Tv's cobweb: Git for Computer Scientists

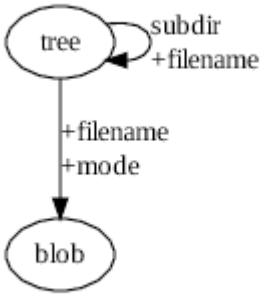
Abstract

Quick introduction to git internals for people who are not scared by words like Directed Acyclic Graph.

Storage

In simplified form, git object storage is "just" a DAG of objects, with a handful of different types of objects. They are all stored compressed and identified by an SHA-1 hash (that, incidentally, *isn't* the SHA-1 of the contents of the file they represent, but of their representation in git).

`blob`: The simplest object, just a bunch of bytes. This is often a file, but can be a symlink or pretty much anything else. The object that points to the `blob` determines the semantics.



`tree`: Directories are represented by `tree` object. They refer to `blob`s that have the contents of files (filename, access mode, etc is all stored in the `tree`), and to other `tree`s for subdirectories.

When a node points to another node in the DAG, it *depends* on the other node: it cannot exist without it. Nodes that nothing points to can be garbage collected with `git gc`, or rescued much like filesystem inodes with no filenames pointing to them with `git lost-found`.

About
About me and this site

Blog
A journal of my thoughts and things in my life

Articles
Collection of articles I've published

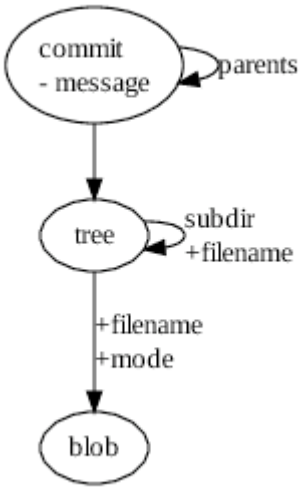
Talks
Presentations I have delivered recently

Software
Software I have written

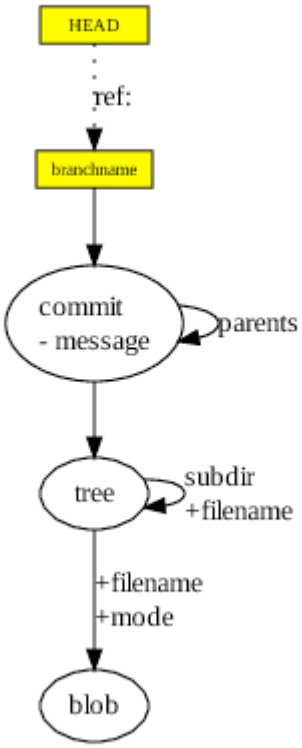
Links
My link collection

Photos
My picture collection, featuring the two chihuahua dogs Nano and Pico, totalling just over 4kg.

Videos
My video collection, starring the dogs again. Hosted by Revver.



`commit`: A `commit` refers to a `tree` that represents the state of the files at the time of the commit. It also refers to 0..n other `commits` that are its parents. More than one parent means the commit is a merge, no parents means it is an initial commit, and interestingly there can be more than one initial commit; this usually means two separate projects merged. The body of the `commit` object is the commit message.

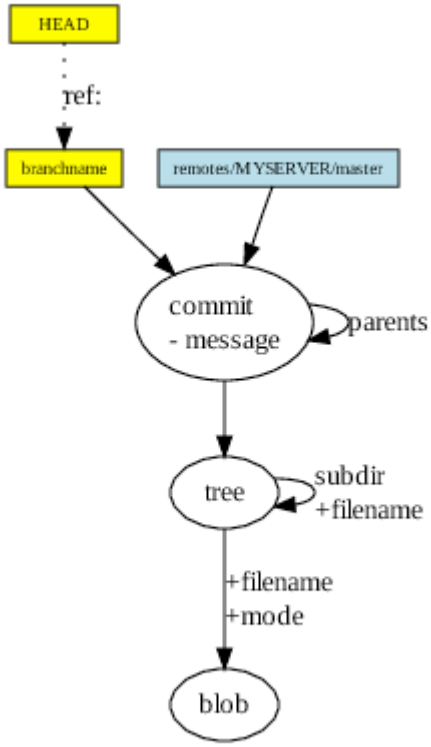


refs: References, or heads or branches, are like post-it notes slapped on a node in the DAG. Where as the DAG only gets added to and existing nodes cannot be mutated, the post-its can be moved around freely. They

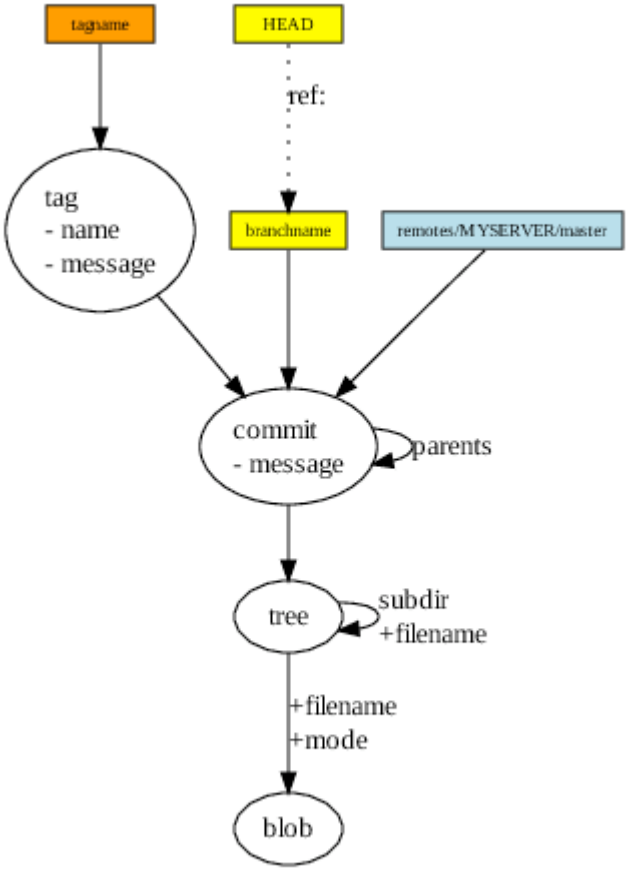
don't get stored in the history, and they aren't directly transferred between repositories. They act as sort of bookmarks, "I'm working here".

`git commit` adds a node to the DAG and moves the post-it note for current branch to this new node.

The `HEAD` ref is special in that it actually points to another ref. It is a pointer to the currently active branch. Normal refs are actually in a namespace `heads/XXX`, but you can often skip the `heads/` part.



remote refs: Remote references are post-it notes of a different color. The difference to normal refs is the different namespace, and the fact that remote refs are essentially controlled by the remote server. `git fetch` updates them.



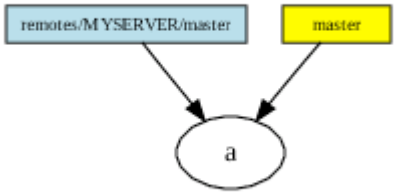
`tag`: A `tag` is both a node in the DAG and a post-it note (of yet another color). A `tag` points to a `commit`, and includes an optional message and a GPG signature.

The post-it is just a fast way to access the tag, and if lost can be recovered from just the DAG with `git lost-found`.

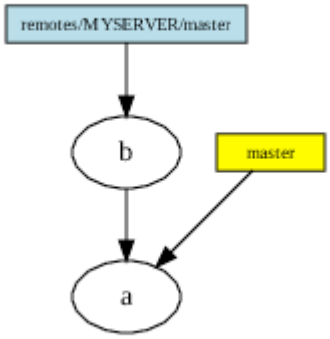
The nodes in the DAG can be moved from repository to repository, can be stored in more effective form (packs), and unused nodes can be garbage collected. But in the end, a `git` repository is always just a DAG and post-its.

History

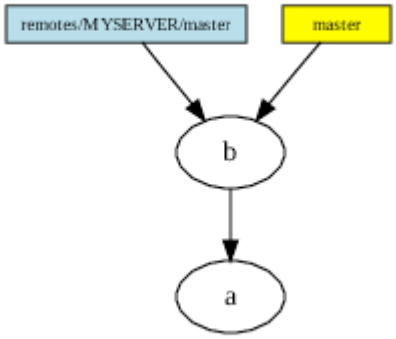
So, armed with that knowledge on how `git` stores the version history, how do we visualize things like merges, and how does `git` differ from tools that try to manage history as linear changes per branch.



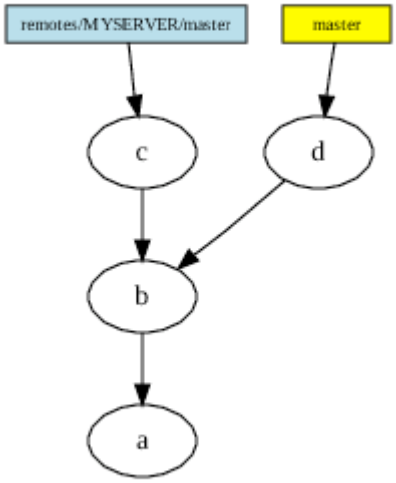
This is the simplest repository. We have `cloned` a remote repository with one commit in it.



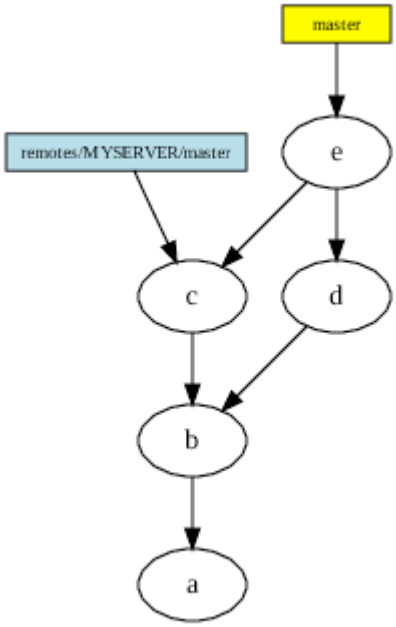
Here we have `fetched` the remote and received one new commit from the remote, but have not merged it yet.



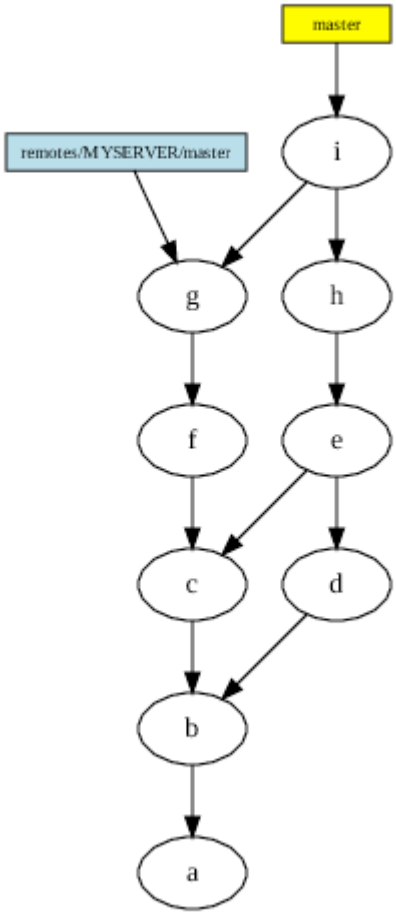
The situation after `git merge remotes/MYSERVER/master`. As the merge was a fast forward (that is, we had no new commits in our local branch), the only thing that happened was moving our post-it note and changing the files in our working directory respectively.



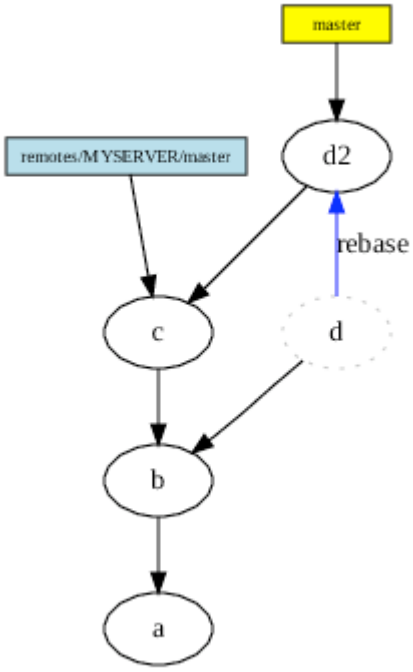
One local `git commit` and a `git fetch` later. We have both a new local commit and a new remote commit. Clearly, a merge is needed.



Results of `git merge remotes/MYSERVER/master`. Because we had new local commits, this wasn't a fast forward, but an actual new `commit` node was created in the DAG. Note how it has two parent `commits`.



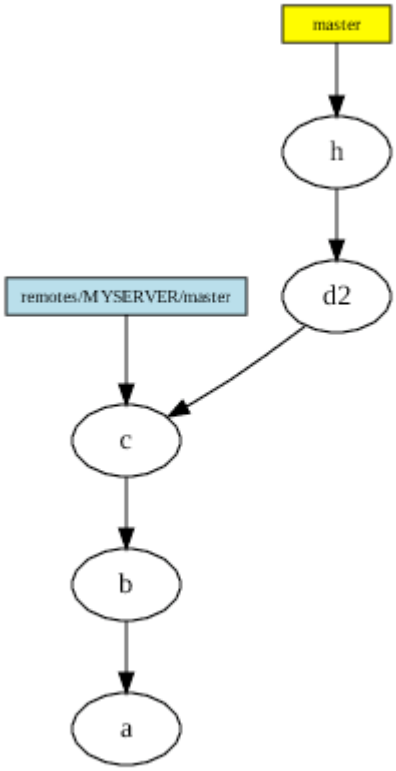
Here's what the tree will look after a few commits on both branches and another merge. See the "stitching" pattern emerge? The `git` DAG records exactly what the history of actions taken was.



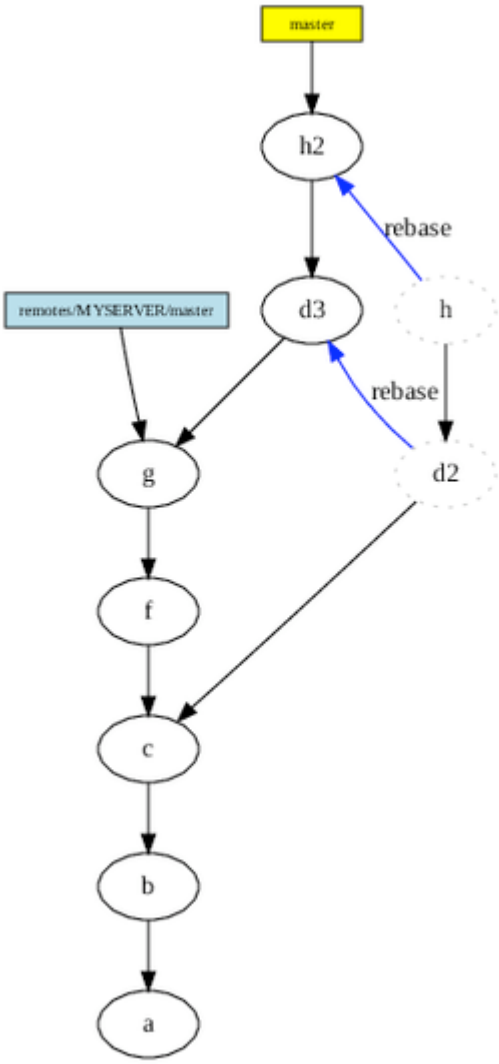
The "stitching" pattern is somewhat tedious to read. If you have not yet published your branch, or have clearly communicated that others should not base their work on it, you have an alternative. You can `rebase` your branch, where instead of merging, your commit is replaced by another commit with a different parent, and your branch is moved there.

Your old commit(s) will remain in the DAG until garbage collected. Ignore them for now, but just know there's a way out if you screwed up totally. If you have extra post-its pointing to your old commit, they will remain pointing to it, and keep your old commit alive indefinitely. That can be fairly confusing, though.

Don't rebase branches that others have created new commits on top of. It is possible to recover from that, it's not hard, but the extra work needed can be frustrating.



The situation after garbage collecting (or just ignoring the unreachable commit), and creating a new commit on top of your `rebased` branch.



`rebase` also knows how to rebase multiple commits with one command.

That's the end of our brief intro to `git` for people who are not intimidated by computer science. Hope it helped!

Powered by atomat.