Photo by Gianmarco Boscaro on Unsplash

Git Repository Transfer Keeping All History

How to replicate your Git repo and keep all previous commits, branches, and tags

Nassos Michas Follow
Dec 24, 2019 · 4 min read ★

Git is nowadays the *de facto* standard in version control, distributed or not. It comes with a vast array of powerful features, some of them go unnoticed by the majority of its casual users. One such feature is Git's capability to connect your local repository with multiple remote repositories, as well as removing and adding remote locations as you wish.

On this post, we'll use the feature of adding and removing remote repositories to effectively transfer one repository to another while keeping all commit history, branches, and tags intact.

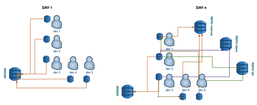
. . .

Git remotes

In contrast to older version control tools, such as CVS and SVN, where all code was centralised in a single repository, Git allows code to co-exist in an unlimited number of remote repositories.

In fact, the mere notion of the original, central, or "origin", repository in Git is just a convention. It just *happens* to be the repository most users started synchronising with when they did that initial `git clone`.

As the project progresses repository location and preferences may change, so what was once, in the beginning, the *origin* may start to seem irrelevant:



Remotes can be introduced at will in Git (image by author)

. . .

Transferring methods

We've established so far that you can have as many remotes as you wish and that their name doesn't have any significant role in Git's ecosystem. So, how can we leverage this feature of Git to transfer files from one remote to another?

Losing all history (developer-god mode)

The quick and dirty way is to just clone the new, empty repository, copy/paste the files from the source repository, and commit/push. Although that approach might save you a few minutes and doesn't require any other than the standard Git commands you already know, it comes with a severe drawback.

Since you are the one committing all the files to the new repository, you are now becoming the sole author of every single file of the source repository. Any previous commit history, kept in the source repository, is now lost forever and the whole project looks like it has been created by a single person (you).

Developer-god syndrome satisfied; original authors pissed off.

Keeping history

Copying resources from one repository to another while keeping commit history intact not only acknowledges everybody else's original work but might come with useful perks. For example, you can search previous commit messages to find an explication that sheds light into an issue you're investigating; if you realise a specific developer is prone to the same type of mistakes, you can easily filter out his/her commits to re-check them, and many more.

. . .

Transferring commands

If you search for how to transfer a Git repository you'll end up with several different suggestions, ranging from simple sequences of Git commands to pure Git black magic.

The approach suggested next may not necessarily be "the best out there", however, it has been tested on repositories with thousands of commits and is honouring two basic principles: 1/ Keeping the complete commit history including all branches and tags, and 2/ not interfering with the original repository.

So, let's start with the commands.

① Start by creating a mirrored clone of your old repository

```
git clone --mirror old-repo-url new-repo
```

Replace `old-repo-url` with the Git URL of your old repo and give an appropriate name to the `new-repo` folder on which it will be cloned.

② Remove the remote reference to the original/old repository

```
cd new-repo
git remote remove origin
```

You may get a warning if your original repository contains branches but you can ignore it. Technically speaking, this step is not necessary since as discussed above you can have multiple remotes, however, it makes future commands a bit simpler to type and completely detaches you from the original repository.

③ Add the remote reference for the new repository

```
git remote add origin new-repo-url
```

Replace `new-repo-url` with the Git URL of your new repository. It goes without saying that you should have had the new repo created beforehand in your favourite Git management system.

④ Push everything to the new repository

```
git push --all
git push --tags
```

⑤ Clone the new repository

```
cd ..
rm -rf new-repo
git clone new-repo-url new-repo
```

Replace `new-repo-url` and `new-repo` with your new repository.

This last step is necessary because the clone performed in step (1) was a mirrored clone that can't be used as such for further commits. Let me know in the comments if you're aware of a way to convert it, effectively bypassing this re-cloning step.

. . .

Conclusion

Although Git's branches and tags allow you to effectively use a repository for multiple different purposes, sometimes you just want to start in a new repository. On this post, we saw how to transfer the content of a Git repository to another new repository, while moving all commit history, branches, and tags of the original repository too.

Git | Programming | Software Development | Software | DevOps

WRITTEN BY
Nassos Michas
Software engineer | Cert. Scrum master | Cert. Professional for Requirements Engineering | CTO at European Dynamics

Follow

ITNEXT
ITNEXT is a platform for IT developers & software engineers to share knowledge, connect, collaborate, learn and experience next-gen technologies.

Follow

More From Medium

WebRTC for beginners; How it all works from the outside!
Karthikeyan S in ITNEXT

Breaking down and fixing Kubernetes
Andrei Kvapil in ITNEXT

Advanced Git Features You Didn't Know You Needed
Martin Heinz in ITNEXT

Yes, here are 4 ways to handle SEO with Vue (even without Node SSR)
@maisonfutari in ITNEXT

Amazon EKS Upgrade Journey From 1.18 to 1.19
Marcin Cuber in ITNEXT

How to structure my VueJs project
Manu Ustenko in ITNEXT

Writing SQL in C# or When You should not use ORM
Dmitriy Tikhonov in ITNEXT

How to setup Kafka cluster for 15K events per second on AWS using Docker
Abhinav Dhasmana in ITNEXT