new blog post: confusing git terminology

1 message

Julia Evans <julia@wizardzines.com> Fri, Nov 3, 2023 at 4:16 PM
To: mperdikeas@gmail.com

Two blog posts this week:

* Confusing git terminology
* Some miscellaneous git facts

Confusing git terminology

Hello! I'm slowly working on explaining git. One of my biggest problems is that after almost 15 years of using git, I've become very used to git's idiosyncracies and it's easy for me to forget what's confusing about it.

So I asked people on Mastodon:

> what git jargon do you find confusing? thinking of writing a blog post that explains some of git's weirder terminology: "detached HEAD state", "fast-forward", "index/staging area/staged", "ahead of 'origin/main' by 1 commit", etc

I got a lot of GREAT answers and I'll try to summarize some of them here. Here's a list of the terms:

- HEAD and "heads"
- "detached HEAD state"
- "ours" and "theirs" while merging or rebasing
- "Your branch is up to date with 'origin/main'"
- HEAD^, HEAD~ HEAD^^, HEAD~~, HEAD^2, HEAD~2
- .. and …
- "can be fast-forwarded"
- "reference", "symbolic reference"
- refspecs
- "tree-ish"
- "index", "staged", "cached"
- "reset", "revert", "restore"
- "untracked files", "remote-tracking branch", "track remote branch"
- checkout
- reflog
- merge vs rebase vs cherry-pick
- rebase –onto
- commit
- more confusing terms

I've done my best to explain what's going on with these terms, but they cover basically every single major feature of git which is definitely too much for a single blog post so it's pretty patchy in some places.

`HEAD` and "heads"

A few people said they were confused by the terms `HEAD` and `refs/heads/main`, because it sounds like it's some complicated technical internal thing.

Here's a quick summary:

- "heads" are "branches". Internally in git, branches are stored in a directory called `.git/refs/heads`. (technically the official git glossary says that the branch is all the commits on it and the head is just the most recent commit, but they're 2 different ways to think about the same thing)
- `HEAD` is the current branch. It's stored in `.git/HEAD`.

I think that “a `head` is a branch, `HEAD` is the current branch” is a good candidate for the weirdest terminology choice in git, but it’s definitely too late for a clearer naming scheme so let’s move on.

There are some important exceptions to “HEAD is the current branch”, which we’ll talk about next.

“detached HEAD state”

You’ve probably seen this message:

```
<code>$ git checkout v0.1
You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.
[...]
</code>
```

Here’s the deal with this message:

- In Git, usually you have a “current branch” checked out, for example `main`.
- The place the current branch is stored is called `HEAD`.
- Any new commits you make will get added to your current branch, and if you run `git merge other_branch`, that will also affect your current branch
- But `HEAD` doesn’t **have** to be a branch! Instead it can be a commit ID.
- Git calls this state (where HEAD is a commit ID instead of a branch) “detached HEAD state”
- For example, you can get into detached HEAD state by checking out a tag, because a tag isn’t a branch
- if you don’t have a current branch, a bunch of things break:
  - `git pull` doesn’t work at all (since the whole point of it is to update your current branch)
  - neither does `git push` unless you use it in a special way
  - `git commit`, `git merge`, `git rebase`, and `git cherry-pick` **do** still work, but they’ll leave you with “orphaned” commits that aren’t connected to any branch, so those commits will be hard to find
- You can get out of detached HEAD state by either creating a new branch or switching to an existing branch

“ours” and “theirs” while merging or rebasing

If you have a merge conflict, you can run `git checkout --ours file.txt` to pick the version of `file.txt` from the “ours” side. But which side is “ours” and which side is “theirs”?

I always find this confusing and I never use `git checkout --ours` because of that, but I looked it up to see which is which.

For merges, here’s how it works: the current branch is “ours” and the branch you’re merging in is “theirs”, like this. Seems reasonable.

```
<code>$ git checkout merge-into-ours # current branch is "ours"
$ git merge from-theirs # branch we're merging in is "theirs"
</code>
```

For rebases it’s the opposite – the current branch is “theirs” and the target branch we’re rebasing onto is “ours”, like this:

```
<code>$ git checkout theirs # current branch is "theirs"
$ git rebase ours # branch we're rebasing onto is "ours"
</code>
```

I think the reason for this is that under the hood `git rebase main` is merging the current branch into main (it’s like `git checkout main; git merge current_branch`), but I still find it confusing.

This nice tiny site explains the “ours” and “theirs” terms.

A couple of people also mentioned that VSCode calls “ours”/“theirs” “current change”/“incoming change”, and that it’s confusing in the exact same way.

“Your branch is up to date with ‘origin/main’”

This message seems straightforward – it’s saying that your `main` branch is up to date with the origin!

But it’s actually a little misleading. You might think that this means that your `main` branch is up to date. It doesn’t. What it **actually** means is – if you last ran `git fetch` or `git pull` 5 days ago, then your `main` branch is up to date with all the changes **as of 5 days ago**.

So if you don't realize that, it can give you a false sense of security.

I think git could theoretically give you a more useful message like "is up to date with the origin's `main` **as of your last fetch 5 days ago**" because the time that the most recent fetch happened is stored in the reflog, but it doesn't.

`HEAD^, HEAD~ HEAD^^, HEAD~~, HEAD^2, HEAD~2`

I've known for a long time that `HEAD^` refers to the previous commit, but I've been confused for a long time about the difference between `HEAD~` and `HEAD^`.

I looked it up, and here's how these relate to each other:

- `HEAD^` and `HEAD~` are the same thing (1 commit ago)
- `HEAD^^^` and `HEAD~~~` and `HEAD~3` are the same thing (3 commits ago)
- `HEAD^3` refers the the third parent of a commit, and is different from `HEAD~3`

This seems weird – why are `HEAD~` and `HEAD^` the same thing? And what's the "third parent"? Is that the same thing as the parent's parent's parent? (spoiler: it isn't) Let's talk about it!

Most commits have only one parent. But merge commits have multiple parents – they're merging together 2 or more commits. In Git `HEAD^` means "the parent of the HEAD commit". But what if HEAD is a merge commit? What does `HEAD^` refer to?

The answer is that `HEAD^` refers to the the **first** parent of the merge, `HEAD^2` is the second parent, `HEAD^3` is the third parent, etc.

But I guess they also wanted a way to refer to "3 commits ago", so `HEAD^3` is the third parent of the current commit (which may have many parents if it's a merge commit), and `HEAD~3` is the parent's parent's parent.

I think in the context of the merge commit ours/theirs discussion earlier, `HEAD^` is "ours" and `HEAD^^` is "theirs".

`..` and `...`

Here are two commands:

- `git log main..test`
- `git log main...test`

What's the difference between `..` and `...`? I never use these so I had to look it up in man git-range-diff. It seems like the answer is that in this case:

```
<code>A - B main
  \
    C - D test
</code>
```

- `main..test` is commits C and D
- `test..main` is commit B
- `main...test` is commits B, C, and D

But it gets worse: apparently `git diff` also supports `..` and `...`, but they do something completely different than they do with `git log`? I think the summary is:

- `git log test..main` shows changes on `main` that aren't on `test`, whereas `git log test...main` shows changes on *both* sides.
- `git diff test..main` shows `test` changes *and* `main` changes (it diffs `B` and `D`) whereas `git diff test...main` diffs `A` and `D` (it only shows you the diff on one side).

this blog post talks about it a bit more.

"can be fast-forwarded"

Here's a very common message you'll see in `git status`:

```
<code>$ git status
On branch main
Your branch is behind 'origin/main' by 2 commits, and can be fast-forwarded.
  (use "git pull" to update your local branch)
</code>
```

What does “fast-forwarded” mean? Basically it’s trying to say that the two branches look something like this: (newest commits are on the right)

```
<code>main:        A - B - C
origin/main: A - B - C - D - E
</code>
```

or visualized another way:

```
<code>A - B - C - D - E (origin/main)
        |
       main
</code>
```

Here `origin/main` just has 2 extra commits that `main` doesn’t have, so it’s easy to bring `main` up to date – we just need to add those 2 commits. Literally nothing can possibly go wrong – there’s no possibility of merge conflicts. A fast forward merge is a very good thing! It’s the easiest way to combine 2 branches.

After running `git pull`, you’ll end up this state:

```
<code>main:        A - B - C - D - E
origin/main: A - B - C - D - E
</code>
```

Here’s an example of a state which **can’t** be fast-forwarded.

```
<code>             A - B - C - X  (main)
                     |
                     - - D - E  (origin/main)
</code>
```

Here `main` has a commit that `origin/main` doesn’t have (X). So you can’t do a fast forward. In that case, `git status` would say:

```
<code>$ git status
Your branch and 'origin/main' have diverged,
and have 1 and 2 different commits each, respectively.
</code>
```

“reference”, “symbolic reference”

I’ve always found the term “reference” kind of confusing. There are at least 3 things that get called “references” in git

- branches and tags like `main` and `v0.2`
- `HEAD`, which is the current branch
- things like `HEAD^^^` which git will resolve to a commit ID. Technically these are probably not “references”, I guess git calls them “revision parameters” but I’ve never used that term.

“symbolic reference” is a very weird term to me because personally I think the only symbolic reference I’ve ever used is `HEAD` (the current branch), and `HEAD` has a very central place in git (most of git’s core commands’ behaviour depends on the value of `HEAD`), so I’m not sure what the point of having it as a generic concept is.

refspecs

When you configure a git remote in `.git/config`, there’s this `+refs/heads/main:refs/remotes/origin/main` thing.

```
<code>[remote "origin"]
    url = git@github.com:jvns/pandas-cookbook
    fetch = +refs/heads/main:refs/remotes/origin/main
</code>
```

I don’t really know what this means, I’ve always just used whatever the default is when you do a `git clone` or `git remote add`, and I’ve never felt any motivation to learn about it or change it from the default.

“tree-ish”

The man page for `git checkout` says:

<code> git checkout [-f|--ours|--theirs|-m|--conflict=<style>] [<tree-ish>] [--]
<pathspec>...
</code>

What's `tree-ish`??? What git is trying to say here is when you run `git checkout THING .`, `THING` can be either:

- a commit ID (like `182cd3f`)
- a reference to a commit ID (like `main` or `HEAD^^` or `v0.3.2`)
- a subdirectory **inside** a commit (like `main:./docs`)
- I think that's it????

Personally I've never used the "directory inside a commit" thing and from my perspective "tree-ish" might as well just mean "commit or reference to commit".

"index", "staged", "cached"

All of these refer to the exact same thing (the file `.git/index`, which is where your changes are staged when you run `git add`):

- `git diff --cached`
- `git rm --cached`
- `git diff --staged`
- the file `.git/index`

Even though they all ultimately refer to the same file, there's some variation in how those terms are used in practice:

- Apparently the flags `--index` and `--cached` do not generally mean the same thing. I have personally never used the `--index` flag so I'm not going to get into it, but this blog post by Junio Hamano (git's lead maintainer) explains all the gnarly details
- the "index" lists untracked files (I guess for performance reasons) but you don't usually think of the "staging area" as including untracked files"

"reset", "revert", "restore"

A bunch of people mentioned that "reset", "revert" and "restore" are very similar words and it's hard to differentiate them.

I think it's made worse because

- `git reset --hard` and `git restore .` on their own do basically the same thing. (though `git reset --hard COMMIT` and `git restore --source COMMIT .` are completely different from each other)
- the respective man pages don't give very helpful descriptions:
  - `git reset`: "Reset current HEAD to the specified state"
  - `git revert`: "Revert some existing commits"
  - `git restore`: "Restore working tree files"

Those short descriptions do give you a better sense for which noun is being affected ("current HEAD", "some commits", "working tree files") but they assume you know what "reset", "revert" and "restore" mean in this context.

Here are some short descriptions of what they each do:

- `git revert COMMIT`: Create a new commit that's the "opposite" of COMMIT on your current branch (if COMMIT added 3 lines, the new commit will delete those 3 lines)
- `git reset --hard COMMIT`: Force your current branch back to the state it was at `COMMIT`, erasing any new changes since `COMMIT`. Very dangerous operation.
- `git restore --source=COMMIT PATH`: Take all the files in `PATH` back to how they were at `COMMIT`, without changing any other files or commit history.

"untracked files", "remote-tracking branch", "track remote branch"

Git uses the word "track" in 3 different related ways:

- `Untracked files:` in the output of `git status`. This means those files aren't managed by Git and won't be included in commits.
- a "remote tracking branch" like `origin/main`. This is a local reference, and it's the commit ID that `main` pointed to on the remote `origin` the last time you ran `git pull` or `git fetch`.
- "branch foo set up to **track** remote branch bar from origin"

The “untracked files” and “remote tracking branch” thing is not too bad – they both use “track”, but the context is very different. No big deal. But I think the other two uses of “track” are actually quite confusing:

- `main` is a branch that tracks a remote
- `origin/main` is a remote-tracking branch

But a “branch that tracks a remote” and a “remote-tracking branch” are different things in Git and the distinction is pretty important! Here’s a quick summary of the differences:

- `main` is a branch. You can make commits to it, merge into it, etc. It’s often configured to “track” the remote `main` in `.git/config`, which means that you can use `git pull` and `git push` to push/pull changes.
- `origin/main` is not a branch. It’s a “remote-tracking branch”, which is not a kind of branch (I’m sorry). You **can’t** make commits to it. The only way you can update it is by running `git pull` or `git fetch` to get the latest state of `main` from the remote.

I’d never really thought about this ambiguity before but I think it’s pretty easy to see why folks are confused by it.

checkout

Checkout does two totally unrelated things:

- `git checkout BRANCH` switches branches
- `git checkout file.txt` discards your unstaged changes to `file.txt`

This is well known to be confusing and git has actually split those two functions into `git switch` and `git restore` (though you can still use checkout if, like me, you have 15 years of muscle memory around `git checkout` that you don’t feel like unlearning)

Also personally after 15 years I still can’t remember the order of the arguments to `git checkout main file.txt` for restoring the version of `file.txt` from the `main` branch.

I think sometimes you need to pass `--` to `checkout` as an argument somewhere to help it figure out which argument is a branch and which ones are paths but I never do that and I’m not sure when it’s needed.

reflog

Lots of people mentioning reading reflog as `re-flog` and not `ref-log`. I won’t get deep into the reflog here because this post is REALLY long but:

- “reference” is an umbrella term git uses for branches, tags, and HEAD
- the reference log (“reflog”) gives you the history of everything a reference has ever pointed to
- It can help get you out of some VERY bad git situations, like if you accidentally delete an important branch
- I find it one of the most confusing parts of git’s UI and I try to avoid needing to use it.

merge vs rebase vs cherry-pick

A bunch of people mentioned being confused about the difference between merge and rebase and not understanding what the “base” in rebase was supposed to be.

I’ll try to summarize them very briefly here, but I don’t think these 1-line explanations are that useful because people structure their workflows around merge / rebase in pretty different ways and to really understand merge/rebase you need to understand the workflows. Also pictures really help. That could really be its whole own blog post though so I’m not going to get into it.

- merge creates a single new commit that merges the 2 branches
- rebase copies commits on the current branch to the target branch, one at a time.
- cherry-pick is similar to rebase, but with a totally different syntax (one big difference is that rebase copies commits FROM the current branch, cherry-pick copies commits TO the current branch)

`rebase --onto`

`git rebase` has an flag called `onto`. This has always seemed confusing to me because the whole point of `git rebase main` is to rebase the current branch **onto** main. So what’s the extra `onto` argument about?

I looked it up, and `--onto` definitely solves a problem that I've rarely/never actually had, but I guess I'll write down my understanding of it anyway.

```
<code>A - B - C (main)
     \
      D - E - F - G (mybranch)
          |
          otherbranch
</code>
```

Imagine that for some reason I just want to move commits `F` and `G` to be rebased on top of `main`. I think there's probably some git workflow where this comes up a lot.

Apparently you can run `git rebase --onto main otherbranch mybranch` to do that. It seems impossible to me to remember the syntax for this (there are 3 different branch names involved, which for me is too many), but I heard about it from a bunch of people so I guess it must be useful.

commit

Someone mentioned that they found it confusing that commit is used both as a verb and a noun in git.

for example:

- verb: "Remember to commit often"
- noun: "the most recent commit on `main`"

My guess is that most folks get used to this relatively quickly, but this use of "commit" is different from how it's used in SQL databases, where I think "commit" is just a verb (you "COMMIT" to end a transaction) and not a noun.

Also in git you can think of a Git commit in 3 different ways:

1. a **snapshot** of the current state of every file
2. a **diff** from the parent commit
3. a **history** of every previous commit

None of those are wrong: different commands use commits in all of these ways. For example `git show` treats a commit as a diff, `git log` treats it as a history, and `git restore` treats it as a snapshot.

But git's terminology doesn't do much to help you understand in which sense a commit is being used by a given command.

more confusing terms

Here are a bunch more confusing terms. I don't know what a lot of these mean.

things I don't really understand myself:

- "the git pickaxe" (maybe this is `git log -S` and `git log -G`, for searching the diffs of previous commits?)
- submodules (all I know is that they don't work the way I want them to work)
- "cone mode" in git sparse checkout (no idea what this is but someone mentioned it)

things that people mentioned finding confusing but that I left out of this post because it was already 3000 words:

- blob, tree
- the direction of "merge"
- "origin", "upstream", "downstream"
- that `push` and `pull` aren't opposites
- the relationship between `fetch` and `pull` (pull = fetch + merge)
- git porcelain
- subtrees
- worktrees
- the stash
- "master" or "main" (it sounds like it has a special meaning inside git but it doesn't)
- when you need to use `origin main` (like `git push origin main`) vs `origin/main`

github terms people mentioned being confused by:

- "pull request" (vs "merge request" in gitlab which folks seemed to think was clearer)

- what “squash and merge” and “rebase and merge” do (I’d never actually heard of `git merge --squash` until yesterday, I thought “squash and merge” was a special github feature)

it’s genuinely “every git term”

I was surprised that basically every other core feature of git was mentioned by at least one person as being confusing in some way. I’d be interested in hearing more examples of confusing git terms that I missed too.

There’s another great post about this from 2012 called the most confusing git terminology. It talks more about how git’s terminology relates to CVS and Subversion’s terminology.

If I had to pick the 3 most confusing git terms, I think right now I’d pick:

- a `head` is a branch, `HEAD` is the current branch
- “remote tracking branch” and “branch that tracks a remote” being different things
- how “index”, “staged”, “cached” all refer to the same thing

that’s all!

I learned a lot from writing this – I learned a few new facts about git, but more importantly I feel like I have a slightly better sense now for what someone might mean when they say that everything in git is confusing.

I really hadn’t thought about a lot of these issues before – like I’d never realized how “tracking” is used in such a weird way when discussing branches.

Also as usual I might have made some mistakes, especially since I ended up in a bunch of corners of git that I hadn’t visited before.

Some miscellaneous git facts

I’ve been very slowly working on writing about how Git works. I thought I already knew Git pretty well, but as usual when I try to explain something I’ve been learning some new things.

None of these things feel super surprising in retrospect, but I hadn’t thought about them clearly before.

The facts are:

- the “index”, “staging area” and “–cached” are all the same thing
- the stash is a bunch of commits
- not all references are branches or tags
- merge commits aren’t empty

Let’s talk about them!

the “index”, “staging area” and “–cached” are all the same thing

When you run `git add file.txt`, and then `git status`, you’ll see something like this:

People usually call this “staging a file” or “adding a file to the staging area”.

When you stage a file with `git add`, behind the scenes git adds the file to its object database (in `.git/objects`) and updates a file called `.git/index` to refer to the newly added file.

This “staging area” actually gets referred to by 3 different names in Git. All of these refer to the exact same thing (the file `.git/index`):

- `git diff --cached`
- `git diff --staged`
- the file `.git/index`

I felt like I should have realized this earlier, but I didn’t, so there it is.

the stash is a bunch of commits

When I run `git stash` to stash my changes, I’ve always been a bit confused about where those changes actually went. It turns out that when you run `git stash`, git makes some commits with your changes and labels them with a reference called `stash` (in `.git/refs/stash`).

Let’s stash this blog post and look at the log of the `stash` reference:

Now we can look at the commit `2ff2c273` to see what it contains:

Unsurprisingly, it contains this blog post. Makes sense!

`git stash` actually creates 2 separate commits: one for the index, and one for your changes that you haven't staged yet. I found this kind of heartening because I've been working on a tool to snapshot and restore the state of a git repository (that I may or may not ever release) and I came up with a very similar design, so that made me feel better about my choices.

Apparently older commits in the stash are stored in the reflog.

not all references are branches or tags

Git's documentation often refers to "references" in a generic way that I find a little confusing sometimes. Personally 99% of the time when I deal with a "reference" in Git it's a branch or `HEAD` and the other 1% of the time it's a tag. I actually didn't know ANY examples of references that weren't branches or tags or `HEAD`.

But now I know one example – the stash is a reference, and it's not a branch or tag! So that's cool.

Here are all the references in my blog's git repository (other than `HEAD`):

Some other references people mentioned in reponses to this post:

- `refs/notes/*`, from `git notes`
- `refs/pull/123/head`, and `refs/pull/123/head` for GitHub pull requests (which you can get with `git fetch origin refs/pull/123/merge`)
- `refs/bisect/*`, from `git bisect`

merge commits aren't empty

Here's a toy git repo where I created two branches `x` and `y`, each with 1 file (`x.txt` and `y.txt`) and merged them. Let's look at the merge commit.

If I run `git show 96a8afb`, the commit looks "empty": there's no diff!

But if I diff the merge commit against each of its two parent commits separately, you can see that of course there **is** a diff:

It seems kind of obvious in retrospect that merge commits aren't actually "empty" (they're snapshots of the current state of the repo, just like any other commit), but I'd never thought about why they appear to be empty.

Apparently the reason that these merge diffs are empty is that merge diffs only show **conflicts** – if I instead create a repo with a merge conflict (one branch added `x` and another branch added `y` to the same file), and show the merge commit where I resolved the conflict, it looks like this:

It looks like this is trying to tell me that one branch added `x`, another branch added `y`, and the merge commit resolved it by putting `z` instead. But in the earlier example, there was no conflict, so Git didn't display a diff at all.

(thanks to Jordi for telling me how merge diffs work)

that's all!

I'll keep this post short, maybe I'll write another blog post with more git facts as I learn them.