# Git底层原理

## Git起源

1991 年 Linus 创建了开源的 Linux，自此 Linux 不断发展成为服务器系统首选，2002 年之前各地开源 Linux 的代码贡献者通过 diff 方式把源代码发给 Linus，然后 Linus 本人通过手工方式合并代码，Linus 那时坚定反对 CSV 和 SVN，因为这些集中式版本控制系统不但速度慢而且需要联网才行，虽然有一些好用的商用版本控制系统，可惜需要付费，这与 Linus 提倡的开源精神不符合，到 2002 年 Linux 代码库已经大到 Linus 很难维护了，社区成员也对此表示不满，于是 Linus 选择使用了商业版本控制系统 BitKeeper，该系统的公司处于人道主义授权 Linux 社区免费使用该版本控制系统，可是在 2005 年的时候，Liunx 社区中的开发 Samba 的安德鲁视图破解 BitKeeper 协议，其实还有社区里的其他人，被 BitKeeper 公司发现，一气之下，公司决定收回 Linux 社区免费授权，之后 Linus 没有选择向 BitKeeper 所有的公司道歉，而是选择自己花了两周左右的时间自己用 C 写了一个分布式版本控制系统，这就是 Git，一个月的之内 Linux 系统源码已经可以被 Git 管理了，接着 Git 就迅速成为最流行的分布式版本控制系统，2008 年的时候，GitHub 上线，无数开源项目通过 Git 存储在 GitHub 中直到现今

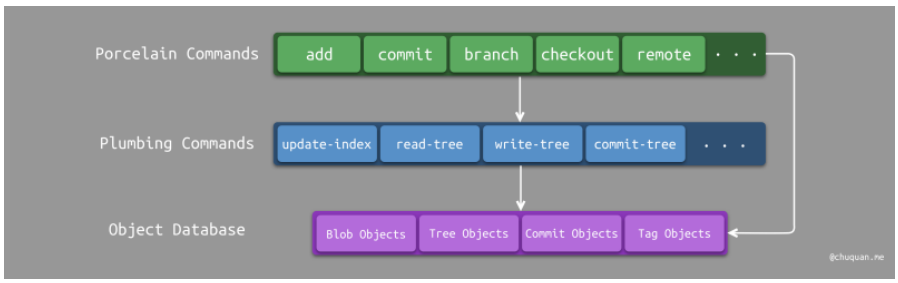
## Git整体架构

主要包含三部分：

(1) 上层命令

(2) 底层命令

(3) 对象数据库




## 上层命令

在日常开发中，我们所使用的 Git命令基本上都是上层命令，如：`commit`、`add`、`checkout`、`branch`、`remote` 等。上层命令通过组合底层命令或直接操作底层数据对象，使 Git 底层实现细节对用户透明，从而为用户提供了一系列简单易用的命令集合。

## 底层命令

在日常开发中，我们基本接触不到 Git 的底层命令，如果要想使用这些底层命令，我们必须要对 Git 的设计原理有一定的认知。Linus Torvalds 的第一版 Git，其实就是实现了几个核心的底层命令，如：`update-cache`、`write-tree`、`read-tree`、`commit-tree`、`cat-file`、`show-diff` 等。注意，在底层命令的命名上，我们当前版本与最初版本存在细微的差异，下表是几个核心底层命令的简单对照。

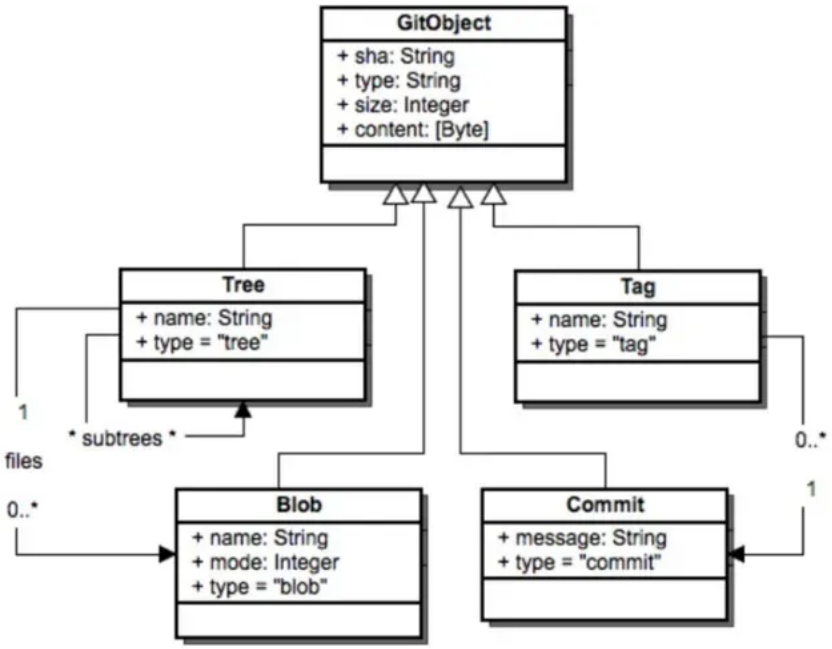
| 当前版本 | 原始版本 |
| --- | --- |
| `git update-index` | `update-cache` |
| `git write-tree` | `write-tree` |
| `git read-tree` | `read-tree` |
| `git commit-tree` | `commit-tree` |
| `git cat-file` | `cat-file` |

## 对象数据库

Git **最核心、最底层** 的部分则是其所实现的一套 **对象数据库**（Object Database），其本质是一个基于 Key-Value 的内容寻址文件系统（Content-addressable File System）。

**存储方式：**

在 Git 文件系统中，**使用 Object 存储所有类型的内容**，也称为 Git 对象，不同类型的 Object 共同构成了一整套对象模型。Git对象模型主要包括以下四种对象。




所有对象均存储在.git/objects/目录下，并采用相同的格式进行表示，其可以分为两部分：

- 头部信息：类型 + 空格 + 内容字节数 + \0
- 存储内容

Git使用两部分未压缩内容的40位SHA-1值（前两位作为子目录，后38位作为文件名）作为快照文件的唯一标识，并对他们进行zlib压缩，然后将压缩后的结果作为快照文件的实际内容进行存储

先创建一个git仓库，然后添加两个文件

```
 echo test1 > 1.txt
 echo test2 > 2.txt
 git add .
```

查看.git/objects/目录

```
.git/objects/
├── 18
│   └── 0cf8328022becee9aaa2577a8f84ea2b9f3827
├── a5
│   └── bce3fd2565d8f458555a0c6f42d0504a848bd5
├── info
└── pack
```

```
# python3
1 import hashlib
2
3 content = "test1"
4 header = f"blob 6\0"
5 store = header + content
6 m = hashlib.sha1()
7 m.update(store.encode("utf-8"))
8 print(m.hexdigest())
```

```
zxy@DESKTOP-H780R31:/tmp/test-git$ file .git/objects/18/0cf8328022becee9aaa2577a8f84ea2b9f3827
.git/objects/18/0cf8328022becee9aaa2577a8f84ea2b9f3827: zlib compressed data
zxy@DESKTOP-H780R31:/tmp/test-git$
```

```
git cat-file -t 180c    #查看对象类型
git cat-file -s 180c    #查看对象的内容长度
git cat-file -p 180c    #查看对象的内容
```

继续创建test目录，然后touch 3.py文件进去，git add . && git commit

```
.git/objects/
├── 06
│   └── 11ca6b60062f2a615807132de3906a8b7b3366
├── 18
│   └── 0cf8328022becee9aaa2577a8f84ea2b9f3827
├── 1f
│   └── 45f0bd7c5f46dc15775977360d1fb07d16e926
├── 29
│   └── 61c246fe9e76c1d9bbd57390daabf4fe7ce0e9
├── a5
│   └── bce3fd2565d8f458555a0c6f42d0504a848bd5
├── e6
│   └── 9de29bb2d1d6434b8b29ae775ad8c2e48c5391
├── info
└── pack
```

我们来看一下每个对象的类型

```
zxy@DESKTOP-H780R31:/tmp/test-git$ git cat-file -t 0611
tree
zxy@DESKTOP-H780R31:/tmp/test-git$ git cat-file -t 1f45
commit
zxy@DESKTOP-H780R31:/tmp/test-git$ git cat-file -t 2961
tree
zxy@DESKTOP-H780R31:/tmp/test-git$ git cat-file -t e69d
blob
zxy@DESKTOP-H780R31:/tmp/test-git$
```

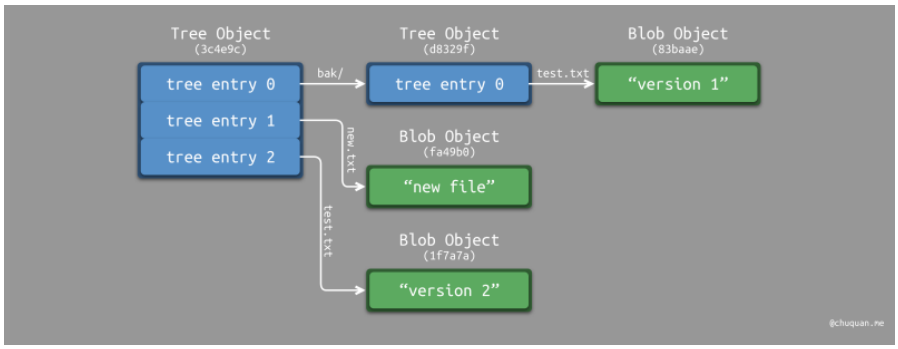
我们来看一下Tree的内容

```
zxy@DESKTOP-H780R31:/tmp/test-git$ git cat-file -p 2961
100644 blob a5bce3fd2565d8f458555a0c6f42d0504a848bd5    1.txt
100644 blob 180cf8328022becee9aaa2577a8f84ea2b9f3827    2.txt
040000 tree 0611ca6b60062f2a615807132de3906a8b7b3366    test
zxy@DESKTOP-H780R31:/tmp/test-git$ git cat-file -p 0611
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    3.py
zxy@DESKTOP-H780R31:/tmp/test-git$
```

Tree对象记录的结构

树对象记录的结构（Git v2.0.0）为：**文件模式 + 空格 + 树对象记录的字节数 + 文件路径 + \0 + SHA-1**。

如果某一时刻，Git 仓库的文件结构如下所示，那么在 Git 文件系统中，会建立一个对象关系图，如下图所示。



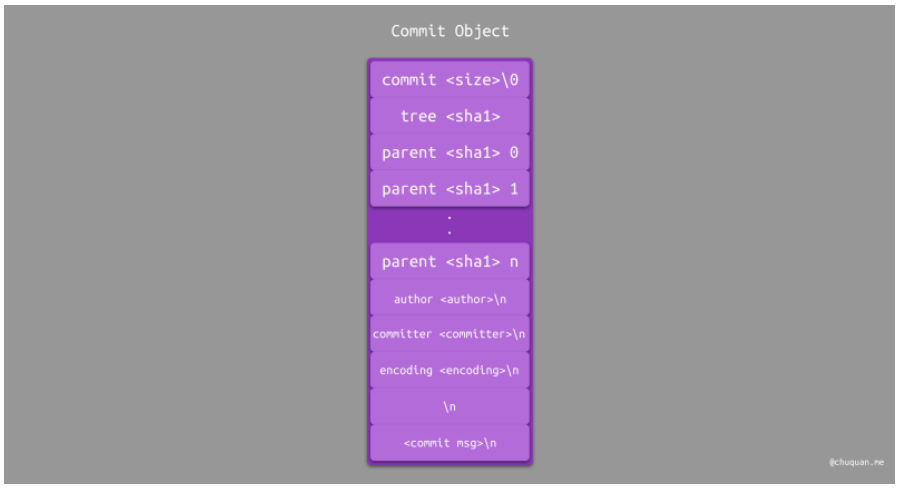

！--- 图需要重新画一下，commit对不上

Tree Object 和 Blob Object **用于表示版本快照**，Commit Object 则不同，它**用于表示版本索引和版本关系**。此外，Tree Object 和 Blob Object 的 SHA-1 值是根据内容计算得到的，只要内容相同，SHA-1 值相同；而 Commit Object 会结合内容、时间、作者等数据，因此 SHA-1 值很难出现冲突。

Commit Object 的头部信息为 **“commit” + 空格 + 内容字节数 + \0**，存储内容包含多个部分（Git v2.0.0），具体如下图所示。

- 对应的根 Tree Object 对应的 SHA-1

- 一个或多个父级 Commit Object 对应的 SHA-1。当进行分支合并时就会出现多个父级 Commit Object。
- 提交相关内容，包括：作者信息、提交者信息、编码、提交描述等




上面的例子，我们来看一下commit对象的内容

```
zxy@DESKTOP-H780R31:/tmp/test-git$ git cat-file -p 1f45
tree 2961c246fe9e76c1d9bbd57390daabf4fe7ce0e9
author neuliyiping <neu_liyiping@qq.com> 1677413521 +0800
committer neuliyiping <neu_liyiping@qq.com> 1677413521 +0800

test
```

接下来我们进行将1.txt中的内容进行下修改

```
echo "test111" > 1.txt
```

然后进行add && commit，此时的目录为

```
.git/objects/
├── 06
│   └── 11ca6b60062f2a615807132de3906a8b7b3366
├── 18
│   └── 0cf8328022becee9aaa2577a8f84ea2b9f3827
├── 1f
│   └── 45f0bd7c5f46dc15775977360d1fb07d16e926
├── 29
│   └── 61c246fe9e76c1d9bbd57390daabf4fe7ce0e9
├── 40
│   └── a3b4ae4ce2db1bf31e68b2315ad446d1858b58
├── 71
│   └── 60f3310fcdc3d003dae73d38b3afeb7c6df9e8
├── 9d
│   └── 7b82b4ee9ad6b4015185a84225aef8ed29455d
```

```
├── a5
│   └── bce3fd2565d8f458555a0c6f42d0504a848bd5
├── e6
│   └── 9de29bb2d1d6434b8b29ae775ad8c2e48c5391
├── info
└── pack
```

可以看出多了三个object

分别是40a3, 7160, 9d7b

```
zxy@DESKTOP-H780R31:/tmp/test-git$ git cat-file -t 40a3
blob
zxy@DESKTOP-H780R31:/tmp/test-git$ git cat-file -t 7160
tree
zxy@DESKTOP-H780R31:/tmp/test-git$ git cat-file -t 9d7b
commit
zxy@DESKTOP-H780R31:/tmp/test-git$
```

内容分别是

```
zxy@DESKTOP-H780R31:/tmp/test-git$ git cat-file -p 7160
100644 blob 40a3b4ae4ce2db1bf31e68b2315ad446d1858b58    1.txt
100644 blob 180cf8328022becee9aaa2577a8f84ea2b9f3827    2.txt
040000 tree 0611ca6b60062f2a615807132de3906a8b7b3366    test
zxy@DESKTOP-H780R31:/tmp/test-git$ git cat-file -p 40a3
test111
zxy@DESKTOP-H780R31:/tmp/test-git$ git cat-file -p 9d7b
tree 7160f3310fcdc3d003dae73d38b3afeb7c6df9e8
parent 1f45f0bd7c5f46dc15775977360d1fb07d16e926
author neuliyiping <neu_liyiping@qq.com> 1677414520 +0800
committer neuliyiping <neu_liyiping@qq.com> 1677414520 +0800

test2
zxy@DESKTOP-H780R31:/tmp/test-git$
```

可以看出只要修改了一个文件，就会创建新的blob和tree对象。

每一个 Commit Object 索引一个版本快照，每一个版本快照则是由一个 Tree Object 作为根节点进行构建。不同的版本快照之间会进行数据复用，从而最大限度地节省磁盘空间。每一个 Commit Object 记录了其父版本的索引信息，即另一个 Commit Object 的 SHA-1 值，从而构建了一个完整的版本关系图（有向无环图）。通过版本关系图，我们可以基于一个 Commit Object 回溯其任意历史版本。

git 区别与其他 vcs 系统的一个最主要原因之一是：git 对文件版本管理和其他 vcs 系统对文件版本的实现理念完成不一样。这也就是 git 版本管理为什么如此强大的最核心的地方。

Svn 等其他的 VCS 对文件版本的理念是以文件为水平维度，记录每个文件在每个版本下的 delta 改变。

Git 对文件版本的管理理念却是以每次提交为一次快照，提交时对所有文件做一次全量快照，然后存储快照引用。

Git 在存储层，如果文件数据没有改变的文件，Git 只是存储指向源文件的一个引用，并不会直接多次存储文件，这一点可以在 pack 文件中看见。

引用：


Git 底层原理*abcnull的博客-CSDN博客*git底层原理

一文讲透 Git 底层数据结构和原理 - 知乎 (zhihu.com)

深入理解 Git 底层实现原理 | 楚权的世界 (chuquan.me)

这才是真正的Git——Git内部原理揭秘！ - 知乎 (zhihu.com)

图解Git (marklodato.github.io)