基本的 Git 内部机制与核心操作

马浩琨

Nov 10, 2025

1 导言

大多数开发者在使用 Git 时，往往停留在 `git add`、`git commit` 和 `git push` 等高层命令层面，将 Git 视为一个神秘的「黑盒」。这种使用方式虽然高效，但在面对复杂冲突或状态异常时，却容易陷入困境。理解 Git 的内部机制不仅能帮助开发者精准排查问题，还能深化对 `reset`、`rebase` 和 `merge` 等操作的区别认知，从而建立正确的「Git 数据模型」心智模型。本文旨在通过解析 `.git` 目录结构，深入探讨 Git 的核心对象模型，并引导读者手动操作底层命令及编写简单脚本，模拟实现 `git init`、`git add` 和 `git commit` 等核心功能。本文面向有一定 Git 使用经验的中高级开发者，希望通过实践让读者真正「拥有」Git。

2 Git 的基石 —— 内容寻址文件系统

要理解 Git 的内部机制，首先需要探索 `.git` 目录的结构。执行 `tree .git` 命令后，可以看到一个典型仓库的骨架。其中，`objects` 目录是 Git 的数据存储核心，所有文件、目录和提交都存储于此；`refs` 目录则用于存储引用，包括分支和标签；`HEAD` 文件作为一个引用，指向当前所在分支；而 `index` 文件是暂存区的物理体现，以二进制形式记录文件状态。

Git 的核心在于内容寻址机制。这种机制不是通过文件名来访问数据，而是基于文件内容计算出一个唯一密钥，即 SHA-1 哈希值（未来可能过渡到 SHA-256）。具体来说，密钥的计算公式为 $Key = \text{SHA1}(\text{“blob”} + \text{文件内容长度} + \backslash 0 + \text{文件内容})$。例如，我们可以使用命令行工具手动计算一个字符串的 SHA-1 值。执行 `echo -e 'blob 16\0Hello Git World!' | openssl dgst -sha1` 或 `printf blob 16\0Hello Git World! | shasum`，输出结果便是该内容的唯一标识。内容寻址的优势在于确保数据的完整性 —— 任何微小改动都会导致密钥变化，防止数据被篡改；同时，它还支持去重，相同内容在对象库中仅存储一份。

3 Git 的核心对象模型

Git 的对象模型由三种核心类型构成：Blob、Tree 和 Commit。每种对象都承担着特定角色，并通过有向无环图（DAG）相互关联。

Blob 对象负责存储文件数据本身，但不包含任何文件名信息。我们可以通过底层命令 `git hash-object -w` 来创建并存储一个 Blob。例如，执行 `echo Hello, Git > hello.txt` 创建一个文件，然后运行 `git hash-object -w hello.txt`，该命令会输出一个 SHA-1 哈希值（如 8ab686eafeb1f44702738c8b0f24f2567c36da6d），同时将对象文件存入 `.git/objects` 目录。使用

find .git/objects -type f 可以查看新生成的文件，这验证了 Blob 的存储过程。
Tree 对象则代表一个目录结构，它存储文件名、文件模式（权限）以及指向对应 Blob 或其他 Tree 的引用。创建 Tree 对象需要先通过 git hash-object -w 生成多个 Blob，然后使用 git update-index 将这些 Blob 加入一个「假」的暂存区，最后通过 git write-tree 将当前索引状态写入一个 Tree 对象。这个过程模拟了 Git 如何组织文件系统目录。
Commit 对象用于存储提交的元数据，包括指向一个顶层 Tree 对象（代表项目快照）、父 Commit 对象（首次提交无父提交，合并提交有多个）、作者信息、提交时间戳和提交信息。我们可以基于已有的 Tree 对象，使用 echo First commit | git commit-tree <tree-sha> 来创建一个 Commit 对象。例如，如果 Tree 的 SHA-1 为 abc123，则命令会生成一个新的 Commit 哈希，这标志着一次提交的诞生。
这些对象之间的关系构成了 Git 版本历史的基础。Commit 指向 Tree，Tree 则包含多个 Blob 或子 Tree，形成一个有向无环图。这种结构确保了数据的高效存储和检索，是 Git 强大版本控制能力的核心。

4 实现核心操作 —— 从底层命令到脚本

通过底层命令模拟 Git 的核心操作，可以帮助我们更直观地理解其工作原理。首先，从 git init 开始。我们可以手动创建仓库骨架：建立 .git 目录及其子目录（如 objects、refs/heads 和 refs/tags），然后初始化 HEAD 文件，内容为 ref: refs/heads/master。这个过程本质上是构建 Git 仓库的基础环境。
接下来，模拟 git add 操作。该命令实际上执行两个步骤：将工作区文件内容创建为 Blob 对象并存入 objects 目录，同时更新索引文件（.git/index）以记录文件名、模式和 Blob 的 SHA-1。我们可以使用 git hash-object -w 创建 Blob，然后用 git update-index 更新索引。例如，执行 git update-index --add --cacheinfo 100644 <blob-sha> filename.txt 将文件加入索引，再通过 git ls-files --stage 查看索引内容，验证文件状态。
最后，模拟 git commit 操作。这一过程涉及三个关键步骤：用当前索引创建 Tree 对象（git write-tree）、基于 Tree 和父 Commit 创建 Commit 对象（git commit-tree），以及更新分支引用。具体来说，先运行 tree_sha=$(git write-tree) 获取 Tree 哈希，然后执行 commit_sha=$(echo My commit msg | git commit-tree $tree_sha) 生成 Commit 哈希，最后通过 echo $commit_sha > .git/refs/heads/master 将分支指向新提交。此时，使用 git log --oneline $commit_sha 可以查看刚刚创建的提交历史，这标志着一个完整提交周期的实现。
除了核心对象，Git 还包含其他重要概念，如 Tag 对象和 Packfiles。Tag 对象是一种特殊类型，指向特定 Commit，用于提供永久性标记；Packfiles 则是 Git 的压缩机制，将多个松散对象打包以节省空间。这些机制进一步优化了 Git 的性能和可用性。
高层命令与底层命令之间存在紧密联系。例如，git status 通过比较 HEAD、index 和工作区三者的 Tree 差异来报告状态；git branch 本质上是在 refs/heads 下创建或删除文件；而 git checkout 则用指定 Commit 的 Tree 覆盖工作区并更新 HEAD。理解这些关系有助于在复杂场景中灵活运用 Git。
回顾全文，Git 的本质是一个「内容寻址文件系统」，其强大之处源于 Blob、Tree 和 Commit 对象构建的版本控制模型。鼓励读者在遇到问题时，多用 git cat-file -p 和 git ls-tree 等命令探查内部状态，以巩固理解。下一步，可以尝试用 Python 或 Go 等语言实现一个简单的 my-git 工具，这将进一步深化对 Git 原理的掌握。通过这种从理论到实践的探索，我们不仅能揭开 Git 的魔法外衣，还能在开发中游刃有余。