Otros Idiomas: Deutsch English Français Italiano 日本語 한국어 Português Русский Slovenčina Tiếng Việt 简体中文 正體中文

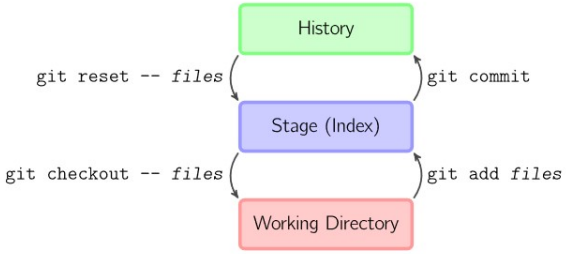
Una referencia visual de Git

Si las imágenes no funcionan, puedes intentar la versión sin-SVG de esta página.

Esta página da una referencia breve y visual para los comandos más comunes en git. Una vez que conozcas un poco sobre la forma de trabajo de git, este sitio puede fortalecer tu entendimiento. Si estás interesado en cómo se creó este sitio, mirá mi repositorio de GitHub.

Contenidos

Uso Básico

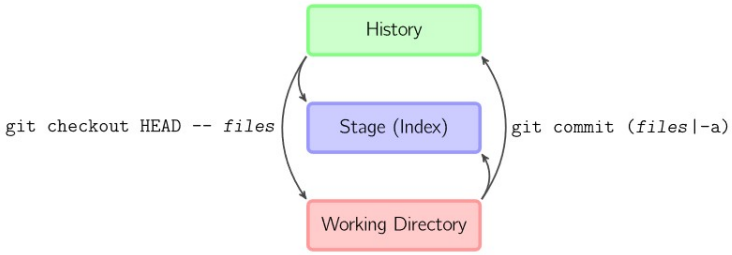


Los cuatro comandos de arriba copian archivos entre el directorio de trabajo, el stage (también llamado index), y la historia (en la forma de commits).

- `git add archivos` copia *archivos* (en su estado actual) al stage.
- `git commit` guarda una snapshot del stage como un commit.
- `git reset -- archivos` quita *archivos* de stage; esto es, que copia *archivos* del último commit al stage. Usá este comando para "deshacer" un `git add archivos`. También podés usar `git reset` para quitar de stage todo lo que hayas agregado.
- `git checkout -- archivos` copia *archivos* desde el stage al directorio de trabajo. Usá esto para descartar los cambios locales.

Podés usar `git reset -p`, `git checkout -p`, o `git add -p` en lugar de (o en combinación con) especificar archivos particulares para elegir interactivamente qué partes copiar.

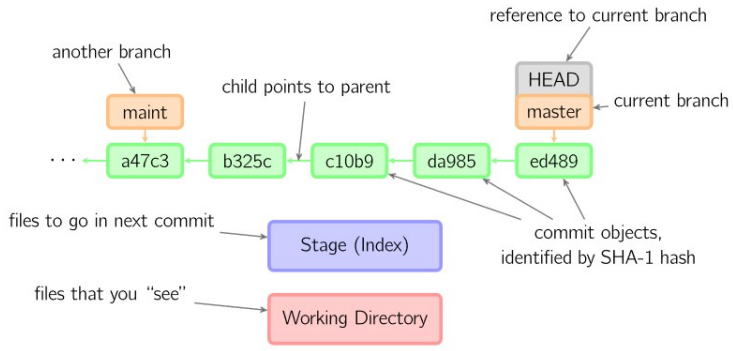
También es posible, además, saltarse el stage y hacer check-out de los archivos directamente desde los archivos de commit sin hacer primero el staging.



- `git commit -a` es equivalente a utilizar el comando `git add` en todos los archivos que hubo en el último commit, y correr luego el comando `git commit`.
- `git commit archivos` crea un nuevo commit incluyendo los contenidos del último commit, además de un snapshot de *archivos* tomado del directorio de trabajo. Adicionalmente, los *archivos* se copian al stage.
- `git checkout HEAD -- archivos` copia *archivos* del último commit a stage y al directorio de trabajo (a ambos)

Convenciones

En el resto del documento, usaremos gráficos de la siguiente forma.

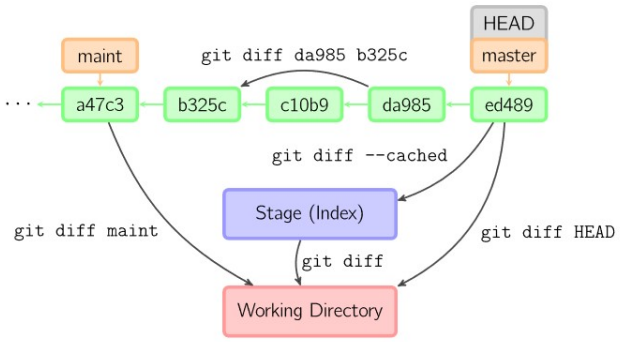


Los commits se muestran en verde como identificadores de 5 caracteres, y apuntan a sus padres. Los branch se muestran en naranja, y apuntan a un commit en particular. El branch actual se identifica por medio de una referencia especial *HEAD*, que es "adjuntada" al branch. En esta imagen, se muestran los cinco últimos commits, con *ed489* siendo el más reciente. *master* (el branch actual) apunta a este commit, mientras que *maint* (otro branch) apunta a un antecesor del commit de *master*.

Comandos en Detalle

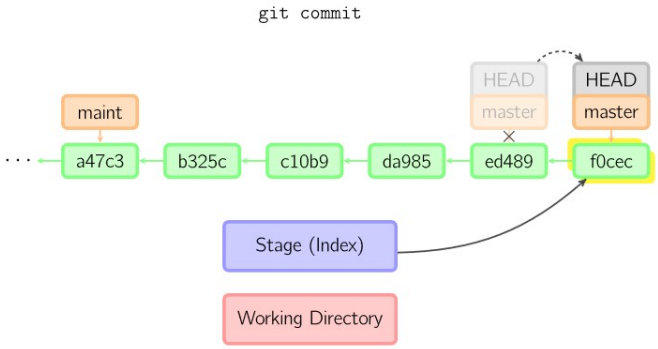
Diff

Hay varias formas de ver las diferencias entre commits. Debajo están algunos ejemplos comunes. Cualquiera de esos comandos pueden tomar opcionalmente como argumento nombres de archivos que limiten las diferencias a los archivos nombrados.

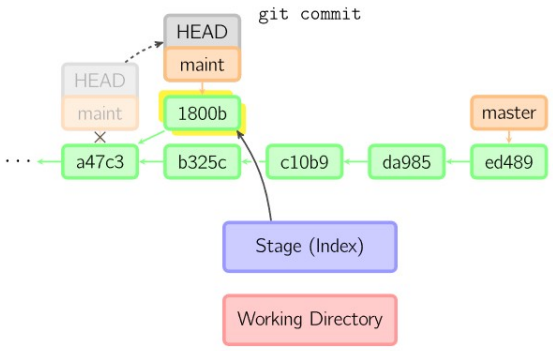


Commit

Cuando commiteás, git crea un nuevo objeto de commit utilizando los archivos del stage y establece el padre al commit actual. Entonces apunta el presente branch a este nuevo commit. En la imagen debajo, el branch actual es *master*. Antes de que se ejecutase el comando, *master* apuntaba a *ed489*. Luego, un nuevo commit, *f0cec*, se crea, con el padre *ed489*, y luego *master* se mueve al nuevo commit.

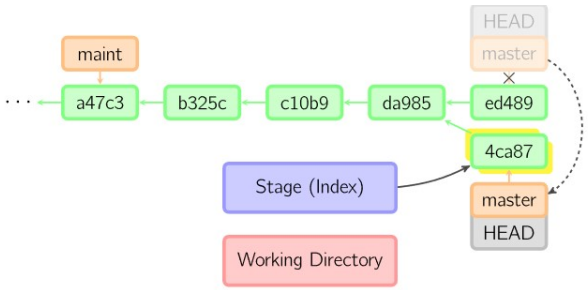


Este mismo proceso sucede aún cuando el branch actual es un ancestro de otro. Debajo, un commit ocurre en el branch *maint*, el cual fue un ancestro de *master*, resultando en *1800b*. Luego, *maint* no es más un ancestro de *master*. Para reunir las dos historias, será necesario un merge (o rebase).



A veces pueden cometerse errores en un commit, pero son fáciles de corregir con `git commit --amend`. Cuando usás este comando, git crea un nuevo commit con el mismo padre que el commit actual. (El commit anterior será descartado si nada más lo referencia.)

```
git commit --amend
```



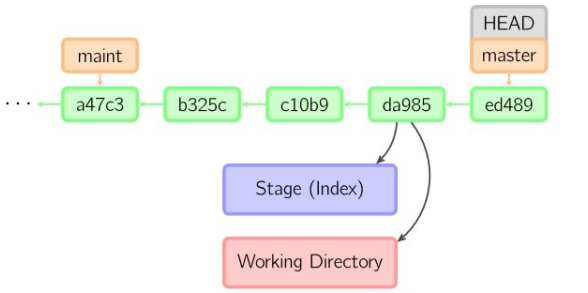
Un cuarto caso es commitear con un HEAD detachado, como explicaremos después.

Checkout

El comando checkout se usa para copiar archivos de los commits (o desde stage) al directorio de trabajo, y para opcionalmente intercambiar branches.

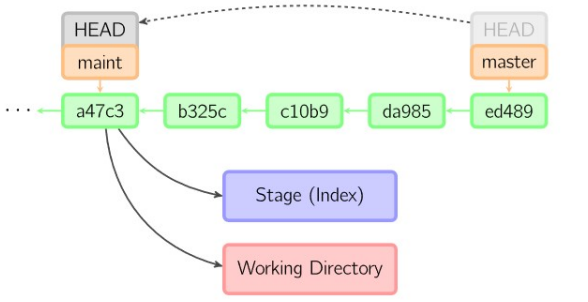
Cuando un nombre de archivo (y/o `-p`) se proporciona, git copia esos archivos desde el commit dado hacia stage y el directorio de trabajo. Por ejemplo, `git checkout HEAD~ foo.c` copia el archivo `foo.c` desde el commit llamado *HEAD~* (el padre del commit actual) al directorio de trabajo, y además lo pasa a stage. (Si no se proporciona nombre de commit, los archivos se copiarán desde stage.) Notá que el branch actual no se cambia.

`git checkout HEAD~ files`



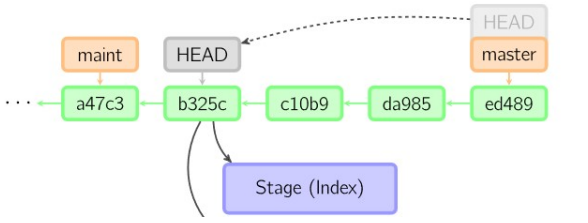
Cuando *not* se proporciona un nombre de archivo, pero la referencia es un branch (local), *HEAD* se mueve a ese branch (esto es, nosotros "cambiamos" ese branch), y entonces el stage y el directorio de trabajo se establecen para coincidir con los contenidos de ese commit. Cualquier archivo que existe en el nuevo commit (*a47c3* debajo) se copia; cualquier archivo que exista en el anterior commit (*ed489*) pero no en el que está borrado; y cualquier archivo que exista será ignorado.

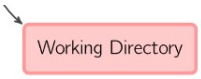
`git checkout maint`



Cuando *not* se proporciona un nombre de archivo y la referencia *no* es un branch (local) — digamos, es un tag, un branch remoto, un identificador SHA-1, o algo como *master~3* — obtenemos un branch anónimo, llamado un *HEAD detachado*. Esto es útil para saltar a través de la historia. Digamos que querés compilar la versión 1.6.6.1 de git. Podés hacer `git checkout v1.6.6.1` (que es un tag, no un branch), compilar, instalar, y luego cambiar de nuevo a otro branch, digamos `git checkout master`. Sin embargo, commitear trabajos levemente diferentes con un HEAD detachado; esto es cubierto debajo.

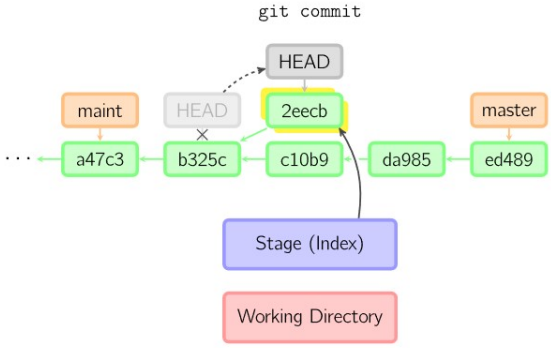
`git checkout master~3`



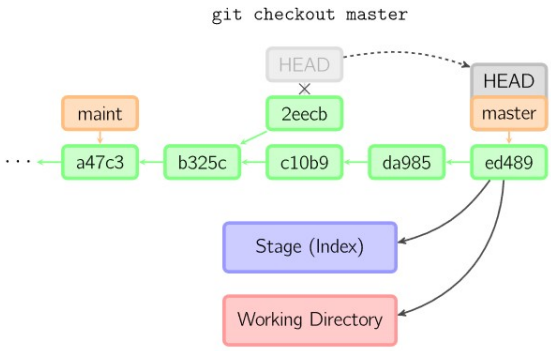


Commiteando con un HEAD detachado

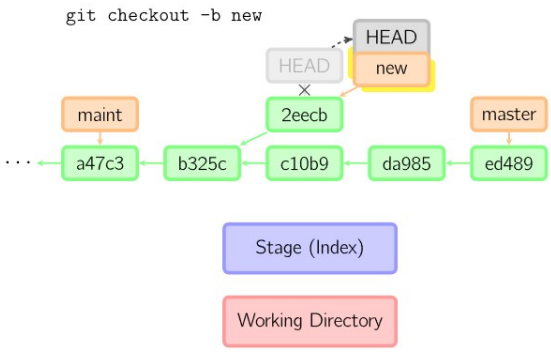
Cuando el *HEAD* está detachado, commit trabaja como siempre, excepto que en un branch sin nombre. (Podés pensar en esto como un branch anónimo.)



Una vez que hacés un check-out de algo más, digamos *master*, el commit (supuestamente) no es referenciado más por nada, y se pierde. Notá que luego de ese comando, no hay nada referenciando *2eecb*.



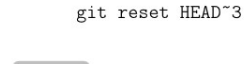
Si, por otro lado, querés guardar este estado, podés crear un nuevo branch con nombre usando `git checkout -b nombre`.

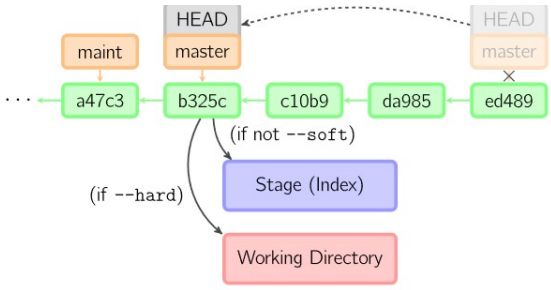


Reset

El comando reset mueve el branch actual a otra posición, y opcionalmente actualiza el stage y el directorio de trabajo. Además se usa para copiar archivos desde los commits a stage sin tocar el directorio de trabajo.

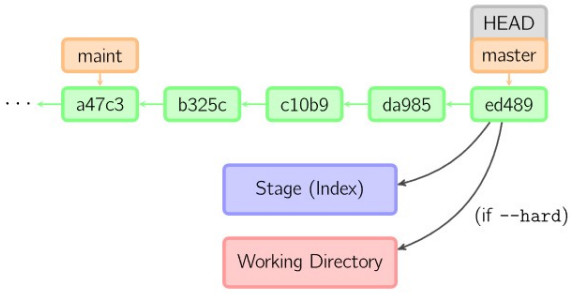
Si un commit se hace sin especificar nombres de archivos, el branch actual se mueve a ese commit, y luego el stage se actualiza para coincidir con ese commmit. Si se proporciona el parámetro `--hard` el directorio de trabajo también se actualiza. Si se proporciona el parámetro `--soft` ninguno se actualiza.





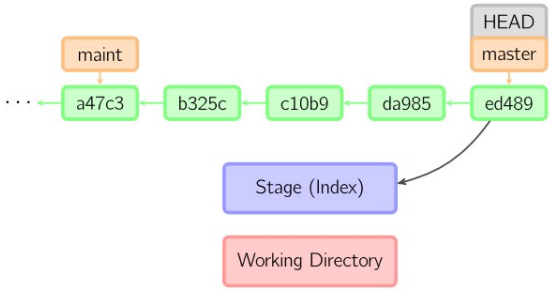
Si no se proporciona un commit, por defecto se refiere a *HEAD*. En este caso, el branch no se mueve, pero se pasa a stage (y opcionalmente al directorio de trabajo si se especifica el parámetro `--hard`) y reinicia a los contenidos del último commit.

`git reset`



Si se proporciona un nombre de archivo (y/o `-p`), entonces el comando trabaja en forma similar al checkout con un nombre de archivo, excepto que solo el stage (y no el directorio de trabajo) se actualice. (Además deberías especificar el commit del cual tomar los archivos, en lugar de *HEAD*.)

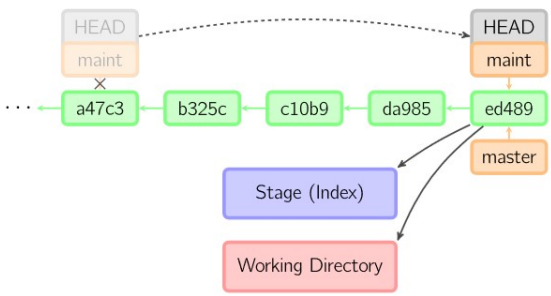
`git reset -- files`



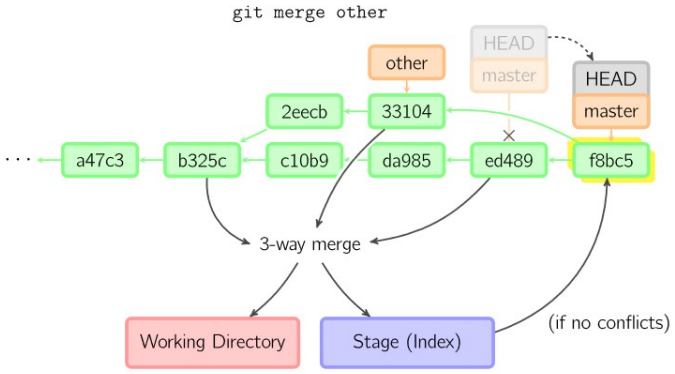
Merge

Un merge crea un nuevo commit que incorpora cambios de otros commit. Antes de mezclar, el stage debe coincidir con el commit actual. El caso trivial es si el otro commit es un ancestro del commit actual, en cuyo caso no se hace nada adicional. El siguiente más simple es si el commit actual es un ancestro del otro commit. Esto resulta en una mezcla *fast-forward*. La referencia simplemente se mueve, y luego al nuevo commit se le hace check out.

`git merge master`

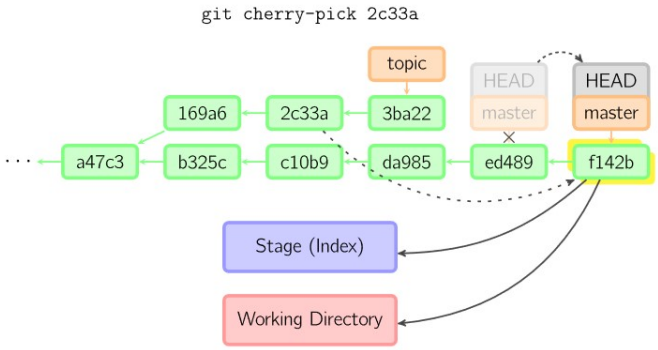


De otro modo, una mezcla "real" debe ocurrir. Podés elegir otras estrategias, pero por defecto se realiza una mezcla "recursive", la cual básicamente toma el commit actual (*ed489* debajo), el otro commit (*33104*), y su ancestro común (*b325c*), y realiza una mezcla de tres vías. El resultado se guarda al directorio de trabajo y al stage, y luego ocurre un commit, con un padre adicional (*33104*) para el nuevo commit.



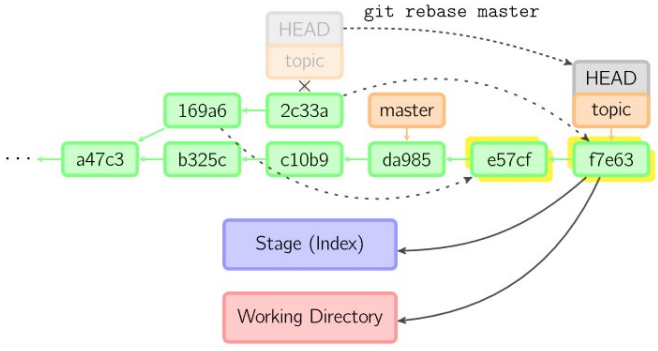
Cherry Pick

El comando cherry-pick "copia" un commit, creando un nuevo commit en el branch actual con el mismo mensaje y patch que otro commit.



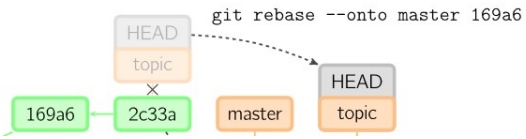
Rebase

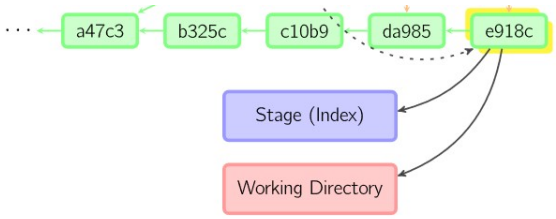
Un rebase es una alternativa al merge para combinar múltiples branches. Mientras que un merge crea un solo commit con dos padres, dejando una historia no lineal, un rebase reproduce los commits del branch actual en otro, dejando una historia lineal. En esencia, esta es una forma automatizada de realizar muchas cherry-pick de una vez.



El comando de arriba toma todos los commits que existen en *topic* pero no en *master* (nombrados *169a6* y *2c33a*), los reproduce dentro de *master*, y luego mueve el branch head al nuevo punto. Notá que los commits viejos serán recogidos por el recolector de basura si no son referenciados.

Para limitar qué tanto va hacia atrás, usá la opción `--onto`. El siguiente comando reproduce dentro de *master* el más reciente commit del branch actual desde *169a6* (exclusivo), concretamente *2c33a*.





Además existe `git rebase --interactive`, que permite hacer cosas más complicadas que simplemente reproducir commits, descartar, reordenar, modificar o juntar commits. No hay una imagen obvia para graficar este concepto; ver git-rebase(1) para más detalles.

Notas Técnicas

Los contenidos de los archivos no se almacenan en realidad en el índice (*.git/index*) o en objetos commit. En su lugar, cada archivo se almacena en el objeto database (*.git/objects*) como un *blob*, identificado por su hash SHA-1. El archivo de índice lista los nombres de los archivos junto con el identificador del blob asociado, así como otros datos. Para los commits, hay un tipo de dato adicional, un *tree*, también identificado por su hash. Los trees se corresponden con directorios del directorio de trabajo, y contienen una lista de trees y blobs que se corresponden a cada nombre de archivo dentro de ese directorio. Cada commit almacena el identificador de su tree de alto nivel, el cual a su vez contiene todos los blobs y otros trees asociados con ese commit.

Si hacés un commit usando un HEAD detachado, el último commit es el que se referencia por algo: el reflog para HEAD. Sin embargo, esto expirará luego de un tiempo, por eso el commit eventualmente será recogido por el garbage collector, de forma similar a los commits descartados con `git commit --amend` o `git rebase`.
