Git documentation

Miguel Sánchez de León Peque

September 22, 2011

Contents

List of Figures

List of Tables

1 Introduction

1.1 Collaborative projects

When software is developed by a group of programmers, some method is necessary to control access and track changes made in source code.

Version control (also known as revision or source control) is the art of managing software through its process of development.

Every change is usually identified by a number or letter code which is called the revision number (or simply revision). Each revision is associated with a time stamp and the person making the change.

1.2 Distributed version control systems

A distributed version control system allows developers to work on a project without needing to be connected to a common network.

In centralized systems (such as the well known Subversion) there's a client-server approach, while in distributed systems there's a peer-to-peer approach. That means there are only working and full functional copies: common operations (such as commits, viewing history, and reverting changes) are fast, because there is no need to communicate with a central server. Furthermore, this provides natural protection against data loss. Communication is only necessary when pulling or pushing changes from or to other peers.

2 Git

2.1 About Git

Git is a distributed version control system which was initially designed by Linus Torvalds[1] for the Linux kernel development. Just as the Linux kernel, Git has been released as free software[2] under the General Public License (GPL).

It's probably the best version control system ever made as it has probed to be amazingly fast (specially on POSIX-based systems) and versatile. It's the most used in new software projects and many older projects have started using it. Some important projects that are currently using Git are: Linux kernel, GIMP, Git (itself), VLC media player, Android, Eclipse, jQuery, PostgreSQL, Wine, Chromium (Google Chrome), Perl, KDE, GNOME, Qt 4, GTK+, phpBB, many GNU projects (GNU Compiler Collection, GNU C library, grep, tar...), Drupal, X.Org, rsync, and many, many more.

2.2 Git overview

Most projects have a main or master branch where most of the development occurs. Changes which will quickly add new functionality or fix bugs are made here.

Whenever you want to start developing a bigger new feature or code refactoring, you'd better not be in the master branch, as the first changes will probably introduce some errors to the master branch, where others may be coding at that time. If you break the project's functionality in the main branch, you'll stop other developers from doing their job.

[1] Linus Torvalds started the Linux project back in 1991. The Linux kernel has become the *de facto* standard kernel for almost all GNU distributions.

[2] The term *free* refers to software that can be used, studied, and modified without restriction, and which can be copied and redistributed in modified or unmodified form (*free* as in *freedom*).

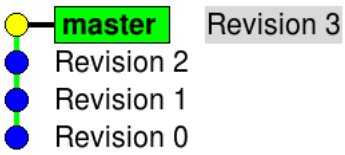

Figure 1: Development in master branch

The best thing you can do is to create a new branch. That way, you'll have all the advantages of the version control system, but your changes wont affect (for now) other developers. You may want to share this branch with other developers (uploading it to the server) or keep it locally until the job is done.

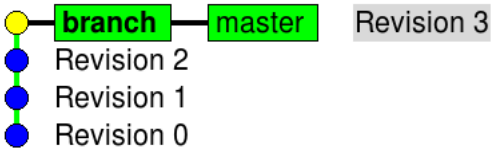

Figure 2: Creating a new branch

Once you are in the new branch, you can start making changes just as in the master branch.

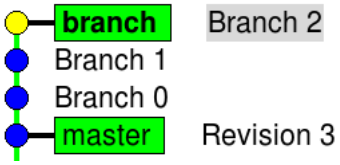

Figure 3: Development in your own branch

Notice that while you are working on your branch, development may occur simultaneously in the master branch. But don't worry, because those changes wont affect your branch either.

Once you are done with your branch, you'll only need to merge those changes with the master branch.

2.3 Git interface

Although there might be some GUIs for Git, we'll be using the command-line interface which, once you get used to it, will bring you the way to reach the best results.

Every command you'll use will start with the word `git`, followed by the task you want to perform. Some examples are: `git status`, `git add <filename>`, `git commit -m 'Changes you made'`...

We'll be learning more about this commands soon, but first, why don't we check out an example?

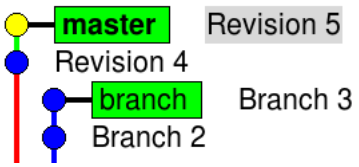

Figure 4: Simultaneous development

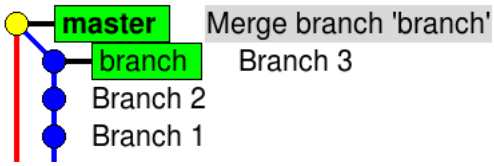

Figure 5: Merging changes

3 A Git example

3.1 First steps on Git

Installing Git is really simple in GNU/Linux systems. You can use your package manager or, alternatively, the terminal. For example, in Fedora, installing Git would be as easy as:

```
$ su -c 'yum install -y git gitk'
```

To initialize a Git repository, create an empty folder which will contain the full project.

```
$ mkdir git-test
$ cd git-test
$ git init
```

The command `git init` initializes an empty Git repository in `git-test` folder. That will allow you to perform many other Git operations in the future, when some changes are made within the project.

Now, lets create a file named `test-file.txt`, with one text line:

```
$ echo Line 0 > test-file.txt
```

That would be our initial commit.

```
$ git add .
$ git commit -m 'Initial commit'
```

We'll use the `git add` command whenever we want to confirm changes for a commit. In this case, we are adding the whole current folder '.', which is `git-test`. After adding the changes we made, we'll make the commit to create the next revision. A commit is essentially giving a description to the changes we added. The option `-m` will let us easily add a short description to the commit. The description must be written after the `-m` option and between simple quotes.

You may ommit the `-m` option but, doing so, you'll enter a console editor (such as Vi) for writting your description. Managing this editors is not the matter of this documentation.

There's a command we can use to see the project status. It will show us the branch we are working in, the files or folders we changed (added, deleted or modified), which of them are already added and if there are still commits waiting to be uploaded to the server. In our example, we can see we are in the `master` branch, and we have no changes to be commited.

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

Lets say we make a change to the `text-file.txt`. Lets delete the line we wrote and write a new one.

```
$ echo New line 0 > test-file.txt
```

Now, our status has changed:

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working dir.)
#
# modified:   test-file.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

We can see `text-file.txt` has been modified. As Git is suggesting, we've got two options to perform: add the changes for a later commit (`git add text-file.txt`) or discard those changes (`git checkout -- text-file.txt`). But before continuing, lets introduce a new tool: `gitk`.

`gitk` is a graphical interface that will allow us to easily checkout the project's history and also the local changes we make. Lets see how it looks like.

```
$ gitk
```

On the upper-left of the window, we can see the history tree, where each commit is represented by a circle and its description. In our example, this consist only in the initial commit and a red circle which represents some changes that haven't been added yet.

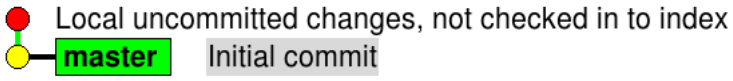

Figure 6: Unstaged changes

At the right of each revision or commit, we can see the name of the committer with his or her email and the date the commit was made.

Notice that you haven't configured your name and email address, so, although you made the initial commit, your name and email wont be displayed (instead, you'll find some session and system information).

We can navigate through the history tree to see the changes that were made in each of them. If we select the upper line (local uncommitted changes), we'll see the following information:

```
@@ -1 +1 @@
-Line 0
+New line 0
```

Figure 7: Diff file

Red lines represent those which have been deleted, and green lines those which have been added. On the right, we can see a list of the files that have been modified.

So, lest now close the window and go back to the terminal. We want to confirm the changes we made, as we have already reviewed them with the `gitk` tool. So, we need to add them:

```
$ git add test-file.txt
```

Lets see what happened.

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# modified:   test-file.txt
#
```

And if we use the `gitk` tool, we'll see the upper line has now a green circle, which represents changes waiting to be commited.

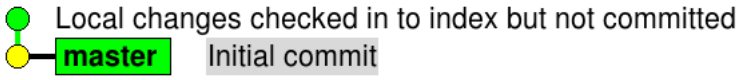

Figure 8: Changes to be commited

Now we can do two things: make the commit (`git commit -m 'Changed line in test-file.txt'`) or revert the last operation (`git reset HEAD test-file.txt`) to unstage the file. If the perform the last operation, we'll see again a red circle with `gitk`, which means changes need to be added again. But normally, we want to commit the changes:

```
$ git commit -m 'Changed line in test-file.txt'
[master 107f4e6] Changed line in test-file.txt
 1 files changed, 1 insertions(+), 1 deletions(-)
$ git status
# On branch master
nothing to commit (working directory clean)
```

The `gitk` tool will show us a new commit:

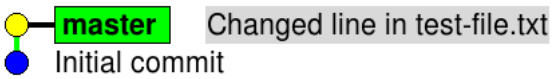

Figure 9: New revision in the history tree

Normally, we'll keep working and making new commit, but there's a chance that (even if we "checked" it before with `gitk`) we made an erroneous commit. Again, we've got two options: we can make a new commit to correct the last one or, if you didn't already pushed your changes to the server (which is something we haven't talked about yet), undo the last commit in the local tree:

```
$ git reset --soft HEAD^
```

That way, we'll go back to the previous status.

Using the `--soft` option is highly recommended, as other options (as `--hard`) will also unstage the changes. Try to always perform this operations step by step. Remember: it's for your own safety!

3.2 Branching

Now that we know how to make commits, lets get into branching. You can easily change between branches using `git checkout <branch_name>` but, you'll need to add the `-b` option if the branch doesn't exist (that way, you'll create a new branch called `<branch_name>`).

```
$ git checkout -b newbr
Switched to a new branch 'newbr'
```

Now we are exactly at the same revision as before, but in a new branch.

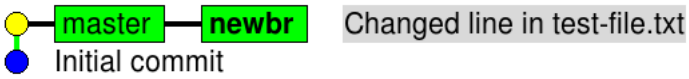

Figure 10: New branch

Important: `gitk` shows by default the history of the branch we are currently in. If we want to see the full tree, we need to use the `--all` option: `gitk --all`. For now on, we'll be using that option whenever we need to use the `gitk` tool.

So, lets add a new line to our `test-file.txt` and commit the changes:

```
$ echo Line 1 written from newbr branch >> test-file.txt
$ git add test-file.txt
$ git commit -m 'Added line 1 to test-file.txt'
[newbr 405f629] Added line 1 to test-file.txt
 1 files changed, 1 insertions(+), 0 deletions(-)
```

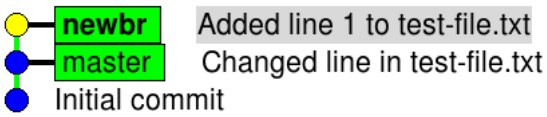

Figure 11: New commit in `newbr` branch

Now, our branch `newbr` is ahead in the history tree as the commit we made hasn't affect the `master` branch.

But, what happens if a new commit is made in the `master` branch? Lets change to it and make a new commit from there:

```
$ git checkout master
Switched to branch 'master'
$ echo Line 1 written from master branch >> test-file.txt
$ git add test-file.txt
$ git commit -m 'Added line 1 to test-file.txt'
[master c373bbc] Added line 1 to test-file.txt
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Notice that when we change from one branch to another, all the files within the project's folder change to the situation in which they are in the new branch.

Now the `master` branch is ahead in the history tree.

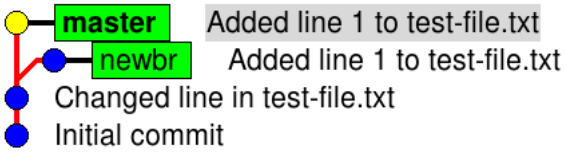

Figure 12: New commit in `master` branch

Lets say we finished making changes in `newbr` branch. Now, we would like to merge the changes we made with the master branch (we need to do that from the branch we want to merge the changes to: in this case from `master`).

```
$ git merge newbr
Auto-merging test-file.txt
CONFLICT (content): Merge conflict in test-file.txt
Automatic merge failed; fix conflicts and then commit the result.
```

What happened here? Well, sometimes when you are working in a project with other people, you may find that others have made changes to the same sources you are working on in your branch. In our example, the second line of the file was modified in both branches, and Git doesn't know which one should it take.

Lets see what Git says about this status:

```
$ git status
# On branch master
# Unmerged paths:
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#
# both modified:      test-file.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

For solving merging conflicts manually, we'll first a look at the history tree with `gitk`. We'll find which lines conflict in the file.

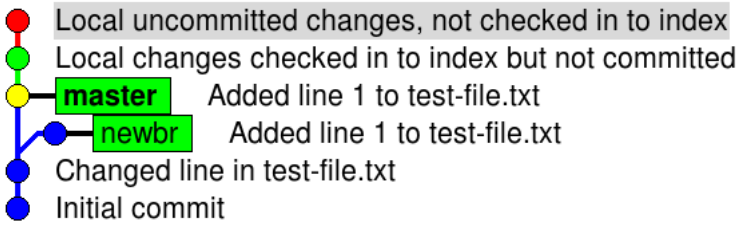

Figure 13: History tree after a merging conflict

```
  New line 0
++<<<<<<< HEAD
 +Line 1 written from master branch
++=======
+ Line 1 written form newbr branch
++>>>>>>> newbr
```

Figure 14: Conflicting lines

Those lines in red color, which are under the `HEAD` tag, represent changes in our current branch (`master`). Those in blue, which are over the `newbr` tag, represent changes in the `newbr` branch. What we need to do is to open the file and remove those lines we don't want to keep there.

Lets say we want to keep the changes we made in the `newbr` branch. We need to delete the line in red (which corresponds to the line added from the `master` branch) and also the delimiter lines that Git added to the file to separate the conflicting lines.

If we do that correctly, we can complete the merge just as if we were making a normal commit:

```
$ git add test-file.txt
$ git commit -m 'Merge branch newbr into master (conflicts solved)'
[master aea5947] Merge branch newbr into master (conflicts solved)
```

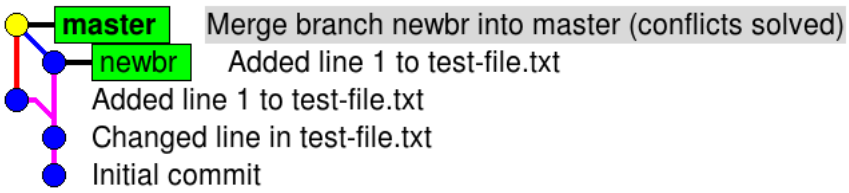

Figure 15: Branch `newbr` merged into `master`

Conflicts while merging branches may happen to you from time to time, specially if you modified many files or if many other people are working at the same time in the project. Git usually solves most of the conflicts by it self though.

4 Working with Git in the cookie-libs project

4.1 Cloning an already existing repository

The cookie-libs project is already hosted in a server, more precisely in SourceForge. That means you'll need to have a SourceForge account and you'll need to ask the administrator to join the developers team in order to have write permissions in the server.

You can contribute to the cookie-libs project without being part of the developers team, generating and sending patches to them. Anyway, we'll not cover how to make patches within this document.

Once you've joined the development team, you can clone the server's repository to create a local copy of the project to work with. The `USERNAME` is your SourceForge user's name and the password you are asked for is your SourceForge's as well:

```
$ git clone ssh://USERNAME@cookie-libs.git.sourceforge.net/gitroot/\
cookie-libs/cookie-libs

Cloning into cookie-libs...
USERNAME@cookie-libs.git.sourceforge.net's password:
remote: Counting objects: 452, done.
remote: Compressing objects: 100% (447/447), done.
remote: Total 452 (delta 251), reused 0 (delta 0)
Receiving objects: 100% (452/452), 5.24 MiB | 109 KiB/s, done.
Resolving deltas: 100% (251/251), done.
```

Once the process has been completed, you'll see a folder called `cookie-libs` in which you'll find the entire project. Inside the folder, you'll be able to use the Git command line interface and the `gitk` tool as you did before, as it's a local repository as well.

```
$ cd cookie-libs
$ git status
# On branch master
nothing to commit (working directory clean)
```

Before starting making changes, as we'll be now sharing our work with the rest of the developers, we'll need to configure Git so that it will associate our commits with our name and email address[3].

```
$ git config --global user.name "Here Comes My Name"
$ git config --global user.email "here.comes@my.email"
```

Do not forget the `" "` around your name and email! If you want to check your configuration, just execute:

```
$ git config -l
```

You should see your name and email address displayed (among other things).

[3]Using our real name and email address is important for other developers to know who are you and how to contact you in case they need it.

4.2 Git, from local to server

Although everything is very similar, you may notice one difference: there are two tags in the latest revision. Those tags correspond to the local branch (`master`) and the remote branch (`remotes/origin/master`).

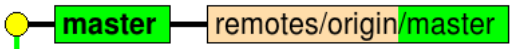

Figure 16: Both the local and remote `master` branches

You can work in your local copy with the same commands we have learned, but, once you make a commit, it will be written only in your local `master` branch.

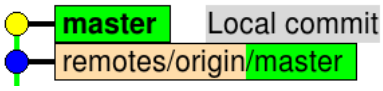

Figure 17: New local commit

Of course, we can keep doing local development but, at some time, we will want to share our changes with the other developers, so we will upload them to the server (also it is safer to have a copy of your local work). To do so, we'll use the `push` command once we've already made the commit in our local repository.

```
$ git push
```

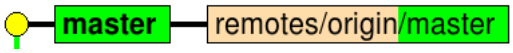

Figure 18: Again, both local and remote branches are at the same revision

Although it is possible, we encorage you not to undo commits that you have already pushed to the server. If you want to fix your last commit, just create a new one with the necessary changes. If you want to undo the last commit, you can revert the changes you made:

```
$ git revert <SHA1 ID>
```

Where `<SHA1 ID>` is the commit's ID (or the begining of it)[4].

[4]You can execute `git revert 0e81e5588b4da07a4ff8388fee01dfed776a7e14` or `git revert 0e81e`

Just as simple! There's only one more thing: how can we download changes from the server? (those which other developers made). That's a very simple task too:

```
$ git pull
```

Remember to use `git pull` *frequently, so you can have your local copy updated and notice what other developers are doing within the project.*

4.3 Remote branches

Working with remote branches has the same problem. Local changes do not apply until we execute the corresponding command. Once we have created a local branch, we can initialize it in the server with:

```
$ git push origin new_branch_name
```

Once the branch has been initialized, we can share our changes with a normal `git push`. If we want to download changes in a specific branch, we can use:

```
$ git pull origin branch_name
```

Again, it is not a good idea to delete branches from the server once we have pushed them. Anyway, if we already finished working with one branch and it has been already merged with the `master` branch, we may want to delete it to keep the history tree clean. First, we'll delete our local branch:

```
$ git branch -d branch_name
```

If we started a branch that we don't want to merge with the `master` *branch (we may have started developing a crazy idea) and we want to delete it either way, we can use the command* `git branch -D branch_name`*.*

After that, and after consulting other developers in the project, we can delete the remote branch:

```
$ git push origin :branch_name
```

5 Git command reference

5.1 Commands we have reviewed withing this document

Table 1: Initial configuration

Command	Options	Description
`clone`	`<git_repo>`	Clone a remote Git repository
`config`	`--global user.name <user_name>`	Configure user's name
	`--global user.email <user_email>`	Configure user's mail
	`-l`	List configuration settings

Table 2: Simple commands

Command	Options	Description
`status`	(none)	Show current status
`add`	`<file_name>`	Confirm changes to be commited (new files)
`rm`	`<file_name>`	Confirm changes to be commited (deleted files)
`commit`	(none)	Edit commit's description with a command line editor
	`-a`	Commit all changes (even those not confirmed yet)
	`-m`	Commit's description
`revert`	`<commit_ID>`	Revert one commit
`reset`	`--soft HEAD^`	Undo last local commit
	`--hard HEAD^`	Undo all the changes in last local commit
	`HEAD <file_name>`	Unconfirm added changes
`pull`	(none)	Download remote changes from the server
`push`	(none)	Upload local commits to the server

Table 3: Working with branches

Command	Options	Description
`push`	`origin <new_branch_mame>`	Initialize new local branch in the server
	`origin :<branch_name>`	Delete remote branch (use with care!)
`pull`	`origin <branch_name>`	Download changes from the remote branch
`merge`	`<branch_name>`	Merge `<branch_name>` to the current branch
`checkout`	`<branch_name>`	Move to `<branch_name>`
	`-b <new_branch_name>`	Create a new branch `<branch_name>`
`branch`	`-d <branch_name>`	Delete `<branch_name>` (once merged)
	`-D <branch_name>`	Delete branch irrespective of its merged status

5.2 More useful commands

Table 4: Working with tags

Command	Options	Description
`tag`	(none)	List all tags
	`-a <tag_name>`	Add the tag name
	`-m '<tag_desc>'`	Add the tag description
`push origin`	`--tags`	Push all tags to the server
	`<tag_name>`	Push tag (with <tag_name>) to the server

Example:

```
$ git tag -a v0.1 -m 'versión 0.1'
$ git push origin v0.1
```

Table 5: Patching

Command	Options	Description
`format-patch`	`origin/branch_name`	Create a patch for remote `branch_name`
`apply`	`<patch_name>`	Apply patch in the project

Example:

```
$ git format-patch origin/master
$ git apply filename.patch
```