Corso Git 2014

Riccardo Binetti
me@rbino.com

22 Ottobre 2014

Perché usare un VCS

- "Questo codice funziona bene, però chissà se funzionerebbe se togliessi questo e mettessi quest'altro"
- Si inizia a commentare via codice. "Tanto c'è l'undo"
- Ops, devi decommentare la terzultima modifica ma lasciando intatte le ultime due
- Ops, si è chiuso l'editor di testo hai perso la history
- "Questo era commentato prima o l'ho commentato ora? Come diavolo era il codice di partenza?"

Perché usare un VCS/2

- Alice e Bob devono scrivere qualcosa (del codice, un documento) insieme
- "Idea! Mettiamolo su una cartella condivisa così ci possiamo lavorare insieme"
- Alice apre il documento. Anche Bob apre il documento.
- Alice aggiunge una sezione, salva e chiude. Bob aggiunge una sezione, salva e chiude
- Riuscite ad intravedere il problema?

Introducing Git

Introducing Git

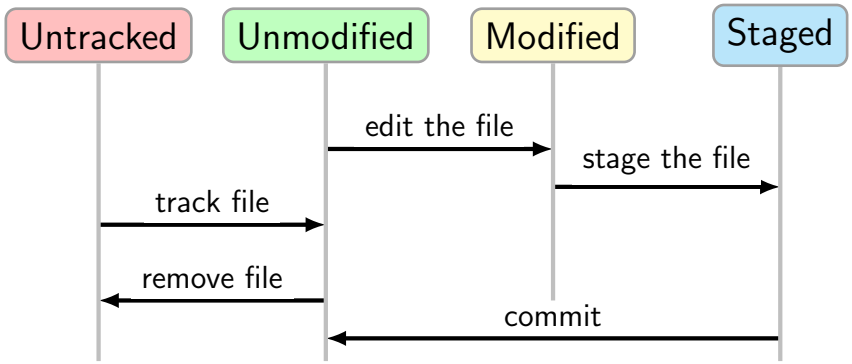
- Git è un VCS (Version Control System) creato nel 2005 da Linus Torvalds
- È uno dei VCS più utilizzati
- Permette uno sviluppo distribuito a differenza di VCS centralizzati come SVN
- È estremamente leggero, quindi invoglia a versionare tutto
- I commit sono un'operazione quasi senza costo e locale, questo aiuta a mantenere una grana fine nelle modifiche
- È disponibile per GNU/Linux, Windows, MacOS X[1]

[1] http://git-scm.com/downloads

Distribuito vs Centralizzato

- VCS centralizzato:
  - C'è una solo copia "centrale" del progetto
  - I cambiamenti vengono committati su questa copia
  - Bisogna essere connessi a internet per poter committare
- VCS distribuito:
  - Ogni copia può essere quella principale
  - I cambiamenti vengono committati in locale
  - Si può lavorare anche senza internet

Funzionamento di Git



- Git tiene traccia solo dei file che gli indichiamo esplicitamente
- Inizialmente un file sarà o untracked o unmodified
- Applichiamo le modifiche che vogliamo applicare
- Spostiamo il file nella staging area
- Quando si fa un commit vengono salvate tutte le modifiche presenti nella staging area
- Si rinizia da capo
- N.B.: tutto ciò viene svolto in locale

Comandi base

- Git è fatto per essere utilizzato principalmente da linea di comando
- Nel dubbio digitate `git help`
- Oppure `man git`
- Solitamente c'è il man anche dei sottocomandi (e.g. `man git clone`)
- Esistono anche delle GUI (`gitg`, `gitk`, `git gui`), possono avere un'utilità in alcuni casi che mostrerò nella demo

`git init`

- `git init`
- Serve a inizializzare un repository Git in locale
- Va digitato nella cartella che volete inizializzare come repository

`git clone`

- Serve quando volete “clonare” un repository Git già esistente
- Il repository può essere sia remoto che locale
- `git clone <url del repo>` lo clona in una cartella chiamata come il repository
  - `git clone <url del repo> <nomecartella>` se volete clonarlo in un’altra cartella

git status

```
→  UnrealEngine git:(4.5) ✗ git status
Sul branch 4.5
Your branch is up-to-date with 'origin/4.5'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   OAUTH

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   Engine/Build/BatchFiles/Linux/GetAssets.py
        deleted:    Engine/Source/ThirdParty/LinuxNativeDialogs/UELinuxNativeDialogs/build/.empty

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        Engine/Source/Programs/AutomationTool/bin/
        Engine/Source/ThirdParty/LinuxNativeDialogs/UELinuxNativeDialogs/build/CMakeCache.txt
        Engine/Source/ThirdParty/LinuxNativeDialogs/UELinuxNativeDialogs/build/CMakeFiles/
```

- Uno dei comandi più utili
- Mostra una serie di informazioni:
  - Su che branch siete
  - Se siete “avanti” o “indietro” rispetto al branch remoto che state tracciando
  - I file modificati non ancora nella staging area
  - I file nuovi non ancora tracciati
  - I file tracciati che sono stati rimossi

git add

- Serve per spostare le modifiche nella staging area o per cominciare a tracciare file non tracciati
- Va fatto con i file che volete siano compresi nel prossimo commit
- `git add <file>`
  - Se `<file>` è una cartella, aggiunge ricorsivamente tutte le modifiche contenute in essa

`git add`

- `git add -a` – aggiunge tutte le modifiche sui file già tracciati
- `git add -A` – come sopra ma aggiunge anche i file non ancora tracciati
- Attenzione: `git add` non aggiunge *il file* alla staging area, aggiunge *le modifiche fatte al file nel momento in cui viene eseguito il comando*
  - Se successivamente modifichiamo ancora quel file e non rifacciamo `git add`, il commit successivo conterrà solo le prime modifiche

.gitignore

- Per dichiarare a Git i file che vogliamo esplicitamente ignorare, dobbiamo inserirli nel file .gitignore nella root del repository
- Il file accetta sia nomi completi che globbing e supporta la negazione
  - *.o (escludi tutti i file che finiscono con .o)
  - !libreriaBuffa.o (...ma non libreriaBuffa.o)

.gitignore

- Best practice: non committare file binari e file che si possano ricreare nel processo di build
  - Quindi se lavoriamo ad un programma C committeremo solo il sorgente
- Se non sapete da dove partire, ne esistono di già pronti per molti tipi di progetto[2]
- Attenzione: .gitignore è un file del vostro repository, quindi va addato e committato!

[2] https://www.gitignore.io/, https://github.com/github/gitignore

`git commit`

- Crea un commit contenente le modifiche che sono nella staging area
- Di default apre un editor di testo per permettere di digitare il messaggio di commit
  - `git commit -m "Messaggio di commit"` lo fa senza aprire l'editor di testo

git commit

- Best practice: i messaggi di commit devono avere senso
  - "Fatte varie cose" ←Nope.
  - Di solito la loro significatività è direttamente proporzionale al tempo che manca alla deadline
- I commit in Git sono velocissimi da fare, quindi è molto meglio farne tanti e piccoli
  - Evitare cose come "Riscritta tutta la funzione A e aggiunta funzione B e migliorata funzione C"
  - Seguendo questa pratica si può trovare con più facilmente il punto in cui si è rotto qualcosa
  - ...ed annullare solo il pezzo che rompe tutto

Identificare i commit

- Ogni commit è identificato univocamente da un id
- l'ID è formato facendo un hash (SHA1) di un po' di roba[3]
- Di solito (date le proprietà degli hash) se vogliamo riferirci ad uno specifico commit bastano le prime 6 o 7 cifre dell'hash
- Se vogliamo possiamo dare dei nomi a degli specifici commit

[3] https://gist.github.com/masak/2415865

git tag

- `git tag <nometag>` – assegna il tag all'ultimo commit
  - `git tag <nometag> <commit>` – assegna il tag al commit specificato
- È possibile usare questa feature per assegnare dei numeri di versione ai commit
- `git tag -a <nometag>` – crea un tag annotato, che include anche chi ha creato il tag, la data in cui è stato creato e un commento
- Il tag può essere usato ovunque può essere usato l'ID del commit

Altri comandi

- `git rm <nomefile>` – rimuove un file tracciato
  - `git rm --cached` <nomefile> – rimuove il file dal tree ma non lo elimina
- `git mv` – sposta un file tracciato
  - Spostandolo solo con mv git lo vede come una rimozione e un'aggiunta nella nuova posizione
- `git log` – mostra la history dei commit
- `git show <commit>` – mostra uno specifico commit

Altri comandi

- `git diff` – mostra il diff tra le modifiche già nella staging area e quelle non ancora in staging area
  - `git diff --staged` – mostra il diff tra l'ultimo commit e la staging area
- `git reset` – toglie tutto dalla staging area (le modifiche rimangono)
  - `git reset --hard` – riporta tutti i file allo stato in cui erano all'ultimo commit (Attenzione: si perdono tutte le modifiche)
- `git blame <nomefile>` – mostra il file con il commit e l'autore che hanno generato ogni singola riga
  - Attenzione: può fare finire amicizie

Demo

Cos'è un branch in Git

- In Git ogni commit contiene un puntatore al commit precedente
- In questo modo viene costruita la history (quella che vedete con `git log`)
- Un branch in Git è semplicemente un puntatore ad uno specifico commit
- Il puntatore HEAD indica a Git qual è il branch su cui si sta lavorando attualmente
- Quando si fa un commit su due branch diversi che derivano dallo stesso commit, la history di Git diverge
- Per farla riconvergere si può fare un merge (che è un commit speciale in quanto ha due puntatori ai predecessori) o un rebase

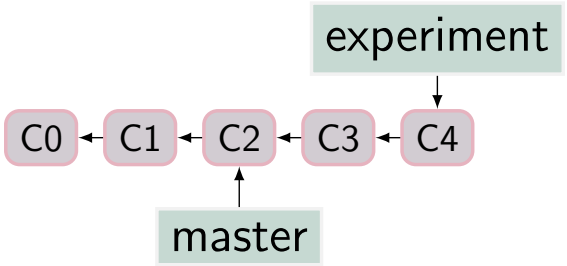
Branching

- In Git il branching è molto agile, quindi conviene usarlo
- Ad esempio: tengo un branch per il ramo stabile e un branch per il ramo di sviluppo
  - Poi ne posso creare un altro per introdurre nuove funzionalità
  - Oppure un altro per la localizzazione
  - Possibilities are endless!
- Vediamo un po' come funziona il tutto

Comandi di base dei branch

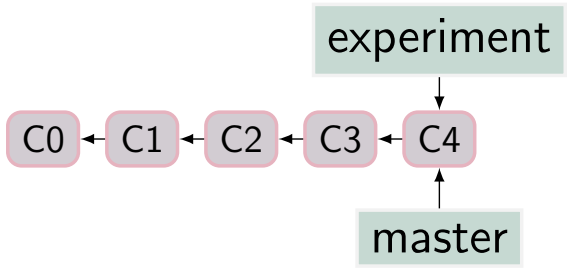
- Se avete clonato un repo, vi troverete tutti i branch che ci sono sul repo remoto anche in locale
- `git branch` – elenca i branch locali
- `git checkout <branch>` – fa passare la cartella di lavoro a quel branch
  - Per fare checkout la working directory dev'essere pulita
- `git branch <branch>` – crea un nuovo branch che punta al commit attuale (però non fa il checkout)
- `git checkout -b <branch>` – crea un nuovo branch e fa il checkout di quel branch
- `git merge <branch>` – effettua il merge del branch corrente con il branch passato come argomento
- `git rebase <branch>` – effettua il rebase del branch corrente sul branch passato come argomento

Merge: Fast Forward



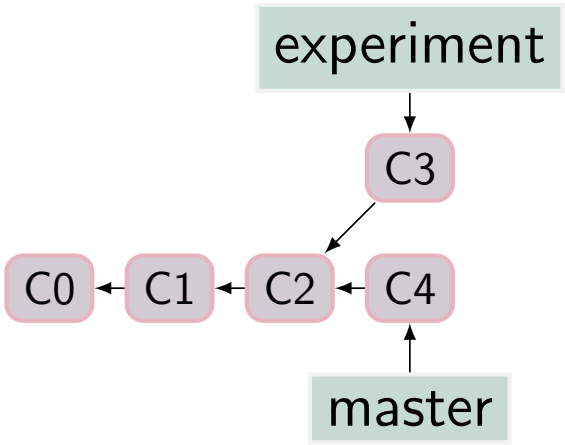
- Uno dei due possibili casi di merge è il fast forward
- In questo caso non è stato fatto nessun nuovo commit sul branch master da quando è stato creato il branch experiment
- Più in generale, è possibile fare un merge fast-forward quando partendo dal branch che vogliamo unire è possibile arrivare al branch in cui ci troviamo seguendo i puntatori ai commit precedenti
- `git checkout master`
- `git merge experiment`

Merge: Fast Forward



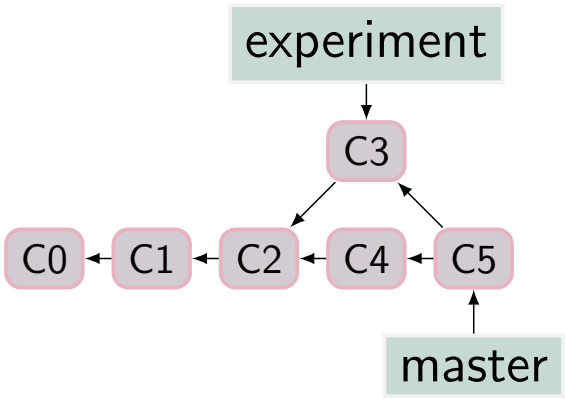
- Il risultato è questo
- Non viene creato un nuovo commit per il merge
- Nei merge fast forward non ci possono essere conflitti

Merge: 3-way merge



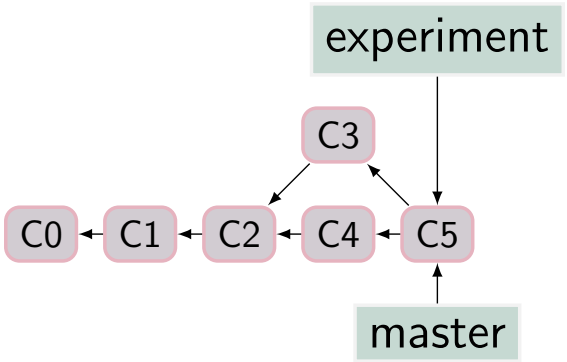
- In questo caso non c'è un percorso orientato da experiment a master
- Git risale il grafo fino a trovare un parente comune (C2)
- Quindi, dopo aver svolto il diff tra il parente e i 2 nodi coinvolti nel merge, svolge il cosiddetto 3-way merge
  - Se tutti e due hanno introdotto la stessa modifica nella stessa sezione, la modifica viene introdotta
  - Se uno dei due ha introdotto una modifica in una sezione e l'altro non ha tocccato quella sezione, la modifica viene introdotta
  - Se tutti e due hanno modificato in modo diverso in una sezione, viene creato un conflitto

Merge: 3-way merge



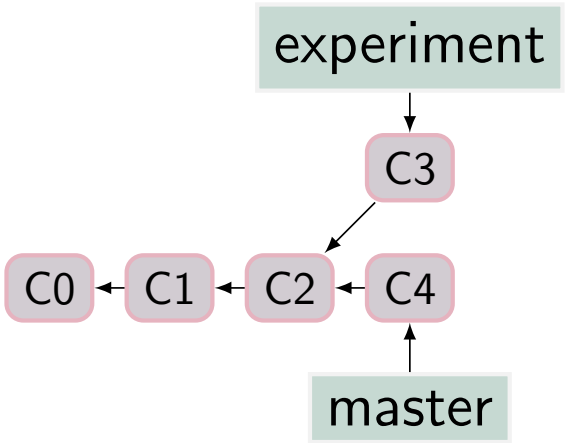
- Il risultato finale è quello mostrato in figura
- In questo caso viene creato un nuovo commit
- Il commit merge è speciale (ha due puntatori ai predecessori)
- Vedremo tra poco come risolvere gli eventuali conflitti

Merge: 3-way merge



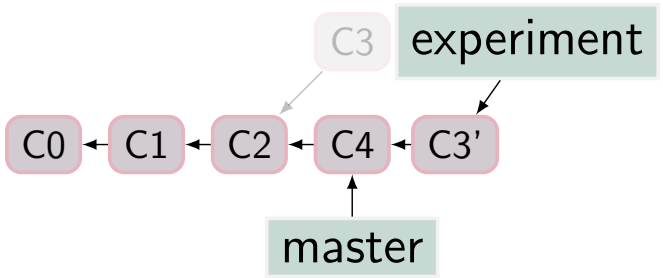
- Possiamo continuare a lavorare su experiment "in parallelo" o fare il merge con master
- `git checkout experiment`
- `git merge master`
- Questo merge sarà un fast forward dato che ora possiamo raggiungere experiment da master con un percorso orientato

Rebase



- Eseguiamo un rebase di experiment su master
- `git checkout experiment`
- `git rebase master`
- Git "riavvolge" tutto il lavoro che abbiamo fatto sul branch experiment fino al parente in comune con master
- Applica tutti i cambiamenti che sono stati fatti su master
- Tenta di applicare i commit che abbiamo fatto sul branch experiment partendo dall'ultimo commit del branch master
- Se tutto va bene ok, altrimenti genera uno o più conflitti

Rebase



- Il risultato finale è quello mostrato in figura
- Non viene creato un nuovo commit, è solo la history che viene modificata
- Non è consigliabile fare rebase dopo che si è pushato il proprio lavoro su un server remoto (potrebbe creare problemi agli altri utenti che usano il repo)
  - Quando dico "Non è consigliabile" intendo **"Non fatelo"**

Conflitti

```
→  lulzgit git:(master) git merge pippo
Auto-merging ciao
CONFLICT (content): Merge conflict in ciao
Merge automatico fallito; risolvi i conflitti ed eseguire il commit
del risultato.
```

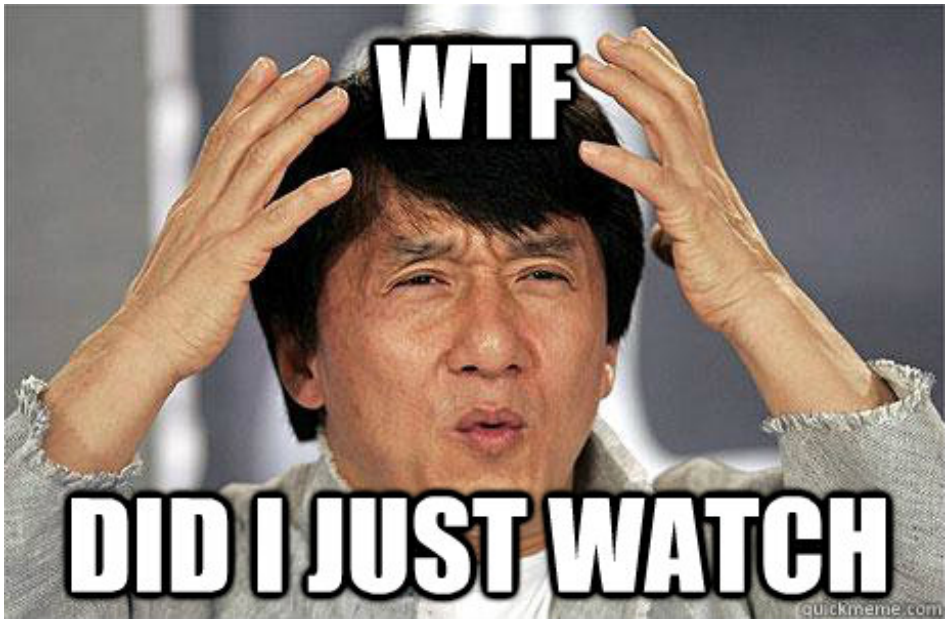
- Se esce un messaggio di questo tipo, c'è stato un conflitto
- I conflitti vengono segnalati all'interno del file con questa sintassi:
  <<<<<<< HEAD
  Cambiamenti presenti nel branch corrente
  =======
  Cambiamenti presenti nell'altro branch
  >>>>>>> nomealtrobranch

Risoluzione dei conflitti

- I conflitti si possono risolvere anche "a mano" con un editor di testo
- Esistono però dei tool appositi
- Se ne avete installato uno, eseguendo `git mergetool` verrà eseguito
  - Se non lo avete installato `git mergetool` vi darà comunque un elenco di tool che si possono usare
- Dopo aver risolto i conflitti di un merge bisogna eseguire `git commit`
- Dopo aver risolto i conflitti di un rebase bisogna eseguire `git rebase --continue`

Demo

Domande?

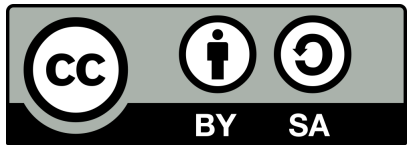


Fonti

- http://git-scm.com/book
- https://stackoverflow.com/
- Google, your usual friend

That's all folks

Grazie per l'attenzione!



`http://www.poul.org`