Una referencia de Visual Git

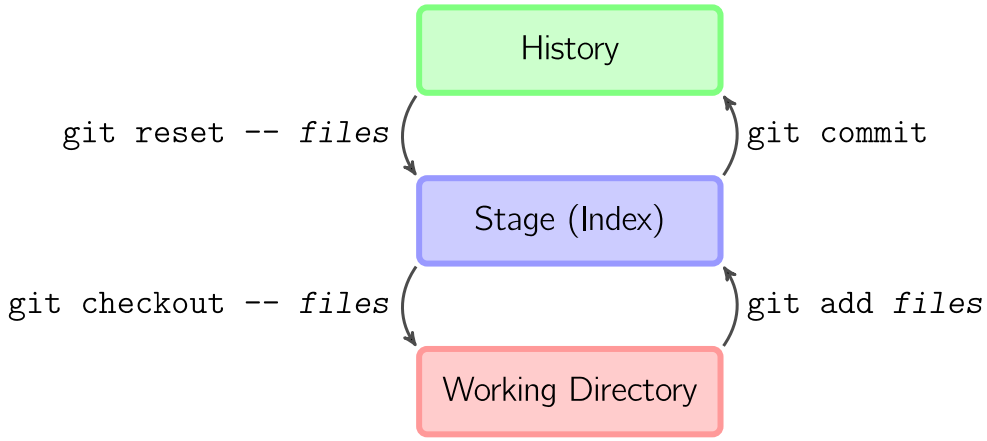
Si las imágenes no funcionan, puede probar la versión Non-SVG de esta página.

Esta página ofrece una breve referencia visual de los comandos más comunes en git. Una vez que sepa un poco sobre cómo funciona Git, este sitio puede solidificar su comprensión. Si está interesado en cómo se creó este sitio, consulte mi repositorio de GitHub .

También se recomienda: Visualizar conceptos de Git con D3

Contenido

Uso Básico

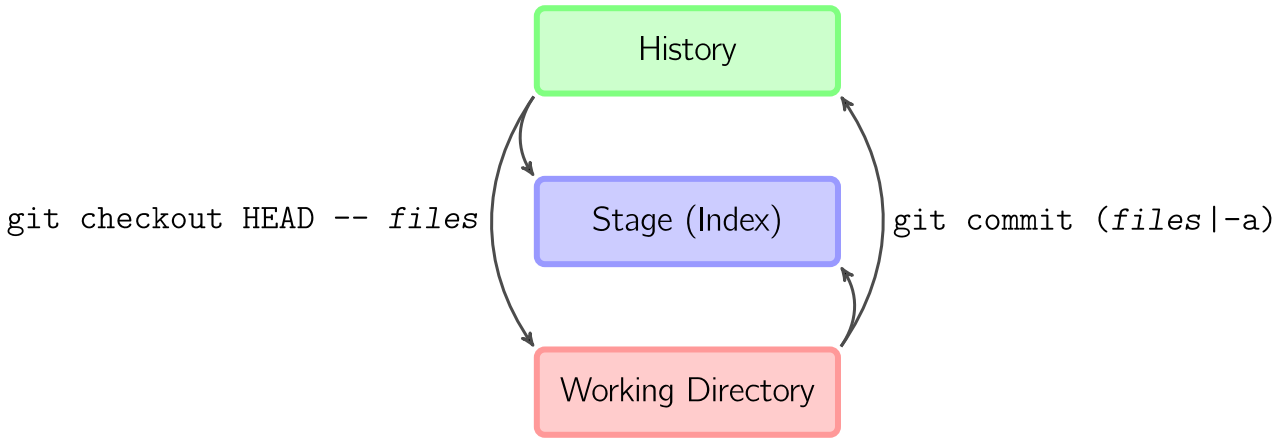


Los cuatro comandos anteriores copian archivos entre el directorio de trabajo, el escenario (también llamado índice) y el historial (en forma de confirmaciones).

- `git add files`copia *archivos* (en su estado actual) al escenario.
- `git commit` guarda una instantánea del escenario como una confirmación.
- `git reset -- files`archivos de las etapas; es decir, copia los *archivos* del último compromiso al escenario. Use este comando para "deshacer" a . También puedes borrar todo.`git add files``git reset`
- `git checkout -- files`copia *archivos* desde el escenario al directorio de trabajo. Use esto para deshacerse de los cambios locales.

Puede usar `git reset -p`, `git checkout -p`o en `git add -p`lugar de (o además de) especificar archivos particulares para elegir de forma interactiva qué copia de trozos.

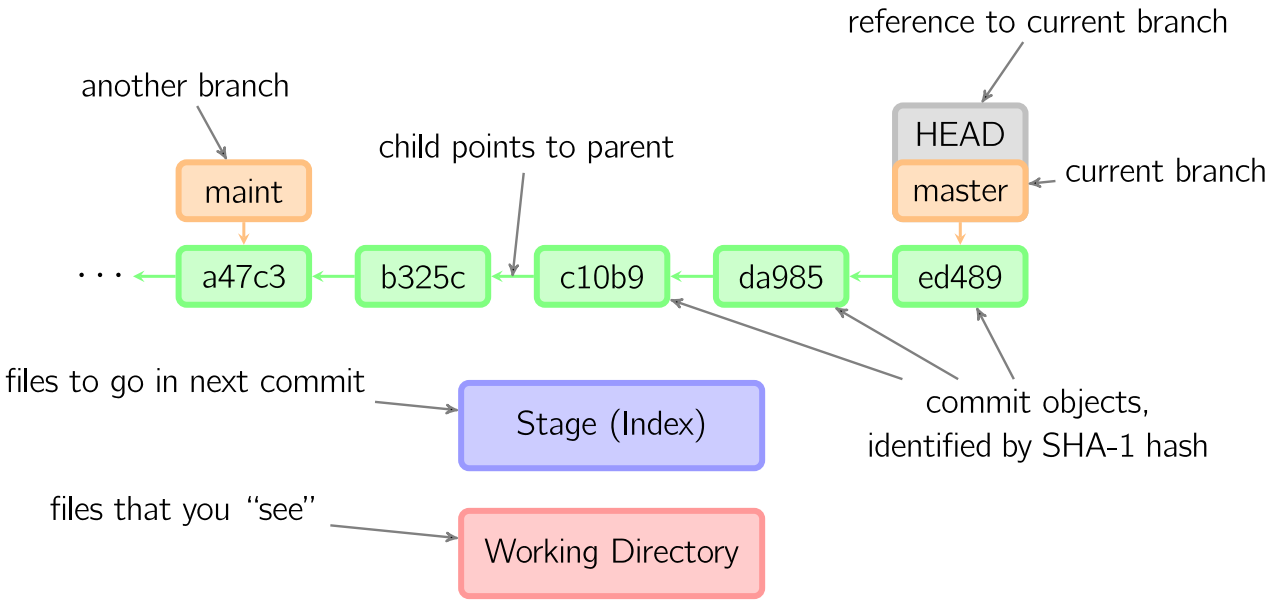
También es posible saltar sobre el escenario y verificar archivos directamente desde el historial o confirmar archivos sin organizar primero.



- `git commit -a` es equivalente a ejecutar `git add` en todos los nombres de archivo que existían en la última confirmación, y luego ejecutar `git commit` .
- `git commit files`crea una nueva confirmación que contiene los contenidos de la última confirmación, más una instantánea de los *archivos* tomados del directorio de trabajo. Además, los *archivos* se copian en el escenario.
- `git checkout HEAD -- files`copia *archivos* de la última confirmación para el escenario y el directorio de trabajo.

Convenciones

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

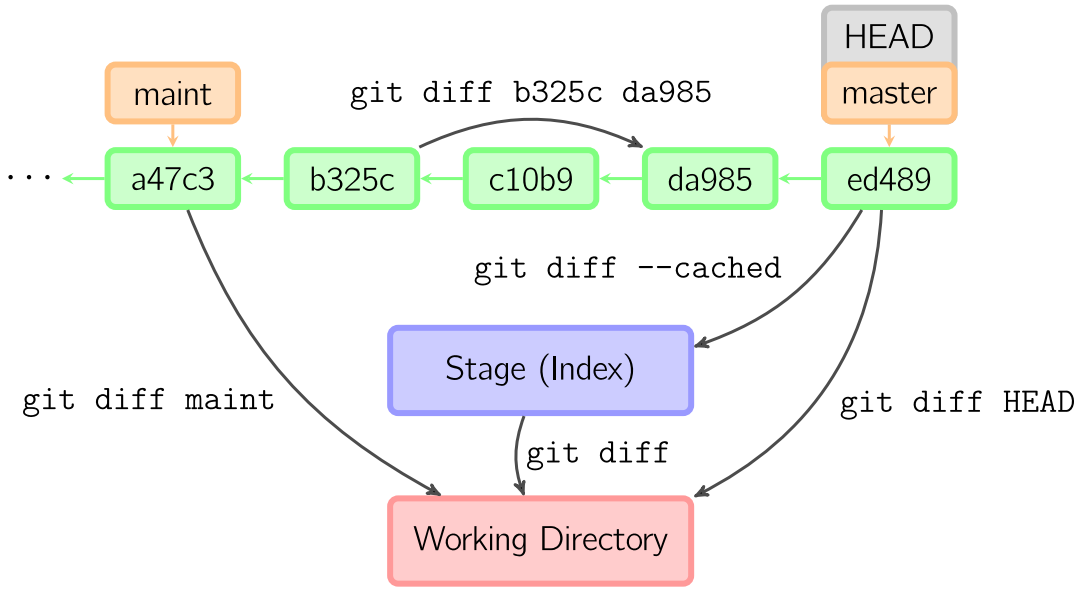


Los commits se muestran en verde como ID de 5 caracteres, y apuntan a sus padres. Las ramas se muestran en naranja, y apuntan a compromisos particulares. La rama actual se identifica por la *CABEZA de* referencia especial , que está "conectada" a esa rama. En esta imagen, se muestran los cinco últimos commits, siendo *ed489* el más reciente. *master* (la rama actual) apunta a este commit, mientras que *maint* (otra rama) apunta a un antecesor de la confirmación del *maestro* .

Comandos en detalle

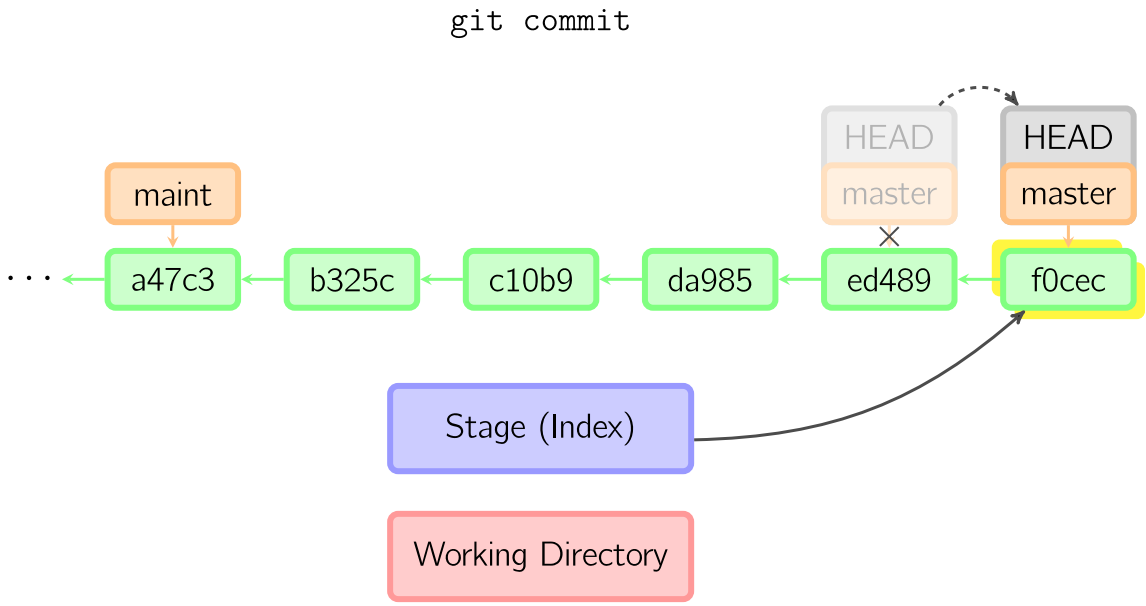
Diferencia

Hay varias maneras de ver las diferencias entre commits. A continuación hay algunos ejemplos comunes. Cualquiera de estos comandos puede tomar opcionalmente argumentos adicionales de nombre de archivo que limiten las diferencias a los archivos nombrados.

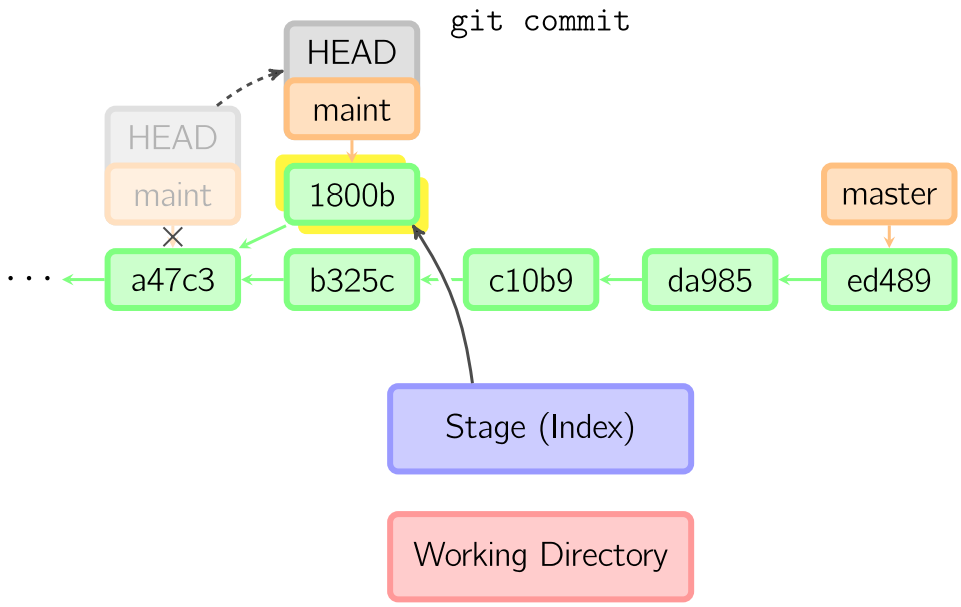


Cometer

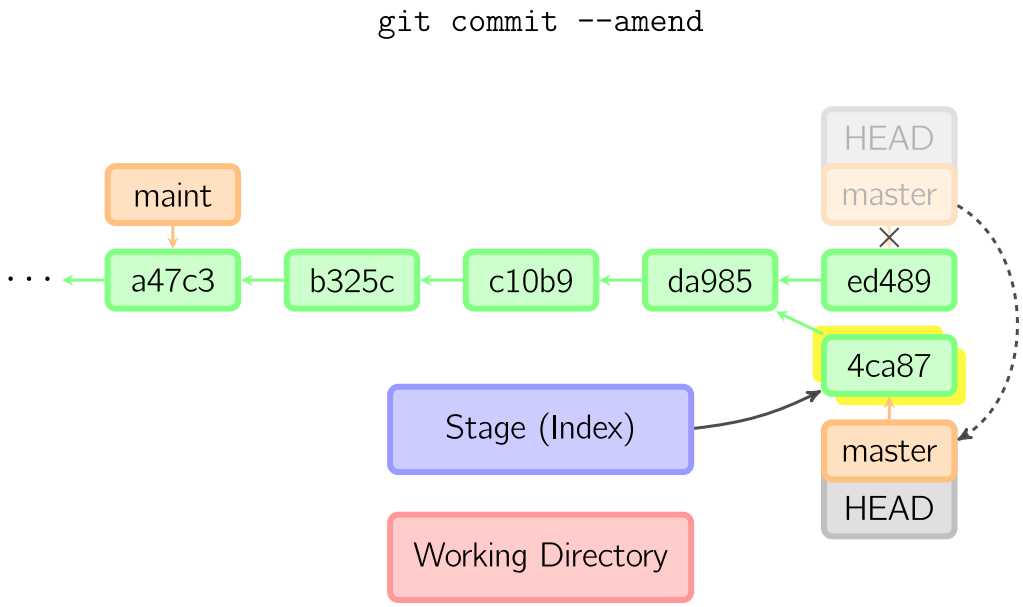
When you commit, git creates a new commit object using the files from the stage and sets the parent to the current commit. It then points the current branch to this new commit. In the image below, the current branch is *master*. Before the command was run, *master* pointed to *ed489*. Afterward, a new commit, *f0cec*, was created, with parent *ed489*, and then *master* was moved to the new commit.



This same process happens even when the current branch is an ancestor of another. Below, a commit occurs on branch *maint*, which was an ancestor of *master*, resulting in *1800b*. Afterward, *maint* is no longer an ancestor of *master*. To join the two histories, a merge (or rebase) will be necessary.



Sometimes a mistake is made in a commit, but this is easy to correct with `git commit --amend`. When you use this command, git creates a new commit with the same parent as the current commit. (The old commit will be discarded if nothing else references it.)



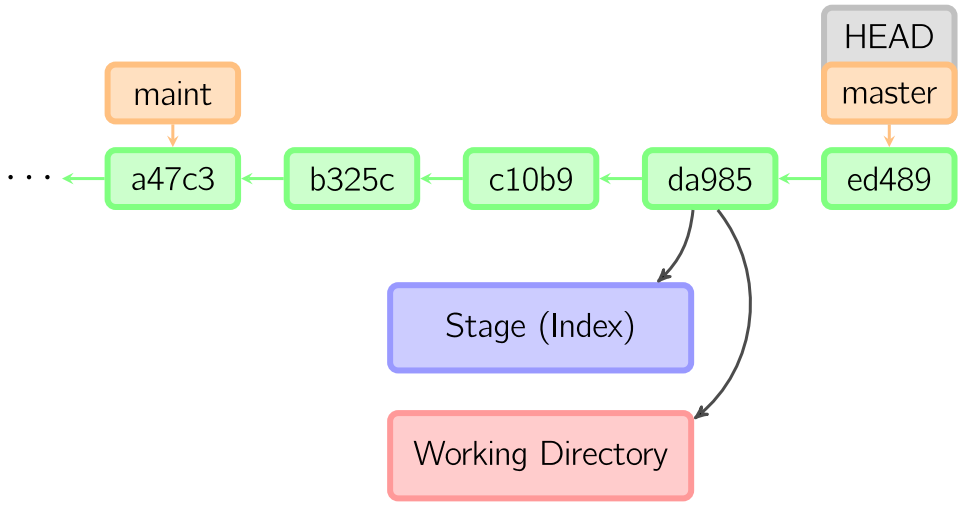
A fourth case is committing with a detached HEAD, as explained later.

Checkout

The checkout command is used to copy files from the history (or stage) to the working directory, and to optionally switch branches.

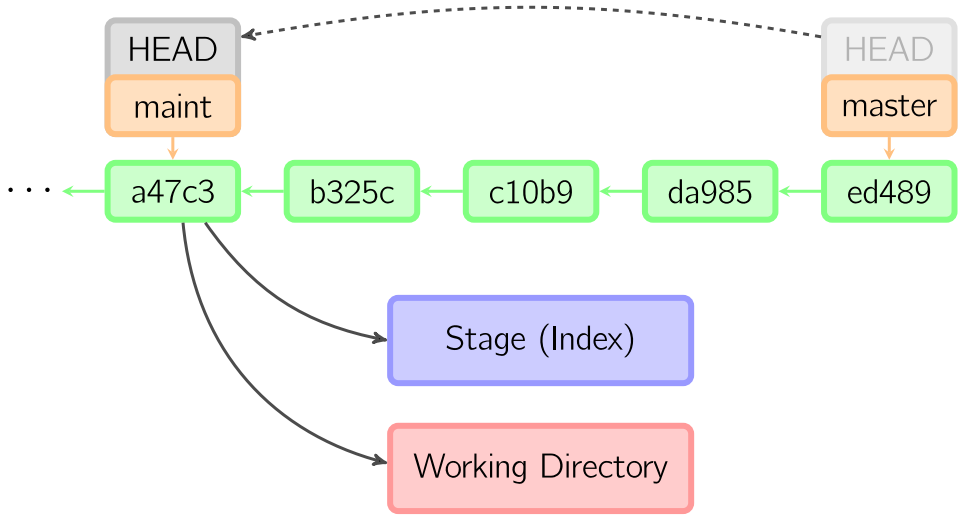
When a filename (and/or `-p`) is given, git copies those files from the given commit to the stage and the working directory. For example, `git checkout HEAD~ foo.c` copies the file `foo.c` from the commit called *HEAD~* (the parent of the current commit) to the working directory, and also stages it. (If no commit name is given, files are copied from the stage.) Note that the current branch is not changed.

`git checkout HEAD~ files`

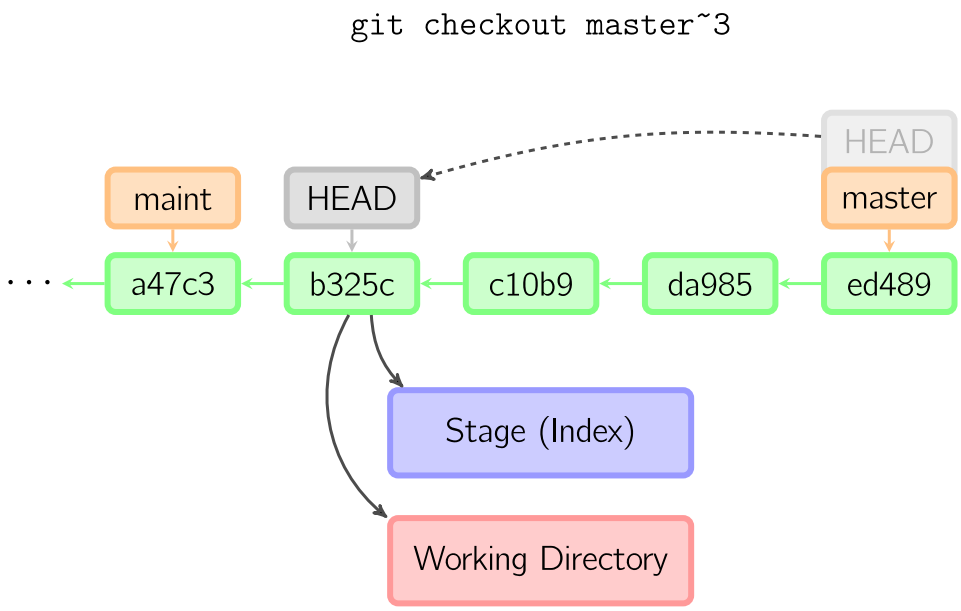


When a filename is *not* given but the reference is a (local) branch, *HEAD* is moved to that branch (that is, we "switch to" that branch), and then the stage and working directory are set to match the contents of that commit. Any file that exists in the new commit (*a47c3* below) is copied; any file that exists in the old commit (*ed489*) but not in the new one is deleted; and any file that exists in neither is ignored.

`git checkout maint`

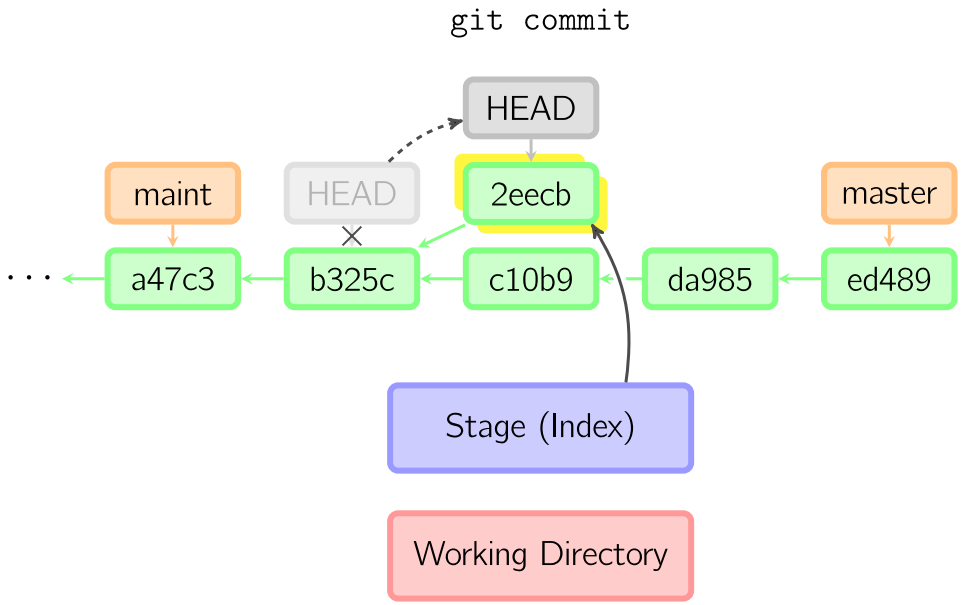


When a filename is *not* given and the reference is *not* a (local) branch — say, it is a tag, a remote branch, a SHA-1 ID, or something like *master~3* — we get an anonymous branch, called a *detached HEAD*. This is useful for jumping around the history. Say you want to compile version 1.6.6.1 of git. You can `git checkout v1.6.6.1` (which is a tag, not a branch), compile, install, and then switch back to another branch, say `git checkout master`. However, committing works slightly differently with a detached HEAD; this is covered below.

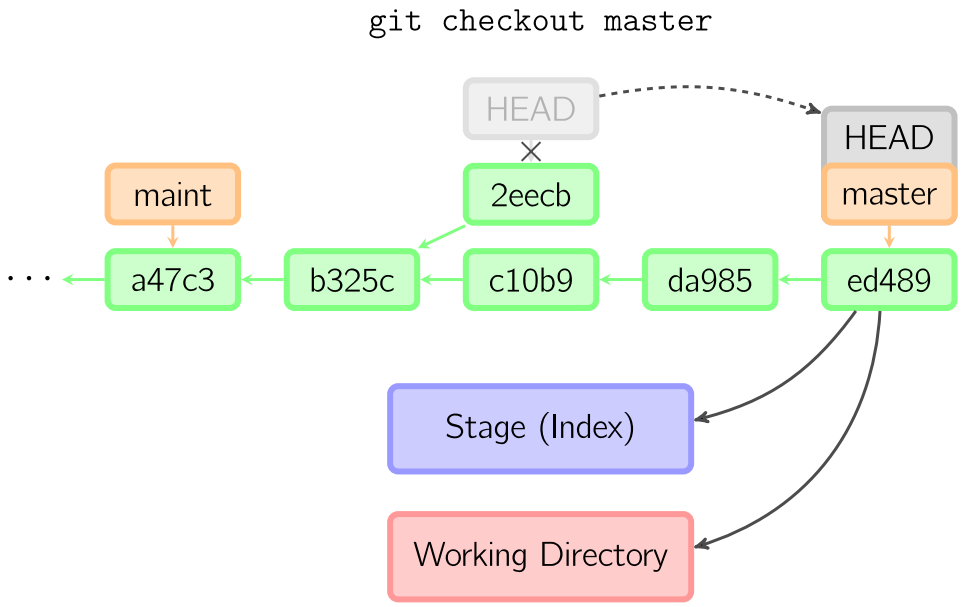


Committing with a Detached HEAD

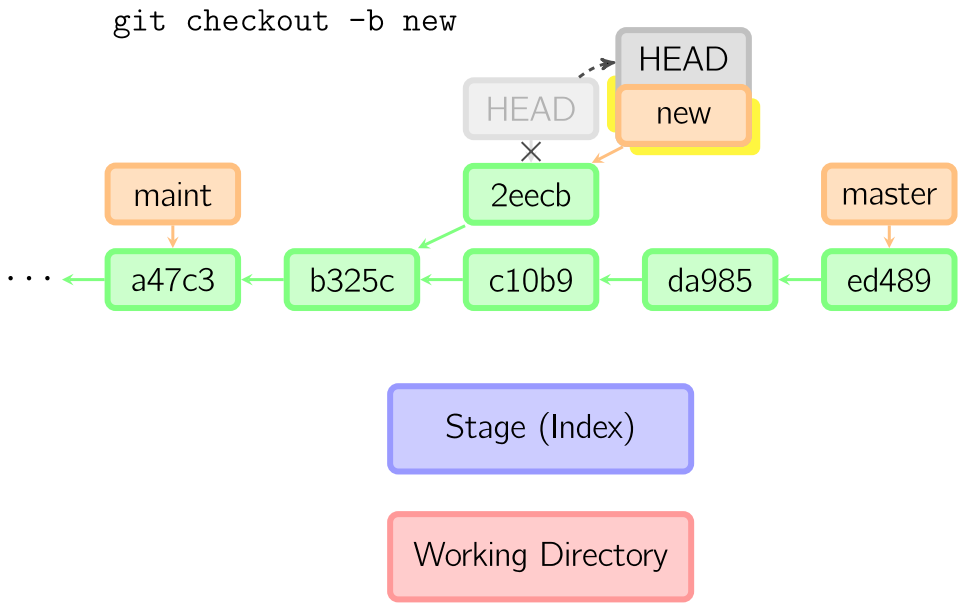
When *HEAD* is detached, commits work like normal, except no named branch gets updated. (You can think of this as an anonymous branch.)



Once you check out something else, say *master*, the commit is (presumably) no longer referenced by anything else, and gets lost. Note that after the command, there is nothing referencing *2eecb*.



If, on the other hand, you want to save this state, you can create a new named branch using `git checkout -b name`.

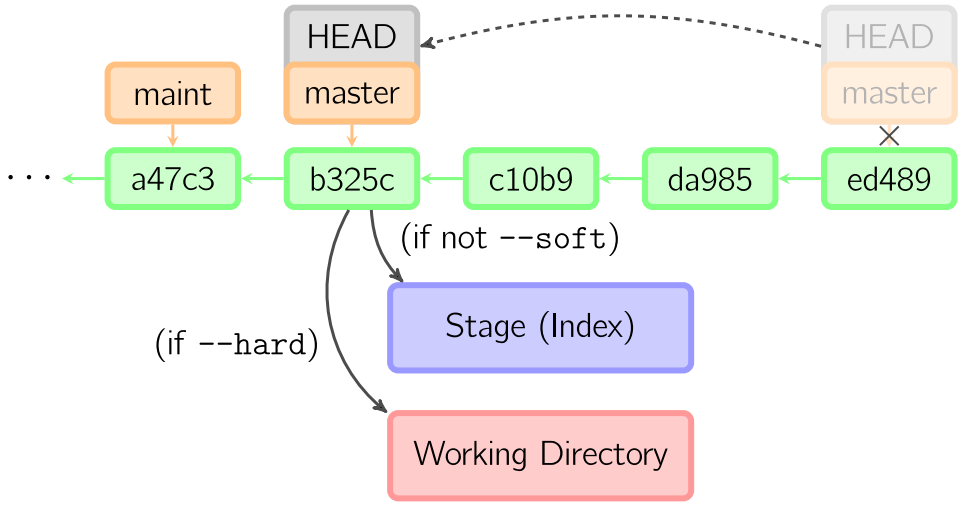


Reset

The reset command moves the current branch to another position, and optionally updates the stage and the working directory. It also is used to copy files from the history to the stage without touching the working directory.

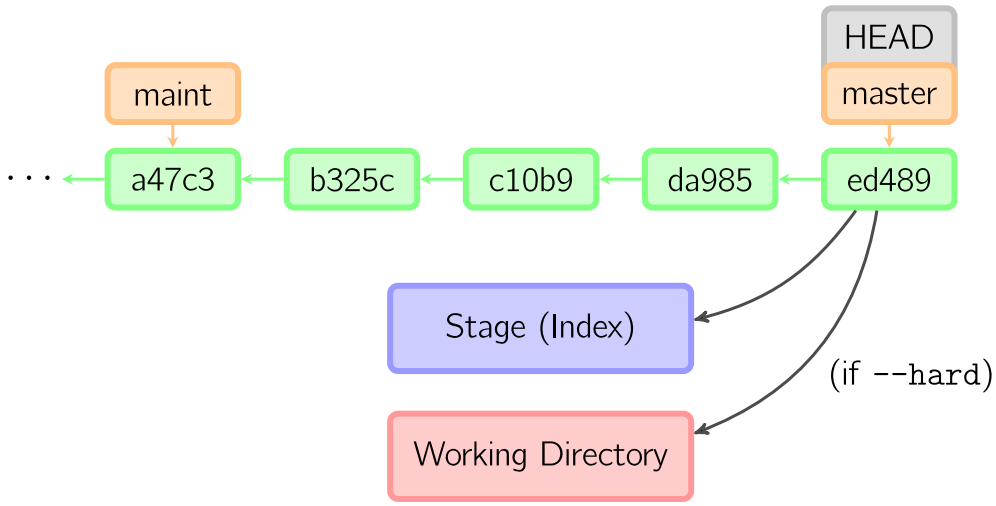
If a commit is given with no filenames, the current branch is moved to that commit, and then the stage is updated to match this commit. If `--hard` is given, the working directory is also updated. If `--soft` is given, neither is updated.

`git reset HEAD~3`



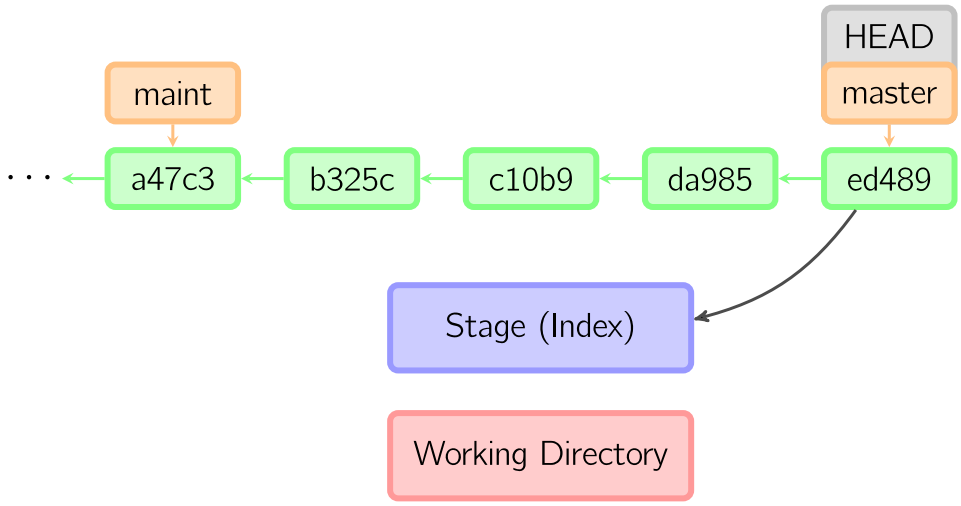
If a commit is not given, it defaults to *HEAD*. In this case, the branch is not moved, but the stage (and optionally the working directory, if `--hard` is given) are reset to the contents of the last commit.

`git reset`



If a filename (and/or `-p`) is given, then the command works similarly to checkout with a filename, except only the stage (and not the working directory) is updated. (You may also specify the commit from which to take files, rather than *HEAD*.)

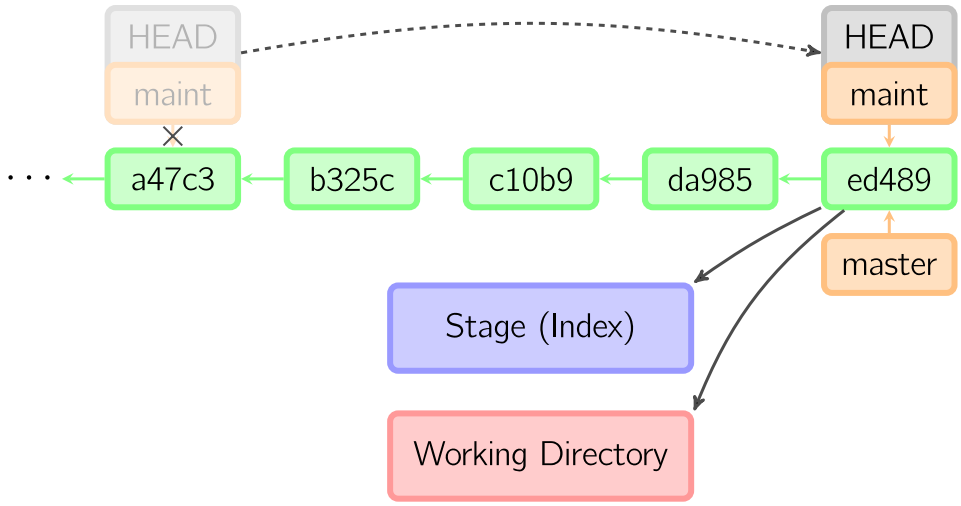
`git reset -- files`



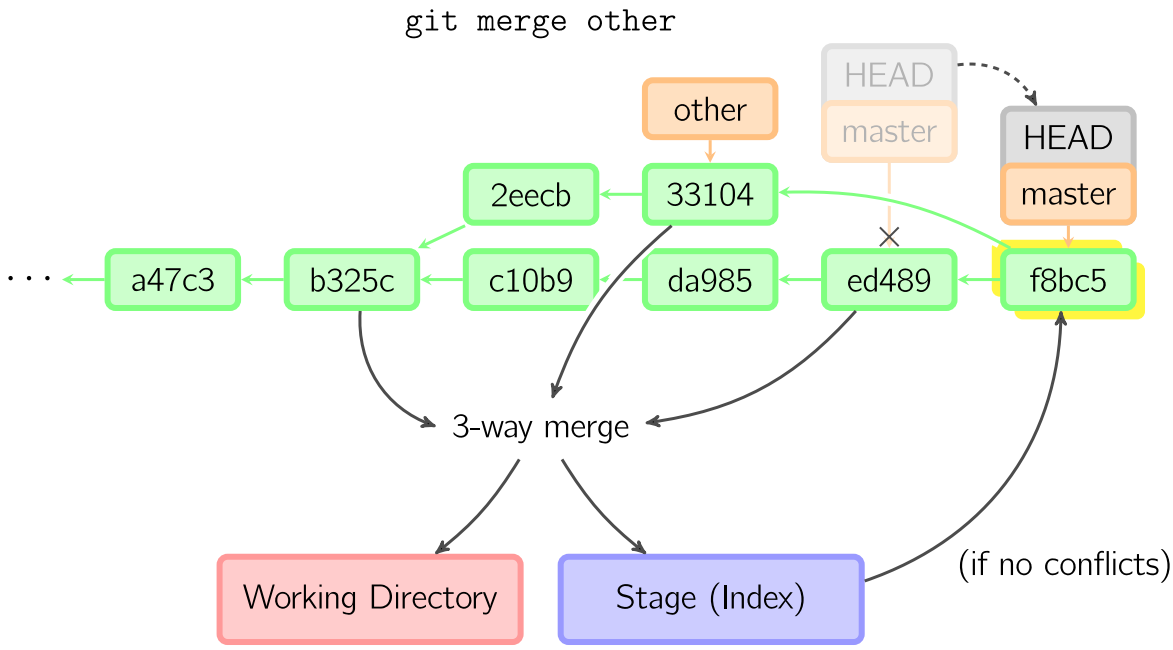
Merge

A merge creates a new commit that incorporates changes from other commits. Before merging, the stage must match the current commit. The trivial case is if the other commit is an ancestor of the current commit, in which case nothing is done. The next most simple is if the current commit is an ancestor of the other commit. This results in a *fast-forward* merge. The reference is simply moved, and then the new commit is checked out.

`git merge master`

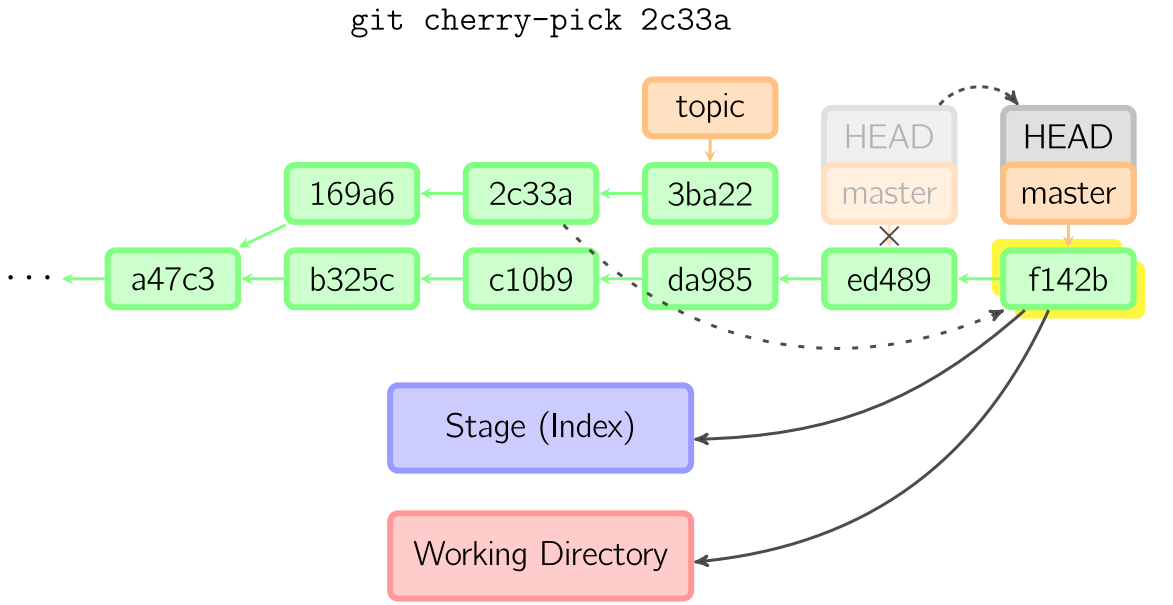


Otherwise, a "real" merge must occur. You can choose other strategies, but the default is to perform a "recursive" merge, which basically takes the current commit (*ed489* below), the other commit (*33104*), and their common ancestor (*b325c*), and performs a three-way merge. The result is saved to the working directory and the stage, and then a commit occurs, with an extra parent (*33104*) for the new commit.



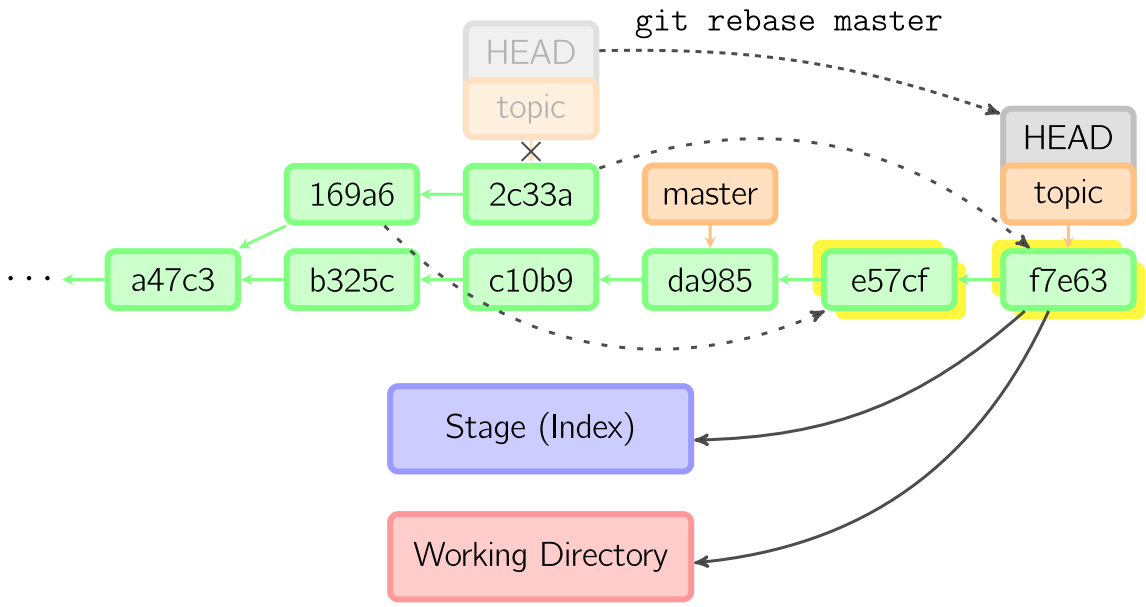
Cherry Pick

The cherry-pick command "copies" a commit, creating a new commit on the current branch with the same message and patch as another commit.



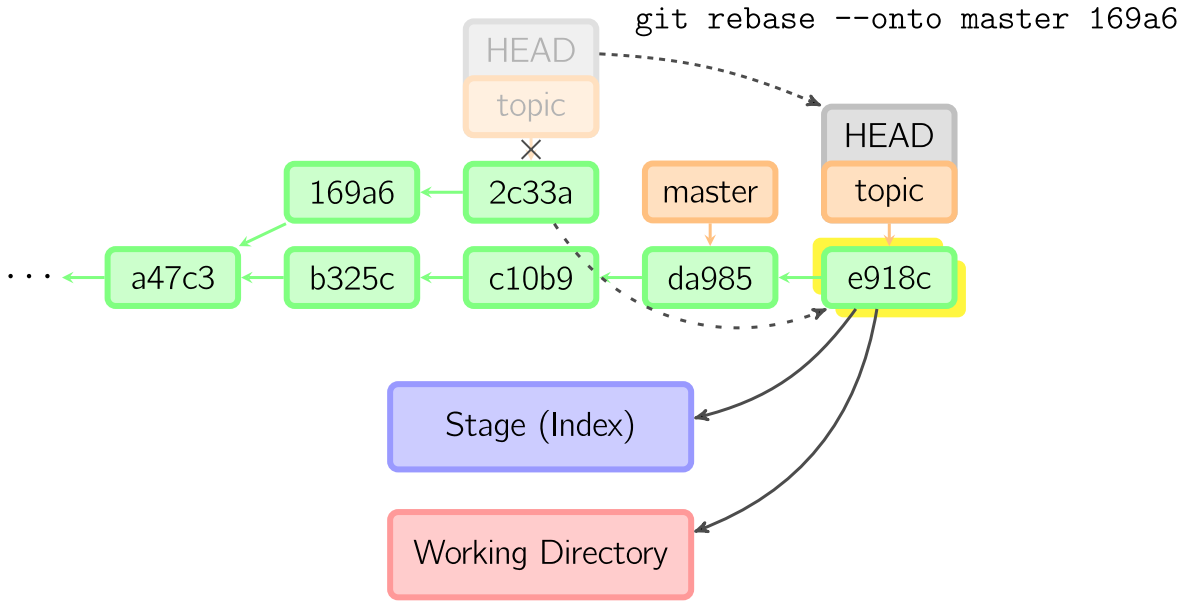
Rebase

A rebase is an alternative to a merge for combining multiple branches. Whereas a merge creates a single commit with two parents, leaving a non-linear history, a rebase replays the commits from the current branch onto another, leaving a linear history. In essence, this is an automated way of performing several cherry-picks in a row.



The above command takes all the commits that exist in *topic* but not in *master* (namely *169a6* and *2c33a*), replays them onto *master*, and then moves the branch head to the new tip. Note that the old commits will be garbage collected if they are no longer referenced.

To limit how far back to go, use the `--onto` option. The following command replays onto *master* the most recent commits on the current branch since *169a6* (exclusive), namely *2c33a*.



There is also `git rebase --interactive`, which allows one to do more complicated things than simply replaying commits, namely dropping, reordering, modifying, and squashing commits. There is no obvious picture to draw for this; see git-rebase(1) for more details.

Technical Notes

The contents of files are not actually stored in the index (*.git/index*) or in commit objects. Rather, each file is stored in the object database (*.git/objects*) as a *blob*, identified by its SHA-1 hash. The index file lists the filenames along with the identifier of the

associated blob, as well as some other data. For commits, there is an additional data type, a *tree*, also identified by its hash. Trees correspond to directories in the working directory, and contain a list of trees and blobs corresponding to each filename within that directory. Each commit stores the identifier of its top-level tree, which in turn contains all of the blobs and other trees associated with that commit.

If you make a commit using a detached HEAD, the last commit really is referenced by something: the reflog for HEAD. However, this will expire after a while, so the commit will eventually be garbage collected, similar to commits discarded with `git commit --amend` or `git rebase`.

Walkthrough: Watching the effect of commands

The following walks you through changes to a repository so you can see the effect of the command in real time, similar to how Visualizing Git Concepts with D3 simulates them visually. Hopefully you find this useful.

Start by creating some repository:

```
$ git init foo
$ cd foo
$ echo 1 > myfile
$ git add myfile
$ git commit -m "version 1"
```

Now, define the following functions to help us show information:

```
show_status() {
  echo "HEAD:     $(git cat-file -p HEAD:myfile)"
  echo "Stage:    $(git cat-file -p :myfile)"
  echo "Worktree: $(cat myfile)"
}

initial_setup() {
  echo 3 > myfile
  git add myfile
  echo 4 > myfile
  show_status
}
```

Initially, everything is at version 1.

```
$ show_status
HEAD:     1
Stage:    1
Worktree: 1
```

We can watch the state change as we add and commit.

```
$ echo 2 > myfile
$ show_status
HEAD:     1
Stage:    1
Worktree: 2
$ git add myfile
$ show_status
HEAD:     1
Stage:    2
Worktree: 2
$ git commit -m "version 2"
[master 4156116] version 2
 1 file changed, 1 insertion(+), 1 deletion(-)
$ show_status
HEAD:     2
Stage:    2
Worktree: 2
```

Now, let's create an initial state where the three are all different.

```
$ initial_setup
HEAD:     2
Stage:    3
Worktree: 4
```

Let's watch what each command does. You will see that they match the diagrams above.

`git reset -- myfile` copies from HEAD to stage:

```
$ initial_setup
HEAD:     2
Stage:    3
Worktree: 4
```

```
$ git reset -- myfile
Unstaged changes after reset:
M   myfile
$ show_status
HEAD:     2
Stage:    2
Worktree: 4
```

`git checkout -- myfile` copies from stage to worktree:

```
$ initial_setup
HEAD:     2
Stage:    3
Worktree: 4
$ git checkout -- myfile
$ show_status
HEAD:     2
Stage:    3
Worktree: 3
```

`git checkout HEAD -- myfile` copies from HEAD to both stage and worktree:

```
$ initial_setup
HEAD:     2
Stage:    3
Worktree: 4
$ git checkout HEAD -- myfile
$ show_status
HEAD:     2
Stage:    2
Worktree: 2
```

`git commit myfile` copies from worktree to both stage and HEAD:

```
$ initial_setup
HEAD:     2
Stage:    3
Worktree: 4
$ git commit myfile -m "version 4"
[master 679ff51] version 4
 1 file changed, 1 insertion(+), 1 deletion(-)
$ show_status
HEAD:     4
Stage:    4
Worktree: 4
```

Want to translate into another language?