

An introduction to the Git version control system

Felix Gruber

IGPM, RWTH Aachen

February 22, 2018 IGPM Oberseminar

Outline

- 1 Versioncontrol using Git
- 2 Basic Commands
- 3 Working with Branches
- 4 Collaborating via Remote Repositories
- 5 Best Practices
- 6 Further Reading
- 7 Questions and Answers

What Is Versioncontrol?

- a repository of all older versions of your code
- a tool to compare different versions and understand changes
- a framework to cooperatively write code (or other text)

Versioncontrol using Git

Why use versioncontrol?

- understand changes in your code → indispensable when working with other developers
- Simplifies merging changes from different developers
- helps when debugging regressions

Why use Git?

- by now the most commonly used VCS → high probability that you will later get involved in a project using Git; large amount of helpful Tutorials, StackOverflow threads, etc.
- decentral → local access to the whole history of your code, thus many operations can be done much faster than e.g. in SVN.

Versioncontrol using Git

Where to use Git?

- writing code
- writing papers
- tracking changes in configuration files like `.bashrc`, `.vimrc`, etc.
for this you might want to look at specialized Git wrappers like `vcsh`
- my 10k lines long `.bib` file
- generally: anything mostly text based, especially if you want to collaborate on it with other people

Example History Graph

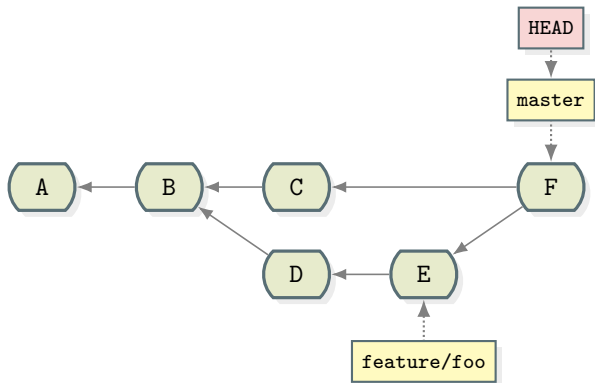


Figure: Git's version history is a directed acyclic graph (DAG) and its branches and tags are named pointers to commits

See `git help glossary` for terminology.

What Does a Commit Look Like?

```
commit b40c51576b644ebc61d5f644d73dbfd5e6bab777
Author:      Felix Gruber <gruber@igpm.rwth-aachen.de>
AuthorDate:  Tue Feb 20 18:59:53 2018 +0100
Committer:   Felix Gruber <gruber@igpm.rwth-aachen.de>
CommitDate:  Tue Feb 20 18:59:53 2018 +0100
ParentCommit(s): 82e9c34d7dfa7148b3aabf91479c26e05f340352
```

short description of commit

This is an example for a Git commit which is identified by its commit hash (the cryptic hex code above).

After the short one-line description it is customary to include a longer description of your changes and clarify why they were done.

```
<hash of file tree of version represented by this commit>
```

First Time Setup

- Set user name and email address for commits:

```
git config --global user.name "Your Name"
```

```
git config --global user.email name@igpm.rwth-aachen.de
```

- For old versions of Git: set colored output for `git diff`, `git status`, etc.

```
git config --global color.ui auto
```

- Set editor for commit messages, etc.

```
git config --global core.editor nano
```

(default is `$EDITOR`, or if unset `vi`)

Creating a Git Repository

- cloning an existing repository

```
git clone https://example.com/remote_repository.git
```

- creating a new repository

```
git init hello_world
```

Creating commits

working directory → index → commit

- 1 add changes to index

`git add filename`

or graphical frontend: `git gui`

- 2 see which changes get committed

`git show`

`git diff --cached`

- 3 create commit

`git commit`

This will open your editor to let you type in the commit message

Viewing the History

- `git log`
- `git diff old_commit new_commit`
shows changes **after** `old_commit` up to (and including) `new_commit`
- Graphical front-ends:
`gitk --all`
`git gui blame filename`

Viewing the History

- `git log`
- `git diff old_commit new_commit`
shows changes **after** `old_commit` up to (and including) `new_commit`
- Graphical front-ends:
`gitk --all`
`git gui blame filename`

Commits can be referred to by

- (truncated) hash
- branch or tag name
- `commit^` for previous commit
- for a complete list see `git help revisions`

Branching

- show list of branches

```
git branch
```

- create new branch

```
git checkout -b new_branch
```

```
git branch new_branch
```

- switch to branch (this changes the working directory)

```
git checkout branch
```

- rename branch

```
git branch -m old_branch new_branch
```

- delete branch

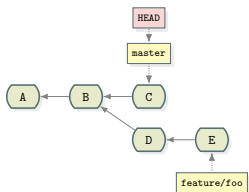
only merged branches:

```
git branch -d branchname
```

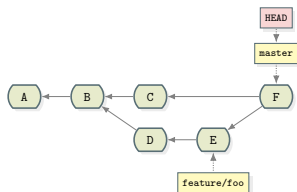
unmerged branches (**Warning: possible data loss**):

```
git branch -D branchname
```

Merging and Rebasing

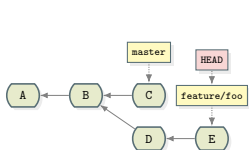


(a) Before...

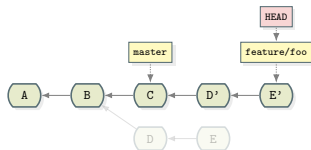


(b) ... and after the merge

Figure: git merge feature/foo



(a) Before...



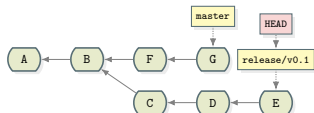
(b) ... and after the rebase

Figure: git rebase master

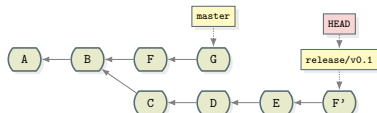
Resolving Merge Conflicts

- When the same lines get changed on both branches, merging will not succeed
- `git status` shows files affected by the conflict and gives instructions how to solve the conflict
- conflicting lines are marked with <<<<<<, ===== and >>>>>>
- resolve conflicts in your editor
- check changes with `git diff` and `git add file`
- when all conflicts are resolved `git commit`

Cherry-Pick



(a) Before...



(b) ... and after the cherry-pick

Figure: git cherry-pick F

Can be used to pick bugfixes to a (release) branch

- `git stash` is used to save the current state of the working directory on a stack
- Useful when shortly switching branches
- Apply last stashed changes with `git stash pop` to the current working directory

Talking to Remote Git Repositories

- Until now, all Git commands (except for clone) were local operations.
- remote operations:

- ▶ Add a new remote repository:

```
git remote add name https://example.com/repo.git
```

- ▶ Push currently checked-out branch to remote repository:

```
git push
```

- ▶ Update remote tracking branches:

```
git fetch
```

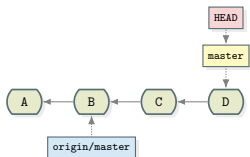
- ▶ Fetch + Merge to currently checked-out branch:

```
git pull
```

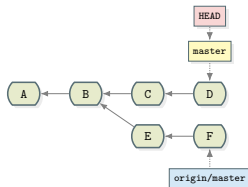
- ▶ Show all remote tracking branches:

```
git branch -r
```

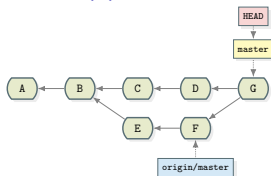
Remote Examples



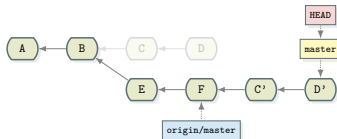
(a) Before



(b) `git fetch`



(c) `git pull` (fetch + merge)



(d) `git pull --rebase` (fetch + rebase)

Figure: Different ways to pull changes

Git Servers

Instead of setting up your own Git server, you probably want to use one of the following servers:

- IGPM Git Server

`https://username@www.igpm.rwth-aachen.de/git/repository`

Ask Frank if you want to create a new user or repository

- RWTH GitLab Server

`https://git.rwth-aachen.de/`

Up to 50 private or public repositories for employees and students of RWTH.

- GitLab

`https://gitlab.com`

- BitBucket

`https://bitbucket.org`

- GitHub

`https://github.com`

Rewriting History

Changing existing commits might be used to

- clean up a feature branch before merging
- fixing errors introduced in the last commit
- **Warning:** never do that on commits that have already been pushed!

Rewriting History

Changing existing commits might be used to

- clean up a feature branch before merging
- fixing errors introduced in the last commit
- **Warning:** never do that on commits that have already been pushed!

The following commands can be used to change commits:

- change last commit (after adding changes to index)

```
git commit --amend
```

- interactive rebase

```
git rebase -i
```

To help fix regressions, one can use `git bisect`

- 1 `git bisect start` bad good
- 2 Git then checks out the commit in the middle between bad and good
- 3 Test for regression and mark commit with `git bisect good` or `git bisect bad`
- 4 lather, rinse, repeat until only one commit left
- 5 congratulations, you have found the commit that caused the regression

see `git bisect --help` for automation, visualization and more

Best Practices

- Commit early, commit often.
→ small atomic commits
- Use meaningful commit messages consisting of
 - ▶ 1 line short description (max. 50 characters)
 - ▶ followed by a blank line
 - ▶ followed by a long description answering why the commit was done and what it changes

Take a look at the Linux kernel's Git repository for examples.

- Clean up your commits before pushing (`git rebase -i`); don't touch them afterwards (since this might confuse your collaborators).
- Use `.gitignore` files to exclude generated files like compiler artifacts and log files.

Recommendations for organizing branches

- use the master branch for semi-stable development
- use short lived feature branches when doing larger changes
- “Cactus” branching for releases
- use branch prefixes like `release/`, `feature/` and `cleanup/`

- Git Annex, Git LFS

Git does not work good with binary files. To keep track of my collection of papers, I use Git Annex to sync between work and home computers.

- vcsh

Wrapper around Git that helps to keep different configuration files in `$HOME` in different Git repositories.

Further Reading

- Git's man pages:

```
git subcommand --help
```

```
git help
```

```
git help -g
```

- 1 page quick overview to print out:

<http://www.cheat-sheets.org/saved-copy/git-cheat-sheet.pdf>

- short git tutorial: `git help tutorial`

- short exercises to learn Git:

<https://github.com/pragma-training/gitkatas>

- For those who want a more detailed introduction:

<http://git-scm.com/book>

- A list of some useful tips not just for beginners:

<https://www.andyjeffries.co.uk/25-tips-for-intermediate-git-users/>

Questions? → Answers!

Any Questions?

- Was something unclearly explained?
- Anything you ever wanted to know about Git?