Git Helper
(document under construction)

Armando Nolasco Pinto

February 10, 2019

Contents

1 Git Helper **3**

1.1 Starting with Git . . . 3
1.2 Data Model . . . 3
1.2.1 Objects Folder . . . 6
1.3 Refs . . . 6
1.3.1 Refs Folder . . . 7
1.3.2 Branch . . . 7
1.3.3 Heads . . . 7
1.4 Git Logical Areas . . . 7
1.5 Merge . . . 7
1.5.1 Fast-Forward Merge . . . 8
1.5.2 Three-Way Merge . . . 8
1.6 Remotes . . . 8
1.6.1 GitHub . . . 9
1.7 Commands . . . 10
1.7.1 Porcelain Commands . . . 10
1.7.2 Pluming Commands . . . 14
1.8 Navigation Helpers . . . 16
1.9 Configuration Files . . . 16
1.10 Pack Files . . . 16
1.11 Applications . . . 17
1.11.1 Meld . . . 17

1.11.2 GitKraken 17
1.12 Error Messages 17
1.12.1 Large files detected 17
1.13 Git with Overleaf 17
Bibliography 19

Chapter 1

Git Helper

1.1 Starting with Git

Git is a free and open source distributed version control system [1]. Git creates and maintains a database that records all changes that occur in a folder. The Git database is named a repository. It also allows to merge repositories that shared a common state in the past. These can be local repositories, i.e. stored in the same machine, or can be remote repositores, i.e. stored anywhere.

To create this database for a specific folder the Git application must be installed on the computer. The Git application and directions for its installation in all majors platforms can be obtained in the Git website (http://git-scm.com). You can access to the Git commands through the console or through a GUI interface. Here, we assume that you are going to use the console.

After the installation, to create the Git initial database open the console program and go to a folder and execute the following command:

git init

The Git database is created and stored in the folder *.git* in the root of your folder. The *.git* folder is your repository.

The Git commands allow you to manipulate this database, i.e. this repository.

1.2 Data Model

To understand Git is fundamental to understand the Git data model.

Git manipulates the following objects:

- commits - text files that store a description of the repository;

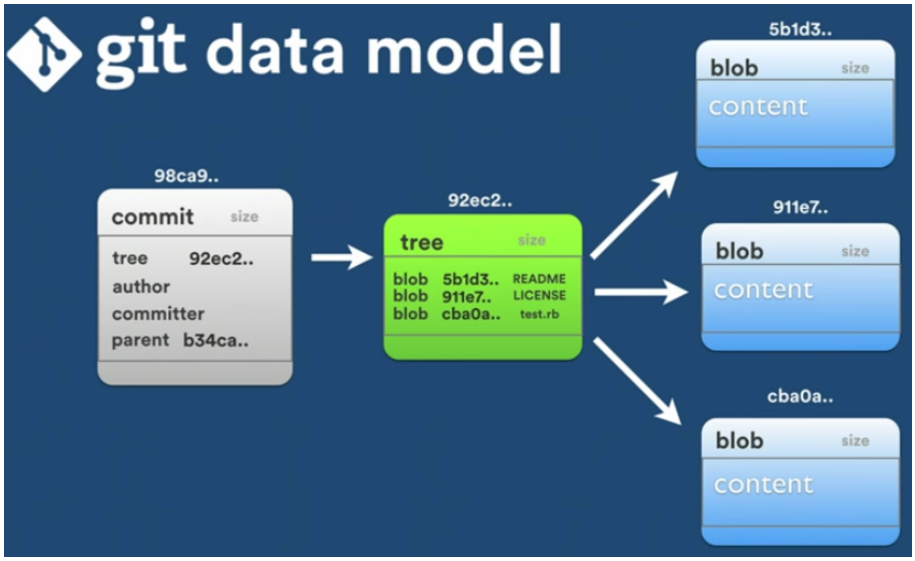

Figure 1.1: Git data model.

- trees - text files that store a description of a folder;
- blobs - the files that exist in your repository;
- tags - text files that store information about commits.

The objects are stored in the folder *.git/objects*. Each store object is identified by its SHA1 hash value, i.e. 20 bytes which identifies unequivocally the object. The SHA1 is just an algorithm that accepts some binary information and generates 20 bytes which are ideally unique for that information. The probability of collisions is extremely low, i.e. the probability that different information generates the same hash value is extremely low. Note that 20 bytes can be represented by a 40 characters hexadecimal string. The identifier of each object is the 40 characters hexadecimal string. Each particular object is stored in a sub-folder inside the *.git/objects*. The name of the sub-folder is the two most significative characters of the SHA1 hash value. The name of the file that is inside the sub-folder is the remanning thirty eight characters of the SHA1 hash value. The Git stores all committed versions of a file. The Git maintains a contend-addressable file systems, i.e. a file system in which the

files can be accessed based on its contend. Lets look more carefully in each stored object.

A **commit** object is identified by a SHA1 hash value, and has the following information: a pointer for a tree (the root), a pointer for the previous commit, the author, the committer and a commit message. The author is the person who did the work. The committer is the person who validate the work and who apply the work by doing the commit. By doing this difference git allow both to get the credit, the author and the committer. Example of a commit file contend:

tree 2c04e4bad1e2bcc0223239e65c0e6e822bba4f16
parent bd3c8f6fed39a601c29c5d101789aaa1dab0f3cd
author NetXPTO <netxpto@gmail.com> 1514997058 +0000
committer NetXPTO <netxpto@gmail.com> 1514997058 +0000

Here goes the commit message.

A **tree** object is identified by a SHA1 hash value, and has a list of blobs and trees that are inside that tree. A tree object identifies a folder and its contend. Example of a tree file contend:

100644	blob	bdb0cabc87cf50106df6e15097dff816c8c3eb34	.gitattributes
100644	blob	50492188dc6e12112a42de3e691246dafdad645b	.gitignore
100644	blob	8f564c4b3e95add1a43e839de8adbfd1ceccf811	bfg-1.12.16.jar
040000	tree	de44b36d96548240d98cb946298f94901b5f5a05	doc
040000	tree	8b7147dbfdc026c78fee129d9075b0f6b17893be	garbage
040000	tree	bdfcd8ef2786ee5f0f188fc04d9b2c24d00d2e92	include
040000	tree	040373bd71b8fe2fe08c3a154cada841b3e411fb	lib
040000	tree	7a5fce17545e55d2faa3fc3ab36e75ed47d7bc02	msbuild
040000	tree	b86efba0767e0fac1a23373aaf95884a47c495c5	mtools
040000	tree	1f981ea3a52bccf1cb00d7cb6dfdc687f33242ea	references
040000	tree	86d462afd7485038cc916b62d7cbfc2a41e8cf47	sdf
040000	tree	13bfce10b78764b24c1e3dfbd0b10bc6c35f2f7b	things_to_do
040000	tree	232612b8a5338ea71ab6a583d477d41f17ebae32	visualizerXPTO
040000	tree	1e5ee96669358032a4a960513d5f5635c7a23a90	work_in_progress

A **blob** is identified by a SHA1 hash value, and has the file contend compressed. A git header and tailor is added to each file and the file is compressed using the zlib library. The git header is just the object type, a space character, the file size in bytes and the \NUL caracter, for instance "blob 13\NUL", the tailor is just the \n caracter. The compressed blob (header+file contend+tailer) is stored as a binary file.

There are two types of **tags**, lightweight and annotated tags. Lightweight tags are only a ref to a commit, see section below. Annotated tags are objects stored as text files, which as has information about the commit, to each the tag point, the tagger (name and e-mail), tag date and a tag message.

1.2.1 Objects Folder

Git stores the database and the associated information in a set of folders and files inside the the folder *.git* in the root of your repository.

The folder *.git/objects* stores information about all objects (commits, trees, blobs and annotated tags). The objects are stored in files inside folders. The name of the folders are the 2 first characters of the SHA1 40 characters hexadecimal string. The name of the files are the other 38 hexadecimal characters of the SHA1. The information is compressed to save space.

1.3 Refs

SHA1 hash values are hard to memorize by humans. To make life easier to humans we use refs. A ref associate a name, easier to memorize by humans, with a SHA1 hash value, used by the computer. Therefore refs are pointers to objects. Refs are implementes by text files, the name of the file is the name of the ref and inside the file is a string with the SHA1 hash value. Tags and branches are example of refs. Tags are static references, i.e. tags are never updated, and branches are dynamic references, i.e. branches are always automatically updated.

1.3.1 Refs Folder

The *.git/refs* folder has inside the following folders *heads*, *remotes*, and *tags*. The *heads* has inside a ref for all local branches of your repository. The *remotes* folder has inside a set of folders with the name of all remote repositories, inside each folder is a ref for all branches in that remote repository. The *tag* folder has a ref for each tag.

1.3.2 Branch

A branch is a ref that points for a commit that is originated by a divergence from a common point. A branch is automatically atualize so that it always points for the most recent commit of that branch.

1.3.3 Heads

Heads is a pointer for the branch where we are. If we are in a commit that is not pointed by a branch we are in a detached HEAD situation.

1.4 Git Logical Areas

Git uses several spaces.

- Working tree - is your directories where you are working;
- Staging area or index - temporary area used to specify which files are going to be committed in the next commit;
- History - recorded commits;

All information related with the staging area and the history is stored in the *.git* folder.

1.5 Merge

Merge is a fundamental concept to git. It is the way you consolidate your work.

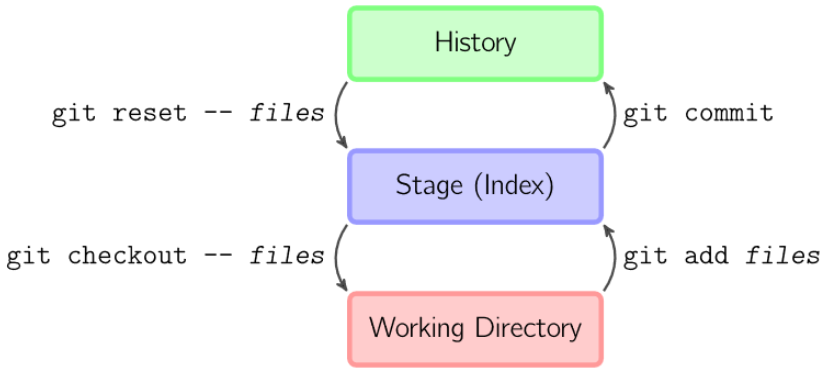

Figure 1.2: Git logical areas. Figure adapted from [2].

1.5.1 Fast-Forward Merge

It is used when there is a direct path between the two branches, the older branch is just updated.

1.5.2 Three-Way Merge

It is used when there is no direct path between the two branches. In this case a common commit is found considering the two branches and the merge is performed using a recursive strategy. In this process conflicts can occur. The recursive strategy is applied to each modified file and line by line. If the line was not modified in both branches the line is not modified in the merged file. If the line was modified only in one branch the line is going to be modified in the merged file. If the line is modified in both branches Git cannot make a decision and a conflict occur. So, conflicts occur when branches that change the same file in the same line are being merged.

1.6 Remotes

A remote is a repository in another location. The remote location can be the http://github.com or the location of any other Git server.

1.6.1 GitHub

GitHub is a Git server that stores public repositories. A GitHub user will have a user name and password and inside his or her account creates public repositories. This repositories can be transferred to a local machine using the command *git clone <repository url>*. The local and remote repository will be linked and the local repository can be updated using the *git fetch* or *git pull* command. The remote repository can be updated using the *git push* command. When the remote repository is cloned an alias, named origin, is created that points to that remote repository. Another way to create a GitHub repository is by forking another existente repository. Forked repositories are linked and they can be syncronized using pull requests.

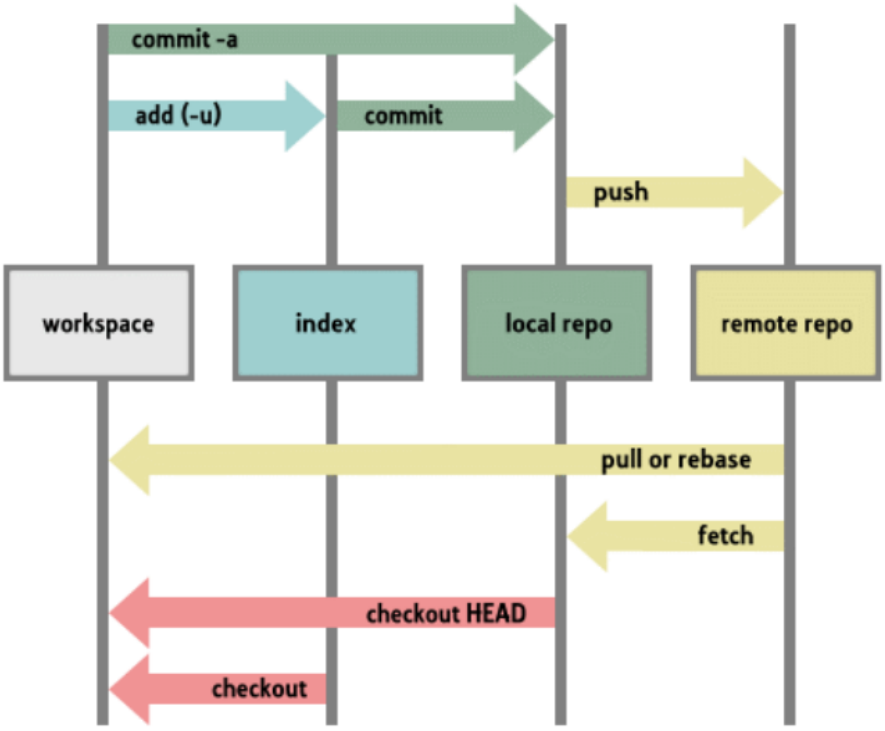

Figure 1.3: Git remotes.

1.7 Commands

1.7.1 Porcelain Commands

Porcelain commends are high-level commands.

git add
git add, adds a a new or modified file (or files) to the staging area.

git branch
git branch, lists local branches.
git branch -r, lists remote branches.
git branch -a, lists all branches.
git branch -u <remote>/<branch>, links the local branch in which you are in with a remote branch.
git branch –set-upstream-to=<remote>/<branch>, longer version of the previous command.
git branch -u <remote>/<branch> <local_branch>, links the local branch with a remote branch.
git branch –set-upstream-to=<remote>/<branch> <local_branch>, longer version of the previous command.

git cat-file
git cat-file -t <hash>, shows the type of the object identified by the hash value.
git cat-file -p <hash>, shows the contend of the file associated with the object identified by the hash value.

git checkout
git checkout -b <new_branch_name>, creates a new branch in the same position as the current branch and move to it.
git checkout -b <new_branch_name> <branch_name>, create a new branch in the position of <branch_name> and move to it.

git clean

git clean -f -d, removes from the working directory all untracked directories (d) and files (f).

git clone
git clone <url>, downloads the contend of the <url> repository, for instance http://www.github.com/netxpto/linkplanner.git, and creates a local repo.

git config
git config –global user.name "netxpto", sets the user name globally.
git config –global user.email "netxpto.gmail.com", sets the user e-mail globally.
git config –global user.emmail "netxpto.gmail.com", sets the user e-mail globally.
git config –global alias.<alias name> <commands>, creates a global alias for the commands.

git diff
git diff, shows the changes between the working space and the staging area.
git diff –name-only, shows the changes between the working space and the staging area, in the specified files.
git diff –cached, shows the changes between the staging area and the current branch history. Note that –cached or –staged are synonymous.
git diff –cached –name-only, shows the changes between the staging area and the current branch history, in the specified files.

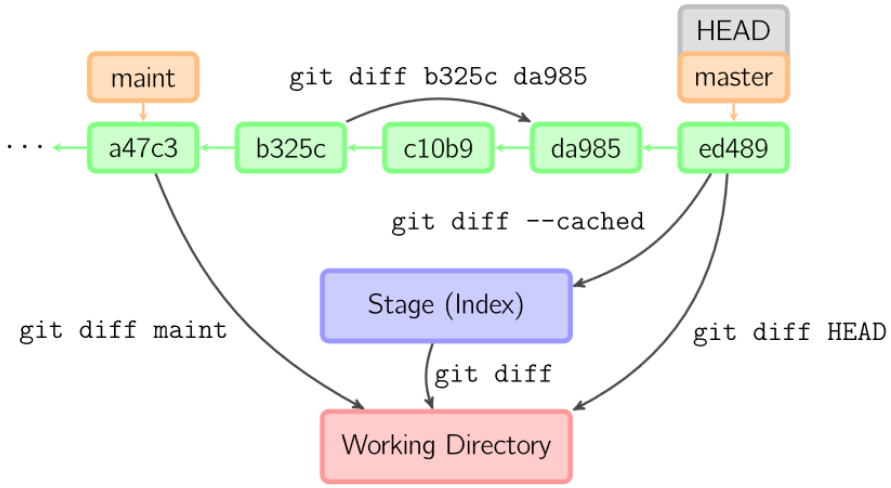

Figure 1.4: Git diff command. Figure adapted from [2].

git fetch
git fetch –all, downloads all history from all remote repositories.
git fetch <repository>, downloads all history from the remote repository.

git init
git init, inicializes a git repository. It creates the *.git* folder and all its sub-folders and files.

git log
git log, shows a list of commits from reverse time order.
git log –graph –decorate –oneline <commit1>..<commit1>, shows the history of the repository in a colourful way.
git log –graph, shows a graphical representation of the repository commits history.
git log –stat, shows the name of the files that were changed in each commit

git log –follow <file>, lists version history for a file, including rename.

git ls-files

git ls-remote

git merge
git merge <branch>, combines the specified branch's history into the current branch.

git pull
git pull, downloads remote branch history and incorporates changes.

git push
git push, uploads local branch commits to remote repository branch.

git rebase
git rebase <branch2 or commit2>, finds a common point between the current branch and branch2 or commit2, reapply all commits of your current branch from that divergent points on top of branch2 or commit2, one by one.

git remote
git remote, shows a list of existing remotes.
git remote -v, shows the full location of existing remotes.
git remote add <remote name> <remote repository url>, adds a remote.
git remote remove <remote name>, removes a remote.

git reset
git reset –soft <commit_hash_value>, moves to the commit identified by <commit_hash_value> but leaves in the staging area all modified files.
git reset –hard <commit_hash_value>, moves to the commit identified by <commit_hash_value> and cleans all modified tracked files.
git reset <commit_hash_value>, this is a mix reset (is the default reset), puts all modified files in the working area.

git reset <file>, unstages the file, but preserve its contents.

git reflog
git reflog, shows all commands from the last 90 days. Git only perform garbaged collection after 30 days.

git rm
git rm <file>, deletes the file from the working directory and stages the deletion.
git rm –cached <file>, removes the file from version control but preserves the file locally.

git show
git show, shows what is new in the last commit.

git stash
git stash, temporarily stores all modified tracked files.
git stash –list, shows what is in the stash.
git stash pop, restores the most recently stashed files.
git stash drop, discards the most recently stashed changeset.

git status
git status, lists all new or modified files to be committed.

1.7.2 Pluming Commands

Pluming commands are low-level commands.

git cat-files
git cat-files -p <sha1>, shows the contend of a file in a pretty (-p) readable format.
git cat-files -t <sha1>, shows the type of a object, i.e. blob, tree or commit.

git count-object

git count-object -H, counts all object and shows the result in a (-H) human readable form.

git gc
git gc, garbage collector, eliminates all objects that has no reference associated with.
git gc –prune=all

git hash-object
git hash-object <file>, calculates the SHA1 hash value of a file plus a header.
git hash-object -w <file>, calculates the SHA1 hash value of a file plus a header and write it in the *.git/objects* folder.

git merge-base
git merge-base <branch1> <brach2>, finds the base commit for the three-way merge between <branch1> and <branch2>.

git update-index
git update-index –add <file name>, creates the hash and adds the <file_name> to the index.

git ls-files
git ls-files –stage, shows all files that you are tracking.

git write-tree

git commit-tree

git rev-parse
git rev-parse <ref>, return the hash value of <ref>.
git rev-parse <short_ hash_ value>, return the full hash value associated with <short_hash_value>.

git update-ref

git update-ref refs/heads/<branch name> <commit sha1 value>, creates a branch that points to the <commit sha1 value>.

git verify-pack

1.8 Navigation Helpers

<ref>^, one commit before <ref>.
<ref>^^, two commits before <ref>.
<ref>~*5*, five commits before <ref>.
<ref1>..<ref2>, between commit <ref1> and <ref2>.
<branch>^*tree*, identifies the tree pointed by the commit pointed by <branch>.
<commit>:<file_ name>, identifies the version of a file in a given commit.

1.9 Configuration Files

There is a config file for each repository that is stored in the *.git/* folder with the name *config*.

There is a config file for each user that is stored in the *c:/users/<user name>/* folder with the name *.gitconfig*.

To open the *c:/users/<user name>/.gitconfig* file type:

git config –global -e

1.10 Pack Files

Pack files are binary files that git uses to save data and compress your repository. Pack files are generated periodically by git or with the use of gc command.

1.11 Applications

1.11.1 Meld

1.11.2 GitKraken

1.12 Error Messages

1.12.1 Large files detected

Clean the repository with the BFG Repo-Cleaner.

Run the Java program:

java -jar bfg-1.12.16.jar --strip-blobs-bigger-than 100M

This program is going to remote from your repository all files larger than 100MBytes. After do:

git push --force.

1.13 Git with Overleaf

You can use git with overleaf. For that you have to create a project on overleaf. Associate with that overleaf project it is also create a git repository. The address of that git repository is almost the same as the overleaf project that you can obtain going to the overleaf Menu/Share. Let's assume that the overleaf project address is:

https://www.overleaf.com/12925162jkwbhrdkwrfm

In this case the repository address is https://git.overleaf.com/12925162jkwbhrdkwrfm
The only change was the replacement of **www** by **git**.

Now you can just do
git clone https://git.overleaf.com/12925162jkwbhrdkwrfm
and clone your repository.

You can also do
git push https://git.overleaf.com/12925162jkwbhrdkwrfm

Bibliography

[1] Scott Chacon and Ben Straub. *Pro Git, 2nd Edtion*. Apress, 2014.

[2] Feb. 4, 2019. URL: `https://marklodato.github.io/visual-git-guide`.