Deleting merged branch gives "error: The branch X is not fully merged...'

Asked 8 years, 1 month ago Modified 6 years, 8 months ago

Viewed 19k times

17

I have been caught in a pickle. And am unable to find my answers from other SO questions or from reading the git documentation. I would really appreciate your help here.

After merging my branch on Github, I delete my branch from the UI (remote). Periodically I prune my local branches:

```
git checkout master
git pull origin master
git fetch -p
git branch -d X
Error: error: The branch X is not fully merged. If
you are sure you want to delete it, run 'git
branch -D X'
```

The branch X has been merged. And I can see the commits from X when I do `git log` on `master`. Why does this happen? I want to use `-d` and not `-D` since this could lead to disastrously deletion of branches that were genuinely not merged.

git github merge

Share

Improve this question

Follow

asked Nov 9, 2016 at 20:14

Struggler
682 ● 2 ● 9 ● 22

You can use the `-f` to force the local deletion – panza Jul 3, 2020 at 18:11

2 Answers

Sorted by: Highest score (default)

14

There are several different tricky bits here.

The key question for `git branch -d` (delete without forcing[1]) is not "is the branch merged?", because that question is literally un-answerable. The question is instead "is the branch merged *into* <fill in something here>?"

This test was changed in Git version 1.7 (which means everyone should have it today, but check your Git version), in [commit 99c419c91554e9f60940228006b7d39d42704da7](commit) by Junio C Hamano:

> branch -d: base the "already-merged" safety on the branch it merges with
>
> When a branch is marked to merge with another ref (e.g. local 'next' that merges from and pushes back to origin's 'next', with 'branch.next.merge'

> set to 'refs/heads/next'), it makes little sense to base the "branch -d" safety, whose purpose is not to lose commits that are not merged to other branches, on the current branch. It is much more sensible to check if it is merged with the other branch it merges with.

Hence, there are two—or sometimes three, if you look at the above commit—questions that you (or Git) must answer to see whether `-d` is permitted:

1. What is the upstream name for the branch we propose to delete? (If none, see below.)
2. The branch name points to one specific commit (let's call this *T* for Tip). The upstream branch name also points to one specific commit (call this commit *U*).

   Is *T* an ancestor of *U*?

If the answer to question 2 is yes (i.e., $T \le U$), the deletion is permitted.

What if there *is* no upstream? Well, that's where "changed in 1.7" comes in: the original test was not $T \le U$ but rather $T \le HEAD$. That test is *still in there*. Right now, if there is no upstream, the HEAD test is used instead of the upstream test. Meanwhile, for a "transition period" while everyone adjusts to the newfangled Git 1.7 behavior, you may also get a warning, or extra explanation, when there *is* an upstream. This warning persists even today, in Git 2.10 (almost 7 years later now: this is a very long transition period!):

```
if ((head_rev != reference_rev) &&
    in_merge_bases(rev, head_rev) != merged) {
        if (merged)
            warning("deleting branch ... not yet
merged to HEAD.");
        else
            warning("not deleting branch ... even
though it is merged to HEAD.");
}
```

(I trimmed some of the code for display purposes here): basically, if `HEAD` resolves to some other commit than the commit we used in the *T* ≤ test, *and* we'd get a *different* result for *T* ≤ *HEAD* than we got for *T* ≤ *U*, we add the extra warning message. (Note that the first part of the test is redundant: if we compared to HEAD because of the pre-1.7 compatibility and a missing upstream, then we'll get the *same* result if we compare to HEAD again. All we really need is the `in_merge_bases` test.)

How do you know what the upstream is? Well, there's actually an easy command line way to find out:

```
$ git rev-parse --abbrev-ref master@{u}
origin/master
$ git rev-parse --symbolic-full-name master@{u}
refs/remotes/origin/master
```

The `--abbrev-ref` variant gives you Git's typical abbreviated version, while `--symbolic-full-name` gives you the full reference name. Both fail if run on a branch with no upstream set. Of course, you can also use `git branch -vv` to see the abbreviated upstreams (for all branches).

How do you test this whole $T \leq U$ thing? The `git merge-base --is-ancestor` command does that for shell scripts, so:

```
$ git merge-base --is-ancestor master
origin/master && echo yes || echo no
```

will tell you if `master` is an ancestor of `origin/master` (and for this purpose, "points to the same commit" counts as "is an ancestor", i.e., this is that very same ≤ test).

Whenever this transition period finally ends, the "use HEAD" test might go away entirely, or might continue be used for branches that have no upstream set. In any case, though, the question is always "is the to-be-deleted branch merged *into* ____? (fill in the blank)" and you must look at what fills the blank.

The commit(s) on the upstream that correspond to the commits on the branch you are proposing to delete must be (or at least have, in their history) *the same* commit, by commit hash ID, for that "is merged into" test to succeed. If `feature/X` was merged into `origin/develop` via a so-called "squash merge" (which is not a merge, though it's done by merging[2]), that commit ID won't match, and the "is merged into" test will always fail.

---

[1]Incidentally, as of Git 2.3, you can now add `--force` to `git branch -d`, instead of being required to use `git branch -D`, although of course `-D` still works.

[2]The distinction here is between "a merge"—merge as a noun, which means "a commit that is a merge commit" (which uses merge as an adjective), and "to merge"—merge as a verb, meaning the action of combining some sets of changes. The `git merge` command can:

- do a fast forward, which neither merges-as-a-verb nor makes a merge-as-a-noun, so that there is no merge at all; or
- do a real merge, which *merges* (verb) some changes to produce *a merge* (noun); or
- do a "squash merge", which merges (verb) but produces a non-merge (which, for some inexplicable reason, you must manually commit).

A "squash merge" happens only if you ask for it, while a "fast-forward non-merge" happens only if it can *and* you don't *prohibit* it.

Share Improve this answer Follow

answered Nov 9, 2016 at 22:45

torek
485k ● 68 ● 735 ● 860

2 This is all helpful information, but I wish it provided a guide on how to solve (or at least ameliorate) the problem described in the question. – Alex Johnson Nov 27, 2018 at 18:58

2 @AlexJohnson: the problem is that *there is no solution*. – torek Nov 27, 2018 at 19:13

Fair enough. It's just surprising to me that there's not a better workflow for the extremely common situation of deleting a feature branch locally and remotely after merging into upstream. – Alex Johnson Nov 27, 2018 at 19:18

3 @AlexJohnson: If you use a real merge (including GitHub's "merge" button in true merge mode), `git branch -d` locally will work (after `git fetch` -ing and updating your other local branches). If you use GitHub's "rebase and merge" or "squash and merge", you're stuck with forcing the delete. It's not very pretty. – torek Nov 27, 2018 at 19:20

2 I squashed before that commit was merged, so that makes sense. I guess I have to choose whether it's pretty when merging (squash) or when deleting (non-forced delete). Thanks for the explanation. – Alex Johnson Nov 27, 2018 at 19:23

7

This happens when the content of the commits differ enough from the branch you are attempting to delete. This could be because of a squash commit or a branch that is attached to a previous base then when they were merged by someone else on the remote.

If you feel confident that the changes were in fact merged (as you said) then a `branch -D` is safe.

Lastly,

> disastrously deletion of branches

Is untrue. Deleting a branch is not an undo-able issue. All actions are stored in the `git reflog` and the commits that a branch pointed to stay for at least 30 days. If truly paranoid you could rename the branch instead. But in practice it is easier to just delete the branch.

In other words a branch s just a pointer. Deleting the pointer does not delete the content. And so you can easily resurrect the branch if you needed.

Share Improve this answer Follow

answered Nov 9, 2016 at 20:27

Sukima
10.1k ● 3 ● 48 ● 60

2 Be careful with deletion, because deleting a branch name also deletes that branch's reflog. The `HEAD` reflog may still have entries for the commits that the branch's reflog would protect, but that's *may* (as in might), not necessarily *does*. In general, though, it's true that the objects themselves tend to persist for some time: this happens even if they are unprotected. You would have to unprotect them *and* trigger (or manually run) `git gc` or `git prune` to have them go away. – torek Nov 9, 2016 at 21:47