FreeBSD mini-Git Primer

BY WARNER LOSH

The FreeBSD project has begun its transition from Subversion to Git. This move has been over a year in the planning and reprensets the next step in FreeBSD's continuing efforts to improve its workflow. The project hopes that the larger ecosystem for Git will help it improve its continuous integration (CI) efforts; make it easier to submit patches; and generally increase the quality of the project.

This article is aimed at the FreeBSD user who downloads sources, has local changes and sometimes contributes them back to the project. It will provide an introduction to the FreeBSD's use of Git for an audience already generally familiar with the basics of Git. Where possible, a pointer to a more in-depth treatment of Git will be provided. There are many primers for Git on the web, but the Git Book provides one of the better treatments.

Keeping Current With FreeBSD src Tree

First step: cloning a tree. This downloads the entire tree. There are two ways to download. Most people will want to do a deep clone of the repo. However, there are times that you may wish to do a shallow clone.

Branch Names

The branch names in the new Git repo are similar to the old names. For the stable branches, they are stable/X where X is the major release (like 11 or 12). The main branch in the new repo is `main`. The main branch in the old GitHub mirror is `master`. Both reflect the defaults of Git at the time they were created. The main/master branch is the default branch if you omit the `-b branch` or `--branch branch` options below.

Repositories

At the moment, there are two repositories. The hashes are different between them. The old GitHub repo is similar to the new cgit repo. However, there are a large number of mistakes in the GitHub repo that required us to regenerate the export when we migrated to having a Git repo be the source of truth for the project.

The GitHub repo is at https://github.com/freebsd/freebsd.git The new production repo is at either https://git.freebsd.org/src.git or ssh://anonssh@git.freebsd.org/src.git depending on which transport you wish to use. These will be $URL in the commands below.

Note: The project doesn't use submodules as they are a poor fit for our workflows and development model. How we track changes in third-party applications is discussed elsewhere and generally of little concern to the casual user.

Deep Clone

A deep clone pulls in the entire tree, as well as all the history and branches. It's the easiest to do. It also allows you to use Git's worktree feature to have all your active branches checked out into separate directories but with only one copy of the repository.

```
% git clone  -o freebsd $URL -b branch [dir]
```

is how you make a deep clone. `branch` should be one of the branches listed in the previous section. It is optional if it is the main/master branch. dir is an optional directory to place it in (the default will be the name of the repo you are cloning (freebsd or src)).

You'll want a deep clone if you are interested in the history, plan on making local changes, or plan on working on more than one branch. It's the easiest to keep up to date as well. If you are interested in the history, but are working with only one branch and are short on space, you can also use `--single-branch` to only download the one branch (though some merge commits will not reference the merged-from branch which may be important for some users who are interested in detailed versions of history).

Shallow Clone

A shallow clone copies just the most current code, but none or little of the history. This can be useful when you need to build a specific revision of FreeBSD, or when you are just starting out and plan to track the tree more fully. You can also use it to limit history to only so many revisions.

```
% git clone  -o freebsd -b branch --depth 1 $URL [dir]
```

This clones the repository, but only has the most recent version in the repository. The rest of the history is not downloaded. Should you change your mind later, you can do `git fetch --unshallow` to get the old history.

Building

Once you've downloaded, building is done as described in the handbook, eg:

```
% cd src
% make buildworld
% make buildkernel
% make installkernel
% make installworld
```

so that won't be covered in depth here.

Updating

To update both types of trees uses the same commands. This pulls in all the revisions since your last update.

```
% git pull --ff-only
```

will update the tree. In Git, a `fast forward` merge is one that only needs to set a new branch pointer and doesn't need to re-create the commits. By always doing a `fast forward` merge/

pull, you'll ensure that you have an identical copy of the FreeBSD tree. This will be important if you want to maintain local patches.

See below for how to manage local changes. The simplest is to use `--autostash` on the `git pull` command, but more sophisticated options are available.

Selecting a Specific Version

In Git, the `git checkout` checks out both branches and specific versions. Git's versions are the long hashes rather than a sequential number.

When you checkout a specific version, just specify the hash you want on the command line (the Git Log command can help you decide which hash you might want):

```
% git checkout 08b8197a74
```

and you have that checked out. You'll be greeted with a message similar to the following:

```
Note: checking out '08b8197a742a96964d2924391bf9fdfeb788865d'.

You are in 'detached HEAD' state. You can look around, make experimental chang-
es and commit them, and you can discard any commits you make in this state without
impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may do so
(now or later) by using -b with the checkout command again. Example:

git checkout -b

HEAD is now at 08b8197a742a hook gpiokeys.4 to the build
```

where the last line is generated from the hash you are checking out and the first line of the commit message from that revision. The hash can be abbreviated to the shortest unique length. Git itself is inconsistent about how many digits it displays.

Bisecting

Sometimes, things go wrong. The last version worked, but the one you just updated to does not. A developer may ask to bisect the problem to track down which commit caused the regression.

If you've read the last section, you may be thinking to yourself "How the heck do I bisect with crazy version numbers like that?" then this section is for you. It's also for you if you didn't think that, but also want to bisect.

Fortunately, one uses the `git bisect` command. Here's a brief outline of how to use it. For more information, I'd suggest https://www.metaltoad.com/blog/beginners-guide-git-bisect-process-elimination or https://git-scm.com/docs/git-bisect for more details. The man page is good at describing what can go wrong, what to do when versions won't build, when you want to use terms other than `good` and `bad`, etc., none of which will be covered here.

`git bisect start` will start the bisection process. Next, you need to tell a range to go through. `git bisect good XXXXXX` will tell it the working version and `git bisect bad`

`XXXXX` will tell it the bad version. The bad version will almost always be HEAD (a special tag for what you have checked out). The good version will be the last one you checked out.

A quick aside: if you want to know the last version you checked out, you should use `git reflog`:

```
5ef0bd68b515 (HEAD -> master, freebsd/master, freebsd/HEAD) HEAD@{0}:
pull --ff-only: Fast-forward
a8163e165c5b (upstream/master) HEAD@{1}: checkout: moving from b6fb97efb682994f59b-
21fe4efb3fcfc0e5b9eeb to master
```

shows me moving the working tree to the master branch (a816...) and then updating from upstream (to 5ef0...). In this case, bad would be HEAD (or 5rf0bd68) and good would be a8163e165. As you can see from the output, HEAD@{1} also often works, but isn't foolproof if you've done other things to your git tree after updating, but before you discover the need to bisect.

Back to git bisect. Set the good version first, then set the bad (though the order doesn't matter). When you set the bad version, it will give you some statistics on the process:

```
% git bisect start
% git bisect good a8163e165c5b
% git bisect bad HEAD
Bisecting: 1722 revisions left to test after this (roughly 11 steps)
[c427b3158fd8225f6afc09e7e6f62326f9e4de7e] Fixup r361997 by balancing parens. Duh.
```

You'd then build/install that version. If it's good you'd type `git bisect good` otherwise `git bisect bad`. You'll get a similar message to the above each step. When you are done, report the bad version to the developer (or fix the bug yourself and send a patch). `git bisect reset` will end the process and return you back to where you started (usually tip of main). Again, the git-bisect manual (linked above) is a good resource for when things go wrong or for unusual cases.

Ports Considerations

The ports tree operates the same way. The branch names are different and the repos are in different locations.

The GitHub mirror is at https://github.com/freebsd/freebsd-ports.git. The cgit mirror is https://cgit-beta.freebsd.org/ports.git for now. The production Git repo will be https://git.freebsd.org/ports.git or ssh://anonsshgit.freebsd.org/ports.git when the time comes. The plan is to switch the ports repository from Subversion to Git at the end of Q1 2021.

As with ports, the `current` branches are `master` and `main` respectively. The quarterly branches are named the same as in FreeBSD's svn repo. Due to bugs in the converter, there will likely be a hash respin when the ports svn repo migrates to git, just like the src and doc repos.

Coping with Local Changes

This section addresses tracking local changes. If you have no local changes, you can stop reading now (it's the last section and OK to skip).

One item that's important for all of them: all changes are local until pushed. Unlike svn, Git uses a distributed model. For users, for most things, there's very little difference. However, if you have local changes, you can use the same tool to manage them as you use to pull in changes from FreeBSD. All changes that you've not pushed are local and can easily be modified (git rebase, discussed below, does this).

Keeping Local Changes

The simplest way to keep local changes (especially trivial ones) is to use `git stash`. In its simples form, you use `git stash` to record the changes (which pushes them onto the stash stack). Most people use this to save changes before updating the tree as described above. They then use `git stash apply` to re-apply them to the tree. The stash is a stack of changes that can be examined with `git stash list`. The git-stash man page (https://git-scm.com/docs/git-stash) has all the details.

This method is suitable when you have tiny tweaks to the tree. When you have anything non trivial, you'll likely be better off keeping a local branch and rebasing. Stashing is also integrated with the `Git pull` command: just add `--autostash` to the command line.

Keeping a Local Branch

It's much easier to keep a local branch with Git than Subversion. In Subversion you need to merge the commit, and resolve the conflicts. This is manageable, but can lead to a convoluted history that's hard to upstream should that ever be necessary, or hard to replicate if you need to do so. Git also allows one to merge, along with the same problems. That's one way to manage the branch, but it's the least flexible.

In addition to merging, Git supports the concept of `rebasing` which avoids these issues. The `git rebase` command replays all the commits of a branch at a newer location on the parent branch. We'll cover the most common scenarios that arise using it.

Create a Branch

Let's say you want to make a hack to FreeBSD's ls command to never, ever do color. There are many reasons to do this, but this example will use that as a baseline. The FreeBSD ls command changes from time to time, and you'll need to cope with those changes. Fortunately, with git rebase it usually is automatic.

```
% cd src
% git checkout main
% git checkout -b no-color-ls
% cd bin/ls
% vi ls.c # hack the changes in
% git diff # check the changes
diff --git a/bin/ls/ls.c b/bin/ls/ls.c
index 7378268867ef..cfc3f4342531 100644
--- a/bin/ls/ls.c
+++ b/bin/ls/ls.c
@@ -66,6 +66,7 @@ __FBSDID("$FreeBSD$");
 #include <stdlib.h>
 #include <string.h>
 #include <unistd.h>
```

```
+#undef COLORLS
 #ifdef COLORLS
 #include <termcap.h>
 #include <signal.h>
% # these look good, make the commit...
% git commit ls.c
```

The commit will pop you into an editor to describe what you've done. Once you enter that, you have your own local branch in the Git repo. Build and install it like you normally would, following the directions in the handbook. Git differs from other version control systems in that you have to tell it explicitly which files to use. I've opted to do it on the commit command line, but you can also do it with `git add` which many of the more in depth tutorials cover.

Time to Update

When it's time to bring in a new version, it's almost the same as w/o the branches. You would update like you would above, but there's one extra command before you update, and one after. The following assumes you are starting with an unmodified tree. It's important to start rebasing operations with a clean tree (Git usually requires this).

```
% git checkout main
% git pull --no-ff
% git rebase -i main no-color-ls
```

This will bring up an editor that lists all the commits in it. For this example, don't change it at all. This is typically what you are doing while updating the baseline (though you also use the git rebase command to curate the commits you have in the branch).

Once you're done with the above, you've move the commits to ls.c forward from the old version of FreeBSD to the newer one.

Sometimes there's merge conflicts. That's OK. Don't panic. You'd handle them the same as you would any other merge conflicts. To keep it simple, I'll just describe a common issue you might see. A pointer to a more complete treatment can be found at the end of this section.

Let's say the merge includes changes upstream in a radical shift to terminfo as well as a name change for the option. When you updated, you might see something like this:

```
Auto-merging bin/ls/ls.c
CONFLICT (content): Merge conflict in bin/ls/ls.c
error: could not apply 646e0f9cda11... no color ls
Resolve all conflicts manually, mark them as resolved with
"git add/rm <conflicted_files>", then run "git rebase --continue".
You can instead skip this commit: run "git rebase --skip".
To abort and get back to the state before "git rebase", run "git rebase --abort".
Could not apply 646e0f9cda11... no color ls
```

which looks scary. If you bring up an editor, you'll see it's a typical 3-way merge conflict resolution that you may be familiar with from other source code systems (the rest of ls.c has been omitted):

```
<<<<<<< HEAD
#ifdef COLORLS_NEW
#include <terminfo.h>
=======
#undef COLORLS
#ifdef COLORLS
#include <termcap.h>
>>>>>>> 646e0f9cda11... no color ls
```

The new code is first, and your code is second. The right fix here is to just add a `#undef COLORLS_NEW` before `#ifdef` and then delete the old changes:

```
#undef COLORLS_NEW
#ifdef COLORLS_NEW
#include <terminfo.h>
```

save the file. The rebase was interrupted, so you have to complete it:

```
% git add ls.c
% git rebase --continue
```

which tells Git that ls.c has changed and to continue the rebase operation. Since there was a conflict, you'll get kicked into the editor to update the commit message if necessary. If the commit message is still accurate, just exit the editor.

If you get stuck during the rebase, don't panic. `git rebase --abort` will take you back to a clean slate. It's important, though, to start with an unmodified tree.

For more on this topic, https://www.freecodecamp.org/news/the-ultimate-guide-to-git-merge-and-git-rebase/ provides a rather extensive treatment. It is a good resource for issues that arrise occasionally but are too obscure for this guide.

Switching to a Different FreeBSD Branch

If you wish to shift from stable/12 to the current branch and if you have a deep clone, the following will suffice:

```
% git checkout main
% # build and install here...
```

If you have a local branch, though, there are one or two caveats. First, rebase will rewrite history, so you'll likely want to do something to save it. Second, jumping branches tends to encounter more conflicts. If we pretend the example above was relative to stable/12, then to move to main, I'd suggest the following:

```
% git checkout no-color-ls
% git checkout -b no-color-ls-stable-12 # create another name for this branch
% git rebase -i stable/12 no-color-ls --onto main
```

What the above does is `checkout no-color-ls`. Then create a new name for it (`no-color-ls-stable-12`) in case you need to get back to it. Then you rebase onto the main branch. This will find all the commits to the current no-color-ls branch (back to where it meets up with the `stable/12 branch`) and then it will replay them onto the main branch creating a new `no-color-ls branch` there (which is why I had you create a place holder name).

Migrating from an Existing Git Clone

If you have work based on a previous Git conversion or a locally running git-svn conversion, migrating to new repository can encounter problems because Git has no knowledge about the connection between the two.

If do not have a lot of local changes, the easiest way would be to cherry-pick your changes to the new base:

```
% git checkout main
% git cherry-pick old_branch..your_branch
```

Or alternatively, you can do the same thing with rebase:

```
% git rebase --onto main master your_branch
```

If you do have a lot of changes, you would probably want to perform a merge instead. The idea is to create a merge point that consolidates the history of the old_branch, and the new source of truth (main).

We intend to publish a set of pairs of SHA1s for this, but if you are running a local conversion, you can find out by looking up the same commit that are found on both parents:

```
% git show old_branch
```

You will see a commit message, now search for that in the new branch:

```
% git log --grep="commit message on old_branch" freebsd/main
```

You would get a SHA1 on the new main branch, create a helper branch (in the example we call it `stage`) from that SHA1:

```
% git checkout -b stage SHA1_found_from_git_log
Then perform a merge of the old branch:
% git merge -s ours -m "Mark old branch as merged" old_branch
```

With that, it's possible to merge your work branch or the main branch in any order without problem. Eventually, when you are ready to commit your work back to main, you can perform a rebase to main or do a squash commit by combining everything into one commit.

WARNER LOSH is a Senior Software Engineer at Netflix and has been a FreeBSD contributor for over 20 years. Warner has improved a number of systems—for example, the boot loader—in FreeBSD. Prior to Netflix, he produced flash drives and measured atomic clocks for accuracy. His code still measures some of the clocks that create UTC!