Introduzione a `git`

Luca Tagliavini, Stefano Volpe

Università di Bologna, corso di Laurea in Informatica

8 novembre 2022

In questa sezione

Problemi

Quando lavoriamo con molti *file* (redigiamo documenti, creiamo arte digitale, scriviamo codice...), ricorrono alcuni **problemi**:

Problemi

Quando lavoriamo con molti *file* (redigiamo documenti, creiamo arte digitale, scriviamo codice...), ricorrono alcuni **problemi**:

- mantenere una **cronologia** delle modifiche;

Problemi

Quando lavoriamo con molti *file* (redigiamo documenti, creiamo arte digitale, scriviamo codice...), ricorrono alcuni **problemi**:

- mantenere una **cronologia** delle modifiche;
- **ripristinare** subito un qualsiasi stato precedente;

Problemi

Quando lavoriamo con molti *file* (redigiamo documenti, creiamo arte digitale, scriviamo codice...), ricorrono alcuni **problemi**:

- mantenere una **cronologia** delle modifiche;
- **ripristinare** subito un qualsiasi stato precedente;
- collaborare a **copie diverse** dello stesso progetto.

Problemi

Quando lavoriamo con molti *file* (redigiamo documenti, creiamo arte digitale, scriviamo codice...), ricorrono alcuni **problemi**:

- mantenere una **cronologia** delle modifiche;
- **ripristinare** subito un qualsiasi stato precedente;
- collaborare a **copie diverse** dello stesso progetto.

Copie di cartelle e blocchi note condivisi in rete non bastano più!

Rimedi

Definizione

Un **sistema per il controllo di versione** (o VCS, *version control system*) è un applicativo che gestisce modifiche a grandi insiemi di informazioni.

Rimedi

Definizione

Un **sistema per il controllo di versione** (o VCS, *version control system*) è un applicativo che gestisce modifiche a grandi insiemi di informazioni.

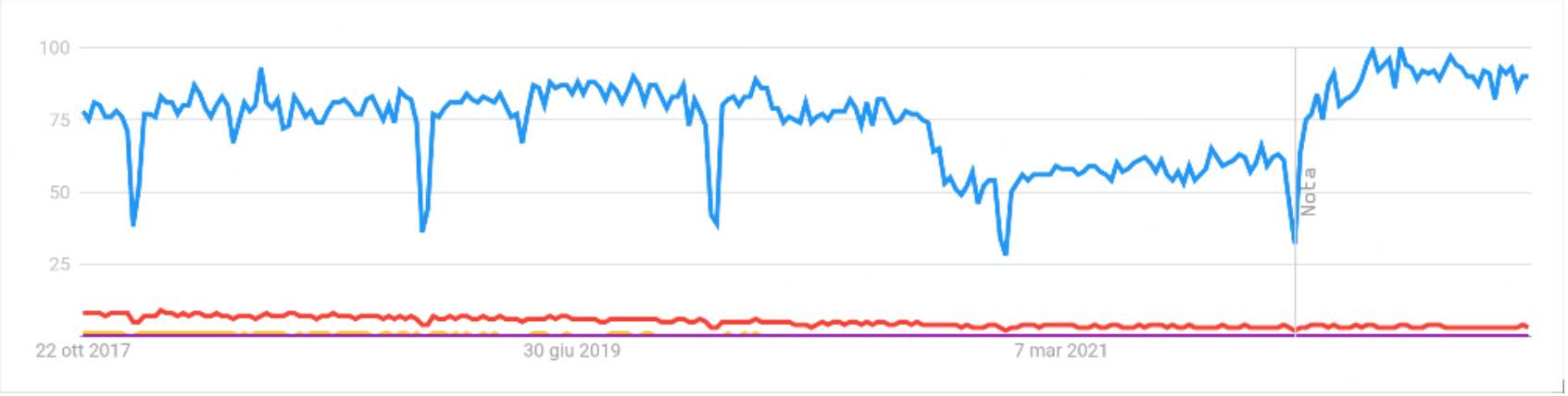

Figura: popolarità su Google dei cinque principali VCS (`git`, SVN, Mercurial, Perforce, CVS) negli ultimi cinque anni. In ordinata, 100 indica il massimo storico del più popolare di questi in tale lasso di tempo.

Ambiente di lavoro | Scopo

Nei prossimi lucidi, collegandoci alle macchine di laboratorio con i nostri utenti, useremo `git` da linea di comando.

Ambiente di lavoro | Scopo

Nei prossimi lucidi, collegandoci alle macchine di laboratorio con i nostri utenti, useremo `git` da linea di comando.

Consiglio

Tenete aperta questa presentazione anche sulla vostra macchina durante il laboratorio: potete trovarla su ▸ csunibo.github.io/lab.

Ambiente di lavoro | Materiali

Materiali:

1. un utente nella rete dipartimentale (es. `stefano.volpe2`);

Se siete senza utente...

... (e non avete `git` neanche sulla vostra macchina), seguite con chi siede al vostro fianco.

Ambiente di lavoro | Materiali

Materiali:

1. un utente nella rete dipartimentale (es. `stefano.volpe2`);

Se siete senza utente...

... (e non avete `git` neanche sulla vostra macchina), seguite con chi siede al vostro fianco.

2. un client `ssh`. Su Linux e macOS è già installato; su Windows, se vi manca, c'è ▸ PuTTY;

Ambiente di lavoro | Materiali

Materiali:

1. un utente nella rete dipartimentale (es. `stefano.volpe2`);

Se siete senza utente...

... (e non avete `git` neanche sulla vostra macchina), seguite con chi siede al vostro fianco.

2. un client `ssh`. Su Linux e macOS è già installato; su Windows, se vi manca, c'è ▸ PuTTY;
3. un utente su una piattaforma di *hosting* per progetti `git`.

Ambiente di lavoro | Preparazione

Preparazione:

1. scegliere una delle ▸ macchine di laboratorio (es. `XXX.cs.unibo.it`, dove `XXX` è il nome della macchina scelta)

Ambiente di lavoro | Preparazione

Preparazione:

1. scegliere una delle ▸ macchine di laboratorio (es. `XXX.cs.unibo.it`, dove `XXX` è il nome della macchina scelta)
2. collegarsi:

   ```
   $ ssh stefano.volpe2@XXX.cs.unibo.it
   ```

Ambiente di lavoro | Preparazione (2)

```
[...]
Are you sure you want to continue connecting
(yes/no/[fingerprint])? yes
[...]
stefano.volpe2@XXX.cs.unibo.it's password:
```

Ambiente di lavoro | Preparazione (3)

3. configurare `git` con i vostri dati:

```
$ git config --global user.name "Stefano Volpe"
$ git config --global user.email \
    "stefano.volpe2@studio.unibo.it"
```

4. generare una chiave ssh:

```
$ ssh-keygen -t ed25519 -C \
    "stefano.volpe2@studio.unibo.it"
```

In questa sezione

`git init`

Definizione

Una *repository* è un progetto manutenuto con `git`.

`git init`

Definizione

Una *repository* è un progetto manutenuto con `git`.

Per inizializzare una *repository* Git vuota in una nuova cartella `<directory>`:

```
$ git init [<directory>]
```

Nota

I file interni di `git` sono nella sottocartella `<directory>/.git/`

Nota 2

Se `<directory>` non è specificata, viene usata la cartella corrente.

Stati dei *file*

I *file* di una *repository* sono sempre in esattamente uno dei seguenti quattro stati:

1. *untracked* (non tracciato da `git`);

Stati dei *file*

I *file* di una *repository* sono sempre in esattamente uno dei seguenti quattro stati:

1. *untracked* (non tracciato da `git`);
2. *unmodified* (non modificato rispetto all'ultima "istantanea" di `git`);

Stati dei *file*

I *file* di una *repository* sono sempre in esattamente uno dei seguenti quattro stati:

1. *untracked* (non tracciato da `git`);
2. *unmodified* (non modificato rispetto all'ultima "istantanea" di `git`);
3. *modified* (modificato rispetto all'ultima "istantanea" di `git`);

Stati dei *file*

I *file* di una *repository* sono sempre in esattamente uno dei seguenti quattro stati:

1. *untracked* (non tracciato da `git`);
2. *unmodified* (non modificato rispetto all'ultima "istantanea" di `git`);
3. *modified* (modificato rispetto all'ultima "istantanea" di `git`);
4. *staged* (modificato rispetto all'ultima "istantanea" di `git` e pronto a essere registrato).

git status

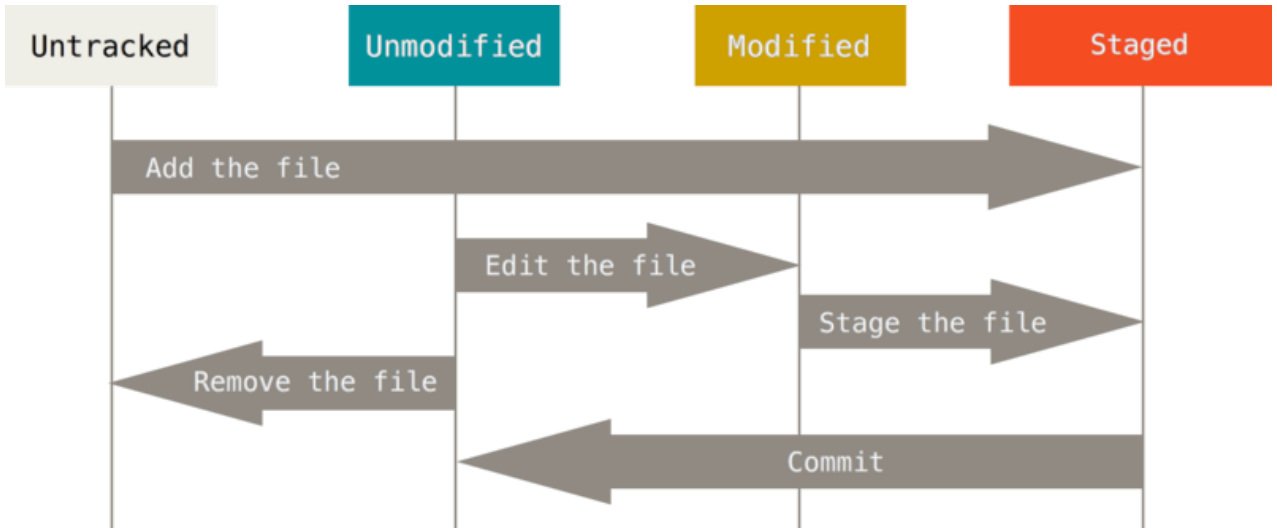

Figura: Gli stati fra cui i *file* di una *repository* si muovono. *Pro Git*, figura 8 di Scott Chacon e Ben Straub, rilasciata sotto licenza CC BY_NC_SA 3.0.

git status

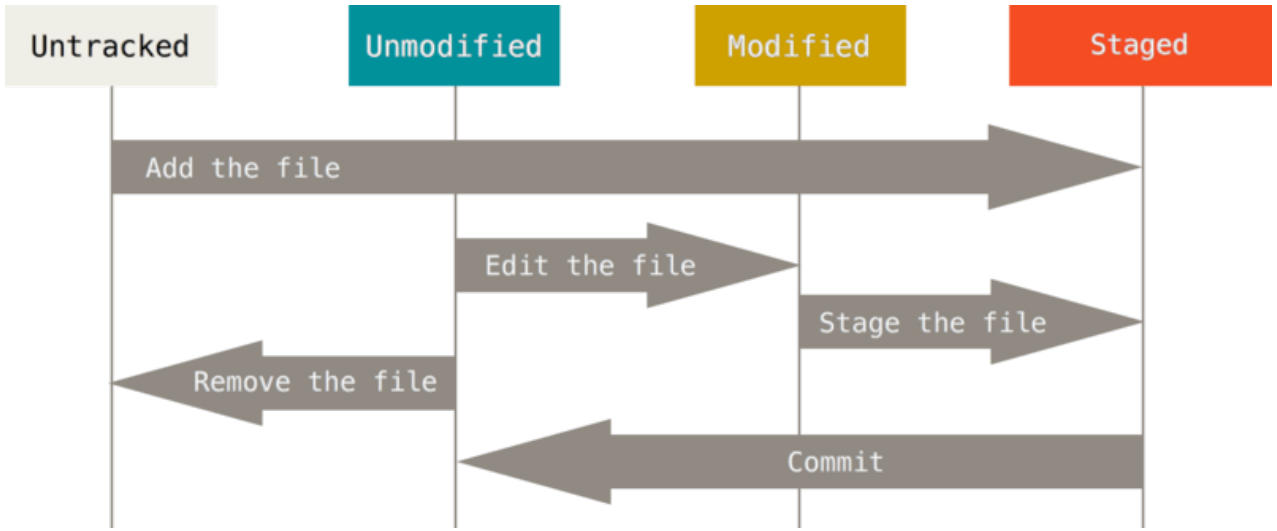

Figura: Gli stati fra cui i *file* di una *repository* si muovono. *Pro Git*, figura 8 di Scott Chacon e Ben Straub, rilasciata sotto licenza CC BY_NC_SA 3.0.

Con `git status`, mostriamo lo stato attuale del nostro "albero di lavoro".

`.gitignore`

Si possono ignorare file che non si desiera vengano aggiunti alla *repository*, come binari compilati e file di caching, elencandoli in un file denominato `.gitignore` situato nella radice dell'albero di lavoro.
La sintassi è dettata dalle seguenti regole:

1. una riga contiene il percorso ad un file da ignorare;
2. si possono usare raggruppamenti *glob* per selezionare più file in modo conciso;
3. iniziando la linea con un `!` il file viene **incluso** anche se ignorato.

git add

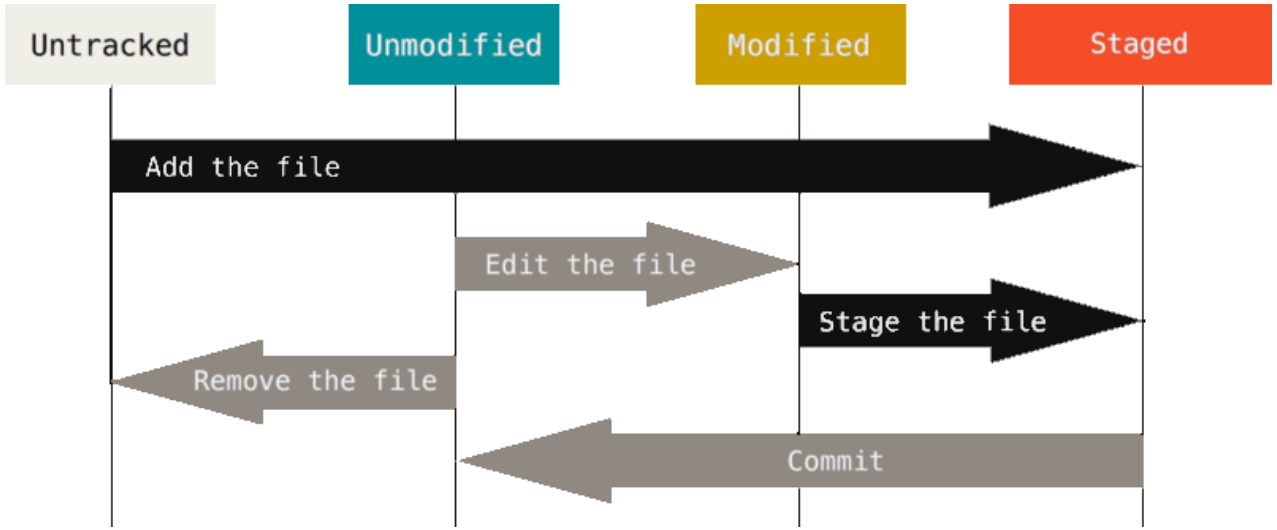

Figura: Gli stati fra cui i *file* di una *repository* si muovono. *Pro Git*, figura 8 di Scott Chacon e Ben Straub, rilasciata sotto licenza CC BY_NC_SA 3.0 / Parz. ricolorata.

Aggiunge `<pathspec>` (non tracciato o modificato) agli *staged files*:

```
$ git add [<pathspec>...]
```

git commit

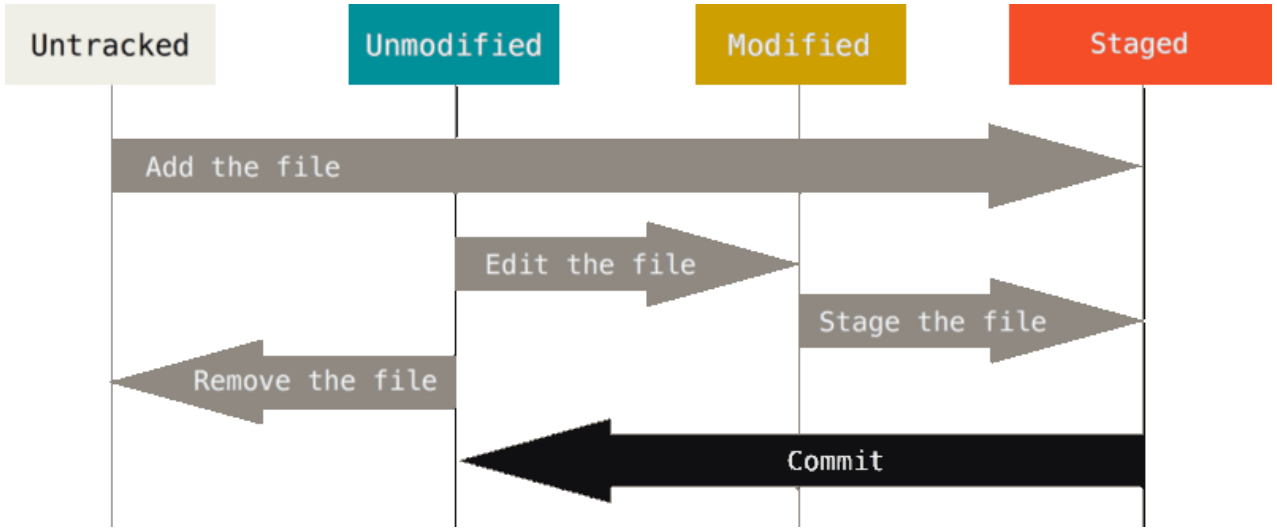

Figura: Gli stati fra cui i *file* di una *repository* si muovono. *Pro Git*, figura 8 di Scott Chacon e Ben Straub, rilasciata sotto licenza CC BY_NC_SA 3.0 / Parz. ricolorata.

Registra le modifiche agli *staged files* commentadole con `msg`: *staged files*:

```
$ git commit [--amend] [-m <msg>]
```

`git commit`

Nota

Se non uso `-m`, viene aperto un *buffer* di testo in cui posso specificare il messaggio.

Nota 2

Con `--amend`, correggo nome e/o contenuti del *commit* precedente.

`git log`

Mostriamo il registro dei *commit*:

```
$ git log [--graph]
```

Se aggiungiamo `--graph`, il registro è rappresentato come grafo.

`git show`

Possiamo mostrare dettagli su uno o più *commit* a scelta:

```
$ git show [<object> ...]
```

Un modo di riferirsi a un `<object>` è un prefisso univoco del codice `sha1` del *commit*.

`git show`

Possiamo mostrare dettagli su uno o più *commit* a scelta:

```
$ git show [<object> ...]
```

Un modo di riferirsi a un `<object>` è un prefisso univoco del codice `sha1` del *commit*. Se non si specifica nulla, viene usato `HEAD`, il *commit* corrente.

git branch, checkout

Per evitare confusione, vogliamo mantenere più versioni dello stesso progetto:

- quella "buona";
- quella su cui stiamo sviluppando una funzione ancora sperimentale;
- quella su cui stiamo correggendo un problema...

git branch, checkout

Per evitare confusione, vogliamo mantenere più versioni dello stesso progetto:

- quella "buona";
- quella su cui stiamo sviluppando una funzione ancora sperimentale;
- quella su cui stiamo correggendo un problema...

In `git`, ogni versione (*branch*, ramificazione) si alterna nella stessa cartella.

```
$ git branch # elenca le branch
```

git branch, checkout

```
$ git checkout -b <branch> # crea <branch> e vi si sposta
```

git branch, checkout

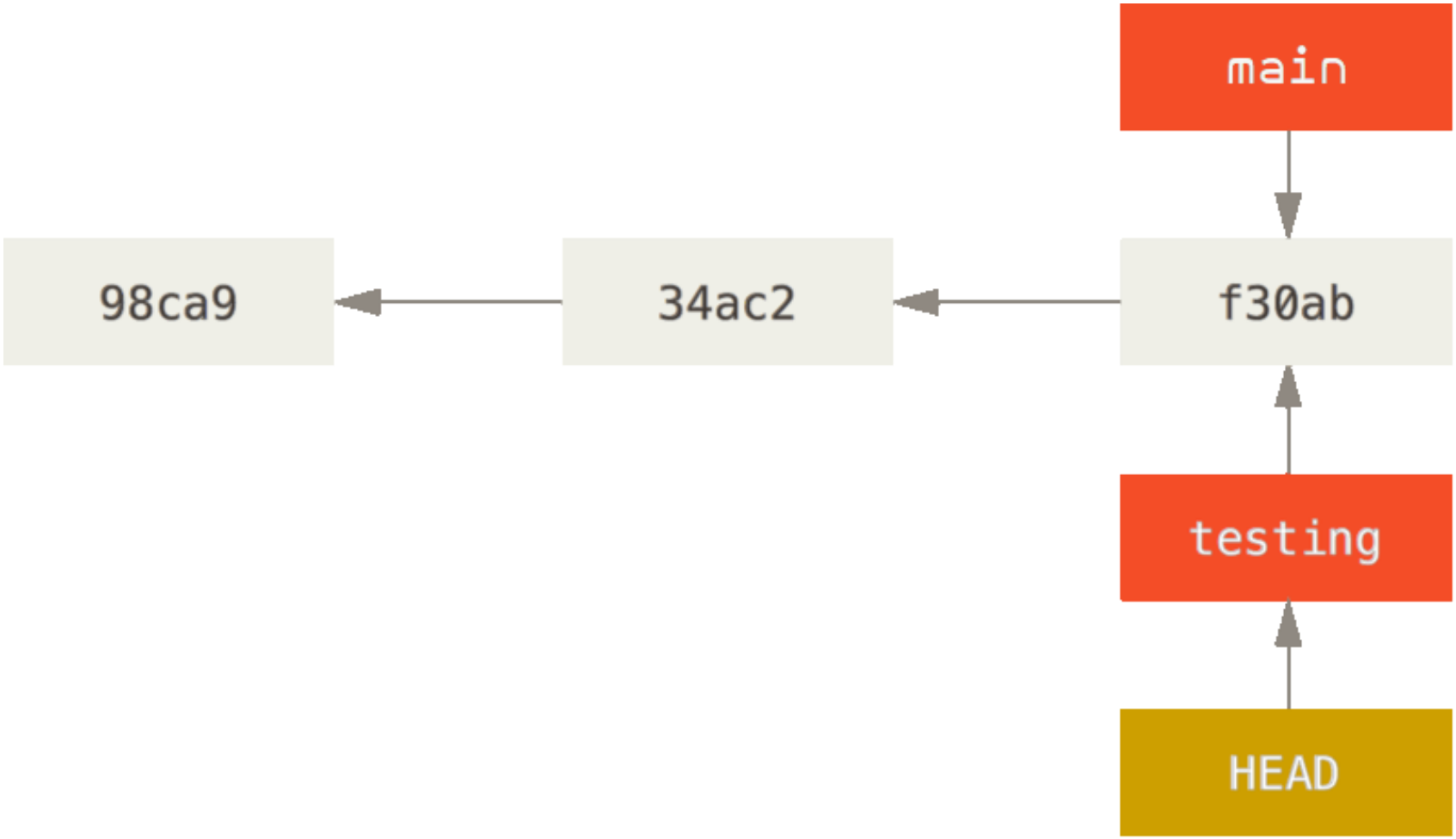

Figura: *Pro Git*, figura 14 di Scott Chacon e Ben Straub, rilasciata sotto licenza CC BY_NC_SA 3.0 / Testo modificato.

git branch, checkout

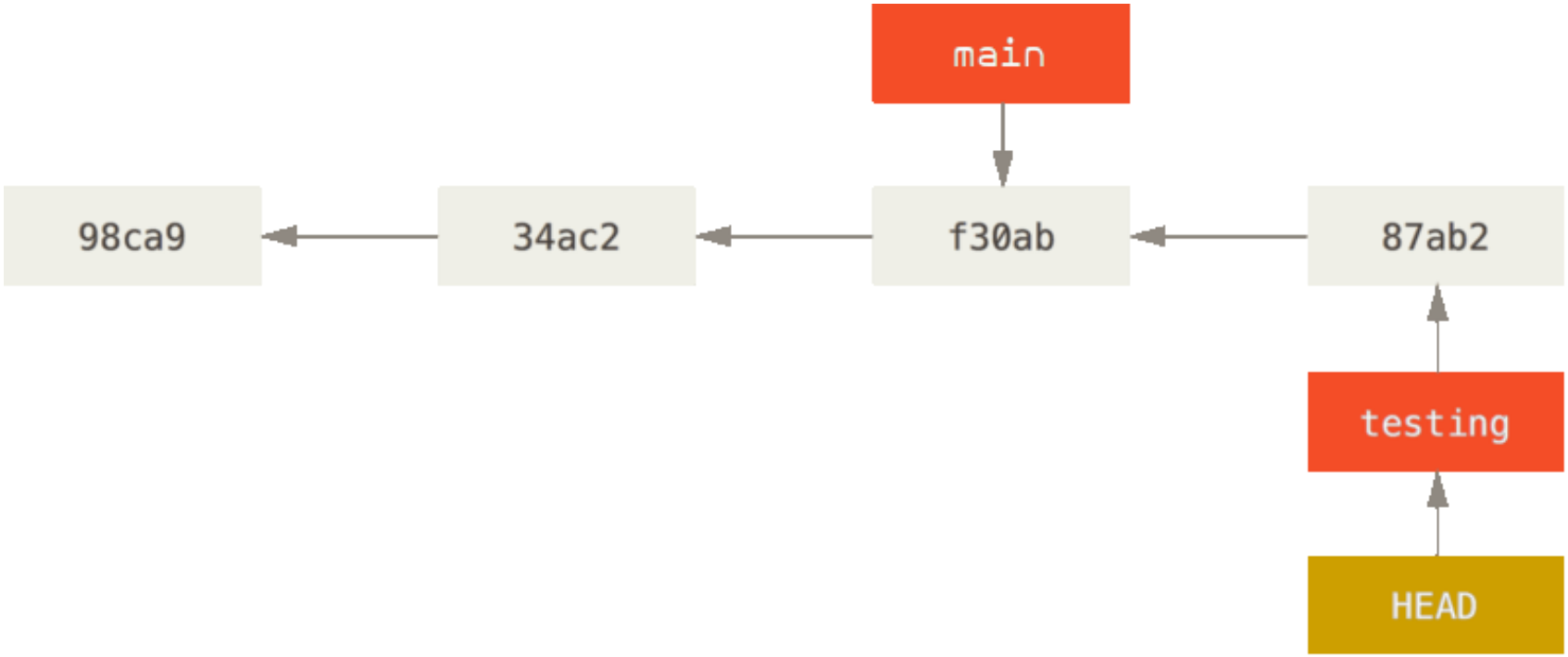

Figura: *Pro Git*, figura 15 di Scott Chacon e Ben Straub, rilasciata sotto licenza CC BY_NC_SA 3.0 / Testo modificato.

git branch, checkout

```
$ git checkout <branch> # si sposta su branch
```

git branch, checkout

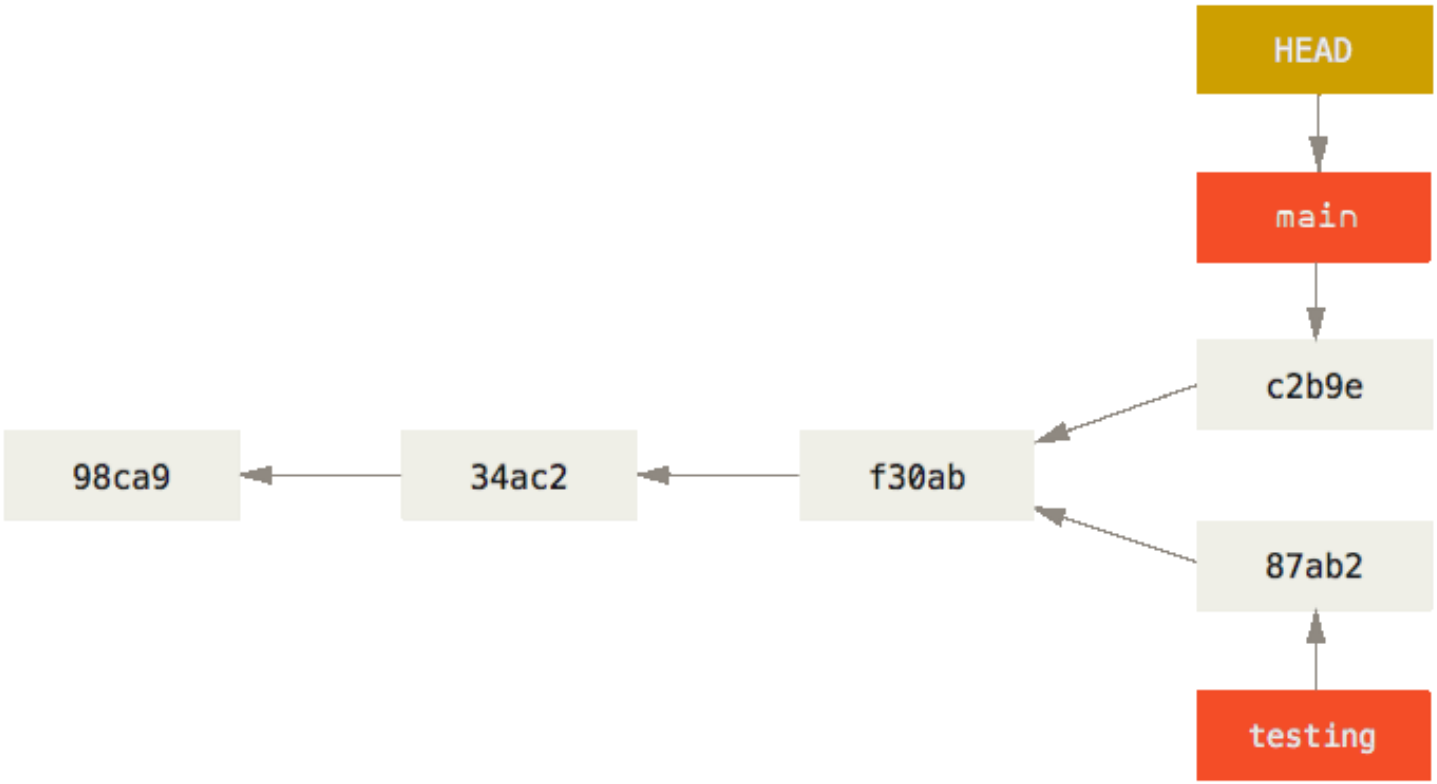

Figura: *Pro Git*, figura 17 di Scott Chacon e Ben Straub, rilasciata sotto licenza CC BY_NC_SA 3.0 / Testo modificato.

git branch, checkout

```
$ git branch -d <branch> # rimuove <branch>
```

In questa sezione

Perchè?

Una repository locale può essere collegata ad una **remota** per semplificare una serie di operazioni comuni e desiderabili:

- collaborazione tra sviluppatori, interni o esterni al proprio gruppo di lavoro;

Perchè?

Una repository locale può essere collegata ad una **remota** per semplificare una serie di operazioni comuni e desiderabili:

- collaborazione tra sviluppatori, interni o esterni al proprio gruppo di lavoro;
- distribuzione del codice (versionamento e rilascio);

Perchè?

Una repository locale può essere collegata ad una **remota** per semplificare una serie di operazioni comuni e desiderabili:

- collaborazione tra sviluppatori, interni o esterni al proprio gruppo di lavoro;
- distribuzione del codice (versionamento e rilascio);
- backup del codice.

`git remote [show]`

La gestione degli *endpoint* remoti a cui si vuole collegare la propria repository avviene tramite la famiglia di comandi `git remote`. Per ottenere la lista dei remoti collegati ad un albero di git si può usare:

```
$ git remote [-v]
```

Applicando la flag `-v` si rende la risposta verbosa e si ottengono ancora più dettagli, ad esempio gli `URI` degli endpoint[1].

[1] Per ottenere ancora più informazioni si può usare il comando `git remote show <name>`

`git remote add, remove`

Per aggiungere un nuovo remoto alla repository possiamo usare il comando:

```
$ git remote add <name> <uri>
```

Il nome standard per il remoto principale è *origin*. L'`URI` deve usare uno dei protoccoli accettati[2].

[2] Vedi: stackoverflow.com/a/51112344

`git remote add, remove`

Per aggiungere un nuovo remoto alla repository possiamo usare il comando:

```
$ git remote add <name> <uri>
```

Il nome standard per il remoto principale è *origin*. L'`URI` deve usare uno dei protoccoli accettati[2].
`remove` è il sottocomando speculare ad `add` che consente di rimuovere un remoto dato il nome:

```
$ git remote remove <name>
```

[2] Vedi: stackoverflow.com/a/51112344

`git remote add, remove` | Nota sugli URI

Nella maggior parte dei servizi moderni che ospitano repository di `git` l'utilizzo di `HTTP(S)` come protocollo per il collegamento remoto è stato deprecato (per effettuare `push`) per ragioni di sicurezza. In questa guida useremo sempre remoti con protocollo `SSH` e voi dovreste fare altrettanto.

git push

Una volta aggiunto un endpoint è possibile caricare i propri commit alla repository remota:

```
$ git push [-u <remote> <name>]
```

`git push`

Una volta aggiunto un endpoint è possibile caricare i propri commit alla repository remota:

```
$ git push [-u <remote> <name>]
```

Nota

Al primo `push` per una nuova branch sarà necessario impostare l'*upstream* per questo ramo, ovvero la branch remota a cui caricare i cambiamenti. La flag `-u` del comando `push` imposta una branch su un remoto come `upstream`.

git clone

Per duplicare una repository remota sul proprio sistema si ha a disposizione il comando:

```
$ git clone [-b <branch>] <uri>
```

Dove l'URI è l'indirizzo dove è salvato il vostro codice.

git clone

Per duplicare una repository remota sul proprio sistema si ha a disposizione il comando:

```
$ git clone [-b <branch>] <uri>
```

Dove l'URI è l'indirizzo dove è salvato il vostro codice.

Consiglio

Per clonare una specifica branch si può usare la flag `-b` specificando subito dopo il nome della branch remota desiderata.

`git merge, rebase`

Per unire il lavoro svolto su branch separate si hanno due opzioni:

1. `git merge <name>`: applica tutti i *nuovi commit*[3] presenti sulla branch `<name>` nella branch in cui ci troviamo;

[3] Tutti quelli a partire dal *fork-point*

`git merge, rebase`

Per unire il lavoro svolto su branch separate si hanno due opzioni:

1. `git merge <name>`: applica tutti i *nuovi commit*[3] presenti sulla branch `<name>` nella branch in cui ci troviamo;
2. `git rebase <name>`: mette da parte i *nuovi commit*[3], avanza la branch all'ultimo commit su `<name>` e riapplica i commit "messi da parte".

[3] Tutti quelli a partire dal *fork-point*

git merge

Il caso d'uso tipico per il comando `merge` è l'unione di una branch secondaria con quella principale. Il comando ha la seguente sintassi:

```
$ git merge <branch>
```

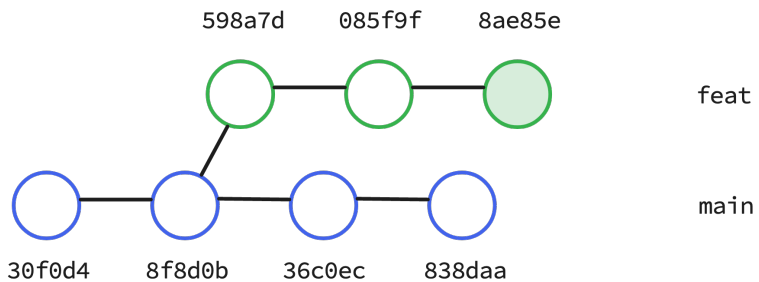

Figura: Log dei commit prima di un merge

git merge

Il caso d'uso tipico per il comando `merge` è l'unione di una branch secondaria con quella principale. Il comando ha la seguente sintassi:

```
$ git merge <branch>
```

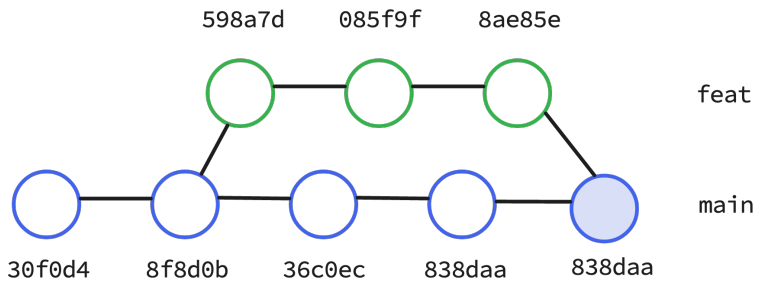

Figura: Log dei commit dopo `git merge feat`

Nota

È sconsigliato usare `merge` con un spazio di lavoro sporco

git rebase

Il comando `rebase` si usa quando si vuole cambiare il commit da cui si dirama la branch attuale. La sintassi è la seguente:

```
$ git rebase <branch>
```

Di seguito è raffigurata una casistica d'esempio.

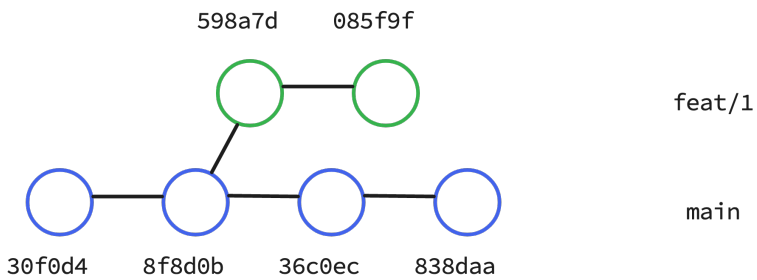

Figura: Log dei commit prima di un rebase

`git rebase`

Il comando `rebase` si usa quando si vuole cambiare il commit da cui si dirama la branch attuale. La sintassi è la seguente:

```
$ git rebase <branch>
```

Di seguito è raffigurata una casistica d'esempio.

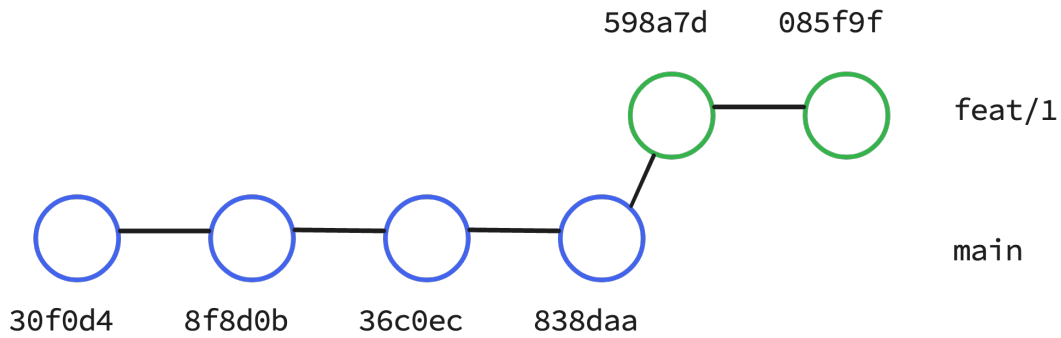

Figura: Log dei commit dopo `git rebase main`

`git merge, rebase` | Caso triviale

In un caso triviale, ovvero quando tra le branch da unificare solo una contiene cambiamenti dopo il *fork-point*, i comandi `merge` e `rebase` hanno lo stesso effetto[4]:

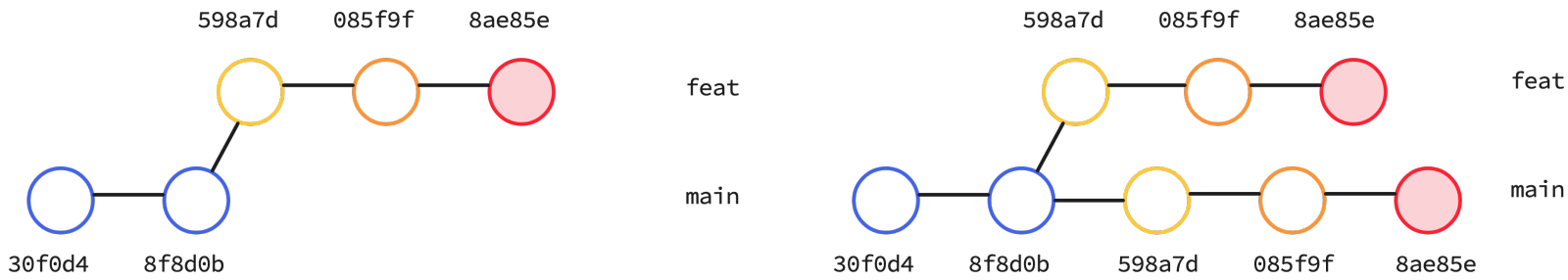

Figura: Prima e dopo un `git merge feat` o `git rebase feat`

[4] Nel caso del `merge` non viene aggiunto nessun commit, si dice che viene fatto un *fast forward*

`git merge, rebase` | Conflitti

Quando si vanno a unire due branch è possibile che si trovino conflitti tra i vari commit che si stanno provando ad unire. `git` si comporta in modo diverso in base al tipo di unione che si sta svolgendo:

- nel caso di un `merge` tutti i cambiamenti vengono applicati e i file contenenti conflitti vengono segnalati all'utente. Sarà suo compito correggerli e creare il merge commit;

In entrambi i casi il comando `git status` mostra informazioni utili sullo stato della procedura di unione.

`git merge, rebase` | Conflitti

Quando si vanno a unire due branch è possibile che si trovino conflitti tra i vari commit che si stanno provando ad unire. `git` si comporta in modo diverso in base al tipo di unione che si sta svolgendo:

- nel caso di un `merge` tutti i cambiamenti vengono applicati e i file contenenti conflitti vengono segnalati all'utente. Sarà suo compito correggerli e creare il merge commit;
- nel caso di un `rebase` i commit vengono applicati uno ad uno, fermandosi quando si trovano conflitti. L'utente ha il compito di correggerli e far ripartire la procedura di `rebase`, iterando in questo modo fino al completamento.

In entrambi i casi il comando `git status` mostra informazioni utili sullo stato della procedura di unione.

`git merge` | Pregi e difetti

- pregi:
  - preserva *chiaramente* la storia dei commit;
  - può essere usato su branch pubbliche.
- difetti:
  - aggiunge un nuovo commit alla storia;
  - appare come un solo commit sulla branch a cui ci si è uniti[5].

Da usare quando si collabora con altre persone e/o si vuole rendere la storia dei cambiamenti facilmente comprensibile.

[5] I singoli commit della branch unita rimarranno comunque visibili, anche anche dopo la sua rimozione.

`git rebase` | Pregi e difetti

- pregi:
  - non crea commit aggiuntivi, rende la storia lineare;
  - si possono compiere modifiche ai commit precedenti durante l'unione delle due branch.
- difetti:
  - non è possibile collegare i commit riapplicati a quelli originali;
  - **non** può essere usato su branch pubbliche;
  - può sovrascrivere la storia e di conseguenza potrebbe richiedere un `push` forzato.

Torna utile quando si lavora in locale o su branch private, per coprire i propri sbagli o ripulire una storia dei commit poco ordinata.

git fetch

Per ricevere i cambiamenti caricati ad un remoto si può usare il comando:

```
$ git fetch [<remote>]
```

git fetch

Per ricevere i cambiamenti caricati ad un remoto si può usare il comando:

```
$ git fetch [<remote>]
```

Quando viene omesso `<remote>` il comando scarica gli aggiornamenti dal remoto configurato come *upstream* per la branch attuale o dal remoto *origin*.

`git pull`

Quasi sempre si deisdera anche unire la branch attuale con quella remota. Il comando `git pull` esegue `fetch` e in seguito `merge` o `rebase` (in base alla configurazione di `git`) per riconciliare l'albero locale con quello remoto.

`git pull`

Quasi sempre si deisdera anche unire la branch attuale con quella remota. Il comando `git pull` esegue `fetch` e in seguito `merge` o `rebase` (in base alla configurazione di `git`) per riconciliare l'albero locale con quello remoto.
La *best practice* in questo caso è usare la tecnica di rebase, in quanto i cambiamenti in locale che hanno portato ai conflitti sono per definizione *privati* ed è preferibile mantenere la storia dei commit lineare. Si può impostare questa opzione come default tramite:

```
$ git config [--global] pull.rebase true
```

`git stash [push]`

La famiglia di comandi `git stash` è usata per pulire l'albero di lavoro memorizzando i cambiamenti nella cosiddetta *stashing area* per poterli riapplicare in seguito. Quest'area si comporta come una coda LIFO.
Per portare i propri cambiamenti locali nell'area di *stashing*:

```
$ git stash [push]
```

git stash pop, apply

Per riportare l'ultimo set di cambiamenti nella stash nel nostro albero di lavoro si possono usare i comandi:

`$ git stash pop` o `$ git stash apply`

Entrambi riportano le modifiche dell'ultimo stash sull'albero di lavoro, ma il pop rimuove anche la *stash entry* dalla *stashing area*.

`git stash` | Utilizzo

Un tipico utilizzo di `git stash` è quando si vuole effettuare un `rebase`, che richiede l'albero di lavoro pulito. Possiamo allora mettere da parte i cambiamenti locali con un `git stash`, eseguire `git rebase` e riapplicare i nostri cambiamenti con `git stash pop`.

Demo