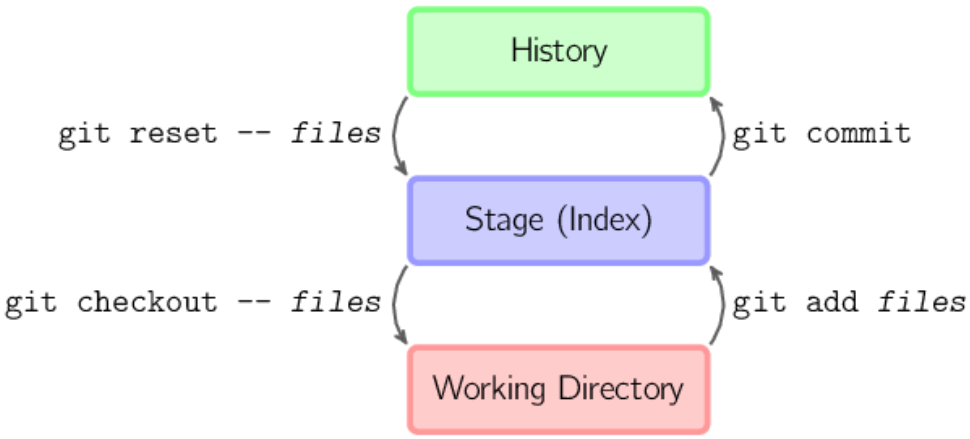
Guida visuale a Git

Immagini SVG disabilitate. (Riattiva SVG)

Questa pagina offre una breve guida visuale per i comandi più comuni di Git. Una volta comprese le basi di Git, questo sito potrà aiutarti a fissare questi concetti. Se ti interessa conoscere come è stato creato questo sito visita il mio repository GitHub.

Contenuti

Utilizzo di Base

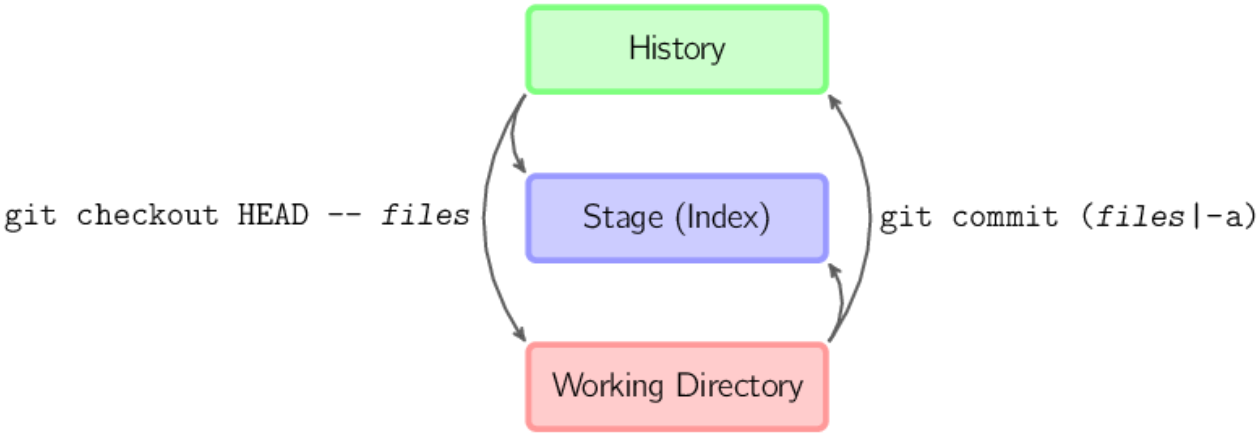


I quattro comandi precedenti copiano i file tra la working directory, lo stage (chiamato anche index) e l'history (rappresentata dai commit).

- `git add file` copia i *file* (nel loro stato corrente) nello stage.
- `git commit` crea uno snapshot dello stage sotto forma di commit.
- `git reset -- file` rimuove i file dallo stage; ovvero copia i *file* dell'ultimo commit nello stage. Utilizza questo comando per annullare un `git add file`. Puoi anche utilizzare `git reset` per rimuovere tutto dallo stage.
- `git checkout -- file` copia i *file* dallo stage alla working directory. Utilizza questo comando per eliminare tutte le modifiche locali.

Puoi usare `git reset -p`, `git checkout -p`, oppure `git add -p` al posto di (o in aggiunta di) specificare i file specifici, per determinare in modo interattivo, che blocchi copiare.

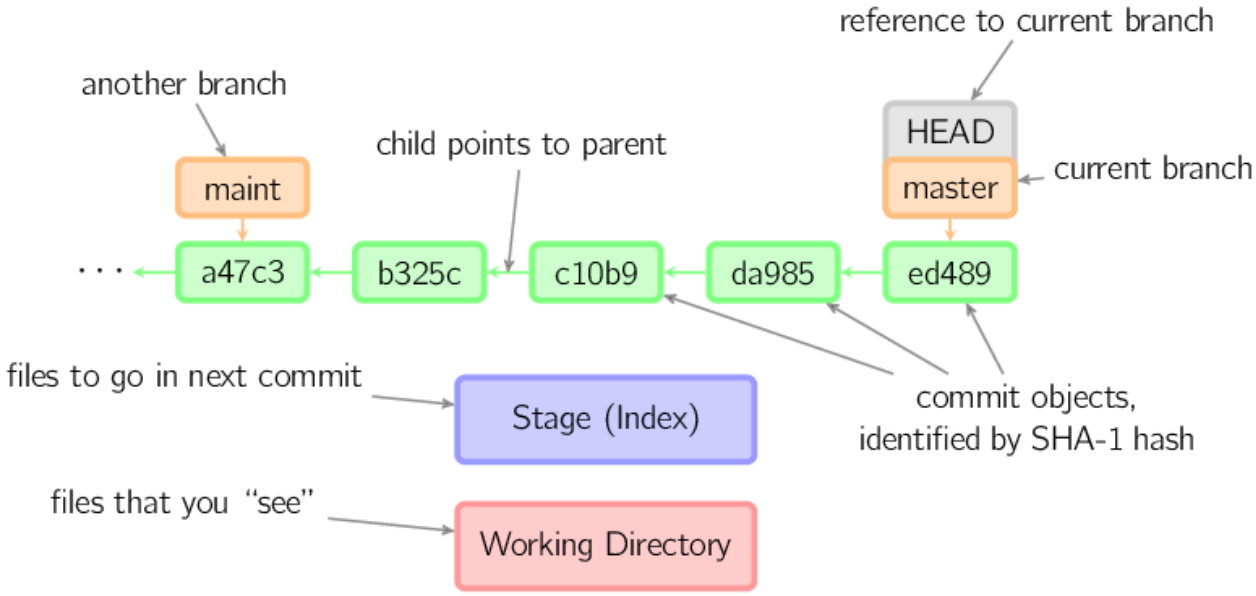
È anche possibile evitare lo stage ed eseguire il check out di file direttamente dall'history o eseguire il commit senza inserire prima i file nello stage.



- `git commit -a` è equivalente ad eseguire `git add` su tutti i file esistenti nell'ultimo commit, seguito da `git commit`.
- `git commit file` crea un nuovo commit contenente i contenuti dell'ultimo commit oltre allo snapshot dei *file* presi dalla working directory. Inoltre i *file* vengono copiati nello stage.
- `git checkout HEAD -- file` copia i *file* dall'ultimo commit sia allo stage che alla working directory.

Convenzioni

Nel resto di questo documento useremo grafici come quello seguente.

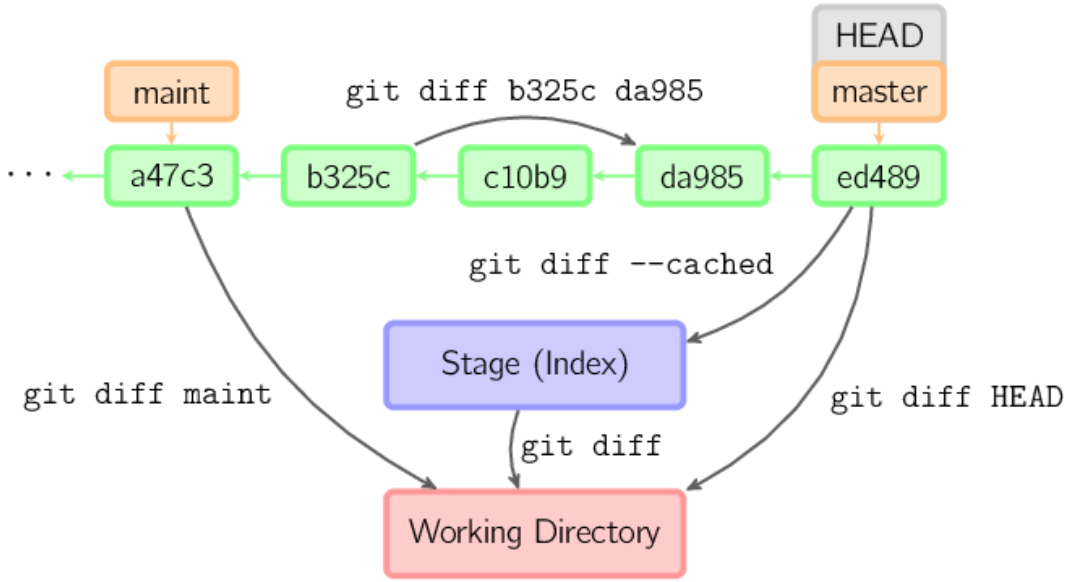


I commit sono mostrati in verde con ID da 5 caratteri, essi puntano al loro genitore. I branch sono mostrati in arancione, essi puntano a specifici commit. Il branch corrente è identificato dalla speciale referenza *HEAD*, che è "attaccata" a quel branch. In questa immagine vengono mostrati gli ultimi 5 commit, il più recente è il *ed489*. *master* (il branch corrente) punta a questo commit, mentre *maint* (un altro branch) punta ad un antenato del commit del *master*.

Comandi nei Dettagli

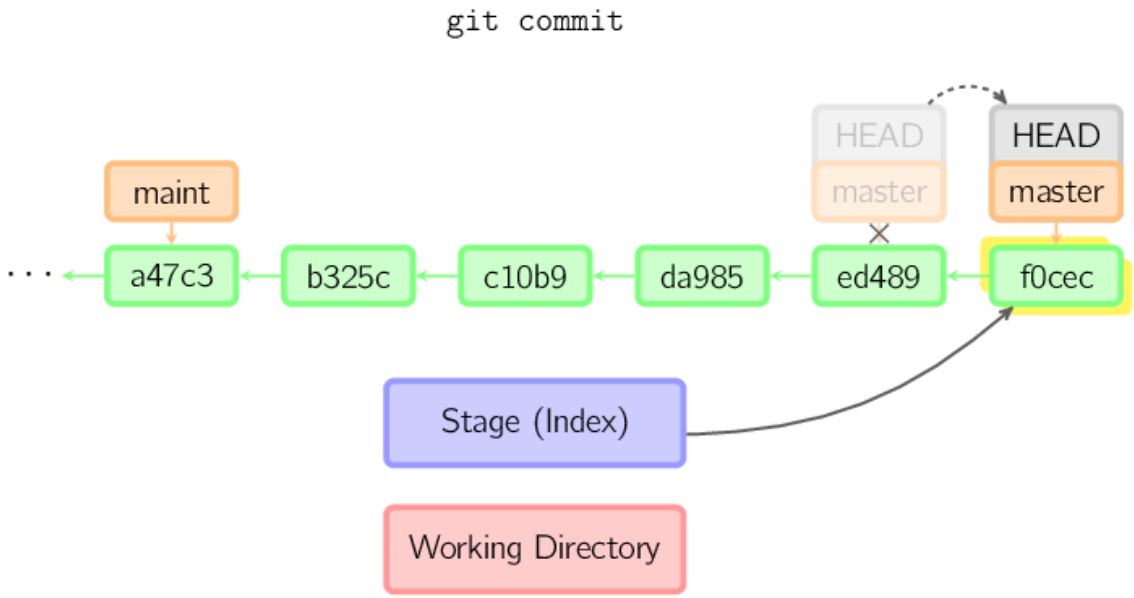
Diff

Esistono diversi modi per verificare le differenze tra i commit. Di seguito alcuni esempi comuni. Ognuno di questi comandi può opzionalmente ricevere nomi di file come ulteriori parametri per limitare le differenze solo ai file specificati.

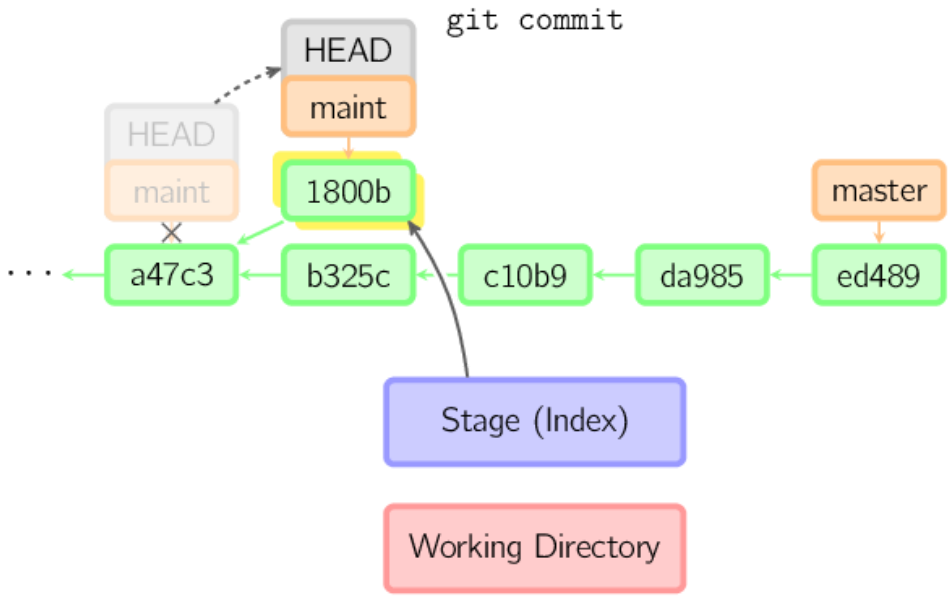


Commit

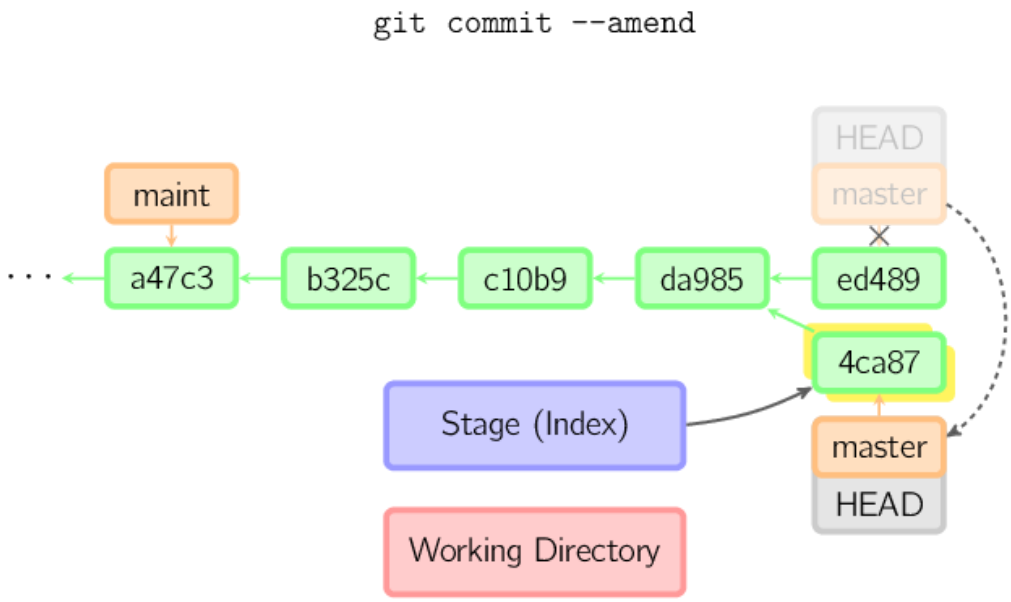
Quando si esegue un commit git crea un nuovo oggetto commit utilizzando i file dello stage ed impostando il genitore al commit attuale. Successivamente punta il branch corrente a questo nuovo commit. Nell'immagine seguente il branch corrente è *master*. Prima dell'esecuzione del comando *master* puntava a *ed489*. In seguito, un nuovo commit, *f0cec*, è stato creato con genitore *ed489*, quindi *master* è stato spostato sul nuovo commit.



Lo stesso processo avviene anche quando il branch corrente è un antenato di un altro. Di seguito vediamo un commit sul branch *maint*, che è un antenato di *master*, risultante in *1800b*. Di conseguenza, *maint* non è più un antenato di *master*. Per unire le due history sarà necessario un merge (o un rebase).



A volte può capitare di commettere un errore in un commit, questo è facile da correggere con `git commit --amend`. Quando si utilizza questo comando git crea un nuovo commit con lo stesso genitore del commit corrente. (Il vecchio commit verrà scartato se non ha referenze da nessuno.)

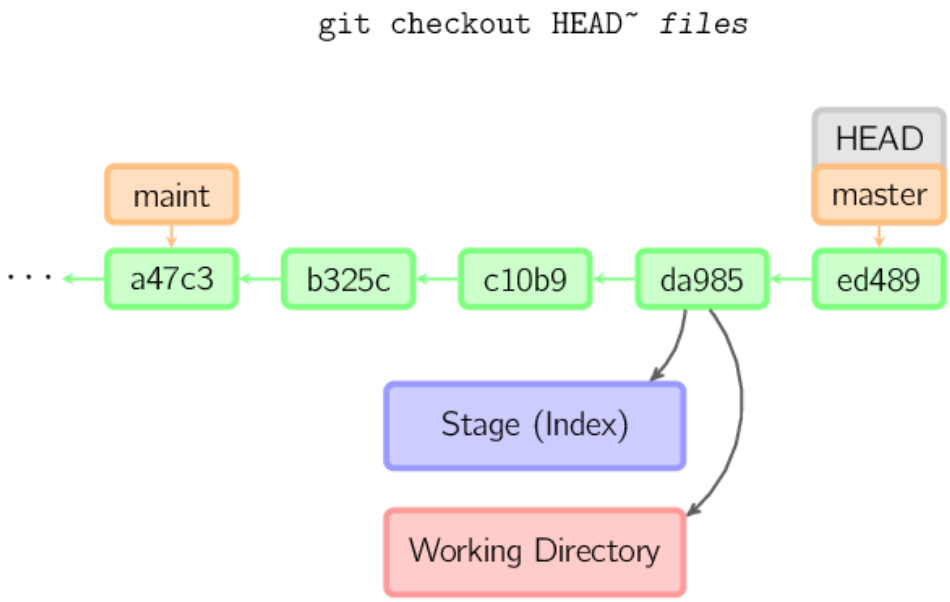


Un quarto caso è rappresentato dal commit con HEAD isolata, come verrà spiegato in seguito.

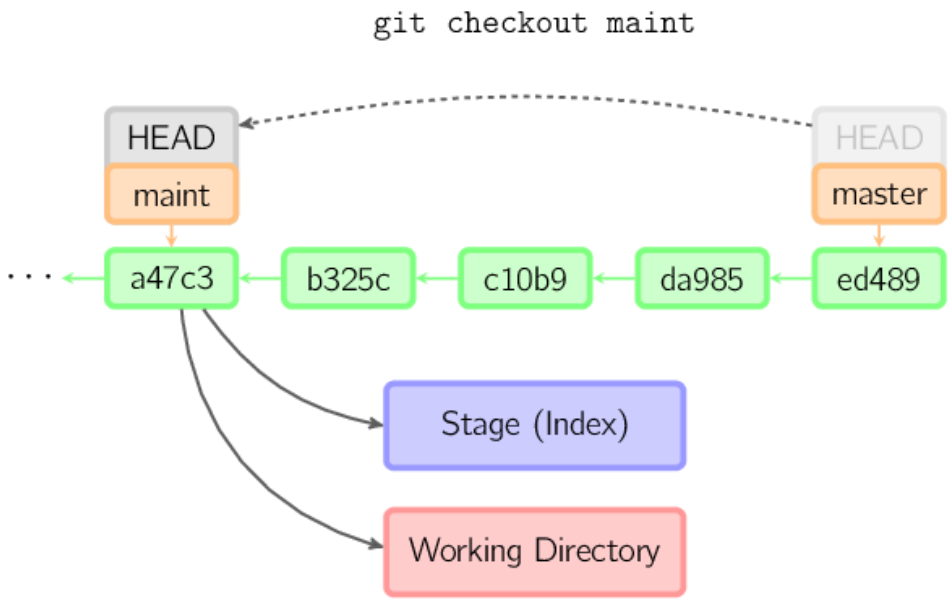
Checkout

Il comando checkout viene utilizzato per copiare i file dall'history (o dallo stage) alla working directory, opzionalmente per scambiare i branch.

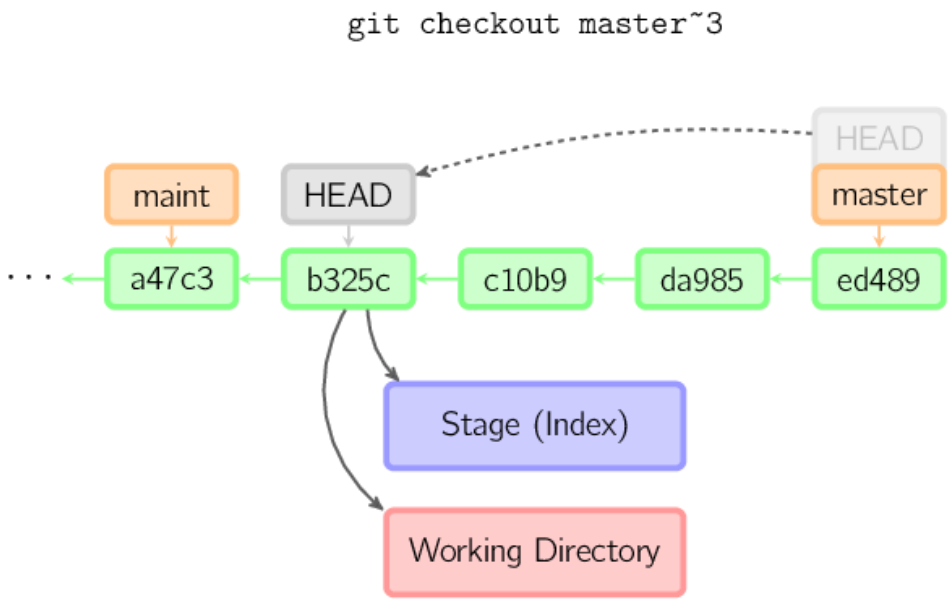
Quando un file (e/o `-p`) viene passato, git copia questi file da un certo commit allo stage ed alla working directory. Per esempio, `git checkout HEAD~ foo.c` copia il file `foo.c` dal commit chiamato *HEAD~* (il genitore del commit corrente) alla working directory, oltre ad inserirlo nello stage. (Se non viene specificato un commit, i file vengono copiati dallo stage.) Si noti che il branch corrente non è cambiato.



Quando il nome di un file *non* viene passato ma la referenza è un branch (locale), *HEAD* viene associata a quel branch (ovvero si fa lo "switch a" quel branch), in seguito lo stage e la working directory vengono impostate per rispecchiare i contenuti di quel commit. Ogni file esistente nel nuovo commit (*a47c3* qui sotto) viene copiato; ogni file esistente nel vecchio commit (*ed489*) ma non nel nuovo viene eliminato; ogni file esistente in entrambi viene ignorato.

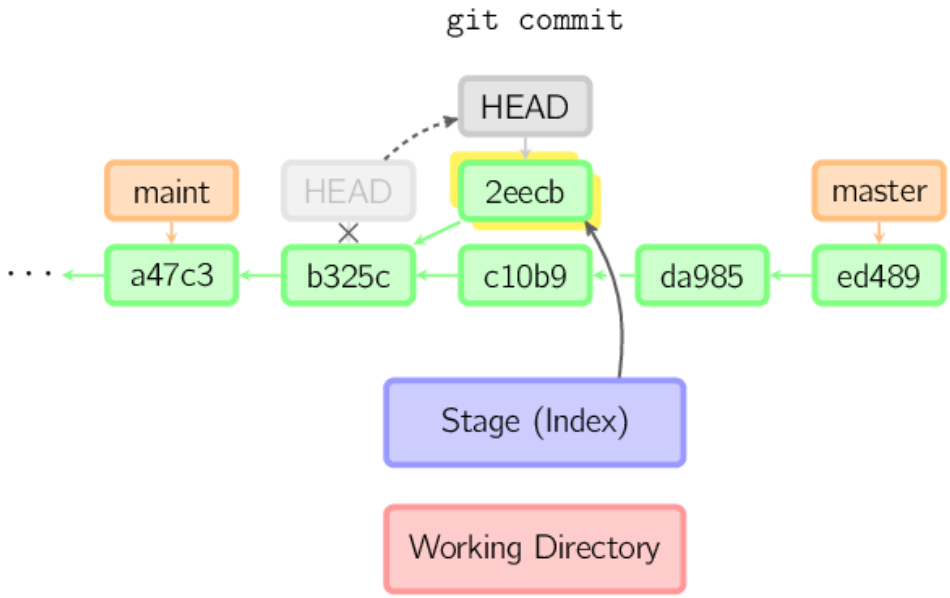


Quando il nome di un file *non* viene passato e la referenza *non* è ad un branch (locale) — si ipotizzi sia un tag, un branch remoto, un ID SHA-1 o qualcosa tipo *master~3* — abbiamo a che fare con un branch anonimo chiamato *HEAD isolata*. Questo è utile per muoversi liberamente nell'history. Supponiamo di voler compilare la versione 1.6.6.1 di git. Si può ricorrere a `git checkout v1.6.6.1` (che è un tag, non un branch), compilare, installare, e successivamente tornare ad un altro branch, per esempio `git checkout master`. Tuttavia eseguire dei commit con HEAD isolata richiede una procedura leggermente diversa; ne parliamo di seguito.

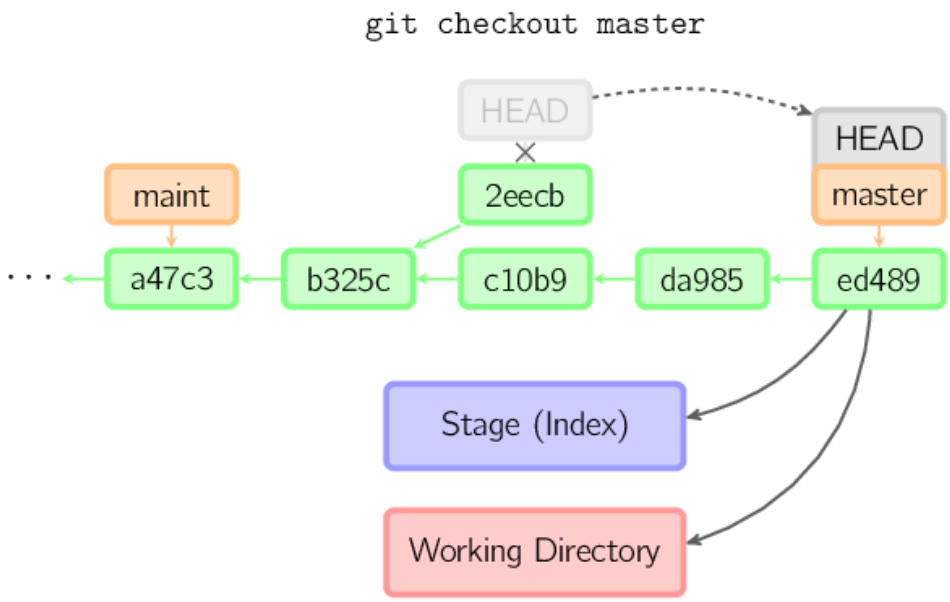


Commit con HEAD isolata

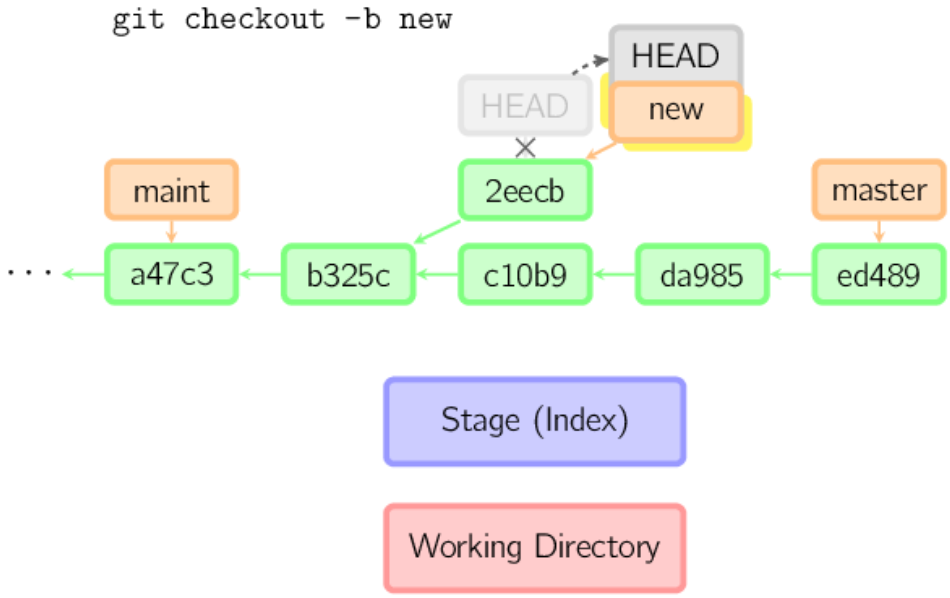
Quando *HEAD* è isolata, i commit funzionano normalmente, l'unica eccezione è data dal fatto che il branch senza nome viene aggiornato. (Si può pensare ad esso come ad un branch anonimo.)



Una volto eseguito il checkout di qualcos'altro, per esempio di *master*, il commit (presumibilmente) non essendo più referenziato da nessun altro viene perso. Si noti come dopo il comando non ci sia più alcuna referenza a *2eecb*.



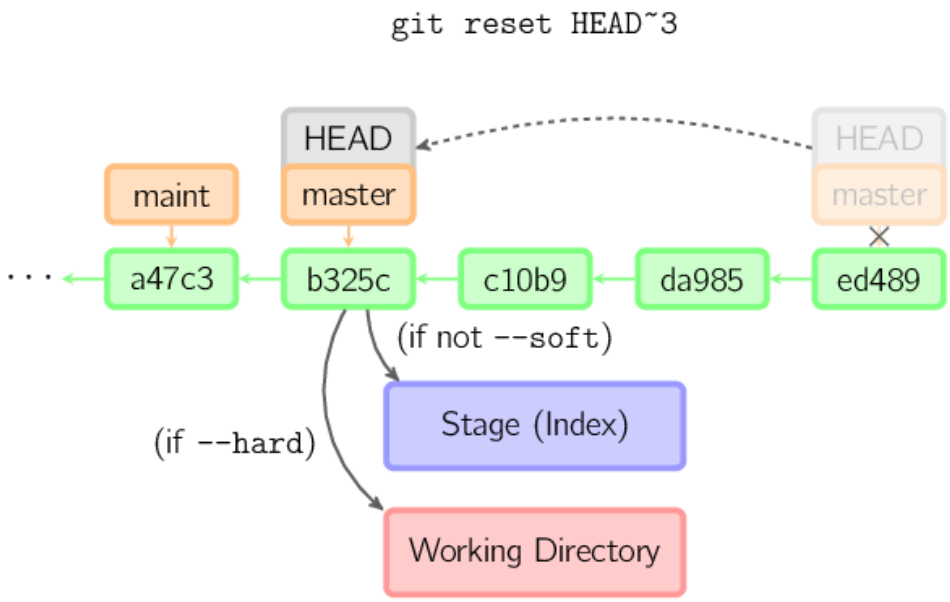
Se, d'altronde, si volesse memorizzare questo stato, sarebbe necessario creare un nuovo branch con nome utilizzando `git checkout -b name`.



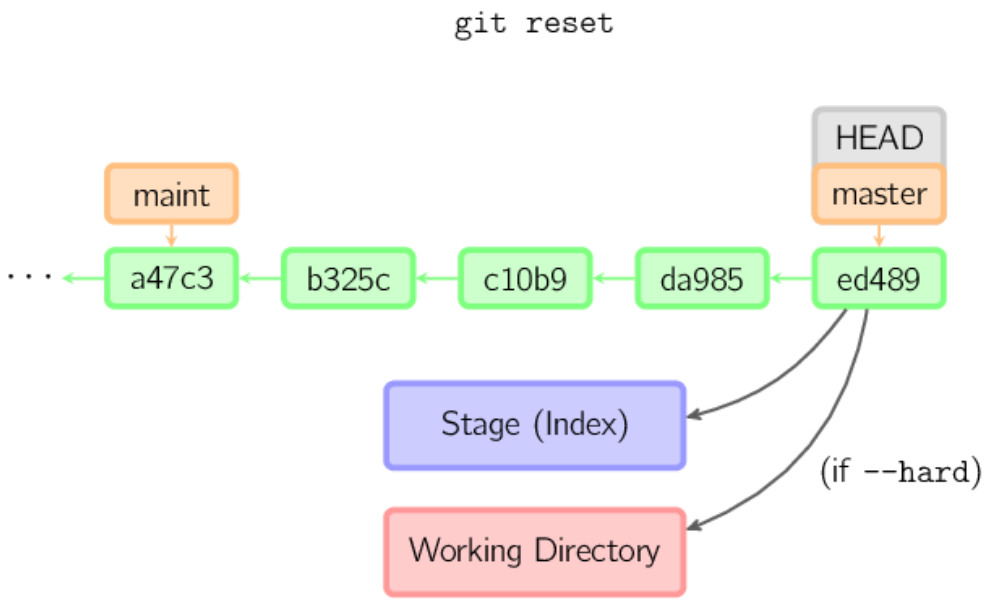
Reset

Il comando reset sposta il branch corrente in un'altra posizione ed opzionalmente aggiorna lo stage e la working directory. Viene anche utilizzato per copiare file dall'history allo stage senza tirare in ballo la working directory.

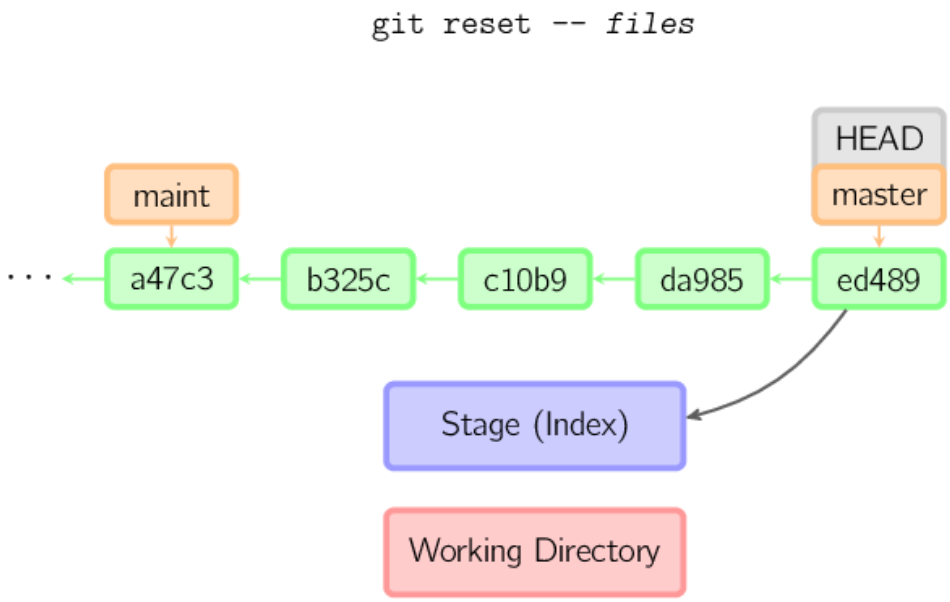
Se un commit viene passato senza nomi di file il branch corrente viene spostato a quel commit e quindi lo stage viene aggiornato per rispecchiare quel commit. Se viene utilizzato `--hard` viene aggiornata anche la working directory. Se viene passato `--soft` non viene aggiornato nulla.



Se non viene passato un commit il default utilizzato è *HEAD*. In questo caso il branch non viene spostato, ma lo stage (ed opzionalmente anche la working directory se viene passato `--hard`) vengono resettati ai contenuti dell'ultimo commit.

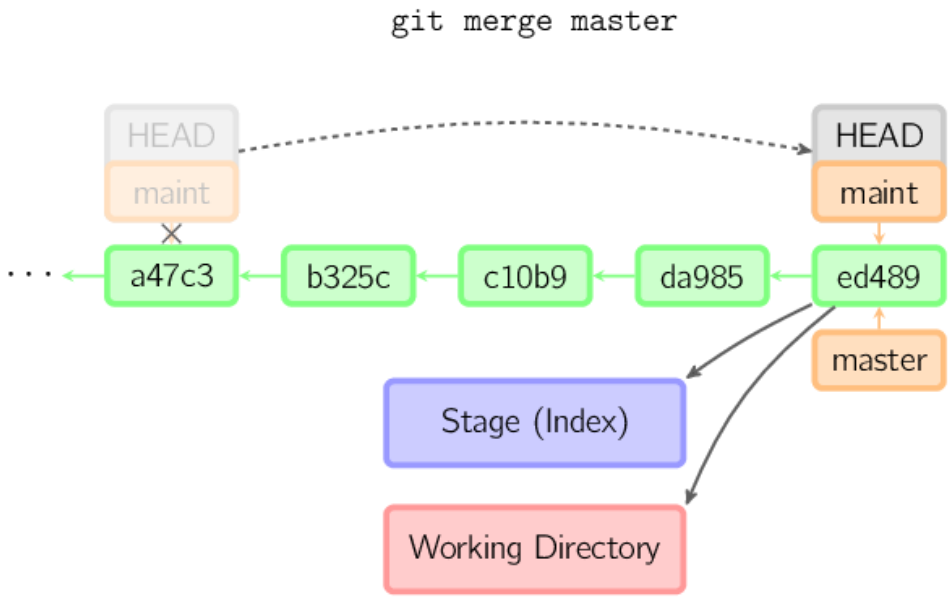


Se viene passato un nome di file (e/o `-p`) allora il comando si comporta in modo simile a checkout con un nome di file, l'unica differenza è che solo lo stage viene aggiornato (e non la working directory). (È anche possibile specificare il commit dal quale prendere i file al posto di *HEAD*.)

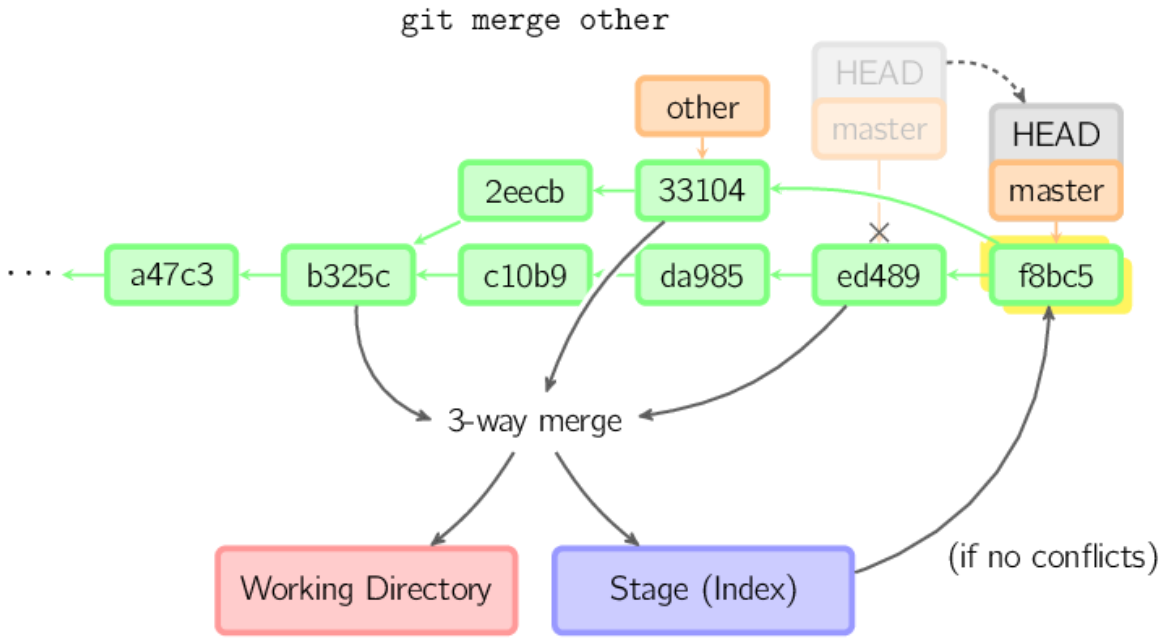


Merge

Merge crea un nuovo commit che include modifiche provenienti da altri commit. Prima di eseguire il merge lo stage deve corrispondere al commit corrente. Il caso banale è quello in cui l'altro commit è un antenato del commit corrente, non serve fare niente. Un altro caso semplice è quello in cui il commit corrente è un antenato dell'altro commit. Avremo quindi un merge *fast-forward*. La referenza viene semplicemente spostata e viene eseguito il checkout sul nuovo commit.

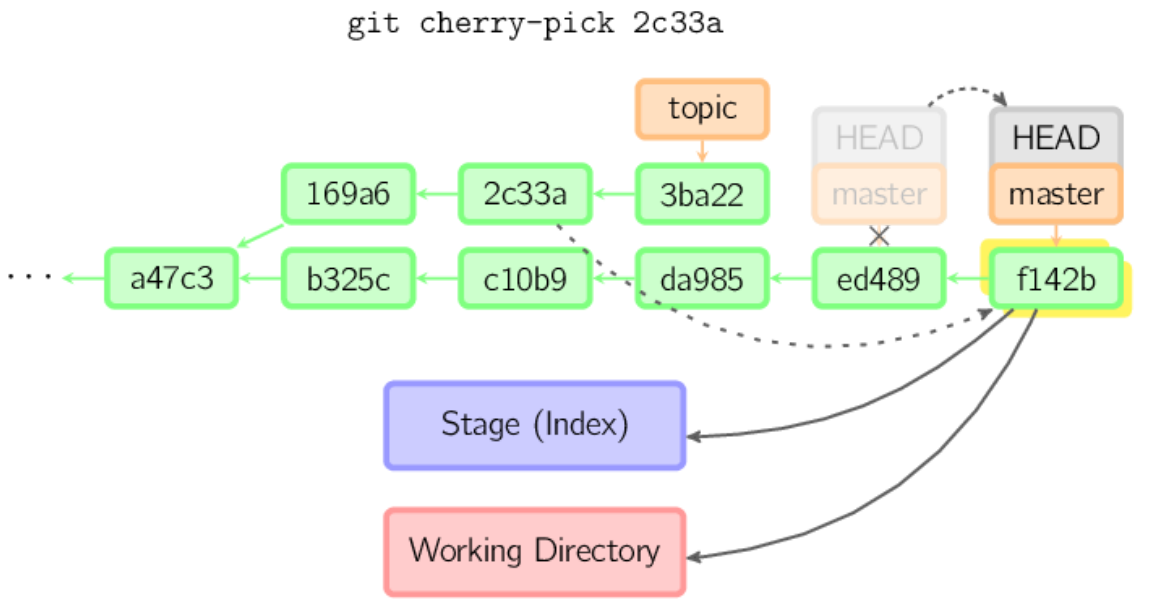


Negli altri casi avremo un merge "reale". Si possono scegliere altre strategie ma quella di default è rappresentata dal merge "ricorsivo" che semplicemente prende il commit corrente (*ed489* below), l'altro commit (*33104*), il loro antenato comune (*b325c*) ed esegue un merge a tre vie. Il risultato viene memorizzato nella working directory e nello stage, seguirà un commit con un genitore extra (*33104*).



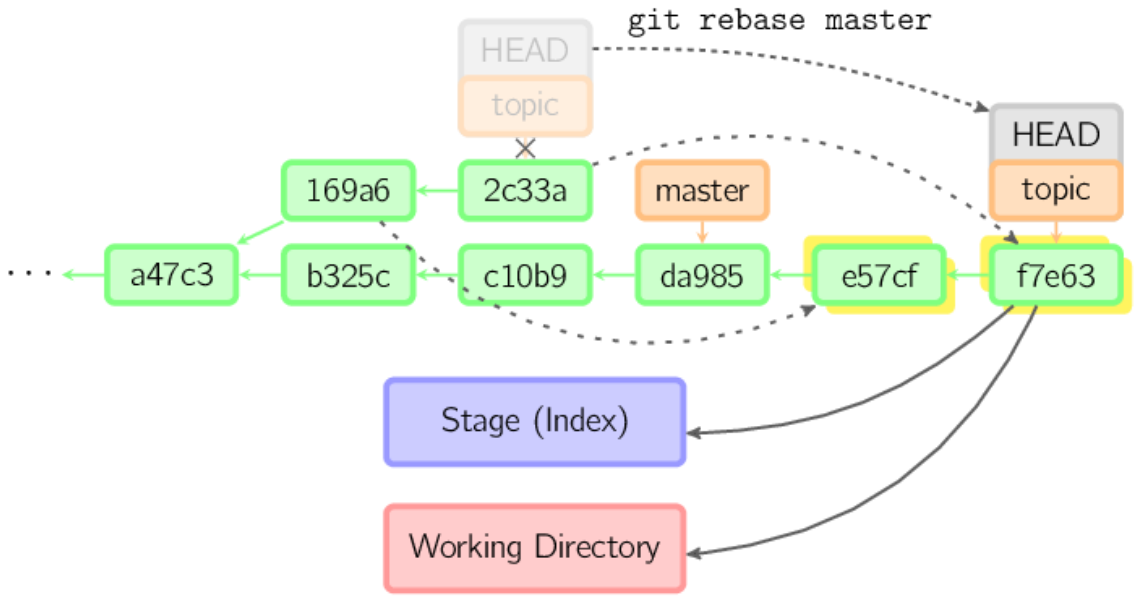
Cherry Pick

Il comando cherry-pick "copia" un commit creandone uno nuovo nel branch corrente con lo stesso messaggio applicando le modifiche come se fosse un commit diverso.



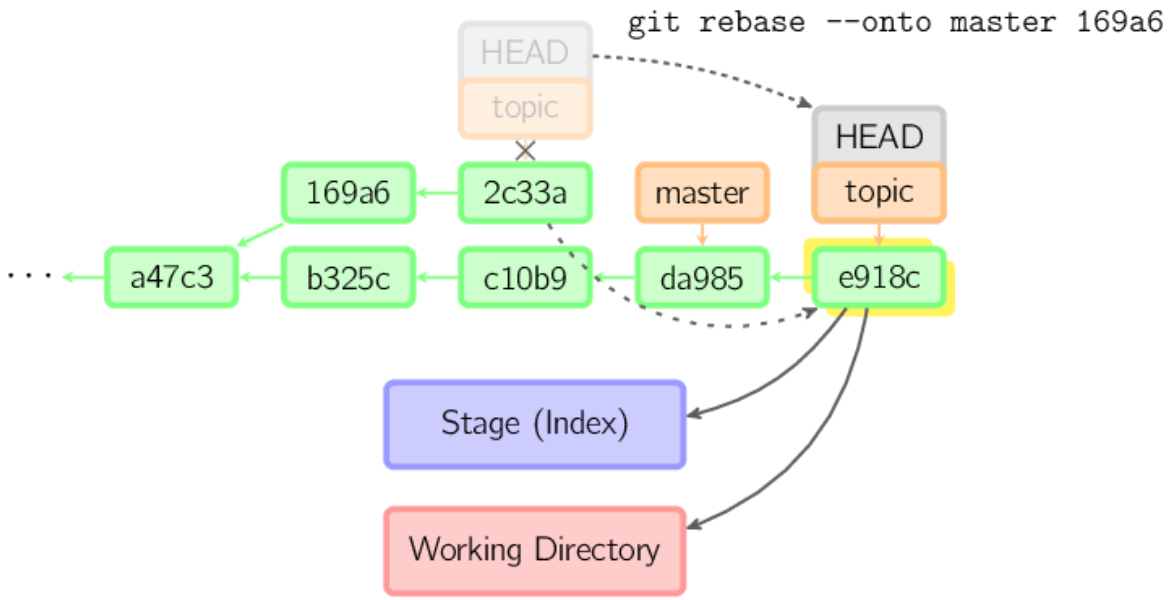
Rebase

Rebase è un'alternativa a merge per combinare assieme più branch. Mentre un merge crea un singolo commit con due genitori, preservando l'history non lineare, un rebase riporta i commit dal branch corrente su un altro producendo un history lineare. In sostanza è un modo automatizzato per eseguire diversi cherry-pickin sequenza.



Il comando precedente prende tutti i commit esistenti in *topic* ma non in *master* (precisamente *169a6* e *2c33a*), li riporta sul *master* e poi sposta l'head del branch sulla nuova estremità.

Per limitare quanto andare all'indietro utilizzare l'opzione `--onto`. Il comando seguente riporta sul *master* i commit più recenti del branch corrente da *169a6* (escluso), precisamente *2c33a*.



Esiste anche `git rebase --interactive`, che permette di fare cose più complicate rispetto a riportare semplicemente commit, più precisamente eliminare, riordinare, modificare i commit. Non esiste un caso evidente per un'immagine per questo; vedere git-rebase(1) per maggiori dettagli.

Technical Notes

I contenuti dei file non sono memorizzati nell'index (*.git/index*) o negli oggetti dei commit. Ogni file è memorizzato nel database degli oggetti (*.git/objects*) come *blob*, identificato dal suo hash SHA-1. Il file index elenca i nomi dei file assieme all'identificatore del blob associato oltre ad altri dati. Per i commit esiste un ulteriore tipo di dato, *tree*, anch'esso identificato da un suo hash. I tree corrispondono alle directory presenti nella working directory e contengono una lista di tree e blob corrispondenti ad ogni nome di file in quella directory. Ogni commit memorizza l'identificatore del suo tree principale che a sua volta contiene tutti i blob e gli altri tree associati a quel commit.

Se viene eseguito un commit utilizzando un'HEAD isolata l'ultimo commit viene referenziato da qualcosa: dal reflog per HEAD. Tuttavia questo scompare in pochi istanti, quindi il commit verrà gestito dal garbage collector come succede per i commit scartati con `git commit --amend` o `git rebase`.
