Algorithm Engineering

Jens K. Mueller

jkm@informatik.uni-jena.de

Department of Mathematics and Computer Science
Friedrich Schiller University Jena

Tuesday 21st October, 2014

Version Control with Git

Version Control

Distributed Version Control with Git

What is Git?

- Distributed version control system
- Developed by Linus Torvalds
- Runs almost everywhere
- Used by Linux kernel, Samba, X.Org, Qt, GNOME, Android, ...

Features:

- Very flexible work flows
- Fast and scalable
- Cryptographic secure history

How Git stores its Data

Everything is an object with a SHA1

All git objects have a type, content, and size (of the content). For a given object its (object) name is a 40-digit hash (SHA1) of its content.

Object Types:

- Blob Object
  Content: data
- Tree Object
  Content: list of blob and tree names with its type and file name

How Git stores its Data (cont.)

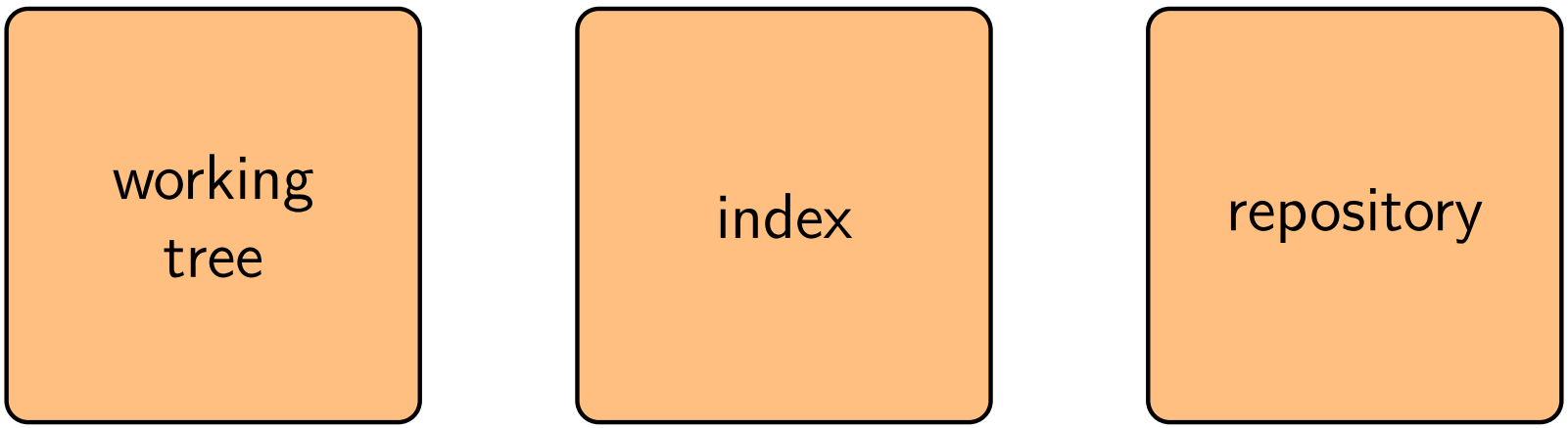
Everything is an object with a SHA1

- Commit Object
  Content: 1 tree name, 0+ parent commit name(s), author (with date), committer (with date), and a commit message
- Tag Object
  Content: type, name, tagger, tag message

The commit history of a project forms a directed acyclic graph.

Assuming SHA1 is safe: Given a commit and its SHA1 the whole history of the project is secured. I.e. a change in the history can be noticed.

Interacting with Git



Creating a Repository

- Make the current directory a repository
  `$ git init`
- Create a repository in the directory `myrepository/`
  `$ git init myrepository`
- Create a bare repository accessible by all
  `$ git init --bare --shared=all /git/myrepository.git`

Adding Content

- Adding a file pattern to the index
  `$ git add filepattern`
  This starts tracking files and also stage (i.e. add to index) changes of tracked files.
  `$ git add -p filepattern`
  Interactively stage only parts of a file.
- Adding a file to the index ignoring untracked files
  `$ git add -u filepattern`
- Automatically stage tracked files and commit
  `$ git commit -a`

Status

- `$ git status`
  Reports the status of the working directory and the index (i.e. what will be committed in case of `$ git commit`).
  Further it shows the untracked files.
  For a condensed output run `$ git status -s`.

Showing Modifications

- Diff file against the index
  `$ git diff file`
  This shows what is not staged yet.
- Diff file in index against the repository
  `$ git diff --cached file`
  This shows what will go into the next commit.
- Diff file against the repository
  `$ git diff HEAD file`

Use other difftools using `$ git difftool -t <tool>`.

Committing

- Record the staged changes
  `$ git commit`
- `$ git commit -a`
  Automatically add tracked files to the index and then commit. Shorthand for `$ git add -u; git commit`.

Commit each logical separate change in a different commit and provide a useful commit message.

Commit Message

```
A summary in a single line less than 50 charaters

A more detailed explaination, if necessary. Wrapped at
about 72 characters. Write in imperative present tense.
It should include your motivation for the change and
contrast the new implementation/behavior with the old.

- bullet point
  indent here

- bullet point
```

Unstaging

- Unstage a staged file
  `$ git reset HEAD file`
  This is useful if you accidentally staged a file but actually you don't want to commit it (yet).

Since reset is quite a bad name, you can create an alias for it with `$ git config --global alias.unstage 'reset HEAD'`

Untracking Content

- Remove a file from index
  `$ git rm file`
  Removes file from the index and your working tree.
  Use `$ git rm --cached file` to remove it only from the index.
- Moving a file
  `$ git mv file.old file.new` is equivalent to `$ git rm --cached file.old; mv file.old file.new; git add file.new`
  That means git does not track the renaming of files.

Equivalently (without git rm file) you can do `$ rm file; git add file`. Notice this way you can use `$ git add -u` and `$ git commit -a`.
`$ git mv file.old file.new` is equivalent to `$ mv file.old file.new; git add file.old file.new`.

Checkout

- Checkout a file from the index
  ```
  $ git checkout -- file
  ```
- Checkout a file from the repository
  ```
  $ git checkout HEAD file
  ```

Branching

- Creating a branch
  `$ git branch mynewbranch`
- List (local) branches
  `$ git branch`
- Switching to a branch
  `$ git checkout mynewbranch`
  You can create and switch to a branch with the command `$ git checkout -b mynewbranch`, i.e. `$ git branch mynewbranch; git checkout mynewbranch`.
- Deleting a branch
  `$ git branch -d mynewbranch`

It's useful to separate the work in different branches. E.g. if you start working on a new feature, you do `$ git checkout -b featureX`.
There is by default a main branch called master.

Merging

- Merge the branch featureX into the current branch
  `$ git merge featureX`
  Usually you do `$ git checkout master` and `$ git merge featureX`.

When merging you may end up having conflicts. You'll need to fix these and `$ git add` the fixes and `$ git commit`.

Fast-Forward Merge

By default a separate commit is avoided, if the merge is a fast-forward. A merge is a fast-forward if the current branch's head is an ancestor of the to be merged branch head.

Branching and Merging

1. Create a branch
   ```
   $ git checkout -b mynewbranch
   ```
2. Hack, hack, hack
3. Commit
   ```
   $ git commit -a
   ```
4. Go to master
   ```
   $ git checkout master
   ```
5. Merge the branch (generating a merge conflict even if fast-forward)
   ```
   $ git merge [--no-ff] mynewbranch
   ```
6. Optionally delete branch
   ```
   $ git branch -d mynewbranch
   ```

Branching and Merging (cont.)

Instead of merging to an unchanged branch (fast-forward) we now change the master branch before merging.

4. Go to master
   ```
   $ git checkout master
   ```
5. Hack, hack, hack
6. Commit
   ```
   $ git commit -a
   ```
7. Merge the branch
   ```
   $ git merge mynewbranch
   ```
8. Resolve conflicts and commit
   ```
   $ git commit -a
   ```

Commit History

Find out how you ended up here

- To get the commit history
  ```
  $ git log
  ```
- More compact version
  ```
  $ git log --oneline
  ```
- Showing the directed acyclic commit graph
  ```
  $ git log --graph
  ```

Tagging

Give it a name for release

- Create a tag
  `$ git tag v1.0`
- `$ git tag -a v0.1 -m "Creating the first official version."`
  Creates an annotated tag. I.e. annotate with a date, tagger (person who created the tag), and a message.
- Delete a tag
  `$ git tag -d v1.0`
- Show tags
  `$ git tag`
- Show tags in history
  `$ git log --decorate`

In most cases one uses annotated tags. You can crytographically sign an annotated tag.

Interact with Remote Repositories

Since git is distributed. There is no central repository. But to collaborate with others it's nice to have a public repository. You can synchronize your repository with other repositories.

Your can host git repositories yourself or use e.g. GitHub.com, Gitorious.org, repo.or.cz, ...

Managing Remote Repositories

- List the remote repositories
  `$ git remote [-v]`
- Adding a the remote repository `git@github.com:me/my_project.git` as a remote named `github`
  `$ git remote add github git@github.com:me/my_project.git`
- Removing an remote repository
  `$ git remote rm github`

Cloning a Repository

- Clone the repository at `git://github.com/somerepository.git` to local repository `somerepository`
  `$ git clone`
  `git://github.com/somerepository.git`

When cloning a repository, a remote repository named origin is automatically added (see `$ git remote`).

Local repositories can be cloned via `/path/to/repo.git` or `file:///path/to/repo.git`. Additionally you can clone via ssh, http[s], ftp[s], and rsync.

Fetching from Remote Repositories

Remote branches are identical to local branches except you cannot check them out. But you can merge, diff, log etc. on them.
You can list all branches (including remote ones) with `$ git branch -a`.

- Synchronize with a remote repository
  `$ git fetch otherrepository`
- Synchronize with all remote repositories
  `$ git fetch --all`

Synchronizing means copying any data that is not locally available and updating informations about the remote branches.

Fetching from Remote Repositories (cont.)

You can inspect/merge a remote branch.

- Inspect the changes
  `$ git log otherrepository/master ^master`
- Merge the changes
  `$ git merge otherrepository/master`.

To fetch and merge the changes into the current branch use `$ git pull otherrepository`.

Git References

- The Git Community. The Git Community Book. 2010. URL: http://book.git-scm.com/
- GitHub team. Git Reference. 2010. URL: http://gitref.org/index.html
- Scott Chacon. Pro Git. 2010. URL: http://progit.org/book/

There is much more:

- Much documentation (man pages, cheat sheets, tutorials, ...)
- Many users