Git Internals: How Commits, Trees, and Refs Work

Git is not just a version control system; it's a content-addressable file system with a powerful graph-based history model. Understanding how Git works under the hood—how commits, trees, and refs operate—can demystify its seemingly complex behavior and empower developers to troubleshoot or optimize their workflows.

2. Blobs and Trees

Blob

A `blob` is a binary large object storing the contents of a file.

```
echo "hello" | git hash-object --stdin
```

This generates a SHA-1 hash and stores it in `.git/objects/`.

Tree

A `tree` represents a directory. It maps filenames to blob (file) or tree (subdirectory) objects.

```
git cat-file -p <tree-hash>
```

This shows filenames, permissions, and associated object hashes.

Example:

```
100644 blob a1b2c3...    README.md
040000 tree d4e5f6...    src
```

4. Refs and HEAD

Refs are human-readable names pointing to commit hashes:

- `refs/heads/master` → current branch pointer
- `refs/tags/v1.0` → tag pointer
- `HEAD` → points to current branch (symbolic ref)

```
cat .git/HEAD
```

Example:

```
ref: refs/heads/main
```

Changing branches updates the `HEAD` pointer.

6. DAG and History

Commits form a **Directed Acyclic Graph (DAG)**. Each commit points to its parent(s). This allows Git to:

- Rebase: move a commit subtree elsewhere.
- Merge: combine histories from multiple parents.
- Cherry-pick: reapply changes elsewhere.

---

7. Conclusion

Understanding Git internals—objects, trees, commits, and references—helps explain Git's robustness and flexibility. It's why operations like branching, merging, and rebasing are fast and efficient: Git is simply rewriting or redirecting pointers to snapshots.

For power users, tools like `git cat-file`, `git rev-parse`, and `git ls-tree` open a deeper understanding of what's going on behind the scenes.