Practical Versionning with Git

Guille Polito, Stéphane Ducasse

July 10, 2019

Layout and typography based on the sbabook LaTeX class by Damien Pollet.

Contents

Illustrations

CHAPTER 1

Practical Git by Scenarios

1.1 Exploring the History

The `git log` command

The commit graphs we have shown so far are not evident at all while when we use the `git status` command. There is however a way to ask Git about them using the `git log` command.

```
$ git log
commit 0c0e5ff55b56fe8eabc1661a1da64b41f9d74472
Author: Guille Polito <guillermopolito@gmail.com>
Date:   Wed Mar 21 15:37:32 2018 +0100

    Adding a title

commit 37adf4eaa945cbd7460991f88bff5aa902db06ce
Author: Guille Polito <guillermopolito@gmail.com>
Date:   Wed Mar 21 14:02:43 2018 +0100

    first version
```

`git log` prints the list of commits in order of parenthood. The one on the top is the most recent commit, our last commit. The one below is its parent, and so on. As you can see, each commit has an id, the author name, the timestamp and its message.

To display a more compact version (commit ids + message) of the log use

```
git log --oneline
```

We can also ask Git what are the changes introduced in a particular commit using the command `git show`.

```
$ git show 0c0e5ff55b56fe8eabc1661a1da64b41f9d74472
commit 0c0e5ff55b56fe8eabc1661a1da64b41f9d74472
Author: Guille Polito <guillermopolito@gmail.com>
Date:   Wed Mar 21 15:37:32 2018 +0100

    Adding a title

diff --git a/README.md b/README.md
index e69de29..cad05f1 100644
--- a/README.md
+++ b/README.md
@@ -0,0 +1 @@
+! a title
\ No newline at end of file
```

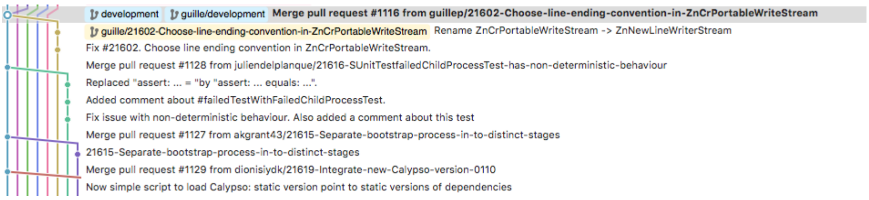
That will give us the commit description as in `git log` plus a (not so readable) diff of the modified files showing the inserted, modified and deleted lines. More advanced graphical tools are able to read this description and show a more user-friendly diff.

Seeing the history graph

Git's log provides a more graphish view on the terminal using some cute ascii art. This view can be accesses through the `git log --graph --oneline --all` command. Here is an example of this view for a more complex project. In this view, stars represent the commits with their ids and commit messages, and lines represent the parenthood relationships.

```
$ git log --graph --oneline --all
* 4eb8446 Documenting
* e5a3e2e Add tests
* 680a79a Some other
| *   ed4854f Merge pull request #1137
| |\
| | * 9e30e37 Some feature
| * |   ba7f65c Merge pull request #1138
| |\ \
| | * | 31a40c4 Some Enhancement
| | |/
| * |   2d4698d Merge pull request #1139
| |\ \
| | * | 20c0ff4 Some fix
| | |/
| * |   ae3ec45 Merge pull request #1136
```

However, we are not always in the mood of using the terminal, or of wanting to decode what was done in ascii art. There are tools that are more suitable to explore the history of a project, usually providing some nice graphical ca-



development guille/development **Merge pull request #1116 from guillep/21602-Choose-line-ending-convention-in-ZnCrPortableWriteStream**
guille/21602-Choose-line-ending-convention-in-ZnCrPortableWriteStream Rename ZnCrPortableWriteStream -> ZnNewLineWriterStream
Fix #21602. Choose line ending convention in ZnCrPortableWriteStream.
Merge pull request #1128 from juliendelplanque/21616-SUnitTestfailedChildProcessTest-has-non-deterministic-behaviour
Replaced "assert: ... = "by "assert: ... equals: ...".
Added comment about #failedTestWithFailedChildProcessTest.
Fix issue with non-deterministic behaviour. Also added a comment about this test
Merge pull request #1127 from akgrant43/21615-Separate-bootstrap-process-in-to-distinct-stages
21615-Separate-bootstrap-process-in-to-distinct-stages
Merge pull request #1129 from dionisiydk/21619-Integrate-new-Calypso-version-0110
Now simple script to load Calypso: static version point to static versions of dependencies

Figure 1-1 Example of SourceTree's commit graph view

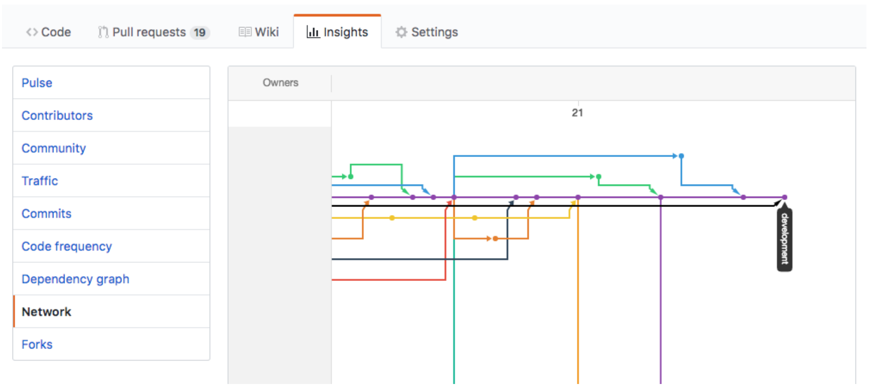


Figure 1-2 Example of Github's commit graph view

pabilities. This is the case of tools such as SourceTree (Figure 1-1) or Github's network view (Figure 1-2).

1.2 Discarding your local changes

It comes the time for every woman/men to make mistakes and want to discard them. Doing so may be dangerous, since once discarded you will not able to recover your changes. It is however possible to instruct Git to do so. For it, there are two Git comments that will perform the task for you and when combined they will completely discard every dirty file and directory in your repository: `git reset` and `git clean`

```
$ git reset --hard <commit_id>
$ git clean -df
```

The `-d` option removes untracked directories in addition to untracked files, while the `-f` option is a shortcut `--force`, forcing the corresponding deletions.

The reason for needing two commands instead of one relies on the fact that Git has several staging areas (such as the ones used to keep the tracked files), which we usually would like to clean when we discard the repository. Of course, experienced readers may search why they would need both in Git's documentation.

To do what about this[1] ?

1.3 Ignoring files

Many times we will find that we do not want to commit some files that are in our repository's directory. This is mostly the case of generated or automatically downloaded files. For example, imagine you have a C project and some makefiles to compile it, generating a binary library. While it would be good to store the result of compilation from time to time, storing it in a Git repository (or SVN, or Bazar) may be a cause of headaches. First, as you will see in **??**, this may be a cause for conflicts. Second, since we should be able to generated such binary library from the sources, having the already compiled result in the repository does not add so much value.

This same ideas can be used to ignore any kind of generated file. For example, pdfs generated by document generation tools, meta-data files generated by IDEs and tools (e.g., Eclipse), compiled libraries (e.g., dll, so, or dylib files).

In such cases, we can tell Git to ignore cetain files using the `.gitignore` file. The `.gitignore` file is an optional text file that we can write in the root of our repository with a list of file paths to ignore.

```
# Example of .gitignore file

# Lines starting with hashtags are comments

# A file name will ignore that file
someignoredfile.txt

# A file name will ignore that file
someignoredfile.txt

# A file pattern will ignore all pdf files
*.pdf
```

Once your file is ready, you have to add it and commit it to Git to make it take effect.

```
$ git add .gitignore
$ git commit -m "Added gitignore"
```

From this moment on, all listed files will be ignored by `git add` and `git status`. And you will be able to perform further commands to add "all but ignored files":

```
$ git add .
```

Note If a file or a file type is tracked but you want git to ignore its changes afterward, adding it to .gitignore file will not make the job i.e. git will continue to track it.

To avoid keeping track of it in the future, but secure it locally in your working directory, it must be removed from the tracking list using ==git rm –cached <file> (.<file_type>)==. Nevertheless, be aware that the file is still present in the past history!

To do Link to **??** to remove a (sensitive) file from history

1.4 Getting out of Detached HEAD

Detached head means no other than ”HEAD is not pointing to a branch”. Being in a detached HEAD state is not bad in itself, but it may provoke loss of changes. As a matter of fact, any commit that is not properly referenced by another commit or by another Git reference (tag, branch) may be garbage collected.

Git will not forbid you to commit in this state, but any new commit you create will only reachable if you remember the commit hash. To get out of dettached HEAD, the easiest solution is to checkout a branch, as we will see in the next section. Checking out a branch will set HEAD to point to a branch instead of a commit, saving you some HEADaches.

1.5 Accessing your Repository through SSH

To be able to access your repository from your local machine, you need to setup your credentials. Think it this way: you need to tell the server who you are on every interaction you have with it. Otherwise, Github will reject any operation against your repository. Such a setup requires the creation and uploading of SSH keys.

An SSH key works as a lock: a key is actually a pair of a public and a private key. The private key is meant to reside in your machine and not be published at all. A public key is meant to be shared with others to prove your identity. Whenever you want to prove your identity, SSH will exchange messages encrypted with your public key, and see if you are able to decrypt it using your private key.

To create an SSH key, in *nix systems you can simply type in your terminal

```
$ ssh-keygen -t rsa -b 4096 -C "your_email@some_domain.com"
```

Follow the instructions in your terminal such as setting the location for your key pair (usually it is `$HOME/.ssh`) and the passphrase (a kind of password).

Finally, you'll end up with your public/private pair on the selected location. It is now time to upload it to Github.

Connect yourself to your Github settings (usually https://github.com/settings/profile) and go to the "SSH and GPG keys" menu. Import there the contents of your public key file. You should be now able to use your repository.

1.6 Rewriting the history

Many times it happens that we accidentally commit something wrong. Maybe we wanted to commit more or less things, maybe a completely different content, or we did a mistake in the commit's message. In these cases, we can rewrite Git's history, e.g, undo our current commit and go back to the previous commit, or rewrite the current commit with some new properties.

Be careful! Rewriting the history can have severe consequences. Imagine that the commit you want to undo was already pushed. This means that somebody else could have pulled this commit into her/his repository. If we undo this already publised commit, we are making everybody else's repositories obsolete! This can be indeed problematic depending on the number of users the project has, and their knowledge on Git to be able to solve this issue.

Undo a commit using `git reset --hard`

To undo the last commit, it is as easy as:

```
$ git reset --hard HEAD~1
```

`git reset --hard [commitish]` makes your current branch point to [commitish]. `HEAD` is your current head, and you can read `~1` as "minus one". In other words, `HEAD~1` is head minus one, which boils down to the parent of head, our previous commit.

You can use this same trick to rewrite the history in any other way, since you can use any commitish expression to reset. For example, `HEAD~17` means 17 versions before head, or `someBranch~4` means four commits before the branch `someBranch`.

Update a commit's message using `git commit --amend`

To change our current commit's message you can use the following command:

```
$ git commit --amend -m "New commit message"
```

Or, if you don't use the `-m` option, a text editor will be prompt so you can edit a commit message.

```
$ git commit --amend
```

You can use the same trick not only to modify a commit's message but to modify your entire commit. Actually, just adding new things with `git add` before an `--amend` will replace the current commit with a new commit merging the previous commit changes with what you just added.

1.7 How to overwrite/modify commits

WARNING: It is highly not recommended to rewrite the history of a repo especially when part of it has already been pushed to a remote. Modifying the history will most likely break the history shared by the different collaborators and you may deal with an inextricable merge conflict.

Change the last commit

Imagine you have just committed your changes and have not pushed them yet, but

1. you are not satified with the commit message

```
$ git log --oneline
$ git commit --amend -m "Updated commit message"
$ git log --oneline
```

2. you forgot to save some modification or to add some files before committing. Then make your changes and use

```
$ git commit -a --amend --no-edit
```

Merge two commits

First, it is worth repeating that you must think twice before modifying the history of the repo. Now, assume that you have not pushed the corresponding commits. Merging two consecutive commits is a way to work with a cleaner tree. Consider you want to merge commits ”Intermediate” and ”Old”

```
$ git log --oneline

eae7846 New
71c0c64 Intermediate
f039832 Old
cca92f1 Even older
```

Then, you can interactively `-i` focus on the last three `HEAD~3` commit.

```
$ git rebase -i HEAD~3

pick f039832 Old
pick 71c0c64 Intermidiate
pick eae7846 New
```

Note Observe that the commits are displayed in the reversed order.

Now you can squash the commit ”Intermediate” into its parent commit ”Old”

```
pick f039832 Old
squash 71c0c64 Intermidiate
pick eae7846 New
```

And set the message e.g. ”Merge intermediate + old” attached to the single.

```
# This is a combination of 2 commits.
# This is the 1st commit message:

Old

# This is the commit message #2:

Intermediate

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
```

```
$ git log --oneline

eae7846 New
651375a Merge intermediate + old
cca92f1 Even older
```

Note The commit id has changed

Pushing rewritten history

As soon as the history we have rewritten was never pushed before, we can continue working normally and pushing our changes then without problems. However, if we have already pushed the commit we want to undo, this means that we are potentially impacting all users of our repository. Because of the problems it can pose to other people, pushing a rewritten history is not a completely favoured by Git. Better said, it is not allowed by default and you’ll be warned about it:

```
$ git push
To git@github.com:REPOSITORY_OWNER/YOUR_REPOSITORY.git
 ! [rejected]        YOUR_BRANCH -> YOUR_BRANCH (non-fast-forward)
```

```
error: failed to push some refs to
    'git@github.com:REPOSITORY_OWNER/YOUR_REPOSITORY.git'
hint: Updates were rejected because the tip of your current branch
    is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for
    details.
```

With this message Git means that you should not blindly overwrite the history. Also, it suggests to pull changes from the remote repository. However, doing that will bring back to our repository the history we wanted to undo! What we want to do is to impose our current (undone) state in the remote repository. To do that, we need to **force** the push using the `git push --force` or the `git push -f` option.

```
$git push -f
Total 0 (delta 0), reused 0 (delta 0)
To git@github.com:REPOSITORY_OWNER/YOUR_REPOSITORY.git
+ a1713f3...6e0c7bf YOUR_BRANCH -> YOUR_BRANCH (forced update)
```