The Git version control system

Jeetaditya Chatterjee

June 10, 2021

What is version control?

version control is a way to manage and back up source code It is the industry standard practice to keep code under some kind of version control and the most widly used VCS is called git. you have probably heard of it from services such as github and gitlab.

- breaking code changes into revisions or "commits" each identifyable using a uniqe hash.
- allowing for people to non destructivly make copys of the code and merge them in when developed
- facilitating the ability to checkout and roll back commits
- push the source code to a remote for backups
- **Some features of Version control:**
- breaking code changes into revisions or "commits" each identifiable using a unique hash.
- allowing for people to non destructively make copies of the code and merge them in when developed.
- facilitating the ability to checkout and roll back commits thorughout gits history
- push the source code to a remote for backups and easy sharing of the source code.

Why use Version control? (git)

other than it being the industry standard it also has a lot of benefits such as

- Breaking up code revisions into smaller chunks called "commits" which makes it easier to see how the code base changed over time
- Allowing you to roll back commits for whatever reason git also provides tools to find the last good commit called bisecting
- Allowing you to change your code base without affecting a master copy of your code Meaning you can add features and merge them in when they are ready while also shipping a stable version
- making it easier to collaborate and scale your code base to include more people
- as well as other benefits
- Breaking up code revisions into smaller chunks called "commits"
- Allowing you to roll back commits for whatever reason
- Allowing you to change your code base without affecting a master copy of your
- Making it easier to collaborate and scale your code base to include more people

Some concepts

Starting off

init

Lets say you need to start a project you can just do a git init to initialise an empty git repository

```
$ git init
```

clone

If you need to work on some other project or reclone one you have already started then you can use the clone command

```
$ git clone https://some.website/repo.git
# eg
$ git clone https://github.com/jeetelongname/example.git # you can actually clone this
```

status

git status tells you what is happening in your repository. it will tell you what is staged what needs commiting and so on

```
git status
```

Staging

Staging is the git term for getting files ready for commits. When you need to add new changes you need to add the file to the stage before commiting it. This may seem tedios but means you don't need to commit all of your changes by one go and commit based on the task you are completing.

add

To add a file / your changes to a file you use the git add command you specify the file you want to add.

too add all of your changes you can use the . symbol (which stands for the current directory as we discussed from our last lesson)

```
git add file

git add .
```

restore

To remove a file you can use the restore command. provide it with a file and it will unstage the file. to unstage all files you use dot like before. be careful as without the –staged flag you would delete your changes which is not fun. You can set an alias for it tho which would probably make your life easier to make a git alias

```
git restore --staged file # removes file from the stage

git restore --staged . # removes all staged changes

git reset # also works

git config --global alias.unstage 'restore --staged'

git unstage file
```

Commits

We have discussed the precursor to commiting so now we need to actually commit to it

A git commit is a collection of changes that will be added to your git history. commits represent the backbone of git and its important you make your commits small and to the point. don't try and stuff too many features into one commit as you lose a lot of the benefits of git. (as a rule of thumb try and keep each commit down to one fix or feature. eg a small bug fix or the addition of a function)

commit

To create a commit you call the `commit` command. this will open up an editor for you to then type in a commit message. I won't go deep into commit etiquette but I recommend you search conventional commits as it provides a good framework for commit messages

There are 2 flags that are useful but not recommended for proper projects the -a flag which commits all changes in the current directory and the -m flag which will allow you to type a message inline without using an editor.

```
git commit # opens an editor where you type a message

git commit -a # commit all changes
git commit -m "commit message provided here"
```

Branches

Branching is another really powerful feature of git. It allows you to make sweeping changes to your code without damaging the master copy of your code.

Branches are cheap to make (taking up very little space) meaning you have no reason to use them!

The main use of branches is to separate stable code from new features or bug fixes. This allows you to change the code to your hearts content without damaging your main copy

branch

creating branches is quite easy. you just call the branch command and it will create a branch starting at the current branch. You can speify a different branch by providing it

to delete a branch you add the -d flag

If there are unmerged changes and the branch is not backed up you will need to force git by using the -D flag

```
git branch <branch_name>

git branch <branch_name> <base_branch_name>


git branch -d <branch_name>

git branch -D <branch_name>
```

checkout

We have created branches but now we need to use them so we use the checkout command

git actually has a shortcut to create a new branch and switch to it. by adding the -b flag to the checkout command you can create a new branch there and then

```
git checkout <branch_name> # switch to that branch

git checkout -b <new_branch_name>
```

Merging

Now that we have these branches we need to actually do somthing with them.. we discussed deleting them but thats not that useful. We need a way to merge them and update them as time goes on

merge

merging takes the commits of the provided beach and *merges* them into the current branch by making a merge commit. this tells git what commits have been merged into the current branch. As its a commit if you are not happy with the merge you can rollback the commit like any other. This is also known as non destructive merging

the problem here is that there will be a commit everytime you merge the branch which can make the history of the branch messy and not that great. that is where the next kind of merging comes into play

```
git checkout master
git merge feature # merge feature into master

git merge master feature # merge feature into master
```

rebase

rebasing rewrites the history of the current branch to incorporate the changes of the merging branch. this changes the history of the branch which is pretty dangerous that being said it also makes the code history much more readable and makes the project history linear. there are no forks to contend with making it much easier to follow a projects history.

This comes at the cost of safety you are rewriting your history which every time travel show I have watched has said is a really dangerous thing to do. You also lose some context provided by the merge commit

as for where to use which. I reccomend you rebase your main branch onto your feature branches and merge your feature branches into your main. This is what I see happen a lot but this is not a hard and fast rule

```
git checkout feature
git rebase master # rebase master onto feature

git rebase feature master # samething but one line
```

Remotes

We have reached another conundrum all of this code is local. We need a way to get it out into the world. We could put all of this code in a drop box folder and share that but I think you know that i am going to show you how to use git to do that

remote

A remote is an online location for your code. people upload there code to github or gitlab some people even host there own server. but all are valid remotes

to add one you call the remote add command and provide it with a name and the url

to change the url call the set-url command

and then rename and remove are self explanatory

```
git remote add <remote_name> https://your.url.here/repo.git

git remote add origin https://github.com/jeetelongname/example.git

git remote set-url origin https://git.sr.ht/~jeetelongnamr/example.git # not real

git remote rename orign upstream

git remote remove upstream
```

push

To send your changes to your new fangled remote you use the push command. it takes the argument of the remote and the branch to push. when pushing the branch for the first time you should add the -u flag which tracks the branch

You may need to also overwirte the remote for some reason this is risky as you could lose work other people push.

```
git push origin master
git push -u origin devel # pusing for the first time
git push origin master --force # overwrite the remote
```

fetch

git fetch will download the files from a remote without doing anything with them. This allows you to look at what other people are doing without affecting your local copy. you can then merge it into your local copy later if you wish.

note when you check out `some-branch` you will be in a detached head state which means that you can edit all of this and it will not affect your history

```
git fetch origin # fetch all of the branches named origin
git fetch origin some-branch

git checkout some-branch
```

pull

pull is used to update your local branch with the changes of upstream. this is used a lot when working in a group and you need to get the changes from upstream. It fetches from upstream and then merges it into your code. If you want to rebase instead of merge you can use -r.

```
git pull origin master # merges
git pull -r origin master # rebases
```

Reverting

Oh no we made a mistake in one of our commits and now we have angered all of the customers. We need to get back to a working commit. first we need to find the commit and then revert back to it

log

the log command shows you a timeline of your code on the commit level you can then look through and get the unique hash for you to then pass onto the next command

the -p flag provides a "diff" which shows you the changes in each commit

revert

revert takes a hash (or the amount of commits you want to go back from) and will make a new commit with those changes applied. essentially rolling back to that commit.

also note you don't need to paste in the entire commit hash you can get away with the first 5 terms and git will figure out the rest

If that is still too much work you can then use some special syntax to roll back a certain amount of commits from the current one or the HEAD commit All its saying is roll back one commit behind head. we can put any number there but its not really a good solution if you need to go back to a specific commit

```
git revert b4e73eef1e7a1620... # full hash works

git revert b4e73 # also works

git revert HEAD~1 # roll back one commit
```

What do you do now?

Well you need to use git. I recommend you try and use git with any and every one of your projects. I actually used git for my NEA and it helped keep a record of what I have done and how long it took.

And this is not the only way to use git. most text editors worth there salt have some sort of git integration and there are usually 3rd party front ends that can make using git much nicer and faster.

Any Questions?