Git/Internal structure

Understanding something of the internal structure of Git is crucial to understanding how Git works.

Contents

Naked Git Structure

The following is a freshly initialized git v1.9.0 repository.[1]

```
.
└── .git/
    ├── HEAD
    ├── branches/
    ├── config
    ├── description
    ├── hooks/
    │   ├── applypatch-msg.sample
    │   ├── commit-msg.sample
    │   ├── post-update.sample
    │   ├── pre-applypatch.sample
    │   ├── pre-commit.sample
    │   ├── prepare-commit-msg.sample
    │   ├── pre-push.sample
    │   ├── pre-rebase.sample
    │   └── update.sample
    ├── info/
    │   └── exclude
    ├── objects/
    │   ├── info/
    │   └── pack/
    └── refs/
        ├── heads/
        └── tags/
```

Additional files and folders may appear as activity happens on the repository.

Specific Files

COMMIT_EDITMSG

The message for a commit being made is saved here by a text editor.

FETCH_HEAD

Information is saved here from the last `git-fetch(1) (http://git-scm.com/docs/git-fetch)` operation, for use by a later `git-merge(1) (http://git-scm.com/docs/git-merge)`.

HEAD

HEAD indicates the currently checked out code. This will usually point to the branch you're currently working on.

You can also enter what git calls a "detached HEAD" state, where you are not on a local branch. In this state the HEAD points directly to a commit rather than a branch.

config

The configuration file for this git repository. It can contain settings for how to manage and store data in the local repository, remote repositories it knows about, information about the local user and other configuration data for git itself.

You can edit this file with a text editor, or you can manage it with the `git-config(1) (http://git-scm.com/docs/git-config)` command.

description

Used by repository browser tools - contains a description of what this project is. Not normally changed in non-shared repositories.

index

This is the staging area. It contains, in a compact form, all changes to files that have been staged for the next commit.

info/exclude

This is your own personal exclude file for your copy of the repo.

info/refs

If this file exists, it contains definitions, one to a line, of branches (both local and remote) and tags defined for the repository, in addition to ones that may be defined in individual files in `refs/heads` and `refs/tags`. This file seems to be used for large repositories with lots of branches or tags.

ORIG_HEAD

Operations that change commit history on the current branch save the previous value of HEAD here, to allow recovery from mistakes.

Folders Containing Other Files

branches

Never seems to be used.

hooks

Contains scripts to be run when particular events happen within the git repository. Git gives you a set of initial example scripts, with `.sample` on the ends of their names (see the tree listing above); if you take off the `.sample` suffix, Git will run the script at the appropriate time.

Hooks would be used, for example, to run tests before creating each commit, filter uploaded content, and implement other such custom requirements.

logs

The reflogs are kept here.

objects

This is where all the files, directory listings, commits and such are stored.

There are both unpacked objects in numbered directories under this, and "packs" containing many compressed objects within a pack directory. The uncompressed objects will be periodically collected together into packs by automatic "git gc" runs.

refs/heads

Can contain one file defining the head commit for each local branch (but see `info/refs` above).

refs/remotes

Can contain one subdirectory for each remote repository you have defined. Within each subdirectory, there is a file defining the tip commit for each branch on that remote.

refs/tags

Can contain one file defining the commit corresponding to each tag (but see `info/refs` above).

svn

This directory will appear if you use `git-svn(1) (http://git-scm.com/docs/git-svn)` to communicate with a Subversion server.

Basic Concepts

Object Types

A Git repository is made up of these object types:

- A *blob* holds the entire contents of a single file. It doesn’t hold any information about the name of the file or any other metadata, just the contents.
- A *tree* represents the state of a directory tree. It contains the pathnames of all the component files and their modes, along with the IDs of the blobs holding their contents. Note that there is no representation for a directory on its own, so a Git repository cannot record the fact that subdirectories were created or deleted, only the files in them.
- A *commit* points to a tree representing the state of the source tree as of immediately after that commit. It also records the date/time of the commit, the author/committer information, and pointers to any parent(s) of that commit, representing the immediately-prior state of the source tree.
- A *tag* is a name pointing to a commit. These are useful, for example, to mark release milestones. Tags can optionally be digitally signed, to guarantee the authenticity of the commit.
- A *branch* is a name pointing to a commit. The difference between a branch and a tag is that, when a branch is the currently-checked-out branch, then adding a new commit will automatically update the branch pointer to point to the new commit.

Blobs, trees and commits all have IDs which are computed from SHA-1 hashes of their contents. These IDs allow different Git processes on different machines to tell whether they have identical copies of things, without having to transfer their entire contents over. Because SHA-1 is a cryptographically strong hash algorithm, it is practically impossible to make a change to the contents of any of these objects without changing its ID. Git doesn’t prevent you from rewriting history, but you cannot hide the fact that you have done so.

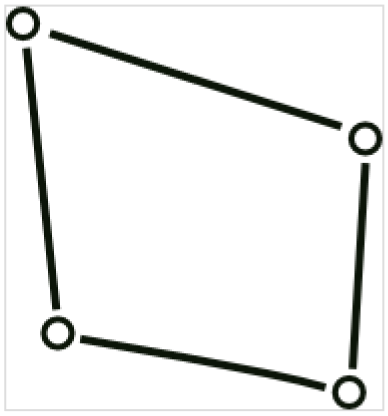
A commit may have 0, 1 or more parents. Typically there is only one commit with no parents—a *root commit*—and that is the first commit to the repository. A commit which makes some change to one branch will have a single parent, the previous commit on that branch. A commit which is a merge from two or more branches will have two or more parent commits.

Note that a branch points to a *single* commit; the chain of commits is implicit in the parent(s) of that commit, and their parents, and so on.

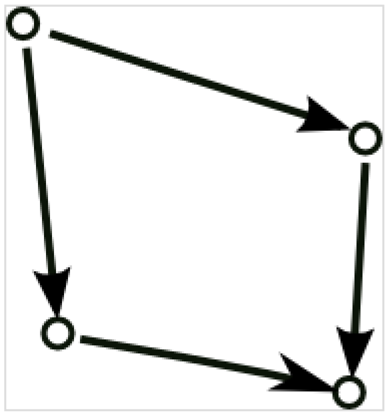
Topology Of Commits

The commit history in Git is arranged as a *directed acyclic graph* (DAG). To understand what this means, let's take the terms step by step.

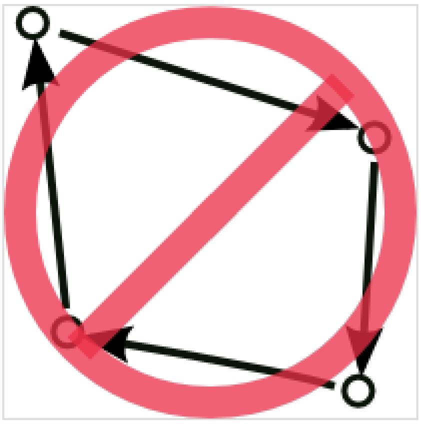
- In mathematical terms, a *graph* is a bunch of points (*nodes*) connected by lines (*edges*).


just a graph

- A *directed* graph is one where each edge has a direction, represented here by an arrowhead. Note that the arrow points from the child to the parent, not the other way round; it is the child that records who its parent(s) are, the parent does not record who its children are, because the set of children can change at any time, but the parent cannot change without invalidating its SHA-1 hash.


directed edges

- *Acyclic* means that, if you start from any point and traverse edges in the direction of the arrows, you can never get back to your starting point, no matter what choice you make at any branch. No child can ever be a (direct or indirect) parent of itself!


cycles not allowed

In Git terms, each node represents a commit, and the lines and arrows represent parent-child relationships. Banning cycles simply means that a commit cannot be a (direct or indirect) parent or a child of itself!

The Reflog

The *reflog*s record changes that are not saved as part of the commit history—things like rebases, fast-forward merges, resets and the like. There is one reflog per branch. The reflog is not a public part of the repository, it is strictly specific to your local copy, and information is only kept in it for a limited time (2 weeks by default). It provides a safety net, allowing you to recover from mistakes like deleting or overwriting things you didn't mean to.

Reachability And Garbage Collection

A commit is *reachable* if it is pointed to by a branch, tag or reflog entry, or is a parent of a commit which is reachable. A tree is correspondingly reachable if it is pointed to by a reachable commit, and a blob is reachable if it is pointed to by a reachable tree. Other commit/tree/blob objects are *unreachable*, and are not really serving any purpose beyond taking up space.

It is quite normal for your repositories to accumulate unreachable objects over time, perhaps as a result of aborted commits, deletion of unwanted branches, that kind of thing. Such objects will be deleted from the repository by a `git gc` command. This is also done automatically every now and then by some other commands, so it is rarely necessary to invoke `git gc` explicitly.

Git files out of .git folder

.gitkeep

Placed in a directory, it guarantees that it will be committed, even if empty.

.gitignore

Contains the files and folders to exclude from versioning. Ex:

```
/var/
/vendor/
```

```
/.env.*.local
```

.gitattributes

Contains somes attributes[1]. For example, to ignore .gitattributes and .gitignore in the "git archive" exports:

```
.gitattributes export-ignore
.gitignore export-ignore
```

Footnotes

1. ^ Generated with tree v1.5.1.1 using `tree -AnaF`.

1. https://git-scm.com/docs/gitattributes