http://git-scm.com/book/en/v2

What is "version control", and why should you care? 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. For the examples in this book you will use software source code as the files being version controlled, though in reality you can do this with nearly any type of file on a computer.

Distributed version system (git, mercurial..)

clients don’t just check out the latest snapshot of the files: they fully mirror the repository.

Server Computer

Version Database

Version 3

Version 2

Version 1

Computer A Computer B

File File

Version Database Version Database

Version 3 Version 3

Version 2 Version 2

Version 1 Version 1

Git thinks of its data more like a set of snapshots of a miniature filesystem. Every time you commit, or save the state of your project in Git, it basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot. To be efficient, if files have not changed, Git doesn’t store the file again, just a link to the previous identical file it has already stored. Git thinks about its data more like a **stream of snapshots**.

Checkings over time ->

Version1 Version2 Version3 Version4 Version5

File A A1 A1 A2 A2

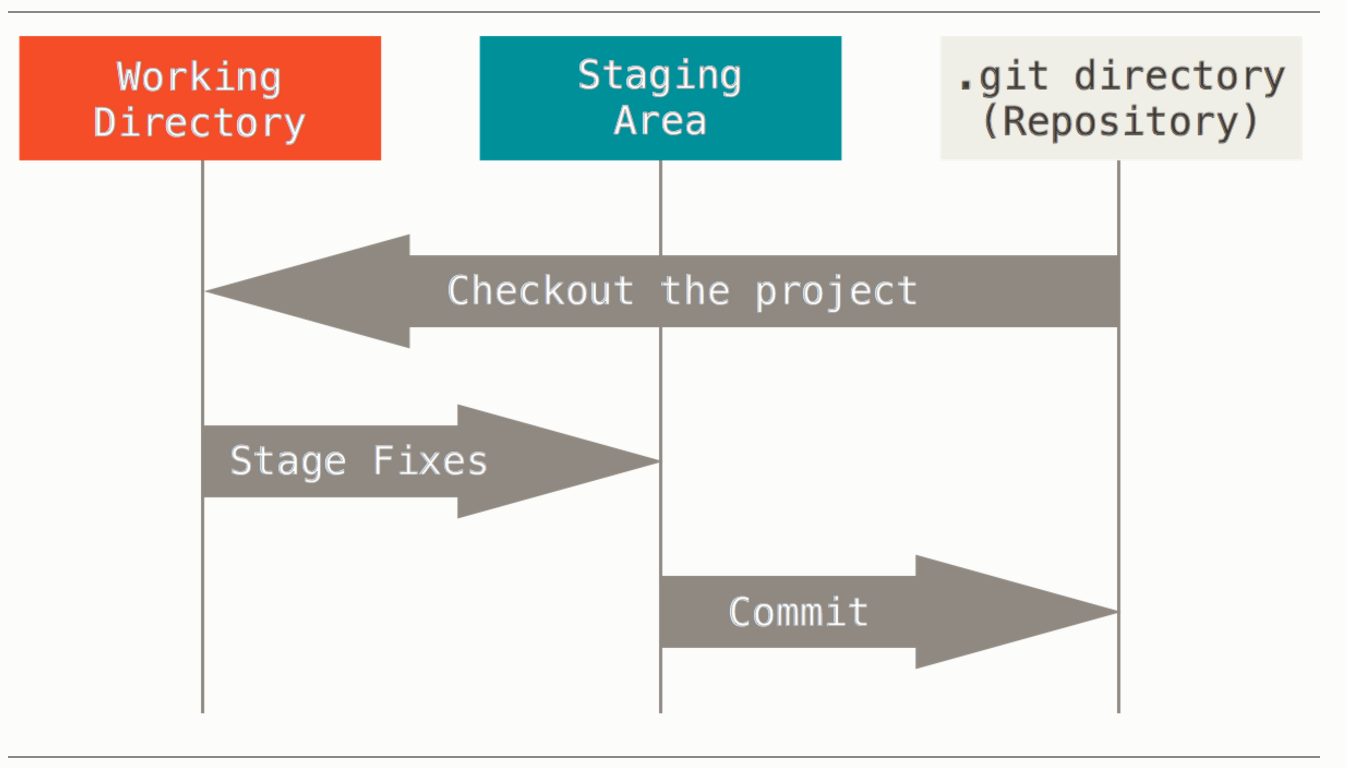
File B B B B1 B2

File C C1 C2 C2 C3

Most operations in Git only need local files and resources to operate.

Everything in Git is check-summed before it is stored and is then referred to by that checksum. This means it’s impossible to change the contents of any file or directory without Git knowing about it.

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. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.

Install git, if you’re on a Debian-based distribution like Ubuntu, try apt-get:

**$** sudo apt-get install git-all

Git comes with a tool called git config that lets you get and set configuration variables that control all aspects of how Git looks and operates. These variables can be stored in three different places:

1. /etc/gitconfig file: Contains values for every user on the system and all their repositories. If you pass the option --system to git config, it reads and writes from this file specifically.
2. ~/.gitconfig or ~/.config/git/config file: Specific to your user. You can make Git read and write to this file specifically by passing the --global option.
3. config file in the Git directory (that is, .git/config) of whatever repository you’re currently using: Specific to that single repository.

Each level overrides values in the previous level, so values in .git/config trump those in/etc/gitconfig.

### [**Your Identity**](http://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup#Your-Identity)

The first thing you should do when you install Git is to set your user name and email address. This is important because every Git commit uses this information, and it’s immutably baked into the commits you start creating:

**$** git config --global user.name "John Doe"  
**$** git config --global user.email [johndoe@example.com](mailto:johndoe@example.com)

### [**Your Editor**](http://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup#Your-Editor)

**$** git config --global core.editor emacs

### [**Checking Your Settings**](http://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup#Checking-Your-Settings)

If you want to check your settings, you can use the git config --list command to list all the settings Git can find at that point:

**$** git config --list  
user.name=John Doe  
user.email=johndoe@example.com  
color.status=auto

You can also check what Git thinks a specific key’s value is by typing git config <key>:

**$** git config user.name  
John Doe

## **Getting Help**

For example, you can get the manpage help for the config command by running

**$** git help config

## **Getting a Git Repository**

You can get a Git project using two main approaches. The first takes an existing project or directory and imports it into Git. The second clones an existing Git repository from another server.

### [**Initializing a Repository in an Existing Directory**](http://git-scm.com/book/en/v2/Git-Basics-Getting-a-Git-Repository#Initializing-a-Repository-in-an-Existing-Directory)

If you’re starting to track an existing project in Git, you need to go to the project’s directory and type:

**$** git init

This creates a new subdirectory named .git that contains all of your necessary repository files – a Git repository skeleton. At this point, nothing in your project is tracked yet. (See [Git Internals](http://git-scm.com/book/en/v2/1-git-internals/_git_internals) for more information about exactly what files are contained in the .git directory you just created.)

If you want to start version-controlling existing files (as opposed to an empty directory), you should probably begin tracking those files and do an initial commit. You can accomplish that with a few git add commands that specify the files you want to track, followed by a git commit:

**$** git add \*.c  
**$** git add LICENSE  
**$** git commit -m 'initial project version'

### [**Cloning an Existing Repository**](http://git-scm.com/book/en/v2/Git-Basics-Getting-a-Git-Repository#Cloning-an-Existing-Repository)

If you want to get a copy of an existing Git repository – for example, a project you’d like to contribute to – the command you need is git clone. If you’re familiar with other VCS systems such as Subversion, you’ll notice that the command is "clone" and not "checkout". This is an important distinction – instead of getting just a working copy, Git receives a full copy of nearly all data that the server has. Every version of every file for the history of the project is pulled down by default when you run git clone. In fact, if your server disk gets corrupted, you can often use nearly any of the clones on any client to set the server back to the state it was in when it was cloned (you may lose some server-side hooks and such, but all the versioned data would be there – see [Getting Git on a Server](http://git-scm.com/book/en/v2/1-git-server/_git_on_the_server) for more details).

You clone a repository with git clone [url]. For example, if you want to clone the Git linkable library called libgit2, you can do so like this:

**$** git clone https://github.com/libgit2/libgit2

That creates a directory named “libgit2”, initializes a .git directory inside it, pulls down all the data for that repository, and checks out a working copy of the latest version. If you go into the new libgit2directory, you’ll see the project files in there, ready to be worked on or used. If you want to clone the repository into a directory named something other than “libgit2”, you can specify that as the next command-line option:

**$** git clone https://github.com/libgit2/libgit2 mylibgit

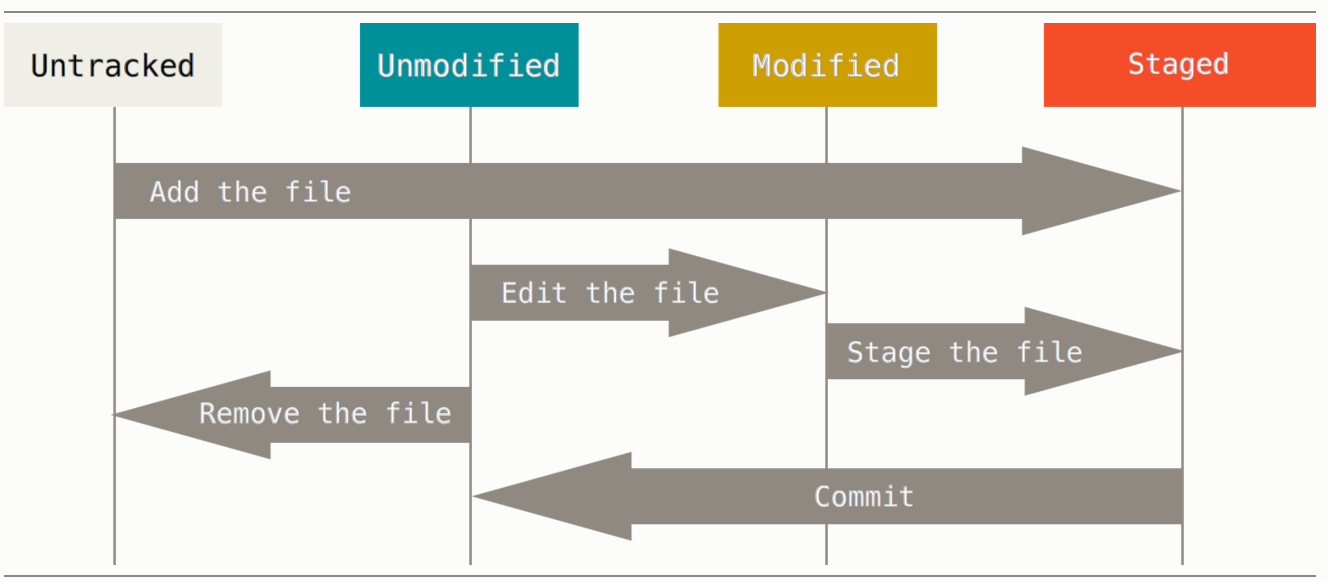
That command does the same thing as the previous one, but the target directory is called mylibgit.

Git has a number of different transfer protocols you can use. The previous example uses the https://protocol, but you may also see git:// or user@server:path/to/repo.git, which uses the SSH transfer protocol. [Getting Git on a Server](http://git-scm.com/book/en/v2/1-git-server/_git_on_the_server) will introduce all of the available options the server can set up to access your Git repository and the pros and cons of each.

## **Recording Changes to the Repository**

Remember that each file in your working directory can be in one of two states: tracked or untracked. Tracked files are files that were in the last snapshot; they can be unmodified, modified, or staged. Untracked files are everything else – any files in your working directory that were not in your last snapshot and are not in your staging area. When you first clone a repository, all of your files will be tracked and unmodified because you just checked them out and haven’t edited anything.

As you edit files, Git sees them as modified, because you’ve changed them since your last commit. You stage these modified files and then commit all your staged changes, and the cycle repeats.



### [**Checking the Status of Your Files**](http://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository#Checking-the-Status-of-Your-Files)

The main tool you use to determine which files are in which state is the git status command. If you run this command directly after a clone, you should see something like this:

**$** git status  
On branch master  
nothing to commit, working directory clean

This means you have a clean working directory – in other words, there are no tracked and modified files. Git also doesn’t see any untracked files, or they would be listed here. Finally, the command tells you which branch you’re on and informs you that it has not diverged from the same branch on the server.

Let’s say you add a new file to your project, a simple README file. If the file didn’t exist before, and you run git status, you see your untracked file like so:

**$** echo 'My Project' &gt; README  
**$** git status  
On branch master  
Untracked files:  
 (use "git add &lt;file&gt;..." to include in what will be committed)  
  
 README  
  
nothing added to commit but untracked files present (use "git add" to track)

### [**Tracking New Files**](http://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository#Tracking-New-Files)

In order to begin tracking a new file, you use the command git add. To begin tracking the README file, you can run this:

**$** git add README

If you run your status command again, you can see that your README file is now tracked and staged to be committed:

**$** git status  
On branch master  
Changes to be committed:  
 (use "git reset HEAD &lt;file&gt;..." to unstage)  
  
 new file: README

You can tell that it’s staged because it’s under the “Changes to be committed” heading. If you commit at this point, the version of the file at the time you ran git add is what will be in the historical snapshot. You may recall that when you ran git init earlier, you then ran git add (files) – that was to begin tracking files in your directory. The git add command takes a path name for either a file or a directory; if it’s a directory, the command adds all the files in that directory recursively.

### [**Staging Modified Files**](http://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository#Staging-Modified-Files)

Let’s change a file that was already tracked. If you change a previously tracked file called “CONTRIBUTING.md” and then run your git status command again, you get something that looks like this:

**$** git status  
On branch master  
Changes to be committed:  
 (use "git reset HEAD &lt;file&gt;..." to unstage)  
  
 new file: README  
  
Changes not staged for commit:  
 (use "git add &lt;file&gt;..." to update what will be committed)  
 (use "git checkout -- &lt;file&gt;..." to discard changes in working directory)  
  
 modified: CONTRIBUTING.md

The “CONTRIBUTING.md” file appears under a section named “Changes not staged for commit” – which means that a file that is tracked has been modified in the working directory but not yet staged. To stage it, you run the git add command. git add is a multipurpose command – you use it to begin tracking new files, to stage files, and to do other things like marking merge-conflicted files as resolved. It may be helpful to think of it more as “add this content to the next commit” rather than “add this file to the project”.Let’s run git add now to stage the “CONTRIBUTING.md” file, and then run git status again:

**$** git add CONTRIBUTING.md  
**$** git status  
On branch master  
Changes to be committed:  
 (use "git reset HEAD &lt;file&gt;..." to unstage)  
  
 new file: README  
 modified: CONTRIBUTING.md

Both files are staged and will go into your next commit. At this point, suppose you remember one little change that you want to make in CONTRIBUTING.md before you commit it. You open it again and make that change, and you’re ready to commit. However, let’s run git status one more time:

**$** vim CONTRIBUTING.md  
**$** git status  
On branch master  
Changes to be committed:  
 (use "git reset HEAD &lt;file&gt;..." to unstage)  
  
 new file: README  
 modified: CONTRIBUTING.md  
  
Changes not staged for commit:  
 (use "git add &lt;file&gt;..." to update what will be committed)  
 (use "git checkout -- &lt;file&gt;..." to discard changes in working directory)  
  
 modified: CONTRIBUTING.md

What the heck? Now CONTRIBUTING.md is listed as both staged *and* unstaged. How is that possible? It turns out that Git stages a file exactly as it is when you run the git add command. If you commit now, the version of CONTRIBUTING.md as it was when you last ran the git add command is how it will go into the commit, not the version of the file as it looks in your working directory when you run git commit. If you modify a file after you run git add, you have to run git add again to stage the latest version of the file:

**$** git add CONTRIBUTING.md  
**$** git status  
On branch master  
Changes to be committed:  
 (use "git reset HEAD &lt;file&gt;..." to unstage)  
  
 new file: README  
 modified: CONTRIBUTING.md

### [**Ignoring Files**](http://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository#Ignoring-Files)

Often, you’ll have a class of files that you don’t want Git to automatically add or even show you as being untracked. These are generally automatically generated files such as log files or files produced by your build system. In such cases, you can create a file listing patterns to match them named .gitignore.Here is an example .gitignore file:

**$** cat .gitignore  
\*.[oa]  
\*~

The first line tells Git to ignore any files ending in “.o” or “.a” – object and archive files that may be the product of building your code. The second line tells Git to ignore all files that end with a tilde (~), which is used by many text editors such as Emacs to mark temporary files. You may also include a log, tmp, or pid directory; automatically generated documentation; and so on.

### [**Viewing Your Staged and Unstaged Changes**](http://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository#Viewing-Your-Staged-and-Unstaged-Changes)

If the git status command is too vague for you – you want to know exactly what you changed, not just which files were changed – you can use the git diff command. We’ll cover git diff in more detail later, but you’ll probably use it most often to answer these two questions: What have you changed but not yet staged? And what have you staged that you are about to commit? Although git statusanswers those questions very generally by listing the file names, git diff shows you the exact lines added and removed – the patch, as it were.

Let’s say you edit and stage the README file again and then edit the CONTRIBUTING.md file without staging it. If you run your git status command, you once again see something like this:

**$** git status  
On branch master  
Changes to be committed:  
 (use "git reset HEAD &lt;file&gt;..." to unstage)  
  
 modified: README  
  
Changes not staged for commit:  
 (use "git add &lt;file&gt;..." to update what will be committed)  
 (use "git checkout -- &lt;file&gt;..." to discard changes in working directory)  
  
 modified: CONTRIBUTING.md

To see what you’ve changed but not yet staged, type git diff with no other arguments:

**$** git diff  
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md  
index 8ebb991..643e24f 100644  
--- a/CONTRIBUTING.md  
+++ b/CONTRIBUTING.md  
@@ -65,7 +65,8 @@ branch directly, things can get messy.  
 Please include a nice description of your changes when you submit your PR;  
 if we have to read the whole diff to figure out why you're contributing  
 in the first place, you're less likely to get feedback and have your change  
-merged in.  
+merged in. Also, split your changes into comprehensive chunks if your patch is  
+longer than a dozen lines.  
  
 If you are starting to work on a particular area, feel free to submit a PR  
 that highlights your work in progress (and note in the PR title that it's

That command compares what is in your working directory with what is in your staging area. The result tells you the changes you’ve made that you haven’t yet staged.

If you want to see what you’ve staged that will go into your next commit, you can use git diff --staged. This command compares your staged changes to your last commit:

**$** git diff --staged  
diff --git a/README b/README  
new file mode 100644  
index 0000000..03902a1  
--- /dev/null  
+++ b/README  
@@ -0,0 +1 @@  
+My Project

It’s important to note that git diff by itself doesn’t show all changes made since your last commit – only changes that are still unstaged. This can be confusing, because if you’ve staged all of your changes,git diff will give you no output.

For another example, if you stage the CONTRIBUTING.md file and then edit it, you can use git diffto see the changes in the file that are staged and the changes that are unstaged. If our environment looks like this:

**$** git add CONTRIBUTING.md  
**$** echo '# test line' &gt;&gt; CONTRIBUTING.md  
**$** git status  
On branch master  
Changes to be committed:  
 (use "git reset HEAD &lt;file&gt;..." to unstage)  
  
 modified: CONTRIBUTING.md  
  
Changes not staged for commit:  
 (use "git add &lt;file&gt;..." to update what will be committed)  
 (use "git checkout -- &lt;file&gt;..." to discard changes in working directory)  
  
 modified: CONTRIBUTING.md

Now you can use git diff to see what is still unstaged:

**$** git diff  
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md  
index 643e24f..87f08c8 100644  
--- a/CONTRIBUTING.md  
+++ b/CONTRIBUTING.md  
@@ -119,3 +119,4 @@ at the  
 ## Starter Projects  
  
 See our [projects list](https://github.com/libgit2/libgit2/blob/development/PROJECTS.md).  
+# test line

and git diff --cached to see what you’ve staged so far (--staged and --cached are synonyms):

**$** git diff --cached  
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md  
index 8ebb991..643e24f 100644  
--- a/CONTRIBUTING.md  
+++ b/CONTRIBUTING.md  
@@ -65,7 +65,8 @@ branch directly, things can get messy.  
 Please include a nice description of your changes when you submit your PR;  
 if we have to read the whole diff to figure out why you're contributing  
 in the first place, you're less likely to get feedback and have your change  
-merged in.  
+merged in. Also, split your changes into comprehensive chunks if your patch is  
+longer than a dozen lines.  
  
 If you are starting to work on a particular area, feel free to submit a PR  
 that highlights your work in progress (and note in the PR title that it's

### [**Committing Your Changes**](http://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository#Committing-Your-Changes)

Now that your staging area is set up the way you want it, you can commit your changes. Remember that anything that is still unstaged – any files you have created or modified that you haven’t run git add on since you edited them – won’t go into this commit. They will stay as modified files on your disk. In this case, let’s say that the last time you ran git status, you saw that everything was staged, so you’re ready to commit your changes. The simplest way to commit is to type git commit:

**$** git commit

Doing so launches your editor of choice. (This is set by your shell’s $EDITOR environment variable – usually vim or emacs, although you can configure it with whatever you want using the git config --global core.editor command

The editor displays the following text (this example is a Vim screen):

# Please enter the commit message for your changes. Lines starting  
# with '#' will be ignored, and an empty message aborts the commit.  
# On branch master  
# Changes to be committed:  
# new file: README  
# modified: CONTRIBUTING.md  
#  
~  
~  
~  
".git/COMMIT\_EDITMSG" 9L, 283C

You can see that the default commit message contains the latest output of the git status command commented out and one empty line on top. You can remove these comments and type your commit message, or you can leave them there to help you remember what you’re committing. (For an even more explicit reminder of what you’ve modified, you can pass the -v option to git commit. Doing so also puts the diff of your change in the editor so you can see exactly what changes you’re committing.) When you exit the editor, Git creates your commit with that commit message (with the comments and diff stripped out).

Alternatively, you can type your commit message inline with the commit command by specifying it after a -m flag, like this:

**$** git commit -m "Story 182: Fix benchmarks for speed"  
[master 463dc4f] Story 182: Fix benchmarks for speed  
 2 files changed, 2 insertions(+)  
 create mode 100644 README

Now you’ve created your first commit! You can see that the commit has given you some output about itself: which branch you committed to (master), what SHA-1 checksum the commit has (463dc4f), how many files were changed, and statistics about lines added and removed in the commit.

Remember that the commit records the snapshot you set up in your staging area. Anything you didn’t stage is still sitting there modified; you can do another commit to add it to your history. Every time you perform a commit, you’re recording a snapshot of your project that you can revert to or compare to later.

### [**Removing Files**](http://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository#Removing-Files)

To remove a file from Git, you have to remove it from your tracked files (more accurately, remove it from your staging area) and then commit. The git rm command does that, and also removes the file from your working directory so you don’t see it as an untracked file the next time around.

If you simply remove the file from your working directory, it shows up under the “Changed but not updated” (that is, *unstaged*) area of your git status output:

**$** rm PROJECTS.md  
**$** git status  
On branch master  
Your branch is up-to-date with 'origin/master'.  
Changes not staged for commit:  
 (use "git add/rm &lt;file&gt;..." to update what will be committed)  
 (use "git checkout -- &lt;file&gt;..." to discard changes in working directory)  
  
 deleted: PROJECTS.md  
  
no changes added to commit (use "git add" and/or "git commit -a")

Then, if you run git rm, it stages the file’s removal:

**$** git rm PROJECTS.md  
rm 'PROJECTS.md'  
**$** git status  
On branch master  
Changes to be committed:  
 (use "git reset HEAD &lt;file&gt;..." to unstage)  
  
 deleted: PROJECTS.md

The next time you commit, the file will be gone and no longer tracked. If you modified the file and added it to the index (staged( already, you must force the removal with the -f option. This is a safety feature to prevent accidental removal of data that hasn’t yet been recorded in a snapshot and that can’t be recovered from Git.

Another useful thing you may want to do is to keep the file in your working tree but remove it from your staging area. In other words, you may want to keep the file on your hard drive but not have Git track it anymore. This is particularly useful if you forgot to add something to your .gitignore file and accidentally staged it, like a large log file or a bunch of .a compiled files. To do this, use the --cachedoption:

**$** git rm --cached README

You can pass files, directories, and file-glob patterns to the git rm command. That means you can do things such as:

**$** git rm log/**\\***.log

Note the backslash (\) in front of the \*. This is necessary because Git does its own filename expansion in addition to your shell’s filename expansion. This command removes all files that have the .logextension in the log/ directory. Or, you can do something like this:

**$** git rm **\\***~

This command removes all files that end with ~.

### [**Moving Files**](http://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository#Moving-Files)

Unlike many other VCS systems, Git doesn’t explicitly track file movement. If you rename a file in Git, no metadata is stored in Git that tells it you renamed the file. However, Git is pretty smart about figuring that out after the fact – we’ll deal with detecting file movement a bit later.

Thus it’s a bit confusing that Git has a mv command. If you want to rename a file in Git, you can run something like:

**$** git mv file\_from file\_to

and it works fine. In fact, if you run something like this and look at the status, you’ll see that Git considers it a renamed file:

**$** git mv README.md README  
**$** git status  
On branch master  
Changes to be committed:  
 (use "git reset HEAD &lt;file&gt;..." to unstage)  
  
 renamed: README.md -&gt; README

## **Viewing the Commit History**

After you have created several commits, or if you have cloned a repository with an existing commit history, you’ll probably want to look back to see what has happened. The most basic and powerful tool to do this is the git log command.

These examples use a very simple project called “simplegit”. To get the project, run

**$** git clone https://github.com/schacon/simplegit-progit

When you run git log in this project, you should get output that looks something like this:

**$** git log  
commit ca82a6dff817ec66f44342007202690a93763949  
Author: Scott Chacon &lt;schacon@gee-mail.com&gt;  
Date: Mon Mar 17 21:52:11 2008 -0700  
  
 changed the version number  
  
commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7  
Author: Scott Chacon &lt;schacon@gee-mail.com&gt;  
Date: Sat Mar 15 16:40:33 2008 -0700  
  
 removed unnecessary test  
  
commit a11bef06a3f659402fe7563abf99ad00de2209e6  
Author: Scott Chacon &lt;schacon@gee-mail.com&gt;  
Date: Sat Mar 15 10:31:28 2008 -0700  
  
 first commit

By default, with no arguments, git log lists the commits made in that repository in reverse chronological order – that is, the most recent commits show up first. As you can see, this command lists each commit with its SHA-1 checksum, the author’s name and email, the date written, and the commit message.

A huge number and variety of options to the git log command are available to show you exactly what you’re looking for. Here, we’ll show you some of the most popular.

One of the more helpful options is -p, which shows the difference introduced in each commit. You can also use -2, which limits the output to only the last two entries:

**$** git log -p -2  
commit ca82a6dff817ec66f44342007202690a93763949  
Author: Scott Chacon &lt;schacon@gee-mail.com&gt;  
Date: Mon Mar 17 21:52:11 2008 -0700  
  
 changed the version number  
  
diff --git a/Rakefile b/Rakefile  
index a874b73..8f94139 100644  
--- a/Rakefile  
+++ b/Rakefile  
@@ -5,7 +5,7 @@ require 'rake/gempackagetask'  
 spec = Gem::Specification.new do |s|  
 s.platform = Gem::Platform::RUBY  
 s.name = "simplegit"  
- s.version = "0.1.0"  
+ s.version = "0.1.1"  
 s.author = "Scott Chacon"  
 s.email = "schacon@gee-mail.com"  
 s.summary = "A simple gem for using Git in Ruby code."  
  
commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7  
Author: Scott Chacon &lt;schacon@gee-mail.com&gt;  
Date: Sat Mar 15 16:40:33 2008 -0700  
  
 removed unnecessary test  
  
diff --git a/lib/simplegit.rb b/lib/simplegit.rb  
index a0a60ae..47c6340 100644  
--- a/lib/simplegit.rb  
+++ b/lib/simplegit.rb  
@@ -18,8 +18,3 @@ class SimpleGit  
 end  
  
 end  
-  
-if $0 == \_\_FILE\_\_  
- git = SimpleGit.new  
- puts git.show  
-end  
\ No newline at end of file

This option displays the same information but with a diff directly following each entry. This is very helpful for code review or to quickly browse what happened during a series of commits that a collaborator has added. You can also use a series of summarizing options with git log. For example, if you want to see some abbreviated stats for each commit, you can use the --stat option:

**$** git log --stat  
commit ca82a6dff817ec66f44342007202690a93763949  
Author: Scott Chacon &lt;schacon@gee-mail.com&gt;  
Date: Mon Mar 17 21:52:11 2008 -0700  
  
 changed the version number  
  
 Rakefile | 2 +-  
 1 file changed, 1 insertion(+), 1 deletion(-)  
  
commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7  
Author: Scott Chacon &lt;schacon@gee-mail.com&gt;  
Date: Sat Mar 15 16:40:33 2008 -0700  
  
 removed unnecessary test  
  
 lib/simplegit.rb | 5 -----  
 1 file changed, 5 deletions(-)  
  
commit a11bef06a3f659402fe7563abf99ad00de2209e6  
Author: Scott Chacon &lt;schacon@gee-mail.com&gt;  
Date: Sat Mar 15 10:31:28 2008 -0700  
  
 first commit  
  
 README | 6 ++++++  
 Rakefile | 23 +++++++++++++++++++++++  
 lib/simplegit.rb | 25 +++++++++++++++++++++++++  
 3 files changed, 54 insertions(+)

As you can see, the --stat option prints below each commit entry a list of modified files, how many files were changed, and how many lines in those files were added and removed. It also puts a summary of the information at the end.

Another really useful option is --pretty. This option changes the log output to formats other than the default. A few prebuilt options are available for you to use. The oneline option prints each commit on a single line, which is useful if you’re looking at a lot of commits. In addition, the short, full, andfuller options show the output in roughly the same format but with less or more information, respectively:

**$** git log --pretty=oneline  
ca82a6dff817ec66f44342007202690a93763949 changed the version number  
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test  
a11bef06a3f659402fe7563abf99ad00de2209e6 first commit

The most interesting option is format, which allows you to specify your own log output format. This is especially useful when you’re generating output for machine parsing – because you specify the format explicitly, you know it won’t change with updates to Git:

**$** git log --pretty=format:"%h - %an, %ar : %s"  
ca82a6d - Scott Chacon, 6 years ago : changed the version number  
085bb3b - Scott Chacon, 6 years ago : removed unnecessary test  
a11bef0 - Scott Chacon, 6 years ago : first commit

[Table 2-1](http://git-scm.com/book/en/v2/ch00/pretty_format) lists some of the more useful options that format takes.

|  |  |
| --- | --- |
| **Option** | **Description of Output** |
| %H | Commit hash |
| %h | Abbreviated commit hash |
| %T | Tree hash |
| %t | Abbreviated tree hash |
| %P | Parent hashes |
| %p | Abbreviated parent hashes |
| %an | Author name |
| %ae | Author email |
| %ad | Author date (format respects the --date=option) |
| %ar | Author date, relative |
| %cn | Committer name |
| %ce | Committer email |
| %cd | Committer date |
| %cr | Committer date, relative |
| %s | Subject |

## **Undoing Things**

At any stage, you may want to undo something. Here, we’ll review a few basic tools for undoing changes that you’ve made. Be careful, because you can’t always undo some of these undos. This is one of the few areas in Git where you may lose some work if you do it wrong.

One of the common undos takes place when you commit too early and possibly forget to add some files, or you mess up your commit message. If you want to try that commit again, you can run commit with the--amend option:

**$** git commit --amend

This command takes your staging area and uses it for the commit. If you’ve made no changes since your last commit (for instance, you run this command immediately after your previous commit), then your snapshot will look exactly the same, and all you’ll change is your commit message.

The same commit-message editor fires up, but it already contains the message of your previous commit. You can edit the message the same as always, but it overwrites your previous commit.

As an example, if you commit and then realize you forgot to stage the changes in a file you wanted to add to this commit, you can do something like this:

**$** git commit -m 'initial commit'  
**$** git add forgotten\_file  
**$** git commit --amend

You end up with a single commit – the second commit replaces the results of the first.

### 

### 

### 

### [**Unstaging a Staged File**](http://git-scm.com/book/en/v2/Git-Basics-Undoing-Things#Unstaging-a-Staged-File)

The next two sections demonstrate how to wrangle your staging area and working directory changes. The nice part is that the command you use to determine the state of those two areas also reminds you how to undo changes to them. For example, let’s say you’ve changed two files and want to commit them as two separate changes, but you accidentally type git add \* and stage them both. How can you unstage one of the two? The git status command reminds you:

**$** git add \*  
**$** git status  
On branch master  
Changes to be committed:  
 (use "git reset HEAD &lt;file&gt;..." to unstage)  
  
 renamed: README.md -&gt; README  
 modified: CONTRIBUTING.md

Right below the “Changes to be committed” text, it says use git reset HEAD <file>... to unstage. So, let’s use that advice to unstage the CONTRIBUTING.md file:

**$** git reset HEAD CONTRIBUTING.md  
Unstaged changes after reset:  
M CONTRIBUTING.md  
**$** git status  
On branch master  
Changes to be committed:  
 (use "git reset HEAD &lt;file&gt;..." to unstage)  
  
 renamed: README.md -&gt; README  
  
Changes not staged for commit:  
 (use "git add &lt;file&gt;..." to update what will be committed)  
 (use "git checkout -- &lt;file&gt;..." to discard changes in working directory)  
  
 modified: CONTRIBUTING.md

The command is a bit strange, but it works. The CONTRIBUTING.md file is modified but once again unstaged.

###### NOTE

While git reset *can* be a dangerous command if you call it with --hard, in this instance the file in your working directory is not touched. Calling git reset without an option is not dangerous - it only touches your staging area.

.[**Unmodifying a Modified File**](http://git-scm.com/book/en/v2/Git-Basics-Undoing-Things#Unmodifying-a-Modified-File)

What if you realize that you don’t want to keep your changes to the CONTRIBUTING.md file? How can you easily unmodify it – revert it back to what it looked like when you last committed (or initially cloned, or however you got it into your working directory)? Luckily, git status tells you how to do that, too. In the last example output, the unstaged area looks like this:

Changes not staged for commit:  
 (use "git add &lt;file&gt;..." to update what will be committed)  
 (use "git checkout -- &lt;file&gt;..." to discard changes in working directory)  
  
 modified: CONTRIBUTING.md

It tells you pretty explicitly how to discard the changes you’ve made. Let’s do what it says:

**$** git checkout -- CONTRIBUTING.md  
**$** git status  
On branch master  
Changes to be committed:  
 (use "git reset HEAD &lt;file&gt;..." to unstage)  
  
 renamed: README.md -&gt; README

You can see that the changes have been reverted.

Remember, anything that is *committed* in Git can almost always be recovered. Even commits that were on branches that were deleted or commits that were overwritten with an --amend commit can be recovered (see [Data Recovery](http://git-scm.com/book/en/v2/1-git-internals/_data_recovery) for data recovery). However, anything you lose that was never committed is likely never to be seen again.

## **Working with Remotes**

To be able to collaborate on any Git project, you need to know how to manage your remote repositories. Remote repositories are versions of your project that are hosted on the Internet or network somewhere. You can have several of them, each of which generally is either read-only or read/write for you. Collaborating with others involves managing these remote repositories and pushing and pulling data to and from them when you need to share work. Managing remote repositories includes knowing how to add remote repositories, remove remotes that are no longer valid, manage various remote branches and define them as being tracked or not, and more. In this section, we’ll cover some of these remote-management skills.

### [**Showing Your Remotes**](http://git-scm.com/book/en/v2/Git-Basics-Working-with-Remotes#Showing-Your-Remotes)

To see which remote servers you have configured, you can run the git remote command. It lists the shortnames of each remote handle you’ve specified. If you’ve cloned your repository, you should at least see origin – that is the default name Git gives to the server you cloned from:

**$** git clone https://github.com/schacon/ticgit  
Cloning into 'ticgit'...  
remote: Reusing existing pack: 1857, done.  
remote: Total 1857 (delta 0), reused 0 (delta 0)  
Receiving objects: 100% (1857/1857), 374.35 KiB | 268.00 KiB/s, done.  
Resolving deltas: 100% (772/772), done.  
Checking connectivity... done.  
**$** cd ticgit  
**$** git remote  
origin

You can also specify -v, which shows you the URLs that Git has stored for the shortname to be used when reading and writing to that remote:

**$** git remote -v  
origin https://github.com/schacon/ticgit (fetch)  
origin https://github.com/schacon/ticgit (push)

If you have more than one remote, the command lists them all. For example, a repository with multiple remotes for working with several collaborators might look something like this.

**$** cd grit  
**$** git remote -v  
bakkdoor https://github.com/bakkdoor/grit (fetch)  
bakkdoor https://github.com/bakkdoor/grit (push)  
cho45 https://github.com/cho45/grit (fetch)  
cho45 https://github.com/cho45/grit (push)  
defunkt https://github.com/defunkt/grit (fetch)  
defunkt https://github.com/defunkt/grit (push)  
koke git://github.com/koke/grit.git (fetch)  
koke git://github.com/koke/grit.git (push)  
origin git@github.com:mojombo/grit.git (fetch)  
origin git@github.com:mojombo/grit.git (push)

### [**Adding Remote Repositories**](http://git-scm.com/book/en/v2/Git-Basics-Working-with-Remotes#Adding-Remote-Repositories)

We’ve mentioned and given some demonstrations of adding remote repositories in previous sections, but here is how to do it explicitly. To add a new remote Git repository as a shortname you can reference easily, run git remote add [shortname] [url]:

**$** git remote  
origin  
**$** git remote add pb https://github.com/paulboone/ticgit  
**$** git remote -v  
origin https://github.com/schacon/ticgit (fetch)  
origin https://github.com/schacon/ticgit (push)  
pb https://github.com/paulboone/ticgit (fetch)  
pb https://github.com/paulboone/ticgit (push)

Now you can use the string pb on the command line in lieu of the whole URL. For example, if you want to fetch all the information that Paul has but that you don’t yet have in your repository, you can run git fetch pb:

**$** git fetch pb  
remote: Counting objects: 43, done.  
remote: Compressing objects: 100% (36/36), done.  
remote: Total 43 (delta 10), reused 31 (delta 5)  
Unpacking objects: 100% (43/43), done.  
From https://github.com/paulboone/ticgit  
 \* [new branch] master -&gt; pb/master  
 \* [new branch] ticgit -&gt; pb/ticgit

### [**Fetching and Pulling from Your Remotes**](http://git-scm.com/book/en/v2/Git-Basics-Working-with-Remotes#Fetching-and-Pulling-from-Your-Remotes)

As you just saw, to get data from your remote projects, you can run:

**$** git fetch [remote-name]

The command goes out to that remote project and pulls down all the data from that remote project that you don’t have yet. After you do this, you should have references to all the branches from that remote, which you can merge in or inspect at any time.

If you clone a repository, the command automatically adds that remote repository under the name “origin”. So, git fetch origin fetches any new work that has been pushed to that server since you cloned (or last fetched from) it. It’s important to note that the git fetch command pulls the data to your local repository – it doesn’t automatically merge it with any of your work or modify what you’re currently working on. You have to merge it manually into your work when you’re ready.

If you have a branch set up to track a remote branch (see the next section and [Git Branching](http://git-scm.com/book/en/v2/1-git-branching/_git_branching) for more information), you can use the git pull command to automatically fetch and then merge a remote branch into your current branch. This may be an easier or more comfortable workflow for you; and by default, the git clone command automatically sets up your local master branch to track the remote master branch (or whatever the default branch is called) on the server you cloned from. Running git pull generally fetches data from the server you originally cloned from and automatically tries to merge it into the code you’re currently working on.

### [**Pushing to Your Remotes**](http://git-scm.com/book/en/v2/Git-Basics-Working-with-Remotes#Pushing-to-Your-Remotes)

When you have your project at a point that you want to share, you have to push it upstream. The command for this is simple: git push [remote-name] [branch-name]. If you want to push your master branch to your origin server (again, cloning generally sets up both of those names for you automatically), then you can run this to push any commits you’ve done back up to the server:

**$** git push origin master

This command works only if you cloned from a server to which you have write access and if nobody has pushed in the meantime. If you and someone else clone at the same time and they push upstream and then you push upstream, your push will rightly be rejected. You’ll have to pull down their work first and incorporate it into yours before you’ll be allowed to push. See [Git Branching](http://git-scm.com/book/en/v2/1-git-branching/_git_branching) for more detailed information on how to push to remote servers.

### [**Inspecting a Remote**](http://git-scm.com/book/en/v2/Git-Basics-Working-with-Remotes#Inspecting-a-Remote)

If you want to see more information about a particular remote, you can use the git remote show [remote-name] command. If you run this command with a particular shortname, such as origin, you get something like this:

**$** git remote show origin  
\* remote origin  
 Fetch URL: https://github.com/schacon/ticgit  
 Push URL: https://github.com/schacon/ticgit  
 HEAD branch: master  
 Remote branches:  
 master tracked  
 dev-branch tracked  
 Local branch configured for 'git pull':  
 master merges with remote master  
 Local ref configured for 'git push':  
 master pushes to master (up to date)

It lists the URL for the remote repository as well as the tracking branch information. The command helpfully tells you that if you’re on the master branch and you run git pull, it will automatically merge in the master branch on the remote after it fetches all the remote references. It also lists all the remote references it has pulled down.

**Forking git workflow**

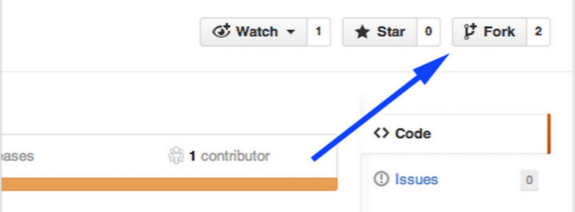
os Ubuntu 14.04 via Terminal

contributed repository account on github [http://github.com/contrib](http://github.com/contrib/repo)

my repository account on github <http://github.com/user>

repository name repo

**Fork repository you contribute to from github and clone forked**



$ git clone “<http://github.com/user/repo>

**Add upstream to update fork when neccessary**

$ git remote add upstream “[http://github.com/contrib](http://github.com/contrib/repo)/repo.git”

Show remote

$ git remote -v

origin [http://github.com/](http://github.com/contrib/repo)user/repo.git (fetch)

origin [http://github.com/](http://github.com/contrib/repo)user/repo.git (push)

upstream [http://github.com/](http://github.com/contrib/repo)contrib/repo.git (fetch)

upstream [http://github.com/contrib](http://github.com/contrib/repo)/repo.git (fetch)

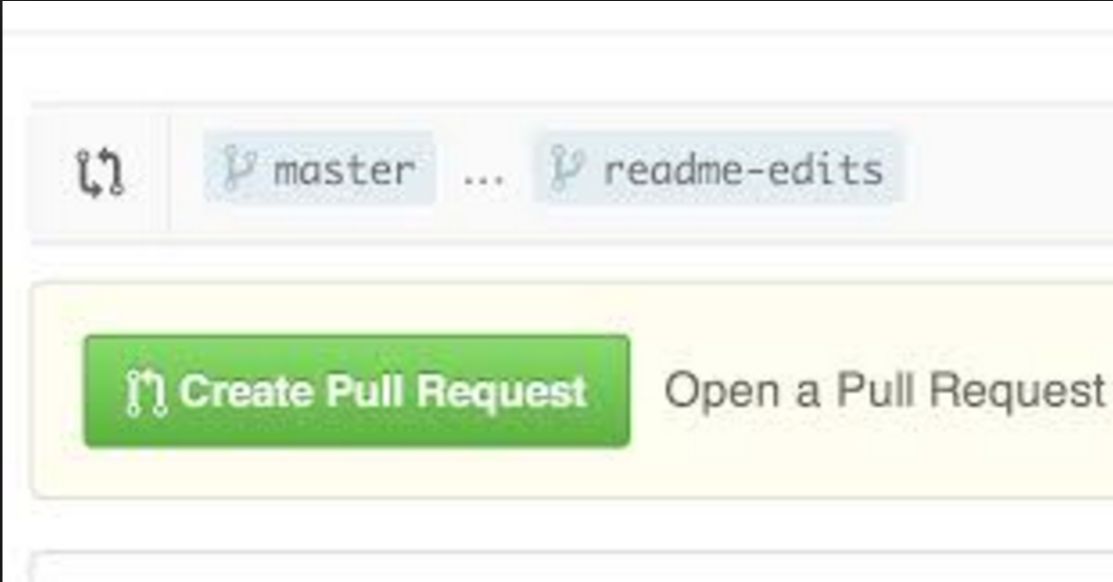
**Work on particular feature and add commits using add and commit**

$ git add file

$ git commit -m “Adding file”

**Push to your fork and let it be review by others on github by creating pull request**

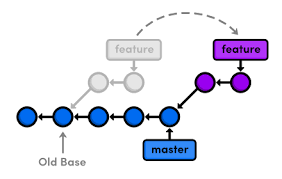
$ git push origin master



**Update repository to synchronize with upstream**

$ git fetch upstream

$ git rebase upstream/master (Put your commits on top of upstream master )



**Various scenarios of update**

There are three possibilities what could happen,

1. no new commits on upstream and origin

**Update repository to synchronize with upstream**

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

2. new commits on upstream, origin without changes, push changes from upstream to your origin

**Update repository to synchronize with upstream**

$ git status

Your branch is ahead of 'origin/master' by 1 commit.

(use "git push" to publish your local commits)

$ git push origin master

3. new commits on upstream, new commits on origin. For a simplification imagine, you commited new file1 to origin, and created pull request. In between somebody commited file2 to upstream. Your collegue commented on you Pull Request Changes and you correct them with another commited file3 to origin. Now you update your origin

**Update repository to synchronize with upstream**

$ git status

Your branch and 'origin/master' have diverged,

and have 3 and 1 different commit each, respectively.

(use "git pull" to merge the remote branch into yours)

(this means upstream history -> file1 (one commit)

origin history -> file2 -> file1 ->fil3 (three commits)

**Resolving diverged branch merge**

1. Using patches

Export your commits as series of patches

$ git format-patch -M @{upstream}

will produce 3 patch files -- one for each of your commited files file1, file2 and file3

0001\_file2.patch

0002\_file1.patch

0003\_file3.patch

Reset your origin to match upstream

git reset --hard @{upstream}

(this means origin history -> file1

Now you can apply patches to update your origin to use your last commits (file3) or use changes from others that you are dependent on (file2)

$ git am 0003\_file3.patch

$ git am 0001\_file2.patch

(this means origin history -> file1 -> file2 -> file3

You can update your files with adding patches and removing latest n commits like

$ git am 0001\_file2.patch

$ git reset --soft HEAD~1

After doing your work, when you are prepared to commit again and make your pull request do not forget to remove the commits that are not your own (from your collegues), in our case simply apply reset to last commit and push your changes to origin

$ git reset --soft HEAD~1 (to remove 0001\_file2.patch)

$ git push origin master

Create Pull Request

(this means origin history -> file1 -> file3 before merge)

2. Using cherry-picking

Create a separate branch but stay in master

$ git branch file

Reset your origin master to the upstream master

$ git reset --hard @{upstream}

Cherry pick the commits you need from file branch to your master branch, or reset them

$ git cherry-pick file~1 git cherry-pick file~3 git cherry-pick file~2 …

$ git reset --soft HEAD~2

(this means history -> file1 -> file3)

$ git push origin master

**Reset last commit on your repo, clearing index files and working tree, no unsaved changes restored**

$ git reset --hard HEAD~1

$git push origin master -f (force)