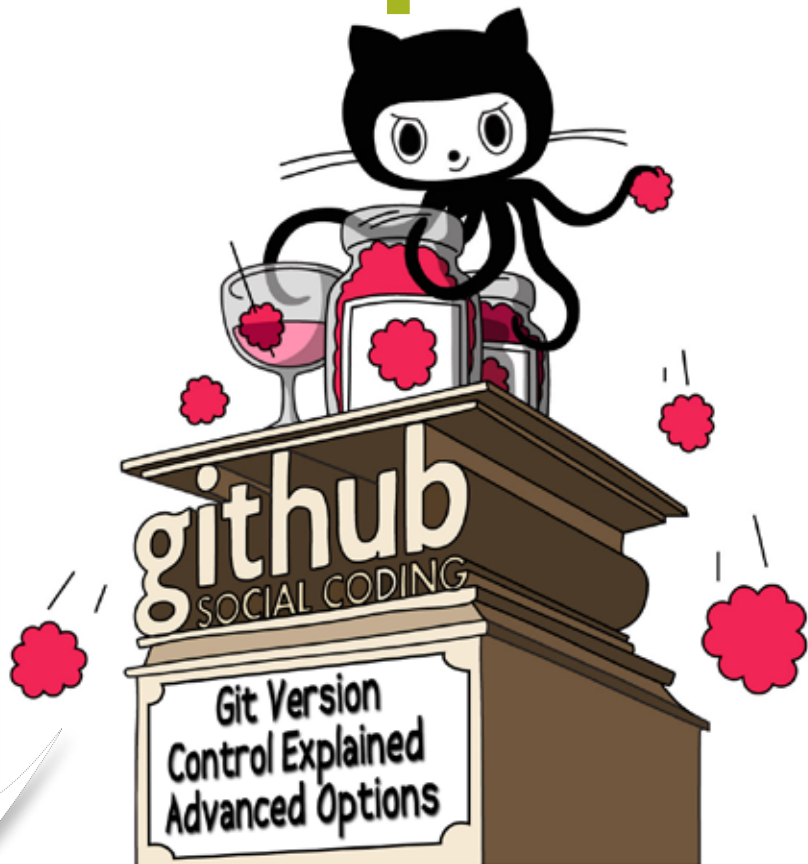


Git Version Control Explained

Advanced Options

The article published in the September issue of *LFY* covered the fundamentals of Git, including the basic data structures and commands to get started with a local repository and work with a remote repository. This article will focus on the advanced features and commands available, using an example to demonstrate how to collaboratively integrate with GitHub for coding.



Let's take a look at the advanced features and commands available in Git.

Merging branches

In Git, you create short-lived branches to develop some feature. Once the feature is complete, you merge the code to the main branch, and delete the feature branch or topic-branch. To merge a branch to the current branch, the command `git merge branch_name` is used.

Pulling changes from a remote branch

When you want to sync changes from a remote repository, use the `git pull` command—`git pull branch_name`, for example; though, if you are pulling from the origin repo, just `git pull` is enough.

This will fetch changes from the remote repository and will try to merge them to the current local repository. Thus,

`git pull` is the same as `git fetch` or `git merge`. If multiple people are committing to a repository, `git pull` is not recommended—you should use `git fetch + rebase` (see the next section).

Rebasing changes

The `merge` command tries to put the commits from the branch to be merged on top of the `HEAD` of the current local branch checkout. The `git rebase branch_name` command is a type of merge; it tries to find out the common ancestor between the current local branch and the branch you are trying to merge. It then puts the commits from the branch you are trying to merge to the current local branch by modifying the order of commits in the current local branch. For example, if the current local branch has the commits A-B-C-D, and the merge branch has the commits A-B-X-Y, then the merge command will try to turn the local branch into something like A-B-C-D-X-Y. The rebase command, however, tries to formulate the

repo as A-B-X-Y-C-D.

When multiple developers work on a single remote repository, you cannot modify the order of commits in the remote repository. Hence, you should try to formulate your local commits on top of the remote repository commits. So use `rebase` to put your local commits on top of remote repository commits and push the changes. To add your local commits on top of remote repo commits, use the following command:

```
$ git fetch
$ git rebase -i origin/master
```

This command will interactively modify the local commits (master branch) on top of commits from the origin of the remote repo's master branch.

Tracking the remote branch on a local branch

It is possible to create a local branch that tracks a remote branch with the following command:

```
git checkout -b localbranch remote_repo/newbranch
```

Fetching changes

The `git fetch` command is used to download commits from remote repositories, but it does not attempt to merge the code. An optional parameter is the repository name, as in `git fetch repo_name`.

Stashing changes

Stash is a very interesting feature. When you make local changes, you may need to switch to some other branch to check something and come back, but may not want to lose your local changes that are not yet committed. The `git stash` command preserves the current repository state for the current branch (with every local change) in its memory. The Git stash memory is a stack -- a LIFO list. To get the stashed state back, you can first use `git stash list` to view the contents of the stash memory; to apply the last stashed item back, use `git stash apply`.

Displaying objects

The `git show` command is used to display any object in the Git repo—commits, blobs, the tree, etc.

Tagging

Marking release versions of code helps you track code with respect to features or based on different criteria. Git has the built-in capability to tag different commits with tag descriptions. To list available tags, use `git tag -l`. To tag the current `HEAD`, use the following command:

```
$ git tag -a 'tagname' -m 'tag message' HEAD
```

To tag a particular commit, use `COMMIT_ID` instead of `HEAD`. To display details about a tag, just use `git show`

`tagname`. To push tags to a remote repo, use `git push --tags`.

To delete a tag from local and remote repos, use the command given below:

```
$ git tag -d tag_name // delete local tag
$ git push origin :tag_name // delete tag from remote repository
```

Creating and using patches

Commits in Git can be represented as a series of diffs or patches applied one after the other. You can create patches from commits, email them, and the recipients can apply them to their repository. To create a patch from `COMMIT_ID` to the current `HEAD`, use:

```
$ git format-patch COMMIT_ID
```

This will create a series of `.patch` files. To apply and create commits on a repo from these `.patch` files, use `git am patch_file.patch`. To apply the patch's changes to only modify files locally, but not create the commits, use `git apply patch_file.patch`.

Cherry-picking changes

Using Git, it is possible to pick an individual commit from any branch in the current repo, and try to apply it on top of the current branch `HEAD`. This feature is known as cherry-picking. To cherry-pick a commit, use: `git cherry-pick COMMIT_ID`.

Creating archives from the repository

It is possible to create a source-code archive from the Git repository. To create `file.tar` with all the latest code in the `HEAD`, run `git archive -o file.tar HEAD`.

Cleaning a repository

While working with code, you create lot of temporary files in the working directory. At some point, you may need to clean out all the files except those tracked by the Git repository. For this, use the `git clean` command.

Migrating an SVN repository to Git

Migrating all code from a Subversion repository without losing any history is super easy with the following steps. First, create a `users.txt` file in the following format, listing the Subversion users and Git username:

```
svn_user1 = Git User1 <email>
svn_user2 = Git User2 <email>
```

Next, initialise the Git repo and do a fetch, as shown below:

```
$ git svn init svn_repo_url
$ git config svn.authorsfile users.txt
$ git svn fetch
```

You just successfully migrated all your code to a Git

repository.

You can now add a remote Git repo, and sync your code to the remote repository as follows:

```
$ git remote add origin git@github.com:user/my_remote_repo.git
```

Grepping through files

Searching through source code is a big part of a programmer's life, and *grep* is used extensively for this. Git has an inbuilt *grep* command to search through files tracked by the repository—*git grep text*.

Git describe

The *git describe* command describes the current commit based on the last tagged version. Try *git describe* and you may get something like:

```
1.0-62-g4e975db
```

This auto description can be interpreted as follows:

- 1.0 – The recent tag
- 62 – The *HEAD* is the 62nd commit after the tagged commit 1.0
- g4e975db – The first eight characters of the *HEAD* commit ID.

Integrating with Github

Github is a great Git repository hosting website, which has lots of facilities to collaborate with many developers on a project, including features that help to code, integrate, merge and perform code reviews. Let us go through an example Git workflow along with GitHub integration.

Example workflow

The following is the most common Git workflow that everyone uses daily.

```
$ mkdir myproject # Decided to start a new project
$ git init #Add project as a git repo
# Created few files README, main.c. Now I wanted to add those files to repo
$ git add README main.c
$ git commit -m "Initial commit"
$ git log # View the log for the commit
#Later I want to change the commit message to a more appropriate one.
$ git commit --amend
$ git diff # I modified main.c. I would like to see the diff
# Now I would like to add the changes into two commits
$ git add -p main.c #Interactively select few code chunks and add
# Commit the selected changes
$ git commit # vim opens up and we add a commit message such as:
Fix bug-0234 - limit the size of the array
```

```
<Blank line>
Description about bug-0234 and fix
#Add rest of the changes to another commit
$ git add main.c
$ git commit
$ git log
# Now to sync my repo to a remote (Github) repository I created.
# Add a remote repo to current repo
git remote add origin https://github.com/t3rmin4l/projectname.git
$ git push origin master # Push master branch (Default) to remote repo
$ git branch -a # List branches
```

Now, if you want to work on a feature that takes a lot of time, while some other development goes on in parallel, create a branch for the feature development as shown below:

```
$ git branch feature-x
$ git checkout feature-x
```

To add some changes related to *feature-x* (such as, adding a *TODO.txt*):

```
$ git add TODO.txt
$ git commit -m "Added TODO for feature-x"
```

To go back to the master branch and make a few commits (like adding and changing few files), issue the following commands:

```
$ git add core.c
$ git commit -m "Move core functions to core.c"
$ git add main.c
$ git commit -m "Refactor main.c"
```

If you want to work on *feature-x*, do a *git checkout feature-x*, then add a bunch of files to build the feature, and then use the following code:

```
$ git add feature.c main.c core.c
$ git commit (message as follows)
Add feature-x - logger module for xxx
<Blank line>
More description about the feature
```

Switch back to the master branch to make some more changes:

```
$ git checkout master
$ git add README (commit some changes from README)
$ git commit -m "Improve README"
```

If you then want to merge the *feature-x* I developed into the main branch, use the *git merge feature-x*. However, some

merge conflicts will occur. Fix them by editing the listed files, and commit them as follows:

```
$ git add core.c
$ git commit -m "Merge feature-x into master"
```


Now, to sync the code with the remote repository, use `git push origin master`. Then, to delete the feature branch that is now merged into main, use `git branch -D feature-x`. By this time, another developer may also have joined this project and committed to the remote Git repository; so to sync back the remote repo to your repo in order to get his changes from there applied to your local repo, use `git pull`.

After making some commits in the local branch, you may find that `git push origin master` failed, because the other developer added a few extra commits to the remote repo. So, to rebase your code on top of the latest commits from the remote repo, use the following commands:

```
$ git fetch
$ git rebase origin/master
```

This lists some conflicts. Resolve those commits by editing the listed files manually, and continue the rebase operation with `git rebase --continue`. After a successful rebase, try to push to the remote repo again, by using `git push origin master` once again, which will now succeed.

In the meantime, if you have created another branch, `branch-x`, which you may want to push to the remote repo, use `git push origin branch-x`.

Git is a unique and amazing version control system. In this article, I have gone through most of the essential commands. Using Git for hobby projects is the best method to master good practices. You can learn more of Git by joining [Github.com](https://github.com) and exploring it through a social coding experience. You can use the reference learn.github.com. Happy hacking, till we meet again. 

By: Sarath Lakshman

The author is a Kerala-based hacktivist of Free and Open Source Software. He loves working on the GNU/Linux environment and contributes to the Pardus Linux distro project. He has recently authored the *Linux Shell Scripting Cookbook*, which gives insights on shell scripting through 119 recipes. He can be reached via his website <http://www.sarathlakshman.com>.