***Git Basics***

Scope:

Git can version-controlling scope is only below its parent directory (include all files and all subdirectory’s files (can have more and more subdirectories) by default. It shows like below:

dir

Dir1

Dir2

.git

F1

F2

Dir11

Dir121

Dir12

F121

F111

F22

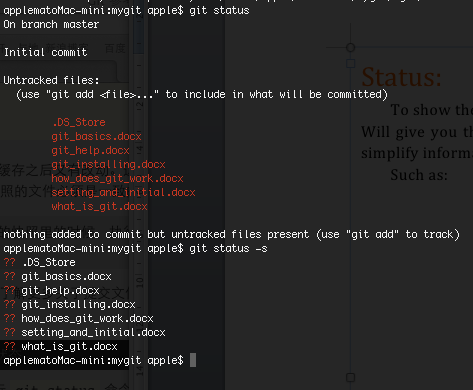
This shows that the git can only tracing the read-path (green file(s) or directory (directories)). It means all of content must below the “Dir1”.

Status:

To show the status between workspace and stage, using “git status”. “git status”

Will give you the full information about the status, and “git status -s” will get the simplify information.

Such as:



This will tell you under the “Untracked files” list is all files that you hadn’t been traced, and the ‘-s’ argument’s result shows in front of the file if have “??” it means that the file did not had been traced. And it tell you that to trace any files use “git add <file>” to include in what will be committed.

More about “git status” will be explained in “git add” or “git commit”, or other case (other command(s)).

Add

After you created a git-repo, if you want to trace the file(s), first of all you need to add the file(s) to the stage. You can do that like below:

//Add specify file

git add [path]file\_name

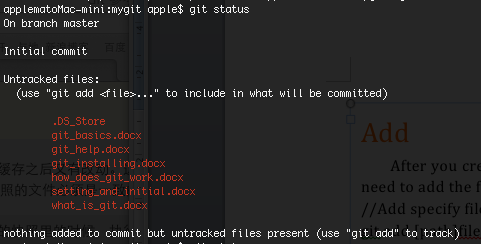
//Add all files included in the director

git add [path]dir

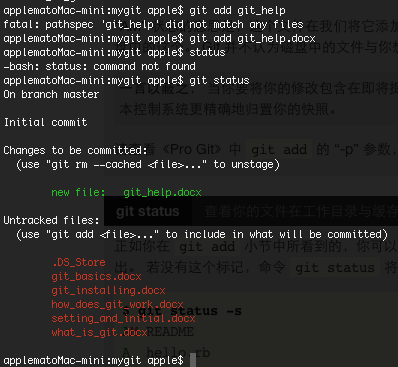
//Add all files in current director which file’s extends name is .c

git add \*.c

Before all of “add” you maybe need to check the status.



Now, It shows that no file had been traced, so add a file in the stage.

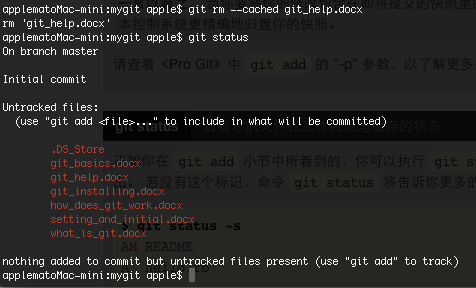


This shows that,

1 you must type the full name of the file.

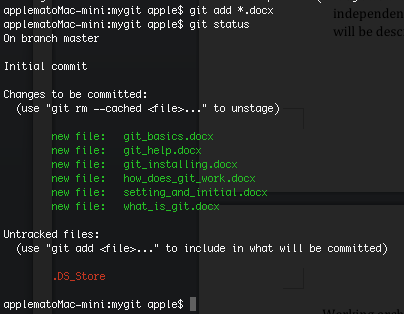
2 If any other error occurs, you would get the relation information with the command, other wise no extra information been showed.

3 This shows that the “git\_help.docx” file is new for tem-git-repo (stage), and already adds to stage (did not in actually git-repo), you can remove it from the stage. Why do not have a try?



Okay! The “git rm –cached file\_name” command do the work very well.

Now, use the wildcard character add all “.docx” files into stage.

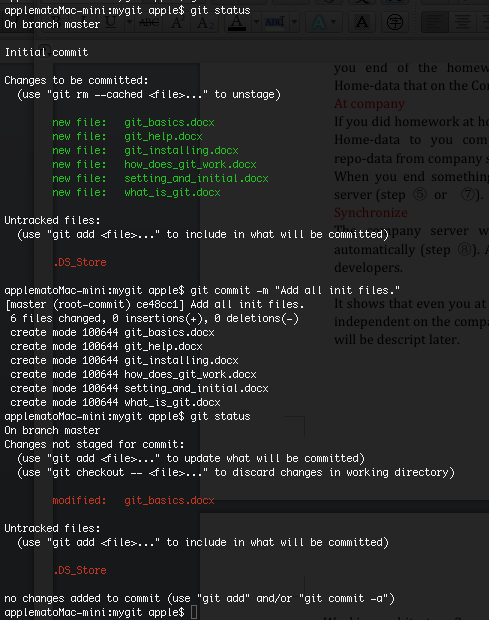


You can use the wildcard character remove all stage file also.

“Changes to be committed” tell you commit the changes, the “commit” had been explained below (see “Commit”).

Commit and Three statuses:

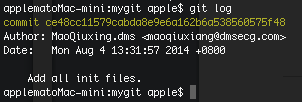
After you add all files you want to trace into the stage, if you want to actually save the stage content (all files) to the git repository, then use “git commit –m “Something commentate statements”, or “git commit” and then Git will open an editor to let you type the comments. This means that, you must use “git commit” command to save the stage to disk and must type the comment.



After “git commit” if no error occur, it will give you the message about the “commit”. Such as count of file, insertions rows, deletions rows, activity (here is create), and on which branch, and the unique ID, of course the comment.

After commit, use “status” can see the status between workspace and stage, because I write something in the git\_basics.docx file after add in to stage, so the status shows that git\_basics.docx had been modified, and any other files I did not edit, so nothing been showed.

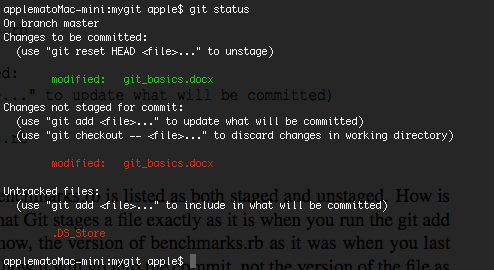
The unique ID been consist of 40 characters, it only show some characters before of the full characters. If you use the “git log” command you will get the full characters, it shows like:



You can see, the given characters by commit are the sub-characters of log’s.

The log command shows the things: full unique ID, who commit it, commit date and commit’s comment.

What is this? (What the heck?)



Two lines show that “ modified git\_basics.docx”. This means that, this first line is you added the file to stage but don’t commit it, and the second line is that the stage’s file had been update than the storage’s (HEAD’s) file, because the stage’s file been update when you use “add” command.

You can understand this as, “add” is writes workspace to stage and “commit” is writes stage to storage (actually object in the git-repo). This says that a file (or many files) can have thread status (workspace-stage-storage).

We can make a test like below to under the three statuses:

Create a simple file. And add it to the git repository, and then change its state.



Now, we compare the three version of test.txt file use “git diff” command.

Before change the file’s content:



First line compare between workspace and stage.

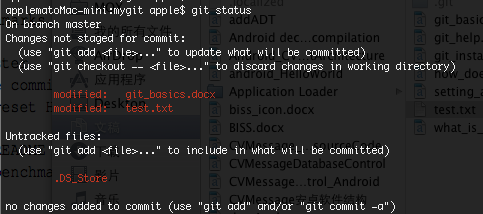
Second line is stage and HEAD.

Third is workspace and HEAD.

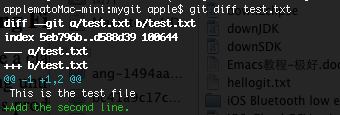
All of this compare have nothing information to show, this means that all of these compares result are the same.

Fourth line is open the test.txt use “vi” (a editor). And I add a new line in test.txt (Add the second line.).

Then do the commands order and get the result:



Changed the file (add a new line), but did not “add” it to stage.



Comparing workspace with stage. “a” file is stage’s file, and “b” file is workspace’s file, showing by

--- a/test.txt

+++ b/test.txt

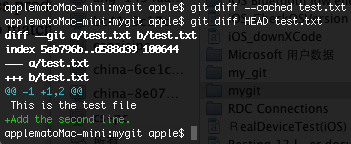
“-1” means the –file only has one line (start from 1th line).

“+1,2” means the +file (+b) has two lines (start from 1th line).

@@ -1 +1,2 @@

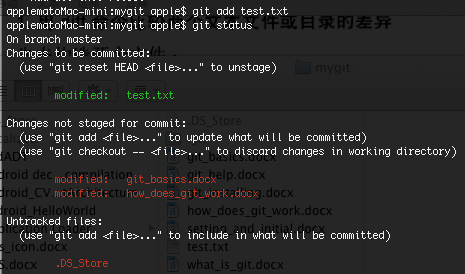
This is the test file //This is the public line.

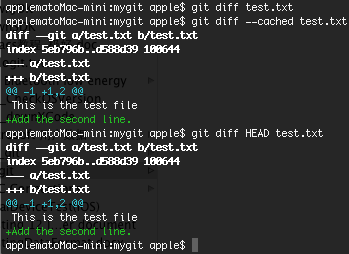
+Add the second line. //This is the line only in (+b) workspace file.



This is the same like workspace and stage.

Now, “add” test.txt file to stage, and re-comparing.



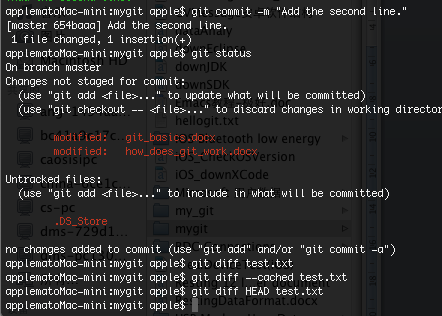


“git diff” given the same result (nothing log info). Means workspace is the same as stage (this is of cause, because we override stage use workspace (the “add” do that)).

“git diff --cached” shows storage file (a) only has one line and stage file (b) has two lines.

The similar in “git diff HEAD”

Now, commit it and re-comparing.

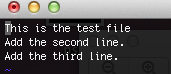


Now, all of the three statuses are the same again.

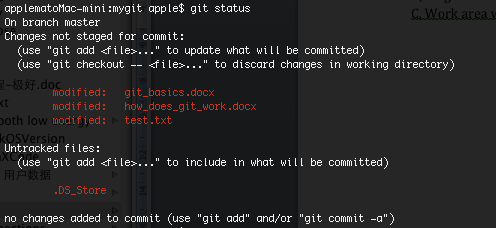
To make the three statuses is not the same to each other, keep the “HEAD” on and change the workspace, then “add” it to stage, change the workspace again.



Add a new line (Add the third line).

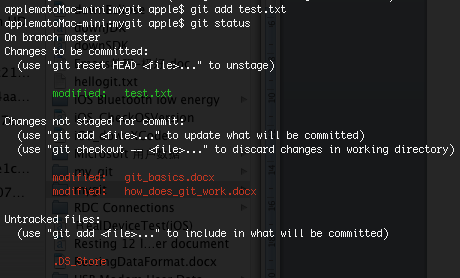


Check the status:



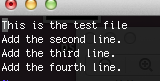
This shows that the file had been modified but not “add” to the stage(w!=s).

“add” and recheck status:

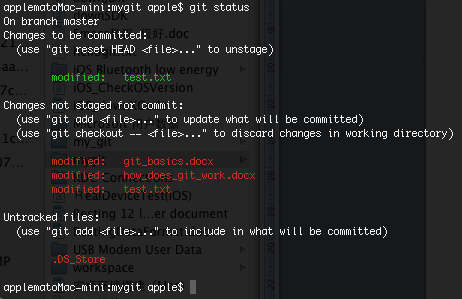


This shows that “add” to the stage but not commit(w=s, s!=H).

Add a new line again:



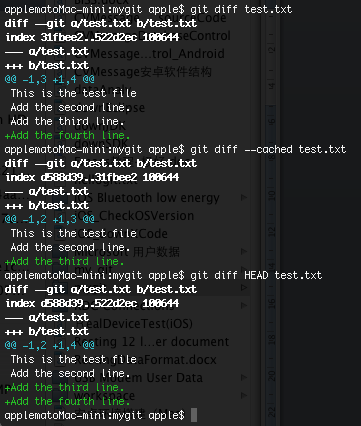
Check status:



The green line means is the stage had been changed but not “commit”.

The red line about “test.txt” shows that workspace had been changed but not “add”. Other files are the same that, if you did not add to the stage when you changed the file(s) in workspace, will be list here. And the same to “commit”.

Diff continue:



So, you can see the three status been shows here. The workspace has the addition line (Add the fourth line.) than stage. And stage has the addition line (Add the third line) than HEAD. At the end of diff, the workspace has two addition lines (Add the third line. Add the fourth line) than HEAD.

Ignoring files

If you don’t want to trace some files and don’t want to see them in the untrack list. You can use a file to list that what file or file style to ignore.

Ignore rules:

1. A file can be ignore only then this file didn’t been traced.
2. \*.a All file that end of .a will be ignore, include its sub-directors.
3. !file This file can’t be ignore, event if it is name like test.a and the \*.a had been list in ignore file.
4. /dir All file(s) which in the dir will be ignore (except use 3)), but didn’t include the subdirectory’s file(s). This means that /dir/dir/test.txt will didn’t been ignored.
5. Dir/ All file(s) or dir(s) will be ignored that which are in the Dir.
6. Dir/\*.a Ignored all file(s) which is end of .a and under the Dir/, but didn’t include its subdirectory’s content.
7. If you want to list ignored file(s), using “--ignored –s” append to the “git status” for the argument. Ignored file(s) couldn’t be “add” to stage, except using “-f fileName” for argument, such as “git add –f test.txt”.
8. Blank lines or lines starting with ‘#’ are ignored.

Ignore model:

* Share A

If anyone clone you git-repo this ignore role will be using. This setting is in the file that been named .gitignore. The .gitignore file’s scope is current director where the .gitignore file exist director and all its sub-directors. So it can be put in anywhere.

* Local

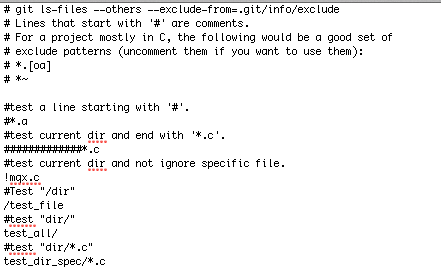
1. B Global, “git config –global core.excludesfile [your\_ignore\_file\_path\_ignorfile]”, the name of ignorefile is not limit.
2. C Specific version, setting the file which in the path: .git/info/exclude.

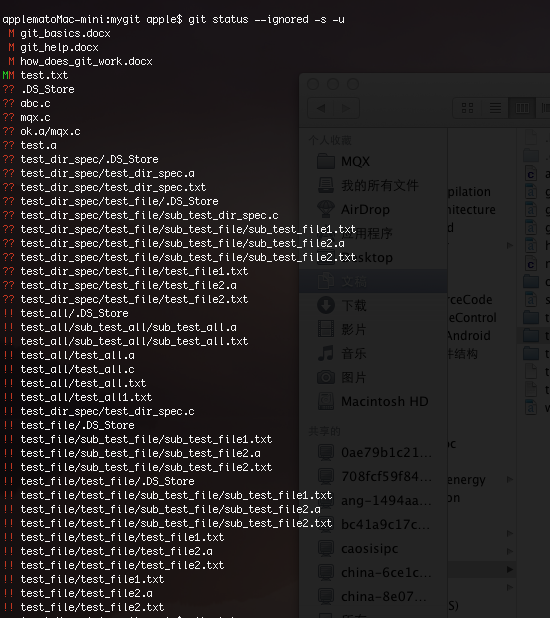
Note:

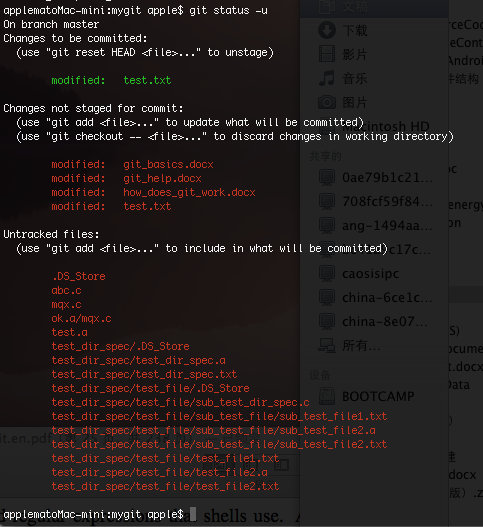
2) Mach all end of .a files or folders. But /dir/\*.a only mach /dir/XXX.a, on other world /dir/abc/XXX.a hadn’t been ignored,

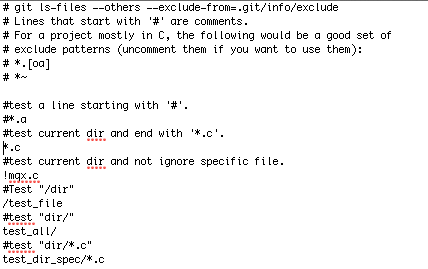
Testing:

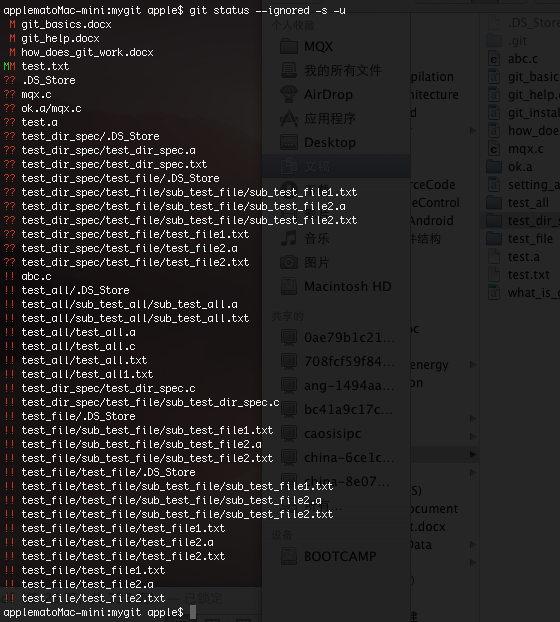
Local Specific version

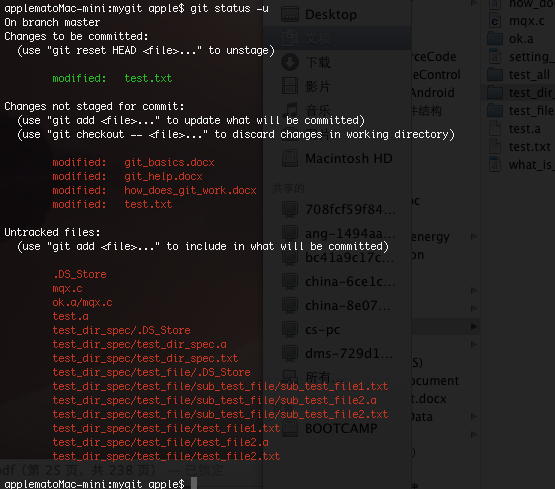








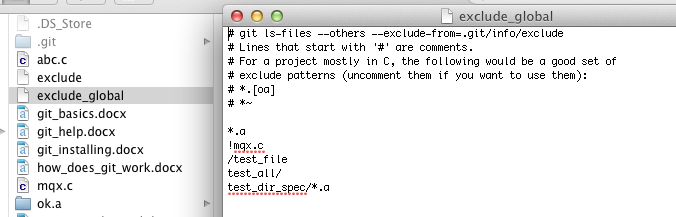




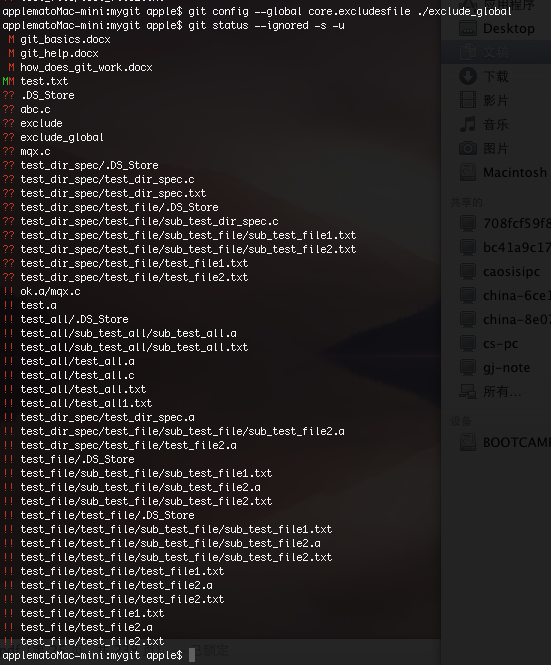
Local Global

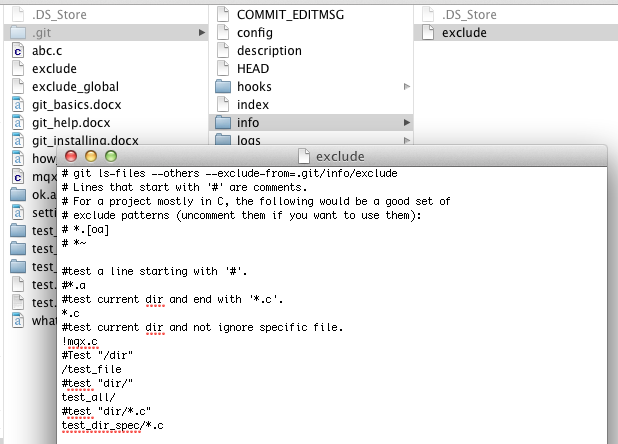
When you clear the .git/info/exclude file and not setting any other ignore roles, no one had been ignored.

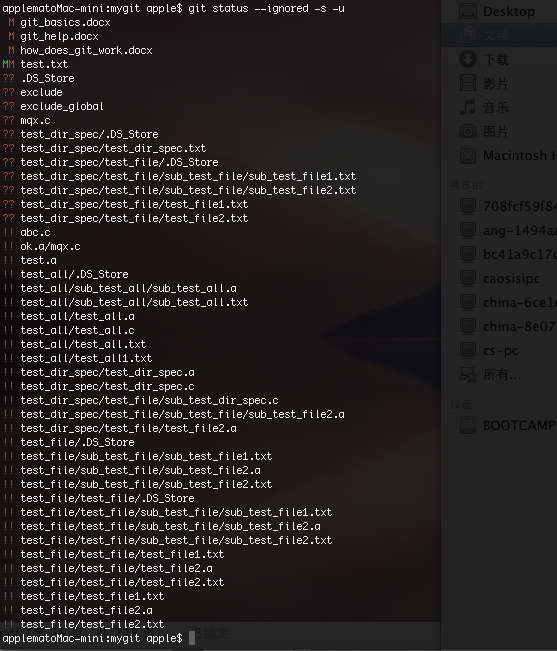
And then create the file like below:



Before I setting the global configuration I clear the ignore in .git/info/exclude file.



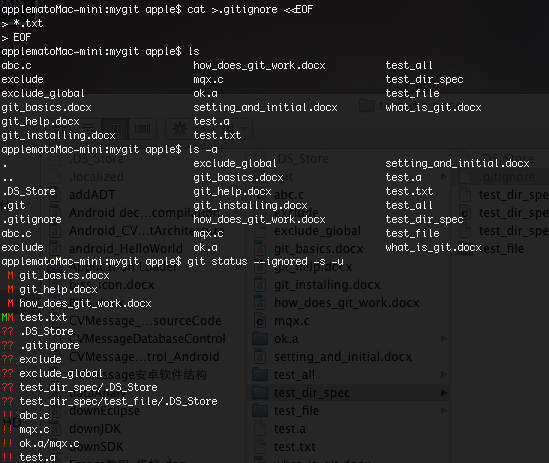




When I clear the exclude\_global file, the result had been change automatically.

And the ignored set is {exclude U exclude\_global}.

Share



Remain of ignored files had not been list (to save space).

These show that, actually ignored = {exclude U exclude\_global U .gitignore}.

Ignored is the result set, and everyone of the ignore\_file is the set too. In the other world they are not the override relations.

Skipping the staging area

When you want to commit the workspace to HEAD and didn’t want to use “git add”, you can use “git commit –a –m ‘your comments.’” to implement. This is says that “git commit –a –m ‘comments’” == “git add files” + “git commit –m ‘comments’”, if you use the ‘-a’, will commit all files what had been tracing and had been changed to the HEAD.

Removing files

* Remove from everywhere

When you do this, the file(s) can’t be recovered from Git.

To do this, you can do like below steps:

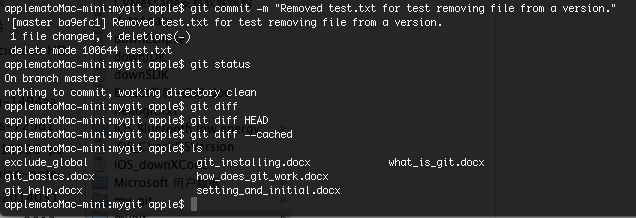
Step 1: delete from workspace (delete directly).

Step 2: delete from stage area (git rm [file\_name or files]).

Step 3:delete from HEAD (commit it).

This only delete the file(s) from current version, but it(them) can be recover from older version (previous commit).





Now, I removed the “test.txt” file from disk driver and any where of git (this commit version).

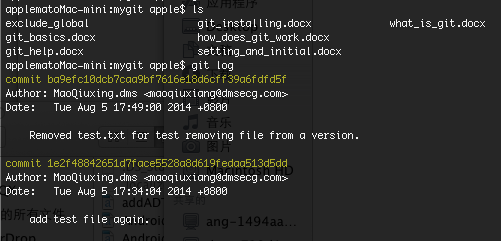
* Remove from Git

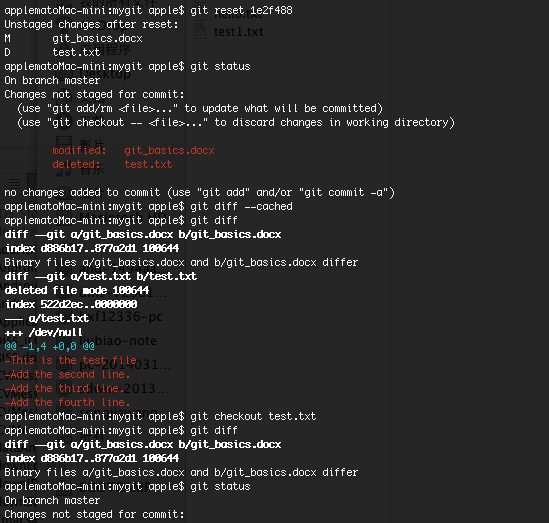
Step 1: remove from stage area (rm from cached).

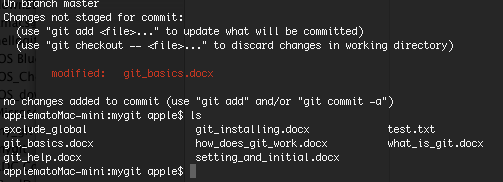
Step 2: remove from HEAD (commit).

Step 3: set the ignore file.

Before do this, we need to go to the older version which had the “test.txt” file.

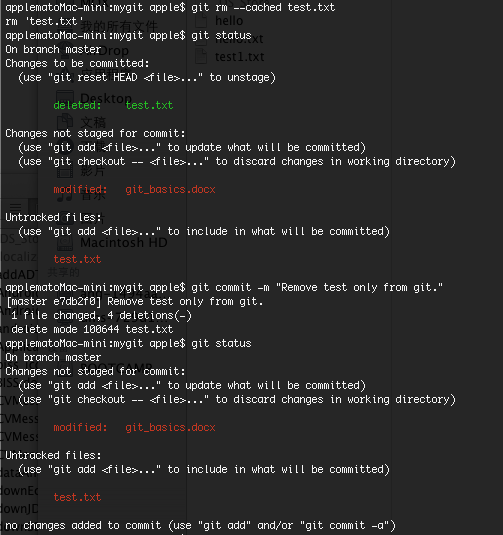






To here we found back the “test.txt” from the older version of git.

Now, delete it from git (delete only in specify version) and not trace it (set the ignore file).



This shows that the “test.txt” file had not been tracked.

Only need to set it in the ignore file if you want don’t show this information again (try it yourself).

* Remove from Git’s all versions

Git log

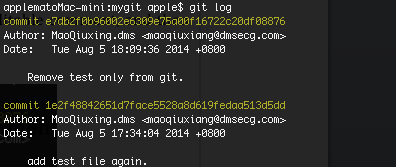
Undo

Modify the comment

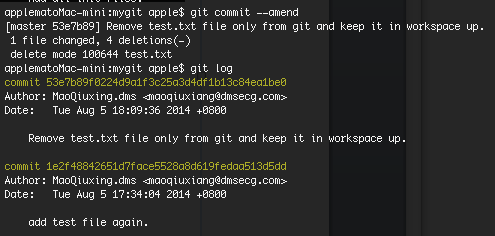
1, The last commit (this will make “local out of date” if relate to remo-repo, and to fix this problem is that keep local git-repo is newer than remote’s on, is a good idea).

Use “git commit --amend” command to open the editable comment, and then modify the comment.

Before modify:



After modify:



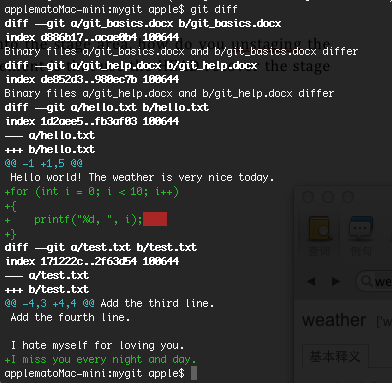
This shows that the commit ID will be changed. The modify comment can’t be redo.

If you forget to add some files and commit, you can add the files then use “commit --amend” modify the commit that is the previous.

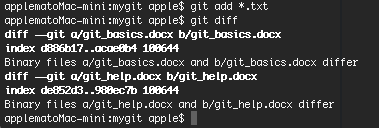
2, The history commit

Unstaging a Staged File

If you already add some files into the stage area, how do you unstaging the specific one or more? This implement is that use the HEAD recover the stage area.

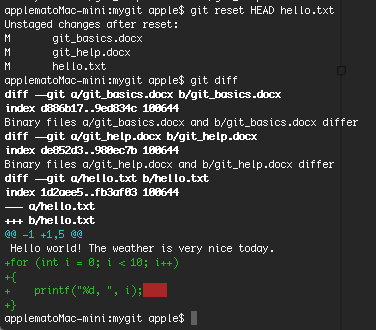


Here, I have two files (hello.txt and test.txt) already been modified and not been “add”. Now let’s “add” it to the stage area.



The problem is now how can I unstage the hello.txt file?

Maybe “git reset HEAD hello.txt” is the best answer now. Let’s have a try.

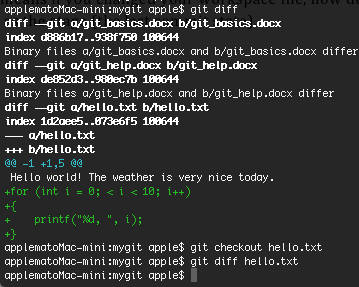


Everything of the hello.txt file had been returned to unstage state.

Of course, you can do that by the wildcard character(s), such as ‘\*’.

Unmodifying a Modified file

This means if you changed your workspace file, how do you can reset it to the state as the stage (the last commit state).



This shows that, use the “git checkout file\_you\_want” can reset the file which is in the workspace the same as stage area’s state. Can use wildcard characters too.

This is the dangerous command, any changes you made to that file are gone—you will lose all of the changed. Because this is only copy the stage to workspace.

Working with Remotes

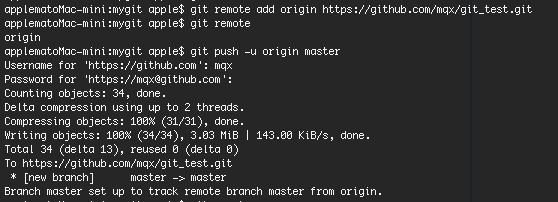
If you do not have a remote repository, you need to create it before do anything.

For create a remote repository, one of the way is show below:

Register in “github.com” if you don’t have an account.

And the guide will guidance you how to do to create a repository.

When you do all the work of the web, the commands will had been done show likes below:

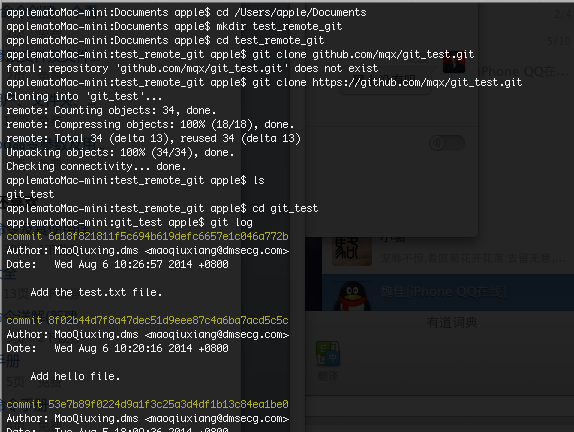


“git add remote …” means setting the remote url (I think).

“git remote” means list the current relate remote, if nothing be showed include the “origin”, means that current git does not relate to any remote repository.

“git push –u origin master” is that push your git’s master branch to the remote repository. Then this mybe request user name and password, this user name and password is that when you register in the web. After checked this, the pushing will be executed automatically.

If you have the remote repository, you only need clone the repository from the remote when the first time at create your local git repository. It maybe shows like below:

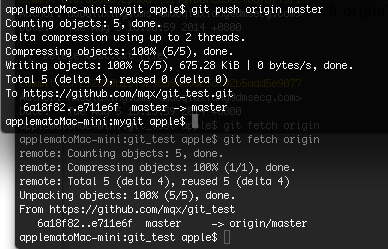


After you relate to the remote repository, when you want to push your commit to the remote repository, use “git push origin master” to accomplish.

Origin—the destination of remote.

Master—the branch of local git to push.

And you can get the new version from remote use “git fetch origin”. This will re-get from the remote.

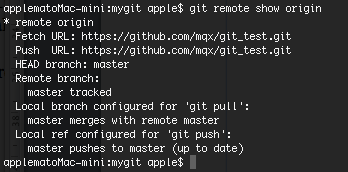


I use two terminate to test this. One is push and the other one is fetch.

Because when you local git-repo is the same to the remote git-repo the “fetch” does nothing. When you use “fetch”, you maybe need to merge origin/master with local master, if you use “git remote show origin” get the info “local out of date”, use need merge manually. To see the diff use “git log master ..origin/m-

aster”, and to merge use “git merge origin/master”.

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:



This command shows push and fetch url and merge branch info.

Renaming Remotes

The name is a reference to the url, the name been created like:

“git remote add temName url”.

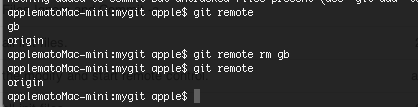
temName=url.

The url is the actual and full url of your remote git-repo, and the name been given by you. Then the rename is rename the temName, such as from temName to abcd. But the url does not change. So if you rename it from temName to abcd, when you use abcd instead of temName, but the actual url value does not had been changed all the time.

If you make a name for the remote git-repo, when you list the remote “git remote” it will list the name, else only list “origin”.

Removing

If you no longer using a particular mirror (remote-repo) when you can remove it using “git remote rm remote\_name”.



All of the name here (rename and remov) it means the name is the your define name. This means that even you rename or remove the name, and you can use the remote repository normally.

Tagging

Auto-Completion and Aliases