Git is about commits

Source: https://stackoverflow.com/questions/72429205/git-rebase-vs-checkout

The basic unit of storage in Git is the **commit**. A Git repository is a collection of commits, stored in a big database that Git calls the object database. A Git repository has several more parts, which we'll get to in a moment, but this first one—the object database—is essential: without it there's no repository.

The object database is a simple key-value store, using what Git calls OIDs or Object IDs to look up the objects. The most important kind of object for our purposes—in fact, the only one we really care about—is the commit object, which holds the first part of any commit. So our commits, in Git, have these OIDs. We'll call them **hash IDs**. Some call them SHA or SHA-1, because Git initially (and currently) uses the SHA-1 cryptographic hash as its hash IDs, but Git is no longer wedded to SHA-1, so "hash ID" or "OID" is more appropriate.

An OID or hash ID is a big ugly string of letters and digits, such as e54793a95afeea1e10de1e5ad7eab914e7416250. This is actually a very large number, expressed in hexadecimal. Git needs these to find its objects. The ID is unique to that particular object: no other object, in the big objects database, can have that ID. Every commit you make has to get a new random-looking number, never-before-used, never to be used again ever, in any Git repository, unless it's being used to store your commit. Making this actually work is hard—technically, it's impossible—but the sheer size of the hash ID makes it work in practice. A Git doomsday may come someday but it won't be for a while yet.

Git is not about branches or files

If Git commits did not store files, Git would be useless. So commits do store files. But commits are not files themselves, and a file is not Git's "unit of work" as it were. Git is about the commits, which sort of accidentally-on-purpose contain files.

The word **branch**, in Git, is very badly overused, almost to the point of meaninglessness. There are at least two or three things people mean when they say branch here, and it can get very confusing.

To help keep this straight, I like to (try at least) to use the phrase branch name when referring to a name like **main** or master, dev or develop, feature, and so on. **A branch name, in Git, is a fast and important way to find one particular commit**. Humans use these because human brains are no good at working with hash IDs: they're too big, ugly, and random-looking.

A repository, therefore, keeps a separate database—another simple key-value store—in which each key is a name and the value is the big ugly hash ID that goes with that name. Branch

names are one of the many kinds of names that Git sticks in this second database. So, you can give Git a branch name; Git will look up the hash ID, and find the latest commit for that branch.

In this sense, we use branches—or more precisely, branch names—in Git to get to our commits. But Git isn't about these branches, really; it's still about the commits.

What's in a commit

Now that we know Git is all about commits, let's take a look at an actual raw commit. Here's the one I referred to above:

```
$ git cat-file -p e54793a95afeea1e10de1e5ad7eab914e7416250
tree dc3d0156b95303a305c69ba9113c94ff114b7cd3
parent 565442c35884e320633328218e0f6dd13f3657d3
author Junio C Hamano <gitster@pobox.com> 1651786597 -0700
committer Junio C Hamano <gitster@pobox.com> 1651786597 -0700

Git 2.36.1

Signed-off-by: Junio C Hamano <gitster@pobox.com>
That's the raw commit object, and it actually consists entirely of the commit's metadata.
```

A commit object has two parts:

- Every commit has a full snapshot of all of the files that make up that particular commit. In a real commit like the one above, that's the tree line, which is required: there must be one and only one tree.
- Every commit also has some metadata. That's the entire chunk of text above, really (including the tree line itself).

Note that the metadata tells us who made the commit, and when: the magic number 1651786597 above is a date-and-time-stamp meaning Thu May 5 14:36:37 2022. The -0700 is the time zone, which in this case is Pacific Daylight Time or UTC-7. It also has the committer's commit message, which in this case is remarkably short: compare with, e.g., a snippet from f8781bfda31756acdc0ae77da7e70337aedae7c9:

```
2.36 gitk/diff-tree --stdin regression fix

This only surfaced as a regression after 2.36 release, but the
breakage was already there with us for at least a year.

The diff_free() call is to be used after we completely finished with
a diffopt structure.  After "git diff A B" finishes producing
output, calling it before process exit is fine.  But there are
commands that prepares diff_options struct once, compares two sets
of paths, releases resources that were used to do the comparison,
```

```
then reuses the same diff_option struct to go on to compare the next
two sets of paths, like "git log -p".

After "git log -p" finishes showing a single commit, calling it
before it goes on to the next commit is NOT fine.  There is a
mechanism, the .no_free member in diff_options struct, to help "git
log" to avoid calling diff_free() after showing each commit and ...
```

which is a much better commit message.

The other really important part of the metadata, which Git sets up on its own, is the parent line. There can be more than one parent line—or, rarely, no parent line—because each commit carries, in its metadata, a list of parent hash IDs. These are just the raw hash IDs of some existing commits in the repository, that were there when you, or Junio, or whoever, added a new commit. We'll see in a moment what these are for.

Review so far

A repository has two databases

- One (usually much bigger) contains commits and other objects. These have hash IDs; Git needs the hash IDs to find them.
- The other (usually much smaller) contains names, such as branch and tag names, and maps each name to one hash ID. For a branch name, the one hash ID we get here is, by definition, the latest commit for that branch.

The commits are the reason that all of this exists. Each one stores two things: a full snapshot, and some metadata.

A working tree

Now, one of the tricks to making the hash IDs work, in Git, is that no part of any object can ever change. A commit, once made, is the way it is forever. That commit, with that hash ID, holds those files and that metadata and thus has that parent (or those parents) and so on. Everything is frozen for all time.

The files inside a commit are stored in a special, read-only, compressed (sometimes highly compressed), de-duplicated format. That avoids having the repository bloat up even though most commits mostly re-use most of the files from their parent commit(s). Because the files are de-duplicated, the duplicates literally take no space. Only a changed file needs any space.

But there's an obvious problem:

- Only Git can read these compressed-and-de-duplicated files.
- Nothing, not even Git itself, can write them.

If we're going to get any work done, we must have ordinary files, that ordinary programs can both read and write. Where will we get those?

Git's answer is to provide, with any non-bare repository, an area in which you can do your work. Git calls this area—a directory-tree or folder full of folders, or whatever terminology you like—your working tree, or work-tree for short. In fact, the typical setup is to have the repository proper live inside a **hidden .git directory** at the top level of the working tree. Everything inside this is Git's; everything outside it, at the top level of the working tree and in any sub-directory (folder) within it other than .git itself, is yours.

What git checkout or git switch is about

When you *check out* some commit—with git checkout or git switch and a branch name—you're telling Git:

- Use the branch name to find the latest commit by hash ID.
- Remove, from my working tree, all the files that came out of whatever commit I've been using.
- Replace, into my working tree, all the files that come out of the commit I just named.

Git takes a big short-cut here when it can: if you're moving from commit a123456 to b789abc, and most of the files in those two commits are de-duplicated, Git won't actually bother with the remove-and-replace for these files. This short-cut becomes important later, but if you start out thinking of git checkout / git switch as meaning: *remove the current commit's files, change to a new current commit, and extract those files* you have a good start.

How commits get strung together

Let's revisit the commit itself for a bit now. Each commit has, in its metadata, some set of parent lines. Most commits (by far in most repositories) have exactly one parent and that's the thing to start with.

Let's draw the commits in a simple, tiny, three-commit repository. The three commits will have three big ugly random-looking hash IDs, but rather than make some up, let's just call them commits A, B, and C in that order. Commit A was the very first commit—which is a bit special because it has no parent commit—and then you made B while using commit A, and made C while using B. So we have this:

```
A <-B <-C
```

That is, commit C, the latest commit, has some files as its snapshot, and has, as its parent, the raw hash ID of commit B. We say that C points to B.

Meanwhile, commit B has some files as its snapshot, and has commit A as its parent. We say that B points to A.

Your branch name, which we'll assume is main, points to the latest commit C:

```
A--B--C   <-- main
```

(here I get lazy about drawing the arrows between commits as arrows, but they're still backwards-pointing arrows, really).

When you git checkout main, Git extracts all the commit-C files into your working tree. You have those files available to view and edit.

If you do edit some, you use git add and git commit to make a new commit. This new commit gets an all-new, never been used before anywhere in any Git repository in the universe, hash ID, but we'll just call this new commit D. Git will arrange for new commit D to point backwards to existing commit C, because C is the one you've been using, so let's draw in new commit D:

```
A--B--C   <-- main
       \
        D
```

(The backwards slash going up-and-left from D to C is why I get lazy about the arrows, so we just have to imagine the arrow from D to C.)

But now D is the latest main commit, so git commit also stores D's hash ID into the name main so that main now points to D:

```
A--B--C
       \
        D   <-- main
```

(and now there's no reason to use extra lines to draw things; I just kept it for visual symmetry).

This is one way a branch grows, in Git. You check out the branch, so that it's your current branch. Its tip-most commit—the one towards the right in this drawing, or towards the top in git log --graph output—becomes your current commit and those are the files you see in your working tree. You edit those files, use git add, and run git commit, and Git packages up the new files—with automatic de-duplication, so that if you change a file back to the way it was in B or A, it gets de-duplicated here!—into a new commit, then stuffs the new commit's hash ID into the current branch name.

How branches form

Let's say we start out with that same three-commit repository:

```
A--B--C   <-- main
```

Let's now create a new branch name dev. This name must point to some existing commit. There are only three commits, so we have to pick one of A, B, or C, for the name dev to point-to. The obvious one to use is the most recent: we probably don't need to go back in time to commit B or A to start adding new commits. So let's add dev so that it also points to C, by running:

git branch dev

We get:

```
A--B--C   <-- dev, main
```

It's hard to tell from our drawing: are we on dev or main? That is, if we run git status, which will it say, "on branch dev" or "on branch main"? Let's add a special name, HEAD in all uppercase like this, and attach it to one of the two branch names, to show which name we are using:

A--B--C <-- dev, main (HEAD)

We are "on" branch main. If we make a new commit now, commit D will point back to commit C as usual, and Git will stick the new hash ID into the name main.

But if we run:

git checkout dev

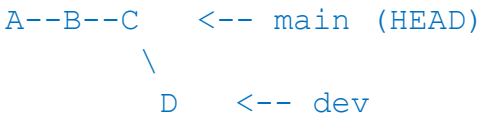
Git will remove, from our working tree, all the commit-C files, and put in all the commit-C files instead. (Seems kind of silly, doesn't it? Short-cut! Git won't actually do any of that!) Now we have:

```
A--B--C   <-- dev (HEAD), main
```

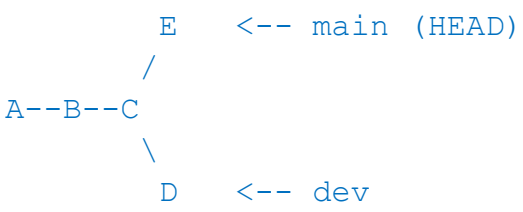
and when we make our new commit D we get:

```
A--B--C   <-- main
       \
        D   <-- dev (HEAD)
```

If we git checkout main, Git will remove the commit-D files and install the commit-C files, and we'll be back to:



and if we now make another new commit we will get:



This is how branches work in Git. A branch name, like main or dev, picks out a last commit. From there, Git works backwards. Commit E might be the last main commit, but commits A-B-C are on main because we get to them when we start from E and work backwards.

Meanwhile, commit D is the last dev commit, but commits A-B-C are on dev because we get to them when we start from D and work backwards. Commit D is not on main because we never reach commit D when we start from E and work backwards: that skips right over D.

Review

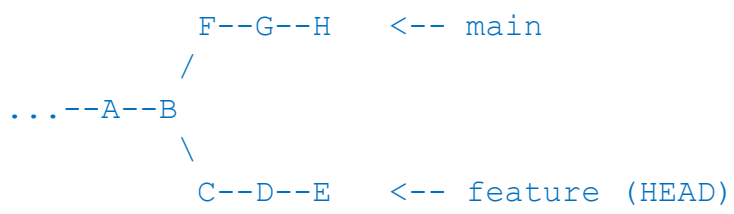
We now know:

- Git is about commits.
- Commits store snapshots and metadata.
- We organize the commits into branches using branch names to find the last commit.
- We check out a commit to see its files as files, and to work on them. Otherwise they're special weird Gitty things that only Git can see.
- No part of any commit can ever change, once it's made.

Now we'll get to git rebase.

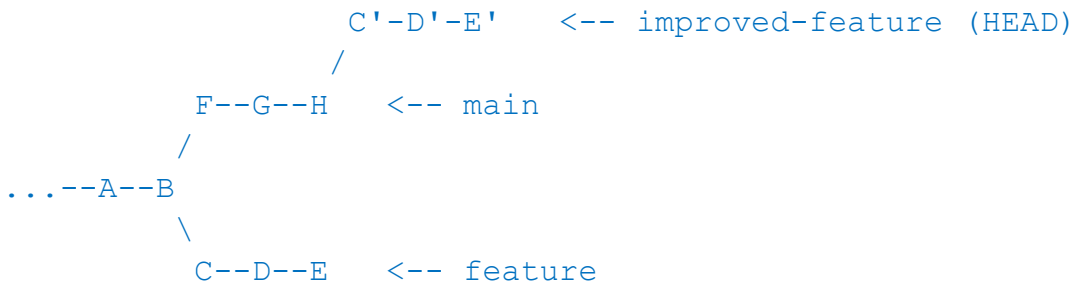
What git rebase is about

We often find ourselves using Git and stuck in this kind of situation:



and we say to ourselves: Gosh, it would be nice if we had started out feature later, when main had commit G and/or H in it, because we need what's in those now.

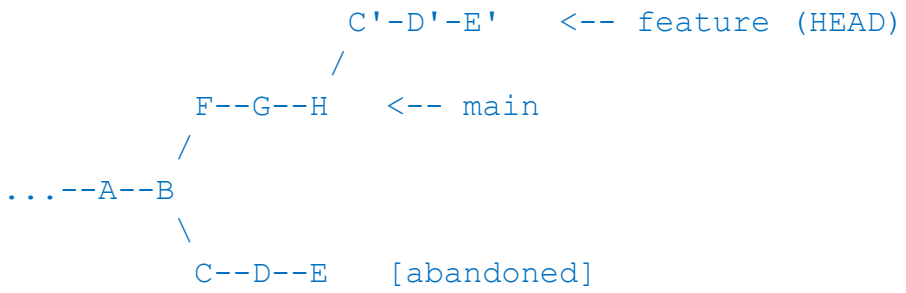
There's nothing fundamentally wrong with commits C-D-E and we could just use git merge, but for whatever reason—the boss says so, the co-workers have decided they like a rebase flow, whatever it might be—we decide that we're going to "improve" our C-D-E commits. We're going to re-make them so that they come after F-G-H, like this:



We can, quite literally, do this by check out commit H, making a new branch, and then re-doing our work:

```
git switch main
git switch -c improved-feature
... redo a bunch of work ...
```

What git rebase does is automate this for us. If we were to do it manually, each "redo" step would involve using git cherry-pick (which I won't go into in any detail here). The git rebase command automates the cherry-picking for us, and then adds one other twist: instead of requiring a new branch name like improved-feature, it simply yanks the old branch name off the old commits and makes it point to the new ones:



The old abandoned commits are actually still there, in Git, for at least 30 days or so. But with no name by which to find them, you can only see those commits if you have saved their hash IDs, or have some trick by which to find those hash IDs

When the rebase finishes completely, our original commits are copied to new-and-improved commits. The new commits have new and different hash IDs, but since no human ever notices the actual hash IDs, a human who looks at this repository just sees three feature-branch-only commits and assumes they have magically been changed into the new improved ones.

Comments

- You're missing the core concept of how a commitish relates to a treeish, and how Git tracks history. Blobs and trees may or may not change, but while Git is a DAG it isn't an immutable blockchain. You can make modifications anywhere within the DAG and the ancestry (and thus the "history") will change as well. This is important to understanding Git, just as it's important to understand that Git tracks content rather than files or directories as first-class items. Conceptually, most commits are just a snapshot of pointers to collections of hashes at a given checkpoint. – Todd A. Jacobs
- @ToddA.Jacobs: I'm as error-prone as anyone, but I think I have covered that above, particularly with the part about rebase. When we rebase, we change the commit selected by some name. That changes which commits we see in the DAG, when we choose the name. The commit-ish vs tree-ish is also covered above: a commit represents a tree and there's a one-to-one mapping from commit to tree (but not vice versa, it's a surjection from commit to tree, not a bijection). – torek