Git Guide

Meher Krishna Patel

Created on : Octorber, 2017

Last updated : October, 2018

Table of contents

Chapter 1

Commands Summary

Following is the list of commands which are used in this tutorial.

Table 1.1: Commands summary

Commands	Descriptions
Local Repository	
sudo apt-get install git	Install git in Linux Ubuntu
git config –global user.name *meher*	Set username as 'meher'
git config –global user.email *abc@gmail.com*	Set email as 'abc@gmail.com
git config –global core.editor "vim"	Set 'vim' as default text-editor
git config –global credential.helper cache	Cache username and password
git init	Initialize git repository
git status	File status i.e. modified and untracked etc.
git add .	Add all untracked files
git add *file1 file2*	Add (stage) file1 and file2
git rm –cached *file1*	Remove the staged file *file1*
git commit -m *"commit message"*	Commit stage file with 'commit message'
git log	Show detail list of commits
git log –oneline	Show hash and commit name only
git log –graph	Show commits in the form of graph
git log –oneline –graph	Show online-commit in the form of graph
git diff	Differences between unstaged files and previous commit
git diff –cached	Differences between staged files and
git diff –stat	Show only changed filenames (not the details)
git reset	Remove *all files* from stage list (i.e. back to modified)
git reset *file1*	Remove *file1* from stage list (i.e. back to modified)
git reset –hard *13802e3*	Reset to previous commit with hash 13802e3
git reset HEAD –hard	remove all changes after last commit
git checkout *file1*	Remove changes from non-staged file1 to previous commit
git rm *file1*	Delete file1 from git (but available in previous commit
git branch	Show all the branches
git branch *branch1*	Create branch1
git branch -d *branch1*	Delete branch1
git checkout *branch1*	Go to branch1
git checkout master	Go to master branch
git merge *branch1*	Merge the *brach1* to current branch e.g. master
git checkout *13802e3*	Create new branch from previous commit 13802e3
git checkout -b *branch1*	First checkout and then create branch
Remote repository	
git remote add *repoName https://url_of_repo*	Add remote repo with name 'repoName'

Continued on next page

Table 1.1 – continued from previous page

Commands	Descriptions
git remote -v	Show list of added repoNames
git remote remove *repoName*	Remove repoName from list
git push *repoName branch1*	Push 'branch1' to 'repoName'
git push *repoName* –all	Push all branches to repoName
git clone https://nameOfRemoteRepository	Clone or download remote repository
git clone –depth 1 https://nameOfRemoteRepository	Clone only last branch
git pull *repoName branchName*	Download and merge 'branchName' of repoName
git fetch *repoName branchName*	Download, but not merge repoName

Chapter 2

Git Local repository

Git is the version control system which is used for tracking the changes in the codes. Also, if we made some wrong changes in the code or we are not happy with current changes, then the previous versions of codes can be restored using git. Lastly, version control systems allows to work in a group, so that different people can solve different problems and finally all the solution can be merge together. In this way, projects are more manageable and can be completed efficiently. Further, git repository can be stored on the local machine as well remotely on the web. In this chapter, git local repository is discussed, whereas git remote repository described in next chapter. Finally, last chapter shows the list of command which we learn in this tutorial.

2.1 Installation

We need to install git to use this tutorial. Git can be install on **Ubuntu** using following command.

```
sudo apt-get install git
```

For more information or to install Git on other linux systems, window or macs, please visit the git website http://git-scm.com/

2.2 Git Configuration

First we need to configure the git setting so that git can identify the person who is making changes in the codes. For this use following commands,

First two commands set the username and email id, whereas third command (optional) sets the default text-editor for git. For example, vim is set as default editor here, therefore vim will be open for the commands which requires some inputs from users.

```
$ git config --global user.name meher
$ git config --global user.email mkpatel@gmail.com
$ git config --global core.editor "vim"
```

Git stores these information in .gitconfig file, which is stored in the home directory. In Ubuntu, content of this file can be seen using following commands,

```
$ cd
$ vim .gitconfig
[user]
  email = mkpatel@gmail.com
  name = meher
[core]
  editor = vi
```

Use following command to cache the username and password, so that we need not to give credential values, i.e. username and password, for each git connection to remote repository,

```
$ git config --global credential.helper cache
```

2.3 Create project directory

Lets create a project directory at any desired location and initialize the git repository in that directory as follows,

```
$ mkdir gitProject
$ cd gitProject
$ git init
```

Last command will create a git repository, which is store in .git folder. Use following commands to view the content of this folder,

```
$ ls -A
.git

$ cd .git && ls -A
branches  config  description  HEAD  hooks  info  objects  refs

$ cd ..
```

Now, any changes in the folder 'gitProject' will be tracked by the git. To see the change status in the folder, use 'status' command as follows,

```
$ git status
Initial commit
nothing to commit
```

Since, no changes are made in the directory, therefore status commands shows that there is 'nothing to commit'.

2.4 Add files

Next, add one file e.g. 'hello.py' in the folder, with following contents,

```
# hello.py


print("hello 1")
```

Now, run the status command again and the changes in the directory will be displayed as below,

```
$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

  .hello.py.swp
  hello.py

nothing added to commit but untracked files present (use "git add" to track)
```

Above message shows that there are two untracked files in the project directory i.e. hello.py and .hello.py.swp. Second file i.e. .swp is the file generated by vim editor. Since, we do not want to keep track of files which are generated by editors or software e.g. .pyc (python) or .so(C/C++), therefore these files should be removed from tracking list; which can be done by using .gitignore file as shown next.

2.5 .gitignore

Create a .gitingore file in project directory; add file extensions and folder names, to avoid tracking of certain files e.g. automatically generated files i.e. .pyc and .swp or folders i.e. build etc. These files can be added to .gitignore file as shown below,

```
.gitignore
*.swp
*.pyc

# sphinx-build folders
build
:generated

# Byte-compiled / optimized / DLL files
__pycache__/
*.py[cod]
*$py.class

# C extensions
*.so
```

After adding .gitingore file, run status command again as below; we can see that .swp file is not displayed now, as it is removed from the tracking list.

```
$ git status
[...]
Untracked files:
  (use "git add <file>..." to include in what will be committed)

  hello.py
[...]
```

2.6 Untracked, Staged and Modified file

2.6.1 Untracked file

The newly added files in the project directory are considered as untracked files. Since hello.py is created curretly and it is new to git therefore it is shown as 'Untracked' file by status command. Status will remain the same until this file is added to git as shown next.

2.6.2 Staged file

When any file (untracked or modified) is added to git, then it is called staged file. 'git add' command' is used to add file to git as shown in below commands,

```
$ git add hello.py
$ git status
[...]
Changes to be committed:
```

(continues on next page)

(continued from previous page)

```
  (use "git rm --cached <file>..." to unstage)

  new file:   hello.py
[...]
```

Results of above status commands shows two things.

- First, use "git rm –cached <file>..." to unstage, which means that files is staged and we can use 'git rm' command to unstage it.
- Further, stage files are the files which are ready for the backup (but not backed up yet), hence git displays the message that 'changes to be committed, new file: hello.py'. Commit is used to backup the staged file as shown next.

2.6.3 Commit

'git commit' command is used to store the changes in the git so that these can be recovered later. Each commit requires a name so that we can identify the changes made during those commits. Commit operation can be performed as below,

```
$ git commit -m "commit1 hello.py is added"
[master (root-commit) 5e1b96b] commit1 hello.py is added
 1 file changed, 5 insertions(+)
 create mode 100644 hello.py

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

Since, all the changes are added to git using commit, therefore status command displays that there is nothing to commit now.

Note that, if 'git commit' is used in place of 'git commit -m "commit1 hello.py is added"', then a vim editor will pop-up (as vim is set as default editor) and we need to enter the commit message on the top of that file i.e. "commit1 hello.py is added".

2.6.4 Modified files

If we change the 'hello.py' file again, then git considered as modified file (instead of new file). To understand this, let add one more line at the end of hell.py as below,

```
# hello.py


print("hello 1")
print("hello 2")
```

Now, use git status command and it will show that the hello.py file is modified file. Also, we can add and commit the modified file (not new file) using one line command i.e. git -am as shown below,

```
$ git status
[...]
Changes not staged for commit:
[...]
  modified:   hello.py

$ git commit -am "Commit2 hello.py modified"
[master d524017] Commit2 hello.py modified
 1 file changed, 1 insertion(+)
```

2.7 Git log

Git log command is used to see the list of commits. Following git log commands are quite useful,

```
$ git log
commit d52401733f7cd237cd837cd362bf3e0c546aef47
Author: meher <mkpatel@gmail.com>
Date:   Mon Jan 30 22:13:08 2017 +0000

    Commit2 hello.py modified

commit 5e1b96bd7e89c94ccb1b6b85704ed23958cdff59
Author: meher <mkpatel@gmail.com>
Date:   Mon Jan 30 22:00:42 2017 +0000

    commit1 hello.py is added

$ git log --oneline
d524017 Commit2 hello.py modified
5e1b96b commit1 hello.py is added
```

'git log' displays the details of commits whereas 'git log –oneline' give the hashes of the commits (i.e. numbers at the beginning) and commit name only.

To see list of commits in the form of graph, use following command (red lines will be shown on the side of the commits, which are quite useful, when branching is used in later part of the tutorial).

```
$ git log --graph
[...]

$ git log --oneline --graph
[...]
```

2.8 Git diff

'git diff' is used to see the difference between last commit and current files. Let add one more line at the end of hello.py file. Then run the 'git diff' command to see the changes in the file as below.

```
# hello.py


print("hello 1")
print("hello 2")
print("hello 3")
```

- **git diff**: shows the differences between **unstaged files** and previous commit
- **git diff - -cached** shows the differences between **staged files** and previous commit.
- **- -stage**: it is used to avoid details of the differences.

```
$ git diff
diff --git a/hello.py b/hello.py
index 78860b8..181ed03 100644
--- a/hello.py
+++ b/hello.py
@@ -3,4 +3,5 @@

 print("hello 1")
 print("hello 2")
+print("hello 3")
```

(continues on next page)

(continued from previous page)

```
$ git diff --stat
 hello.py | 1 +
 1 file changed, 1 insertion(+)

$ git diff --cached
```

Note that - -cached command does not show any difference as there is no difference between stage file and previous commit. Once hello.py is staged, then - -cached command will show the differences as displayed below,

```
$ git add hello.py
$ git diff

$ git diff --cached
diff --git a/hello.py b/hello.py
index 78860b8..181ed03 100644
--- a/hello.py
+++ b/hello.py
@@ -3,4 +3,5 @@

 print("hello 1")
 print("hello 2")
+print("hello 3")

$ git diff --cached --stat
 hello.py | 1 +
 1 file changed, 1 insertion(+)
```

Finally commit these changes as follows,

```
$ git commit -m "commit3 hello 3 is added"
```

2.9 Git reset

'git reset' is opposite of 'git add'.. Reset command can be used in two ways, i.e. to reset the staged files in current working directory or reset the directory to previous commits, as discussed in this section.

2.9.1 Reset stage file

Reset can be used when we want to unstage some file which is added accidentally. Let add one more line in the end of hello.py as below,

```
# hello.py


print("hello 1")
print("hello 2")
print("hello 3")
print("hello 4 Password : 1234")
```

Next, stage this file using 'git add' and then unstage it using 'git reset' command as shown below,

```
$ git status
[...]
Changes not staged for commit:
[...]
    modified:   hello.py
```

(continues on next page)

(continued from previous page)

```
[...]

$ git add hello.py
$ git status
[...]
Changes to be committed:
[...]
    modified:   hello.py

$ git reset hello.py
Unstaged changes after reset:
M   hello.py

$ git status
[...]
Changes not staged for commit:
[...]
    modified:   hello.py
[...]
```

2.9.2 Reset to previous commit

In previous section, reset command is used to unstage the file which is not committed. If file is committed, then we need to reset the header i.e. go to previous commits to remove the file from the commit.

This can be very useful, when we committed something wrong (e.g. saved the password in some file) and want to remove those changes. In such cases, we need to reset the header, i.e. we need to go back to previous commit, as discussed next.

First stage and commit the changes made in the previous section, as shown below,

```
$ git commit -am "commit4 password added wrongly"
```

Now, use 'revert –hard' command to remove the changes i.e. the last line which contains the password. First see the list of commits using 'git log' command,

```
$ git log --oneline
 b78daef commit4 password added wrongly
 13802e3 commit3 hello 3 is added
 d524017 Commit2 hello.py modified
 5e1b96b commit1 hello.py is added
```

Header 'b78daef' is the commit in which password is stored, and we want to go to previous commit i.e. header 13802e3. For this use reset command as shown below,

```
$ git reset --hard 13802e3
HEAD is now at 13802e3 commit3 hello 3 is added

$ git log --oneline
13802e3 commit3 hello 3 is added
d524017 Commit2 hello.py modified
5e1b96b commit1 hello.py is added
```

Note that, the header 'b78daef' is removed now. If we look at hello.py again, then we will see that the last line, i.e. ' print("hello 4 Password : 1234")' is removed from the file.

2.10 Git checkout

Suppose we add one more line at the end of hello.py, which is **not staged** till now. Then, 'git checkout' command can be used to remove the changes. To understand this, add one line to hello.py as below,

```
# hello.py


print("hello 1")
print("hello 2")
print("hello 3")
print("hello 4 checkout example")
```

Now, run the checkout command as below. Since, hello.py file is modified, therefore 'M hello.py' is shown by status command. Next, checkout command is used for hello.py which revert the changes in hello.py to previous commit. When status command is again run, it does not display anything as everything is same as previous commit. Also, if we look hello.py, then we find that the last line i.e. 'print("hello 4 checkout example")' is removed from the file.

```
$ git status -s
M hello.py

$ git checkout hello.py
$ git status -s
```

2.11 Git rm

'git rm' is used to delete the file and commit it as shown below,

```
$ git rm hello.py
rm 'hello.py'

$ git status
[...]
    deleted:    hello.py

$ git commit -m "hello.py delete"

$ git log --oneline
8c168c6 hello.py delete
13802e3 commit3 hello 3 is added
d524017 Commit2 hello.py modified
5e1b96b commit1 hello.py is added
```

Use reset command to restore the file as shown below,

```
$ git reset --hard 13802e3
HEAD is now at 13802e3 commit3 hello 3 is added

$ git log --oneline
13802e3 commit3 hello 3 is added
d524017 Commit2 hello.py modified
5e1b96b commit1 hello.py is added
```

2.12 Git branch

Branches are the useful concept in git. Currently, we have only one branch i.e. master, which can be seen using git branch command,

```
$ git branch
  * master
```

With the help of branches, we can experiment with the codes, without touching the code in master branch. Later, after completing the the experiments, we can add useful changes to master branch as shown below,

2.12.1 Create and delete git branch

Let create one branch with name 'add2Num',

```
$ git branch add2Num
$ git branch
    add2Num
  * master
```

'git branch' shows two branches i.e. add2Num and master. The * sign shows that currently we are in master branch. 'git branch -d' is used to delete a branch, e.g.

```
$ git branch diff2Num
$ git branch
    add2Num
    diff2Num
  * master

$ git branch -d diff2Num
$ git branch
    add2Num
  * master
```

2.12.2 Switch between branches

Checkout command is used to switch between branches,

```
$ git checkout add2Num
Switched to branch 'add2Num'

$ git branch
* add2Num
  master
```

Now, * is on add2Num branch. Next make some changes in the hello.py again. **Add following lines at the end of the code, not on the top.** If we add code at the top, git will generate 'conflict', which we will discuss later.

```
# hello.py


print("hello 1")
print("hello 2")
print("hello 3")

x = 2
y = 3
print (x+y)
```

Next, commit these changes as below,

```
git status -s
M hello.py
```

(continues on next page)

(continued from previous page)

```
$ git commit -am "commit from add2Num"
```

Now, switch back to **master branch** and see the code. **We can see that, when we switch back to master, the last 3 lines are removed**.

```
$ git checkout master
```

Next, switch to **add2Num branch** again as below, and **last three line will appear again**. In this way, we can experiment with the codes without affecting the main branch,

```
$ git checkout add2Num
```

Also, note the differences between 'git log' for both the branches. Currently, 'master' branch contains only 3 headers, whereas add2Num branch contains 4 header as below,

```
$ git checkout master
Switched to branch 'master'

$ git log --oneline
13802e3 commit3 hello 3 is added
d524017 Commit2 hello.py modified
5e1b96b commit1 hello.py is added


$ git checkout add2Num
Switched to branch 'add2Num'

$ git log --oneline
223e688 commit from add2Num
13802e3 commit3 hello 3 is added
d524017 Commit2 hello.py modified
5e1b96b commit1 hello.py is added
```

2.12.3 Git merge

In previous section, we create a new branch and modified the code. Now, we are done with the change and want to include those changes in the master branch. Merge command is used for this purpose.

First go to master branch, and then merge the add2Num branch as shown below,

```
$ git checkout master
Switched to branch 'master'

$ git merge add2Num
Updating 13802e3..223e688
[...]

$ git log --oneline
223e688 commit from add2Num
13802e3 commit3 hello 3 is added
d524017 Commit2 hello.py modified
5e1b96b commit1 hello.py is added
```

Now we can see that the master branch has have 4 commits i.e. commits from add2Num branch is added to master.

2.12.4 Conflicts

If two branches contains different code at same lines of same file, then conflict will be generated by git; as git is unable to understand the correct version of the code. Lets, see it with example.

First checkout to add2Num branch and then modify the 'print("Hello 1")' line to 'print("Hello 1 from add2Num")' as below,

```
# hello.py


print("hello 1 from add2Num")
print("hello 2")
print("hello 3")

x = 2
y = 3
print (x+y)
```

Next, commit the changes as below,

```
$ git checkout add2Num
Switched to branch 'add2Num'

$ git commit -am "Hello 1 from add2Num"
```

Now, go to master branch and modify the hello.py again with message "print(hello 1 from master)" as shown below,

```
# hello.py


print("hello 1 from master")
print("hello 2")
print("hello 3")

x = 2
y = 3
print (x+y)
```

Commit the changes and try to merge the branches. We can see that, git generates a conflict as shown below, because both branches are changing the same file with different content as same line,

```
$ git commit -am "Hello 1 from master"

$ git merge add2Num
 Auto-merging hello.py
 CONFLICT (content): Merge conflict in hello.py
 Automatic merge failed; fix conflicts and then commit the result.
```

2.12.5 Removing conflicts

Conflicts need to be removed manually. If we open the hello.py file, then it will look as below,

```
# hello.py


<<<<<<< HEAD
print("hello 1 from master")
=======
print("hello 1 from add2Num")
```

(continues on next page)

(continued from previous page)

```
>>>>>>> add2Num
print("hello 2")
print("hello 3")

x = 2
y = 3
print (x+y)
```

The lines between arrows, are the line which are in conflict. Also, = sign separates the line which are generating conflicts. Now, we need to modify the code manually, which can be done as below,

```
# hello.py


print("hello 1 from master and add2Num")

print("hello 2")
print("hello 3")

x = 2
y = 3
print (x+y)
```

Finally, commit the changes and see the 'git log' as below.

```
$ git commit -am "Hello 1 accepted from both"

$ git log --oneline
54133a4 Hello 1 accepted from both
6f0d127 Hello 1 from master
5336b3c Hello 1 from add2Num
223e688 commit from add2Num
13802e3 commit3 hello 3 is added
d524017 Commit2 hello.py modified
5e1b96b commit1 hello.py is added
```

2.12.6 Create branches through older commits

Suppose we want to create a new branch 'diff2Num' from the commit3 i.e. through the header '13802e3'. For this, first we need to checkout the header and then create a branch there using 'git branch -b' command as below,

```
$ git checkout 13802e3
Note: checking out '13802e3'.
[...]

$ git checkout -b diff2Num

$ git branch
  add2Num
* diff2Num
  master
```

In above listing, * shows that currently git is in diff2Num branch. Next, check the 'git log' command to confirm that the new branch starts from third commit as below.

> $ git log –oneline 13802e3 commit3 hello 3 is added d524017 Commit2 hello.py modified 5e1b96b commit1 hello.py is added

Now, we can modify the code from third commit and merge those changes to master or work separately on this branch.

2.13 Git GUI

There are various graphical user interfaces are available e.g. git-cola, git-ext and giggle etc. Once, basic concepts are understood, then we can use those interfaces to perform various operations using mouse clicks.

Chapter 3

Git remote repository

In previous chapter, we make a local repository which stores all the changes on our local machine. To store the repository online, i.e. remote repository, we need to create an account on git-hosting-sites e.g. "BitBucket (which provides free private repository)" or "GitHub" etc.

3.1 Creating repository

First login to hosting site and create a repository there. Further, set it as public or private repository.

3.2 Add Repository to local git

Use following command to remote repository location to the git,

```
$ git remote add repoName https://url_of_repository
```

In above command, replace following according to your project,

- https://url_of_repository : replace this with actual web address of the repository.
- repoName : give it a suitable name, which can be used as short-name of above url. In the other words, we can use url and repoName interchangeably. Further, **origin** is the preferred name for this purpose.

Use following, command to see the detail of added repository

```
$ git remote -v
repoName  https://url_of_repository (fetch)
repoName  https://url_of_repository (push)
```

In this way, we can add other contributers repository as well, which are working on the same project, which is discuss later part of the tutorial. Further, use following command to remove certain repoName,

```
$ git remote remove repoName
```

3.3 Push changes on repository

'git push' command is used to push the local changes to remote repository as below,

```
$ git push repoName branchName
$ git push repoName --all
```

In above command, replace following according to your project,

- reponame : replace it with the name of remote repository or with complete url address of the repository.
- branchName: replace it with correct branch name e.g. master and add2Num here. To push all the branches use - -all keyword.

3.4 Clone repository

Once repository is pushed on the remote, then it can be cloned (downloaded) on other computers by using 'git clone' command as below,

```
git clone https://bitbucket.org/Userid/nameOfRemoteRepository
```

Replace url 'https://bitbucket.org/Userid/nameOfRemoteRepository' with your project's url.

Also, we can use '–depth 1' option, which will clone the latest branch only. This can reduce the download time,

```
git clone --depth 1 https://bitbucket.org/Userid/nameOfRemoteRepository
```

3.5 Pull changes from repository

If the repository is maintained by more than one user, then we can get the updates of others to local repository using 'fetch' and 'pull' commands as shown below,

3.5.1 git remote add

First, we need to add other users repository to our repository. Use 'git add remote' command to add the other users repository, who is contributing the same repository, as follows,

```
$ git remote add userName https://bitbucket.org/otherUserid/nameOfRemoteRepository
```

In above command, replace following according to your project,

- 'userName' : replace it with desired name, which will be used as short name of the url
- otherUserid : replace otherUserid with other user's BitBucket or Git username
- nameOfRemoteRepository : replace it with the remote repository name.

Next, use the 'git remote -v' command to see the list of contributor, which are added to our local repository,

```
$ git remote -v
origin  https://BitBucket.com/myId/nameOfRemoteRepository (fetch)
origin  https://BitBucket.com/myId/nameOfRemoteRepository (push)
userName  https://BitBucket.com/otherUserid/nameOfRemoteRepository (fetch)
userName  https://BitBucket.com/otherUserid/nameOfRemoteRepository (push)
```

Here, origin is the name of out repository, where as userName is the name of repository of other contributor. We can add more contributor using 'git remote add' command.

3.5.2 git pull

This command will download the remote copy of current local branch and **merge immediately** to local branch.

```
$ git pull userName
```

Note that, userName is the name given for the url of remote repository in 'git remote add' section. Further, pull command will download the userName's repo and merge it immediately with your repository if there is not conflict. In case of conflict, we need to manually remove the conflicts.

3.5.3 git fetch

If we **do not want merge the repository immediately**, then Fetch command should be used to get the remote copy of specific branch or all branches.

```
$ git fetch nameOfRemoteRepository
$ git fetch nameOfRemoteRepository branchName
```

Here, first command fetch all the branches, whereas second command will fetch the specific branch. To merge these changes, first go to the master branch and then run merge commands.

```
$ git checkout master
$ git merge userName/master
```

Above commands will merge the userName's master branch to our master branch.