Controllo versione semplice con Git

Licenza

Questo lavoro è basato sulla seconda edizione del libro **Pro Git** di *Scott Chacon e Ben Straub* edito da *Apress*. Il libro è disponibile per il download gratuito dal sito ufficiale di Git.

Prefazione

Ho deciso di lanciarmi nella scrittura di questo manuale perchè volevo qualcosa di rapido ma esaustivo che mi permettesse, e che permettesse a chiunque avesse poi intenzione di leggerlo, di fare una carrellata di informazioni fondamentali su Git in poche righe e poco tempo e soprattutto in italiano.

Spero che chiunque utilizzerà questo testo possa trovarlo utile ed esaustivo.

Per qualsiasi segnalazione, suggerimento, lamentela mandatemi liberamente una mail all'indirizzo clementefnc@gmail.com. Su OpenPGP potete trovare la mia chiave GPG.

Riguardo i sistemi di controllo versione (VCS)

Un sistem di controllo versione è un sistema che si occupa di memorizzare le modifiche effettuate ad un file o un insieme di file nel tempo di modo che si possa analizzare questi stati intermedi in un momento futuro ed eventualmente capire dove le cose possono essere andate storte.

Tipologie

Sistemi di controllo versione locali (LVCS)

Molte persone sfruttano cartelle etichettate con date contententi un progetto aggiornato a quel giorno; fondamente delle ridondanze infinite organizzate in cartelle.

Il limite è che basta poco per aprire la cartella sbagliata e rovinare tutto.

Per questo sono stati inventati i sistemi di controllo versione.

Un esempio di VCS che lavora sul nostro filesystem è RCS che lavora memorizzando le differenze tra le varie versione dei file in uno specifico formato; quindi è in grado di spostarsi tra i vari stadi di un progetto aggiungendo o togliendo ciò che è stato modificato nel frattempo.

Sistemi di controllo versione centralizzati (CVCS)

Quando si lavora in gruppo però, come gestire il fatto che le persone posseggono le loro copie del progetto? Ci vuole qualcuno che poi si preoccupi di mettere tutto insieme ed eviti di far casini.

Per questo sono nati i sistemi di controllo versione centralizzati, come ad esempio Subversion e Perforce, che possiedono un singolo server che contiene tutte le versioni dei file. A questi server si connettono un certo numero di clients che scaricano la versione più aggiornata del progetto adl server e poi ricaricano la versione su cui hanno fatto le modifiche.

La criticità di questo sistema è che se il server va down, finchè non viene ripristinato, nessuno può collaborare; se il server viene corrotto o violato, tutto il lavoro è perso eccetto ciò che le persone hanno sulle loro macchine.

Sistemi di controllo versione distribuiti (DVCS)

In un sistema di controllo versione distribuito, come Git, Mercurial, Bazaar o Darcs, il client non solo scarica l'ultimo snapshot del progetto, ma crea una copia completa di ciò che è nella repository, quindi tutta la sua storia. Così nel caso in cui un nodo qualsiasi venisse meno, chiunque potrebbe diventare il nuovo "server" e tutti potrebbero riscaricarsi il progetto e la sua storia.

Ogni client possiede un clone del server e quindi un backup completo dei dati.

Oltretutto questi sistemi lavorano senza problemi con più di una repository remota, in questo modo è possibile collaborare con diversi gruppi di persone in diversi modi, simultaneamente, sullo stesso progetto.

Cos'è Git e come lavora?

Git è un sistema di controllo centralizzato che pensa in modo completamente innovativo ai suoi dati. La maggior parte dei sistemi di controllo versione pensano alle varie versioni di un progetto come a delle differenze rispetto alla versione precedente (*delta-based version control*), Git pensa ai suoi dati come ad una serie di **snapshot** (istantanee) di un piccolo filesystem.

Ogni volta che si effettua un commit o si salva lo stato del progetto, Git effettua una fotografia dello stato attuale dei files. Per essere efficiente, se i file non sono cambiati, non li salva nuovamente, aggiunge soltanto un link alla versione precedente.

Git pensa ai suoi dati come ad una sequenza di snapshots.

La maggior parte delle operazioni effettuate da Git sono locali proprio perchè si ha l'intera storia del progetto memorizzata sul filesystem. Questo significa che si può continuare a lavorare anche quando si va offline.

Ogni cosa salvata in Git viene processata da un algoritmo di checksum; ciò implica che è impossibile cambiare il contenuto di un file o cartella senza che se ne accorga.

Questo rende impossibile corrompere il database senza che Git se ne accorga.

Il meccanismo di checksum utilizzato da git è basato sull'algoritmo di hashing SHA-1. L'algoritmo consta nel processare i file e produrre per ciascuno una stringa esadecimale di 40 caratteri che ne rappresenta una sorta di impronta digitale.

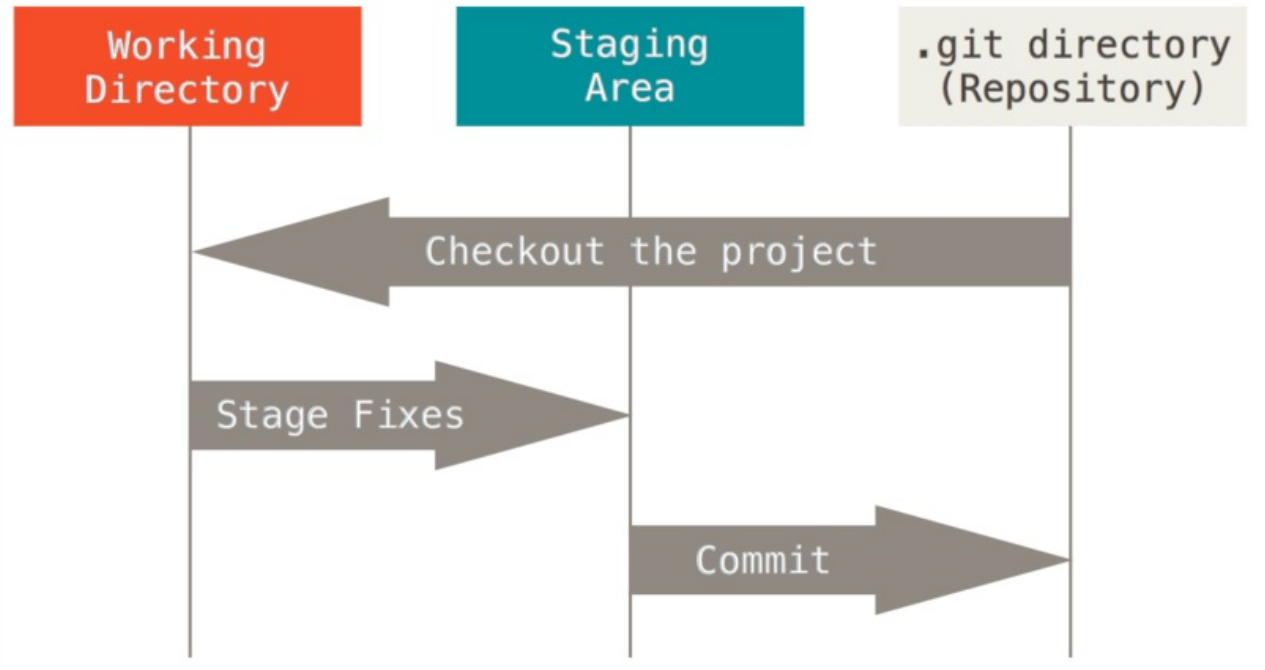
Queste stringhe vengono utilizzate da Git per rappresentare il proprio database; ogni cosa è memorizzata non attraverso il suo nome ma attraverso l'hash del suo contenuto.

I tre stati

Questa è la cosa più importante da capire quando si lavora con Git di modo da non avere perplessità quando si effettua una qualunque operazione.

Git possiede tre stati principali in cui un file può trovarsi:

- Committed: significa che il dato è stato memorizzato in modo sicuro nel database locale
- Modified: significa che il file ha delle modifiche rispetto alla versione precedente ma non è stato ancora fatto il commit
- Staged: significa che le modifiche della versione attuale rispetto alla precedente sono state segnalate e che questa versione è pronta per essere inserita nel prossimo commit



Questo ci porta a parlare anche delle tre sezioni principali di un progetto Git: la git directory, il working tree e la staging area.

- Git directory: è dove Git memorizza i metadati e gli oggetti del database per il nostro progetto; è la parte più importante per Git ed è ciò che viene copiato quando si *clona* una repository da un altro computer
- Working tree: è un singolo checkout di una versione del progetto; questi file sono estratti da un database compresso nella git directory e posizionate sul disco per essere utilizzati o modificati
- Staging area: è un file, generalmente contenuto nella git directory, che memorizza informazioni circa cosa verrà inserito nel prossimo commit; in gergo è detto *index*

Generalmente il workflow con Git è il seguente:

- Si modifica un file nel working tree
- Si selezionano solo i file le cui modifiche si vogliono inserire nel prossimo commit; questo aggiunge **solo** questi cambiamenti alla staging area
- Si effettua il commit che prende i file della staging area e ne memorizza uno snapshot nella git directory

Una versione particolare di un file che è nella git directory, è **committed**. Se è stata modificata ed aggiunta alla staging area allora è **staged**. Se sono stati fatte delle modifiche ma non è stato aggiunto alla staging area allora è **modified**.

La riga di comando

Ci sono vari modi per usare Git. È stato però pensato per essere utilizzato da riga di comando, nonostante esistano ormai svariati tool grafici che ne rendono l'uso più semplice pre i meno avvezzi al terminale.

D'ora in avanti si da per scontato che ci si sappia quantomeno destreggiare con i comandi basi di una shell bash.

Intallare Git

Prima di iniziare ad utilizzarlo, è necessario installare Git. In alcuni casi è già installato. Nel caso in cui non fosse così fate riferimento al vostro package manager se siete su Linux; se siete su MacOS lo trovate nei developer tool che, se non avete già installato, potete installare semplicemente digitando `git` e premendo invio di modo che vi verrà chiesto di installare i developer tools; per installarlo su windows far riferimento alle istruzioni sul sito ufficiale; infine è possibile anche compilarselo autonomamente.

Primo avvio e setup

Ora che abbiamo installato Git sul nostro sistema è necessario fare un paio di cose per configurare il nostro ambiente. Questa procedura si deve fare soltanto una volta su ogni pc su cui installiamo git.

Git ha un tool chiamato `git config` che permette di visualizzare ed impostare le variabili che controllano ogni aspetto relativo alle modalità in cui git opera. Queste variabili possono essere memorizzate in tre punti diversi ed hanno un significato diverso a seconda di dove sono posizionate.

- `/etc/gitconfig`: contiene i parametri che si applicano ad ogni utente del sistema. Si manipola con l'opzione `--system` passata a `git config`.
- `~/.gitconfig` o `~/.config/git/config`: questo file contiene i valori specifici di un singolo utente. Si manipola con l'opzione `--global` passata a `git config`.
- `config`: questo file si trova nella Git directory ed è specifico della repository corrente

Ogni livello sovrascrive ciò che è nel livello precedente, per cui la configurazione nella Git directory, se presente, è la principale.

La nostra identità

La prima cosa da fare è configurare il nostro nome ed indirizzo email. Questo è importante perchè ogni commit conterrà questa informazione. Ora diamo i seguenti comandi sostituendo nome e mail con le nostre informazioni personali

```
$ git config --global user.name "clementefnc"
$ git config --global user.email clementefnc@gmail.com
```

Fatto la prima volta non sarà più necessario rifarlo, a meno che non si vogliano modificare o ci si trovi su un nuovo computer.

Fatto ciò, volendo si può impostare anche l'editor di default che ci verrà mostrato da git ogni volta che sarà necessario inserire un messaggio. Se non viene configurato, usa l'editor di default del sistema. Nel caso in cui ne volessimo uno differente come ad esempio Vim possiamo dare il seguente comando

```
$ git config --global core.editor vim
```

Per controllare lo stato attuale della configurazione basterà utilizzare il comando `git config --list` che produrrà un output simile al seguente

```
$ git config --list
user.name=clementefnc
user.email=clementefnc@gmail.com
core.editor=vim
```

Volendo si può anche vedere uno specifico valore

```
$ git config user.name
clementefnc
```

Ottenere aiuto

Qualora fosse necessario ottenere aiuto su Git, ci sono due modi per ottenere la pagina del manuale relativa per ogni comando git:

```
$ git help <verb>
$ man git-<verb>
```

Ad esempio qualora si volesse la pagina del manuale relativa a `git config` basterà digitare

```
$ git help config
```

In alternativa, nel caso in cui non si voglia l'intera manpage ma un semplice richiamo veloce è possibile chiedere un output conciso con le opzioni `-h` oppure `--help`.

```
$ git add -h
usage: git add [<options>] [--] <pathspec>...

    -n, --dry-run         dry run
    -v, --verbose         be verbose

    -i, --interactive     interactive picking
    -p, --patch           select hunks interactively
    -e, --edit            edit current diff and apply
    -f, --force           allow adding otherwise ignored files
    -u, --update          update tracked files
    --renormalize         renormalize EOL of tracked files (implies -u)
    -N, --intent-to-add   record only the fact that the path will be added
later
    -A, --all             add changes from all tracked and untracked files
    --ignore-removal      ignore paths removed in the working tree (same as -
-no-all)
    --refresh             don't add, only refresh the index
    --ignore-errors       just skip files which cannot be added because of
errors
    --ignore-missing      check if - even missing - files are ignored in dry
run
    --chmod (+|-)x        override the executable bit of the listed files
```

Le basi di Git

Se volessi leggere un solo capitolo per iniziare ad usare Git, è questo. Una volta finito sarai in grado di fare tutte le cose che si fanno per la maggior parte del tempo in git: inizializzare e configurare una repository, iniziare e smettere di tracciare file, effettuare lo stage e commit delle modifiche, ignorare alcuni file o pattern, riparare ad errori, visualizzare la storia del progetto e le modifiche effettuate tra i

vari commit ed infine come effettuare *push* e *pull* dalle repository remote.

Ottenere una repository Git

Ci sono fondamentalmente due modi per ottenere una repository Git:

- Prendere una directory locale che non è sotto controllo versione e trasformarla in una repository Git
- Clonare una repository Git esistente altrove

In entrambi i casi ci si ritroverà con una repository Git sul proprio pc pronta ad essere utilizzata.

Inizializzare una repository in una cartella esistente

Se si ha una cartella contenente un progetto che non è sotto controllo versione e si vuole iniziare ad usare Git per farlo, prima di tutto bisogna recarsi nella cartella del progetto e digitare il comando

```
$ git init
```

Questo creerà una nuova sottodirectory chiamata `.git` che contiene tutti i file necessari a Git per funzionare, il suo scheletro.

A questo punto nulla è *tracked*.

Se si vuole iniziare ad effettuare controllo versione sui files esistenti, è necessario iniziare ad effettuare il *tracking* di questi files ed effettuare un commit iniziale.

Ad esempio con i seguenti comandi:

```
$ git add *.c
$ git add LICENSE
$ git commit -m 'stato iniziale del progetto'
```

Vedremo a cosa servono questi comandi tra poco. Ora avremo una repository Git con dei file *tracked* ed un commit iniziale.

Clonare una repository esistente

Se si vuole ottenere una copia di una repository Git esistente, ad esempio di un progetto al quale si vuole contribuire, il comando necessario è

```
$ git clone <url>
```

In questo modo si cloneranno tutti i dati di una data repository presenti su un server. Questo è uno degli aspetti caratteristici di git, come dicevamo all'inizio; in questo modo possiedermo tutta la storia del progetto sul nostro computer esattamente come è presente sul server.

Nel caso in cui volessimo clonare ad esempio la libreria `libgit2` è possibile con il comando

```
$ git clone https://github.com/libgit2/libgit2
```

Questo creerà una cartella chiamata `libgit2`, inizializzera una `.git` directory al suo interno ed effettuerà un *pull* dei dati relativi a questa repository ed infine farà un *check out* della *working copy* dell'ultima versione.

Fondamentalmente entrando nella cartella `libgit2` si troveranno i file di progetto pronti ad essere utilizzati.

Nel caso in cui si voglia clonare una repository in una cartella con differente nome è possibile con la seguente sintassi

```
$ git clone https://github.com/libgit2/libgit2 mylibgit
```

Git supporta svariati protocolli di trasferimento: nei comandi sopra si usa `https`, ma è possibile utilizzare anche `ssh` (con url del tipo `git://` oppure `user@server:path/al/repo.git`) o altri.

Memorizzare i cambiamenti nella repository

A questo punto si avrà una repository Git sul computer ed un *checkout* o *working copy* di tutti i suoi files.

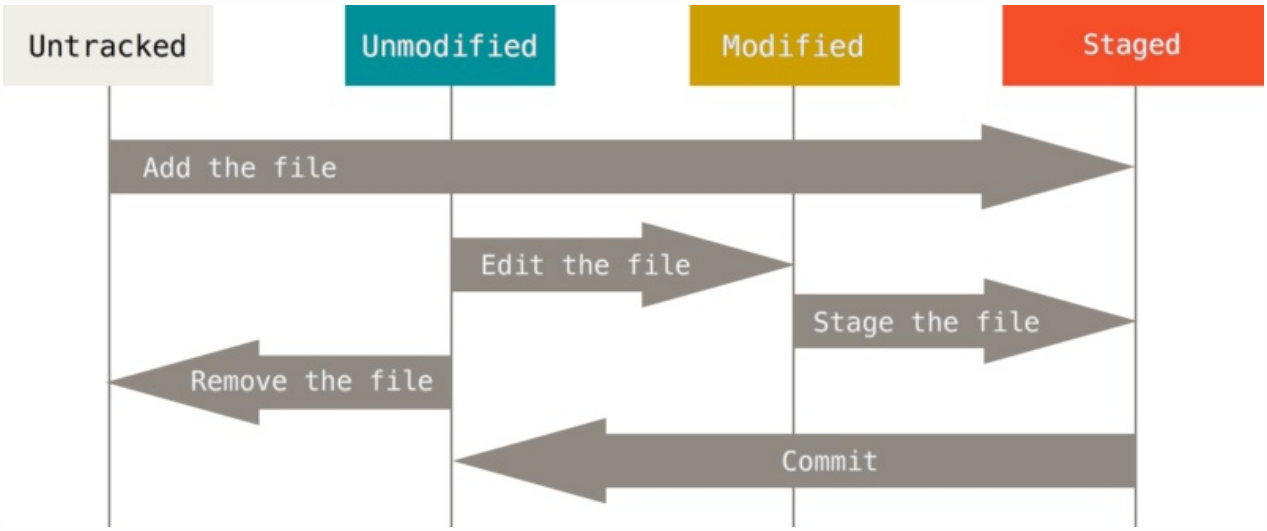
Tipicamente si vorranno effettuare modifiche a questi file ed effettuare commit degli *snapshot* di questi cambiamenti nel repository ogni qual volta il progetto avrà raggiunto uno stadio che si vuole salvare.

Ricordiamo che ogni file può essere in uno dei due stati: *tracked* o *untracked*. I primi sono quelli che erano nell'ultimo *snapshot* e possono essere *unmodified*, *modified* o *staged*. Brevemente, quelli *tracked* sono i files di cui git è "a conoscenza".

Gli *untracked* sono invece tutti gli altri, praticamente qualsiasi file che è nella *working directory* che non era nell'ultimo *snapshot* e non è nella *staging area*.

Quando si clona una repository, tutti i file saranno *tracked* e *unmodified* perchè Git ha appena effettuato un *checkout* e non si è ancora effettuata alcuna modifica.

Quando si modificano i files, Git li mostra come *modified* perché sono stati cambiati rispetto all'ultimo commit. Man mano che si lavora questi file diventeranno *staged* e poi se ne effettuerà il commit di tutti i cambiamenti *staged* ed il ciclo si ripete.


Untracked
Unmodified
Modified
Staged
Add the file
Edit the file
Stage the file
Remove the file
Commit