Git Best Practices (转载)

Copyright

This is a fairly common question, and there isn't a One True Answer, but still, this represents a consensus from #git

TL; DR

全文很长，因此把最重要的部分放到了最前面：什么是不该做的。尤其是第一条：不要提交任何可以被生成的文件——`.o`，`.class`，`.pdf`，`.swp`，…

Don't

In this list of things to *not* do, it is important to remember that there are legitimate reasons to do all of these. However, you should not attempt any of these things without understanding the potential negative effects of each and why they might be in a best practices "Don't" list.

DO NOT

- commit anything which can be regenerated from other things than were committed.
- commit configuration files
  Specifically configuration files which might change from environment to environment or for any reasons. See Information about local versions of configuration files (https://gist.github.com/1423106)
- use git-grafts
  This is deprecated in favor of git-replace.
- use git-replace
  But don't use git-replace either.
- rewrite public history
  See section about this topic
- change where a tag points
  This is another way to rewrite public history.
- use git-filter-branch
  Still another way to rewrite public history.
- use clone --shared or --reference
  This can lead to problems for non-normal git actions, or if the other repository is deleted/moved. See git-clone manual page (http://jk.gs/gitworkflows.html).
- use reset (--hard || -merge) without committing/stashing
  This can often overwrite the working directory without hope of recourse.
- use checkout in file mode
  This will overwrite some (or potentially all with .) of the working directory without hope of recourse.
- use `git clean` without previously running with "-n" first
  This will delete untracked files without hope of recourse.
- prune the reflog
  This is removing your safety belt.

- expire "now"
  This is cutting your safety belt.
- use git as a backup tool
  Yes people have done it successfully, but usually with lots of scripts around it and with some toothing pains. Git was not written as a dedicated backup tool, and such tools do exist.
- use git as a web deployment tool
  Yes it can be done in a sufficiently simple/non-critical environment with something like http://toroid.org/ams/git-website-howto to help. However, this does not give you atomic updates, synchronized db updates, or other accouterments of an industrial deployment system.
- commit large binary files (when possible)
  Large is currently relative to the amount of free RAM you have. Remember that not everyone may be using the same memory configuration you are.
  Consider using Git annex (http://git-annex.branchable.com/) or Git media (https://github.com/schacon/git-media)
- create very large repositories (when possible)
  Git can be slow in the face of large repositories. There are git-config options which can help. `pack.threads=1; pack.deltaCacheSize=1; pack.windowMemory=512m; core.packedGitWindowSize=16m; core.packedGitLimit=128m.` Other likely ones exist.

Read about git

Knowing where to look is half the battle. I strongly urge everyone to read (and support) the Pro Git book. The other resources are highly recommended by various people as well.

- Pro Git (http://progit.org/book/)
- Git for Computer Scientists (http://eagain.net/articles/git-for-computer-scientists/)
- Git from the Bottom Up (http://ftp.newartisans.com/pub/git.from.bottom.up.pdf)
- Git for Web Designers (http://www.webdesignerdepot.com/2009/03/intro-to-git-for-web-designers/)
- Other resources (http://git-scm.com/documentation)
- Git wiki (http://git.wiki.kernel.org/)

Commit early and often

Git only takes full responsibility for your data when you commit. If you fail to commit and then do something poorly thought out, you can run into trouble. Additionally, having periodic checkpoints means that you can understand how you broke something.

People resist this out of some sense that this is ugly, limits `git-bisect`ion functionality, is confusing to observers, and might lead to accusations of stupidity. Well, I'm here to tell you that resisting this is ignorant. *Commit Early And Often*. If, after you are done, you want to pretend to the outside world that your work sprung complete from your mind into the repository in utter perfection with each concept fully thought out and divided into individual concept-commits, well git supports that: see Sausage Making below. However, don't let tomorrow's beauty stop you from performing continuous commits today.

Personally, I commit early and often and then let the sausage making be seen by all except in the most formal of circumstances. Just look at the history of this gist!

Don't panic

As long as you have committed your work (or in many cases even added it with `git add`) your work will not be lost for at least two weeks unless you really work at it (run commands which manually purge it).

When attempting to find your lost commits, first make *sure* you will not lose any current work. You should commit or stash your current work before performing any recovery efforts which might destroy your current work. After finding the commits you can reset, rebase, cherry-pick, merge, or otherwise do what is necessary to get the commit history and work tree you desire.

There are three places where "lost" changes can be hiding. They might be in the reflog (`git log -g`), they might be in lost&found (`git fsck --unreachable`), or they might have been stashed (`git stash list`).

- reflog
  The reflog is where you should look first and by default. It shows you each commit which modified the git repository. You can use it to find the commit name (SHA-1) of the state of the repository before (and after) you typed that command. While you are free to go through the reflog manually, you can also visualize the repository using the following command (Look for dots without children and without green labels):
  ```
  gitk --all --date-order $(git log -g --pretty=%H)
  ```
- Lost and found
  Commits or other git data which are no longer reachable though any reference name (branch, tag, etc) are called "dangling" and may be found using fsck. There are legitimate reasons why objects may be dangling through standard actions and normally over 99% of them are entirely uninteresting for this reason.
  - Dangling Commit
    These are the most likely candidates for finding lost data. A dangling commit is a commit no longer reachable by any branch or tag. This can happen due to resets and rebases and are normal. `git show SHA` will let you inspect them.
    The following command helps you visualize these dangling commits. Look for dots without children and without green labels.
    ```
    gitk --all --date-order (git fsck --no-reflog | grep "dangling commit"
    | awk '{print 3;}')
    ```
  - Dangling Blob
    A dangling blob is a file which was not attached to a commit. This is often caused by `git add` s which were superceded before commit or merge conflicts. Inspect these files with
    ```
    git show SHA
    ```
  - Dangling Tree
    A dangling tree is a directory tree of files that was not attached to a commit. These are rarely interesting, and often caused by merge conflicts. Inspect these files with `git ls-tree -r SHA`
  - Stashes
    Finally, you may have stashed the data instead of committing it and then forgotten about it. You can use the `git stash list` command or inspect them visually using:
    ```
    gitk --all --date-order (git stash list | awk -F: '{print 1};')
    ```
  - Misplaced
    Another option is that your commit is not lost. Perhaps the commit was just made on a different branch from what you remember. Using `git log -Sfoo --all` and `gitk --all --date-order` to try and hunt for your commits on known branches.
  - Look elsewhere
    Finally, you should check your backups, testing copies, ask the other people who have a copy of the repo, and look in other repos.

Backups

Everyone always recommends taking backups as best practice, and I am going to do the same. However, you already may have a highly redundant distributed ad-hoc backup system in place! This is because essentially every clone is a backup. In many cases, you may want to use a clone for git experiments to perfect your method before trying it for real (this

is most useful for `git filter-branch` and similar commands where your goal is to permanently destroy history without recourse—if you mess it up you may not have recourse). Still, perhaps you want a more formal system.

Traditional backups are still appropriate. A normal tarball, cp, rsync, zip, rar or similar backup copy will be a perfectly fine backup. As long as the underlying filesystem doesn't reorder git I/O dramatically, there is not a long time delay between the scanning of the directory and the retrieval of the files, the resulting copy of .git should be consistent under almost all circumstances. Of course, if you have a backup from in the middle of a git operation, you might need to do some recovery. The data should all be present though. When performing git experiments involving the working directory, a copy instead of a clone may be more appropriate.

However, if you want a "pure git" solution, something like, which clones everything in a directory of repos, this may be what you need:

```
cd /src/backupgit
ls -F . | grep / > /tmp/.gitmissing1
ssh -n git.example.com ls -F /src/git/. | grep / > /tmp/.gitmissing2
diff /tmp/.gitmissing1 /tmp/.gitmissing2 | egrep '^>' |
  while read x f; do
    git clone --bare --mirror ssh://git.example.com/src/git/$$f $$f
  done
rm -f /tmp/.gitmissing1 /tmp/.gitmissing2
for f in */.; do (cd $$f; echo $$f; git fetch); done
```

Don't change published history

Once you `git push` (or in theory someone pulls from your repo, but people who pull from a working repo often deserve what they) your changes to the authoritative upstream repository or otherwise make the commits or tags publicly visible, you should ideally consider those commits etched in diamond for all eternity. If you later find out that you messed up, make new commits which fix the problems (possibly by revert, possibly by patching, etc).

Yes, of course git allows you to rewrite public history, but it is problematic for everyone and and thus it is just not best practice to do so.

Choose a workflow

Some people have called git a tool to create a SCM workflow instead of an SCM tool. There is some truth to this.

Branch workflows

Answering the following questions helps you choose a branch workflow:

- Where do important phases of development occur?
- How can you identify (and backport) groups of related change?
- What happens when emergency patches are required?

See the following references for more information on branch workflows.

- Pro Git branching models (http://progit.org/book/ch3-4.html)
- Git-flow branching model (http://nvie.com/posts/a-successful-git-branching-model/)
  With the associated gitflow tool (https://github.com/nvie/gitflow)
- Gitworkflows man page (http://jk.gs/gitworkflows.html)
- A Git Workflow for Agile Teams (http://reinh.com/blog/2009/03/02/a-git-workflow-for-agile-teams.html)
- What git branching models actually work (http://stackoverflow.com/questions/2621610/what-git-branching-models-actually-work)

- Our New Git Branching Model (http://blogs.remobjects.com/blogs/mh/2011/08/25/p2940)

However, also understand that everyone already has an implicit private branch due to their cloned repository: they can do work locally do a `git pull --rebase` when they are done, perform final testing, and then push their work out. If you run into a situation where you might need the benefits of a feature branch before you are done, you can even retroactively commit&branch then optionally reset your primary branch back to @{u}. Once you push you lose that ability.

Some people have been very successful with just master and RELEASE branches (RELEASE branch for QA and polishing, master for features, specific to each released version.) Other people have been very successful with many feature branches, integration branches, QA, and release branches. The faster the release cycle and the more experimental the changes, the more branches will be useful—continuous releases or large refactoring project seem to suggest larger numbers of branches (note the number of branches is the tail, not the dog: more branches will not make you release faster).

Oh, and decide branch naming conventions. Don't be afraid of / in the branch name when appropriate.

Distributed workflows

Answering the following questions helps you choose a distributed workflow:

- Who is allowed to publish to the master repository?
- What is the process between a developer finishing coding and the code being released to the end-user?
- Are there distinct groups which work on distinct sections of the codebase and only integrate at epochs? (Outsourcing)
- Is everyone inside the same administrative domain?

See the following references for more information on distributed workflows.

- Pro Git distributed models (http://progit.org/book/ch5-1.html)
- Gitworkflows man page (http://jk.gs/gitworkflows.html)

Cathedrals (traditional corporate development models) often want to have (or to pretend to have) the one true centralized repository. Bazaars (linux, and the Github-promoted workflow) often want to have many repositories with some method to notify a higher authority that you have work to integrate (pull requests).

However, even if you go for, say, a traditional corporate centralized development model, don't forbid self-organized teams to create their own repositories for their own tactical reasons. Even having to fill out a justification form is probably too cumbersome.

Release workflow

Deciding on your release workflow (how to get the code to the customer) is another important area to decide on. I will not touch on this much, but it can have an effect on how you use git. Obviously branching and distributed workflows might affect this, but less obviously, it may affect how and when you perform tagging.

At first glance, it is a no-brainer. When you release something you tag something. However, tags should be treated as immutable once you push. Well, that only makes sense, you might think to yourself. Consider this. Five minutes after everyone has signed off on the 2.0 release, it has been tagged and pushed, but before any customer has seen the resulting product someone comes running in "OMFG, the foobar is broken when you frobnoz the baz." What do you do? Do you skip release 2.0 and tag 2.0.1? Do you do a take-back and go to every repo of every developer and delete the 2.0 tag?

Two ideas for your consideration. Instead of a release tag, use a release branch (and then stop committing to that branch after release, disabling write access to it in gitolite or something). Another idea, use an internal tag name which is not directly derived from the version number which marketing wishes to declare to the outside world. Both are problematic in practice, but less so than pure marketing-version tags.

Security model

You might ask why security is not a top level item and is near the end of the workflow section. Well that is because in an ideal world your security should support your workflow not be an impediment to it.

For instance, did you decide certain branches should only have certain people being allowed to access it? Did you decide that certain repositories should only have certain people able to access/write to them?

While git allows users to set up many different types of access control, access methods, and the like; the best for most deployments might be to set up a centralized git master repository with a gitolite manager to provide fine grained access control with ssh based authentication and encryption.

Of course, security is more than access control. It is also assurance that what you release is what was written by the people it should be written by, and what was tested. Git provides you this for free, but certain formal users may wish to use signed tags. Watch for signed pushes in a future version of git.

Dividing work into repositories

Repositories sometimes get used to store things that they should not, simply because they were there. Try to avoid doing so.

- One conceptual group per repository.
  Does this mean one per product, program, library, class? Only you can say. However, dividing stuff up later is annoying and leads to rewriting public history or duplicative or missing history. Dividing it up correctly beforehand is much better.
- Read access control is at the repo level
  If someone has access to a repository, they have access to the entire repo, all branches, all history, everything. If you need to compartmentalize read access, separate the compartments into different repositories.
- Separate repositories for files which might be needed by multiple projects
  This promotes sharing and code reuse, and is highly recommended.
- Separate repositories for large binary files
  Git doesn't handle large binary files ideally yet and large repositories can be slow. If you must commit them, separating them out into their own repository can make things more efficient.
- Separate repositories for planned continual history rewrites
  You will note that I have already recommended against rewriting public history. Well, there are times when doing that just makes sense. One example might be a cache of pre-built binaries so that most people don't need to rebuild them. Yet older versions of this cache (or at least older versions not at tag boundaries) may be entirely useless and so you want to pretend they never happened to save space. You can rebase, filter, or squash these unwanted commits away, but this is rewriting history and can cause problem. So if you really must do so, isolate these files into a repository so that at least everything else will not be affected.
- Group concepts into a superproject
  Once you have divided, now you need to conquer. You can assemble multiple individual repositories into a superproject to group all of the concepts together to create your unified work.
  There are two main methods of doing this.

- git-submodules
  Git submodules is the native git approach which provides a strong binding between the superproject repository and the subproject repositories for every commit. This leads to a baroque and annoying process for updating the subproject. However, if you do not control the subproject (solvable by "forking") or like to perform blame-based history archeology where you want to find out the absolute correspondence between the different projects at every commit, it is very useful.
- gitslave
  gitslave (http://gitslave.sf.net) is a useful tool to add a subsidiary git repositories to a git superproject when you control and develop on the subprojects at more or less the same time as the superproject, and furthermore when you typically want to tag, branch, push, pull, etc all repositories at the same time. There is no strict correspondence between superproject and subproject repositories except at tag boundaries (though if you need to look back into history you can usually guess pretty well and in any case this is rarely needed).

Useful commit messages

Creating insightful and descriptive commit messages is one of the best things you can do for others who use the repository. It lets people quickly understand changes without having to read code. When doing history archeology to answer some question, good commit messages likewise become very important.

The normal git rule of using the first line to provide a short (72 character) summary of the change is also very good. Looking at the output of `gitk` or `git log --oneline` might help you understand why.

While this touches with the next topic of integration with external tools, including bug/issue/request tracking numbers in your commit messages provides a great deal of associated information to people trying to understand what is going on.

Integration with external tools

- Web views

This is pretty standard stuff, but still a best practice. Setting up a tool like gitweb (or cgit or whatever) to allow URL reference to commits (among other visualization interfaces it provides) gives people a great way to refer to commits in email and conversations. If someone can click on a link vs having to fire up git and pull down the latest changes and start up some visualization tool they are much more likely to help you.

- Bug tracking

Industry best practice suggests that you should have a bug tracking system. Hopefully you do. Well, I'm hear to tell you that integrating your bug tracking system with git makes the two systems one thousand times more effective. Specifically, come up with a standard for tagging commits with bug numbers (eg. "Bug 1234: Adjust the frobnoz down by .5") and then have a receive hook on the upstream repo which automatically appends that commit information to the ticket. If you really love your developers, develop syntax which lets them close the ticket as part of the commit message (eg. "Bug 1235r: Adjust the frobnoz up by .25").

The easier a system is for people to use, the more likely they will use it. Being able to see the context which caused the commit to happen (or contrary-wise, being able to find the commit which solved a problem) is incredibly useful. When you send out your commit announcements, make sure to hyperlink the bug tracker in the commit message, and likewise in the tracker message, hyperlink to the web view of the commit.

Notes: some commits can apply to multiple bugs. Generate a standard and code to handle this standard. Also, if you do hour tracking, you may want a syntax to handle that. (eg. "Bug 12346w/5: Bug 12347rw/3: Adjust the frobnoz up by .3")

- IRC/chat rooms

This is not a global best practice, but for certain sized organizations may be very useful. Specifically, to have a chat room (IRC) to discuss issues and problems, and to have a robot in that chat room to provide assistance. When someone talks about Bug 1234, provide a hyperlink to that ticket. When someone pushes some commits, announce those commits. All sorts of things are possible, but there is a fine line between usefulness and overwhelming noise.

Keeping up to date

This section has some overlap with workflow. Exactly how and when you update your branches and repositories is very much associated with the desired workflow. Also I will note that not everyone agrees with these ideas (but they should!)

- Pulling with --rebase

Whenever I pull, under most circumstances I `git pull --rebase`. This is because I like to see a linear history (my commit came after all commits that were pushed before it, instead of being developed in parallel). It makes history visualization much simpler and `git bisect` easier to see and understand.

Some people argue against this because the non-final commits may lose whatever testing those non-final commits might have had since the deltas would be applied to a new base. This in turn might make git-bisect's job harder since some commits might refer to broken trees, but really this is only relevant to people who want to hide the sausage making. Of course to really hide the sausage making you should still rebase (and test the intermediate commits, if any).

- Rebasing (when possible)

Whenever I have a private branch which I want to update, I use rebase (for the same reasons as above). History is clean and simple. However, if you share this branch with other people, rebasing is rewriting public history and should/must be avoided. You may only rebase commits that no-one else has seen (which is why `git pull --rebase` is safe).

- Merging without speeding

`git merge` has the concept of fast-forwarding, or realizing that the code you are trying to merge in is identical to the result of the code after the merge. Thus instead of doing work, creating new commits, etc, git simply changes the branch pointers (fast forwards them) and calls it good.

This is good when doing `git pull` but not so good when doing `git merge` with a non-@{u} (upstream) branch. The reason this is not good is because it loses information. Specifically it loses track of which branch is the first parent and which is not. If you don't ever want to look back into history, then it does not matter. However, if you might want to say "which branch was this commit originally committed onto," if you use fast-forwarding that question is impossible to answer since git will pick one branch or the other (the first parent or second parent) as the one which both branches activities were performed on and the other (original) parent's branch will be anonymous. There are typically worse things in the world, but you lose information that is not recoverable in any other way by a repository observer and in my book that is bad. Use `git merge --no-ff`

Periodic maintenance

- Clean up your git repro every so often.
- Check your stash for forgotten work (`git stash list`)

Experiment!

When you have an idea or are not sure what something does, try it out! Ideally try it out in a clone or copy so that recovery is trivial. While you can normally completely recover from any git experiment involving data which has been fully committed, perhaps you have not committed yet or perhaps you are not sure whether something falls in the category of "trying hard" to destroy history.

Sausage Making

Some people like to hide the sausage making, or in other words pretend to the outside world that their commits sprung full-formed in utter perfection into their git repository. Certain large public projects demand this, but that is not necessarily a good reason for you to demand this as well.

What is a good reason is if you feel you may be cherry-picking commits a lot (though this too if often a sign of bad workflow). Having one or a small number of commits to pick instead of one here, one there, and half of this other one make your problem much much harder later (and typically will lead to merge conflicts when the doner branch is finally merged in).

Another good reason is to ensure each commit compiles (important for git-bisect), represents a different easily understood concept (important for archeology).

`git rebase -i`, `git add -p`, and `git reset -p` can fix commits up in post-production by splitting different concepts, merging fixes to older commits, etc. See also TopGit (http://repo.or.cz/w/topgit.git) and StGit (http://www.procode.org/stgit/).

Thanks

Thanks to the experts on #git for feedback and ideas.

Comments

Comments and improvements welcome.

Add them below, or discuss with SethRobertson (and others) on #git