Git Introduction

Author : Matthieu Pizenberg
Version : V1.0
Date of version : 05/02/2015

Original material : The work on which this document is based is a book attributed to Scott Chacon and Ben Straub. It was also released under a CC BY-NC-SA 3.0 license and published by Apress.

Foreword

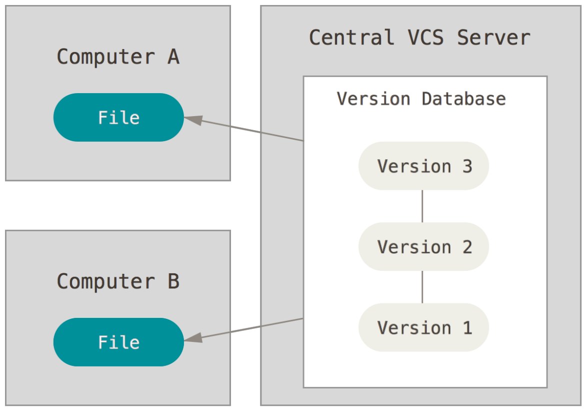
There are lots of amazing tutorials and books about Git so I can not compete with them. In this document, I will recap for us the main features of Git to explain why it is so awesome ! For that I will follow the same plan as the one of that book (http://git-scm.com/book/en/v2) but I will keep only interesting or vital information for us.

Table of contents

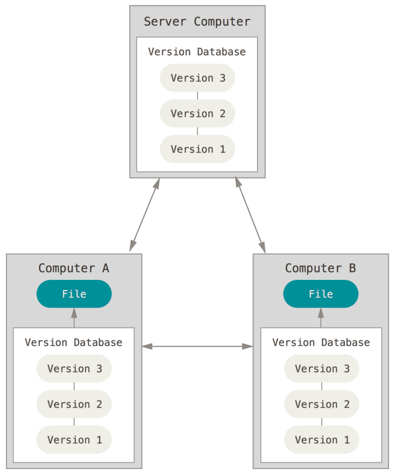
1 Getting Started

1.1 About Version Control

Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. To work in team we generally use CVCS or DVCS (Centralized / Distributed Version Control System).



CVCS



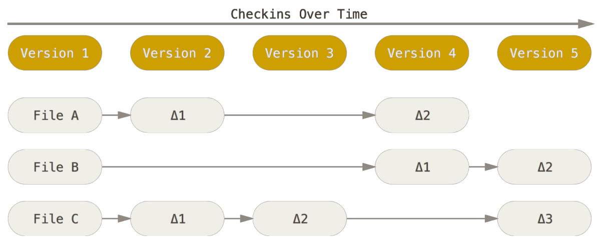
DVCS

In DCVS (like Git) each client has a clone of the server which means that he can continue to work with version control even if the server crashes or if he does not have access to the net. This is a great improvement from CVCS where you need access to the server to make commits. It also means that anyone can restore the server if it crashes.

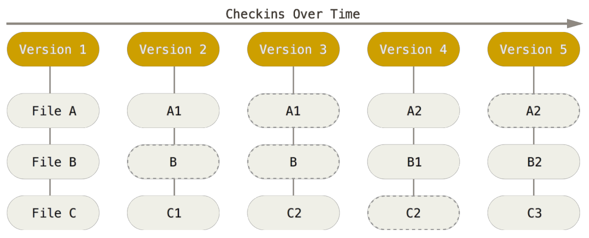
1.2 Git Basics

1.2.1 Snapshots not differences

While many others VCS use differences to store changes over time Git uses snapshots. For each commit, modified files are snapshotted and other files are just linked from the previous commit. Snapshots allow Git to have an efficient branching model.



differences



snapshots

1.2.2 Nearly every operation is local

As it is said in the book :

If you're used to a CVCS where most operations have that network latency overhead, this aspect of Git will make you think that the gods of speed have blessed Git with unworldly powers.

And of course it allows to work offline !

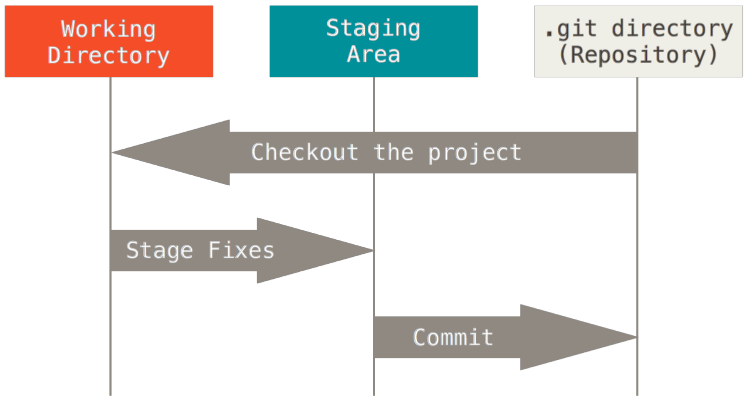
1.2.3 Git has integrity and generally only adds data

Everything in Git is checksummed with SHA-1 hash (looks like this : *24b9da6552252987a a493b52f8696cd6d3b00373*). Furthermore nearly every operation adds data which means it is hard to do something which is not undoable (see "Undoing things").

1.2.4 The three states

It may be a little complex to talk about it now but it is **very important** to understand. Also you might understand it better when practicing git.

Git has three main states that your files can reside in : committed, modified, and staged. Committed means that the data is safely stored in your local database. Modified means that you have changed the file but have not committed it to your database yet. Staged means that you have marked a modified file in its current version to go into your next commit snapshot.



The **Git directory** is where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you clone a repository from another computer.

The **working directory** is a single "checkout" of one version of the project. This is what you see in your browser, the files you can modify. For example when you "checkout" to a certain commit (or a branch) your working directory will change.

The **staging area (or "index")** is a file, generally contained in your Git directory, that stores information about what will go into your next commit.

The basic Git workflow goes something like this:

1. You modify files in your working directory.
2. You stage the files, adding snapshots of them to your staging area.
3. You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

1.3 First-Time Git Setup

I skipped the command line and installation parts (installation), you are able to do it alone.
It is good to configure once for all what can be configurable.

Your personal configuration file generally is `~/.gitconfig`
Here is mine for example :

```
[user]
     name = Matthieu Pizenberg
     email = matthieu.pizenberg@gmail.com

[alias]
     ci = commit
     co = checkout
     st = status
     br = branch
[core]
     editor = vim
[color]
    ui = auto
    diff = auto
    log = auto
    branch = auto
[merge]
    tool = meld
```

You can also have a `config` file in the `.git` folder of a Git repository to have a specific configuration for that specific repository. You can modify the configuration file manually or with the command `git config`. If you pass the option `--global` it will be written in the file `~/.gitconfig` and without that option it will be written in the local config file in the `.git` folder of your current Git repository.

Example of command :
```
git config --global user.name "John Doe"
```

You can check your settings with the command : `git config --list`

2 Git Basics

2.1 Getting a Git Repository

There are 2 ways of getting a Git repository :

1. Initializing your own repository in a folder
2. Cloning the repository of someone else

2.1.1 Initializing your own repository

It is really easy, just go in the folder and use the command : `git init`
It creates a new hidden subdirectory `.git` which contains all metadata (remember part 1.2.4).

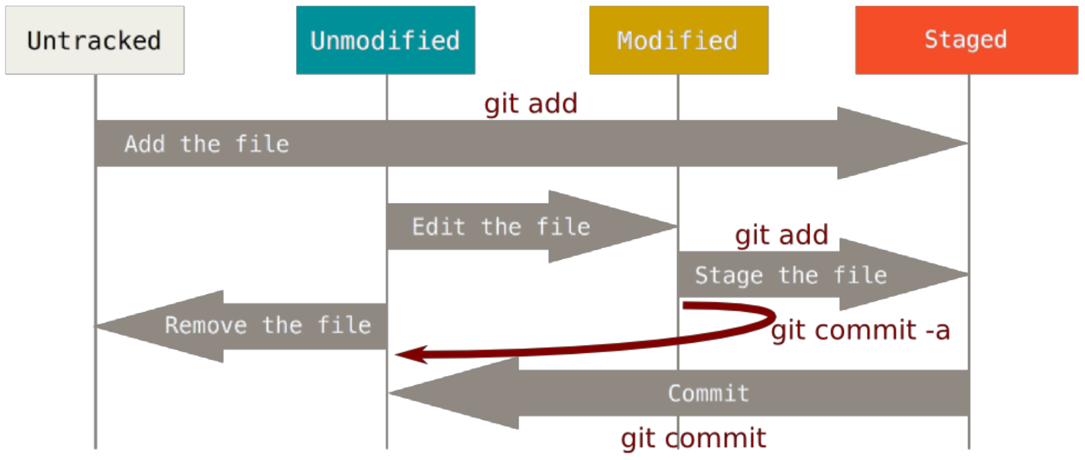
2.1.2 Cloning an existing repository

You can clone a repository with : `git clone [url]`.
The url might be another local git bare repository or the address of an online repository (server) :

```
git clone https://github.com/libgit2/libgit2
```

Usually you can use the HTTPS (like above) or the SSH protocol depending on the permissions you have on the server.

2.2 Recording Changes to the Repository



Life cycle of the status of your files

One of the most important command, the one you should do nearly before and after every other command is : **`git status`**.
It show the status of your repository and indicates all files that are untracked, modified or staged.

2.2.1 Tracking new files

Only tracked files are snapshotted in commits so if you want a modification of a file to be taken in account in your next commit you first need to add it to the tracked files with the command :

```
git add myfile
```

That command accepts some regexp like : `git add *.cpp` for example. If you want to track all files in a folder and its subfolders use : `git add myfolder`.

2.2.2 Staging modified files

Often your files are tracked (since you have already used the `git add` command on them for previous commits) and modified since last commit (means you have worked on them since last commit) but not staged. In that case changes will not be recorded. Only staged modifications will be recorded in next commit (see previous figure). You have two options then :

1. Use the `git add` command to track changes
2. Wait for the moment you will commit and at that moment pass the option `-a` (for example : `git commit -a`) **BE AWARE** that this option will stage **all tracked** and modified files but **no untracked file** before committing.

2.2.3 Ignoring files

Very often there are files in a repository that you do not want to commit. Typically these are automatically generated files or executables or big ressources (videos, …) etc. The reasons why you should not track them are many but the main are :

- They change at each commit so they create conflicts.
- They depend on your local configuration or are temporary files.
- They increase substantially the size of the repository and are not adapted to be versioned.

The way to ignore those file is to write a `.gitignore` file at the root of the Git repository. Again, it supports regular expressions and as a good example is worth plenty of explanations :

```
# no .a files
*.a

# but do track lib.a, even though you're ignoring .a files above
!lib.a

# only ignore the root TODO file, not subdir/TODO
/TODO

# ignore all files in the build/ directory
build/

# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt
```

```
# ignore all .txt files in the doc/ directory
doc/**/*.txt
```

A good initiative is to always set up a `.gitignore` file with rules to ignore at least temporary and system files (`*~, *.swp, .directory, .Trash-*, Desktop.ini, Thumbs.db, .DS_Store, …`). If your are using IDE (such as Eclipse) you should also ignore lots of other files. I can not give an exhaustive list of all but I can give an excellent tool to generate your `.gitignore` files : https://www.gitignore.io/

2.2.4 Viewing your staged and unstaged changes

To have more details than with the `git status` command you can use the `git diff` command. It will detail the changes between last commits and unstaged changes. You can also use `git diff --staged` which is explicit.

2.2.5 Committing your changes

All staged changes will be committed with the command : `git commit`. It will open an editor so you can fill the commit message. In order to avoid opening an editor at a commit you can pass the message with the -m option. For example :

```
git commit -m "What I have made with those changes"
```

2.2.6 Removing and moving files

Generally Git is smart enough to guess that you have moved, renamed or removed files. But it is simpler to tell Git explicitly. Instead of using rm, use `git rm` and instead of using mv, use `git mv`. For more information :
http://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository#Removing-Files

2.2.7 Bilan

Your cycles will very probably be something like that :

```
# modifying some files and testing it works
git add files # staging the files for next commit
git commit -m "What I have done"
```

2.3 Viewing the commit history

There is one simple magic powerful command : `git log`
You can pass to it too many options to describe it in that document. For detailed explanation go see that section of the online book. In a nutshell the more useful options are `--pretty` and `--graph`. For example :

```
$ git log --pretty=format:"%h %s" --graph
* 2d3acf9 ignore errors from SIGCHLD on trap
*  5e3ee11 Merge branch 'master' of git://github.com/dustin/grit
|\
| * 420eac9 Added a method for getting the current branch.
```

```
* | 30e367c timeout code and tests
* | 5a09431 add timeout protection to grit
* | e1193f8 support for heads with slashes in them
|/
* d6016bc require time for xmlschema
*  11d191e Merge branch 'defunkt' into local
```

2.4 Undoing Things

Again I won't get into the details here. See Undoing Things for detailed explanations.
Just know that you can do pretty much everything with those commands :

```
git status
git commit --amend
git checkout --
git reset
```

2.5 Working with remotes

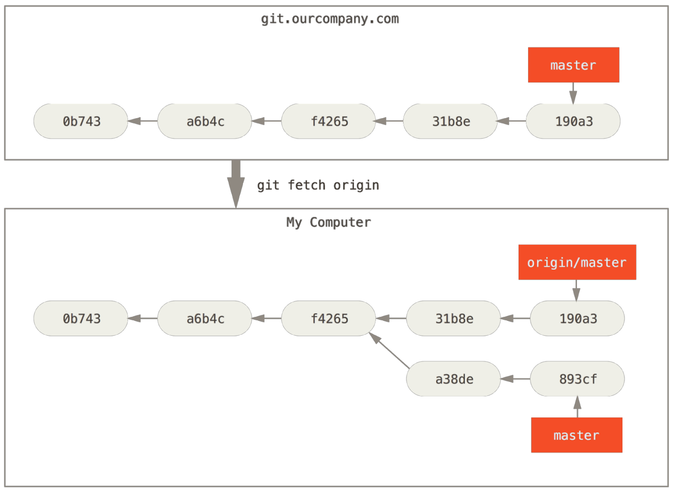
A remote repository is a version of your project that is hosted somewhere on the network (often on the internet like on Github).It allows you to collaborate with others by pulling and pushing data to that remote repository.
Showing your remotes : `git remote -v`
Adding a remote repository : `git remote add [shortname] [url]`
Inspecting a remote : `git remote show [remote-name]`

2.5.1 Fetching and pulling from your remotes

In order to retrieve others work on a remote you need to "fetch" data from the remote :

```
git fetch [remote-name]
```

That command will retrieve the last state of the remote repository but not overwrite your own local modification. The fetch command will produce something like that :



There is another command that we like to use to retrieve data from remotes but that is in fact **unsafe** ! It is the `pull` command : git pull [remote-name].
It is **unsafe** because it will play two commands in one : fetch and merge. and sometimes it is not what you wanted. Here is an example of what it could do :

Before pulling :

```
                A---B---C master on origin(remote)
               /
          D---E---F---G master (local)
          ^
          origin/master in your repository
```

After pulling

```
                A---B---C origin/master
               /         \
          D---E---F---G---H master (local)
```

2.5.2 Pushing to your remotes

This is easy. Providing that what you want to push directly comes from the last commit of the remote repository (this is called a **fast forward** or ff) pushing will work.

```
git push [remote-name] [branch-name]
```

Always fetch before pushing because it might show you that your local copy of the remote repository is not up to date. Here are two cases, in the first pushing will work, in the second it won't :

Work :

```
          A---B---C---D master (local)
              ^
              origin/master (remote)
```

Won't work :

```
                E origin/master (remote)
               /
          A---B---C---D master (local)
```

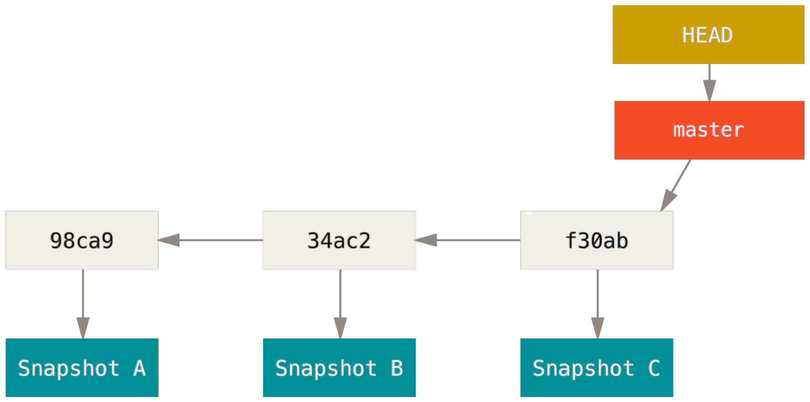
2.5.3 Renaming and removing remotes

Two simple commands :
git remote rename [old-name] [new-name]
git remote rm [remote-name]

3 Git Branching

3.1 Branches in a Nutshell

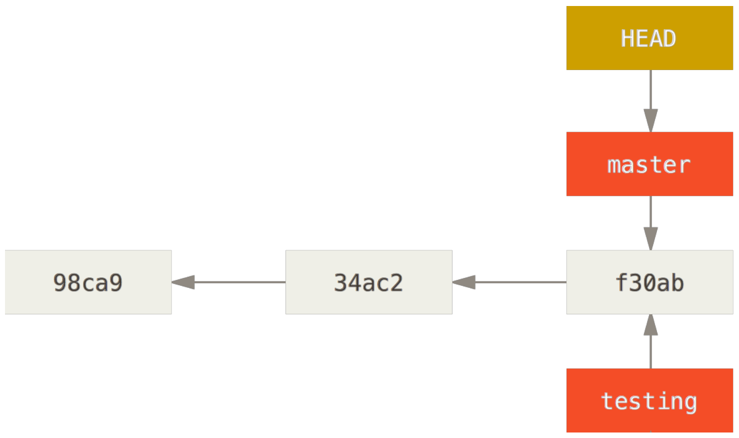
I won't go into the details of commits and branches implementation. If you want details, you can start here. In a simple way, you can remember that **commits and branches are pointers** identifiable uniquely.
Commits point to snapshots of your files and branches point to commits. Thanks to that simple structure it is easy to create delete and change branches without having to calculate huge differences each time.



In practically any repository there is a main **branch called master** but it is not more important than others. It is generated while creating the Git repository with the `git init` command, and people don't bother to change its name.
Git keeps also a special pointer called **HEAD**. It is a pointer to the commit you are currently on.

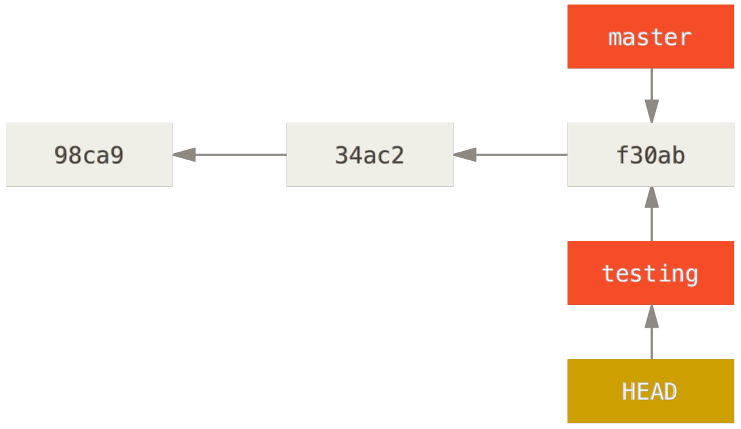
3.1.1 Creating a new branch

Creating a new branch simply create a new pointer but does not change your HEAD you are still at the same commit. For example :
git branch testing



3.1.2 Switching branches

To switch to an existing branch run the `git checkout` command. Here, if for example we change from master to testing (with : `git checkout testing`) the result will be :



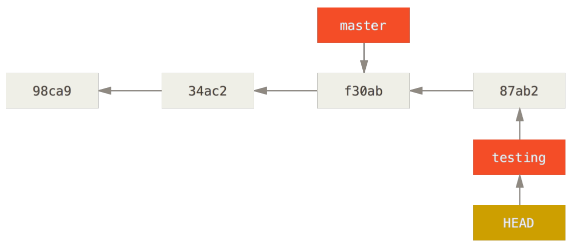
Be careful : switching branches will change your working directory if they don't point to the same commit.

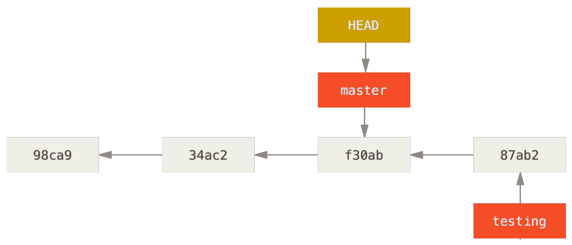
3.1.3 A little example

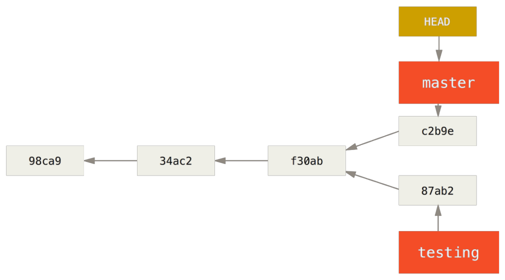
If I play the following commands from the last state :

```
vim test.txt                  # new file
git add test.txt
git commit -m "Created a file"
git checkout master
vim test.txt                  # also new since we came back to master
git add test.txt
git commit -m "Also created a file"
```

The successive states of the history will look like :







3.2 Basic Branching and Merging

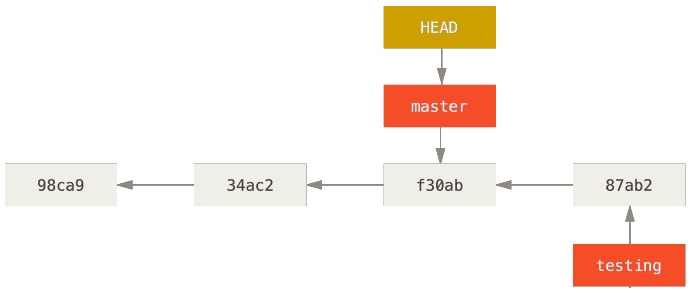
For many reasons you should use more than one branch on a project. Those branches might be for example : development, production and hotfix for a website. In such a configuration, you generally develop the code in the develop branch and then merge it in the production branch (the stable one).
You already know how to create new branches, merging two different branches is also very simple. Let's say I want to retrieve in master the work I did on testing :

```
git checkout master                # places the HEAD on master
git merge testing                  # merges the two branches in master
```

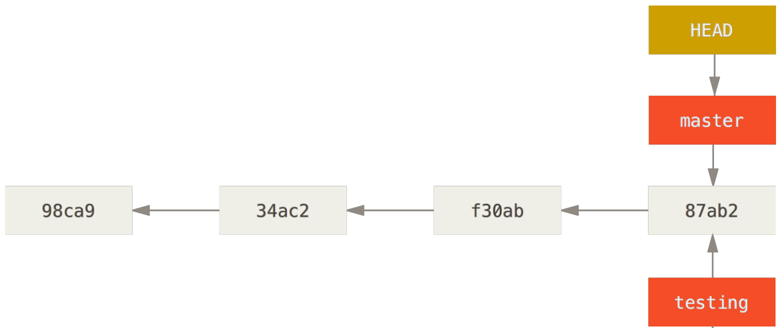
A piece of cake isn't it ^^ ?
The result of that command however will depend on the configuration of the merge command and on the relative positions of the two branches. I explain it in the two next subparts.

3.2.1 Merging fast-forward (--ff)

Let's imagine we are in that configuration :

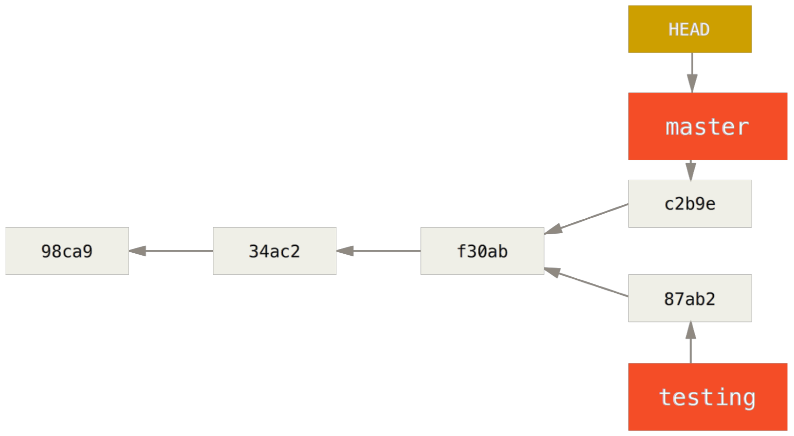


Running the command : `git merge testing`, will produce what is called a **fast-forward merge**. It is called so because the master pointer only has to ascend the graph. The result history will be :

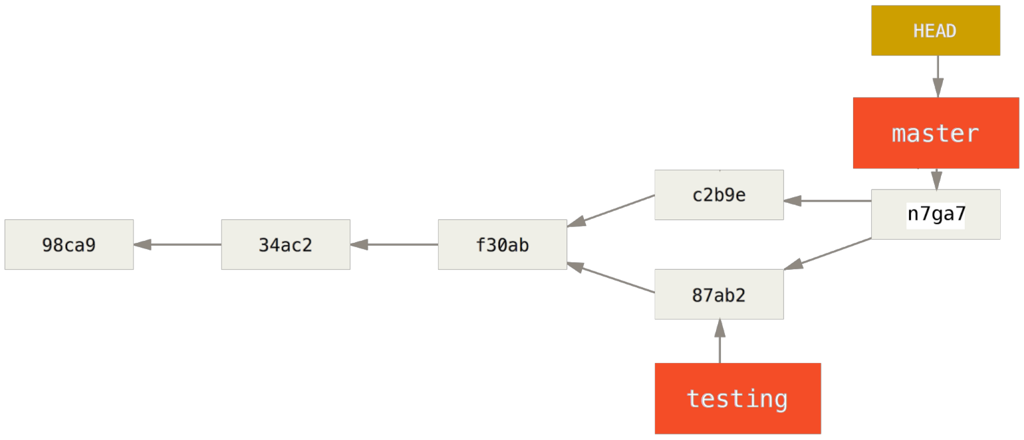


3.2.2 Merging without fast-forward (--no-ff)

This time let's imagine that we are in a configuration where master and testing branches have diverged :



Now the command : `git merge testing` will produce the following result :



3.2.3 Merge conflicts

Sometimes automatic merging will fail. This can happen when the same files have been modified in boths branches. The error message would look something like that :

```
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Resolving merge conflicts is not always easy and I advise to use a graphical tool such as meld. You can find more info about how to resolve merge conflicts here.

4 Bilan

Yeah ! For now I think it will be enough. With what you learned here you are able to understand how to work following our workflow.

Sure you will still have problems but google is your friend and you can still contact me physically or by facebook or by email at : matthieu.pizenberg@gmail.com