Git: funzioni avanzate

Emanuele Santoro
manu@santoro.tk

Corso Git 2016

Sezione 1

Intro

Lo scopo di questa lezione é di spiegare ed ove possibile esemplificare gli strumenti che git offre per andare oltre le operazioni di base (add, commit, fetch/push, pull) e collaboare in maniera efficace ed efficiente con altre persone su un progetto software usando git.

Topics - di cosa parleremo oggi

Questi sono gli argomenti che affronteremo oggi:

1. Bisect
2. Stash
3. Rebase
4. Tag
5. Blame
6. Varie ed eventuali

Sezione 2

Git bisect

Git bisect: ricerca binaria di un commit specifico, alla ricerca di bug (o altro).

- Puó aiutarci a rispondere a domande del tipo:
  - Quale commit ha introdotto questo bug? (e chi è il responsabile?)
  - C'è un bug di sicurezza, da quanto tempo è in produzione?
  - Quando è stato aggiunto questo file al repository? E da chi?

Git bisect (2)

Come funziona?

- Git può, considerando il branch corrente come una successione di commit, provare ad individuare il commit che introduce il problema con una ricerca binaria.
- Ci sono quattro comandi fondamentali:
  1. `git bisect start`
  2. `git bisect good`
  3. `git bisect bad`
  4. `git bisect reset`

  - Extra: `git bisect run`

Git bisect (3)

All'inizio della ricerca:

- `git bisect start`
  - Si inizia il processo di ricerca binaria
- `git bisect good <commit>`
  - marca lo stato del progetto al momento di un certo commit come positivo
- `git bisect bad [<commit>]`
  - marca lo stato del progetto al momento di un certo commit come negativo (se <commit> non é specificato, viene considerato l'ultimo commit)
- good/bad insieme servono a delimitare l'insieme di commit su cui effettuare la ricerca binaria!

Git bisect (3)

All'inizio della ricerca:

- `git bisect start`
  - Si inizia il processo di ricerca binaria
- `git bisect good <commit>`
  - marca lo stato del progetto al momento di un certo commit come positivo
- `git bisect bad [<commit>]`
  - marca lo stato del progetto al momento di un certo commit come negativo (se <commit> non é specificato, viene considerato l'ultimo commit)
- good/bad insieme servono a delimitare l'insieme di commit su cui effettuare la ricerca binaria!

Git bisect (3)

All'inizio della ricerca:

- `git bisect start`
  - Si inizia il processo di ricerca binaria
- `git bisect good <commit>`
  - marca lo stato del progetto al momento di un certo commit come positivo
- `git bisect bad [<commit>]`
  - marca lo stato del progetto al momento di un certo commit come negativo (se <commit> non é specificato, viene considerato l'ultimo commit)
- good/bad insieme servono a delimitare l'insieme di commit su cui effettuare la ricerca binaria!

Git bisect (3)

All'inizio della ricerca:

- `git bisect start`
  - Si inizia il processo di ricerca binaria
- `git bisect good <commit>`
  - marca lo stato del progetto al momento di un certo commit come positivo
- `git bisect bad [<commit>]`
  - marca lo stato del progetto al momento di un certo commit come negativo (se <commit> non é specificato, viene considerato l'ultimo commit)
- good/bad insieme servono a delimitare l'insieme di commit su cui effettuare la ricerca binaria!

Git bisect (4)

- **Durante la ricerca**:
  - `git bisect good`
    - ⋆ Marca il commit in esame come positivo
  - `git bisect bad`
    - ⋆ Marca marca il commit attuale come negativo
- In $log_2 n$ iterazioni git dovrebbe trovare (se presente) il commit incriminato!
- Una volta individuato il commit incriminato, si puó riportare il repository in uno stato operativo, per poterci lavorare sopra:
  - `git bisect reset`
- `git bisect reset` pone fine al processo di ricerca binaria

Git bisect (4)

- **Durante la ricerca**:
  - `git bisect good`
    - ⋆ Marca il commit in esame come positivo
  - `git bisect bad`
    - ⋆ Marca marca il commit attuale come negativo
- In $log_2 n$ iterazioni git dovrebbe trovare (se presente) il commit incriminato!
- Una volta individuato il commit incriminato, si puó riportare il repository in uno stato operativo, per poterci lavorare sopra:
  - `git bisect reset`
- `git bisect reset` pone fine al processo di ricerca binaria

Git bisect (4)

- **Durante la ricerca**:
  - `git bisect good`
    - ⋆ Marca il commit in esame come positivo
  - `git bisect bad`
    - ⋆ Marca marca il commit attuale come negativo
- In $log_2 n$ iterazioni git dovrebbe trovare (se presente) il commit incriminato!
- Una volta individuato il commit incriminato, si puó riportare il repository in uno stato operativo, per poterci lavorare sopra:
  - `git bisect reset`
- `git bisect reset` pone fine al processo di ricerca binaria

Git bisect (4)

- **Durante la ricerca**:
  - `git bisect good`
    - ⋆ Marca il commit in esame come positivo
  - `git bisect bad`
    - ⋆ Marca marca il commit attuale come negativo
- In $log_2 n$ iterazioni git dovrebbe trovare (se presente) il commit incriminato!
- Una volta individuato il commit incriminato, si puó riportare il repository in uno stato operativo, per poterci lavorare sopra:
  - `git bisect reset`
- `git bisect reset` pone fine al processo di ricerca binaria

Git bisect (5)

Extra: `git bisect run`

- Il test del repository può essere specificato come un test (codice eseguibile) che git eseguirà ad ogni iterazione per determinare lo stato positivo/negativo del repository dopo ogni commit.
- Sintassi:
  - `user@host:~/code $ git bisect run <script> <argomenti>`
  - `<script>` deve essere un eseguibile, che accetta `<argomenti>` sulla riga di comando
- Per segnalare lo stato positivo/negativo del repository, `<script>` usa il codice di uscita:
  - 0 – bug assente
  - != 0 – bug presente

Git bisect (6)

Esempi:

- `git bisect run make` #quale bug introduce un errore di compilazione?
- `git bisect run make test` #quale commit rompe la suite di test?

Git bisect (6)

Esempi:

- `git bisect run make` #quale bug introduce un errore di compilazione?
- `git bisect run make test` #quale commit rompe la suite di test?

Git bisect (6)

Esempi:

- `git bisect run make` #quale bug introduce un errore di compilazione?
- `git bisect run make test` #quale commit rompe la suite di test?

Git bisect (esempio)

Esempi:

- Quale commit introduce delle chiavi ssh nel repository?
  - Le chiavi sono nella cartella ssh_keys/

```
#!/usr/bin/env python
import os,sys
if os.path.exists('./ssh_keys') :
»       sys.exit(1)
else :
»       sys.exit(0)
```

```
user@host:~/code $ git bisect start
user@host:~/code $ git bisect bad
user@host:~/code $ git bisect good
b2a5a24991d0044649128426b8d7600c02f0fc03
user@host:~/code $ git bisect run ./test.py
user@host:~/code $ git bisect reset
```

Git bisect: protips

Per andare avanti con le iterazioni di bisect, bisogna esaminare lo stato del repository ad ogni iterazione.
Se non é possibile automatizzare l'ispezione, git consente di marcare manualmente un commit good/bad (new/old).
Succede di sbagliare, e di marcare erroneamente un commit.
In questi casi, git consente di salvare su un file le iterazioni che sono state eseguite e di ri-eseguirle in un secondo momento (eventualmente eliminando l'iterazione sbagliata).

Sezione 3

Cos'é?

A volte puó capitare di dover interrompere il lavoro su un branch prematuramente, e di dover passare ad un altro branch per fare altro. Oppure puó capitare di rendersi conto di aver cominciato a lavorare sul branch sbagliato.

L'operazione di stash consente di mettere da parte salvare un'area temporanea (lo stash, appunto) il lavoro fin'ora compleato al fine di poter cambiare branch (o anche solo fare altro), per poi riprendere in mano il salvataggio in un secondo momento.

Cos'é?

A volte puó capitare di dover interrompere il lavoro su un branch prematuramente, e di dover passare ad un altro branch per fare altro.
Oppure puó capitare di rendersi conto di aver cominciato a lavorare sul branch sbagliato.
L'operazione di stash consente di mettere da parte salvare un'area temporanea (lo stash, appunto) il lavoro fin'ora compleato al fine di poter cambiare branch (o anche solo fare altro), per poi riprendere in mano il salvataggio in un secondo momento.

Git stashing

Git stash: salvare delle modifiche non ancora complete (non abbastanza per fare un commit), lavorare su un altro branch per poi tornare sul ramo precedente, riapplicare le modifiche parziali, ultimarle, fare il commit.

```
user@host :~/code $ ## salvare le modifiche
user@host :~/code $ git stash
user@host :~/code $ ## cambio branch
user@host :~/code $ git checkout master
user@host :~/code $ emacs src/main.c
user@host :~/code $ ## hack hack hack
user@host :~/code $ ## torno sul mio branch
user@host :~/code $ git checkout altro_branch
user@host :~/code $ ## applicare le modifiche
user@host :~/code $ git stash apply
```

Git stashing

Git stash: salvare delle modifiche non ancora complete (non abbastanza per fare un commit), lavorare su un altro branch per poi tornare sul ramo precedente, riapplicare le modifiche parziali, ultimarle, fare il commit.

```
user@host :~/code $ ## salvare le modifiche
user@host :~/code $ git stash
user@host :~/code $ ## cambio branch
user@host :~/code $ git checkout master
user@host :~/code $ emacs src/main.c
user@host :~/code $ ## hack hack hack
user@host :~/code $ ## torno sul mio branch
user@host :~/code $ git checkout altro_branch
user@host :~/code $ ## applicare le modifiche
user@host :~/code $ git stash apply
```

Sezione 4

Riscrivere la history

- Errore nell'ultimo commit
  - `user@host :~/code $ git commit --amend`
- Abbiamo dimenticato di aggiungere un file al commit precedente:
  - `user@host :~/code $ git add file_mancante`
  - `user@host :~/code $ git commit --amend`
- **Attenzione**: `git commit --amend` non modifica l'ultimo commit, ma lo **sostituisce** completamente con uno che integra le nostre modifiche. Pertanto, non bisogna modificare dei commit già pubblicati!

Riscrivere la history

- Errore nell'ultimo commit
  - `user@host :~/code $ git commit --amend`
- Abbiamo dimenticato di aggiungere un file al commit precedente:
  - `user@host :~/code $ git add file_mancante`
  - `user@host :~/code $ git commit --amend`
- **Attenzione**: `git commit --amend` non modifica l'ultimo commit, ma lo **sostituisce** completamente con uno che integra le nostre modifiche. Pertanto, non bisogna modificare dei commit già pubblicati!

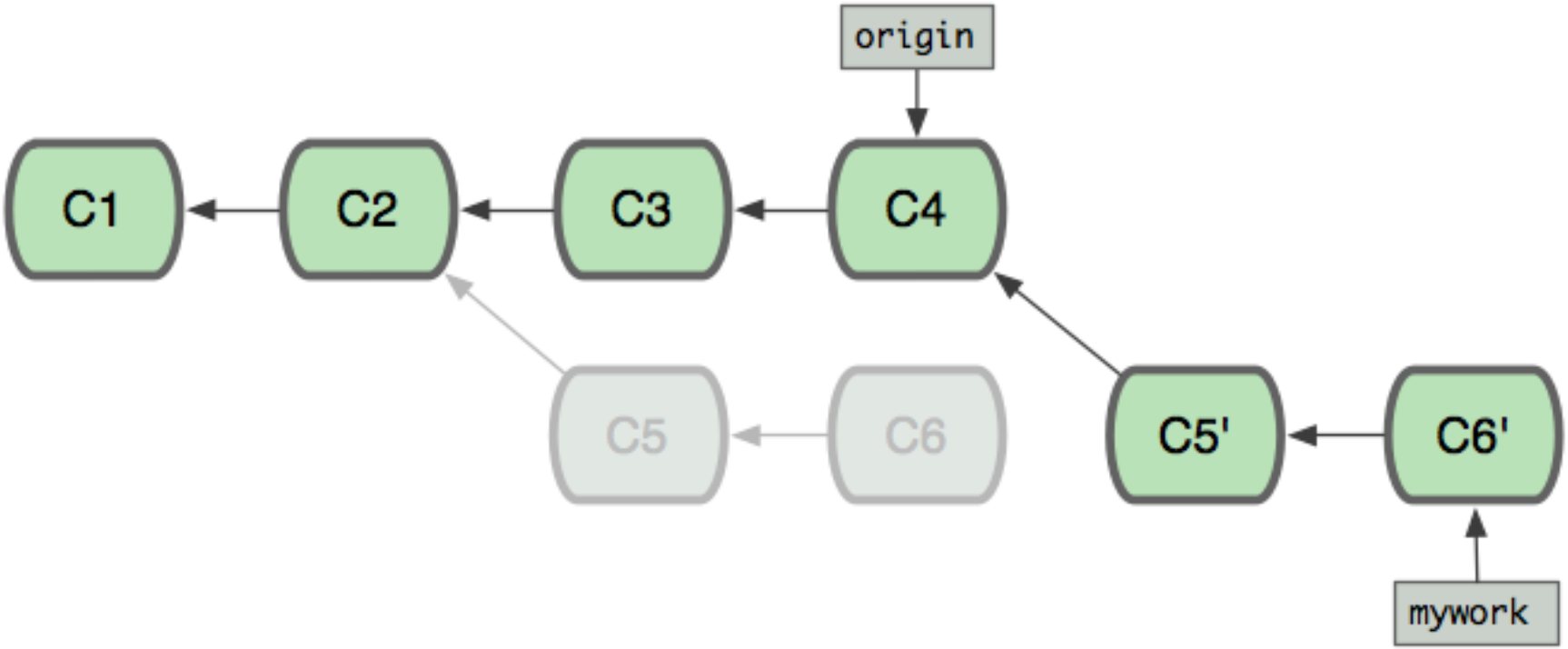
Riscrivere la history

- Errore nell'ultimo commit
  - `user@host :~/code $ git commit --amend`
- Abbiamo dimenticato di aggiungere un file al commit precedente:
  - `user@host :~/code $ git add file_mancante`
  - `user@host :~/code $ git commit --amend`
- **Attenzione**: `git commit --amend` non modifica l'ultimo commit, ma lo **sostituisce** completamente con uno che integra le nostre modifiche. Pertanto, non bisogna modificare dei commit già pubblicati!

Sezione 5

Cos'é?

rebase «sposta» un sottoramo



Facendo alla figura, il branch mywork parte dal commit C2 ed é composto dai commit C5 e C6.
rebase distrugge i commit C5 e C6, e crea i commit C5' e C6' che apportano le stesse modifiche di C5 e C6, ma a partire dal commit C4.

Perché fare rebase?

Due casi d'uso principali:

1. si ribasa il proprio lavoro sull'ultimo commit del ramo master
2. (si prova a) eliminare qualche errore commesso in passato

Rebase sull'ultimo commit

Quando un branch B si puó fondere con un branch A semplicemente appendendo tutti i commit di B su A, si dice che i branch A e B sono «fast-forward mergeable». Tipicamente, questo vuol dire che git sposta il puntatore dell'ultimo commit di A con al puntatore dell'ultimo commit di B, in maniera molto veloce.

Oltre ad essere veloce come operazione, due branch fast-forward-mergeable non danno conflitti da risolvere (ovvero: non devi litigare con qualcuno per fondere due rami).

Ribasare il proprio ramo di lavoro su quello principale rendere questi due rami fast-forward-mergeable, quindi fondibili in maniera immediata e senza problemi.

Praticamente vuol dire rendere enormemente piú facile fondere due rami e quindi integrare una patch fornita da qualcuno, specialmente in progetti open source.

Rebase sull'ultimo commit

Quando un branch B si puó fondere con un branch A semplicemente appendendo tutti i commit di B su A, si dice che i branch A e B sono «fast-forward mergeable». Tipicamente, questo vuol dire che git sposta il puntatore dell'ultimo commit di A con al puntatore dell'ultimo commit di B, in maniera molto veloce.
Oltre ad essere veloce come operazione, due branch fast-forward-mergeable non danno conflitti da risolvere (ovvero: non devi litigare con qualcuno per fondere due rami).
Ribasare il proprio ramo di lavoro su quello principale rendere questi due rami fast-forward-mergeable, quindi fondibili in maniera immediata e senza problemi.
Praticamente vuol dire rendere enormemente piú facile fondere due rami e quindi integrare una patch fornita da qualcuno, specialmente in progetti open source.

Rebase sull'ultimo commit

Quando un branch B si puó fondere con un branch A semplicemente appendendo tutti i commit di B su A, si dice che i branch A e B sono «fast-forward mergeable». Tipicamente, questo vuol dire che git sposta il puntatore dell'ultimo commit di A con al puntatore dell'ultimo commit di B, in maniera molto veloce.
Oltre ad essere veloce come operazione, due branch fast-forward-mergeable non danno conflitti da risolvere (ovvero: non devi litigare con qualcuno per fondere due rami).
Ribasare il proprio ramo di lavoro su quello principale rendere questi due rami fast-forward-mergeable, quindi fondibili in maniera immediata e senza problemi.
Praticamente vuol dire rendere enormemente piú facile fondere due rami e quindi integrare una patch fornita da qualcuno, specialmente in progetti open source.

Rebase sull'ultimo commit

Quando un branch B si puó fondere con un branch A semplicemente appendendo tutti i commit di B su A, si dice che i branch A e B sono «fast-forward mergeable». Tipicamente, questo vuol dire che git sposta il puntatore dell'ultimo commit di A con al puntatore dell'ultimo commit di B, in maniera molto veloce.
Oltre ad essere veloce come operazione, due branch fast-forward-mergeable non danno conflitti da risolvere (ovvero: non devi litigare con qualcuno per fondere due rami).
Ribasare il proprio ramo di lavoro su quello principale rendere questi due rami fast-forward-mergeable, quindi fondibili in maniera immediata e senza problemi.
Praticamente vuol dire rendere enormemente piú facile fondere due rami e quindi integrare una patch fornita da qualcuno, specialmente in progetti open source.

Sezione 6

Git tag: dare i nomi ai tag

Un tag é un nome associato ad un commit. L'uso tipico é di associare un nome significativo per l'organizzazione ad un commit.
Esempio classico: ogni release di un progetto viene taggata (v1, v1.1, v1.2, v1.2.1, v2 etc) e diventa possibile riferirsi allo stato di un progetto in un particolare istante di tempo non solo in base all'hash del commit ma anche tramite il tag associato.

Tag annotati e tag semplici

Ci sono due tipi di tag: tag annotati e tag semplici.
I tag semplici sono dei puntatori ad un tag e niente piú.
I tag annotati sono oggetti a se stanti, con tutti i metadati del caso (data, ora, autore del tag, firma crittografica, commit puntato) ed un messaggio allegato per fornire una descrizione del tag (esempio: viene taggata una release, si allega un changelog.

Tag annotati e tag semplici

Ci sono due tipi di tag: tag annotati e tag semplici.
I tag semplici sono dei puntatori ad un tag e niente piú.
I tag annotati sono oggetti a se stanti, con tutti i metadati del caso (data, ora, autore del tag, firma crittografica, commit puntato) ed un messaggio allegato per fornire una descrizione del tag (esempio: viene taggata una release, si allega un changelog.

Tag annotati e tag semplici

Ci sono due tipi di tag: tag annotati e tag semplici.
I tag semplici sono dei puntatori ad un tag e niente piú.
I tag annotati sono oggetti a se stanti, con tutti i metadati del caso (data, ora, autore del tag, firma crittografica, commit puntato) ed un messaggio allegato per fornire una descrizione del tag (esempio: viene taggata una release, si allega un changelog.

Creare un tag

- Creare un tag annotato:
  - `$ git tag -a v1.4 -m "VERSION 1.4, MANY BUGS FIXED"`
- Creare un tag leggero:
  - `$ git tag v1.4-semplice`
- Creare un tag per un commit specifico:
  - `$ git tag -a v1.2 9fceb02`
- vedere i dettagli di un tag :
  - `$ git show v1.4`

Condividere i tag

Come comportamento predefinito, tutti i tag sono locali. Pubblicando un ramo di sviluppo su un repository remoto non vengono pubblicati anche i tag che puntano ai commit pubblicati.
I devono essere pubblicati esplicitamente. Si puó decidere di pubblicare un tag singolarmente o anche tutti i tag:

- Pubblicazione di un tag singolo:
  - `$ git push origin v1.5`
- Pubblicazione di tutti i tag:
  - `$ git push origin --tags`

Sezione 7

Git blame

- Git blame ci aiuta a rispondere a queste domande!
- Sintassi: `git blame [opzioni] file`
- Esempi:
  - Chi ha modificato il file src/core/server.c ?
    - `user@host:~/code $ git blame src/core/server.c`
  - Chi ha modificato le linee 10-15 del file src/core/server.c ?
    - `user@host:~/code $ git blame -L 10,15 src/core/server.c`
  - Abbiamo appurato che nel file x e nella linea y c'é un bug. Chi é il responsabile?
    - Per rispondere a questa domanda, bisogna ispezionare il commit, e vederne l'autore.

Git blame

- Git blame ci aiuta a rispondere a queste domande!
- Sintassi: `git blame [opzioni] file`
- **Esempi:**
  - Chi ha modificato il file src/core/server.c ?
    - `user@host:~/code $ git blame src/core/server.c`
  - Chi ha modificato le linee 10-15 del file src/core/server.c ?
    - `user@host:~/code $ git blame -L 10,15 src/core/server.c`
  - Abbiamo appurato che nel file x e nella linea y c'é un bug. Chi é il responsabile?
    - Per rispondere a questa domanda, bisogna ispezionare il commit, e vederne l'autore.

Git blame

- Git blame ci aiuta a rispondere a queste domande!
- Sintassi: `git blame [opzioni] file`
- **Esempi:**
  - Chi ha modificato il file src/core/server.c ?
    - `user@host:~/code $ git blame src/core/server.c`
  - Chi ha modificato le linee 10-15 del file src/core/server.c ?
    - `user@host:~/code $ git blame -L 10,15 src/core/server.c`
  - Abbiamo appurato che nel file x e nella linea y c'é un bug. Chi é il responsabile?
    - Per rispondere a questa domanda, bisogna ispezionare il commit, e vederne l'autore.

Git blame

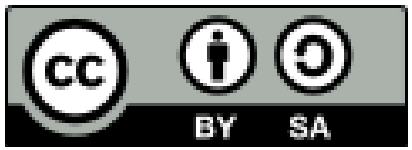
- Git blame ci aiuta a rispondere a queste domande!
- Sintassi: `git blame [opzioni] file`
- **Esempi:**
  - Chi ha modificato il file src/core/server.c ?
    - `user@host:~/code $ git blame src/core/server.c`
  - Chi ha modificato le linee 10-15 del file src/core/server.c ?
    - `user@host:~/code $ git blame -L 10,15 src/core/server.c`
  - Abbiamo appurato che nel file x e nella linea y c'é un bug. Chi é il responsabile?
    - Per rispondere a questa domanda, bisogna ispezionare il commit, e vederne l'autore.

Fonti

- Pro Git book (Scott Chacon): `http://git-scm.com/book/en/Distributed-Git-Distributed-Workflows`

Fine

Grazie per l'attenzione!



`http://www.poul.org`