Git Best Practices

Tips and Tricks

Chris Sims

Engineering and Computer Science Interest Group
URI Student ACM Chapter

6 March 2013

Outline

The Unofficial Standard

Your commit message should communicate your work: [1]

```
Capitalized, short (50 chars or less) summary

More detailed explanatory text, if necessary.
Wrap it to about 72 characters or so.  In some
contexts, the first line is treated as the subject
of an email and the rest of the text as the body.
The blank line separating the summary from the
body is critical (unless you omit the body
entirely); tools like rebase can get confused if
you run the two together.
```

[1] More details here:
http://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html

Commit Early, Commit Often

- Commits and branches are cheap - use them!

Commit Early, Commit Often

- Commits and branches are cheap - use them!
- Smaller commits allow for separating chunks of work, making later debugging or regression fixing easier

Commit Early, Commit Often

- Commits and branches are cheap - use them!
- Smaller commits allow for separating chunks of work, making later debugging or regression fixing easier
- `git` has functionality that allows you to condense commits that you've made

Commit Early, Commit Often

- Commits and branches are cheap - use them!
- Smaller commits allow for separating chunks of work, making later debugging or regression fixing easier
- `git` has functionality that allows you to condense commits that you've made
- A commit history with more discrete commits allows other contributors to better understand the work completed (Implement feature A vs. all the steps to get there)

Backtracking

`git` has a wealth of tools available to recover past work. Some examples:

- `git log`: Review commits in the repository

Backtracking

`git` has a wealth of tools available to recover past work. Some examples:

- `git log`: Review commits in the repository
- `git reflog`: tracks all actions taken in the repository

Backtracking

`git` has a wealth of tools available to recover past work. Some examples:

- `git log`: Review commits in the repository
- `git reflog`: tracks all actions taken in the repository
- Restore an older version of a file: `git checkout SHA -- path/to/filename`

Backtracking

`git` has a wealth of tools available to recover past work. Some examples:

- `git log`: Review commits in the repository
- `git reflog`: tracks all actions taken in the repository
- Restore an older version of a file: `git checkout SHA -- path/to/filename`
- Revert an entire commit (undo changes): `git revert SHA`

Changing History

Be careful rewriting history - generally the answer is never do this if this history is shared with others. Rewriting history that has been shared with others will often cause them to lose their work. If you haven't pushed your commits, there are some tools available:

- `git commit --amend`: Forgot to add something to that last commit, or typo in the commit message? This adds to the previous commit.
- `git pull --rebase`: Replays your commits on top of what you pull. This keeps a generally linear history, and avoids merge commits.

Choose a Workflow

Pick a general workflow for things like adding features, fixing bugs, etc. Some general ideas:

[2] http://nvie.com/posts/a-successful-git-branching-model/
[3] http://git-scm.com/book/ch3-4.html

Choose a Workflow

Pick a general workflow for things like adding features, fixing bugs, etc. Some general ideas:

- Feature branches: take your work for a new feature into a branch, so that others can continue work in other areas. Pull changes from working branch to stay up-to-date.

[2] http://nvie.com/posts/a-successful-git-branching-model/
[3] http://git-scm.com/book/ch3-4.html

Choose a Workflow

Pick a general workflow for things like adding features, fixing bugs, etc. Some general ideas:

- Feature branches: take your work for a new feature into a branch, so that others can continue work in other areas. Pull changes from working branch to stay up-to-date.
- Keep `master` at the latest stable release, with a `dev` branch for the latest.

[2] http://nvie.com/posts/a-successful-git-branching-model/
[3] http://git-scm.com/book/ch3-4.html

Choose a Workflow

Pick a general workflow for things like adding features, fixing bugs, etc. Some general ideas:

- Feature branches: take your work for a new feature into a branch, so that others can continue work in other areas. Pull changes from working branch to stay up-to-date.
- Keep `master` at the latest stable release, with a `dev` branch for the latest.
- Take a look at established workflows like `git-flow` [2] or Pro Git branching models [3]

[2] http://nvie.com/posts/a-successful-git-branching-model/
[3] http://git-scm.com/book/ch3-4.html

Final Tips

- `git stash` should be part of your toolkit - `man git-stash`

[4] http://git-scm.com/book/

Final Tips

- `git stash` should be part of your toolkit - `man git-stash`
- Not quite sure about commands, or feeling nervous? Make an actual backup of your files and the `.git` directory, so you can restore if you mess it up.

[4] http://git-scm.com/book/

Final Tips

- `git stash` should be part of your toolkit - `man git-stash`
- Not quite sure about commands, or feeling nervous? Make an actual backup of your files and the `.git` directory, so you can restore if you mess it up.
- Read the manual! There is documentation all over the place, and the Pro Git book [4] is an outstanding resource.

[4] http://git-scm.com/book/

Final Tips

- `git stash` should be part of your toolkit - `man git-stash`
- Not quite sure about commands, or feeling nervous? Make an actual backup of your files and the `.git` directory, so you can restore if you mess it up.
- Read the manual! There is documentation all over the place, and the Pro Git book [4] is an outstanding resource.
- Practice! Even if your repository never leaves your machine, you can still practice your git workflow and the various commands.

[4] http://git-scm.com/book/