

# **Comprendere Git Concettualmente**

*traduzione di Marco Ciampa, agosto 2010*

*da "Understanding Git Conceptually"*

*di Charles Duan, 17 Aprile 2010*

<http://www.eecs.harvard.edu/~cduan/technical/git/>

## Indice generale

Introduzione.....	3
Una storia.....	3
Comprendere Git.....	3
Repository.....	4
Contenuti dei repository.....	4
Oggetti commit.....	4
Intestazioni.....	5
Un repository semplice.....	5
Fare riferimento ad un commit specifico.....	6
Fusioni.....	7
Fusioni (Merging).....	7
Risoluzione dei conflitti.....	8
Fusioni veloci in avanti (fast forward merge).....	8
Utilizzi comuni delle fusioni.....	9
Cancellazione di un ramo.....	9
Collaborare.....	11
Al lavoro con altri repository.....	11
Sistema di controllo versione distribuito.....	11
Copia del repository.....	11
Ricezione di cambiamenti dal repository remoto.....	12
Invio di cambiamenti al repository remoto.....	13
Invio (push) e fusioni veloci.....	14
Aggiunta e cancellazione di diramazioni remote.....	15
Git con un repository centrale.....	16
Rebasing.....	17
Rebasing.....	17
Pratiche di uso comune del rebasing.....	18
Andare avanti.....	18
Diramazioni (brancing).....	20
Lo scopo delle diramazioni.....	20
Creazione di una diramazione.....	20
Commutazioni tra diramazioni.....	21
Comandi correlati.....	21
Casi d'uso comuni per le diramazioni.....	22
Note finali alla traduzione.....	23

# Introduzione

Questa è una guida al sistema di controllo versione [Git](#).

Git sta diventando rapidamente uno dei più diffusi sistemi di controllo versione in circolazione. Ci sono moltissime guide su Git. Perché questa guida è differente?

## Una storia

Quando ho cominciato a usare Git, oltre al manuale utente, ho letto moltissime guide. Malgrado avessi compreso la struttura dei comandi di base, sentivo di aver afferrato completamente il senso di ciò che succedeva, per così dire, “sotto il cofano”. Tutto ciò spesso comportava l'apparizione di misteriosi messaggi di errore (almeno così mi sembravano allora) causati dal mio metodo, per tentativi ed errori, alla ricerca del comando, relativamente al contesto, più adatto. Le difficoltà aumentarono non appena le mie necessità cominciarono a crescere di livello, richiedendo comandi più sofisticati (e meno documentati).

Dopo qualche mese, cominciai a capire i concetti sottostanti. Appena ciò successe, all'improvviso, tutto divenne più logico e lineare. Finalmente capivo completamente le pagine del manuale e riuscivo ad eseguire tutti i comandi senza problemi. Tutto quello che all'inizio mi sembrava criptico e oscuro era ora diventato chiaro e limpido.

## Comprendere Git

La conclusione che ho tratto da ciò è che **si può usare Git solo se si comprende appieno come funziona**. Memorizzare semplicemente i comandi da usare quando servono, funzionerà all'inizio, ma è solo questione di tempo prima di ritrovarsi bloccati, o peggio, di rovinare qualcosa.

Metà delle risorse esistenti su Git, sfortunatamente, seguono quest'approccio: vi guidano nell'uso del comando appropriato, e si aspettano che tutto funzioni perfettamente seguendo le istruzioni. L'altra metà attraversano tutti i concetti, ma da quello che ho potuto vedere, spiegano Git in modo da assumere che già si capisca come funziona.

Questa guida, perciò, attuerà un **approccio concettuale** a Git. Il mio obiettivo sarà, prima di tutto, spiegare il mondo di Git ed i suoi obiettivi e, solo in un secondo luogo, l'illustrazione dell'uso dei comandi di Git per la gestione di tale mondo.

Si comincerà col descrivere il modello dati di Git, cioè il repository.

- Da qui si descriveranno le varie operazioni che Git fornisce per l'elaborazione del repository, cominciando dalla più semplice (aggiunta di dati ad un repository) passando progressivamente verso le operazioni più complesse di diramazione (branching) e fusione (merging).
- Poi si discuterà dell'uso di Git in un contesto collaborativo.
- Infine si analizzerà la funzione di rebase di Git, che fornisce un'alternativa al merging, esponendone i pro e i contro.

# Repository

## Contenuti dei repository

Lo scopo di Git (N.d.T.: come ogni sistema di controllo versione) è la gestione di un progetto, o un'insieme di file, che cambiano nel tempo. Git memorizza queste informazioni in una struttura dati chiamata repository.

Un **repository** Git contiene, insieme ad altre cose, i seguenti elementi:

- Un insieme di **oggetti commit**.
- Un insieme di riferimenti a oggetti commit, chiamati **intestazioni** (o **head**).

Il repository Git viene memorizzato nella stessa directory del progetto, in una sottodirectory di nome “.git”. Da notare le differenze con i sistemi di controllo versione a repository centralizzato come CVS o Subversion:

- C'è solo una directory **.git**, nella directory radice del progetto.
- Il repository è memorizzato in file a fianco al progetto. Non c'è un repository del server centrale.

## Oggetti commit

Un **oggetto commit** contiene tre cose:

- Un insieme di **file**, che riflette lo stato di un progetto in un dato punto nel tempo.
- Uno o più riferimenti agli **oggetti commit genitori**.
- Un **nome SHA1**, una stringa a 40-caratteri che identifica univocamente l'oggetto commit. Il nome viene creato componendo un hash (N.d.T: cioè una sorta di checksum) di aspetti rilevanti del commit, per cui commit identici avranno sempre lo stesso nome.

Gli oggetti commit genitori sono quei commit che sono stati modificati per produrre lo stato successivo del progetto. In generale un oggetto commit avrà un commit genitore, dato che generalmente si prende il progetto in un dato stato, si fanno alcuni cambiamenti, e si salva il nuovo stato del progetto. La sezione sottostante sulle fusioni (merge) spiega come un oggetto commit può avere due o più genitori.

Un progetto possiede sempre un oggetto commit senza genitori. Questo è il primo commit eseguito sul repository del progetto.

Basandoci su ciò che si è spiegato poc'anzi, è possibile immaginare un repository come un grafo aciclico diretto di oggetti commit, con puntatori ai commit genitori che puntano sempre indietro nel tempo, fino al primo commit. Partendo da qualsiasi commit, si può camminare lungo l'albero, scorrendo i commit genitori, per vedere la cronistoria dei cambiamenti che portano a quel particolare commit.

L'idea che sta dietro a Git è che il controllo versione consiste nella elaborazione di questo grafo di commit. Ogniqualvolta si voglia eseguire una qualche operazione per interrogare o modificare il repository, si può pensare l'operazione in questi termini: “in che modo voglio interrogare o modificare il grafo dei commit?”.

## Intestazioni

Una **intestazione** (o **head**) è semplicemente un riferimento ad un oggetto commit. Ogni intestazione possiede un nome. Come impostazione predefinita, c'è una intestazione in ogni repository chiamata *master*. Un repository può contenere qualsiasi numero di intestazioni. In ogni istante, c'è una intestazione selezionata come “intestazione corrente”. Questa intestazione ha come alias *HEAD*, sempre scritto in maiuscolo.

Si noti questa differenza: una “head” (minuscolo) si riferisce ad una qualsiasi delle intestazioni con nome presenti nel repository; “*HEAD*” (maiuscolo) si riferisce esclusivamente alla intestazione correntemente attiva. Questa distinzione viene menzionata spesso nella documentazione di Git. In questo testo si userà anche la convenzione di impostare in nomi delle intestazioni, inclusa *HEAD*, in corsivo.

## Un repository semplice

Per creare un repository, create una directory per il progetto, se questa non esiste già, entratevi, ed eseguite il comando `git init`. La directory non deve necessariamente essere vuota.

```
mkdir [progetto]
cd [progetto]
git init
```

Ciò creerà una directory `.git` nella directory `[progetto]`.

Per creare un commit, bisogna fare due cose:

1. Dire a Git quali file includere nel commit, con il comando `git add` (aggiungi). Se un file non è cambiato dal commit precedente (il commit “genitore”), Git lo includerà automaticamente nel commit che si sta effettuando. Perciò sarà necessario aggiungere solo i file che sono stati aggiunti o modificati. Notare che il comando aggiunge ricorsivamente le directory, per cui `git add .` aggiungerà ogni cosa cambiata.
2. Chiamare `git commit` per creare l'oggetto commit. Il nuovo oggetto commit avrà *HEAD* come suo genitore.

Come scorciatoia, `git commit -a` aggiungerà automaticamente tutti i file modificati (ma non quelli nuovi).

Notare che se si modifica un file ma non lo si aggiunge, allora Git includerà le versioni precedenti (prima della modifica) al commit. Il file modificato rimarrà sul posto.

Diciamo che si siano creati tre commit in questo modo. Il repository avrà questo aspetto:

```
----> tempo ---->

(A) <-- (B) <-- (C)
                ^
                |
                master
                ^
                |
                HEAD
```

dove (A), (B), e (C) sono rispettivamente il primo, secondo e terzo commit.

Altri comandi utili a questo punto sono:

- `git log` mostra il registro di tutti i commit cominciando da HEAD sino al commit iniziale (naturalmente può fare molto di più di ciò...).
- `git status` mostra quali file sono cambiati tra lo stato corrente del progetto e HEAD. I file vengono divisi in tre categorie: nuovi file che non sono stati aggiunti (con il comando `git add`), file modificati che non sono stati aggiunti, e file che sono stati aggiunti.
- `git diff` mostra le differenze tra HEAD e lo stato corrente del progetto. Con l'opzione `--cached` compara i file aggiunti con HEAD; altrimenti compara i file non ancora aggiunti.
- `git mv` e `git rm` marciano rispettivamente i file da spostare (o rinominare) e da rimuovere, in modo simile a `git add`.

Il mio lavoro vede spesso la seguente sequenza:

1. Fare un po' di programmazione.
2. `git status` per controllare che file ho cambiato.
3. `git diff [file]` per vedere esattamente che modifiche ho effettuato.
4. `git commit -a -m [messaggio]` per fare il commit.

## Fare riferimento ad un commit specifico

Ora che si sono creati dei commit, come ci si può riferire ad uno specifico commit? Git prevede diversi metodi per fare ciò. Eccone alcuni:

- Dal nome SHA1, che si può ottenere da `git log`.
- Dai primi caratteri del nome SHA1.
- Per intestazione. Per esempio, HEAD si riferisce all'oggetto commit riferito da *HEAD*. Si può usare anche il nome, come per esempio *master*.
- Relativamente a un commit. Ponendo il carattere dell'accento circonflesso (^) dopo il nome del commit fa in modo di recuperare il genitore di quel commit. Per esempio, *HEAD^* è il genitore dell'intestazione del commit corrente.

# Fusioni

## Fusioni (Merging)

Dopo aver finito di implementare una nuova caratteristica in una diramazione del progetto (branch), si potrebbe desiderare di riportare tale caratteristica nel ramo principale (main) del progetto, in modo tale che tutti possano usarla. Ciò può essere eseguito con i comandi `git merge` o `git pull`.

La sintassi per tali comandi è la seguente:

```
git merge [intestazione]
git pull . [intestazione]
```

Il risultato è identico (malgrado il comando con `merge` sembri, per ora, la versione più semplice, la ragione dell'esistenza della versione con `pull` sarà evidente quando si parlerà dell'uso con più sviluppatori in contemporanea).

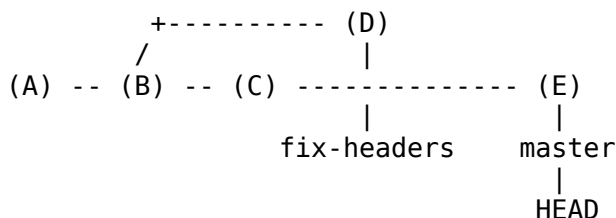
Questi comandi eseguono le seguenti operazioni. Chiamiamo l'intestazione corrente *current*, e quella da fondere *merge*.

1. Identificare l'antenato comune di *current* e *merge*. Chiamiamolo *commit-antenato*.
2. Partendo dalle cose più semplici, se *commit-antenato* è uguale a *merge*, allora non serve fare nulla. Se *commit-antenato* è uguale a *current*, allora è necessario eseguire un **fast forward merge**.
3. Altrimenti, determinare i cambiamenti tra *commit-antenato* e *merge*.
4. Cercare di fondere tali cambiamenti nei file in *current*.
5. Se non ci sono conflitti, creare un nuovo commit, con due genitori, *current* e *merge*. Impostare *current* (e *HEAD*) in modo da puntare a questo nuovo commit, e aggiornare i file di lavoro del progetto in modo appropriato.
6. Se c'era un conflitto, inserire gli appropriati marcatori di conflitto e informare l'utente. Non verrà creato nessun commit.

**Nota importante:** quando si esegue una fusione, Git può confondersi molto, se ci sono cambiamenti di cui non sono stati fatti i commit dei file. Perciò è meglio assicurarsi di fare il commit di ogni cambiamento effettuato **prima** di effettuare una fusione.

Così, per completare l'esempio di cui sopra, diciamo che si voglia portar fuori l'intestazione *master* nuovamente e finire di scrivere i nuovi dati per la propria tesina. Poi si voglia importare tali cambiamenti che si erano effettuati nelle intestazioni (*fix-headers*).

Il repository avrà questo aspetto:

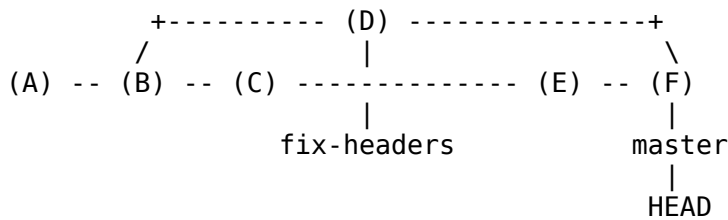


dove (E) è il commit che riflette la versione completa con i nuovi dati.

Si eseguirà:

```
git merge fix-headers
```

Se non ci sono conflitti, il repository ottenuto sarà:



Il commit merge è (F), che ha come genitori (D) e (E). Dato che (B) è l'antenato comune tra (D) e (E), i file in (F) dovrebbero contenere i cambiamenti tra (B) e (D), cioè le correzioni delle intestazioni, incorporate nei file da (E).

**Nota sulla terminologia:** quando si dice “fondere (merge) intestazione A *in* intestazione B,” si intende che intestazione B è la intestazione corrente, e che si sta traendo i cambiamenti dalla intestazione A in essa. Intestazione B viene aggiornata; nulla viene effettuato su intestazione A (se si sostituisce la parola “fusione” con la parola “estrazione” si rende più il senso dell'operazione).

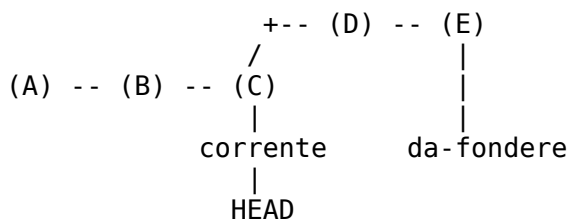
## Risoluzione dei conflitti

Un conflitto si verifica se il commit da fondere ha un cambiamento in esso in una determinata posizione, e il commit corrente possiede un cambiamento nella stessa posizione. Git non ha modo di sapere quale dei due cambiamenti possa avere la precedenza.

Per risolvere il commit, modificare i file per riparare i cambiamenti in conflitto. Poi eseguire `git add` per aggiungere i file con la soluzione, ed eseguire `git commit` per fare il commit della fusione riparata. Git è in grado di ricordare di essere stato interrotto in mezzo ad una fusione, perciò imposta correttamente i genitori del commit.

## Fusioni veloci in avanti (fast forward merge)

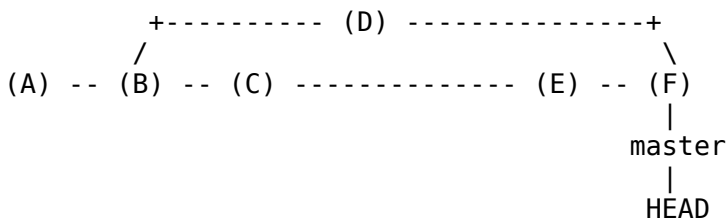
Una fusione veloce in avanti (N.d.T.: la traduzione di questo termine è una libera interpretazione del traduttore. Quando vedrò una versione nazionalizzata in italiano di Git, correggerò di conseguenza) è una semplice ottimizzazione per la fusione. Mettiamo che il repository appaia nel modo seguente:



e si esegua `git merge da-fondere`. In questo caso, tutto ciò che Git necessita di fare è di impostare *corrente* in modo da farlo puntare ad (E). Dato che (C) è il parente comune, non ci sono cambiamenti da “fondere”.

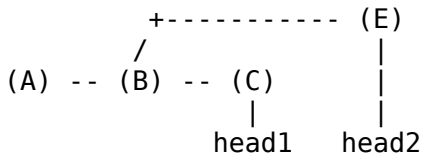
Da cui si deriva che il repository fuso ottenuto appaia in questo modo:





**Nota importante:** `git branch -d` causerà una segnalazione di errore se il ramo da cancellare non è raggiungibile da un'altra intestazione. Perché?

Consideriamo il repository seguente:



Diciamo che si decida di cancellare *head2*. Ora, è possibile usare il commit (E)? Non è possibile farne il check out (N.d.T. ovvero stornarlo), dato che non è un'intestazione. E non appare né nei log né in qualsiasi altra parte, dato che non è un antenato di *head1*. Per cui il commit (E) è praticamente inutile. Nella terminologia Git, è un “dangling commit” (N.d.T. commit pendente) e le sue informazioni sono perse.

Git permette di usare l'opzione `-D` per forzare la cancellazione di un ramo che crei un commit pendente. Comunque, dovrebbero essere rare le situazioni in cui si debba volere effettuare quest'operazione. **Consiglio di ponderare molto attentamente l'uso del comando `git branch -D`.**

# Collaborare

## Al lavoro con altri repository

Una caratteristica chiave di Git è che il repository è memorizzato a fianco alle copie di lavoro dei file sotto controllo. Il vantaggio di ciò consiste nel fatto che il repository memorizza l'intera cronologia del progetto, e che Git può funzionare senza bisogno di collegarsi ad un server esterno.

Ciò significa che, se si vuole gestire il repository, è necessario avere anche l'accesso ai file di lavoro. Di conseguenza, due sviluppatori che usano Git non possono, come impostazione predefinita, condividere lo stesso repository.

Per condividere il lavoro tra più sviluppatori, Git usa un **modello distribuito** di controllo versione. Esso assume che non ci sia un repository centrale. È possibile, naturalmente, usare un repository come “centrale”, ma è necessario comprendere il modello distribuito su cui è basato.

## Sistema di controllo versione distribuito

Poniamo che si voglia lavorare assieme ad un amico su uno stesso documento. L'amico ha già effettuato del lavoro su di esso. Ci sono tre compiti su cui bisogna riflettere perché ciò sia possibile:

1. Come ottenere dall'amico una copia aggiornata del lavoro.
2. Come inserire i cambiamenti effettuati dall'amico nel proprio repository.
3. Come aggiornare l'amico sui propri cambiamenti effettuati.

Git è fornito di un insieme di protocolli di trasporto per la condivisione delle informazioni sul repository, come SSH e HTTP. Il metodo più semplice (che si può usare come test) è comunque accedere simultaneamente ad entrambi i repository dallo stesso filesystem.

Ogni protocollo viene identificato da una “specifica-remota”. Per i repository presenti sullo stesso filesystem, la specifica-remota è semplicemente il percorso necessario per raggiungere l'altro repository.

## Copia del repository

Per fare una copia del repository dell'amico per usarlo personalmente, usare il comando `git clone [specifica-remota]`.

La `[specifica-remota]` identifica la posizione del repository dell'amico, che può anche semplicemente essere un'altra cartella nel filesystem. Qualora il repository dell'amico fosse accessibile tramite un protocollo di rete come ssh, Git si riferisce ad esso tramite un nome stile URL, come il seguente: `ssh://server/repository`.

Per esempio, se il repository dell'amico è posizionato in `~/jdoe/project`, il comando sarà:

```
git clone ~/jdoe/project
```

Ciò effettuerà le operazioni seguenti:

- Creazione di una directory *project* e inizializzazione di un repository in essa.
- Copia di tutti gli oggetti commit e riferimenti alle intestazioni prelevandoli dal progetto al nuovo repository locale.
- Aggiunta di un **riferimento a repository remoto** di nome *origin* nel nuovo repository, e

associazione di *origin* con `~/jdoe/project` come descritto sotto (analogamente a *master*, *origin* è il nome predefinito usato da Git).

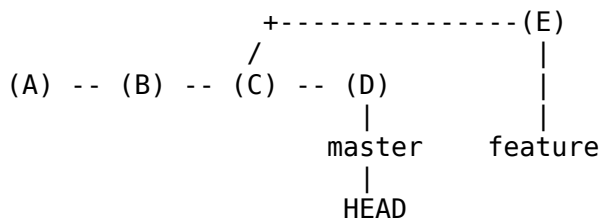
- Aggiunta di **intestazioni remote** di nome *origin/[nome-intestazione]* che corrispondono alle intestazioni nel repository remoto.
- Impostazione di una intestazione nel repository per tenere **traccia** della corrispondente intestazione *origin/[nome-intestazione-corrente]*, cioè quella che era attualmente attiva nel repository clonato.

Un **riferimento repository remoto** è un nome che Git usa per riferirsi al repository remoto. Generalmente esso sarà *origin*. Tra l'altro, Git associa internamente la *specifica-remota* con il riferimento repository remoto, in modo tale da non aver mai bisogno di riferirsi nuovamente al repository originale.

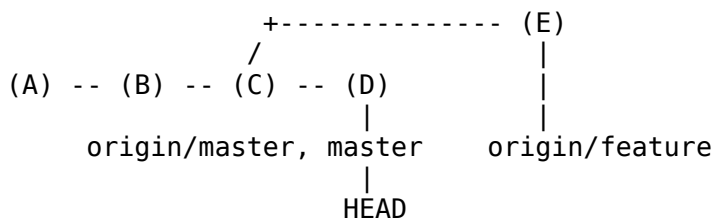
Una diramazione che **traccia** una diramazione remota, mantiene un riferimento interno alla diramazione remota. Questa semplificazione, come descritto sotto, in molte situazioni permette di evitare di battere il nome del repository remoto.

Il fatto importante da notare è che ora si possiede una copia completa dell'intero repository dell'amico. Quando si dirama, commit, fonde, o si opera in altro modo sul repository, si opera solo sul proprio repository. Git interagisce con il repository remoto solamente quando glielo si chiede esplicitamente.

Poniamo che il repository dell'amico abbia la seguente struttura:



Dopo la clonazione, il nostro repository apparirà in questo modo:



Se si desidera lavorare con la diramazione *origin/feature* remota localmente, si deve impostare manualmente una diramazione tracciante. Ciò lo si effettua tramite il comando seguente:

```
git branch --track [nuova-diramazione-locale] [diramazione-remota]
```

Nel nostro esempio, quel comando sarà `git branch --track feature origin/feature`. Si noti che l'opzione `--track` è abilitata in modo predefinito e perciò è ridondante (ma preferisco mantenerla, per chiarezza).

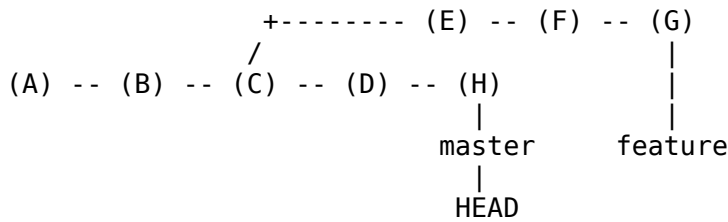
## Ricezione di cambiamenti dal repository remoto

Dopo aver clonato il repository, l'amico aggiunge dei commit al suo repository. Come ottenere le copie di questi cambiamenti?

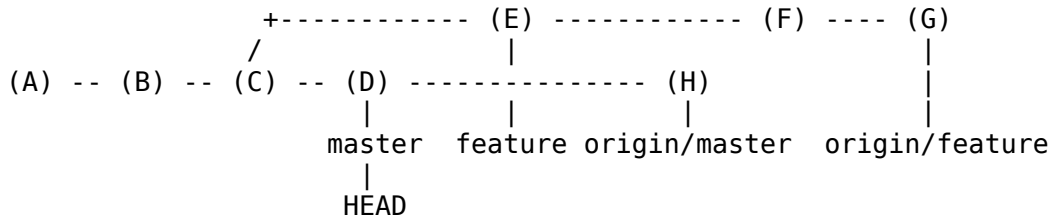
Il comando `git fetch [riferimento-repository-remoto]` recupera i nuovi oggetti

commit del repository dell'amico e crea e/o aggiorna le intestazioni remote di conseguenza. Come impostazione predefinita, il *[riferimento-repository-remoto]* è *origin*.

Poniamo che ora il repository dell'amico appaia in questo modo:



Dopo il comando `git fetch`, il proprio repository dovrebbe apparire nel modo seguente:



Si noti che le proprie intestazioni non sono state modificate. L'unica differenza è che le intestazioni remote, quelle che cominciano con "origin", sono state aggiornate insieme all'aggiunta dei nuovi oggetti commit.

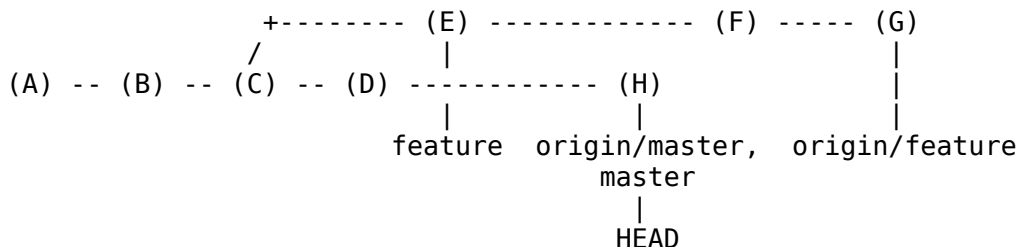
Ora, si vuole aggiornare le proprie intestazioni *master* e *feature* per riflettere i cambiamenti apportati dall'amico. Questo si fa tramite una fusione, di solito materialmente con un comando `git pull`. La sintassi generica è:

```
git pull [riferimento-repository-remoto] [nome-intestazione-remota]
```

Ciò fonderà l'intestazione di nome *[riferimento-repository-remoto]/[nome-intestazione-remota]* in *HEAD*.

Git possiede due caratteristiche che rendono l'operazione più semplice. La prima consiste nel fatto che, se una intestazione viene impostata tracciante, un semplice `git pull` senza argomenti fonderà la corretta intestazione remota. La seconda è che `git pull` eseguirà automaticamente un `fetch`, perciò sarà difficilmente necessario eseguire singolarmente `git fetch`.

Perciò, nell'esempio sopra, effettuato un `git pull` sul proprio repository, la propria intestazione *master* verrà spostata in avanti al commit (H):



## Invio di cambiamenti al repository remoto

Ponendo che si voglia modificare il proprio repository, si vorrà successivamente spedire questi cambiamenti nel repository dell'amico.

Il comando `git push`, che esegue ovviamente (N.d.T. dato che pull significa tirare e push spingere...) l'operazione opposta di `pull`, spedisce i dati al server remoto. La sua sintassi completa è la seguente:

```
git push [riferimento-repository-remoto] [nome-intestazione-remota]
```

Quando il comando viene invocato, Git richiede che il repository remoto faccia le operazioni seguenti:

- Aggiunta di nuovi oggetti commit spediti dal repository inviato.
- Impostazione di `[nome-intestazione-remota]` in maniera da puntare allo stesso commit che punta sul repository inviante.

Sul repository inviante, Git aggiorna anche il riferimento alla corrispondente intestazione remota.

Se non vengono aggiunti argomenti al comando `git push`, esso invierà tutte le diramazioni nel repository impostate come traccianti.

## Invio (push) e fusioni veloci

Git richiede che l'invio (push) provochi una fusione veloce in avanti nel repository remoto.

Ergo, prima della fusione, l'intestazione remota deve puntare ad un antenato del commit a cui dovrà puntare dopo la fusione. Se così non fosse, Git protesterà, e questa protesta dovrà essere presa seriamente in considerazione.

La ragione di ciò è presto detta. Ogni diramazione dovrebbe contenere una versione incrementalmente migliorata del progetto. Una fusione veloce in avanti indica sempre un semplice miglioramento su una diramazione, dato che l'intestazione viene solo spostata in avanti e non ricollegata in un punto differente nell'albero. Se l'intestazione si sposta in un punto differente nell'albero, allora l'informazione su come la diramazione è arrivata al suo stato più recente è persa.

Per esempio, immaginiamo che il repository remoto assomigli al seguente:

```
(A) -- (B) -- (C)
                |
                master
```

Il proprio repository clonato viene modificato fino ad assomigliare a questo:

```

      +----- (D)
     /
(A) -- (B) -- (C)
           |
           other
           |
           master
```

Ora, se si invia `master` al repository remoto, esso prenderà la seguente forma:

```

      +----- (D)
     /
(A) -- (B) -- (C)
           |
           master
```

Ora (C) è un commit pendente. Inoltre, la diramazione `master` ha perso ogni ricordo del fatto che essa puntava a (C). Perciò, (C) non fa più parte della cronistoria di `master`.

A proposito, che succede quando si prova a ricevere i cambiamenti con `pull` dal repository remoto, che non ottengano come risultato una fusione veloce in avanti? Può succedere, ad esempio,



Per elencare le diramazioni remote disponibili, usare `git branch -r`.

## Git con un repository centrale

Se si è compreso come Git trasmette le informazioni tra repository usando `push` e `pull`, allora l'impostazione di un repository centrale per un progetto è semplice. Gli sviluppatori fanno semplicemente il proprio lavoro eseguendo `push` e `pull` verso il repository centrale.

Alla creazione del repository centrale, si può usare un repository “bare”. Esso consiste in un repository che non contiene file di lavoro, solo informazioni del repository. Ciò è utile dato che un repository centrale non dovrebbe avere nessuno che ci lavora sopra direttamente; esso dovrebbe solo ricevere comandi `push` e `pull`.

L'impostazione di un repository Git centrale accessibile tramite un protocollo remoto non è banale. Il modo più semplice per farlo è usare un sistema preesistente come [Gitis](#) o [GitHub](#).

# Rebasing

## Rebasing

Git offre una caratteristica unica chiamata **rebasing** come alternativa alla fusione (merging). La sua sintassi è la seguente:

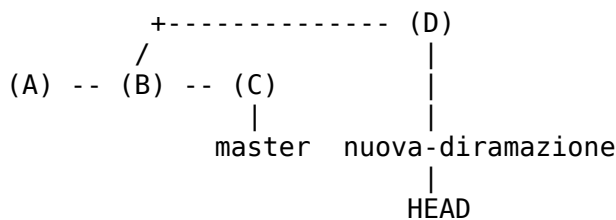
```
git rebase [nuovo-commit]
```

All'invocazione, Git esegue i seguenti passi:

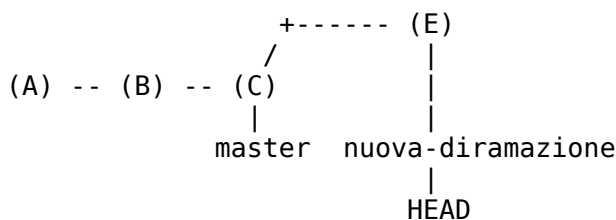
- Identifica ogni commit antenato del commit corrente ma non del [nuovo-commit]. Ciò può essere pensato come un processo a due passi: nel primo, trova l'antenato comune del [nuovo-commit] e del commit corrente; chiamiamolo *commit antenato*. Nel secondo, raccoglie tutti i commit tra il *commit antenato* e il commit corrente.
- Determina cos'è cambiato per ognuno di questi commit, e mette da parte questi cambiamenti.
- Imposta l'intestazione corrente in modo da puntare al [nuovo-commit].
- Per ogni cambiamento messo da parte, riapplica tali cambiamenti nell'intestazione corrente e crea un nuovo commit.

Il risultato è che HEAD, il commit corrente, è un discendente di [nuovo-commit], ma contiene tutti i cambiamenti come se fossero stati fusi con [nuovo-commit].

Per esempio, immaginiamo che un repository assomigli a questo:



Eseguendo `git rebase master` produrrà la situazione seguente:



dove (E) è un nuovo commit che incorpora i cambiamenti tra (B) e (D), ma riorganizzati per porre tali cambiamenti su (C).

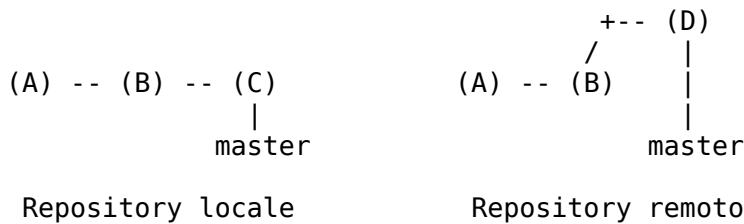
Rebase ha il vantaggio che non c'è la creazione di un commit di fusione. Comunque, dato che HEAD non è un discendente del commit HEAD pre-rebase, il rebasing può essere problematico. Perlomeno per il fatto che una intestazione su cui è stato fatto un rebase non può essere inviata (tramite push) ad un server remoto, dato che essa non è il risultato di una fusione veloce in avanti. Inoltre, esso provoca una perdita di informazioni. Non si può più sapere che (D) era una volta sull'intestazione *nuova-diramazione*. Ciò conduce ad un "cambiamento di cronistoria" che può confondere chi era già a conoscenza del commit (D).

## Pratiche di uso comune del rebasing

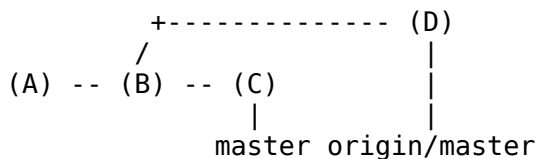
A causa della sovrarmenata pericolosità del rebasing, conviene riservarne l'uso per le seguenti due situazioni.

La prima di queste è quando si sta sviluppando una diramazione per proprio conto non condividendola con nessuno; la si può elaborare con rebase per mantenere la diramazione aggiornata rispetto al ramo principale. Poi, quando alla fine si fonde il proprio ramo di sviluppo nel ramo principale, esso sarà libero da commit di fusione, dato che apparirà come se il ramo di sviluppo fosse un discendente del ramo principale. Inoltre, il ramo principale può spostarsi in avanti con una fusione veloce in avanti, invece che tramite un commit di una fusione normale.

La seconda è quando si esegue un commit di un ramo che cambia contemporaneamente anche in una macchina remota. Si può usare rebase per spostare i propri commit, permettendo di spedire (push) i propri commit nel repository remoto. Dall'esempio di cui sopra:



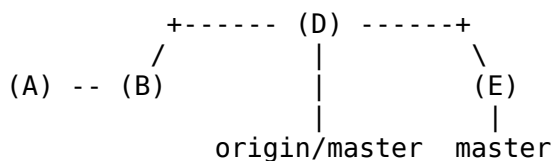
Se si esegue un prelievo tramite `fetch`, il proprio repository somiglierà al seguente:



Ora, assumendo che *master* sia la propria intestazione corrente, se si esegue:

```
git rebase origin/master
```

Il proprio repository assomiglierà a:



Commit (E) contiene i cambiamenti che abbiamo effettuati tra commit (B) e (C), ma ora commit (E) è un discendente di (D). Ciò permette di spedire (push) la propria intestazione *master*, come fusione veloce in avanti, per aggiornare *origin/master*.

## Andare avanti

C'è molto di più in Git. Le pagine del manuale generalmente documentano, in discreto dettaglio, le operazioni possibili.

Se si riesce a comprendere in che modo il repository di Git sia un albero di commit e come operazioni come diramazioni, fusioni, l'invio e il recupero dei dati gestiscano tale albero, allora la comprensione degli altri comandi Git non dovrebbe risultare troppo difficile. Si dovrebbe riuscire a immaginare come ogni comando attraverserà o modificherà l'albero in modo tale da specificare il

giusto commit per far lavorare Git correttamente.  
Perciò, andiamo avanti e lavoriamo sul codice!

# Diramazioni (branching)

## Lo scopo delle diramazioni

Supponiamo si stia lavorando su una tesina. Terminata una prima bozza, questa è stata sottoposta a revisione. In seguito si riescono ad ottenere nuovi dati, e si deve cominciare il processo di integrazione di questi nella tesina. In mezzo a quest'operazione, chiama il comitato di revisione, dicendo che è necessario cambiare alcuni titoli di sezione per conformarsi alle specifiche di formato. Che si fa?

Ovviamente non si vuole spedire i titoli corretti della versione mezza aggiornata. Quello che in genere si vorrebbe fare è saltare indietro alla versione che si è spedita, cambiare i titoli in quella versione, e restituire questa copia, mentre si tiene tutto il lavoro svolto recentemente al sicuro da qualche altra parte.

Questa è l'idea che sta dietro alle diramazioni (**branching**); Git le rende facili da effettuare.

**Nota sulla terminologia:** il termine “diramazione” (“branch”) e “intestazione” (“head”) sono quasi sinonimi in Git. Ogni diramazione è rappresentata da un'intestazione, ed ogni intestazione rappresenta una diramazione. Alle volte, il termine “diramazione” sarà usato per riferirsi ad una intestazione e all'intera cronologia di commit antenati che precedono quella intestazione, dove “intestazione” verrà usata esclusivamente per riferirsi ad un singolo oggetto commit, il commit più recente della diramazione.

## Creazione di una diramazione

Per creare una diramazione, poniamo ad esempio che il nostro repository somigli al seguente:

```
(A) -- (B) -- (C)
                |
                master
                |
                HEAD
```

dove (B) è la versione che si è spedita alla conferenza e (C) è il nuovo stato della nostra revisione (abbandoniamo le frecce considerando che queste puntano sempre a sinistra).

Per saltare indietro al commit (B) e cominciare un nuovo lavoro da lì, è per prima cosa necessario sapere come riferirsi al commit. Si può usare sia il comando `git log` per ottenere il nome SHA1 di (B), che usare `HEAD^` per recuperarlo.

Ora si userà il comando `git branch`:

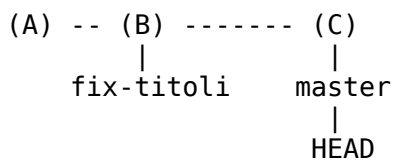
```
git branch [nome-nuova-intestazione] [riferimento-a-(B)]
```

o, per esempio:

```
git branch fix-titoli HEAD^
```

Questo comando creerà una nuova intestazione con il nome dato, e punterà tale intestazione all'oggetto commit richiesto. Se l'oggetto commit viene tralasciato, essa punterà ad `HEAD`.

Ora il nostro albero dei commit somiglierà al seguente:



## Commutazioni tra diramazioni

Per cominciare a lavorare sulle intestazioni, è necessario impostare l'intestazione *fix-titoli* come intestazione corrente. Ciò è possibile con il comando `git checkout`:

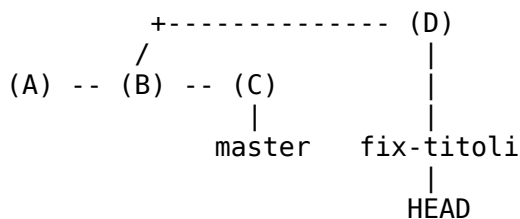
```
git checkout [nome-intestazione]
```

Questo comando esegue le operazioni seguenti:

- Fa puntare *HEAD* all'oggetto commit specificato da *[nome-intestazione]*
- Riscrive tutti i file nella cartella per farli corrispondere ai file memorizzati nel nuovo commit *HEAD*.

**Nota importante:** se ci sono cambiamenti non inseriti nel commit quando si esegue `git checkout`, Git si comporterà in maniera molto strana. Il comportamento è predicibile e qualche volta utile, ma è meglio evitarlo. Per fare ciò basta fare il commit di tutti i nuovi cambiamenti prima di fare il checkout della nuova intestazione.

Dopo aver fatto il checkout dell'intestazione *fix-titoli*, si aggiustano i titoli. Ora si aggiunge e si esegue il commit dei cambiamenti come sopra. Il repository risultante apparirà come di seguito:



(Ora si può comprendere perché l'operazione è chiamata "diramazione": l'albero dei commit è cresciuto di un nuovo ramo. Notare che l'angolo della linea che connette (B) a (D) è irrilevante; i puntatori non memorizzano il fatto di essere obliqui o orizzontali.)

L'ascendenza di *master* è (C), (B) e (A). L'ascendenza di *fix-titoli* è (D), (B) e (A). Ciò può essere osservato tramite il comando `git log`.

## Comandi correlati

A questo punto è conveniente menzionare altri comandi che possono tornare utili:

- `git branch` senza argomenti elenca le intestazioni esistenti, marcando con un asterisco l'intestazione corrente.
- `git diff [intestazione1]..[intestazione2]` mostra le differenze tra i commit riferiti da *intestazione2* e *intestazione1*.
- `git diff [intestazione1]...[intestazione2]` (con tre punti) mostra le differenze tra *intestazione2* e l'antenato comune di *intestazione1* e *intestazione2*. Per

esempio, eseguendo `diff master...fix-titoli` nell'esempio riportato sopra avrebbe mostrato le differenze tra (D) e (B).

- `git log [intestazione1]..[intestazione2]` mostra la registrazione dei cambiamenti (N.d.T.: ovvero il changelog) tra *intestazione2* e l'antenato comune di *intestazione1* e *intestazione2*. Con tre punti, mostra anch'esso i cambiamenti tra *intestazione1* e l'antenato comune; ciò non è poi così utile (ma passare da *intestazione1* e *intestazione2*, d'altra parte, è molto utile).

## Casi d'uso comuni per le diramazioni

Un modo comune di usare le diramazioni di Git è per mantenere un ramo “principale” e creare nuove diramazioni per implementare nuove caratteristiche. Spesso il ramo predefinito di Git, *master*, viene usato come ramo principale.

Perciò, nell'esempio precedente poteva convenire lasciare *master* a (B), nel quale il lavoro era stato sottoposto ai revisori. Poi si sarebbe potuto cominciare un nuovo ramo per memorizzare i cambiamenti concernenti i nuovi dati.

Idealmente, in questo contesto, **il ramo *master* è sempre in uno stato rilasciabile**. Gli altri rami conterranno il lavoro mezzo finito, nuove caratteristiche, e così via.

Questo schema è particolarmente importante quando ci sono più sviluppatori al lavoro su un singolo progetto. Se tutti gli sviluppatori aggiungono commit in sequenza su di un singolo ramo, allora le nuove caratteristiche devono essere aggiunte in un singolo commit, in maniera da non rendere il ramo inusabile. Comunque, se ogni sviluppatore crea una nuova diramazione per creare una nuova caratteristica, allora i commit possono essere eseguiti in qualsiasi momento, completi o no.

Questo è il significato che intendono gli utenti Git quando dicono che i **commits “costano poco” (are cheap)**. Se si sta lavorando sul proprio ramo, non c'è ragione di essere particolarmente attenti su cosa si inserisce nei commit del repository, tanto non influenzerà nessun altro.

# Licenza

Per la licenza uso direttamente le parole dell'autore, come me le ha scritte via email:

“I haven't actually thought about what license to publish it under; I have to go read them particularly. For this, though, I'll just grant you a one-off nonexclusive license to make an Italian translation of the tutorial and distribute it for noncommercial purposes, so long as you keep my name and a link to the original on the distributed copies.”

che tradotte significano:

“Non ho ancora pensato a che tipo di licenza usare per la pubblicazione; devo studiarla bene. Per questa volta concedo semplicemente una licenza singola non-esclusiva per fare una traduzione in italiano del tutorial e per distribuirlo per scopi non commerciali, fintantoché tu mantenga il mio nome e il collegamento al documento originale nelle copie distribuite.”

## Note finali alla traduzione

Il lettore che conosce l'inglese troverà che la traduzione non è precisamente identica all'originale. Questo perché sono convinto che se si vuole rendere un testo tradotto scorrevole, è necessario adeguarsi al modo di esprimersi proprio della lingua obiettivo. Spero che il risultato mi dia ragione.

Ho scritto questa traduzione nella speranza che possa servire a qualcuno oltre che a me.

Si, avete capito bene, ho scritto che serve anche a me.

In effetti ho il brutto limite di avere la sensazione (non so se ciò reale o è solo un'impressione) di capire bene qualcosa solo se lo leggo nella mia lingua nativa. Quindi quando voglio essere sicuro di aver afferrato bene un concetto di qualcosa scritto in inglese, come prima cosa, lo traduco in italiano e *poi* lo leggo. Qualcuno dirà: “ma lo hai già letto!”. È vero, l'ho letto, ma non l'ho capito. Solo quando lo rileggo in italiano, lo capisco veramente. Probabilmente non ho interiorizzato sufficientemente l'inglese. Perciò mi sembrava uno spreco di energie tradurre solo per me, quindi ho deciso di tradurre bene e ripubblicare. Spero che qualcuno apprezzi il mio difetto (e magari lo faccia segnalandomi le inesattezze ;-).

Questo testo è stato scritto con OpenOffice.org.

Marco Ciampa  
ciampix su libero punto it