PRACTICAL PORTS

Enhance Your Git Experience

BY BENEDICT REUSCHLING

Version control has been around for a long time. Having a way to keep a certain version of a file in a state where it is retrieved is the start of our journey. After all, that's what backup software is for. Some people use version control systems (VCS) that way, but that only scratches the surface. The valuable parts of VCS is to be able to review what has changed in the file over its history. When using it with other people to keep important project files in sync with everyone else, features like blaming, diffing, branching and merging become even more relevant. These features make collaboration on anything from text, source code, configuration files, and anything worth sharing within a group possible in the first place. Most big projects that have a large set of those files under version control would simply be too tedious to maintain without sophisticated tooling around it. Version control systems have come and gone, both from commercial and open source vendors. Many of them implement a basic set of functionality that has been established by the people using it, and introducing a whole new set of terminology and complicated tooling is often seen as a hindrance to adoption.

Each code change in the long history of the FreeBSD project has had a descriptive text attached to it.

Each code change in the long history of the FreeBSD project has had a descriptive text, called the commit message, attached to it. What may seem like a nuisance for some is actually helpful when looking for problems that are rooted in the past. Why was a change made 15 years ago? Who made that change, what other files were touched by the same commit, and what are the actual lines of code that were affected? These are all questions that not only the diffs that the VCS produces can relate, but also the commit message is a valuable help in understanding what is going on in the code today. This often bridges the past to the future since Unix systems have been around for a while and are running on hardware that people who committed the changes in the past could not possibly conceive.

With FreeBSD being so old, preserving that rich history is important. Particularly when there is a need to switch to a different version control system. Over time, FreeBSD has used CVS, then migrated to Subversion, and now uses git. (I'm fairly sure there was also RCS involved in the early days.) Each time such a migration took place, one primary goal was to preserve every single change made, along with the commit message.

Switching version control systems is sometimes necessary because a vendor goes out of business or something better comes along that people find more appealing than the current solution. The best is the enemy of the good. Git has become the de-facto standard for version control these days, even if it has some rough edges and a bit of a learning curve. In my view, both as a user/consumer of the FreeBSD project as well as a committer, it took a while to get familiar again and to make git a tool that works for me and not the other way around. I may still not totally grasp all of the inner workings, but luckily, I do not have to. Developers working on the src and ports trees also use a lot more of the features like branching, merging, tagging, cherry-picking, rebasing, and others that are not required in the documentation repository. Still, sometimes a typo fix in a ports description or a man page requires people like me to also get at least a basic understanding of these concepts to do the right thing.

It's also a question of how often something is used. If I use a feature only twice per year, I may need to look it up again because the last time I did was a while ago. The basics of cloning a fresh tree, pulling updates, making changes, committing, and pushing them are all too familiar once you've done it regularly. I compare it to basic vi usage. I can open a document, make changes, save, and exit just fine. But the other features that a powerful editor like vi and friends provides may stay hidden from me forever. Same with git: it has a lot more under the hood which users do not know about or even need. More often than not though, it makes life easier—even for the basic cases—to have a bit of knowledge and configuration around these advanced features. And this holds not only for a developer, but also for a user.

If I use a feature only twice per year, I may need to look it up again.

FreeBSD 14 deprecated the portsnap utility which a lot of people used to get a new or updated version of the ports tree. Instruction in the handbook and other places now directs people to check out the ports tree directly from git. The same is true when a user needs a copy of the src tree, because they need to recompile the kernel module for VirtualBox. These are all good use cases, but there is a catch: both the src and ports trees take a long time to clone because of the long and rich history described above. Let's see if we can add some configuration and tools that speeds up the process. We'll also learn about other neat features that git provides, both for Joe random user as well as Jane developer.

Faster Cloning

First, install git on FreeBSD by running **`pkg install git`**. Since we do not have a ports tree yet and there is no more portsnap, that can turn into a chicken and egg problem on a system that does not have direct access to the FreeBSD mirrors. A poudriere machine in your local network may be the solution here, or download the file on a different machine, copy over the binary, and use **`pkg install ./<packagename>`** to install it.

Note that the git port itself contains a good amount of configurable options. Since we're starting out, we ignore that for now and revisit some of these when we feel more experienced. If you discover something useful, then consider blogging about it or write an article for the *FreeBSD Journal*. After all, why do I have to do all the writing work?

Once git is available, we can return to our use case above: downloading a copy of the ports tree. *The FreeBSD Handbook* Chapter 4 tells us that we can either get the HEAD branch (the latest and greatest) or be a bit less on the bleeding edge and get the quarterly branch. Whichever we chose, we get presented with the all too familiar **`git clone`** command. That's all fine and good, but it takes a long time to finish downloading all those files and directories. Let's look at the quarterly branch at the time of writing this article. I'll use the time(1) command to measure the download time.

```
$ time git clone https://git.FreeBSD.org/ports.git -b 2024Q1 /usr/ports
Cloning into '/usr/ports'...
remote: Enumerating objects: 6125935, done.
remote: Counting objects: 100% (960/960), done.
remote: Compressing objects: 100% (142/142), done.
Receiving objects: 100% (6125935/6125935), 1.20 GiB | 36.28 MiB/s, done.
remote: Total 6125935 (delta 925), reused 833 (delta 818), pack-reused 6124975
Resolving deltas: 100% (3700108/3700108), done.
Updating files: 100% (158490/158490), done.
git clone https://git.FreeBSD.org/ports.git /usr/ports
0.00s user 0.03s system 0% cpu 3:34.48 total
```

Bandwidth aside, this 3:34 is too long for me. We can do better than this. Since we do not need all the history and just the latest version of files, a shallow clone using **`--depth-1`** is much faster.

```
time git clone --depth=1  https://git.FreeBSD.org/ports.git /usr/ports
Cloning into '/usr/ports'...
remote: Enumerating objects: 194509, done.
remote: Counting objects: 100% (194509/194509), done.
remote: Compressing objects: 100% (182218/182218), done.
remote: Total 194509 (delta 11904), reused 120301 (delta 5787), pack-reused 0
Receiving objects: 100% (194509/194509), 85.40 MiB | 10.48 MiB/s, done.
Resolving deltas: 100% (11904/11904), done.
Updating files: 100% (158490/158490), done.
git clone --depth=1 https://git.FreeBSD.org/ports.git /usr/ports
0.01s user 0.01s system 0% cpu 28.709 total
```

Much faster indeed (29s) and I get exactly what I want. What if I need the full history because I'm working on a bug? Then I can use a filter function to first get the whole commit history, but not the history. The latter may come as a separate step. I'm looking to reduce the time to download, so let's try this:

```
time git clone --filter=blob:none https://git.FreeBSD.org/ports.git /usr/ports
Cloning into '/usr/ports'...
remote: Enumerating objects: 3706789, done.
remote: Counting objects: 100% (794/794), done.
remote: Compressing objects: 100% (82/82), done.
remote: Total 3706789 (delta 771), reused 721 (delta 712), pack-reused 3705995
Receiving objects: 100% (3706789/3706789), 704.87 MiB | 48.79 MiB/s, done.
Resolving deltas: 100% (2043361/2043361), done.
remote: Enumerating objects: 152073, done.
remote: Counting objects: 100% (63494/63494), done.
remote: Compressing objects: 100% (61224/61224), done.
```

```
remote: Total 152073 (delta 7810), reused 2276 (delta 2270), pack-reused 88579
Receiving objects: 100% (152073/152073), 78.98 MiB | 10.93 MiB/s, done.
Resolving deltas: 100% (11301/11301), done.
Updating files: 100% (158490/158490), done.
git clone --filter=blob:none https://git.FreeBSD.org/ports.git /usr/ports
 0.00s user 0.03s system 0% cpu 1:51.29 total
```

Git divided the work into two parts: first, all blobs (think files here) get filtered out and fetches history at first. In the second step, the blobs followed. This was faster than a full clone, but slower than the shallow copy. There was also a difference in the retrieved sizes, which contributed to the speedup. In the regular clone, we received 1.20 GB. The two-step process of the blobless clone let git receive 704.87 MB of history followed by 78.98 MB. This benefit comes with a drawback though: when I have found the bug and I want to know when this was introduced, the **`git blame`** operation needs to fetch those revisions from the server first. If I'm on the road without network access, I'm out of luck. The full clone could give me the information, as it has all the history already retrieved. Again, for non-developers interested in getting the files themselves, this does not matter much and the benefit is a better download time.

Scaling Up

Imagine you were working on the man pages, which reside in the src tree. Downloading the whole kernel, userland, tools, and everything in between is a lot for the initial clone. What if you only occasionally work on those man pages? Surely there are changes made by others, which we need to be aware of. Wouldn't it be nice if our system would fetch those changes for us, so that our local copy does not drift too far away from the top of the tree? The scalar tool that is part of git solves it: fast downloads of a big repository and retrieving changes from upstream in regular intervals. This puts the local clone into maintenance mode, which is a fancy word for this functionality. Here is how to use it: replace **`git`** with **`scalar`**, the rest of the command is identical.

```
time scalar clone https://git.FreeBSD.org/src.git /usr/src
Initialized empty Git repository in /usr/src/src/src/.git/
remote: Enumerating objects: 2386494, done.
remote: Counting objects: 100% (258756/258756), done.
remote: Compressing objects: 100% (16493/16493), done.
remote: Total 2386494 (delta 253705), reused 244654 (delta 242263), pack-reused 2127738
warning: fetch normally indicates which branches had a forced update,
but that check has been disabled; to re-enable, use '--show-forced-updates'
flag or run 'git config fetch.showForcedUpdates true'
warning: fetch normally indicates which branches had a forced update,
but that check has been disabled; to re-enable, use '--show-forced-updates'
flag or run 'git config fetch.showForcedUpdates true'
remote: Enumerating objects: 20, done.
remote: Counting objects: 100% (17/17), done.
remote: Compressing objects: 100% (17/17), done.
remote: Total 20 (delta 0), reused 0 (delta 0), pack-reused 3
Receiving objects: 100% (20/20), 196.11 KiB | 16.34 MiB/s, done.
warning: fetch normally indicates which branches had a forced update,
but that check has been disabled; to re-enable, use '--show-forced-updates'
flag or run 'git config fetch.showForcedUpdates true'
```

```
branch 'main' set up to track 'origin/main'.
Switched to a new branch 'main'
Your branch is up to date with 'origin/main'.
crontab: no crontab for root
scalar clone https://git.FreeBSD.org/src.git
0.01s user 0.00s system 0% cpu 31.971 total
```

Ignore those warnings for now, the process finishes nonetheless. There is something about crontab(1) here, responsible for fetching updates in regular intervals. To convert an existing repository to use scalar, no need to clone it again: run **`scalar register`** in the root of your repository and it will convert the local copy to use it. Neat! The scalar command will set up a crontab entry. If you do not have a user-specific crontab (like I have here for the root user), then run **`crontab -e`** to set it up. If all went well, git adds an entry for scalar to run:

```
# BEGIN GIT MAINTENANCE SCHEDULE
# The following schedule was created by Git
# Any edits made in this region might be
# replaced in the future by a Git command.

29 1-23 * * * "/usr/local/libexec/git-core/git" --exec-path="/usr/local/libexec/git-
core" for-each-repo --config=maintenance.repo maintenance run --schedule=hourly
29 0 * * 1-6 "/usr/local/libexec/git-core/git" --exec-path="/usr/local/libexec/git-core"
for-each-repo --config=maintenance.repo maintenance run --schedule=daily
29 0 * * 0 "/usr/local/libexec/git-core/git" --exec-path="/usr/local/libexec/git-core"
for-each-repo --config=maintenance.repo maintenance run --schedule=weekly
# END GIT MAINTENANCE SCHEDULE
```

Adjust those entries to your own needs or leave them as they are. If the machine has a proper mail setup, you'll receive messages containing the fetched revisions when the cronjobs have run. Another thing that **`scalar clone`** and the associated maintenance jobs do is add an entry to your git configuration file, aptly named **`.gitconfig`**.

```
[scalar]
repo = /usr/src
[maintenance]
repo = /usr/src
```

This brings us right into git's configuration file.

Create Your (commit) History Everyday

Chances are that you are working on multiple repositories over time: one for work, another for a private project, and contributing to your favorite open source project. The configuration for those cloned repositories may be different. For example, you may use your corporate email to identify yourself in your commits, which may not be appropriate or even allowed when committing to a private project. As such, we can have a project-specific **`.git/config`** as part of the repo and a global one that applies to any and all repositories you're working on.

The global **`.gitconfig`** is in your home directory. You can either directly edit that file (if you know what you are doing) or use git to manage the contents of the file and set proper values. The latter uses this syntax:

```
git config --global NAME VALUE
```

For example, to register my name for commits, I run:

```
git config --global user.name Benedict Reuschling
```

This results in an entry like this in **.gitconfig**:

```
[user]
        name = Benedict Reuschling
```

You can see that there are categories in brackets for entries like user (email is under there and you better set it, too). Others are **commit**, **diff**, or **branch**.

Changing Commit Behavior

Torches and pitchforks aside, I do not like to use nano to write my commit messages. To define your own editor, execute this command:

```
git config --global core.editor nvim
```

You almost always want to set this as a global option. Having this in a project **.git/config** and committing it will cause productivity to fall sharply as your other contributors will start another holy editor war resulting in a lot of changes in the repo with nothing but trying to change it to their own personal favorite. "Well done", is what you will remember as the last words of your sarcastic boss as he closes the company door behind you forever on the same day.

There are other configuration settings that (more positively) affect your commit experience. There are too many to list here and the defaults are fine. Fiddling with some options can change your git experience somewhat and may remove some personal annoyances (see below). See git-config(1) for details.

How about being a bit more sophisticated and defining aliases for common but tedious to type commands? That's were the alias section comes in handy. I have these defined:

```
[alias]
    last = last -1 HEAD
    lg = log --graph --all --pretty=format:'%Cred%h -%C(yellow)%d%Creset %s %Cgreen(%ci)
%C(bold blue)<%an (%ae)>'
```

Similarly, when looking at the log, I'd like to see at least these fields: commit, the author, the date the author made the change, plus the commit and its date. This is achieved by looking at git-log(1) and figuring out that the option is called **fuller**:

```
git config --global format.pretty fuller
```

In my repository, I can run **git last** to run the equivalent of **git last -1 HEAD**. Getting totally fancy with colors and all, I like the command **git lg** even more when thinking about how few keystrokes it requires now. Try it out for yourself and thank me later.

When I do the actual commit, I want to see what gets committed. Git displays the diff between the head revision and my own changes below the text area for the commit message with this setting:

```
git config --global commit.verbose true
```

Your Signature, Please

Speaking of commits, why don't you sign your commits? "Well, the GPG/PGP setup is too complicated" may be an answer. There is a solution for that: use SSH instead. On services such as GitHub or your corporate (or private) GitLab instance, you have already uploaded a public key to pull repositories over SSH. Signing your commits with the same key gives your changes some extra credit. This is often honored with a "signed" icon or label next to the commit on those platforms. The setup is so easy, I wonder why this is not the default by now. Here's how:

```
git config --global gpg.format ssh
git config --global user.signingKey 'ssh-ed25519 AAAAC3(...)34rve user@host'
```

It's debatable if you want to use the same SSH key everywhere. If not, remove the **`--global`** option from the last line above and make that change for each repo with its own key.

Commits can now go like this:

```
git commit -S
```

To always sign, make it the default:

```
git config --global commit.gpgsign true
```

But what is this? When running **`git show --show-signature`** does not show our signature, but displays an error message instead. Not cool! Good for us, the message also tells us what to change: **`gpg.ssh.allowedSignersFile`** is the option we need to change.

Git complains because SSH does not build a web of trust that signs keys by others. Instead, we need to tell git which keys we are trusting. A separate file contains all trusted SSH signatures. Since we are orderly people, let's put this file in **`~/.config/git/allowed_signers`** (create the paths if they don't exist by now).

The content of **`allowed_signers`** is as follows:

```
email ssh-ed25519 ssh_public_key comment
```

Keen eyes will recognize it as the same format that ssh-keygen(1) uses. We need to at least trust our own SSH key, so put it there. Repeat this for all the other people in your circle who contribute to the repo and sign their commits, too. To teach git about this file, add yet another configuration option (the one the error message complained about earlier):

```
git config --global gpg.ssh.allowedSignersFile "~/.config/git/allowed_signers"
```

Retry the **`git show --show-signature`** command (and create an alias for it) to see the error message replaced by the git signature.

Fixing Minor Annoyances

To update your local copy, it's suggested that you run **`git pull --ff-only`**, which is the default behavior. If you keep forgetting to add the parameter, then set it as the default pull behavior like this:

```
git config --global pull.ff only
```

Of course, you could create an alias for it. This is one of those "fire-and-forget" settings you do not need to revisit in the future.

When looking at diffs, I always wondered why git uses a/ and b/ to distinguish the files from each other. I do not need those, the filenames speak for themselves. I found that disabling this behavior is possible with this option:

```
git config --global diff.noprefix true
```

Speaking of diffs, I would like to see at least 5 lines of context around my changes. That is a personal preference, but anyone can set it to their liking with this option:

```
git config --global diff.context 5
```

Working on an international project like FreeBSD has taught me that there are multiple ways to write a date. The default display that git uses is `Fri Mar 01 12:34:56 2024`. All fine with that, but I'm used to the following way: `2024-03-01 12:34:56`. This option sets it exactly how I like it:

```
git config --global log.date iso
```

Another thing that I found odd was the order in which git lists branches when running `git branch`. I would like to have the branch with the most recent commit at the top and not some other (random?) order. To change this, my `.gitconfig` contains this:

```
git config --global branch.sort -committerdate
```

Now I see exactly which branch received the most recent changes. Time to merge!

Conclusion

My configuration will probably grow over time as I discover other useful options. Git is flexible in its configuration. The defaults are fine for most people and changes are easy to make. This article should get you started writing your own config and ideally reduce some of the teeth grinding involved when working with git.

References:

https://blog.gitbutler.com/git-tips-and-tricks/
https://jvns.ca/blog/2024/02/16/popular-git-config-options/
https://blog.dbrgn.ch/2021/11/16/git-ssh-signatures/

BENEDICT REUSCHLING is a documentation committer in the FreeBSD project and member of the documentation engineering team. In the past, he served on the FreeBSD core team for two terms. He administers a big data cluster at the University of Applied Sciences, Darmstadt, Germany. He's also teaching a course "Unix for Developers" for undergraduates. Benedict is one of the hosts of the weekly bsdnow.tv podcast.