Git Handbook

This handbook is aimed at all those developers who look for a quick but quite complete introduction to this powerful version control system. The version used in all the examples is 1.7.5.3.

1 Some basic definitions

Working directory: this is commonly your project root directory, which contains all the files you are currently working on. The content of this working directory can change when you switch on another branch.

Index: the Git index contains all the files that will go in the next commit. The index may of course differ from the working directory (for example for cause of new changes you don't want to put in this commit).

A git project is identified by the git directory, the hidden directory named ".git" in your working directory. This contains all the necessary information.

2 Cloning or creating a repo

A new repository can be created from existing sources (or no sources) using a simple

```
$ git init
```

in the project root directory. This repository can be copied elsewhere on your system using

```
$ git clone <path to repository>
```

The original repository can also be located in another machine. The `git-clone` command allows us to obtain the repository knowing the URL, using:

```
$ git clone <repository>
```

which creates a new directory with a copy of what is on the server.

The remote repository's URL can be later changed with

```
$ git remote set-url origin <URL>
```

3 Basic configuration.

All the configuration for a project is read from the file `config` in the git directory. You can edit this file manually or via the `git-config` tool.

The file is divided into sections formatted like this:

```
[section]
     variable1 = value1
     …
     variableN = valueN
```

If we want to change `value1` into `newvalue` the command is:

```
$ git config section.variable1 newvalue
```

A new variable can be added with

```
$ git config --add section.variable value
```

There is also a global configuration file in your home directory, called `.gitconfig`, that refers to all the git projects. It is modified adding the `--global` option to the previous lines.

The first variables you'd want to change are `name` and `email` in the `[user]` section: they will appear in your commits. There's of course a lot of sections, subsections and variables to be set in the

configuration files: a list of those is found in the `git-config`(1) manpage.

4 Everyday use.

After you have obtained your own copy of the remote repository, you make some changes to the files, and you want to share them with the rest of the team. The common procedure is:

```
$ git status
```

shows you which files have been modified since the last commit. It shows also all the untracked files (that is, the files git doesn't consider part of the project) in a different section.

Now you want to add those file to the index, ready for the commit to take them. The command is:

```
$ git add <filename>
```

if the file is untracked and you wish to add it to the project, `git-add` does it automatically for you.

All the files added with this command are so added to the index. Finally:

```
$ git commit
```

creates the new commit. You can pass a commit message as a string next to the `-m` option, otherwise git will open your default editor (set in the global config) so that you can write one.

There's a shortcut, the option `-a`, which simply puts into the commit all the modified files (but leaves the untracked ones alone).

Note that this commit only exists locally, stored in a particular object in your git directory. If you're sharing a remote repository with the other developers, the commit must be pushed there using:

```
$ git push
```

which by default uses the remote URL found in the configuration.

In the same way, you surely will have to update your local tree from the remote repository:

```
$ git pull
```

Note that this involves two steps: getting the changes from the remote server, and merging them into the local tree. If you tree have diverged from the remote one and you didn't commit your changes, the automerge procedure can fail, leaving unsolved conflicts in some files, and you will be warned about this on the command line.

In this case, before you can go on the conflicts must be solved. Each conflict is signed in its file by lines looking like this:

```
<<<<<< HEAD
some code
=======
some different code.
```

```
$ git diff
```

in this situation shows all these lines for you (see the Diff section for further explanations about this tool).

After solving all the conflicts, add the files to the index and then run

```
$ git commit
```

This will create a solving commit with a generated message about the merge.

5 Patches.

Another way of showing your new commit to other people is creating and sending a patch; this is for example useful if you don't have permissions to push the commit onto the remote repository.

After creating your commit as usual, run

```
$ git format-patch
```

This command creates a patch for every commit you have made that isn't present on the remote repository. The `-n` option creates a patch for the topmost n commits.

If you receive a patch which you want to apply to your tree, use

```
$ git apply <file>
```

`git-apply` may be used to determine what changes would the patch bring if applied (running a sort of simulation). Good options in this context are:

`--summary` : syntetic summary of the changes to apply

`--check`: see if the patch would be applicable, detecting errors.

`--cached`: apply the changes directly in the index without touching the working tree.

6 Getting diffs.

This command generally “show changes”. Let's see some examples:

```
$ git diff
```

shows the changes between the index and your current working tree. Can be used to discover in details what `git-status` is talking about.

```
$ git diff branch1..branch2
```

Shows differences between the tips of the two branches.

7 Reviewing history.

```
$ git log
```

shows all the commits made from the origin of the project to the current day, with author, date and message. The format in which the history is created can be vary according to many options. One useful option if you want to parse the history with other tools is `--pretty=oneline`, which shows each commit in one line.

8 Branches, rebasing and merging.

A single project can contain more than one “branch” of development. That is, at a certain point the development can diverge in two or more different lines. The default branch is called master, while other branches can be created with the command

```
$ git branch <branchname>
```

All branches are listed using

```
$ git branch
```

which also shows the branch you are currently on, marked with *.

Note that the new branch is originally set as a copy of the branch you're standing on during the creation: this is called the upstream branch. It's however possible to create a so-called “empty branch”, a branch that has no upstream branch and no history in common with your current development. They're usually created for generated documentation.

A branch can exist only locally (for which the above command is sufficient) but if you want the remote repository to have it too type:

```
$ git remote add origin <branch>
```

Finally, if another member of the team created the branch remotely with the previous command and you want to have it locally, use

```
$ git pull origin <branch>
```

You can switch between branches using

```
$ git checkout <branch>
```

This will fail if the current branch has some changes not still committed that would be overwritten by the files in the other branch. To solve this problem, either make the requested commit or read the stashing section below.

To delete a branch, use

```
$ git branch -d <branch>
```

This command issues an error if the newbranch isn't fully merged to its upstream, while `-D` always deletes it.

Ok, now you can start developing and committing on the new branch while your team developers continue their work on the master one.

After some commits, the history of the two branches have surely diverged. So, how do you merge them back into one development branch without messing everything up?

The situation: we have two diverging branches named master and experimental (of course the experimental's upstream branch is master).

The simplest way is to run, while on master

```
$ git merge experimental
```

Any conflict will be signalled the same way it's done during a pull; you solve it and then commit the result; then the branch can be deleted, and you have a new merge commit reflecting a “mix” of the two development branches.

But if you prefer to keep the history of your project as a series of normal commits, you can choose to do a rebase instead:

```
$ git checkout experimental
$ git rebase origin
```

This will remove each commit from experimental (saving them in a temporary hidden directory), apply all the diverged commits from master, and then your commits again.

This process, as the merge does, can discover lots of conflicts in different files. The command will then stop warning you about the files it couldn't automerge. After you solved the conflicts, run

```
$ git rebase --continue
```

to start the process again or

```
$ git rebase --abort
```

to stop it.

After the rebase, you run the merge as above and there won't be any conflict.

9 Reverting changes or commits.

At any point during your development you can delete all the changes you made so that the working tree reflects the last commit, using

```
$ git reset --hard HEAD
```

this will delete even the changes you already added to the index.

This can be done also for a single file:

```
$ git checkout -- <filename>
```

takes it back to the version currently in the index, while

```
$ git checkout HEAD <filename>
```

to the last commit status.

If you already committed what you regret and want to undo, there's two main ways:

if the changes already went on the remote repository, you have to create a new commit to fix them.

If the changes are only on a local commit, there's a command that does this for you:

```
$ git revert HEAD
```

If you want to modify the current commit without having to create a new one

```
$ git commit --amend
```

corrects the commit based on the current state of the index.

10 Stashing.

Say you are switching branches but `git-checkout` is failing because some changes in the current branch would be overwritten, or you have to do a pull and the merge fails because of new changes. You're not ready for a commit yet, but you don't want to lose those changes, just put them somewhere until you can work on them again.

Git-stash comes in your aid in these situations. Just run

```
$ git stash “optional description”
```

Your changes will be saved and the working tree reverted back at the last commit status.

You can see the list of stashes with

```
$ git stash list
stash@{0} branch “description1”
stash@{1} branch “description2”
…
```

```
$ git stash show <stashname>
```

(where `stashname` is for example `stash@{0}`) shows the diff between what is recorded in the stash and the current working tree. This command takes all the format options valid for `git-diff`.

```
$ git stash pop <stashname>
```

applies the given stash to the top of the working tree. The working tree must match the index. An empty stashname by default means stash@{0}. The stash is removed from the list after pop: if you want to keep it, use apply instead of pop.

Applying the stash can of course fail with conflicts. In this case the stash is never removed from the list and you have to solve the conflicts manually.

```
$ git stash drop <stashname>
```

The stash is deleted without being applied. The same rules for stash names are valid here.

```
$ git stash clear
```

deletes all the entries in the stash list.

11 Some advanced topics

11.1 Regular expressions with git-grep

`git-grep` is a tool to search thorough your entire project for some pattern. Unlike the UNIX grep, git grep is also able to search in previous versions on the code without having to check them out.

```
$ git grep <regexp-pattern>
```

is the simplest use.

Some useful options are:

`--no-index`: searches also the untracked files.

`--ignore-case`: enables a case insensitive search.

`--invert-match`: shows non-matching lines.

`--line-number`: shows the line numbers too.

`--count`: shows only the number of matching lines.

11.2 Searching an entire history.

The `git-bisect` tool helps in those situations where a previous commit broke down something in the project, but you don't know the specific commit. This tool performs a binary search to speed up the search.

```
$ git bisect start
```

starts the procedure. Now you need to mark the bad version (normally the current one)

```
$ git bisect bad
```

and the last good one you're sure about

```
$ git bisect good v.X
```

this means version X is the last tested one you're sure is good. Instead of version numbers, you can use the commit SHA. Every commit is identified by a unique 40-digit SHA number: you can read it from `git-log`.

The tool now prints something like:

```
Bisecting: X revisions left to test after this.
```

At this point you're moved into a temporary branch as the tool checks out different versions (warning you about roughly how many revisions are left to check); you test them and tell git if they're good or bad with

```
$ git bisect good/bad
```

After a certain number of tests (based on how the X number printed before) git will print all the information (author, message, etc.) about the commit which caused the error.

At any point of your search,

```
$ git bisect visualize
```

shows how much revisions are left to test. At the end:

```
$ git bisect reset
```

takes you back on your normal working branch.

11.3 Blaming people :)

```
$ git blame <filename>
```

prints out all the lines in the filename preceded by the author's name and the first digits of the SHA identifier of the commit which “caused” that line (or “Not committed yet”); given the uniqueness of the SHA hashes, those digits are often enough to identify the commit.

11 Conclusion.

Git is a very powerful system for managing a big project; the topics covered here are merely an introduction that will help you using git for day-to-day work without too much problems. Of course, each manual page can navigate you through a much more complete knowledge of the possibilities of the tool you're going to use.

shainer <shainer@chakra-project.org>