Introduction into version system GIT

Ing. Jiří Cága

February 1, 2020

GIT

Definition

Git is currently the most popular implementation of a distributed version control system. Git originates from the Linux kernel development and was founded in 2005 by Linus Torvalds. Nowadays it is used by many popular open source projects, e.g., the Android or the Eclipse developer teams, as well as many commercial organizations. The core of Git was originally written in the programming language C, but Git has also been re-implemented in other languages, e.g., Java, Ruby and Python.

What is a version control system?

A version control system (VCS) allows you to track the history of a collection of files. It supports creating different versions of this collection. Each version captures a snapshot of the files at a certain point in time and the VCS allows you to switch between these versions. These versions are stored in a specific place, typically called a repository.

Type version systems

- Localized: it keeps local copies of the files. This approach can be as simple as creating a manual copy of the relevant files.
- Centralized: itprovides a server software component which stores and manages the different versions of the files. A developer can copy (checkout) a certain version from the central sever onto their individual computer.
- Distributed: In a distributed version control system each user has a complete local copy of a repository on his individual computer. The user can copy an existing repository. This copying process is typically called cloning and the resulting repository can be referred to as a clone. Every clone contains the full history of the collection of files and a cloned repository has the same functionality as the

Notes: Localized and centralized approach have the drawback that they have one single point of failure. In a localized version control systems it is the individual computer.

Git repositories

A Git repository contains the history of a collection of files starting from a certain directory. The process of copying an existing Git repository via the Git tooling is called cloning. After cloning a repository the user has the complete repository with its history on his local machine. Of course, Git also supports the creation of new repositories.

If you clone a Git repository, by default, Git assumes that you want to work in this repository as a user. Git also supports the creation of repositories targeting the usage on a server. * bare repositories are supposed to be used on a server for sharing changes coming from different developers. Such repositories do not allow the user to modify locally files and to create new versions for the repository based on these modifications. * non-bare repositories target the user. They allow you to create new changes through modification of files and to create new versions in the repository. This is the default type which is created if you do not specify any parameter during the clone operation.

Notes: A local non-bare Git repository is typically called local repository.

Working tree

A local repository provides at least one collection of files which originate from a certain version of the repository. This collection of files is called the working tree. It corresponds to a checkout of one version of the repository with potential changes done by the user.

The user can change the files in the working tree by modifying existing files and by creating and removing files. A file in the working tree of a Git repository can have different states. These states are the following: * untracked: the file is not tracked by the Git repository. This means that the file never staged nor committed. * tracked: committed and not staged * staged: staged to be included in the next commit * dirty / modified: the file has changed but the change is not staged After doing changes in the working tree, the user can add these changes to the Git repository or revert these changes. Adding to a Git repository via staging and committing After modifying your working tree you need to perform the following two steps to persist these changes in your local repository: * add the selected changes to the staging area (also known as index) via the git add command * commit the staged changes into the Git repository via the git commit command

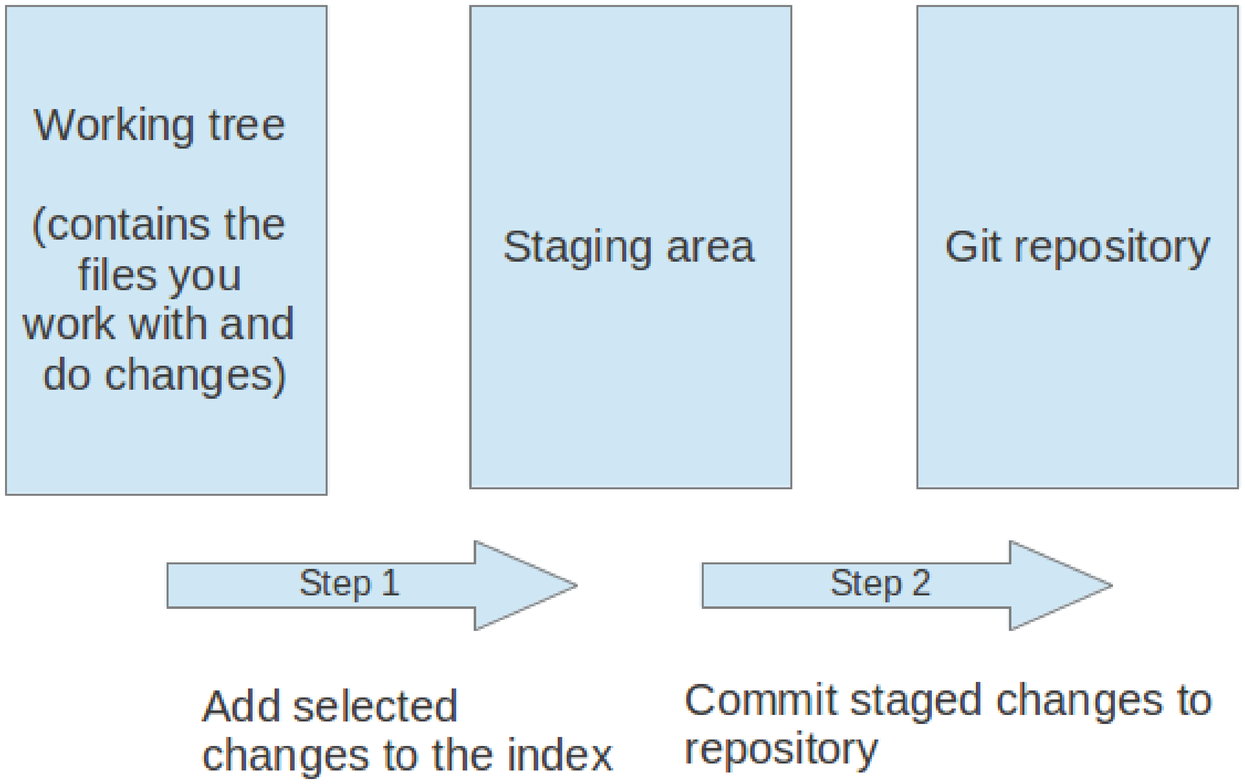
Working tree
(contains the files you work with and do changes)
Staging area
Git repository
Step 1
Step 2
Add selected changes to the index
Commit staged changes to repository

Figure 1: Process adding changes into Git repository

The git add command stores a snapshot of the specified files in the staging area. It allows you to incrementally modify files, stage them, modify and stage them again until you are satisfied with your changes.

Some tools and Git user prefer the usage of the index instead of staging area. Both terms mean the same thing. After adding the selected files to the staging area, you can commit these files to add them permanently to the Git repository.

Committing creates a new persistent snapshot (called commit or commit object) of the staging area in the Git repository. A commit object, like all objects in Git, is immutable.

The staging area keeps track of the snapshots of the files until the staged changes are committed. For committing the staged changes you use the git commit command. If you commit changes to your Git repository, you create a new commit object in the Git repository. See Commit object (commit) for information about the commit object.

The concept of branches

Git supports branching which means that you can work on different versions of your collection of files. A branch allows the user to switch between these versions so that he can work on different changes independently from each other.

For example, if you want to develop a new feature, you can create a branch and make the changes in this branch. This does not affect the state of your files in other branches. For example, you can work independently on a branch called production for bugfixes and on another branch called feature_123 for implementing a new feature.

Branches in Git are local to the repository. A branch created in a local repository does not need to have a counterpart in a remote repository. Local branches can be compared with other local branches and with remote-tracking branches. A remote-tracking branch proxies the state of a branch in another remote repository. Git supports the combination of changes from different branches. The developer can use Git commands to combine the changes at a later point in time.

Core Git terminology

- **Branch:** A branch is a named pointer to a commit. Selecting a branch in Git terminology is called to checkout a branch. If you are working in a certain branch, the creation of a new commit advances this pointer to the newly created commit. Each commit knows their parents (predecessors). Successors are retrieved by traversing the commit graph starting from branches or other refs, symbolic references (for example: HEAD) or explicit commit objects. This way a branch defines its own line of descendants in the overall version graph formed by all commits in the repository. You can create a new branch from an existing one and change the code independently from other branches. One of the branches is the default (typically named master). The default branch is the one for which a local branch is automatically created when cloning the repository.
- **Commit:** When you commit your changes into a repository this creates a new commit object in the Git repository. This commit object uniquely identifies a new revision of the content of the repository. This revision can be retrieved later, for example, if you want to see the source code of an older version. Each commit object contains the author and the committer. This makes it possible to identify who did the change. The author and committer might be different people. The author did the change and the committer applied the change to the Git repository. This is common for contributions to open source projects.
- **HEAD:** is a symbolic reference most often pointing to the currently checked out branch. Sometimes the HEAD points directly to a commit object, this is called detached HEAD mode. In that state creation of a commit will not move any branch. If you switch branches, the HEAD pointer points to the branch pointer which in turn points to a commit. If you checkout a specific commit, the HEAD points to this commit directly.
- **Index:** is an alternative term for the staging area.
- **Repository:** contains the history, the different versions over time and all different branches and tags. In Git each copy of the repository is a complete repository. If the repository is not a bare repository, it allows you to checkout revisions into your working tree and to capture changes by creating new commits. Bare repositories are only changed by transporting changes from other repositories.
- **Revision:** Represents a version of the source code. Git implements revisions as commit objects (or short commits). These are identified by an SHA-1 hash.
- **Staging area:** The staging area is the place to store changes in the working tree before the commit. The staging area contains a snapshot of the changes in the working tree (changed or new files) relevant to create the next commit and stores their mode (file type, executable bit).
- **Tag:** points to a commit which uniquely identifies a version of the Git repository. With a tag, you can have a named point to which you can always revert to. You can revert to any point in a Git repository, but tags make it easier. The benefit of tags is to mark the repository for a specific reason, e.g., with a release. Branches and tags are named pointers, the difference is that branches move when a new commit is created while tags always point to the same commit. Tags can have a timestamp and a message associated with them.
- **URL:** in Git determines the location of the repository. Git distinguishes between fetchurl for getting new data from other repositories and pushurl for pushing data to another repository. Working tree: contains the set of working files for the repository. You can modify the content and commit the changes as new commits to the repository.

Technical details of a commit object

This commit object is addressable via a hash (SHA-1 checksum). This hash is calculated based on the content of the files, the content of the directories, the complete history of up to the new commit, the committer, the commit message, and several other factors. This means that Git is safe, you cannot manipulate a file or the commit message in the Git repository without Git noticing that corresponding hash does not fit anymore to the content.

The commit object points to the individual files in this commit via a tree object. The files are stored in the Git repository as blob objects and might be packed by Git for better performance and more compact storage. Blobs are addressed via their SHA-1 hash. Packing involves storing changes as deltas, compression and storage of many objects in a single pack file. Pack files are accompanied by one or multiple index files which speedup access to individual objects stored in these packs.

Git commands

```
git config -global user.name [name]  # This command sets the author name
```

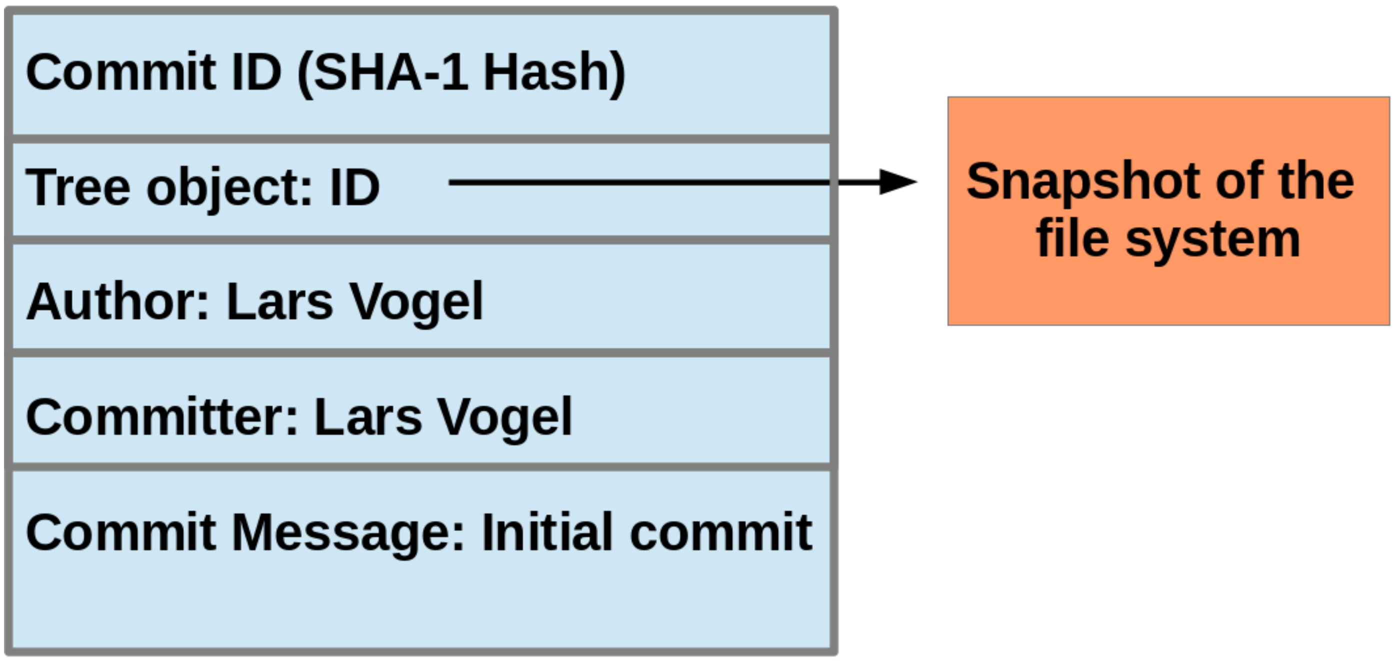

Figure 2: Commit object

```
git config -global user.email [email address] # This command sets the author email
git init [repository name] # This command is used to start a new repository.
git clone # This command is used to obtain a repository from an existing URL.
git add [file]  # This command adds a file to the staging area.
git commit -m [ Type in the commit message]     # This command records or snapshots the file permanently in t
git diff        # This command shows the file differences which are not yet staged.
git diff -staged    # This command shows the differences between the files in the staging area and the latest
git diff [first branch] [second branch]  # This command shows the differences between the two branches mentio
git reset [file]    # This command unstages the file, but it preserves the file contents.
git reset [commit]      # This command undoes all the commits after the specified commit and preserves the ch
git reset -hard [commit]#This command discards all history and goes back to the specified commit.
git status      # This command lists all the files that have to be committed.
git rm [file]       # This command deletes the file from your working directory and stages the deletion.
git log         # This command is used to list the version history for the current branch.
git log -follow[file]   # This command lists version history for a file, including the renaming of files also
git show [commit]   # This command shows the metadata and content changes of the specified commit.
git tag [commitID]  # This command is used to give tags to the specified commit.
git branch          # This command lists all the local branches in the current repository.
git branch [branch name]    # This command creates a new branch.
git branch -d [branch name]     # This command deletes the feature branch.
git checkout [branch name]      # This command is used to switch from one branch to another.
git checkout -b [branch name]   # This command creates a new branch and also switches to it.
git merge [branch name]     # This command merges the specified branchs history into the current branch.
git remote add [variable name] [Remote Server Link]  # This command is used to connect your local repository
git push [variable name] master   # This command sends the committed changes of master branch to your remote
git push [variable name] [branch]  # This command sends the branch commits to your remote repository.
git push -all [variable name]   # This command pushes all branches to your remote repository.
git push [variable name] :[branch name] # This command deletes a branch on your remote repository.
git pull [Repository Link]          # This command fetches and merges changes on the remote server to your wo
git stash save              # This command temporarily stores all the modified tracked files.
git stash pop               # This command restores the most recently stashed files.
git stash list              # This command lists all stashed changesets.
git stash drop              # This command discards the most recently stashed changeset.
```