gitdatamodel

gitdatamodel(7) ==========

NAME ---- gitdatamodel - Git’s core data model

DESCRIPTION -----------

It’s not necessary to understand Git’s data model to use Git, but it’s very helpful when reading Git’s documentation so that you know what it means when the documentation says "object", "reference" or "index".

Git’s core operations use 4 kinds of data:

1. Objects: commits, trees, blobs, and tag objects
2. References: branches, tags, remote-tracking branches, etc
3. The index, also known as the staging area
4. Reflogs: logs of changes to references ("ref log")

OBJECTS -------

All of the commits and files in a Git repository are stored as "Git objects". Git objects never change after they’re created, and every object has an ID, like `1b61de420a21a2f1aaef93e38ecd0e45e8bc9f0a`.

This means that if you have an object’s ID, you can always recover its exact contents as long as the object hasn’t been deleted.

Every object has:

1. an **ID** (aka "object name"), which is a cryptographic hash of its type and contents. It’s fast to look up a Git object using its ID. This is usually represented in hexadecimal, like `1b61de420a21a2f1aaef93e38ecd0e45e8bc9f0a`.
2. a **type**. There are 4 types of objects: commits, trees, blobs, and tag objects.
3. **contents**. The structure of the contents depends on the type.

Here’s how each type of object is structured:

commit
A commit contains these required fields (though there are other optional fields):

1. The full directory structure of all the files in that version of the repository and each file's contents, stored as the **tree** ID of the commit's top-level directory
2. Its **parent commit ID(s)**. The first commit in a repository has 0 parents, regular commits have 1 parent, merge commits have 2 or more parents
3. An **author** and the time the commit was authored
4. A **committer** and the time the commit was committed
5. A **commit message**
   Here's how an example commit is stored:

   ```
   tree 1b61de420a21a2f1aaef93e38ecd0e45e8bc9f0a
   parent 4ccb6d7b8869a86aae2e84c56523f8705b50c647
   author Maya <maya@example.com> 1759173425 -0400
   committer Maya <maya@example.com> 1759173425 -0400

   Add README
   ```

   Like all other objects, commits can never be changed after they're created. For example, "amending" a commit with `git commit --amend` creates a new commit with the same parent.
   Git does not store the diff for a commit: when you ask Git to show the commit with linkgit:git-show[1], it calculates the diff from its parent on the fly.

tree
A tree is how Git represents a directory. It can contain files or other trees (which are subdirectories). It lists, for each item in the tree:

1. The **filename**, for example `hello.py`
2. The **file type**, which must be one of these five types:
   - **regular file**
   - **executable file**
   - **symbolic link**
   - **directory**
   - **gitlink** (for use with submodules)
3. The **object ID** with the contents of the file, directory, or gitlink.
   For example, this is how a tree containing one directory (`src`) and one file (`README.md`) is stored:

   ```
   100644 blob 8728a858d9d21a8c78488c8b4e70e531b659141f README.md
   040000 tree 89b1d2e0495f66d6929f4ff76ff1bb07fc41947d src
   ```

> [!NOTE] In the output above, Git displays the file type of each tree entry using a format that's loosely modelled on Unix file modes (`100644` is "regular file",

> `100755` is "executable file", `120000` is "symbolic link", `040000` is "directory", and `160000` is "gitlink"). It also displays the object's type: `blob` for files and symlinks, `tree` for directories, and `commit` for gitlinks.

blob
A blob object contains a file's contents.

When you make a commit, Git stores the full contents of each file that you changed as a blob. For example, if you have a commit that changes 2 files in a repository with 1000 files, that commit will create 2 new blobs, and use the previous blob ID for the other 998 files. This means that commits can use relatively little disk space even in a very large repository.

tag object
Tag objects contain these required fields (though there are other optional fields):

1. The **ID** of the object it references
2. The **type** of the object it references
3. The **tagger** and tag date
4. A **tag message**, similar to a commit message

Here's how an example tag object is stored:

```
object 750b4ead9c87ceb3ddb7a390e6c7074521797fb3
type commit
tag v1.0.0
tagger Maya <maya@example.com> 1759927359 -0400

Release version 1.0.0
```

> [!NOTE] All of the examples in this section were generated with `git cat-file -p <object-id>`.

REFERENCES ----------

References are a way to give a name to a commit. It's easier to remember "the changes I'm working on are on the `turtle` branch" than "the changes are in commit bb69721404348e". Git often uses "ref" as shorthand for "reference".

References can either refer to:

1. An object ID, usually a commit ID

2. Another reference. This is called a "symbolic reference"

References are stored in a hierarchy, and Git handles references differently based on where they are in the hierarchy. Most references are under `refs/`. Here are the main types:

branches: `refs/heads/<name>`
A branch refers to a commit ID. That commit is the latest commit on the branch.

To get the history of commits on a branch, Git will start at the commit ID the branch references, and then look at the commit's parent(s), the parent's parent, etc.

tags: `refs/tags/<name>`
A tag refers to a commit ID, tag object ID, or other object ID. There are two types of tags:

1. "Annotated tags", which reference a tag object ID which contains a tag message
2. "Lightweight tags", which reference a commit, blob, or tree ID directly
   Even though branches and tags both refer to a commit ID, Git treats them very differently. Branches are expected to change over time: when you make a commit, Git will update your current branch to point to the new commit. Tags are usually not changed after they're created.

HEAD: `HEAD`

`HEAD` is where Git stores your current branch, if there is a current branch. `HEAD` can either be:

1. A symbolic reference to your current branch, for example `ref:` `refs/heads/main` if your current branch is `main`.
2. A direct reference to a commit ID. In this case there is no current branch. This is called "detached HEAD state", see the DETACHED HEAD section of linkgit:git-checkout[1] for more.

remote-tracking branches: `refs/remotes/<remote>/<branch>`
A remote-tracking branch refers to a commit ID. It's how Git stores the last-known state of a branch in a remote repository. `git` `fetch` updates remote-tracking branches. When `git` `status` says "you're up to date with origin/main", it's looking at this.

`refs/remotes/<remote>/HEAD` is a symbolic reference to the remote's default branch. This is the branch that `git` `clone` checks out by default.

Other references
Git tools may create references anywhere under `refs/`. For example, linkgit:git-stash[1], linkgit:git-bisect[1], and linkgit:git-notes[1] all create their own references in `refs/stash`, `refs/bisect`, etc. Third-party Git tools may also create their own references.

Git may also create references other than `HEAD` at the base of the hierarchy, like `ORIG_HEAD`.

> [!NOTE] Git may delete objects that aren't "reachable" from any reference or reflog. An object is "reachable" if we can find it by following tags to whatever they tag, commits to their parents or trees, and trees to the trees or blobs that they contain. For example, if you amend a commit with `git commit --amend`, there will no longer be a branch that points at the old commit. The old commit is recorded in the current branch's reflog, so it is still "reachable", but when the reflog entry expires it may become unreachable and get deleted. Reachable objects will never be deleted.

THE INDEX --------- The index, also known as the "staging area", is a list of files and the contents of each file, stored as a blob. You can add files to the index or update the contents of a file in the index with linkgit:git-add[1]. This is called "staging" the file for commit.

Unlike a tree, the index is a flat list of files. When you commit, Git converts the list of files in the index to a directory tree and uses that tree in the new commit.

Each index entry has 4 fields:

1. The **file type**, which must be one of:
   - **regular file**
   - **executable file**
   - **symbolic link**
   - **gitlink** (for use with submodules)
2. The **blob** ID of the file, or (rarely) the **commit** ID of the submodule
3. The **stage number**, either 0, 1, 2, or 3. This is normally 0, but if there's a merge conflict there can be multiple versions of the same filename in the index.
4. The **file path**, for example `src/hello.py`

It's extremely uncommon to look at the index directly: normally you'd run `git status` to see a list of changes between the index and HEAD. But you can use `git ls-files --stage` to see the index. Here's the output of `git ls-files --stage` in a repository with 2 files:

```
100644 8728a858d9d21a8c78488c8b4e70e531b659141f 0 README.md
100644 665c637a360874ce43bf74018768a96d2d4d219a 0 src/hello.py
```

REFLOGS -------

Every time a branch, remote-tracking branch, or HEAD is updated, Git updates a log called a "reflog" for that reference. This means that if you make a mistake and "lose" a commit, you can generally recover the commit ID by running `git reflog <reference>`.

A reflog is a list of log entries. Each entry has:

1. The **commit ID**
2. **Timestamp** when the change was made
3. **Log message**, for example `pull:` `Fast-forward`

Reflogs only log changes made in your local repository. They are not shared with remotes.

You can view a reflog with `git` `reflog` `<reference>`. For example, here’s the reflog for a `main` branch which has changed twice:

```
$ git reflog main --date=iso --no-decorate
750b4ea main@{2025-09-29 15:17:05 -0400}: commit: Add README
4ccb6d7 main@{2025-09-29 15:16:48 -0400}: commit (initial): Initial commit
```

SEE ALSO -------- linkgit:gitglossary[7]

GIT --- Part of the linkgit:git[1] suite

□□□ #1
□ □□□ □□□ 2026-10-31 20:13:51 UTC
□ □□□ □□□ 2026-10-31 20:13:51 UTC