Guida al sistema di versionamento GIT

Software Synthesis

Indice

1 Introduzione al sistema

In questa sezione illustrerò le principali nozioni da conoscere per un corretto uso del sistema di versionamento Git.

1.1 Un po di storia

Inizialmente per tenere traccia delle versioni di un software i programmatori adottavano il cosiddetto Sistema di Controllo di Versione Locale che consisteva nel copiare, manualmente, una specifica versione in una nuova directory. Il sistema è molto semplice ma anche soggetto a molti errori.

Per risolvere il problema, si svilupparono i Sistemi di Controllo di Versione (Version Control System, VCS), formati da un database che manteneva tutti cambiamenti dei file sotto controllo di versione.

Successivamente si dovette risolvere il problema che programmatori su uno stesso progetto dovevano collaborare tra loro, vennero cosi sviluppati i Sistemi di Controllo di Versione Distribuiti (Centralized Version Control System, CVCS). Questi sistemi hanno un unico server centrale che contiene tutte le versioni e gli utenti che possono amministrarli. Per alcuni anni questo fu lo standard maggiormente utilizzato.

Questo sistema presenta i seguenti vantaggi:

- chiunque sa, ad un certo punto dello sviluppo, cosa stanno facendo gli altri utenti;
- gli amministratori hanno un controllo immediato e preciso su chi può fare cosa;
- è più facile amministrare un database centrale che non uno su ogni client.

Ma presenta pure dei svantaggi:

- Il progetto è memorizzato in unico punto, quindi aumentano le possibilità di perdere tutto il lavoro, o di rallentarlo in caso di guasti momentanei al server.

Infine, memori dell'esperienza, si crearono i Sistemi di Controllo di Versione Distribuiti, (Distribuited Version Control System, DVCS) ovvero ogni client possiede una copia locale dell'intero repository. In questo modo se si blocca un server la copia di un qualsiasi client può essere usata per ripristinare la copia nel server.

Parliamo ora un pò più nel dettaglio di Git. Git è un DVCS scritto principalmente in C, esso nacque nel 2005 dopo la frattura nel rapporto tra la comunità che ha sviluppato il kernel Linux e la società che ha sviluppato BitKeeper. Questa frattura permise a Linus Torvalds di sviluppare un proprio strumento avendo come obbiettivi:

- velocita;
- design semplice;
- forte supporto allo sviluppo non-lineare (possibilità di migliaia di rami paralleli);
- completamente distribuito;
- capacità di gestire, in modo efficiente (velocità e dimensione dei dati), grandi progetti come il kernel Linux.

Una curiosità su questo sistema rigurda il nome ad esso associato. Infatti il nome dato da Linus Torvalds è un termine gergale britannico per indicare una persona stupida o sgradevole, infatti venne cosi motivato dallo stesso autore:

> "Sono un egoista bastardo, e do a tutti i miei progetti un nome che mi riguardi. Prima Linux, ora Git".

Il wiki ufficiale tuttavia fornisce spiegazioni alternative. Per esempio, a causa della difficoltà di utilizzo delle prime versioni, il programma è stato definito:

> "Il sistema di controllo versione progettato per farti sentire più stupido di quanto tu non sia."

1.2 Snapshot non differenze

La principale differenza tra Git ed altri VCS consiste nel modo in cui Git memorizza i cambiamenti apportati al codice sorgente. Comunemente i VCS memorizzano le informazioni che essi mantengono come un insieme di file, con le relative modifiche apportate nel corso del tempo. Più precisamente vengono memorizzati nuovamente anche i file che non hanno subito modifiche.

Git non considera e né immagazzina i dati in questo modo; piuttosto li considera come una serie di istantanee (snapshot) di un mini file-system. Più precisamente, ogni volta che viene eseguito un commit, Git fa una immagine dello stato corrente salvando un riferimento allo snapshot. Per essere efficiente se alcuni file non sono stati cambiati, non li memorizza nuovamente, ma crea un collegamento a file già esistenti.

1.3 Operazioni off-line

In Git la maggior parte delle operazioni avviene off-line. Molti altri sistemi, che rientrano sotto i CVCS, non permettono di svolegere operazioni sul progetto se non si è collegati alla rete, ponendo cosi dei limiti di utilizzo.

Con Git invece si possono svolgere operazioni come la consultazione della storia e/o commit e/o ritorno a versioni precedenti ecc. anche senza essere connessi alla rete. Si potrà in un successivo momento inoltrare agli altri collaboratori le modifiche apportate al progetto.

1.4 Integrità

Git è un sistema integro perchè ogni cosa è controllata, tramite checksum, prima di essere salvata ed è referenziata da un checksum. Ciò significa che non è possibile modificare il contenuto di una qualsiasi file o directory senza che il sistema non se ne accorga.

Il meccanismo utilizzato da Git per il checksum è un hash, denominato SHA-1. Si tratta di una stringa di 40 caratteri, composta da caratteri esadecimali, calcolata in base al contenuto del file o della struttura di directory in Git. Il sistema immagazzina ogni cosa nel proprio database non per nome di file ma tramite codice hash SHA-1.

1.5 I tre stati fondamentali

Git possiede tra stati fondamentali in cui possono trovarsi i file:

committed ovvero il file è immagazzinato e al sicuro nel database locale;

modified ovvero il file è stato modificato ma non ancora inserito nel database locale;

staged ovvero che un file modificato è stato contrassegnato per essere inserito nello snapshot alla prossima operazione di commit;

di conseguenza l'ambiente è suddiviso anch'esso in tre zone:

- la directory di Git;
- la directory di lavoro;
- l'area di staging;

La directory di Git è il luogo dove sono memorizzati i metadati e il database degli oggetti. Questa è la parte fondamentale del repository ed è quella che viene copiata quando si esegue la clonazione del repository da un client ad un altro. Si tratta della cartella nascosta ".git".

La directory di lavoro è un singolo checkout di una versione del progetto. Sono file estratti dal database compresso, nella directory di Git, e posizionati nel disco per poter essere usati o modificati.

L'area di staging consiste in un file, che risiede nella cartella di Git, contenenete le informazioni per il successivo commit.

Da quanto appreso possiamo dire che il normale flusso di lavoro consiste in:

1. modifica di un file nella directory di lavoro;
2. esecuzione dell'operazione di staging per i file modificati;
3. esecuzione dell'operazione di commit per memorizzare in modo permanente i dati.

2 Installazione

Verrà ora illustrato come installare il sistema di versionamento Git nei più comuni sistemi operativi.

2.1 Linux Debian/Ubuntu

Per installare Git nel sistema operativo Linux Ubuntu è sufficiente, dopo aver aperto una finestra di terminale, digitare ed eseguire il seguente comando:

```
sudo apt-get install git-core
```

verranno richieste le credenziali di accesso dell'utente root in quanto si sta modificando la configurazione del sistema.

2.2 Linux Fedora

Per installare Git nel sistema operativo Linux Fedora è sufficiente, dopo aver aperto una finestra di terminale, digitare ed eseguire il seguente comando:

```
su
inserire password utente di root
yum install git
```

2.3 Mac

Il più semplice modo per installare Git su Mac è usare l'installatore grafico scaricabile dalla pagina Google Code:

```
http://code.google.com/p/git-osx-installar
```

2.4 Microsft Windows

Per installare Git su sistema operativo Microsoft Windows scaricare e lanciare l'installatore reperibile al seguente indirizzo:

```
http://code.google.com/p/msysgit
```

Dopo averlo installato si avrà a disposizione la versione a riga di comando, incluso un client con l'interfaccia grafica standard.

3 Configurazione iniziale

Verrà ora illustrato come configurare alcune funzioni di Git secondo le esigenze personali del programmatore.

Lo strumento che permette di configuare Git si chiama "git config" e consiste in variabili che contengono i valori delle nostre configurazioni. Queste variabili possono trovarsi in un:

- file "/etc/gitconfig" che contiene i valori per ogni utente del sistema e per tutti i repository configurati. Per leggere/scrivere questo file passiamo l'opzione "−−system" a "git config";
- "file ˜/.gitconfig" che contiene i valori per uno specifico utente del sistema e per i repository da lui configurati. Per leggere/scrivere questo file passiamo l'opzione "−−global" a "git config";
- file di configurazione locale che risiede nella cartella nascosta ".git" di ogni repository.

I valori di ogni livello sovrascrivono quelli di livello più alto, nello specifico se impostiamo un valore a livello di sistema e lo modifichiamo a livello locale di uno specifico repository, quello locale prevarrà su quello di sistema.

Su sistemi Windows il file ".gitconfig" si trova generalmente nella cartella:

```
C:\Documents and Settings \User
```

per quanto riguarda il file "/etc/gitconfig" si trova all'interno della directory "MSys" e dipende da dove l'utente sceglie di installare il software.

3.1 Identità personale

Git necessita del nome utente e del suo indirizzo e-mail per poter svolgere le operazioni di commit. Per istruire Git con queste informazioni digitiamo i seguenti comandi:

```
git config --global user.name ''Cognome Nome''
git config --global user.email ''indirizzo e-mail''
```

Se si necessita di un nome e un indirizzo e-mail particolare per un progetto specifico rieseguire i comandi precedenti nella cartella del particolare progetto omettendo però l'opzione "−−global". I valori cosi inseriti varranno per quel determinato progetto.

3.2 Editor di testo

E' possibile personalizzare anche l'editor di testo utilizzato per inserire messaggi in Git. Generalmente Git utilizza Vi o Vim ma è possibile specificarne un altro con il comando:

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

E' possibile sostituire "gedit" con il proprio editor predefinito. Per l'opzione "−−global" vale quanto detto in precedenza.

3.3 Diff personalizzato

Lo strumento di "diff" viene usato da Git per risolvere i conflitti durante una fusione (merge). Per impostarne uno digitare il seguente comando:

```
git config --global merge.tool tool
```

dove "tool" va sostituito con uno dei seguenti valori:

- vimdiff;
- kdeff3;
- tkdiff;
- meld;
- xxdiff;
- emerge;
- gvimdiff;
- ecmerge;
- opendiff.

Per l'opzione "−−global" vale quanto detto in precedenza.

3.4 Alias

Se con il proprio progetto ci si appoggia a server esterni, come github.com, per condividere il progetto con altri collaboratori è possibile definire un alias per evitare di ripetere l'inserimento dell'URL ad ogni invio/ricezione di modifiche.

Per definire un alias eseguire il seguente comando all'interno di uno specifico repository:

```
git remote add alias URL
```

dove "alias" va sostituito con un nome significativo a piacere mentre "URL" va sostituito con l'indirizzo remoto dove risiede il repository.

Per rimuovere un alias è sufficiente il seguente comando all'interno del repository che contiene l'alias da eliminare:

```
git remote rm alias
```

dove "alias" va sostituito con l'alias da rimuovere.

Per visualizzare gli alias configurati in un particolare repository digitare all'interno di esso, il comando:

```
git remote -v
```

l'opzione "-v" specifica che oltre ai nomi vogliamo visionare l'URL.

3.5 Visualizzazione dei parametri configurati

Per visualizzare i valori configurati digitare il seguente comando:

```
git config --list
```

Invece se si vuole visualizzare un valore specifico digitare:

```
git config user.name
```

3.6 Colorazione degli stati su shell

Per aiutarsi a riconoscere più velocemente i diversi stati in cui si trovano i file sorgenti e i diversi branch, Git li può colorare, nel terminale, con colorazioni differenti. Per attivare questa funzionalità eseguire i seguenti comandi:

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

4 Lavorare con repository Git

In questa sezione si vuole illustrare i comandi base per un corretto utilizzo di un repository Git.

4.1 Creazione del repository locale

Quando iniziamo a lavorare ad un progetto ci si presentano due situazioni mutualmente esclusive:

- il progetto è creato da noi, quindi un nuovo repository deve essere inizializzato;
- il progetto è già esistente, quindi dobbiamo clonare in locale un repository esistente.

Se ci troviamo nella prima situazione quello che dobbiamo fare per inizializzare un nuovo repository è quella di creare una cartella che ospiti la cartella di Git e l'area di lavoro, successivamente ci spostiamo al suo interno con una finestra di terminale e lanciamo il seguente comando:

```
git init
```

al termine avremo una cartella nascosta denominata ".git", questa è la cartella di Git.

Se invece siamo nel secondo caso, in particolare significa che entriamo come collaboratori in un progetto, dobbiamo spostarci nella cartella che conterrà il pacchetto, ossia la cartella di Git e l'ambiente di lavoro, e lanciare il seguente comando:

```
git clone URL NomeDirectory
```

dove "URL" va sostituita con l'URL del repository remoto. L'opzione "NomeDirectory" specifica il nome della directory locale che conterrà il repository, se omessa il nome sarà quello del server remoto.

4.2 Cambiamenti al repository

Ogni file nella tua directory è sempre in uno dei seguenti stati:

- tracciato;
- non tracciato;

I file tracciati sono presenti nell'ultimo snapshot e possono essere non modificati, modificati o parcheggiati (in stage).

I file non tracciati sono tutti gli altri.

Quando cloni un repository tutti i file al suo interno sono tracciati e non modificati.

4.3 Controllare lo stato del repository

Il comando da utilizzare per vedere lo stato del repository è:

```
git status
```

Se come risultato si ottiene *On branch master nothing to commit (working directory clean)*, significa che la cartella è pulita ossia Git non ha rilevato la presenza di nuovi file e/o file modificati.

Il comando può visualizzare inoltre *On branch master Untracked file* seguito dall'elenco dei file non tracciati, ossia non presenti nello snapshot precedente. I file non tracciati resteranno in questo stato finchè non diremo esplicitamente a Git di tracciarli; questo serve per impedire che vengano aggiunti in automatico file di compilazione o simili.

4.4 Tracciamento dei file sorgente

Dopo aver creato o modificato un file sorgente dobbiamo informare Git che vogliamo versionare tale/i file/s e questo avviene con il comando:

```
git add NomeFile
```

dove con NomeFile si intende il percorso, dalla cartella principale del repository, del file da tracciare. Se NomeFile è una directory, il comando aggiungerà ricorsivamente tutti i file presenti in essa.

Per aggiungere in staging tutti i file, senza dover richiamare il comando "add" su ogni file è possibile digitare il seguente overloading:

```
git add .
```

Se visualizziamo ora lo stato del repository notiamo che Git ci informa che alcuni file sono in stage. Lo si nota dal risulatato del comando di visualizzazione dello stato *On branch master Changes to be committed*. Alla prossima operazione di commit il file verrà aggiunto alla versione registrata quando lo abbiamo tracciato con il comando "git add". Se nel frattempo avessimo modificato tale/i file/s dobbiamo ripetere il comando "git add" per tale/i file/s. Quest'ultima situazione è rappresentata dal fatto che se richiediamo lo stato ottenianmo la sezione *Changed but not updated*.

Ricordarsi di tracciare il file sorgente dopo ogni modifica.

Se per errore si sono introdotti nello stage file che non devono essere versionati è possibile rimuoverli prima dell'operazione di commit tramite il comando:

```
git reset HEAD NomeFile
```

4.5 Ignorare file nel tracciamento

Spesso si avranno classi di file che non si vuole automaticamente aggiungere o far vedere a Git come file non tracciati, come ad esempio file di log o risultati di compilazioni. In questi casi si può creare nella directory radice del repository un particolare file chiamato ".gitignore". Questo file conterrà il nome dei file o le classi di file che git deve ignorare durante la fase di staging e di commit.

Le regole per comporre questo file sono le seguenti:

- le linee che iniziano con il cancelletto (#) sono commenti, quindi ignorati;
- espressioni regolari utilizzate dalla shell per individuare file o gruppi di file;
- per indicare cartelle di file da ignorare terminalre la riga con il carattere slash (/);
- per negare un pattern farlo precedere da un punto esclamativo (!)

Questo è un esempio di file .gitignore:

```
# un commetto - questo sarà ignorato
.a
!lib.a
/TODO
build/
doc/*.txt
```

La prima riga rappresenta un commento e come dice verrà ignorata, la seconda riga dice di non includere tutti i file con estensione "a", la terza riga dice di ammettere il file "lib.a", la quarta riga ignora solamente il file "TODO" nella directory principale, la quinta riga dice di non includere la sottodirectory "build" con i relativi file mentre l'ultima riga dice di ignorare tutti i file con estensione ".txt" che risiedono all'ienterno della sottodirectory "doc" presente nella cartella principale.

Dopo aver composto questo file siamo pronti per informare Git che deve consultare questo file durante le operazioni di staging e commit, dalla cartella principale diamo il seguente comando:

```
git config core.excludefile /.gitignore
```

Ora si può tranquillamente usare la versione del comando "add" che include tutto senza correre il rischio di includere file che non vanno inclusi.

4.6 Eseguire il commit dei file in staging

Dopo aver aggiunto i file nell'area di stage si possono versionare tramite l'operazione di commit.

Per eseguire il commit è sufficiente lanciare il comando:

```
git commit
```

ottenendo come risultato la visualizzazione dell'editor predefinito per l'inserimento del messaggio di commit. Il file contiene come messaggio predefinito, commentato, l'ultimo output del comando di visualizzazione dello stato e la prima riga vuota. Nella prima riga si dovrà inserire il testo del messaggio. Si ricordi inoltre che lasciare la riga vuota farà fallire l'operazione di commit.

Si può comunque eseguire il tutto dalla riga di comando, attraverso il seguente comando:

```
git commit -m ''Messaggio del commit''
```

dove l'opzione "-m" indica che il testo racchiuso tra le virgilette sarà considerato come il testo dell'operazione di commit. Anche qui lasciare vuoto il campo farà fallire l'operazione.

4.7 Rimozione di un file

Per rimuovere un file dal repository è necessario prima rimuoverlo dai file tracciati e poi fare il commit.

Con il comando:

```
git rm NomeFile
```

Git esegue le due precedenti azioni e lo rimuove anche dalla directory di lavoro.

4.8 Movimenti di file

A differenza di altri VCS, Git non traccia esplicitamente i movimenti di file. Se rinomini un file in Git nessun metadata che ti dirà che il file è stato rinominato è immagazzinato. Per eseguire correttamente questa operazione dobbiamo utilizzare il comando:

```
git mv FileFrom FileTo
```

dove con "FileFrom" e "FileTo" si intendo i percorsi di origine e destinazione del file, che può non solo essere spostato nell'albero ma anche rinominato. Il precedente comando è comodo perchè è come se ne eseguissimo tre, più precisamento sono:

```
mv FileFrom FileTo
  git rm FileFrom
   git add FileTo
```

5 Visualizzazione della storia

In questa sezione verrà visualizzato come navigare nella storia del progetto, sia da shell che con una intefaccia grafica.

5.1 Tramite shell dei comandi

Per visualizzare la storia dei comandi tramite shell il comando da usare è il seguente:

```
git log
```

questo comando è quello fondamentale che mostra tutta la storia del repository, è inoltre possibile passare ulteriori parametri per affinare la ricerca come per esempio:

```
git log -n
```

dove "n" va sostituito con un numero naturale (maggiore di zero), e Git ci restituirà gli utlimi "n" commit. Oppure per avere ogni commit su una singola riga:

```
git log --pretty=oneline
```

Ma ve ne sono molti altri che è più utile imparare con la pratica che non con un elenco fine a se stesso.

5.2 Tramite interfaccia grafica

Per visualizzare la storia del repository tramite un interfaccia grafica eseguire il comando:

```
gitk
```

dopo essere stata avviata è possibile navigare nella storia ed eseguire ricerche.

6 Annullamento di procedure

In questa sezione viene illustrato come eseguire l'annullamento di alcune procedure già portate a termine.

6.1 Modifica dell'ultimo commit

Uno degli annullamenti più comuni è quando invii troppo presto un commit e magari dimentichi di aggiungere alcuni file, o dimentichi di inserire una parte del messaggio. Per correggere questi tipi di errori si utilizza il comando:

```
git commit --amend
```

Questo comando prende l'area di staging e la usa per il commit, quindi se abbiamo dimenticato alcuni file vanno aggiunti con il comando di "add" prima di lanciare questo. Se l'area di staging non è stata modificata allora possiamo cambiare il messaggio relativo al commit, infatti dopo aver digitato il comando si aprirà l'editor con il vecchio messaggio che avevamo inserito, lo modifichiamo e salviamo il tutto per ultimare la fase di commit.

6.2 Rimozione di file dall'area di staging

Può capitare di aggiungere all'area di staging file che dovrebbero essere committati in due passi separati. E' possibile rimuovere i file estranei con il seguente comando:

```
git reset HEAD NomeFile
```

il risultato non è la rimozione fisica del file ma la rimozione del file dall'area di staging, esso potrà essere aggiunto in un secondo momento tramite il comando di "add".

6.3 Annullare le modifiche ad un file

Se si volesse rimuovere le modifiche fatte ad un file sorgente e riportare il file a come si trovava nel commit precedente dobbiamo usare il seguente comando:

```
git checkout HEAD ~1 NomeFile
```

dove il numero successivo al carattere tilde (~) rappresenta il numero di quante revisioni vogliamo tornare indietro. Successivamente il file dovrà essere committato.

Si deve osservare che è un comando **PERICOLOSO**: in quanto ogni modifica a quel file è sparita, semplicemente il sistema ha sovrascritto il file presente nell'area di lavoro.

7 Repository remoti

In questa sezione verrà illustrato come condividere le modifiche al repository locale con uno remoto e ricevere i cambiamenti da esso.

7.1 Aggiornare la sorgente remota

Quando vogliamo che i nostri collaboratori abbiano in locale le nostre modifiche dobbiamo aggiornare prima il repository remoto che essi useranno per aggiornare il loro locale. Per aggiornare usiamo il comando:

```
git push URL/Alias NomeRamo
```

Dove con URL/Alias si intende l'indirizzo del repository remoto o l'alias ad esso associato, mentre NomeRamo è il ramo del repository remoto che vogliamo aggiornare, la maggior parte delle volte sarà il ramo principale, ossia il ramo *master*.

Se il comando dovesse essere rifiutato, la causa molto probabilmente sarà che contemporaneamente a noi qualcun altro ha fatto la stessa operazione, quindi noi prima dobbiamo scaricarci in locale i suoi aggiornamenti dopo di che possiamo ripetere la procedura di aggiornamento della sorgente remota.

7.2 Prelevare da sorgenti remote

Per aggiornare la nostra copia locale con i dati della sorgente remota dobbiamo eseguire il seguente comando nella directory principale del nostro repository:

```
git pull URL/Alias NomeRamo
```

dove con "URL/Alias" si intende l'URL del repository remoto oppure il suo alias mentre con "NomeRamo" si intende da quale ramo del repository remoto si vuole prelevare le informazioni.

Il comando scarica i cambiamenti presenti nel server e li fonde con i nostri file nell'area di lavoro.

8 Tagging

Come la maggior parte dei VCS, Git offre la possibilità di aggiungere dei tag, ossia dei riferimenti, a dei punti specifici del progetto che sono ritenuti importanti. In genere si usano per marcare i punti di rilascio del software.

8.1 Elencare i propri tag

Per visualizzare i propri tag è sufficiente digitare il seguente comando:

```
git tag
```

il risultato sarà un elenco, ordinato alfabeticamente, dei tag creati per il progetto.

8.2 Creazione di tag

Prima di illustrare il comando per creare un tag è importante sapere che in Git esistono due tipi di tag:

- semplificati (lightweight);
- commentati (annotated).

Un tag "semplificato" è semplicemente un riferimento ad uno specifico commit, mentre un tag "commentato" è un vero e proprio oggetto memorizzato nel database interno, ne viene calcolato il checksum e possiede un proprietario, proprio come un commit. Quest'ultimo tipo di tag può essere firmato e verificato con GNU privacyguard (GPG)

Per la creazione di tag "commentati" eseguire il comando:

```
git tag -a v1.0 -m ''Messaggio da allegare''
```

l'opzione -a sta per "annotated" mentre "v1.4" è in nome del tag, l'ultimo parametro "-m" significa che verrà allegato come messaggio quello contenuto tra doppi apici. Se non viene specificato un messaggio verrà lanciato il tuo editor per inserirne uno.

Per avere informazioni su uno specifico tag, eseguire il seguente comando:

```
git show v1.0
```

Per firmare un tag "annotated", tramite GPG e assumendo di possedere una chiave privata, lanciare il comando di creazione di un tag "annotated" passando come opzione "−s" al posto di −a.

Per creare un tag "lightweight" si usa il comando:

```
git tag v1.1
```

ovvero lo stesso per i tag "annotated" senza nessuna opzione.

8.3 Verifica di tag

Se si volesse verificare la veridicità di un tag "annotated" bisogna disporre della chiave pubblica di colui che la ha creata nel proprio portachiavi e lanciare il comando:

```
git tag -v 1.4
```

dove "1.4" andrà sostituito con il nome del tag da validare.

8.4 Condivisione di tag

I tag non vengono automaticamente trasferiti nelle sorgenti remote, per trasferirli usare il comando:

```
git push URL/Alias NomeTag
```

dove con "URL/Alias" si intende l'URL o l'alias della sorgente remota mentre "NomeTag" è il nome del tag da trasferire.

9 Branching

Ogni VCS ha un suo modo di supportare il branching, ossia quando ci si distanzia dal flusso principale di sviluppo per effettuare dei test senza però, effettuare dei pasticci nel ramo principale.

Git crea ramificazioni in modo incredibilmente semplice e leggero permettendo operazioni di branching praticamente istantanee. Diversamente da altri VCS, Git incoraggia il modo di lavorare attraverso questa tecnica, ossia creare ramificazioni e successive unioni, anche molte volte in un giorno. Capire e padroneggiare questa funzionalità permette di avere a disposizione uno strumento potente ed unico che può letteralmente cambiare il modo di lavorare.

9.1 Cos'è un branch

Come presentato all'inizio della guida, Git non salva i dati in una serie di piccoli cambiamenti o delta file, ma attraverso una serie di istantanee (*snapshot*).

Quando si esegue un'operazione di commit con Git, esso immagazzina i "commit" come oggetti che contengono un puntatore allo snapshot registrato nell'area di staging, l'autore delle modifiche, il messaggio associato e zero o più puntatori al o ai commit che sono avvenuti precedentemente a quello attuale. Vengono memorizzati zero puntatori quando si esegue il primo commit su un repository appena inizializzato, un puntatore per un commit normale e due o più puntatori nel caso di unione di più branch.

Continuiamo illustrando un esempio pratico che ci aiuterà a comprendere meglio.

Assumiamo di inizializzare un nuovo repository e nella directory di lavoro collochiamo i file "Readme.txt", "Index.php" e "License.php". A questo punto, come da usuale pratica di lavoro, li tracciamo mediante il comando "add" e ne eseguimo il "commit" per memorizzare permanentemente le modifiche.

Quando viene eseguito il comando di "commit", Git calcola il checksum di ogni directory, in questo caso solo della cartella principale, e memorizza i tre oggetti nel repository. Ora Git crea un oggetto "commit" con i metadati precedentemente elencati ed un puntatore alla radice dell'albero in modo da ricreare lo snapshot quando si vuole. Il repository ora contiene i seguenti oggetti:

- un oggetto *blob* per i contenuti di ogni singolo file nell'albero;
- un oggetto *albero* che elenca i contenuti della directory e specifica i nomi dei file che devono essere salvati mediante l'oggetto *blob*;
- un oggetto *commit* con un puntatore alla radice dell'albero e tutti i suoi metadati.

Se ora si eseguono dei cambiamenti, ad uno o più dei precedenti file, e successivamente ne eseguiamo il commit, l'oggetto "commit" memorizzato avrà un puntatore al precedente oggetto "commit".

In Git un **branch** o **ramo** è semplicemente un puntatore ad uno di questi commit. Il nome del ramo principale, presente di default in ogni repository, è *master* ed è quello di default utilizzato.

9.2 Creazione di un nuovo ramo

Per la creazione di un nuovo ramo usiamo il comando

```
git branch NomeRamo
```

dove "NomeRamo" andrà sostituito con un nome per noi significativo. Git per sapere a in quale ramo ci troviamo mantiene uno speciale puntatore, chiamato *HEAD*.

E' importante osservare che l'esecuzione del comando per creare un nuovo ramo non ci sposta da quello in cui attualmente siamo al nuovo ramo creato.

9.3 Spostarsi da un ramo all'altro

Il comando da utilizzare per spostarsi da un ramo all'altro è:

```
git checkout NomeRamo
```

dove "NomeRamo" va sostituito con il nome del ramo in cui ci vogliamo spostare.

Continuiamo con l'esempio precedente per capire meglio cosa accade. Dopo aver lavorato un pò sui tre file precedenti vogliamo modificare il file "Index.php" e "License.txt" poichè vogliamo creare una versione a parte, che quindi avrà una licenza diversa di distribuzione. Dopo aver creato un nuovo ramo per ospitare questa modifica ci spostiamo su di esso con il comando appena illustrato. A questo punto il nostro progetto possiede due rami ma in sostanza è ancora quello di partenza. Ora eseguimo la nostra modifica, nei due file, ed eseguiamo l'operazione di staging e successivamente eseguimo il commit per questi ultimi.

Dopo aver eseguito il commit il ramo appena creato, si è spostato in avanti contenendo queste modifiche ma il ramo master invece, punta ancora allo stato che avevano i tre file prima della creazione del nuovo ramo. Se torniano al ramo principale osserviamo che avvengono due cose sostanzialmente:

- il puntatore *HEAD* è tornato indietro per puntare all'ultimo commit del ramo master;
- i file presenti nella cartella di lavoro sono tornati allo stato rappresentato dall'ultimo commit nel ramo master.

A questo punto il nostro progetto possiede due versioni separate ed indipendenti, ognuna con una sua storia.

9.4 Considerazioni

Concludiamo ora la sezione con alcune considerazioni sul branching in Git. Dato che in Git un ramo è semplicemente un file che contiene i 40 caratteri di checksum SHA-1 del commit al quale punta, i rami possono essere creati e distrutti facilmente. Creare un nuovo ramo è semplice e veloce quanto scrivere 41 byte (40 caratteri ed il fine riga) all'interno di un file.

Questo è in netto contrasto con molti altri VCS, che funzionano copiando tutti i file di un progetto in una seconda directory. Questa operazione può richiedere diversi secondi o minuti a seconda della dimensione del progetto, mentre in Git è istantaneo.

Inoltre, dato che vengono registrati i genitori dei commit, trovare una base di unione per il merging è generalmente molto semplice da portare a termine. La ramificazione sviluppata in quest'ottica aiuta ed incoraggia gli sviluppatori a creare e fare uso dei rami di sviluppo.

9.5 Esempio di utilizzo dell'operazione di branching-merging

Supponiamo tu stia lavarando su un sito web, e che nella cartella di Git sono presenti già diversi commit. Ora decidi che lavorerai ad una particolare richiesta, chiamata per esempio *richiesta #42*. Siccome non hai la certezza che tutto funzioni subito decidi di creare un nuovo ramo per non sporcare la soluzione web fino ad ora ottenuta. Per creare e spostarsi direttamente nel nuovo ramo puoi utilizzare come unico comando il seguente:

```
git checkout -b NomeNuovoRamo
```

Ora puoi portare avanti la richiesta #42 ed eseguire i relativi commit senza perdere il lavoro ottenuto fino ad ora. Mentre ci lavori ti arriva la notifica che è presente un problema sul sito web e devi risolverlo immediatamente. Per non perdere le modifiche che hai fatto nel portare avanti la richiesta #42 eseguirai un operazione di commit prima di spostarti nel ramo master per risolvere il problema riscontrato.

Dopo esserti spostato nuovamente sul ramo master, notando che la situazione nell'area di lavoro è tornata come prima di iniziare il lavoro per la richiesta #42, decidi di creare un nuovo ramo nel quale apporterai le modifiche al progetto per risolvere il problema riscontrato fino a quando non è del tutto risolto. Per fare ciò ripeti comando citato qui sopra cambiando il nome al branch.

Dopo averci lavorato e testato che le modifiche da te apportate risolvono il problema decidi che è il momento di mettere la tua soluzione in produzione,

ossia l'unione di questo nuovo ramo con il ramo principale. Per eseguire ciò è necessaria un operazione di **merging**, ossia di fusione di branch.

Per unire due rami ci dobbiamo spostare nel ramo che conterrà l'unione, in questo caso il master dato che la modifica deve rientrare in produzione. Eseguiamo il comando per spostarci nel ramo master ed esuiguiamo il seguente comando per ultimare la fusione:

```
git merge NomeRamo
```

dove "NomeRamo" è il nome del ramo che dobbiamo fondere con il ramo attuale, ossia quello che contiene le modifiche che hanno portato alla soluzione del problema.

Nell'esecuzione del comando di "merging" si può notare la fase di **"Fast Forward"**. Dato che il commit di unione punta direttamente a monte rispetto al commit in cui ci si trova, Git muove il puntatore in avanti. Per parafrasare in un altro modo, quando provi ad unire un commit con un commit che può essere portato al primo commit della storia, Git semplifica le cose muovendo il puntatore in avanti perchè non c'è un lavoro differente da fondere assieme. Questo sistema è definito "fast forward".

A questo punto le modifiche che hanno risolto il problema sono agganciate al ramo master, quindi il ramo creato per risolverle non è più necessario può essere quindi distrutto attraverso il comando:

```
git branch -d NomeRamo
```

dove "NomeRamo" è il nome del ramo che deve essere cancellato.

A questo punto puoi tornare a lavorare alla richiesta #42 che però non contiene le modifiche appena apportate, ma ciò non è un problema perchè se hai bisogno di quelle modifiche anche in questo ramo puoi fondere il ramo principale, master, con questo e continuare il lavoro per ultimare la richiesta #42 oppure puoi aspettare di integrare quelle modifiche quando integrerai questo ramo con il ramo principale.

10 Merging di branch

In questa sezione verrà illustrato come eseguire la fusione di branch e risolvere eventuali conflitti dovuti alla fusione.

10.1 Merging

Come illustarto dalla sezione precedente per eseguire il merging tra due branch è necessario spostarsi nel ramo che conterrà la fusione e digitare il comando:

```
git merge NomeRamo
```

dove "NomeRamo" è il nome del ramo da fondere con quello in cui siamo.

10.2 Conflitti durante la fusione

Occasionalmente può accadere che durante una fusione tutto non vada a buon fine immediatamente, ossia quando esistono dei conflitti. Un conflitto in un file sorgente lo si ha quando, unendo due rami, lo stesso file sorgente presenta delle differenze in parti che hanno subito il commit. Questo non succede se nel nuovo ramo abbiamo solo fatto aggiunte.

Se Git rileva un conflitto ci avviserà durante l'operazione di merging e tramite il comando di visualizzazione dello stato, sapremo in quale/i file è stato riscontrato questo problema.

Per risolvere i conflitti apriamo l'editor con cui scriviamo il codice sorgente e andiamo manualmente a risolvere i conflitti. La versione che è in stato di "commit" nel ramo attuale è marcata dai marcatori "<HEAD" fino ai simboli "=" tutto ciò che segue sono i conflitti rilevati da risolvere.

Dopo averli risolti, e aver tolto i marcatori, si procederà come un usuale commit, ossia si tracciano i file e si esegue il commit.

E' possibile risolvere i conflitti anche attraverso un tool grafico, configurato come illustrato ad inizio guida, lanciando il comando:

```
git mergetool
```

Al termine della risoluzione Git vorrà sapere se la fusione è andata a buon fine e, nel caso di risposta affermativa, traccerà i file sorgente. A questo punto è necessario solo eseguire l'operazione di commit.