Il database sottostante Git

Arianna Masciolini

11 luglio 2018

Indice

1 Introduzione

Un *Version Control System* (VCS) è un sistema che tiene traccia delle modifiche apportate ad uno o più file in modo da garantire all'utente la possibilità di accedere alle versioni precedenti dei file suddetti in qualsiasi momento.

I primi sistemi di controllo di versione, locali, nacquero con l'idea di risolvere i problemi legati a quello che potrebbe essere definito versioning "manuale", consistente nel conservare più copie dei file d'interesse: la forte suscettibilità a errori e lo spreco di spazio su disco. Tra questi VCS locali, RCS ha goduto ha lungo di grande popolarità: esso salva su disco, in un particolare formato, una serie di *patch*, ossia le differenze tra una versione e l'altra dei file, in modo tale da poter ricostruire lo stato in cui era ognuno di essi in qualsiasi momento, applicandovi una dopo l'altra le varie patch.

Successivamente, ci si pose il problema di permettere a più persone di collaborare a distanza. Per risolverlo nacquero i sistemi centralizzati di controllo di versione (CVCS), come CVS, Subversion, e Perforce. In questi sistemi, per il resto analoghi ad RCS e simili, tutte le versioni dei file controllati sono salvate su un unico server e rese così disponibili ai diversi utenti. Anche questo approccio presenta però problematiche importanti, dovute al fatto che il server centrale rappresenta un punto di vulnerabilità per l'intero sistema.

I DVCS (*Distributed VCS*), di cui Git, Mercurial, Bazaar e Darcs sono gli esempi più noti, risolvono questo problema: i membri del gruppo non si limitano a scaricare la più recente versione dei file, ma copiano l'intero repository, cosicchè ogni client costituisce un backup completo del progetto.

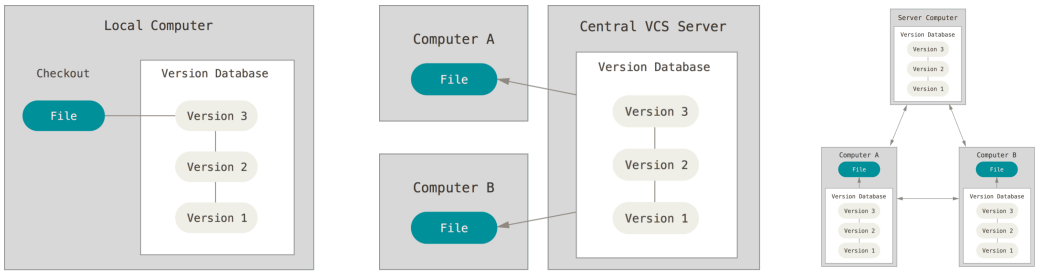


Figura 1: Confronto tra un VCS locale, un CVCS e un DVCS.

In particolare, Git -ad oggi il VCS più diffuso [3]- si differenzia tanto dagli altri sistemi distribuiti quanto dai loro predecessori per il modo in cui memorizza i dati: non come una serie di *patch* legate ai vari file, ma come una serie di "istantanee" di un filesystem in miniatura, accessibili tramite puntatori. L'obiettivo di questa relazione è, per

l'appunto, descrivere l'approccio adottato da Git in tale ambito, in modo da comprendere quali ne siano i vantaggi rispetto alle soluzioni adottate dai sistemi concorrenti. Nella sezione successiva verranno brevemente richiamati alcuni concetti di base riguardo il funzionamento di Git, utili per rendere più chiare e sintetiche le spiegazioni più strettamente legate al suo modello di branching e al sottostante database.

2 Nozioni di base su Git

Git, nato nel 2005 per favorire lo sviluppo del kernel Linux, è un sistema di controllo versione completamente distribuito che deve il suo successo al proprio modello di branching, che lo rende ottimale per la gestione di progetti open source, che hanno quasi sempre uno sviluppo non lineare.

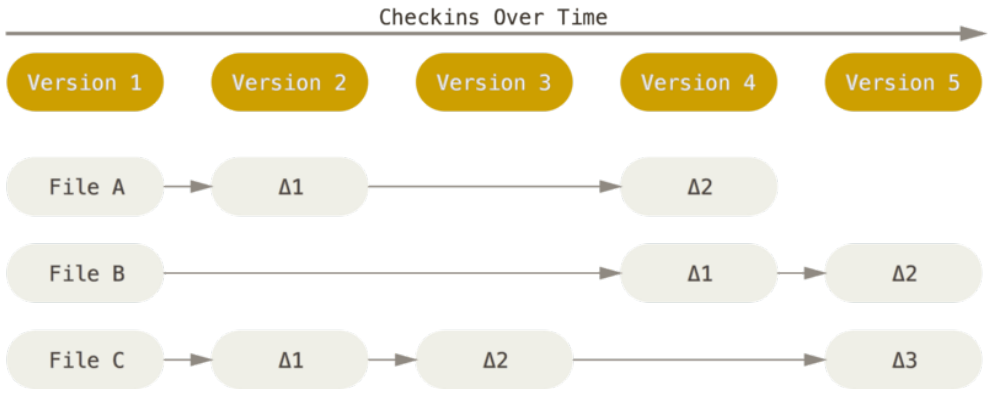
Come si è accennato nella sezione precedente, Git adotta un approccio radicalmente differente rispetto a quello tradizionale, il cosiddetto controllo versione *delta-based*, basato su una lista di *patch*, ossia di modifiche, legate ai singoli file. Un repository Git è infatti una sorta di filesystem Unix-like semplificato, di cui viene fatta una sorta di "istantanea" resa accessibile mediante un puntatore ogniqualvolta lo stato del progetto viene salvato.

Quasi tutte le azioni possibili in Git consistono, in ultima analisi, nell' *aggiungere* dati al sottostante database. Cancellare dei dati e, in generale, compiere azioni distruttive, è reso volutamente difficile, in modo tale da rendere pressoché impossibile perdere il proprio lavoro, se non altro a seguito di un *commit*, l'azione tramite la quale i cambiamenti al repository vengono salvati.

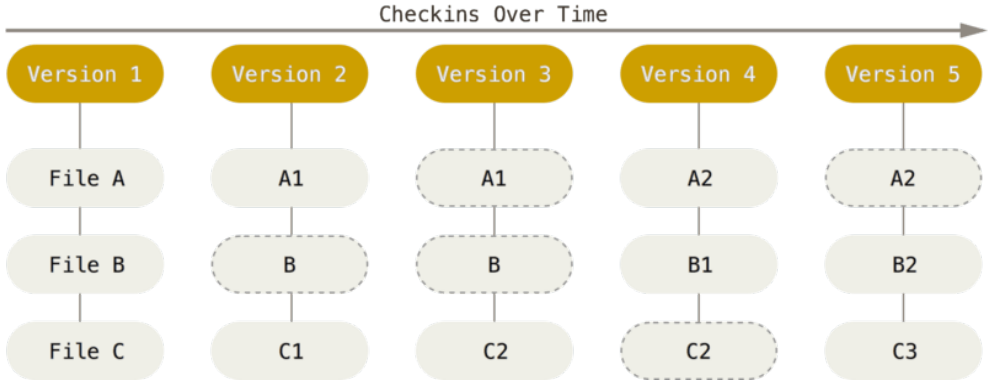
Per meglio comprendere quale sia il significato di un commit in Git e, in generale, per ottenere un buon modello concettuale del funzionamento di tale sistema di controllo versione, è fondamentale conoscere gli stati in cui può trovarsi un file all'interno di un repository:

- *modified*: sono state effettuate delle modifiche al file, non ancora confermate tramite un commit;
- *staged*: il file, che ha subito delle modifiche, è stato marcato come parte del prossimo commit;
- *committed*: il file è salvato permanentemente nel database locale tramite, appunto, un commit.

A questo punto è opportuno parlare della struttura di un repository Git, le cui parti principali sono:



(a)



(b)

Figura 2: Confronto il modo in cui vengono salvati i dati in un VCS delta-based (a) e quanto avviene in Git (b).

- il *working tree*, consistente in una copia di una specifica versione del progetto, estratta dal database in modo da poter essere modificata dall'utente;
- la cartella `.git`, dove vengono si trovano i metadati e l'*object database*, sui cui ci soffermerà nella prossima sezione, relativi al progetto;
- la *staging area*, ossia il contenuto del prossimo commit, le informazioni relative al quale sono memorizzate nel file `index`, generalmente contenuto nella cartella `.git`.

E' facile intuire, a questo punto, quale sia il tipico modo di procedere quando si lavora ad un progetto gestito con Git: dopo aver modificato uno o più file nel *working tree*, si selezionano le modifiche di cui tener traccia nel commit successivo, aggiungendoli così

alla *staging area*. Nel momento in cui viene eseguito il comando `git commit` lo stato dei file nella *staging area* viene salvato nella cartella `.git`.

Checksumming Un'ulteriore, importante funzionalità di Git, parte integrante della sua filosofia, è quella di verificare qualsiasi file o directory, prima di archiviarlo, tramite una checksum che è poi utilizzata per riferirvisi. Ciò significa che è impossibile cambiarne il contenuto senza che Git ne sia a conoscenza, il che permette di rilevare qualsiasi perdita o corruzione di dati in transito. Il meccanismo impiegato per generare tale checksum è la funzione crittografica SHA-1, il cui output, basato sul contenuto di un file o sulla struttura interna di una cartella, è un numero esadecimale di 40 caratteri.

3 Il database di Git

La componente fondamentale di Git è un **database chiave-valore**: all'inserimento di un oggetto in un repository, Git restituirà una chiave, tramite la quale esso sarà univocamente identificato. Si può osservare in prima persona questo comportamento di Git utilizzando uno dei suoi comandi più a basso livello, `git hash-object`, che salva i dati che gli sono passati come input nell'*object database*, ossia nella cartella `.git/objects`, e ne restituisce la checksum. Inizializziamo dunque un nuovo repository, `demo`:

```
$ git init demo
Initialized empty Git repository in /home/harisont/demo/.git/
```

All'interno della cartella `.git` repository appena creato si trova, per l'appunto, la cartella dell'object database, contenente due sottocartelle, ma ancora nessun file[1]:

```
$ cd demo
$ find .git/objects
.git/objects
.git/objects/info
.git/objects/pack
$ find .git/objects -type f
```

Creiamo dunque un nuovo file vuoto ed inseriamolo nell'object database (senza l'opzione `-w`, ne otterremmo la checksum senza scrivere nulla nel database):

[1] Il comando `find .git/objects -type f` serve a cercare tutti i file presenti nel repository.

```
$ touch first
$ git hash-object first -w
e69de29bb2d1d6434b8b29ae775ad8c2e48c5391
```

Chiaramente, a questo punto, la ricerca di file in `.git/objects` avrà un risultato di questo tipo:

```
$ find .git/objects -type f
.git/objects/e6/9de29bb2d1d6434b8b29ae775ad8c2e48c5391
```

Sono state create una sottocartella, il cui nome è costituito dai primi 2 caratteri della checksum, ed un file, ridenominato con i restanti 38. A questo punto, abbiamo tutti gli strumenti per utilizzare, sia pure in modo molto semplice, il nostro sistema di controllo versione: facendo una modifica al file `first` e salvandola tramite `git hash-object`, si fa in modo che l'object database contenga entrambe le versioni del file, che resteranno disponibili anche qualora il file venisse rimosso dal *working tree*:

```
$ echo 'first modification!'  > first
$ git hash-object first -w
$ find .git/objects -type f
.git/objects/e6/9de29bb2d1d6434b8b29ae775ad8c2e48c5391
.git/objects/a9/16b860146f3db545e0e094f650d6e7b3dd4230
```

Tuttavia, utilizzare Git in questo modo è decisamente poco pratico, poiché per recuperare le versioni precedenti dei file è necessario utilizzare la checksum che lo identifica. Ciò è dovuto al fatto che, utilizzando il comando `git hash-object` non viene memorizzato il nome del file, ma solo il suo contenuto. Utilizzando la terminologia di Git, indicheremo gli oggetti di questo tipo come *blob*.

3.1 Tipi di oggetti in Git

Chiaramente, i **blob** non sono i soli oggetti presenti in Git. Per permettere di salvare i nomi dei file e di raggrupparli, Git utilizza anche oggetti di tipo ***tree***. Per meglio comprendere il ruolo che giocano questi due tipi di oggetti, basti pensare all'analogia tra Git ed un filesystem Unix: da una parte, i blob corrispondono, grosso modo, agli *inodes*[2], dall'altra i tree corrispondono alle directory. Un oggetto di tipo tree contiene un puntatore SHA-1 ad un blob o ad un sottoalbero.

[2]in Unix, sono così chiamate le strutture dati che descrivono un oggetto del filesystem, ossia un file o una directory, memorizzandone i metadati e la posizione sul disco [4].

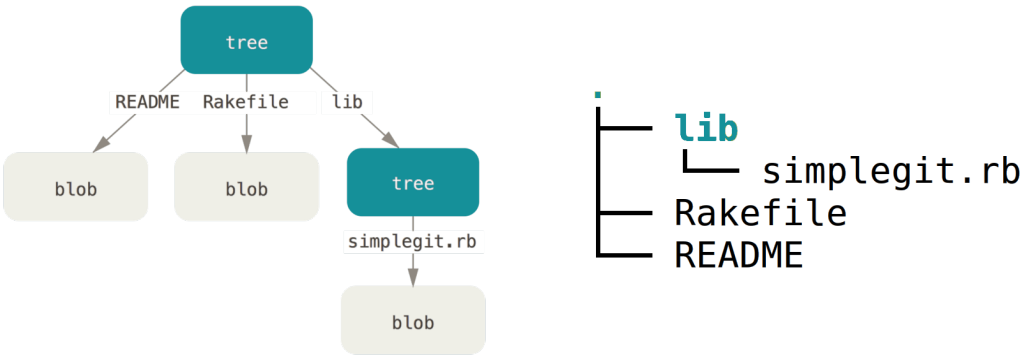


Figura 3: Corrispondenza tra oggetti di tipo tree e cartelle in un repository Git: sulla sinistra, il modello concettuale, sulla destra l'output del comando `tree`.

Poiché gli oggetti di tipo tree sono creati a partire dallo stato della staging area, cioè a partire dal file `index`, la creazione manuale di un oggetto di tale tipo richiede la creazione di un indice *ad hoc*, per la quale si può utilizzare un altro comando a basso livello, `git update-index`, volto appunto ad aggiungere un file alla staging area, o il più noto `git add`, che aggiorna il file `index` con il contenuto del working tree [2]. Supponiamo di aver aggiunto alla staging area il file `README`: a questo punto, è possibile creare un oggetto tree, che sarà memorizzato nell'object database, tramite il comando `git write-tree`, che ne restituisce la checksum:

```
$ git write-tree
543b9bebdc6bd5c4b22136034a95dd097a57d3dd
```

Per verificare il tipo di un oggetto, si può fare uso del comando `git cat-file`, con l'opzione `-t`, mentre l'opzione `-p` serve a visualizzare il suo contenuto:

```
$ git cat-file -t 543b9bebdc6bd5c4b22136034a95dd097a57d3dd
tree
$ git cat-file -p 543b9bebdc6bd5c4b22136034a95dd097a57d3dd
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391 README
```

Gli oggetti di tipo tree costituiscono, in pratica, la forma più semplice delle "istantanee" di cui si è parlato nell'introduzione. Tuttavia, non risolvono tutti i problemi pratici del controllo versione: innanzitutto, con il solo uso di tree e blob, per recuperare le informazioni relative alle versioni precedenti di un progetto è necessario utilizzare le checksum, il che è poco agevole. Inoltre sarebbe utile salvare altre informazioni relative alle suddette "istantanee", come il loro autore, la data ed una descrizione. E' a questo

proposito che sono stati introdotti gli oggetti di tipo ***commit***. La creazione di un oggetto di questo tipo avviene tramite il comando `git commit-tree`, che è in pratica il corrispettivo a basso livello del più usato `git commit`: data la checksum di un tree, prendendo il messaggio di commit dallo standard input, restituisce l'id dell'oggetto creato:

```
$ echo 'first commit' | git commit-tree 543b9bebdc6bd5c4b22136034a95d
  d097a57d3dd
d03724c8f46342c5057107adc1c586feba5793ff
```

La struttura di un commit è la seguente:

```
$ git cat-file -p d03724c8f46342c5057107adc1c586feba5793ff
tree 543b9bebdc6bd5c4b22136034a95dd097a57d3dd
author Arianna Masciolini <uzkamascio@gmail.com> 1531326893 +0200
committer Arianna Masciolini <uzkamascio@gmail.com> 1531326893 +0200

first commit
```

3.2 Memorizzazione degli oggetti

...

3.3 Riferimenti

...

Riferimenti bibliografici

[1] Scott Chacon, Ben Straub. *Git Pro*, seconda edizione, 2014.

[2] Git reference manual.
Ultima visita: 11/07/2018.

[3] `openhub.net`.
Ultima visita: 5/7/2018.

[4] Wikipedia. *inode*.
Ultima visita: 10/7/2018.

[5] Junio C. Hamano. *Fun with updating the cached contents in the index by staging*.
Ultima visita: 11/07/2018.