stefano/prog/GitNoob2ProIta'
```

Per eliminare lo stesso branch anche dal repository remoto:

```
$ git push -d origin develop
```

L'opzione -b di checkout crea un branch se quello passato come parametro non esiste.

4.7 Log, status, Diff

commit hash

5 la cartella .git

La cartella .git si trova nella root del repository e contiene tutti i file utilizzati da git, tra cui informazioni sui branch, sui commit. Nei sistemi operativi unix una cartella preceduta da un punto è nascosta e quindi occorre utilizzare il parametro -a di ls per poterla vedere.

```
$ ls .git/
...
$ ls .git/refs
...
```

è molto importante la cartella refs. Contiene:

5.1 heads

In git una head è un riferimento ad un branch o ad un commit di un determinato branch, locale o remoto. Una lista delle head disponibili:

```
ls .git/refs/heads
```

HEAD è un file che punta all'ultimo commit del branch in cui si sta lavorando nel repository locale.

```
cat .git/HEAD
```

Nel caso in cui si voglia ritornare ad un commit precedente si entra nello stato di *deatached head*, ovvero facendo il checkout su uno specifico commit (identificato con il suo codice hash).

esempio di "dare un occhiata, apportare modifiche sperimentali ed eseguirne il commit..."

5.2 Tags

Vengono assegnati ad un gruppo di commit in cui si è raggiunto un traguardo nel progetto (ad esempio 1.3.5).

per visualizzare i tag...

per aggiungere un tag...

5.3 Remotes

Un remote è il percorso di un repository remoto, di solito è quello del repository sul server. Il primo remote che viene utilizzato in un repository viene chiamato di default **origin** (non è obbligatorio), ma ne possono essere aggiunti altri.

... comando

Otteniamo una lista dei remote disponibili. Di default le operazioni come pull sottointendono che si voglia utilizzare il remote origin, ma l'operazione può essere eseguita su qualsiasi altro remote (ad esempio git pull nuovoRemote).

I un remote può essere aggiunto (git remote add), rinominato (git remote rename) o rimosso (git remote remove).

6 Installazione e configurazione

7 Github

8 Approfondimenti

8.1 I submodule

8.2 Revisionare i commit

rebase, ammend

8.3 Gitignore

9 fonti, link utuli

- benefici dei version controllo
- pro git
- tutti i comandi
- vantaggi di git
- norme sulla scrittura dei commit
- che cos'è head
- differenza tra pull e fetch
- annullare add
- parametri di add
- sovrascrivere con pull
- checkout di un branch remoto
- eliminare un branch
- tornare a commit precedenti
- rinominare un branch cos'è head deatached head
- differenza tra head, master e origin
- tipi di head

- push origin head
- uscire da deatached head