Karlsruher Institut für Technologie

KIT | NHR
National High-Performance Computing Center

Introduction to Git

Michele Mesiti, PhD | 18 April 2023

Outline

1. Additional Git Tools

2. More Good Practices for You to Consider

3. Additional Git Features you might find useful

Additional Git Tools: Difftools

The plain `git diff` command works reasonably well for plain text files and code.
For non plain-text files, specific tools exist to make the differences easier to see:

- Jupyter Notebooks: nbdime
- LaTex: latexdiff
- there are syntax-aware diff tools, diff tools for images...

Additional Git Tools: mergetools

Merging can be difficult at times.
Merge tools can give you a better view of the changes that were introduced the two branches you are trying to merge, and show the state of the file at the latest common commit.

Good practices: General

- Think carefully about commit messages - **Benefit:** faster understanding of what changed in each commit
- Keep your commit history tidy (see the `git rebase` command to have more chances to do that) - **Benefit:** faster understanding of what changed in each commit
- Keep your files small and compose them by `import` or `include` et simila - **Benefit:** your repository will stay small if you change these files
- If you use branches, merge the changes often back to `main` - **Benefit:** less stale branches you need to track/remember of in your head, more informative `git branch` output, easier management of merges

Good practices: Text documents

- When writing text: keep your lines short (use semantic linebreaks) - **Benefit:** given how `git diff` works, spotting small changes in your files will be easier

Good practices: Source Code

- **Use automatic formatting** (e.g., black, clang-format and so on - find one for the language you are using) In this way, the output of `git diff` will be most informative
- Merging is no magic and can actually silently break your code *without conflicts being triggered*. **Do review code before merging it**. An **automatic test suite** greatly helps with spotting issues that can be generated while merging
- Use git hooks for automatic formatting and linting (for real test suites, consider CI/CX)

Good practices: Forks and pull/merge requests

If Bob wants to collaborate on Alice's project, these are more robust workflows.
Either

1. Bob **Forks** the repository on GitHub/GitLab/BitBucket etc., i.e. makes a remote copy of Alice's repository that he owns
2. Bob pushes changes to his remote repository
3. Bob requests that Alice pulls from their repository, with a **pull request**.

or

1. Bob clones Alice's repository on GitHub/GitLab/BitBucket etc., but he is not allowed to work on `main`, which is owned by Alice
2. Bob pushes changes to a branch that he owns on Alice's repository
3. Bob requests that Alice merges his branch into `main` with a **merge request**.

Additional Git Features you might find useful

A list of suggestions for you to consider, in an arbitrary order.
Exercise at the end.

More Undo Power: `git restore`/`git reset`

How to undo `git add <filename>`? `git restore --staged <filename>`
(in older version of Git, you would use `git reset <filename>`)
To restore a file ...

- ...in the index from a commit: `git restore --source <commit> --staged <filename>`
- ...in the working tree from a commit: `git restore --source <commit> <filename>`
- ...in the working tree from the index: `git restore <filename>`

(default behaviour: omitting `--source <commit>` implies `--source HEAD`)
Question: Is this command available in your version of Git? What does the man page say?

More Undo Power: `git commit --amend`

Forgot to add something in the last commit? A typo in the commit message? `git commit --amend` is your friend (*but only before you push... otherwise, what happens?*)

Finesse with `git add`

Interactively fine-tune changes to stage in a long file, which has more independent changes:
`git add -p <filename>`
(long option is `--patch`)
This allows for a less strict discipline in modifying your code.
(The `-p` option works as well for `git restore`)
Note: editor support can make this very convenient

Alternative to Git Merge: Rebase

- Alternative to merging, to include changes from other lines of work.
- Might result in a cleaner commit history.
- The Gist of it: cut off the branch you are working on and stick it on top of another branch (or commit)
- with *interactive rebasing* (`git rebase -i`), you can clean up your commit history, deleting/squashing useless commits, also change the commit messages.
- this command *rewerites history*, fact which entails the...
- Golden Rule of Rebasing: do not do it to Public Branches - someone might have pulled that already
- Conflicts are still possible

Yet another alternative to Git Merge: Cherry-pick

- Alternative to merging and rebasing, to include changes from other lines of work
- Apply the changes introduced by some existing commits you explicitly choose
- Conflicts are still possible

A `checkout` history: The Reflog

If you have been jumping between different branches with `git checkout` for whatever reason (bug fixing, comparison between branches, you name it) and wish to remember which branches or tags you visited but you can't, `git reflog` gives you a history of the movements of `HEAD` (or any reference you choose).

Find the commit that broke the code: Bisect

Find the commit where a particular “thing” changed, e.g. something broke.

1. Start with a “good” and a “bad” commit, which are usually far apart
2. Then use a interactive bisection method to find which commit broke your code (marking each visited commit as bad or good with `git bisect bad` or `git bisect good`, this can be also scripted)

Find who broke the code (and when): Blame

Easily find who changed which line of a file and when:
`git blame <filename>`
(Who broke the code?)

Stash away your work temporarily: Git Stash

Do you need to quickly drop whatever you are doing and switch to a particular commit/branch?
Use `git stash` to store your changes temporarily on top of a stack of changes, ending up with a clean repository status.
(a better alternative to `git switch -c tempbranch; git add -u; git commit -m 'temp commit'`).
Retrieve your stashed changes with `git stash pop`.

Large File Support

Git works best with small text files. What about large binary files?
Some tools might help for that use case!
Alternatives:

- git LFS: simpler workflow (use `git add` normally after proper initialization & configuration)
- git annex: feature rich, full decentralized archiving, manual sync by default, (use separate `git annex add` command after proper initialization & configuration)

Summary:

- By default, save only “pointers” to files
- Recover real file only when needed
- Tell collaborators to do the tutorial

When the repository is huge: Partial Cloning

Imagine you need to clone a repository with a huge history that could be several GB, possibly while traveling on a train with an unstable, slow internet connection
If you wish to work, at least at the beginning, only on the latest versions, you can tell Git to leave out all the files by default (file=*blob*, in Git term) and get them only when needed
(e.g., when you check out a particular commit, the files in the state at that commit only will be downloaded).
Use `--filter=blob:none` ([1]).

[1] You might have seen the `--depth=N` option, but that makes the commits before the last N completely unreachable.

Automatic workflows/pipelines/CI

“Git Forges” offer workflow automation services related to Git repositories (i.e., GitHub Actions, GitLab CI, BitBucket Pipelines) that can be used to do things like

- Build your software or documents (produce *artifacts*)
- publish/upload them somewhere
- run tests on your software

These tasks are run automatically and are triggered by events like

- pushes to specific branches
- pull/merge requests

Exercise and Feedback

Review the list above.

1. For each Git command that you think might be useful, look up the man page with `git help <command>` (or search the Internet for it).
2. Discuss with your neighbours, and write the most interesting one in the pad!
3. Do you have any additions to suggest?
4. Also have a look at this for typical problems and solutions.