

How I Use Git Worktrees

Jul 25, 2024

There are a bunch of posts on the internet about using `git worktree` command. As far as I can tell, most of them are primarily about using worktrees as a replacement of, or a supplement to git branches. Instead of switching branches, you just change directories. This is also how I originally had used worktrees, but that didn't stick, and I abandoned them. But recently worktrees grew on me, though my new use-case is unlike branching.

When a Branch is Enough

If you use worktrees as a replacement for branching, that's great, no need to change anything! But let me start with explaining why that workflow isn't for me.

The principle problem with using branches is that it's hard to context switch in the middle of doing something. You have your branch, your commit, a bunch of changes in the work tree, some of them might be stage and some upstage. You can't really tell Git "save all this context and restore it later". The solution that git suggests here is to use stashing, but that's awkward, as it is too easy to get lost when stashing several things at the same time, and then applying the stash on top of the wrong branch.


Managing git state became much easier for me when I realize that staging area and stash are just bad features, and life's easier if I avoid them. Instead, I just commit whatever and deal with it later. So, when I need to switch a branch in the middle of things, what I do is, basically:

```
1 | $ git add .
2 | $ git commit -m.
3 | $ git switch another-branch
```


And, to switch back,

```
1 | $ git switch -
2 |
3 | # Undo the last commit, but keep its changes in the working tree
4 | $ git reset HEAD~
```

To make this more streamlined, I have a `ggc` utility which does "commit all with a trivial message" atomically.

 Reminder: git is not a version control system, git is a toolbox for building a VCS. Do have a low-friction way to add your own scripts for common git operations.

And I don't always reset HEAD~ — I usually just continue hacking with . in my git log and then amend the commit once I am satisfied with subset of changes

 Reminder: magit, for [Emacs](#) and [VS Code](#), is excellent for making such commit surgery easy. In particular, **instant fixup** is excellent. Even if you don't use magit, you should have an equivalent of instant fixup among your git scripts.

So that's how I deal with switching branches. But why worktrees then?

Worktree Per Concurrent Activity

It's a bit hard to describe, but:

- I have a fixed number of worktrees (5, to be exact)
- worktrees are mostly uncorrelated to branches
- and instead correspond to my concurrent activities during coding

Specifically:

- The **main** worktree is a readonly worktree that contains a recent snapshot of the remote main branch. I use this tree to compare the code I am currently working on and/or reviewing with the master version (this includes things like “how long the build takes”, “what is the behavior of this test” and the like, so not just the actual source code).
- The **work** worktree, where I write most of the code. I often need to write new code and compare it with old code at the same time. But can't actually work on two different things in parallel. That's why main and work are different branches, but work also constantly switches branches.
- The **review** worktree, where I checkout code for code review. While I can't review code and write code at the same time, there is one thing I am implementing, and one thing I am reviewing, but the review and implementation proceed concurrently.
- Then, there's **fuzz** tree, where I run log-running fuzzing jobs for the code I am actively working on. My overall idealized feature workflow looks like this:

```
1 | # go to the `work` worktree
2 | $ cd ~/projects/tigerbeetle/work
3 |
4 | # Create a new branch. As we work with a centralized repo,
5 | # rather than personal forks, I tend to prefix my branch names
```

```
6 # with `matklad/`
7 $ git switch -c matklad/awesome-feature
8
9 # Start with a reasonably clean slate.
10 # In reality, I have yet another script to start a branch off
11 # fresh remote main, but this reset is a good enough approximation.
12 $ git reset --hard origin/main
13
14 # For more complicated features, I start with an empty commit
15 # and write the commit message _first_, before starting the work.
16 # That's a good way to collect your thoughts and discover dead
17 # ends more gracefully than hitting a brick wall coding at 80 WPM.
18 $ git commit --allow-empty
19
20 # Hack furiously writing throughway code.
21 $ code .
22
23 # At this point, I have something that I hope works,
24 # but I would be embarrassed to share with anyone!
25 # So that's the good place to kick off fuzzing.
26
27 # First, I commit everything so far.
28 # Remember, I have `ggc` one liner for this:
29 $ git add . && git commit -m.
30
31 # Now I go to my `fuzz` worktree and kick off fuzzing.
32 # I usually split screen here.
33 # On the left, I copy the current commit hash.
34 # On the right, I switch to the fuzzing worktree,
35 # switch to the copied commit, and start fuzzing:
36
37 $ git add . && git commit -m. |
38 $ git rev-parse HEAD | ctrlc | $ cd ../fuzz
39 $ | $ git switch -d $(ctrlv)
40 $ | $ ./zig/zig build fuzz
41 $ |
42
43 # While the fuzzer hums on right, I continue to furiously refactor
44 # the code on the left and hammer my empty commit with a wishful
45 # thinking message and my messy code commit with `.` message into
46 # a semblance of clean git history
47
48 $ code .
49 $ magit-goes-brrrrr
50
51 # At this point, in the work tree, I am happy with both the code
52 # and the git history, so, if the fuzzer on the right is happy,
53 # a PR is opened!
```

```
54 |  
55 | $  
56 | $ git push --force-with-lease | $ ./zig/zig build fuzz  
57 | $ gh pr create --web | # Still hasn't failed  
58 | $
```

This is again concurrency: I can hack on the branch while the fuzzer tests the “same” code. Note that it is crucial that the fuzzing tree operates in the detached head state (-d flag for git switch). In general, -d is very helpful with this style of worktree work. I am also sympathetic to [the argument](#) that, like the staging area and the stash, git branches are a miss feature, but I haven’t made the plunge personally yet.

- Finally, the last tree I have is **scratch** — this is a tree for arbitrary random things I need to do while working on something else. For example, if I am working on matklad/my-feature in work, and reviewing #6292 in review, and, while reviewing, notice a tiny unrelated typo, the PR for that typo is quickly prepped in the scratch worktree:

```
1 | $ cd ../scartch  
2 | $ git switch -c matklad/quick-fix  
3 | $ code . && git add . && git commit -m 'typo' && git push  
4 | $ cd -
```

TL;DR: consider using worktrees not as a replacement for branches, but as a means to manage concurrency in your tasks. My level of concurrency is:

- main for looking at the pristine code,
- work for looking at my code,
- review for looking at someone elses code,
- fuzz for my computer to look at my code,
- scratch for everything else!

 [Fix typo](#)  [Subscribe](#)  [Get in touch](#)  [matklad](#)