<http://gitimmersion.com/lab_01.html>

# Staging and Committing

A separate staging step in git is in line with the philosophy of getting out of the way until you need to deal with source control. You can continue to make changes to your working directory, and then at the point you want to interact with source control, git allows you to record your changes in small commits that record exactly what you did.

For example, suppose you edited three files (a.rb, b.rb, and c.rb). Now you want to commit all the changes, but you want the changes in a.rb and b.rb to be a single commit, while the changes to c.rb are not logically related to the first two files and should be a separate commit.

You could do the following:

git add a.rb

git add b.rb

git commit -m "Changes for a and b"

git add c.rb

git commit -m "Unrelated change to c"

By separating staging and committing, you have the ability to easily fine tune what goes into each commit.

Getting a listing of what changes have been made is the function of the git log command.

### Execute:

git log

## Controlling Which Entries are Displayed

There are a lot of options for selecting which entries are displayed in the log. Play around with the following options:

git log --pretty=oneline --max-count=2

git log --pretty=oneline --since='5 minutes ago'

git log --pretty=oneline --until='5 minutes ago'

git log --pretty=oneline --author=<your name>

git log --pretty=oneline --all

## Getting Fancy

Here’s what I use to review the changes made in the last week. I’ll add --author=jim if I only want to see changes I made.

git log --all --pretty=format:'%h %cd %s (%an)' --since='7 days ago'

Over time, I’ve decided that I like the following log format for most of my work.

### Execute:

git log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short

It looks like this:

### Output:

$ git log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short

\* 7bf0bf1 2018-09-28 | Added a comment (HEAD -> master) [Jim Weirich]

\* 9cf3f21 2018-09-28 | Added a default value [Jim Weirich]

\* 94e1b8b 2018-09-28 | Using ARGV [Jim Weirich]

\* f656098 2018-09-28 | First Commit [Jim Weirich]

Let’s look at it in detail:

* --pretty="..." defines the format of the output.
* %h is the abbreviated hash of the commit
* %d are any decorations on that commit (e.g. branch heads or tags)
* %ad is the author date
* %s is the comment
* %an is the author name
* --graph informs git to display the commit tree in an ASCII graph layout
* --date=short keeps the date format nice and short

## Dumping the Latest Commit

Using the SHA1 hash from the commit listed above …

### Execute:

git cat-file -t <hash>

git cat-file -p <hash>

**NOTE:** If you defined the ‘type’ and ‘dump’ aliases from the aliases lab, then you can type git type and git dump rather than the longer cat-file commands (which I never remember).

# Bare Repositories

The convention is that repositories ending in ‘.git’ are bare repositories. We can see that there is no working directory in the hello.git repo. Essentially it is nothing but the .git directory of a non-bare repo.

http://ftp.newartisans.com/pub/git.from.bottom.up.pdf

===================================================================

<https://dev.to/digitalocean/how-to-use-git-a-reference-guide-6b6>

## Branches

A branch in Git is a movable pointer to one of the commits in the repository, it allows you to isolate work and manage feature development and integrations. You can learn more about branches by reading the [Git documentation](https://git-scm.com/book/en/v1/Git-Branching-What-a-Branch-Is).

## Show Changes

The git diff command shows changes between commits, branches, and more. You can read more fully about it through the [Git documentation](https://git-scm.com/docs/git-diff).

Compare modified files that are on the staging area

git diff --staged

Display the diff of what is in a-branch but is not in b-branch.

git diff a-branch..b-branch

Show the diff between two specific commits.

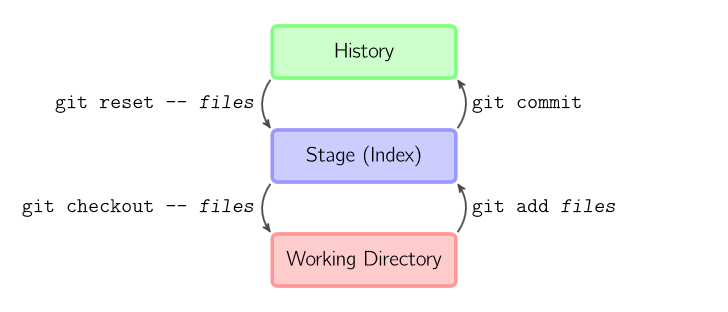
git diff 61ce3e6..e221d9c

## Stashing

Sometimes you’ll find that you made changes to some code, but before you finish you have to begin working on something else. You’re not quite ready to commit the changes you have made so far, but you don’t want to lose your work. The git stash command will allow you to save your local modifications and revert back to the working directory that is in line with the most recent HEAD commit.

<https://docs.google.com/presentation/d/1IQCRPHEIX-qKo7QFxsD3V62yhyGA9_5YsYXFOiBpgkk/edit?usp=sharing>

<http://marklodato.github.io/visual-git-guide/index-en.html>

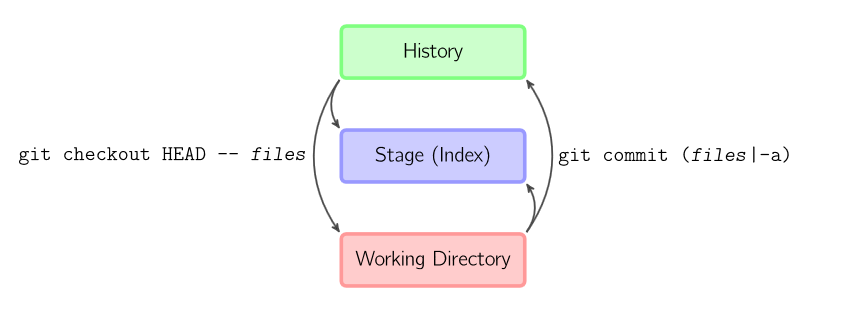


The four commands above copy files between the working directory, the stage (also called the index), and the history (in the form of commits).

* git add *files* copies *files* (at their current state) to the stage.
* git commit saves a snapshot of the stage as a commit.
* git reset -- *files* unstages files; that is, it copies *files* from the latest commit to the stage. Use this command to "undo" a git add *files*. You can also git reset to unstage everything.
* git checkout -- *files* copies *files* from the stage to the working directory. Use this to throw away local changes.

You can use git reset -p, git checkout -p, or git add -p instead of (or in addition to) specifying particular files to interactively choose which hunks copy.

It is also possible to jump over the stage and check out files directly from the history or commit files without staging first.

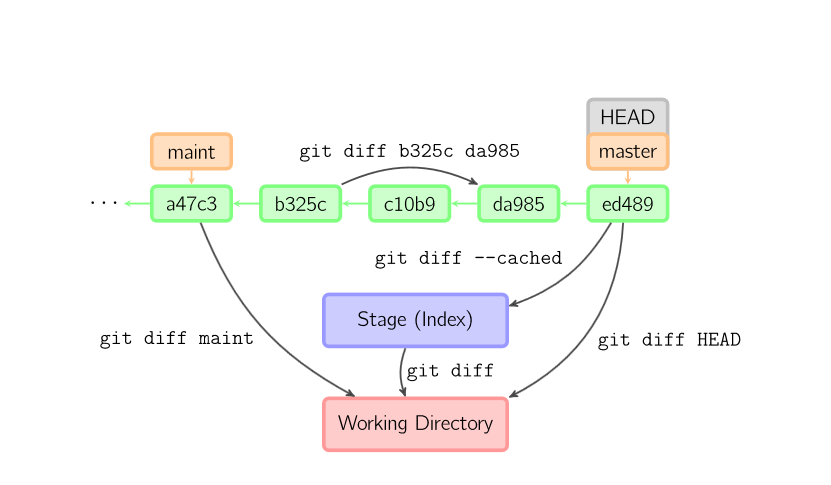


* git commit -a is equivalent to running git add on all filenames that existed in the latest commit, and then running git commit.
* git commit *files* creates a new commit containing the contents of the latest commit, plus a snapshot of *files* taken from the working directory. Additionally, *files* are copied to the stage.
* git checkout HEAD -- *files* copies *files* from the latest commit to both the stage and the working directory.

## Commands in Detail

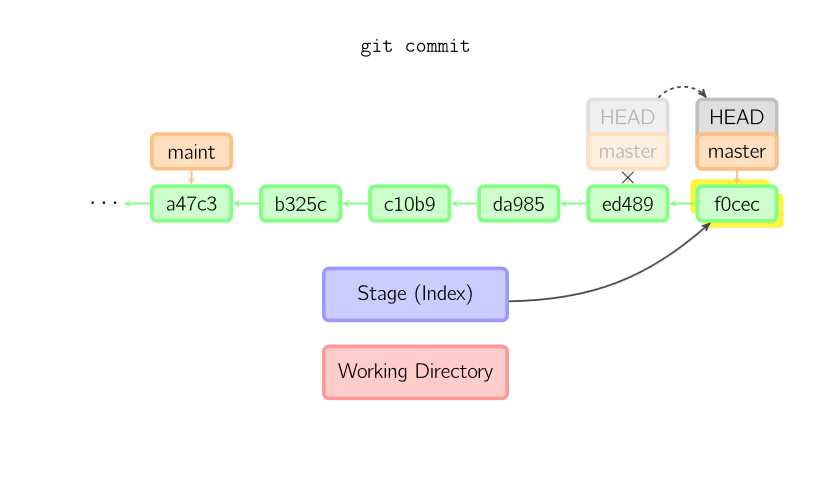
### Diff

There are various ways to look at differences between commits. Below are some common examples. Any of these commands can optionally take extra filename arguments that limit the differences to the named files

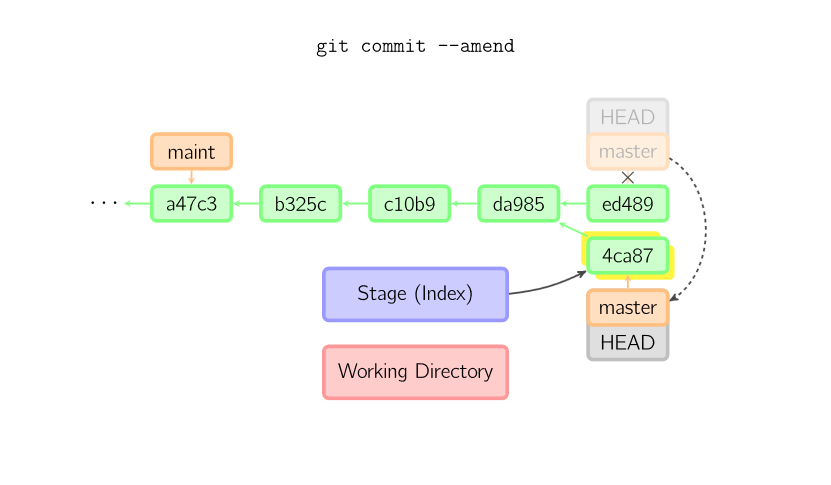


### Commit

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



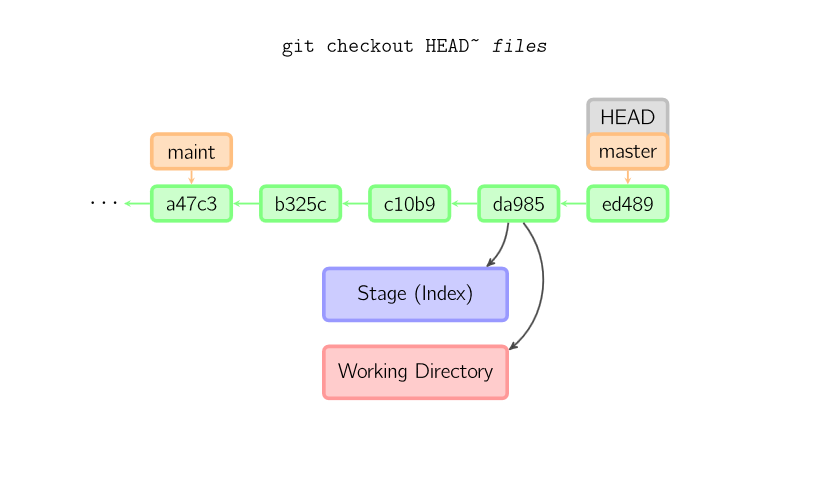
Sometimes a mistake is made in a commit, but this is easy to correct with git commit --amend. When you use this command, git creates a new commit with the same parent as the current commit. (The old commit will be discarded if nothing else references it.)



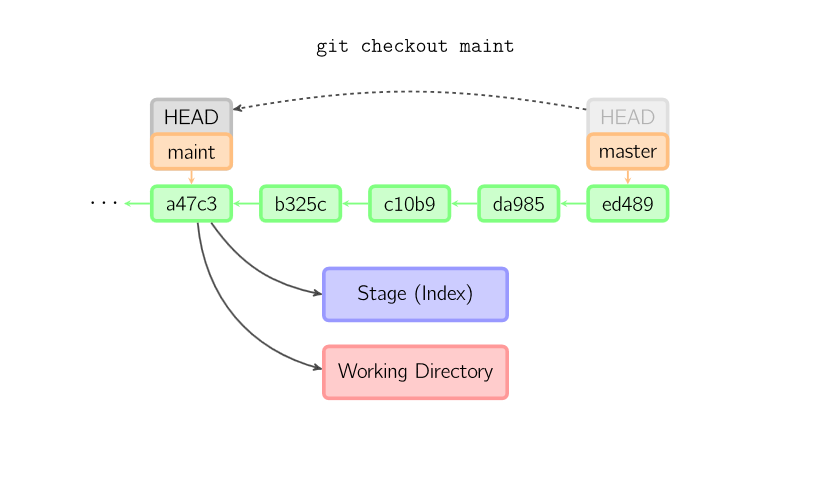
### Checkout

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

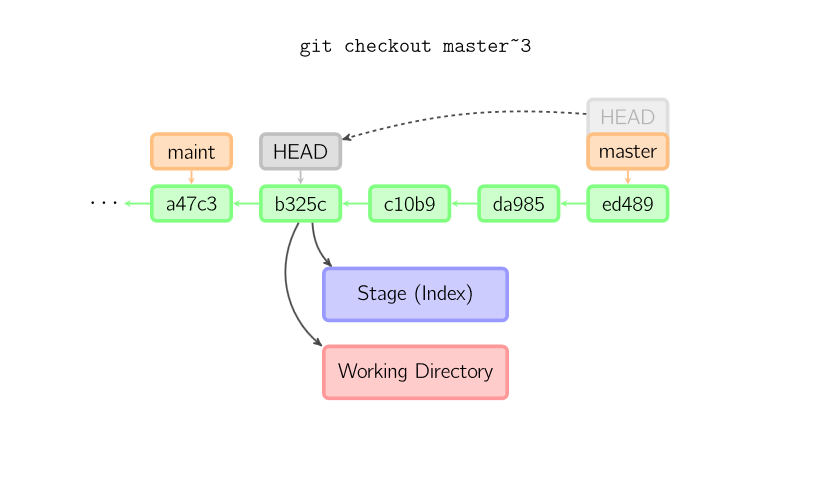
When a filename (and/or -p) is given, git copies those files from the given commit to the stage and the working directory. For example, git checkout HEAD~ foo.c copies the file foo.c from the commit called HEAD~ (the parent of the current commit) to the working directory, and also stages it. (If no commit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



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.

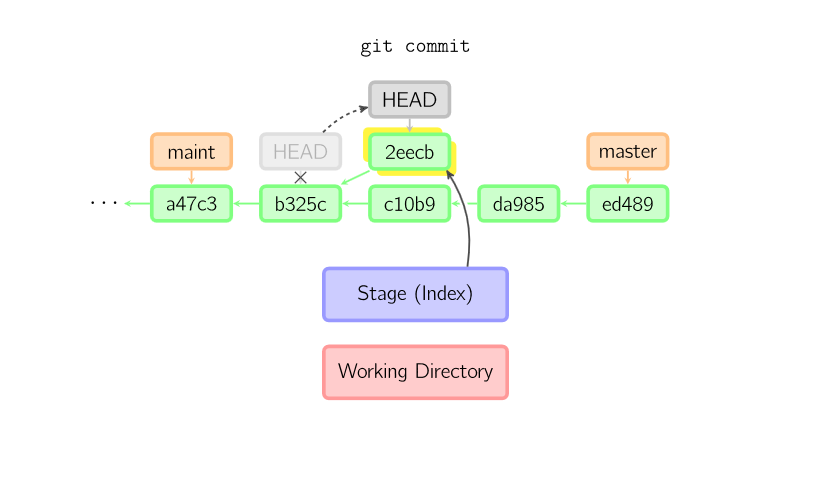


When a filename is not given and the reference is not a (local) branch — say, it is a tag, a remote branch, a SHA-1 ID, or something like master~3 — we get an anonymous branch, called a detached HEAD. This is useful for jumping around the history. Say you want to compile version 1.6.6.1 of git. You can 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](http://marklodato.github.io/visual-git-guide/index-en.html#detached).

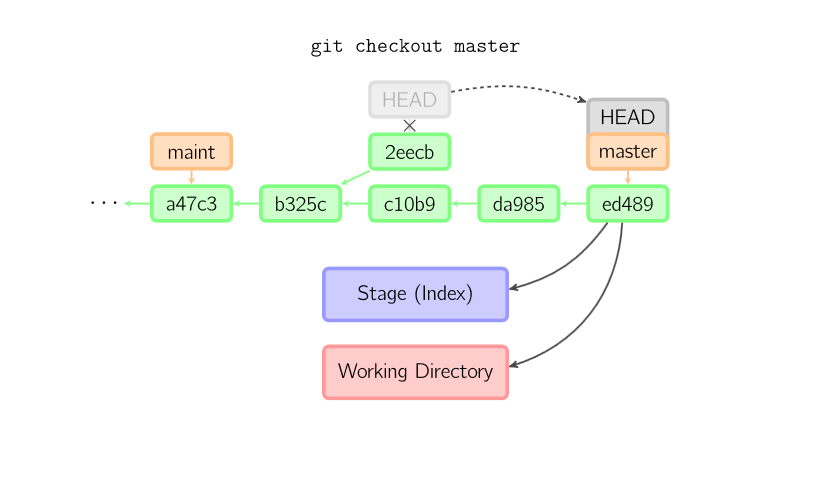


### Committing with a Detached HEAD

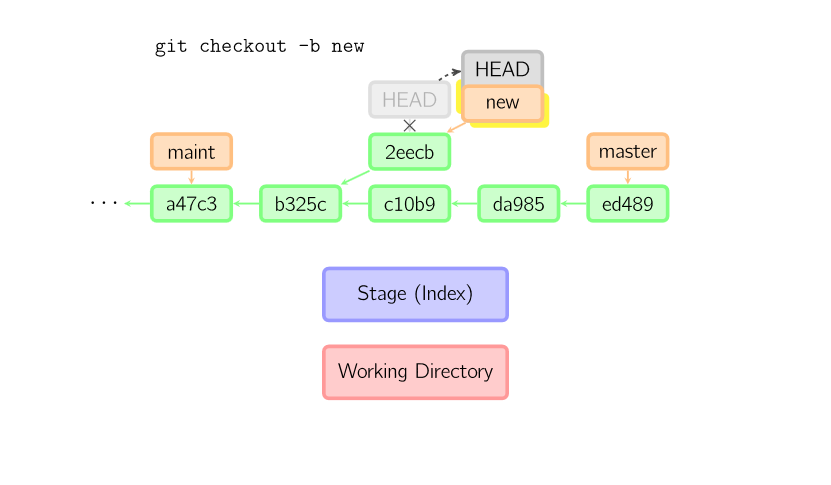
When HEAD is detached, commits work like normal, except no named branch gets updated. (You can think of this as an anonymous branch.)



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.



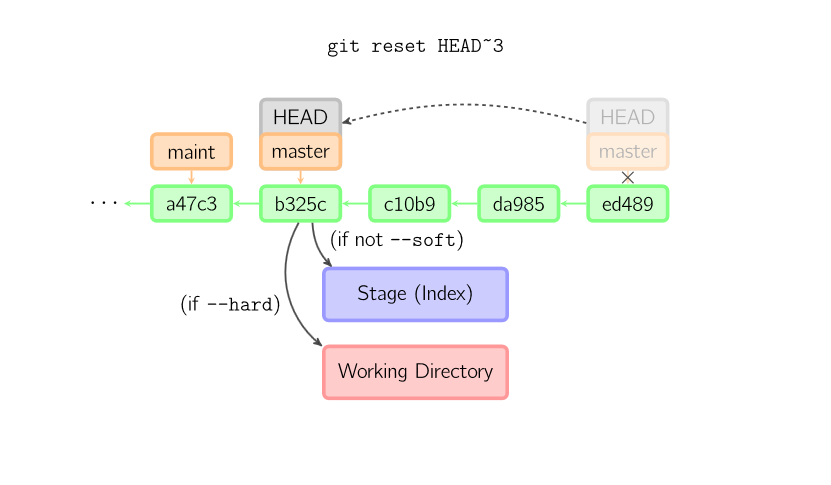
If, on the other hand, you want to save this state, you can create a new named branch using git checkout -b name



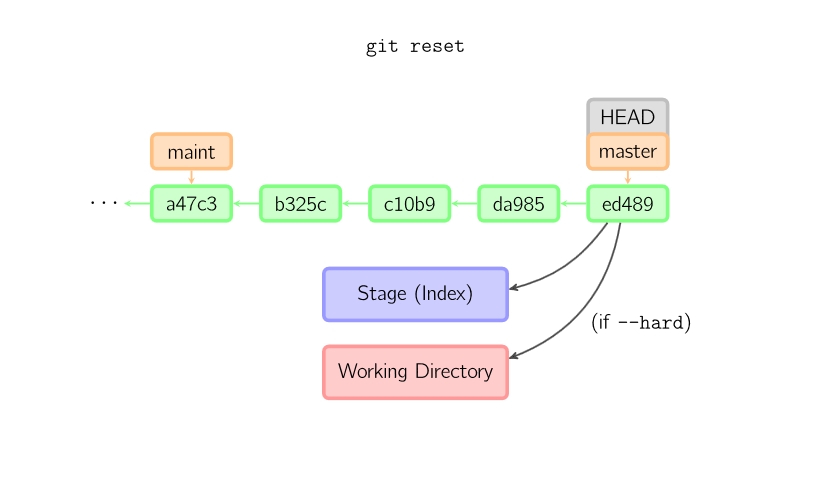
### Reset

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.

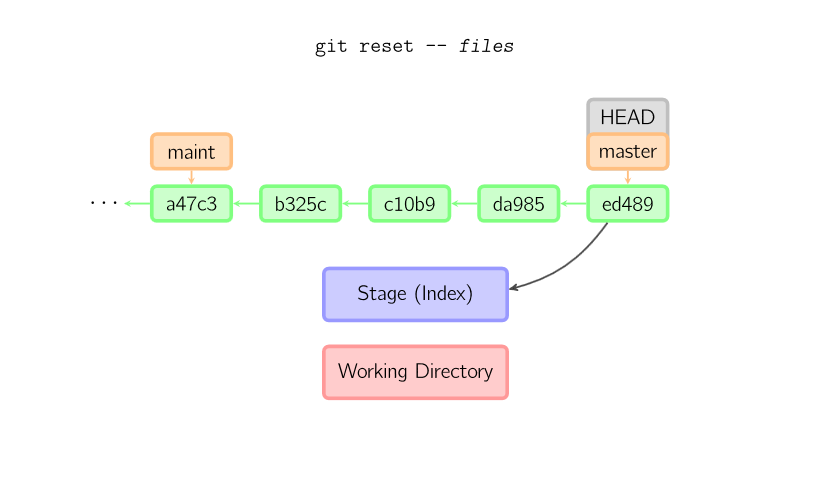
If a commit is given with no filenames, the current branch is moved to that commit, and then the stage is updated to match this commit. If --hard is given, the working directory is also updated. If --soft is given, neither is updated.



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

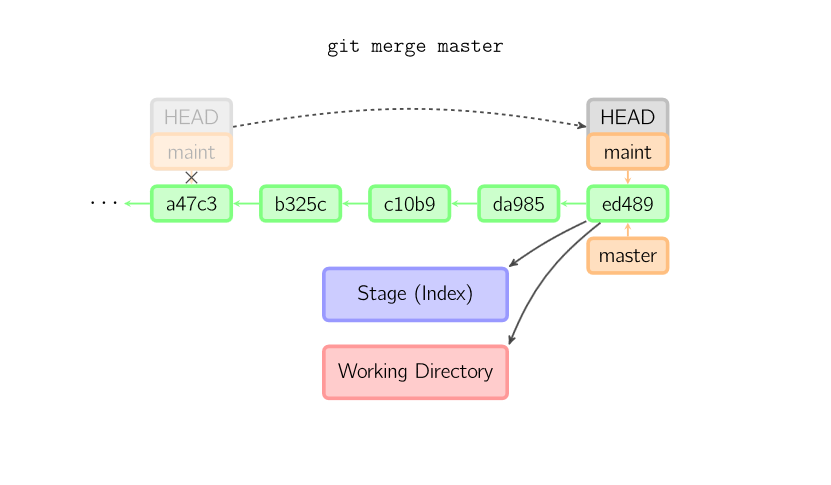


If a filename (and/or -p) is given, then the command works similarly to [checkout](http://marklodato.github.io/visual-git-guide/index-en.html#checkout) with a filename, except only the stage (and not the working directory) is updated. (You may also specify the commit from which to take files, rather than HEAD.)

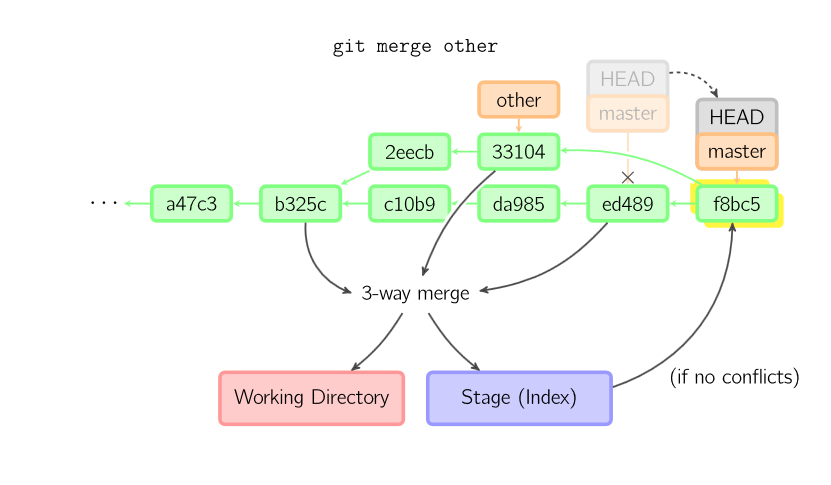


### Merge

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.

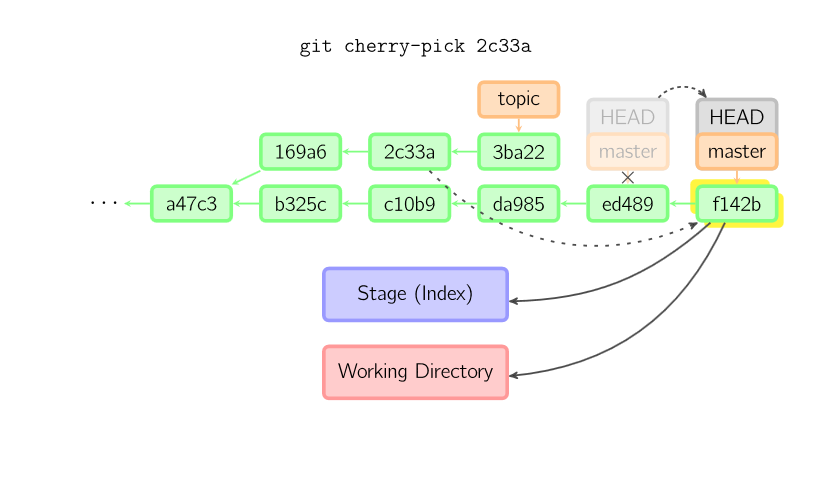


Otherwise, a "real" merge must occur. You can choose other strategies, but the default is to perform a "recursive" merge, which basically takes the current commit (ed489 below), the other commit (33104), and their common ancestor (b325c), and performs a [three-way merge](http://en.wikipedia.org/wiki/Three-way_merge). 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.



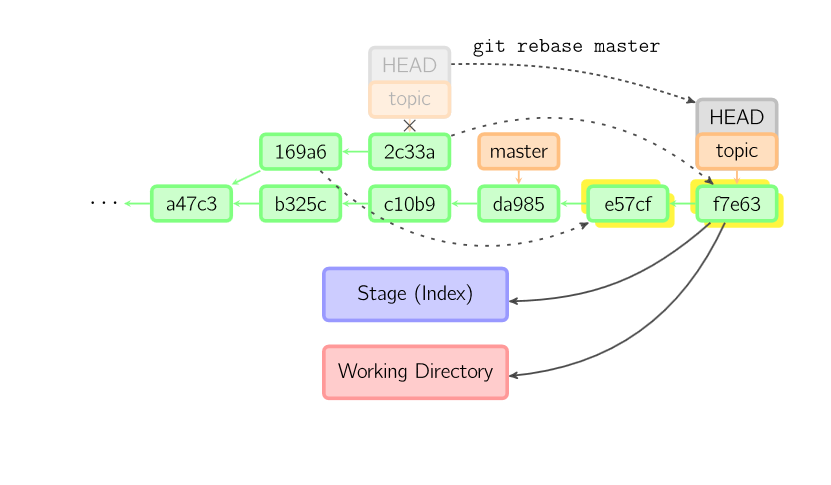
### Cherry Pick

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



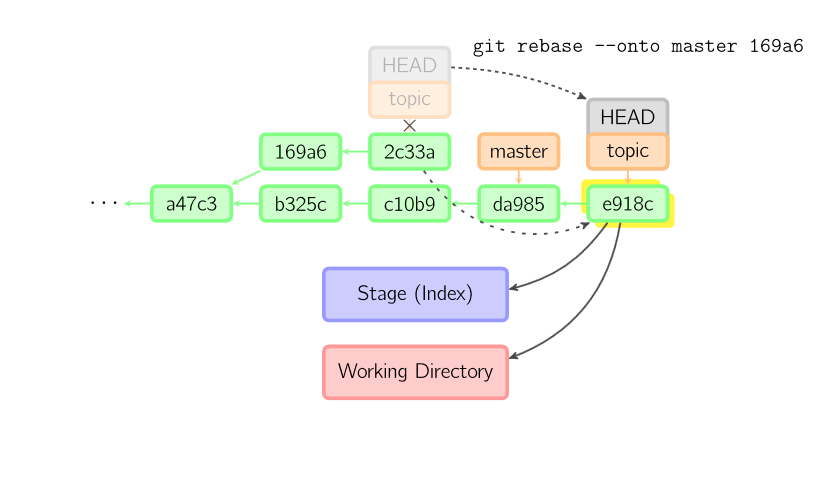
### Rebase

A rebase is an alternative to a [merge](http://marklodato.github.io/visual-git-guide/index-en.html#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-pick](http://marklodato.github.io/visual-git-guide/index-en.html#cherry-pick)s 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)](http://www.kernel.org/pub/software/scm/git/docs/git-rebase.html#_interactive_mode) for more details.

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](http://onlywei.github.io/explain-git-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

<https://davidwalsh.name/tutorials/git>

How to Batch Update Git Commit Messages

**Prepending to Commit Messages**

To prepend text to every commit message in a given range, you'd execute a message like:

git filter-branch --msg-filter 'echo "bug ###### - \c" && cat' master..HEAD

You can also sed to achieve this:

git filter-branch -f --msg-filter 'sed "s/^/bug ###### - /"' master..HEAD

## Appending to Commit Messages

The case for appending to commit messages could be where you want to add the reviewer name(s) to the message.  Appending is roughly the same:

git filter-branch -f --msg-filter 'cat && echo "[Reviewer Walsh]"' master..HEAD

Quick git Commit Searching

One frequent git task is searching a list of commits on master branch which match a given keyword.  Here's how I do that:

git log -i --grep='ckeditor'

Create a Repository Archive with git

By [David Walsh](http://davidwalsh.name) on September 28, 2015

One feature I recently found out about is its archive feature which allows for exporting an entire repository to a zip or tar file.

# Format: git archive {branchname} --format={compression} --output={filename}

git archive master --format=tar --output=kuma.tar

git archive some-feature-branch --format=tar --output=kuma.tar

Sure you could use any archiving utility to archive a given repo, this feature allows for quick archiving of any branch or repository state!

List Recent git Commits from Command Line

By [David Walsh](http://davidwalsh.name) on January 5, 2016

.  I found a useful command for listing commits newest to oldest so on [commandlinefu](http://www.commandlinefu.com/commands/view/15064/show-a-git-log-with-offsets-relative-to-head):

git log --oneline | nl -v0 | sed 's/^ \+/&HEAD~/'

That command will render a listing with the latest commits first and original commit last:

Track Empty Directories with git

# Track Empty Directories with git

By [David Walsh](http://davidwalsh.name) on May 30, 2017

There are times when you'd like to track an empty directory within git but there's a problem: git wont allow you to add a directory that doesn't have a file in it.  The easy solution is putting an empty stub file within the directory, and the industry standard for that stub file name is .gitkeep

You can quickly create the file and commit the "empty" directory from command line:

touch my-empty-dir/.gitkeep

git add my-empty-dir/.gitkeep

git commit -m "Adding my empty directory"

The problem is simple, the solution is easy, but I wanted to highlight that .gitkeep is the industry standard.

Checkout the Previous Branch with git

By [David Walsh](http://davidwalsh.name) on June 29, 2017

I recently found out that you can switch the the previous branch you were on using the following command:

git checkout master

# Do whatever

git pull remote master

# Go back to the previous branch

git checkout -

Using - references the previous branch name, thus allowing you to navigate branches with ease!

Undo File Changes with Git

# Undo File Changes with Git

By [David Walsh](http://davidwalsh.name) on July 10, 2017

* To quickly undo file changes with git, execute the following two commands:
* git reset HEAD path/to/file.ext
* git checkout path/to/file.ext
* The second command (checkout) is required or you'll still see the file listed when running git status again.  With both of those executions, you'll no longer see the file listed with git status.
* git makes version control easy but the two steps needed to essentially revert changes to a file aren't intuitive, thus I thought I would share on this blog.  Happy coding!
* How to Delete a git Remote Branch

# How to Delete a git Remote Branch

* By [David Walsh](http://davidwalsh.name) on March 4, 2018

Once a branch is merged, for example, [we no longer need it around](https://davidwalsh.name/delete-merged-branches-git).

Deleting a branch on a local host machine repo is easy:

git branch -d <branch\_name>

To remove a branch from the remote git repository, like a GitHub-hosted repository, you can execute:

git push <remote\_name> --delete <branch\_name>

Reset File Changes with git

# Reset File Changes with git

By [David Walsh](http://davidwalsh.name) on April 24, 2018

You can restore a file's contents before a patch with the following:

git reset origin/master path/to/file-to-be-changed.ext

Once this shell snippet is executed, the file's contents are restored and can be re-commited to restore the file contents.

Search Git Commits Between Dates

# Search Git Commits Between Dates

By [David Walsh](http://davidwalsh.name) on July 11, 2018

You can use the following git command to list commits between two dates:

$ git log --after="2018-06-30" --before="2018-07-03" --oneline

Especially nice is the --oneline modifier to keep the commit list concise.

Links

<https://www.alexkras.com/category/git/>