改变世界的一次代码提交


https://hutusi.com/articles/the-greatest-git-commit

hutusi Fri Oct, 09 00:33

Git 第一次提交的源代码分析及带来的启示

September 21, 2020

TL;DR 本文较长，如果对 Git 内部实现不感兴趣可以快速跳过中间两个章节。

> 吾诗已成。无论大神的震怒，还是山崩地裂，都不能把它化为无形！
>
> —— 奥维德《变形记》

背景

Linux 作为最大也是最成功的开源项目，吸引了全球程序员的贡献，到目前为止，共有两万多名开发者给 Linux Kernel 提交过代码。令人惊讶的是，在项目的前十年(1991 ~ 2002)中，Linus 作为项目管理员并没有借助任何配置管理工具，而是以手工方式通过 patch 来合并大家提交的代码。倒不是说 Linus 喜欢手工处理，而是因为他对于软件配置管理工具(SCM)非常挑剔，无论是商用的 clearcase 还是开源的 cvs、svn 等都不能入他的法眼。在他看来，一个能够满足 Linux 内核项目开发使用的版本控制系统需要满足几个条件： 1) 快 2)支持多分支场景（几千个分支并行开发场景） 3) 分布式 4) 能够支持大型项目。直到2002年，Linus 终于找到了一款基本满足他要求的工具——BitKeeper, 而 BitKeeper 是商业工具，他们愿意给 Linux 社区免费使用，但是需要保证遵守不得进行反编译等条款。BitKeeper 提供的默认接口显然不能满足社区用户的全部需要，一位社区开发者反编译 BitKeeper 并利用了未公开接口，这让 BitKeeper 公司撤回了免费使用的 License。不得已，Linus 利用假期十天时间，实现一款 DVCS —— Git，并推送给社区开发者们使用。

设计

Git 已经成为全球软件开发者的标配，关于 Git 的介绍和用法不需多说，我今天想要谈谈 Git 的内部实现。不过在看本文之前，我先给大家提一个问题：如果是你来设计 git（或者重新设计 git），你打算怎么设计？第一个版本发布准备实现哪些功能？看完本文，再对照自己的想法做个比较。欢迎留言讨论。

学习 Git 的内部实现，最好的办法是看 Linus 最初的代码提交，checkout 出 git 项目的第一次提交节点（方法参见博客：《阅读开源代码小技巧》），可以看到代码库中只有几个文件：一个 README，一个构建脚本Makefile，剩下几个 C 源文件。这次 commit 的备注写的也非常特别： Initial revision of “git”, the information manager from hell.

```
commit e83c5163316f89bfbde7d9ab23ca2e25604af290
Author: Linus Torvalds <torvalds@ppc970.osdl.org>
Date:   Thu Apr 7 15:13:13 2005 -0700

    Initial revision of 'git', the information manager from hell
```

在 README 中，Linus 详细描述了 Git 的设计思路。看似复杂的 Git 工作，在 Linus 的设计里，只有两种对象抽象：1) 对象数据库(“object database”)； 2) 当前目录缓存(“current directory cache”)。

Git 的本质就是一系列的文件对象集合，代码文件是对象、文件目录树是对象、commit 也是对象。这些文件对象的名称即内容的 SHA1 值，SHA1 哈希算法的值为40位。Linus 将前二位作为文件夹、后38位作为文件名。大家可以在 .git 目录里的 objects 里看到有很多两位字母/数字名称的目录，里面存储了很多38位hash值名称的文件，这就是 Git 的所有信息。Linus 在设计对象的数据结构时按照 <标签ascii码表示>(blob/tree/commit) + <空格> + <长度ascii码表示> + <\0> + <二进制数据内容> 来定义，大家可以用 xxd 命令看下 objects 目录里的对象文件(需 zlib 解压)，比如一个 tree 对象文件内容如下：

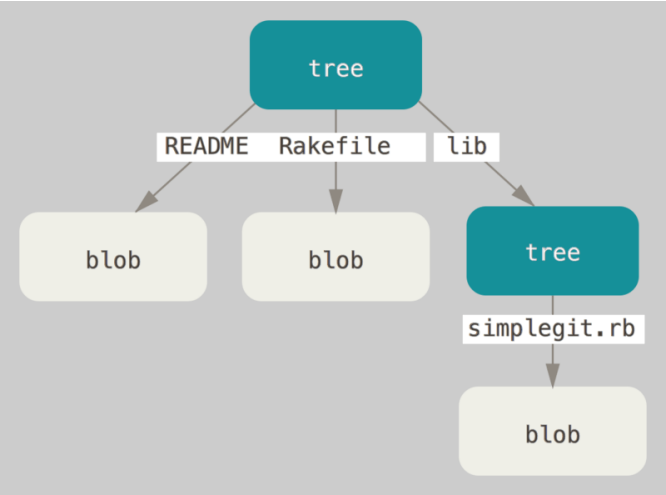
```
00000000: 7472 6565 2033 3700 3130 3036 3434 2068  tree 37.100644 h
00000010: 656c 6c6f 2e74 7874 0027 0c61 1ee7 2c56  ello.txt.'.a..,V
00000020: 7bc1 b2ab ec4c bc34 5bab 9f15 ba         {....L.4[....
```

对象有三种：BLOB、TREE、CHANGESET。

BLOB: 即二进制对象，这就是 Git 存储的文件，Git 不像某些 VCS （如 SVN）那样存储变更 delta 信息，而是存储文件在每一个版本的完全信息。比如先提交了一份 hello.c 进入了 Git 库，会生成一个 BLOB 文件完整记录 hello.c 的内容；对 hello.c 修改后，再提交 commit，会再生成一个新的 BLOB 文件记录修改后的 hello.c 全部内容。Linus 在设计时，BLOB 中仅记录文件的内容，而不包含文件名、文件属性等元数据信息，这些信息被记录在第二种对象 TREE 里。

TREE: 目录树对象。在 Linus 的设计里 TREE 对象就是一个时间切片中的目录树信息抽象，包含了文件名、文件属性及BLOB对象的SHA1值信息，但没有历史信息。这样的设计好处是可以快速比较两个历史记录的 TREE 对象，不能读取内容，而根据 SHA1 值显示一致和差异的文件。另外，由于 TREE 上记录文件名及属性信息，对于修改文件属性或修改文件名、移动目录

而不修改文件内容的情况，可以复用 BLOB 对象，节省存储资源。而 Git 在后来的开发演进中又优化了 TREE 的设计，变成了某一时间点文件夹信息的抽象，TREE 包含其子目录的 TREE 的对象信息（SHA1）。这样，对于目录结构很复杂或层级较深的 Git库 可以节约存储资源。历史信息被记录在第三种对象 CHANGESET 里。



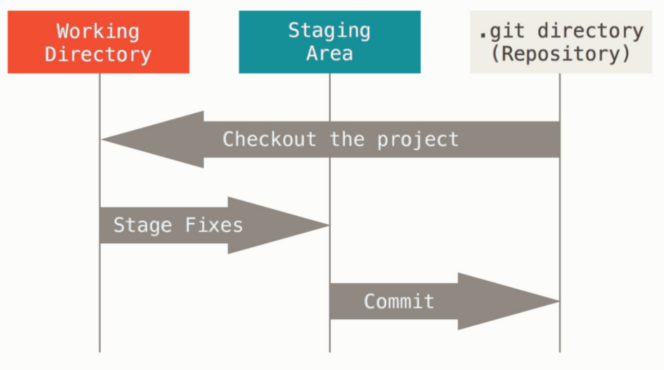
图片摘自： *Pro Git, 10.2 Git Internals - Git Objects* [1]

CHANGESET: 即 Commit 对象。一个 CHANGESET 对象中记录了该次提交的 TREE 对象信息（SHA1），以及提交者(committer)、提交备注(commit message)等信息。跟其他SCM（软件配置管理）工具所不同的是，Git 的 CHANGESET 对象不记录文件重命名和属性修改操作，也不会记录文件修改的 Delta 信息等，CHANGESET 中会记录父节点 CHANGESET 对象的 SHA1 值，通过比较本节点和父节点的 TREE 信息来获取差异。Linus 在设计 CHANGESET 父节点时允许一个节点最多有 16 个父节点，虽然超过两个父节点的合并是很奇怪的事情，但实际上，Git 是支持超过两个分支的多头合并的。

Linus 在三种对象的设计解释后着重阐述了可信(TRUST)：虽然 Git 在设计上没有涉及可信的范畴，但 Git 作为配置管理工具是可以做到可信的。原因是所有的对象都以SHA1编码（Google 实现 SHA1 碰撞攻击是后话，且 Git 社区也准备使用更高可靠性的 SHA256 编码来代替），而签入对象的过程可信靠签名工具保证，如 GPG 工具等。

理解了Git 的三种基本对象，那么对于 Linus 对于 Git 初始设计的“对象数据库”和“当前目录缓存”这两层抽象就很好理解了。加上原本的工作目录，Git 有三层抽象，如下图示：一个是当前工作区(Working Directory)，也就是我们查看/编写代码的地方，一个是 Git 仓库(Repository),即 Linus 说的对象数据库，我们在 Git 仓看到的 .git 文件夹中存储的内容，Linus 在第一版设计时命名为 .dircache，在这两个存储抽象中还有一层中间的缓存区（Staging Area），即 .git/index 里存储的信息，我们在执行 git add 命令时，便是将当前修改加入到了缓存区。

Linus 解释了“当前目录缓存”的设计，该缓存就是一个二进制文件，内容结构很像 TREE 对象，与 TREE 对象不同的是 index 不会再包含嵌套 index 对象，即当前修改目录树内容都在一个 index 文件里。这样设计有两个好处：1. 能够快速的复原缓存的完整内容，即使不小心把当前工作区的文件删除了，也可以从缓存中恢复所有文件；2. 能够快速找出缓存中和当前工作区内容不一致的文件。



图片摘自： *Things About Git and Github You Need to Know as Developer* [2]

实现

Linus 在 Git 的第一次代码提交里便完成了 Git 的最基础功能，并可以编译使用。代码极为简洁，加上 Makefile 一共只有 848 行。感兴趣的同事可以通过上一段所述方法 checkout Git 最早的 commit 上手编译玩玩，只要有 Linux 环境即可。因为依赖库版本的问题，需要对原始 Makefile 脚本做些小修改。Git 第一个版本依赖 openssl 和 zlib 两个库，需要手工安装这两个开发库。在 ubuntu 上执行： `sudo apt install libssl-dev libz-dev` ； 然后修改 makefile 在 `LIBS= -lssl` 行中的 `-lssl` 改成 `-lcrypto` 并增加 `-lz` ；最后执行 make，忽略编译告警，会发现编出了7个可执行程序文件： init-db, update-cache, write-tree, commit-tree, cat-file, show-diff 和 read-tree.

下面分别简要介绍下这些可执行程序的实现：

1. init-db: 初始化一个 git 本地仓库，这也就是我们现在每次初始化建立 git 库式敲击的 `git init` 命令。只不过一开始 Linus 建立的仓库及 cache 文件夹名称叫 `.dircache` ，而不是我们现在所熟知的 `.git` 文件夹。
2. update-cache: 输入文件路径，将该文件（或多个文件）加入缓冲区中。具体实现是：校验路径合法性，然后将文件计算 SHA1值，将文件内容加上 blob 头信息进行 zlib 压缩后写入到对象数据库(.dircache/objects)中；最后将文件路径、文件属性及 blob sha1 值更新到 .dircache/index 缓存文件中。

3. write-tree: 将缓存的目录树信息生成 TREE 对象，并写入对象数据库中。TREE 对象的数据结构为：‘tree‘ + 长度 + \0 + 文件树列表。文件树列表中按照 文件属性 + 文件名 + \0 + SHA1 值结构存储。写入对象成功后，返回该 TREE 对象的 SHA1 值。
4. commit-tree: 将 TREE 对象信息生成 commit 节点对象并提交到版本历史中。具体实现是输入要提交的 TREE 对象 SHA1 值，并选择输入父 commit 节点（最多 16个），commit 对象信息中包含 TREE、父节点、committer 及作者的 name、email及日期信息，最后写入新的 commit 节点对象文件，并返回 commit 节点的 SHA1 值。
5. cat-file: 由于所有的对象文件都经过 zlib 压缩，因此想要查看文件内容的话需要使用这个工具来解压生成临时文件，以便查看对象文件的内容。
6. show-diff: 快速比较当前缓存与当前工作区的差异，因为文件的属性信息（包括修改时间、长度等）也保存在缓存的数据结构中，因此可以快速比较文件是否有修改，并展示差异部分。
7. read-tree: 根据输入的 TREE 对象 SHA1 值输出打印 TREE 的内容信息。

这就是第一个可用版本的 Git 的全部七个子程序，可能用过 Git 的同事会说：这怎么跟我常用的 Git 命令不一样呢？ Git add, git commit 呢？是的，在最初的 Git 设计中是没有我们这些平常所使用的 git 命令的。在 Git 的设计中，有两种命令：分别是底层命令(Plumbing commands)和高层命令(Porcelain commands)。一开始，Linus 就设计了这些给开源社区黑客使用的符合 Unix KISS 原则的命令，因为黑客们本身就是动手高手，水管坏了就撸起袖子去修理，因此这些命令被称为 plumbing commands. 后来接手 Git 的 Junio Hamano 觉得这些命令对于普通的用户可不太友好，因此在此之上，封装了更易于使用、接口更精美的高层命令，也就是我们今天每天使用的 git add, git commit 之类。Git add 就是封装了 update-cache 命令，而 git commit 就是封装了 write-tree, commit-tree 命令。关于底层命令的更详细介绍，大家有兴趣的话可以看 *Pro Git* 中的 Git Internals 章节。

具体的代码实现在这里就不再细述，Linus 的代码风格极为简洁，能一行完成的绝不写两行。另外，对于 Linux API 的使用自然无人出其右，我印象最深的是有好多处使用 mmap 建立文件与内存的映射，省去了内存申请、文件读写等操作，提升了工具性能。正如一位同事说的：Linus 的代码除了不满足编程规范，其他好像真挑不出什么毛病。顺便说一句，Linus 的缩进风格是 Tab 键。

启示

Linus 在提交了第一个 git commit 后，并向社区发布了 git 工具。当时，社区中有位叫 Junio Hamano 的开发者觉得这个工具很有意思，便下载了代码，结果发现一共才 1244 行代码，这更令他惊奇也引发了极大的兴趣。Junio 在邮件列表与 Linus 交流并帮助增加了 merge 等功能，而后持续打磨 git，最后 Junio 完全接手了 Git 的维护工作，Linus 则回去继续维护 Linux Kernel 项目。

如果选历史上最伟大的一次 Git 代码提交，那一定是这 Git 工具项目本身的第一次代码提交。这次代码提交无疑是开创性的，如果说 Linux 项目促成了开源软件的成功并改写了软件行业的格局，那么 Git 则是改变了全世界开发者的工作方式和写作方式。在 Git 诞生后两年，旧金山的一个小酒馆里坐着三位年轻的程序员，决定要用 Git 做点什么，几个月后，GitHub 上线。

回到文中开头提到的问题，如果我来设计 Git 的话，估计还是会从已有工具经验（如SVN使用）上来延伸设计，甚至在我最早接触 Git 时候曾肤浅的认为 Git 就是 SVN + 分布式。正是了解了 Git 的内部原理乃至阅读了 Git 的初始代码后才感叹其设计的精妙，Git 的初始设计和实现大概能给（开源）软件产品如下启发：

1. **解决痛点问题**：Git 的缘起便是 Linus 本人及 Linux 社区的诉求，而这些诉求推而广之是项目协作开发（特别是跨地域项目）的共性诉求。Linus 解决了他本人遇到的痛点问题，顺便达成了一项伟大的成就。
2. **极简设计**：Linus 在设计 Git 工具时并没有受传统 SCM 工具的束缚，考虑文件差异、版本对比等，而是抽象了几种基本对象就把 git 的设计思路给理清楚了。
3. **MVP (minimum viable product, 最小可用产品)**：这个概念大家都懂，但实际操作起来却不容易。一个 MVP 的配置管理工具需要哪些功能？一般来说会想到代码提交、历史追溯、版本比较、分支合并等。但 Linus 却将它拆解开来，快速实现了底层的基本功能，简单到只有开源社区黑客才能用。但这就够了，黑客们因此发现了它的价值，继续给它添砖加瓦。
4. **快速发布，快速迭代**：这也是源于 Linux Kernel 的开发经验；Linus 在实现了 Git MVP 后，便在 Linux 社区邮件列表中公布，并征求意见，迭代完善。
5. **找到合适接班人**：《大教堂与集市》中也有类似的观点，它说的是：“如果你对一个项目失去了兴趣，你最后的职责就是把它交给一个称职的继承者。”不过 Linus 将 Git 交给 Junio 并不是因为失去了兴趣，而是因为他发现在 Git 基础架构建立好之后，Junio 比他更擅长于实现更丰富、对普通用户界面更友好的功能，因此他就放心的将 Git 交给了 Junio. 为开源项目找到更合适的接班人，这既需要魄力也需要智慧。

[3]
