Intermediate Git Webinar

Date: July 12, 2017
Presented by: Roscoe Bartlett, Sandia National Laboratories

The presentation material along with the video can be found in two places:
https://exascaleproject.org/event/intermediate-git/
https://ideas-productivity.org/events/

The material below reflects questions that were asked and answered during the Intermediate Git Webinar on July 12, 2017.

Q: How large is too large of a binary for git?

From Matt Belhorn: We can have the speaker address this at the next pause, but I recommend not committing any binary files that will change over the lifetime of the repo. If the binary is larger than a few megabytes and changes frequently, the repo can quickly consume a relatively large amount of space if the file changes over time. There are ways to work around the problem with things like git LFS.

From Ross Bartlett: I agree with Matt. I will also add that if the binary files are small compared to the rest of the repo and don't change much, it is not that big of a deal. But note that GitHub has a hard limit of 100M for single files and will give annoying warnings if files are larger than 50M.

Q: How far back the reflog will go?

From Jason Gates: Ah, default for git gc is to keep things for the past two weeks, but that can be overridden with --prune=<date>

From Matt Belhorn: Also, the `git reflog` options `--expire=<time>` and `--expire-unreachable=<time>` will manually prune entries.

From Ross Bartlett: But `git gc` does not run that often.

Q: What would you recommend for repositories that already have had large binaries added to history? I've already rewritten history for one of the affected branches, but am loathe to doing that for history that is common to all branches. I think I know the answer, but is there any way to remove trash (binaries) without changing the commit hash of commits that build upon problem commits?

Unfortunately no, if you remove files you have changed what git uses to create the sha1 so necessarily the commit's sha1 and all commits after will be changed. If you want to avoid rewriting history you can delete the file from the repo so as not to compound the "error." But the fill will still exist in history and be taking up space there. It won't continually add to the problem in that case.

From Ross Bartlett: Any filtering always changes the SHA1s (remember, git never ever edits existing commits). What you can do is to filter all of the branches at once and the push the new branches and then have everyone do `git fetch ; git reset --hard @{u}`. But before you do that, you need all of your developers to push every branch they are working on everywhere (but not merge them). That way, you can update everyone all at once. But is HUGE disruption to the entire developer and user community. That is why there is "H) Don't commit large generated (binary) files in a git repo." in https://ideas-productivity.org/resources/howtos/git-tutorial-and-reference-collection/beginner-tips/

Q: Do we have any documentation like "best practices for Git Workflow for scientific code"?

There is some freedom in the Git Workflow and philosophical disagreements as to how that workflow should look like. Sometimes, trivials things can become overly complex.
Here is a reference for this:
https://ideas-productivity.org/wordpress/wp-content/uploads/2016/12/IDEAS-VCHowToVersionControlwithGit-V0.2.pdf

From Ross Bartlett: That above howto links to a document "Design Patterns for Git Workflows" that tries to define git workflow best practices so you can build the best workflow for your current situation. (The will be published as an official document soon but any feedback you have would be helpful.)

Q: What is the best way to refer to versions of code in a lab notebook? My lab has been referring to versions of code used to create results with commit hashes; wondering if there are any other preferred workflows.

You can tag commits that you wish to reference, but be sure to never change or delete the tag. Likewise, commit hashes will change if the code is ever rebased so if you choose to reference code by commit hash, be careful with rebasing. Another option might be to branch out states that you want to reference in a lab notebook. If using github, another option would be to tag a release which will preserve the repo as an archive.

The best way to uniquely identify the version of the code for reproducibility is to run and record the output from `git log --pretty=fuller -1`. That will show the author, the committer, the author date, the commit date, and the commit summary. Even if the repo is later filtered (changing all the SHA1s as mentioned above), you can usually identify the same commit using a combination of the summary message from those fields.   In fact, the commit date does not change when you filter a branch so a combination commit date, author and the summary should be enough.

git pull --rebase is same as git rebase origin/master?

`git pull --rebase` is equivalent to `git fetch origin/master; git rebase origin/master`

what happens when rebase has conflicts? (same process as merge?)

Yes, it will be handled as a normal merge-conflict resolution

Q: Can you go over when it's better to do rebase vs. merge? Or is it always better to do rebase? if you can do it.

As the speaker mentioned, it is difficult to offer an absolute response to this. I would say, however, it is *not* a good idea to rebase commits that have been pushed to a public repo. If changes that would be rebased have already been pushed, a merge is better to sync the local and remote changes. If the changes have not been pushed, I (Matt Belhorn) prefer to rebase.

From Ross Bartlett: You will use a merge or a rebase depending on your adopted workflow and the stage of the workflow. For example, you will see steps where a rebase is used in some steps and an explicit merge commit is used in other steps of a topic branch workflow in the section "addition of topic branches" and in Simple Git Workflow is Simple.

Q: In resolving conflicts if we find there is too much work, is there a way to quit and get out of the merge?

For both the merge and rebase you can use the --abort option to quit what is currently being done and put you back where you were before running the command.

Q: What if I don't want to "merge" changes for a given file but rather want to replace the file with the new contents from the other branch (e.g. log files and other outputs that can't be merged per se but need to be refreshed entirely)? Equivalent to svn resolve -at?

If you just want to get the one file onto the current branch you can use "git checkout <branch name> -- <filename>" This will checkout the file as it exists at the branch name (you could use a sha1 in place of branch name). You would then commit the file as if you'd made the changes by hand. If you want to merge other files in addition and don't want to resolve the conflict(s) by hand and are sure that the file at a specific point in history is exactly what you want then you can use git checkout as above to resolve the conflict. You would still need to use git add to mark the file as resolved and continue with the merge or rebase operation as normal.