# To refer to some detailed information, go to <http://gitbook.liuhui998.com/index.html>

# Fails to make some commits and want to push back out to the GitHub server. The error message is: [Pushing to Git returning Error Code 403 fatal: HTTP request failed](http://stackoverflow.com/questions/7438313/pushing-to-git-returning-error-code-403-fatal-http-request-failed).

Change your repo config on your PC to ssh way: (1) Edit *.git/config* file under your repo directory. (2) Find *url=entry* under section [remote "origin"]. (3) Change it from *url=https://MichaelDrogalis@github.com/derekerdmann/lunch\_call.git* to *url=ssh://git@github.com/derekerdmann/lunch\_call.git*. That is, change all the texts before @ symbol to *ssh://githttp://gitbook.liuhui998.com/index.html*. Save config file and quit. Now you could use *git push origin master* to sync your repo on GitHub. In order to send through SSH protocol, SSH keys should be generated. See <https://help.github.com/articles/generating-ssh-keys>. For some related info:

https://help.github.com/articles/https-cloning-errors. https://help.github.com/articles/which-remote-url-should-i-use

## Git is composed of several simple, unique and run independently commands. The types of command: *git commit* and *git-commit* are totally the same. The types of parameter setting: *-m* (short) or *--message* (long).

## *git init* initialize a Git container on local site. Git creates a hidden folder called *.git* to manage this project and this hidden folder is distributed on the top folder of this project. In the beginning, the *.git/objects* folder is empty. To copy from the remote repository, use *git clone*.

## The key of SHA1 algorithm is 160 bits which is equal to 20 bytes and expressed by 40 hex numbers. The object is stored in *.git/objects/3b/18e512dba79e4c8300dd08aeb37f8e728b8dad*. The first two digits are used to divide into 256 classes to improve the searching efficiency. Exploit the command to search for the complete SHA1 key through short key:

## *$ git rev-parse 3b18*

## *3b18e512dba79e4c8300dd08aeb37f8e728b8dad*

## When you use *git add*, *git rm* and *git mv*, the relative data are created in *.git/index*.

## Before submitting the files to repository, you should setup some environment variables - at least let Git know your name and email address. When you run *git config*, the configuration file is stored in *~/.gitconfig*. Exploit *git config --global* to modify the value of *user.name* and *user.email* in *.gitconfig*:

## *$ git config --global user.name “Price Tseng”*

## *$ git config --global user.email “*[*Everett6802@hotmail.com*](mailto:Everett6802@hotmail.com)*”*

## If you only hope to apply to a specified repository, ignore *--global* parameter.

## *$ git config user.name “Price Tseng”*

## *$ git config user.email “*[*Everett6802@hotmail.com*](mailto:Everett6802@hotmail.com)*”*

## Use *git config -l* to show the detailed configuration and *–-unset* to remove certain a setting:

## *$ git config --unset --global user.email*

## Git exploit SHA1 algorithm to manage each file in the repository. The key of SHA1 algorithm is generated due to the content of the file rather than the file name. So if the contents of two files are the same, the SHA1 keys are also identical. In order to efficiently store each file in each commit, Git only keeps track of the difference between two consecutive commits.

## *git log* can show the information for each commit. If you need more detailed info, use *git show*:

## *$ git show HEAD*

## If you don’t add the SHA key, the last commit is used as default.

## Git divides files into three categories:

## Tracked: Already exists in the repository or index. To make the file be tracked, execute *git add somefile*.

## Ignored: The files ignored by Git. List the file in *.gitignore* .

## Untracked: The file that doesn’t belong to the first two categories.

## The format of *.gitignore* is: 1. *#* symbol. Git doesn’t do anything when the current line is marked as *#*. 2. File name, Git searches for all files of the same filename in every folder. 3. Folder name is marked as */*: Git searches for all folders/sub-folders of the same folder name, but ignore the file and link. 4. Wild word *\**: Like Unix, can be part of file or folder path like *debug/32bit/\*.o*. 5. Exclamation symbol (*!*): The file/folder can’t be ignored, *.gitignore* in local folder has higher priority than the one in parent folder.

## Although *.gitignore* has the special meaning to Git, but the container just seems it as a regular file. Without adding it into index, Git doesn’t track this file. A special case is you want to ignore all the *\*.o* except one file in a specified folder:

## *$ cd my\_package*

## *$ cat .gitignore*

## *\*.o*

## *$ cd my\_package/vendor\_files*

## *cat .gitignore*

## *!driver.o*

## *git commit -a (--all)* command makes the untracked file been tracked. Before committing, the *-a* attribute make Git move all the “tracked” files which are not in *Index* to *Index*. For example:

## *$ touch ready // Create file “ready” and “git add” it to the Index*

## *$ git add ready*

## *$ echo “Test” >> notyet // Modify file “notyet”, leaving it unstaged*

## *$ mkdir subdir // Add a new file in a subdirectory, but don’t add it*

## *$ echo Nope >> subdir/new*

## *$ git status*

## *# Changes to be commited:*

## *# modified: ready*

## *# Chnaged but not updated:*

## *# modified: notyet*

## *# Untracked files:*

## *# subdir/*

## *git commit -a* makes Git travel the container recursively and submits all the files which are in *Index.*

## # Changes to be commited:

## # modified: notyet

## # modified: ready

## # Untracked files:

## # subdir/

## Since *subdir/* is in untracked state, *git commit –a* can’t commit it to Repository.

## When running *git add*, Git exploits SHA1 algorithm to generate a key to the index.

## *git rm* removes the files from working directory and index, and it’s required to run *git commit* to update the change to Repository. *git rm --cached* removes files from the Index but leaves in Working Directory. If you accidentally delete a file you don’t want to, use *git checkout HEAD - WSCGAgent.py* (filename), this file appears again. To rename the file, exploit *git mv*.

## *.git/config* keeps track of the configuration in a specific repository, with higher priority than *~/.gitconfig*.

## For each commit, Git doesn’t record all the files and folders, but record the difference of commit between this and last time.

## *ref* is a hash key referring to certain a Git object. The local branch/remote branch/tag name are all reference. All the references are stored in *.git/refs/*, and begin with *refs/*. *refs/heads/* is for local branch, *refs/remotes/* is for remote branch and *refs/tags/* is for tag. For example, *refs/heads/dev* is a local branch and *dev* for short. If *origin/master* is a remote branch, its full name should be *refs/remotes/origin/master*. If these exists a tag called *v2.6.33*, its full branch should be *refs/tags/v2.6.33*. There are some special branch names below:

## *HEAD*: Always pointed to the latest commit. When you modify the commit, *HEAD* is changed.

## *ORIGIN\_HEAD*: Before merging, this reference keeps track of the last commit so that can be used to return to the old commit.

## *FETCH\_HEAD*: When you use *git fetch*, *.git/FETCH\_HEAD* records the heads of all branches in the remote repository. Only valid when the fetch command is just executed.

## *MERGE\_HEAD:* When starting to merge, the HEAD of another branch is recorded temporarily in this reference.

## *^* represents the different parent objects in the same generation. For example, a commit *C*, and *C^1* and *C^2* is the first and second parent commit.

## *~* represents the ancestor commits. For example, a commit *C*, and *C~1* is the father commit and *C~2* is the grandfather commit.

## *C^* and *C~* is *C^1* and *C~1* for short. *C^^* and *C^1^1* means the first father commit of the first father commit, which is equal to *C~2*.

## Without adding any parameter, *git log* is equal to *git log HEAD*. *git log -1* means only show one commit. Use *git show* command can show more detailed commit information.

## One repository may contain several branches, but only one is active at the moment. Exploit “/” to nominate the branch is a good way to distinguish different branches hierarchically. To create a new branch:

## *$ git branch prs/pr-1138*

## The regular form is *git branch Branch\_Name [The\_Commit\_Where\_The\_Branch\_Start]*. For example:

## $ *git branch prs/pr-1138 db7de5f…*

## Use *–r* attribute to show the remote branches and *–a* represent the branches including local and remote. To switch to another branch, use *git checkout*. When you change to another branch, the distribution of files and folders may be changed. Before switching to another branch, mind that all files changed in the current branch should be commit first. If you don’t care about the changes in the original branch, exploit *git checkout –f*. *git checkout -m new\_branch* is used when you modified some files in old branch, but you would like to switch to new branch and merge the changes into new branch.

## For a specific commit, use the SHA1 hash instead of the branch name as: *git checkout SHA1*.

branches are mutable references and tags are immutable references. For example, we can create a simple tag, based on the current repository’s version, with:

*$ git tag example*

This creates a lightweight tag as a reference in *.git/refs/tags/example*, which points to the current commit. If we want to make it as an annotated tag, we need to supply *-a*, and a message with *-m at a specific commit*:

*$ git tag -a v1 -m "Version 1 release"*

*$ git tag WSG-17865 1064e8 -m”Output format(newAssociation/failedAssociation report)”*

This will create an annotated tag object, containing that message and a pointer to the commit object. Now the reference in *.git/refs/tags/v1* will point to the tag object, which then points to the commit.

To list the local repository’s tags, run *git tag*; or, for a pattern, use *-l* with *\** as a wildcard:

*$ git tag*

*v1*

*v1s*

*$ git tag -l \*s*

*v1s*

Finally, to get rid of tags, you can delete them with *-d*:

*$ git tag -d v1*

*$ git tag*

*v1s*

In order to see what the tag contains, you can use *git show*, as you can with other Git objects:

*$ git show v1s*

*git tag* finds the relative object that the tag points to (*git rev-parse* reverses any kind of reference including tag, branch to SHA1 key):

*$ git rev-parse v1*

*6b698c……*

*$ git cat-file –p 6b69*

*object 6b69…*

*type commit*

*…*

Since a tag is just a reference on your local repository, it’s not sent up by default to the remote repository during pushes. Instead, you can *git push* the tag individually, or run *git push --tags* which will push all tags. To fetch them all, you can do *git fetch --tags* to pull them all in, or *git fetch* *tag* to pull a single one.

## In your current working folder, the files and folders which are not tracked are not been modified. However, if you change some files and then try to switch to another branch without commit, Git issues the warning not to allow this switch.

## To delete a branch, exploit *git branch –d branch*. Git doesn’t allow you to delete the currently active branch. If there are some changes in a branch and you switch to another branch before committing, Git show error messages to stop you doing this. Exploit *–D* instead of *–d* to force Git to switch to new branch, the changes in original branch are lost at this stage.

## Git finally deletes the commits and branches which can’t be referred to. In order to preserve these commits, you should create a new branch or tag to point to them. Otherwise, Git exploits *git gc* to remove them after two weeks (default value).

## *git diff*: Show the difference between current working directory and index.

## *git diff commit*: Find the difference between the current working directory and the commit specified in this command.

## *git diff –-cached commit*: Find the difference between the current index and the commit specified in this command.

## *git diff commit1 commit2*: Find the difference between two specified commits.

## *git diff –-stat*: Get some statistics between commits. *--color*: Mark key word as different color.

## To show the difference in a specified file: *git diff --stat master~5 master Documentation*.

## *git diff CommitA CommitB* and *git diff CommitB CommitA* are different.

## To merge *other\_branch* to *cur\_branch*:

## *$ git checkout cur\_branch*

## *$ git merge other\_branch*

## Your current branch is the target branch and the other is the branch merged into the target branch. While merging branches, and conflict occurs, exploit *git diff* to check the range of conflict. The content between <<<<<<< and ======= is the original content before merging, and ======= and >>>>>>> represents the content after merging. Merge the conflict manually and then use *git commit* to commit the solved conflict. Exploit *git status* to see the files of conflicts.

## Use the following command to detail why conflict occurs:

## *$ git log --merge --left-right –p*

## *--merge*: Show the relative commits about conflict.

## *--left-right*: When commits come from left side (ours) which represent by *<*. When commits come from right side (theirs) which represent by >.

## *-p*: Represent every relationship between every commit and modified file.

## If you are only interested in certain a file: *$ git log --merge --left-right –p hello*.

## *.git/MERGE\_HEAD* contains the SHA1 hash value which you are going to commit.

## You can’t commit the files when the conflict doesn’t be solved.

## To give up merging, exploit: *$ git reset --hard HEAD* before executing the last commit. However, to give up merging after this merge is already complete, use: *$ git reset --hard ORIG\_HEAD*.

## *git reset* adjusts *HEAD* to a specified commit. There are three different cases;

## *git reset --soft commit*: Index and working directory remain un-changed.

## *git reset commit*: Working directory remain un-changed.

## *git reset --hard commit*: All data in working directory and index are changed.

## *git cherry-pick commit* creates a new commit by copying a specified commit from one branch to another. It can be used for re-build a series commits from one branch to another and sometimes you have to solve the conflicts. For example:

## A – B – C – D master

## |

## V – W – X – Y – Z my\_dev

## *$ git checkout master*

## *$ git cherry-pick my\_dev^ #Y*

## *$ git cherry-pick my\_dev~3 #W*

## *$ git cherry-pick my\_dev~2 #X*

## *$ git cherry-pick my\_dev #Z*

## A – B – C – D – Y’ – W’ – X’ – Z’ master

## |

## V – W – X – Y – Z my\_dev

## *git rebase* modify the basic position of a series of commits.

## A – B – C – D – E master

## |

## W – X – Y – Z topic

## All the commits in branch *master* are reserved, only the base of branch *topic* is moved from commit B to E.

## *$ git checkout topic*

## *$ git rebase master*

## Or

## *$ git rebase master topic*

## A – B – C – D – E master

## |

## W’ – X’ – Y’ – Z’ topic

## When you duplicate others’ repository and exploit *git rebase* to move your branch to branch master.

## *git rebase* can exploit *--onto* to move a complete branch from one branch to another totally different branch.

## A – B – C – D - E master

## |

## W – X – Y – Z maint

## |

## P – Q feature

## *$ git rebase --onto master maint^ feature*

## P – Q feature

## |

## A – B – C – D - E master

## |

## W – X – Y – Z maint

## After you solve all the conflicts, check if some files are not in Index, if not, exploits *git add* to add them. Then use *git rebase --continue* to continue the re-base procedure. If you want to give up this re-base, use *git rebase --abort*.

## Exploit *git rebase -i* to re-order, edit, remove or squash several commits.

## $ *git rebase -i master~3*

## pick 3d0f83b Use color instead of colour

## pick 799dba3 Finish my colour haiku

## pick b61b041 Use American spellings

## The first three rows show the range of commits and then change the verb for the commit

## pick 3d0f83b Use color instead of colour

## pick 799dba3 Finish my colour haiku

## squash b61b041 Use American spellings

## The *git clone* command makes the local branches in *refs/heads/* become remote branches in *refs/remotes/*, and the remote branches in *refs/remotes/* aren’t copied. To clone the remote repository:

## *$ git clone git://git.kernel.org/pub/acm/linux/kernel/git/torvalds/linux-2.6.git*

## The *master* branch in original repository becomes the *origin/master* one in the cloned repository.

## Each new cloned repository maintains a new link to the remote (generally called *origin*) to its parent’s repository. This is only one way relation, which means the original repository never knows the information about the one linked to it. Exploit “*--origin name*” to nominate the name of remote repository. Git exploits default *fetch refspec* to set remote repository:

## *fetch = +refs/heads/\*:refs/remotes/origin/\**

## This means Git expects you can update the local repository by fetching the remote repository.

## *git remote* can create, remove, manage and view the content of remote repositories. All the info about remote repository is recorded in *.git/config*. When you clone the remote repository to local, Git create a branch locally to track the remote one.

## The full name of local branch *dev* is *refs/heads/dev* and remote tracking branch *master* is *refs/remote/origin/master*.

## The tracking branches are used for tracking the change of remote branches. It’s incorrect to send a commit to the tracking branch which causes this tracking branch not to synchronize with the remote repository. To track the remote repository, the settings contain two parts:

## URL: Keep track of the name of other container. For example:

## *url = /path/to/repo.git*

## *url = git@github.comn:Everett6802/rat22.git*

## *refspec*: Point out the mapping between local and remote branches. The format is:

## *[+]source:destination*

## Which contains three parts: source reference, semicolon and destination reference. There may be a plus (+) in front of it which means that data transportation doesn’t use security confirmation. Star (\*) is used as wild character.

## *refspecs* is often used by *git fetch* and *git push*. To run *git fetch* or *git push*, you can add multiple *refspecs* so that access the remote repositories simultaneously.

## If you exploit *git init* rather than *git clone* in local folder, there is no *origin* because no remote repository is created. To create remote repository, use *git remote*. This command adds some settings in *.git/config*.

## *$ cat .git/config*

## *[core]*

## *repositoryformatversion = 0*

## *filemode = true*

## *bare = false*

## *logallrefupdates = true*

## *$ git remote add origin /tmp/Depot/public\_html*

## *$ cat .git/config*

## *[core]*

## *repositoryformatversion = 0*

## *filemode = true*

## *bare = false*

## *logallrefupdates = true*

## *[remote “origin”]*

## *url = /tmp/Depot/public\_html*

## *fetch = +refs/heads/\*:refs/remotes/origin/\**

## Generally, the basic repository is nominated by *origin*. To list all branches:

## *$ git branch –a*

## *\* master*

## *$ git remote update*

## *Updating origin*

## *From /tmp/Depot/public\_html*

## *\* [new branch] master -> origin/master*

## *$ git branch –a*

## *\* master*

## *origin/master*

## Git create a new tracking branch called *origin/master* to track the branch *origin* of remote repository. Nobody sends the commit to this branch since its goal is to track the branch *master* of the remote repository *origin*.

## Exploit *git remote show* to show all relative information about remote repository.

## The message “Updating origin” is generated when run *git remote update*, but it means that the branch *origin* in local repository is changed due to remote repository rather than the remote repository is updated. After that, you complete to create a link between local and remote repositories.

## To remove the specified repository, exploit *git remote rm*. Ex: *git remote rm origin*.

## To send a commit to remote repository, exploit *git push*. Git create a new node on *origin/master* branch simultaneously. Exploit *git branch -a* to show all the branch details.

## *$ git branch –a*

## *\* master*

## *origin/HEAD*

## *origin/master*

## *master* is the active local branch. *origin/master* is the tracking branch to track the commits in the branch *master* of remote repository *origin*. *origin/HEAD* is the current active branch in the remote repository.

若有一個以上遠端儲存庫，此命令會列出全部。例如：我的Grit儲存庫包含以下遠端儲存庫。

*$ cd grit*

*$ git remote -v*

*bakkdoor git://github.com/bakkdoor/grit.git*

*cho45 git://github.com/cho45/grit.git*

*origin git@github.com:mojombo/grit.git*

可從伙伴儲存庫取得最新的更新。要留意的是只有origin遠端的URL是SSH。因此它是唯一能上傳的遠端的儲存庫。

Exploit *git pull* to get the codes from the remote repository: *git pull options repository refspecs*.

## *git pull* is the combination of *git fetch* and *git merge*. If *repository* is not specified, Git exploits *origin* as default. If *repository* is specified, *refspecs* is not, *HEAD* in *repository* is exploited.

## If the first step of *git pull*, which is equal to *git fetch*, if the command doesn’t assign the name of remote repository, Git exploits *origin* as default.

## *[remote “origin”]*

## *url = /tmp/Depot/public\_html.git*

## *fetch = +refs/heads/\*:/refs/remotes/origin/\**

## Now Git knows to use the URL: */tmp/Depot/public\_html.git* as the source container.

## It’s not required to use wild character (*refs/heads/\**) to represent all branches. You can only assign specified branches and list them in *.git/config*:

## *[remote “newdev”]*

## *url = /tmp/Depot/public\_html.git*

## *fetch = +refs/heads/dev:refs/remotes/newdev/dev*

## *fetch = +refs/heads/stable:refs/remotes/newdev/stable*

## In the second step of *git pull*, Git merges or rebases the local repository. Generally, Git exploits “fast forward” to merge the tracking branch *origin/master* into your local branch *master*.

## *[branch “master”]*

## *remote = origin*

## *merge = refs/heads/master*

## When running *git fetch* or *git pull,* andGit exploits *master* as the active branch, *origin* is viewed as remote repository. In the second steps (merge) of *git pull*, Git merges from *refs/heads/master* into *master* as default.

## If a new branch is created in the remote repository, Git can add a new branch to track this new remote branch:

## *$ git branch mydev origin/master*

## *[branch “mydev”]*

## *remote = origin*

## *merge = refs/heads/master*

## Origin

## master

## |

## -------------A – B

## Yours

## origin/master

## |

## -------------A – B – X – Y

## |

## master

## If you want to push the commits, Git submits your commit to *origin*. Then Git exploits fast-forward policy.

## Origin

## master

## |

## -------------A – B – C – D

## Yours

## origin/master

## |

## -------------A – B – X – Y

## |

## master

## In this situation, if you try to push the commits, Git reject you and give you the message about the conflict.

## If you really want to overwrite others’ changes, you can do by *git push –f*. Usually, you merge two commits to solve the conflict before submitting to the remote.

## Running *git fetch* to get commits from the remote repository to local:

## Origin

## master

## |

## -------------A – B – C – D

## Yours

## origin/master

## |

## C – D

## /

## -------------A – B – X – Y

## |

## master

## When getting the commit *C* and *D* from the remote, Git doesn’t change the commit X and Y in the local. The next step is to merge these two branches by calling or the second step of *git pull*.

## Yours

## origin/master

## |

## C – D

## / \

## -------------A – B – X – Y – M

## |

## master

## Exploit *git reset --hard ORIG\_HEAD* to give up merging and go back to the former status. In this case, *master* stays in commit *Y* and *origin/master* in commit *D*.

## Since the remote repository never knows any changes about the local repository, Git has to notify the remote. According to the *refspecs* (*[+]source:destination*) format, to add a new branch in the remote repository, only exploit source reference without destination reference:

## *$ cd ~/public\_html*

## *$ git checkout –b foo*

## *Switched to a new branch “foo”*

## *$ git push origin foo*

## *Total 0 (delta 0), reused 0 (delta 0)*

## *To /tmp/Depot/pubic\_html*

## *\* [new branch] foo -> foo*

## When only use the destination reference without source reference and pull to remote, a specified branch in the remote repository is deleted.

## *$ git push origin :foo*

## *To /tmp/Depot/public\_html*

## *- [deleted] foo*

## Git provides three ways to setup and maintain the information of remote repository: *git remote*, *git config* and *.git/config*. They finally modify the *.git/config* file.

## *git config* can modify some parameters of remote repository.

## *$ git config remote.publish.url ‘ssh://git.example.org/pub/repo.git’*

## *$ git config remote.publish.push ‘+refs/heads/\*:refs/heads/\*’*

## *[remote “publish”]*

## *url = ssh://git.example.org/pub/repo.git*

## *push = +refs/heads/\*:refs/heads/\**

*HEAD*好似一個游標，指向最新提交，隨最新提交向前移動。一些Git命令讓你來移動它。例如：

*$ git reset HEAD~3*

將立即向回移動*HEAD*三個提交。這樣所有Git命令都好似沒有做那最後三個提交，然而你的檔案保持在現在的狀態。具體應用參見幫助頁。但如何回到將來呢？過去的提交對將來一無所知。如果你有原先Head的SHA1值，那麼：

*$ git reset 1b6d*

但假設你從來沒有記下呢？像這些命令，Git保存原先的Head為*ORIG\_HEAD*的標記，你可以安全體面的返回：

*$ git reset ORIG\_HEAD*

**Delete a local commit**

Let’s say there is a repository with 4 commits.

$git log --pretty=oneline --abbrev-commit

46cd867 Changed with mistake

d9f1cf5 Changed again

105fd3d Changed content

df33c8a First commit

Commit 46cd867 is the most recent commit and the one we want to delete, for doing that, we will use rebase.

$git rebase -i HEAD~2

That command will open your default text editor with your two latest commits:

pick d9f1cf5 Changed again

pick 46cd867 Changed again with dashes

…

The most recent commit is the one at the bottom. None of options is going to be used, just need to delete the line corresponding to the commit we want to delete and save the file. The change was applied correctly:

$git log --pretty=oneline --abbrev-commit

d9f1cf5 Changed again

105fd3d Changed content

df33c8a First commit

**Delete a remote commit:** *git push -f origin master*

git-show-branch“–”符號是一個分隔符，用於區分各個列用的，在“–”符號上部的內容顯示分支列表，而“–”下部的內容為分支commit的關係。–”符號上部的\*(星號)表HEAD所指的分支，其他分支則標識為!(嘆號)。“–”符號下部的內容：+(加號)表所在分支包含此行所標識的commit； (空格)表所在分支不包含此行所標識的commit；-(減號)表所在分支是經過merge得到的，而所在行的內容即merge的基本信息；\*(星號)表如果需要在某列標識+(加號)，且此列為當前分支所在列，那麼則將+(加號)轉變為\*(星號)。我們來分析一個例子：

|  |
| --- |
| **[** rocrocket **@** wupengchong showbranch **]** $ **git-show-branch**  **\***  **[** master **]** merge  **!**  **[** wukong **]** wukong:Thirdly  --  -  **[** master **]** merge  **\*** + **[** wukong **]** wukong:Thirdly |

在“--”分隔符號之上：1.支為master分支，而wukong分支並非當前分支。2.master分支的HEAD所指向的commit的開發日誌的首行為“merge”字符串，而wukong分支的HEAD所指commit的開發日誌的首行信息為“wukong: Thirdly” 。  
在“--”分隔符號之下：1.根據“- ”符號，master分支的HEAD的最近一次commit是由merge得到的，且merge後commit的信息為“merge”。在“-”後還有一個空格，說明當前行所指commit對wukong分支(第二列)沒有影響，所以用“空格”表示。2.據“\*+”所在列，當前行的commit影響範圍到master分支和wukong分支，此次commit在兩個分支都有效。這是merge的力量，說明在wukong分支的這次commit(開發日誌的後行為“wukong: Thirdly”)被merge到了master分支。  
[Import existing source code to github](http://stackoverflow.com/questions/4658606/import-existing-source-code-to-github)

If you've got local source code you want to add to a new remote new git repository without 'cloning' the remote first, do the following.

1. Create the remote repository, and get the URL such as *git://github.com/youruser/somename.git*.

*If your local GIT repo is already set up, skips steps 2 and 3*

1. Locally, at the root directory of your source, *git init*.
2. Locally, add and commit what you want in your initial repo (for everything, git add . git commit -m 'initial commit comment').
3. Attach your remote repo with the name '*origin*': *git remote add origin [URL From Step 1]*.
4. Push up the master branch (change *master* to something else for a different branch): *git push origin master*.

当在做一项复杂的工作时, 发现了一个和当前工作不相关但是又想先修复bug, 可用[*git stash*](http://www.kernel.org/pub/software/scm/git/docs/git-stash.html)保存当前的工作状态, 等复完bug后,执行'反储藏'(unstash)操作就可以回到之前的工作里.

*$ git stash save "work in progress for foo feature"*

上面命令保存本地修改到储藏(stash)中, 然后将工作目录和索引里的内容全部重置, 回到当前所在分支的上次提交时的状态. 现在就可以开始你的修复工作了.

... edit and test ...

*$ git commit -a -m "blorpl: typofix"*

当修复完bug后, 可用git stash apply来回复到以前的工作状态. Mind to switch to correct branch before recovering the un-complete former codes.

*$ git stash apply*

可多次使用'*git stash*'命令,每执行一次就会把针对当前修改的‘储藏’(stash)添加到储藏队列中. 用'*git stash list*'命令可以查看你保存的'储藏'(stashes):

*$ git stash list*

*stash@{0}: WIP on book: 51bea1d... fixed images*

*stash@{1}: WIP on master: 9705ae6... changed the browse code to the official repo*

用类似'*git stash apply stash@{1}*'来使用在队列中的任一个'储藏'(stashes). '*git stash clear*‘则是清空这个队列.

## 使用Git Grep进行搜索

用[git grep](http://www.kernel.org/pub/software/scm/git/docs/git-grep.html) 命令查找Git库里面的某段文字是很方便的. 当然也可以用unix下的'grep'命令进行搜索, 但是'git grep'命令能让你不用签出(checkout)历史文件, 就能查找它们. 如要看仓库里每个使用'xmmap'函数的地方, 可以运行下面的命令:

*$ git grep xmmap (or git grep –e ‘xmmap’)*

如果你要显示行号, 你可以添加'-n'选项:

*$>git grep -n xmmap*

如果我们想只显示文件名, 我们可以使用'--name-only'选项:

*$>git grep --name-only xmmap*

我们可以用'-c'选项,可以查看每个文件里有多少行匹配内容(line matches):

*$>git grep -c xmmap*

如果要查找git仓库里某个特定版本里的内容, 可以像下面一样在命令行末尾加上标签名(tag reference):

*$ git grep xmmap v1.5.0*

可以组合一些搜索条件, 下面的命令就是查找我们在仓库的哪个地方定义了'SORT\_DIRENT'.

*$ git grep -e '#define' --and -e SORT\_DIRENT*

不但可以进行“与"(both)条件搜索操作，也可以进行"或"(either)条件搜索操作.

### 取消已經暫存的檔案

假如我們有兩個檔案需要 commit，但是不小心按到 git add \* 全部加入到暫存區，那該怎麼恢復呢？

*# On branch master*  
*# Changes to be committed:*  
*#   (use "git reset HEAD <file>..." to unstage)*  
*#*  
*#       modified:   Makefile*  
*#       modified:   user/easy\_setup/easysetup.h*  
*#*

上面是已經在暫存區裡面等待被commit檔案(Changes to be committed)，括號裡有提示如何拿掉 (use “git reset HEAD …” to unstage)，所以我們下：

git reset HEAD user/easy\_setup/easysetup.h

之後會看到 『user/easy\_setup/easysetup.h: locally modified』此訊息，這時候在用 git status 看狀態

*# On branch master*  
*# Changes to be committed:*  
*#   (use "git reset HEAD <file>..." to unstage)*  
*#       modified:   Makefile*  
*# Changed but not updated:*  
*#   (use "git add <file>..." to update what will be committed)*  
*#       modified:   user/easy\_setup/easysetup.h*

### 取消對檔案的修改

如果剛剛針對Makefile的修改覺得不需要，該如何取消修改，以及恢復檔案狀態呢？尚未恢復之前的狀態如下：

*# On branch master*  
*# Changed but not updated:*  
*#   (use "git add <file>..." to update what will be committed)*  
*#       modified:   Makefile*

然後恢復檔案指令：

git checkout -- Makefile

接下來用 git status 看看狀態：

*# On branch master*  
nothing to commit (working directory clean)

可以看到，該檔案已經恢復到修改前的版本。請**注意**：這指令有些危險，針對該檔案的修改都沒有了，因為我們剛剛把之前版本的檔案複製過來覆蓋了此檔案。所以在用這指令前，請務必確定真的不再需要剛剛的修改。

### 恢復 Commit，重新提交 Commit

假如今天您修改了兩個檔案，並且新增到暫存區了，狀態如下：

*# On branch master*  
*# Changes to be committed:*  
*#   (use "git reset HEAD <file>..." to unstage)*  
*#       modified:   Makefile*  
*#       modified:   user/easy\_setup/easysetup.h*

並且已經執行了 git commit -m “first commit”，發現訊息寫錯想要修改，可透過 reset來做，先用 **git log**查看訊息

git log --stat

commit c13eb41110a38ef7145bb8815560641697800659  
Author: appleboy <appleboy.tw at gmail.com>  
Date:   Fri Aug 20 20:02:55 2010 +0800  
    first commit  
 Makefile                    |   10 ++-  
 user/easy\_setup/easysetup.h |  157 ++++++++++++++++++++-----------------------  
 2 files changed, 79 insertions(+), 88 deletions(-)

這要怎麼解決才可以改變此訊息呢？

git reset --soft HEAD^

假如要修改commit訊息，或者是尚未修改好檔案，下完此指令，會發現reset會把舊的 head 複製到 **.git/ORIG\_HEAD**，如果您沒有想要重新建立 commit 訊息，您可以下

git commit -a -c ORIG\_HEAD

git 會跳出剛剛的 first commit 讓您來修改，或者是自己用 git commit -m “XXXXXXX” 這樣也可以喔。