June 26, 2016 by: Brian Vanderwal 4 Comments

< back to Blog Home

Parallelize Development Using Git Worktrees

Recently, I was in a situation in which I really needed two separate copies of my Git repository. I was about to make a full clone of the repository, but I decided to see if Git had a better solution. And in fact, Git introduced the worktree feature not too long ago (as of version 2.5, released July 2015).

A worktree gives you an extra working copy of your repository, and it's almost as easy as creating a new branch. All you need to do is set up a new worktree like this:

```
git worktree add ../new-worktree-dir some-existing-branch
```

This will set up the directory `../new-worktree-dir` as though it were a clone of your repository. You can create this directory anywhere on the same filesystem, but it should be somewhere *outside* of your main repository directory! You can then proceed to use the worktree directory as usual, checking out branches, pushing upstream, etc.

So why would you want one (or more) worktrees? Here a few good reasons:

Run Tests While Working on Another Branch

In a large project with very good test coverage, some test suites can take a long time to run—far beyond a convenient amount of time to sit and wait for them to finish. In these cases, it can be helpful to run multiple test suites in parallel. Many IDEs allow opening multiple projects at once, but each must be in its own directory. You could `git clone` two entirely separate repositories, but worktrees are better:

- Worktrees are implemented using hard links, so they are lightweight and fast (whereas separate `git clone`s copy down the full repository).
- You can share changes between worktrees (as long as they're committed to at least the local repository). With full clones, you would have to push one repo to the remote and then pull it on the other.
- If you accidentally commit some changes to the wrong clone, you'll have to port them over by hand (if they're simple) or by using `patch`. With worktrees, you can just `git cherry-pick` and `git reset` to fix the mistake.

I frequently keep an extra worktree around just for running tests. One limitation of worktrees is that you can't have the same branch checked out in multiple places. I get around this by creating local temporary branches, like so:

```
git co -b TEMP/original-branch-name feature/original-branch-name
```

I use the TEMP prefix to emphasize that the branch is temporary (my shell prompt includes the name of the current branch, so this practically yells it at me). When changes are committed on the original branch, a quick `git merge feature/original-branch-name` will catch up with the temporary branch.

Compare Multiple Versions

Sometimes, you need to compare two versions of a project, but a simple `diff` just doesn't cut it. What you really need is to see both versions simultaneously so you can compare things side-by-side or even run both versions at the same time. Or perhaps you are in the middle of a complicated change, and it's hard to tell what you just broke. You can easily check out a previously tagged version or any arbitrary commit in a worktree.

Work on a Different Branch without Disturbing Your Current Working Copy

Maybe you need to work on a different branch, but your current working directory is in such disarray that even `git stash` can't help. Switching branches may also have undesired side effects depending on your project (for example, causing an IDE to re-index).

Quickly Verify That the Project Works with a Clean Checkout

Everybody has probably had the experience of a build failure because a co-worker forgot to include some files with a commit. It could be that they forgot to add the files to Git, or perhaps some `.gitignore` rules were too broad. If the build works for them but not for you, then you may be missing some files. One way to find out is to test it against a working

BY: BRIAN VANDERWAL
Software Consultant and Developer at Atomic Object Grand Rapids. Clean coder. Minimalist. Rock climber.

READ BIO →

POSTED IN:
● Developer Tools

copy that you know to be clean. Since worktrees give you a clean checkout, they can be used to verify that all of the files that need to be included have been added to Git. (This will only work reliably if you start by creating a new worktree.)

Caveats

Can't work on the same branch simultaneously

This may seem like a limitation, but it's really not a big deal. I do most of my work in the main repository directory and have one worktree directory for any/all of the reasons listed above. It's easy enough to create a temporary branch to mirror an existing one.

Doesn't work with submodules

Repositories that utilize submodules currently cannot take advantage of worktrees.

Cleanup

When you're all finished with a worktree, you can just delete its directory and then run `git worktree prune` from the main repository directory. But once you start using worktrees, you won't need to think about how to get rid of them!

Share this article:

Posted in Developer Tools

Related Posts

Setting up GitHub Actions for a React/Node Project
by Mike Woelmer

Developing Azure Functions in a Docker Container
by Patrick Bacon

macOS Catalina: Fixing Emacs After an Upgrade
by Chris Farber

By commenting below, you agree to the terms and conditions outlined in our (linked) Privacy Policy

4 Comments

Praneeth Bobba
February 9, 2017

Hey Brian

Thanks for an informative blog. As said, I tried to use a worktree like this

git worktree add c:/newcodebase master

But I get this error
fatal: Could not reset index file to revision 'HEAD'.

Any ideas on what might have been wrong ?

Brian
February 11, 2017

Praneeth,

If your main repository already has the master branch checked out, you'll need to use a different branch when creating your worktree (a branch can only be checked out in one worktree at a time). If necessary, you can also use the -b flag to create a new branch at the same time, like "git worktree add c:/newcodebase -b master-copy".

Jonathan
February 22, 2017

This was very helpful. Thanks

Cyprien
August 26, 2017

> With full clones, you would have to push one repo to the remote and then pull it on the other.

or you can add each-other as remotes. Don't forget the power of git :-)

Comments are closed.

We're hiring in Ann Arbor *open positions >* x