Pietro Coronas 433043

ARCHITETTURA DI GIT

Introduzione

Git è un sistema software di controllo di versione distribuito(DVCS), creato da Linus Torvalds nel 2005. Il sistema di controllo di versione registra, nel tempo, i cambiamenti ad un file o ad una serie di file, così da poter richiamare una specifica versione in un secondo momento**.** Git consente il mantenimento di un contenuto digitale (spesso, ma non solo, codice) da parte di molti collaboratori che utilizzano una *rete peer-to-peer* di repository. Git è supportato dalla maggior parte dei sistemi operativi sia tramite riga di comando che tramite strumenti di interfaccia utente, costruiti su un core Toolkit diviso in 2 parti fondamentali:

- Una parte composta da comandi di basso livello che consentono il monitoraggio di contenuti di base e la manipolazione dei grafi aciclici diretti (approfonditi in seguito)
- Una seconda parte composta da un sottoinsieme di comandi destinati agli utenti per la gestione dei repository e per la comunicazione.

1. Funzionalità e requisiti di qualità

Supportando un flusso di lavoro distribuito, Git offre notevole flessibilità e notevoli vantaggi . Le funzionalità principali richieste da Git sono:

- Possibilità di lavorare offline e condividere codice in modo incrementale
- Distribuzione del contenuto tra i vari collaboratori
- Accesso allo "storico" dei repository in modalità offline
- Possibilità di lavorare su più repository

I requisiti qualitativi che Git deve offrire si possono riassumere nei seguenti punti:

- Preservare l'integrità dei dati
- Scalabilità
- Prestazioni
- Sicurezza

Per capire come GIT rispetti i requisiti di qualità, ci soffermiamo sull'analisi della memorizzazione dei contenuti, sulla creazione dello storico e sulla distribuzione dei contenuti.

2. Memorizzazione dei contenuti

a. Primitive fondamentali

Git ha 4 primitive di base in cui vengono costruiti intorno i repository locali. Ogni tipo di oggetto ha le seguenti caratteristiche: tipo, dimensione e contenuto. Le 4 primitive sono:

- *Blob*: Un blob è il contenuto di un file. Git memorizza ogni revisione come un oggetto blob distinto.
- *Tree*: Un oggetto Tree è l'equivalente di una directory: contiene una lista di nomi di file, con tipo e nome di un oggetto blob o un altro albero. Descrive un'istantanea dell'albero dei sorgenti.
- *Commit*: Un commit collega gli oggetti albero in una cronologia. Contiene il nome di un oggetto albero, data e ora e un messaggio di archiviazione (log). Le relazioni tra i vari blob si possono trovare esaminando gli oggetti albero e commit.
- *Tag:* Contenitore che contiene riferimenti ad un altro oggetto. Solitamente è usato come firma digitale di un oggetto commit.

Ogni oggetto è identificato da un codice *Hash* del suo contenuto (SHA). Git calcola il codice Hash e lo utilizza come nome dell'oggetto. Si tratta di una stringa di 40-caratteri, composta da caratteri esadecimali (0–9 ed a–f) e calcolata in base al contenuto di file o della struttura della directory in Git. In riferimento al codice hash, è bene sottolineare che:

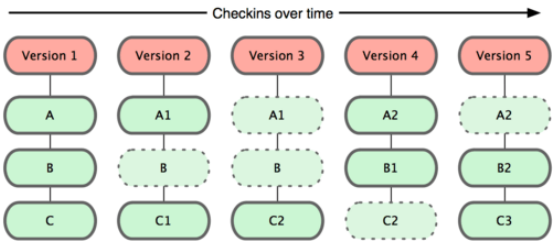
- Se due oggetti sono identici, avranno lo stesso SHA
- Se due oggetti sono diversi, avranno diversi SHA
- Se un oggetto è copiato solo parzialmente o si è verificata una corruzione, l'SHA identificherà tale corruzione.

I primi due punti permettono a Git di adottare un modello distribuito, mentre l'ultimo punto riguarda la salvaguardia contro la corruzione dei file, e quindi preservare l'integrità dei dati. L'SHA è utilizzato da Git anche come checksum. Qualsiasi cosa in Git è controllata tramite *checksum*. Non può capitare che delle informazioni in transito si perdano o che un file si corrompa senza che Git non sia in grado di accorgersene.

b. Salvataggio dei dati

Come permettere a Git di utilizzare in modo efficiente la memoria, di essere veloce e al contempo di preservare l'integrità dei dati? Concettualmente la maggior parte degli altri sistemi salvano l'informazione come una lista di modifiche ai file. Questi sistemi considerano le informazioni che mantengono come un insieme di file, con le relative modifiche fatte ai file nel tempo.

Git considera i propri dati più come una serie di istantanee (*snapshot*) di un mini file system. Ogni volta che viene effettuato un commit o salvataggio dello stato del progetto, Git fa un'immagine di tutti i file in quel momento, salvando un riferimento allo *snapshot*. Se alcuni file non sono cambiati, Git non li risalva, ma crea semplicemente un collegamento al file precedente già salvato.



Gli snapshot vengono successivamente utilizzati per formare un grafo aciclico diretto (DAG), basato sul concetto di diramazione : Diramazione significa divergere dal flusso principale di sviluppo continuando a lavorare senza correre il rischio senza intaccare il flusso principale. Git crea ramificazioni in modo incredibilmente semplice e leggero, permettendo operazioni di diramazione praticamente istantanee come lo sono anche i passaggi da un ramo ad un altro. Diversamente da molti altri VCS, Git incoraggia un metodo di lavoro che sfrutta le ramificazioni e le unioni frequentemente.

I file possono trovarsi in tre stati:, *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, cioè è preparato per essere aggiunto nella nuova revisione. 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. Il funzionamento di Git può essere riassunto in 3 punti:

- si modifica uno o più file presenti nella *working directory*;
- si aggiunge un suo snapshot, cioè una loro copia, nella *staging area;*
- 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.

Vediamo meglio in dettaglio le operazioni che effettua GIT:

- Viene effettuato il Commit di una Directory con alcuni file. Git salva la versione del file nel repository Git (Git fa riferimento ad essi come blob), e aggiunge questi checksum *all'area di staging*.
- Git calcola il checksum di ogni directory (in questo caso, solamente la directory radice del progetto) e salva questi oggetti nel repository Git. Git poi crea un commit dell'oggetto che ha i metadati ed un puntatore alla radice dell'albero del progetto in maniera da ricreare l'istantanea quando si vuole.
- Il repository Git ora contiene diversi tipi di oggetti: un blob per i contenuti di ogni singolo file nell'albero, un albero che elenca i contenuti della directory e specifica i nomi dei file che devono essere salvati come blob, e un commit con il puntatore alla radice dell'albero e a tutti i metadati del commit.

- Se si fanno dei cambiamenti e viene eseguito il commit nuovamente, il commit successivo immagazzinerà un puntatore al commit che lo precede. In Git un ramo è semplicemente un puntatore ad uno di questi commit. Il nome del ramo principale in Git è master. Quando si inizia a fare dei commit, si stanno inviando al ramo master che punterà all'ultimo commit che è stato eseguito. Ogni volta che si effettua un commit, lui si sposterà in avanti automaticamente. Dato che un ramo in Git è semplicemente un file che contiene 40 caratteri di un checksum SHA-1 del commit al quale punta, i rami si possono creare e distruggere facilmente.

Questo è in netto contrasto con il sistema utilizzato da molti altri VCS, che comporta la copia di tutti i file di un progetto in una seconda directory. Questo può richiedere diversi secondi o minuti, a seconda delle dimensioni del progetto, mentre in Git è un processo sempre istantaneo. Inoltre, dato che si registrano i genitori dei commit, trovare la base adatta per la fusione è fatto automaticamente.

Come si può notare, la maggior parte delle operazioni in Git necessitano solo di file e risorse locali per operare, e questo ne aumenta la velocità

c. Tecniche di fusione

I sistemi DVCS permettono a più sviluppatori di modificare lo stesso file nello stesso tempo e, tramite le tecniche di fusione, permettono di combinare le diverse modifiche. Git implementa varie strategie di fusione e si può scegliere un comportamento diverso da quello di default: Il primo è l'algoritmo tradizionale di fusione a 3 vie, ovvero basato su due contenuti A e B e un antenato comune. Un'alternativa è l'algoritmo ricorsivo (di default) ed è una variante dell'algoritmo di fusione a 3 vie. Quando

ci sono più antenati comuni che possono esser usati per una fusione a 3 vie, viene creato un albero di fusione degli antenati comuni e si utilizza come albero di riferimento per la fusione a 3 vie. Questo risulta produrre meno conflitti di fusione.

d. Tecniche di compressione

Analizzare i meccanismi di compressione diventa importante in ambito sia di prestazioni sia nell'individuare eventuale corruzione dei dati. Git affronta il problema imballando gli oggetti in un formato compresso, utilizzando un file indice che punta a offset per individuare oggetti specifici nel corrispondente file compresso. Per evitare la perdita di informazioni durante la compressione, viene incluso un checksum in ogni file compresso nell'indice e alla fine di ogni file compresso vengono inseriti 20 byte ottenuti come elenco ordinato degli SHA presenti nel file.

3. Creazione dello storico

Riguardo lo storico dei commit e dei merge, la maggior parte dei VCS utilizza 2 tecniche

- Storico Lineare
- Grafo Aciclico orientato

Git utilizza anche per lo storico un grafo aciclico diretto (DAG). Ogni commit contiene metadati dei suoi antenati. Un singolo commit può avere zero o molti (in teoria illimitati) genitori: ad esempio, il primo commit avrà zero genitori, mentre il risultato di un merge tra vari commit ne avrà molti. Risalendo i vari nodi del grafo, è possibile individuare lo storico di un determinato contenuto.

Anche durante la fase di merge è possibile risalire allo storico, in quando durante l'operazione di fusione di due rami viene creato un DAG. L'utilizzo dei DAG per rappresentare lo storico rende GIT molto efficiente nel determinare antenati (e quindi contenuti) comuni tra i vari nodi.

È effettivamente leggermente più costoso esaminare la cronologia di modifiche di un singolo file (storico lineare) rispetto a quella dell'intero progetto. Per ottenere la storico delle modifiche che riguardano un dato file, Git deve ripercorrere la cronologia complessiva visitando il DAG e poi determinare quali modifiche hanno riguardato quel file. Questo metodo di esaminare la cronologia, comunque, permette a Git di produrre con altrettanta efficienza una singola cronologia che mostra le modifiche a un arbitrario insieme di file

Come ulteriore meccanismo di sicurezza, lo storico di Git viene memorizzato in modo tale che il nome di un particolare commit dipende dalla completa cronologia di sviluppo che conduce a tale commit. Una volta che è stata pubblicato, non è più possibile cambiare le vecchie versioni senza che ciò venga notato

4. Distribuzione dei contenuti

Il fatto di essere un sistema distribuito è una delle sue principali caratteristiche che lo differenziano dagli altri VCS. Solitamente i VCS vengono divisi in 3 tipologie:

- *Locale*: Questo strumento funziona salvando sul disco una serie di patch (ovvero le differenze tra i file) tra una versione e l'altra, in un formato specifico; può quindi ricreare lo stato di qualsiasi file in qualsiasi momento, aggiungendo le varie patch. Ovviamente come è facile intuire, non supporta la distribuzione dei contenuti.
- *Centralizzati*: Basati su un unico server che contiene tutte le versioni dei file e un numero di utenti che scaricano i file dal server centrale. Offre la distribuzione dei contenuti tra i vari collaboratori ma presenta un singolo punto di rottura del sistema, che non offre nessun tipo di backup

- *Distribuiti (DVCS):* i client controllano lo *snapshot* più recente dei file e fanno una copia completa del repository. In questo modo se un server morisse e i sistemi interagiscono tramite il DVCS, il repository di un qualsiasi client può essere copiato sul server per ripristinarlo. Ogni checkout è un backup completo di tutti i dati. Con questo approccio, è possibile avere più repository remoti su cui poter lavorare, in modo da collaborare con gruppi differenti di persone in modi differenti, simultaneamente sullo stesso progetto. Questo permette di impostare diversi tipi di flussi di lavoro che non sono possibili in sistemi centralizzati, come i modelli gerarchici.

In Git quindi non c'è la necessità di far capo ad un repository principale. Ogni copia del repository è a tutti gli effetti un repository completo. Un sistema DVCS in generale è quindi un software che tiene traccia e permette di controllare i cambiamenti al codice sorgente prodotti da ciascun sviluppatore, condividendone allo stesso tempo la versione più aggiornata o modificata da ciascuno mostrando dunque in tempi brevi lo stato di avanzamento del lavoro di sviluppo.

Con Git, ogni sviluppatore è potenzialmente sia un nodo che un nucleo: infatti ogni sviluppatore può contemporaneamente contribuire al codice di altri repository e mantenere un repository pubblico sul quale gli altri basino il proprio lavoro e verso il quale possano contribuire.

5. Riferimenti

- **http://git-scm.com/book/it/v1/Per-Iniziare**

Utilizzato per l'approfondimento e la chiarificazione di alcuni concetti come la diramazione e i meccanismi di memorizzazione/fusione. 5/10

- **http://it.wikipedia.org/wiki/Git_%28software%29**

Utilizzato per avere una panoramica generale su Git. 5/10

- **http://aosabook.org/en/git.html**

Utilizzato per la maggior parte di contenuti presenti. 9/10

6. Note

A causa di un imprevisto, il collega con il quale ho svolto il primo homework non ha partecipato al progetto. Dato lo scarso preavviso con cui mi ha informato, non ho avuto tempo per trovare un altro collega con il quale svolgerlo.