Make Git show the correct tag on branches with git describe

Asked 5 years, 3 months ago Modified 4 years, 9 months ago

Viewed 11k times

9

`git describe` is documented to find

> the most recent tag that is reachable from a commit.

Source: `git describe --help`

I am a little lost understanding how exactly a tag is reachable from a commit. When run in a branch I'm not seeing the behavior I expect and I don't understand why.

https://github.com/nodemcu/nodemcu-firmware uses a release scheme where all changes go into the `dev` branch, which is then snapped back to `master` regularly. Releases and annotated tags are created from `master`. `git describe` run on `master` produces the expected result.

When run on `dev` I get a tag created over two years ago.

```
~/Data/NodeMCU/nodemcu-firmware (dev) > git
describe
2.0.0-master_20170202-439-ga08e74d9
```

Why is this?

A similar situation is with a (more or less) frozen branch of an old version we keep around for certain users.

```
~/Data/NodeMCU/nodemcu-firmware (1.5.4.1-final) >
git describe
0.9.6-dev_20150627-953-gb9436bdf
```

That branch was created after this annotated tag https://github.com/nodemcu/nodemcu-firmware/releases/tag/1.5.4.1-master_20161201 and only a handful of commits landed on the branch since.

git **github** **git-describe**

Share

Improve this question

Follow

asked Aug 28, 2019 at 5:11

Marcel Stör

23.5k ● 20 ● 102 ● 208

Does the tag that you are looking for exist in the local repository? – ElpieKay Aug 28, 2019 at 6:47

The other tags might not be reachable? – evolutionxbox Aug 28, 2019 at 7:16

Yes, all tags are present in the cloned local repo. I see it when I do `git describe` on `master`. – Marcel Stör Aug 28, 2019 at 7:17

2 Answers Sorted by: Highest score (default)

32

The documentation is a lie. You cannot find a tag from a commit. You *can* find a commit from a tag, and that's what `git describe` *really* does, in a remarkably twisty manner, as we'll see in a moment.

The lie is *meant* to be a useful, descriptive lie. How successful it is at that is, I think, open to question. Let's take a look at how `git describe` really works (without getting too deep into details). First, though, we might need some background.

Background (if you know all this, skip to the next section)

What you need to know before we start:

- There are two "kinds" of tags: annotated tags, and lightweight tags. An annotated tag is one made by `git tag -a` or `git tag -m`, and actually has two parts: it's a lightweight tag *plus* an actual Git object, which we'll get into in a moment.

  By default, `git describe` looks only at annotated tags. Using `--tags` makes it look at *all* tags.

- Tags are a specific form of a more general entity, the *reference* or *ref* for short. Branch names like `master` are also references, and `git describe` is allowed to use any reference, via `--all`.

- You can also find a commit from a commit, using the *commit graph*.

Underpinning all of the above, Git has both *references* and *objects*. These are stored in two separate databases.[1] One stores names, all of the form `refs/...`, which map to a hash value (SHA-1 currently, though SHA-256 is being planned). The other is a simple key-value store indexed by hash values. So:

```
       refs                                  objects
+---------------------------------+    +-----------
-----------+
| refs/heads/master   a123456... |    | 08aef31...
<object> |
| refs/tags/v1.2      b789abc... |    | a123456...
<object> |
+---------------------------------+    | b789abc...
<object> |
                                       | <lots more
of these> |
                                       +-----------
-----------+
```

The object database is usually much, much bigger than the reference database.

There are actually four kinds of objects in it: *commit* objects, *tree* objects, *blob* objects, and *tag* objects. Every object has an *object ID* or OID, which is really just a hash (again, currently SHA-1, eventually SHA-256; the idea behind calling it an OID is to insulate from the eventual changeover). *Blob* objects hold data that Git itself does

not interpret.[2] All others hold data that Git does at least *something* with.

Commit and tag objects are the ones that are particularly interesting here, because tag objects contain an OID that is the *target* of the tag, and commit objects contain an OID for each *parent* of the commit.

Commit refs (`refs/heads/master` and the like) are constrained to contain only OIDs of commit objects. Commit objects' parent OIDs are likewise constrained: each one must be the OID of another commit object. The parent(s) of any commit are some older commit(s) that existed when that particular commit got created.

If we were to look through all the objects in the repository (as, e.g., `git gc` and `git fsck` do), we could build a *graph* of all the commit objects, with one-way arrows linking from each commit to all of its parents. If we zoom in on one particular two-parent commit, we might see:

```
... <commit>  <--  +--------+
                   | commit |  <-- <commit> ...
... <commit>  <--  +--------+
```

Zooming back out, we see an overall *directed acyclic graph* or **DAG** of all commits. Meanwhile the OIDs stored in the branch names—and in any other reference that holds a *commit* hash—act as *entry points* into this graph, places where we can start and then keep following the parent links.

An *annotated tag* is a tag reference—a lightweight tag, more or less—that points to a tag object. If the underlying tag object then points to a commit, that too acts as an entry point to the commit DAG. Tag objects are allowed to point directly to trees or blobs, or to other tag objects, though. The process of *peeling* a tag refers to following an annotated tag that points to another tag object. We just keep following until we reach some non-tag object: that's the final target of this layered tag. If that final target is a commit, that's yet another entry point into the DAG.

So, in the end, we typically have a *branch name* like `master` that points to the *last* commit in a mostly-linear string of commits:

```
... <-o <-o <-o <-o   <--master
```

The fact that the internal arrows all point backwards is usually not terribly interesting, although it affects `git describe`, so I've included it here.

At various times during the repository's lifetime, we pick a commit and add a *tag* to it, either lightweight or annotated. If it's an annotated tag there's an actual tag object:

```
  tag:v1.1  tag:v1.2
      |         |
      v         v
      T         T
      |         |
```

```
      v       v
... <-o <-o <-o <-o   <--master
```

where the `o`s are commit objects and the `T`s are tag objects.

---

[1]The reference database is pretty cheesy: it's really just a flat file, `.git/packed-refs`, plus a bunch of individual files and sub-directories, `.git/refs/**/*`. Still, internally in Git, there is a plug-in interface for adding new databases, and given all the issues with the flat-file and individual files, I expect that in time, there will be a real database as an option.

[2]Mostly, that's your own file data. With symbolic links for instance the target of the symlink is stored as a blob object, so the data then get interpreted by your host OS later.

How `git describe` works

The `git describe` command wants to find some name—usually, some annotated tag object—such that the commit you're asking to describe is a *descendant* of the tagged commit. That is, the tag could point directly to commit X, or to a commit that is the immediate parent of X (one step back), or to a commit that is some number of steps back from X, hopefully not too many.

In Git, it is very difficult find *descendants* of some particular commit. But it is easy to find *ancestors* of some

particular commit. So instead of starting at each tag and working *forwards*, Git has to start at commit X and work *backwards*. Is X itself described by some tag? If not, try each of X's parents: are they the direct target of some tag? If not, try each of X's grandparents: are they the direct target of some tag?

So `git describe` it finds the targets of all, or at least some number of, interesting references (annotated tags, or all tags, or all references). When it does this "interesting refs" on our example, it finds two commits, which we'll mark with `*`:

```
  tag:v1.1  tag:v1.2
      |         |
      v         v
      T         T
      |         |
      v         v
... <-* <-o <-* <-o   <--master
```

Now it starts at the commit we want described: the tip of `master`. From that commit, it can work backwards *one* hop to reach the commit that's starred from `v1.2`. Or, it can work backwards *three* hops to find the commit that's starred from `v1.1`.

Since `v1.2` is "closer", that's the annotated tag name that `git describe` will use. Meanwhile, it *did* have to walk one hop back from `master`. So the output will be:

```
v1.2-1-g<hash>
```

where is the abbreviated OID of the commit to which `master` points.

This diagram—both the graph itself, and the two annotated tags—is *extremely* simple. Most real graphs are terribly knotted, due to branching and merging. Even if we just draw another *fairly* simple one, we can get things like this:

```
                  tag-A       tag-B
                    v           v
         o--o--...--o           o--o   <-- branch1
        /            \         /
...-o--o              o--...--o--o   <-- branch2
        \            /
         o--o--...--o
                ^
             tag-C
```

In this case, tag-A is going to be "closer" to the tip of `branch2`, and should be what `git describe` picks. The actual algorithm inside `git describe` is pretty complicated and it's not clear to me which tag it picks in some of the trickier cases: Git doesn't have an easy way to load the whole graph and do a breadth-first search and the code is very ad hoc. However, it *is* clear that `tag-B` is not suitable, as it points to a commit that *cannot* be reached by starting at `branch2` and working backwards.

Now we can look more closely at your last example. I cloned the repository and did this:

```
$ git log --decorate --graph --oneline
origin/1.5.4.1-final 1.5.4.1-master_20161201
```

```
* b9436bdf (origin/1.5.4.1-final) Replace
unmainted Flasher with NodeMCU PyFlasher
* 46028b25 Fix relative path to firmware sources
* 6a485568 Re-organize documentation
* f03a8e45 Do not verify the Espressif BBS cert
* 1885a30b Add note about frozen branch
* 017b4637 Adds uart.getconfig(0) to get the
current uart parameters (#1658)
* 12a7b1c2 BME280: fixing humidity trimming
parameter readout bug (#1652)
* c8176168 Add note about how to merge master-drop
PRs
* 063cb6e7 Add lua.cross to CI tests. (#1649)
* 384cfbec Fix missing dbg_printf (#1648)
* 79013ae7 Improve SNTP module: Add list of
servers and auto-sync [SNTP module only] (#1596)
* ea7ad213 move init_data from .text to
.rodata.dram section (#1643)
* 11ded3fc Update collaborator section
* 9f9fee90 add new rfswitch module to handle
433MHZ devices (#1565)
* 83eec618 Fix iram/irom section contents (#1566)
* 00b356be HTTP module can now chain requests
(#1629)
* a48e88d4 EUS bug fixes (#1605)
| *   81ec3665 (tag: 1.5.4.1-master_20161201)
Merge pull request #1653 from nodemcu/dev-for-drop
| |\
| |/
|/|
* | 85c3a249 Fix Somfy docs
* |   016f289f Merge pull request #1626 from tae-
jun/patch-2
|\ \
| * | 58321a92 Fix typo at rtctime.md
|/ /
```

Note that commit `b9436bdf`, the tip of `origin/1.5.4.1-final`, *does not* have commit `81ec3665` as an ancestor. Tag `1.5.4.1-master_20161201` points to object `4e415462` which is an annotated tag object that in turn points to commit `81ec3665`:

```
$ git rev-parse 1.5.4.1-master_20161201
4e415462bc7dbc2dc0595a8c55d469740d5149d6
$ git cat-file -p 1.5.4.1-master_20161201
object 81ec3665cb5fe68eb8596612485cc206b65659c9
...
```

The tag you were hoping to find, `1.5.4.1-master_20161201`, is *not* eligible to describe commit `b9436bdf`. There are *no* commits in this particular graph that are descendants of commit `81ec3665`.

Using `git log --all --decorate --oneline --graph`, I find that there are some such commits in the full graph, e.g., `b96e3147`:

```
* | | e7f06395 Update to current version of SPIFFS
(#1949)
| | *   c8ac5cfb (tag: 2.1.0-master_20170521)
Merge pull request #1980 from node mcu/dev
| | |\
| |_|/
|/| |
* | |   787379f0 Merge branch 'master' into dev
|\ \ \
| | |/
| |/|
| * | 22e1adc4 Small fix in docs (#1897)
| * |   b96e3147 (tag: 2.0.0-master_20170202)
Merge pull request #1774 from node mcu/dev
| |\ \
| * \ \   81ec3665 (tag: 1.5.4.1-master_20161201)
Merge pull request #1653 from nodemcu/dev-for-drop
| |\ \ \
| * | | | ecf9c644 Revert "Next 1.5.4.1 master
drop (#1627)"
```

but `b96e3147` itself has its own (annotated) tag, so that's what `git describe` should and does list:

```
$ git describe b96e3147
2.0.0-master_20170202
```

Ultimately the issue here is that there is not a simple "ancestor / descendant" relationship between any given pair of commits. *Some* commits do have such relationships. Others are merely siblings: they have some common ancestor. Still others may have *no* common ancestor, if you have a graph with multiple root commits.

In any case, `git describe` normally needs to work *against* the direction of the internal arrows: it must find a tagged commit such that the to-be-described commit is a descendant of that tag. It literally *can't* do that, so it transforms the problem into one that it can do: find some tagged commit, out of the set of all tagged commits, such that the tagged commit is an ancestor of the desired commit—then, count the number of hops it takes to move backwards from the desired commit to this tagged commit.

edited Jun 20, 2020 at 9:12

Community Bot

1 ● 1

answered Aug 28, 2019 at 16:58

torek

485k ● 68 ● 735 ● 860

3 Wow, just wow! Way more than I could digest in one pass :) In a deleted answer (not all of you can see those) someone suggested "If you want better git describe results in your feature branches such as dev, you should fork dev branch after each new tag." – Marcel Stör Aug 28, 2019 at 21:19

3 Absolutely phenomenal answer! – knite Dec 1, 2020 at 20:43

I'm going to keep a link to this! – Erik Nov 7 at 7:46

4

What is reachable has changed with Git 2.26 (Q1 2020) (v2.26.0-rc0 onwards): "`git describe`" in a repository with multiple root commits sometimes gave up looking for the best tag to describe a given commit with too early, which has been adjusted.

> ## `describe`: don't abort too early when searching tags
>
> Signed-off-by: Benno Evers
>
> Signed-off-by: Junio C Hamano
>
> When searching the commit graph for tag candidates, `git-describe` will stop as soon as there is only one active branch left and it already found an annotated tag as a candidate.
>
> This works well as long as all branches eventually connect back to a common root, but if the tags are found across branches with no common ancestor

```
            B
            o----.
                  \
o-----o---o----x
A
```

it can happen that the search on one branch terminates prematurely because a tag was found on another, independent branch.
This scenario isn't quite as obscure as it sounds, since cloning with a limited depth often introduces many independent "dead ends" into the commit graph.

The help text of `git-describe` states pretty clearly that when describing a commit `D`, the number appended to the emitted `tag X` should correspond to the number of commits found by `git log X..D`.

Thus, this commit modifies the stopping condition to only abort the search when only one branch is left to search *and* all current best candidates are descendants from that branch.

For repositories with a single root, this condition is always true: When the search is reduced to a single active branch, the current commit must be an ancestor of *all* tag candidates.
This means that in the common case, this change will have no negative performance

> impact since the same number of commits as before will be traversed.

Share Improve this answer

Follow

edited Mar 9, 2020 at 20:36

Marcel Stör

23.5k ● 20 ● 102 ● 208

answered Mar 9, 2020 at 17:16

VonC

1.3m ● 558 ● 4.7k ● 5.6k