Learn Git The Not So Super Hard Way[1]

Zenithal
Woshiluo Luo As translator

2024-11-24

[1] Credit to https://github.com/b1f6c1c4/learn-git-the-super-hard-way

为什么需要这个

- 学习 Git 太痛苦了
  - 太多概念 (commit, branch, stage, index)
  - 太多命令 (clone, pull, push)

为什么需要这个

- 学习 Git 太痛苦了
  - 太多概念 (commit, branch, stage, index)
  - 太多命令 (clone, pull, push)
- 状态模型太过复杂
  - 你常常不知道自己在哪个状态下。
  - 冲突! Help me ERIN!

为什么需要这个

- 学习 Git 太痛苦了
  - 太多概念 (commit, branch, stage, index)
  - 太多命令 (clone, pull, push)
- 状态模型太过复杂
  - 你常常不知道自己在哪个状态下。
  - 冲突! Help me ERIN!
- 知道命令是远远不够的
  - 你常常不知道发生了什么
  - 让我们来看一些更基本的组件

为什么需要这个

- 学习 Git 太痛苦了
  - 太多概念 (commit, branch, stage, index)
  - 太多命令 (clone, pull, push)
- 状态模型太过复杂
  - 你常常不知道自己在哪个状态下。
  - 冲突! Help me ERIN!
- 知道命令是远远不够的
  - 你常常不知道发生了什么
  - 让我们来看一些更基本的组件
- 最困难的路，最简单的方法。
  - 你对文件做了改动，你知道发生了什么。

init

git repo structure

- git repo
  - 通常在 the `.git`
  - 通常包含 HEAD, config
- worktree
  - 文件们
  - 通常包含 README.md, main.c, main.h
  - worktree 只是 git repo 的一个 checkout
  - 你可以从 git repo 中重新构建 worktree
  - Git repo 是一切的基石，worktree 并不是

git init

- `mkdir .git`
- `mkdir .git/objects`
  - 必须项
- `mkdir .git/refs`
  - 必须项
- `echo 'ref: refs/heads/master' > .git/HEAD`
  - 建立 HEAD ref
  - HEAD 指向 `.git/refs/heads/master` (尽管现在还不存在)
  - 即: `refs/heads/main`
- config, hooks, info, etc 并非必要
- 现在你可以使用 `git status` 来检查状态

objects

objects

- 你已经创建了 `.git/objects`, 那么什么是 objects

objects

- 你已经创建了 `.git/objects`, 那么什么是 objects
- 四种 objects

objects

- 你已经创建了 `.git/objects`, 那么什么是 objects
- 四种 objects
  - blob: 文件内容

objects

- 你已经创建了 `.git/objects`, 那么什么是 objects
- 四种 objects
  - blob: 文件内容
  - tree: 文件夹
    - 注记: 文件系统中的文件夹
    - 文件名 (先于 blob 储存!)
    - 树和 blobs 的 hash (文件夹结构!)

objects

- 你已经创建了 `.git/objects`, 那么什么是 objects
- 四种 objects
  - blob: 文件内容
  - tree: 文件夹
    - 注记: 文件系统中的文件夹
    - 文件名 (先于 blob 储存!)
    - 树和 blobs 的 hash (文件夹结构!)
  - commit: 根文件夹的一个状态
    - 包含一个指定的树
    - 祖先 (们): 其他 commit
    - author/committer/commit message: meta data

objects

- 你已经创建了 `.git/objects`, 那么什么是 objects
- 四种 objects
  - blob: 文件内容
  - tree: 文件夹
    - 注记: 文件系统中的文件夹
    - 文件名 (先于 blob 储存!)
    - 树和 blobs 的 hash (文件夹结构!)
  - commit: 根文件夹的一个状态
    - 包含一个指定的树
    - 祖先 (们): 其他 commit
    - author/committer/commit message: meta data
  - tag: 不是本课件的讨论范围

blob

- blob: 文件内容

blob

- blob: 文件内容
- `echo 'hello' | git hash-object -t blob --stdin -w`
  - 写入一个内容为 `'hello'` 的 blob/file
  - 得到对于一个类型为 `blob` 的 object，从 `stdin` 输入并写入（write）object database
  - 输出：ce013625030ba8dba906f756967f9e9ca394464a, the hash of the object

blob

- blob: 文件内容
- `echo 'hello' | git hash-object -t blob --stdin -w`
  - 写入一个内容为 `'hello'` 的 blob/file
  - 得到对于一个类型为 `blob` 的 object，从 `stdin` 输入并写入（write）object database
  - 输出: ce013625030ba8dba906f756967f9e9ca394464a, the hash of the object
- `cat .git/objects/ce/013625030ba8dba906f756967f9e9ca394464a`
  - 输出: xKORO\cH, hello 的压缩后数据
  - 注意该文件的路径!

blob

- blob: 文件内容
- `echo 'hello' | git hash-object -t blob --stdin -w`
  - 写入一个内容为 `'hello'` 的 blob/file
  - 得到对于一个类型为 `blob` 的 object，从 `stdin` 输入并写入（write）object database
  - 输出: ce013625030ba8dba906f756967f9e9ca394464a, the hash of the object
- `cat .git/objects/ce/013625030ba8dba906f756967f9e9ca394464a`
  - 输出: `xKORO⊂H`, hello 的压缩后数据
  - 注意该文件的路径!
- 查看真实内容

```
$ printf '\x1f\x8b\x08\x00\x00\x00\x00\x00' \
| cat - .git/objects/ce/013625030ba8dba906f756967f9e9ca394464a \
| gunzip -dc 2>/dev/null | xxd
# 00000000: 626c 6f62 2036 0068 656c 6c6f 0a         blob 6.hello.
```

blob (cont'd)

- 使用这些直接命令很痛苦？当然我们有更高级的指令
- `git cat-file blob ce01`
  - Output: `hello`
- `git show ce01`
  - Output: `hello`

tree

- tree: **文件夹**

tree

- tree: **文件夹**
- **创建一棵** tree

```
(printf '100644 name.ext\x00';
echo '0: ce013625030ba8dba906f756967f9e9ca394464a' | xxd -rp -c 256;
printf '100755 name2.ext\x00';
echo '0: ce013625030ba8dba906f756967f9e9ca394464a' | xxd -rp -c 256) \
| git hash-object -t tree --stdin -w
# 58417991a0e30203e7e9b938f62a9a6f9ce10a9a
```

tree

- tree: 文件夹
- 创建一棵 tree

```
(printf '100644 name.ext\x00';
echo '0: ce013625030ba8dba906f756967f9e9ca394464a' | xxd -rp -c 256;
printf '100755 name2.ext\x00';
echo '0: ce013625030ba8dba906f756967f9e9ca394464a' | xxd -rp -c 256) \
| git hash-object -t tree --stdin -w
# 58417991a0e30203e7e9b938f62a9a6f9ce10a9a
```

- 你也可以（另一种形式）

```
git mktree --missing <<EOF
100644 blob ce013625030ba8dba906f756967f9e9ca394464a$(printf '\t')name.ext
100755 blob ce013625030ba8dba906f756967f9e9ca394464a$(printf '\t')name2.ex
EOF
# 58417991a0e30203e7e9b938f62a9a6f9ce10a9a
```

tree (cont'd)

- **直接查看文件内容**

```
printf '\x1f\x8b\x08\x00\x00\x00\x00\x00' \
| cat - .git/objects/58/417991a0e30203e7e9b938f62a9a6f9ce10a9a \
| gunzip -dc 2>/dev/null | xxd
```

tree (cont'd)

- **直接查看文件内容**

```
printf '\x1f\x8b\x08\x00\x00\x00\x00\x00' \
| cat - .git/objects/58/417991a0e30203e7e9b938f62a9a6f9ce10a9a \
| gunzip -dc 2>/dev/null | xxd
```

- `git cat-file tree 5841 | xxd`

tree (cont'd)

- 直接查看文件内容

```
printf '\x1f\x8b\x08\x00\x00\x00\x00\x00' \
| cat - .git/objects/58/417991a0e30203e7e9b938f62a9a6f9ce10a9a \
| gunzip -dc 2>/dev/null | xxd
```

- `git cat-file tree 5841 | xxd`
- `git ls-tree 5841` (和上文 mktree 相对)

tree (cont'd)

- 直接查看文件内容

```
printf '\x1f\x8b\x08\x00\x00\x00\x00\x00' \
| cat - .git/objects/58/417991a0e30203e7e9b938f62a9a6f9ce10a9a \
| gunzip -dc 2>/dev/null | xxd
```

- `git cat-file tree 5841 | xxd`
- `git ls-tree 5841` (和上文 mktree 相对)
- `git show 5841` (一个更简单的版本)

直接创建文件

```
git hash-object -t commit --stdin -w <<EOF
tree 58417991a0e30203e7e9b938f62a9a6f9ce10a9a
author b1f6c1c4 <b1f6c1c4@gmail.com> 1514736000 +0800
committer b1f6c1c4 <b1f6c1c4@gmail.com> 1514736000 +0800

The commit message
May have multiple
lines!
EOF
# d4dafde7cd9248ef94c0400983d51122099d312a
```

commit (cont'd)

或者一个更高级的指令

```
GIT_AUTHOR_NAME=b1f6c1c4 \
GIT_AUTHOR_EMAIL=b1f6c1c4@gmail.com \
GIT_AUTHOR_DATE='1600000000 +0800' \
GIT_COMMITTER_NAME=b1f6c1c4 \
GIT_COMMITTER_EMAIL=b1f6c1c4@gmail.com \
GIT_COMMITTER_DATE='1600000000 +0800' \
git commit-tree 5841 -p d4da <<EOF
Message may be read
from stdin
or by the option '-m'
EOF
# efd4f82f6151bd20b167794bc57c66bbf82ce7dd
```

这也是为什么你需要执行 `git config --global user.email` 和 `user.name`

commit (cont'd)

- **直接观察文件**

```
printf '\x1f\x8b\x08\x00\x00\x00\x00\x00' \
| cat - ./objects/ef/d4f82f6151bd20b167794bc57c66bbf82ce7dd \
| gunzip -dc 2>/dev/null | xxd
```

[2] https://github.blog/2020-12-17-commits-are-snapshots-not-diffs/

commit (cont'd)

- **直接观察文件**

```
printf '\x1f\x8b\x08\x00\x00\x00\x00\x00' \
| cat - ./objects/ef/d4f82f6151bd20b167794bc57c66bbf82ce7dd \
| gunzip -dc 2>/dev/null | xxd
```

- `git cat-file commit efd4`

[2] https://github.blog/2020-12-17-commits-are-snapshots-not-diffs/

commit (cont'd)

- **直接观察文件**

```
printf '\x1f\x8b\x08\x00\x00\x00\x00\x00' \
| cat - ./objects/ef/d4f82f6151bd20b167794bc57c66bbf82ce7dd \
| gunzip -dc 2>/dev/null | xxd
```

- `git cat-file commit efd4`
- `git show efd4` (一种更简单的方式，以 diff 形式展示)
- **注**: commits 是快照，并非 diffs/patchs[2]

[2] https://github.blog/2020-12-17-commits-are-snapshots-not-diffs/

Lucky commit

- Hash 太无聊了!
- 试试 lucky commit![3]
- ```
  $ git log
  1f6383a Some commit
  $ lucky_commit
  $ git log
  0000000 Some commit
  ```
- 注意前一页的 commit msg，我们可以通过控制他来得到了一个 lucky hash

[3] https://github.com/not-an-aardvark/lucky-commit
  ```

ref

ref

- ref 是一个方便的，指向一个特性 commit 或者其他 ref 的引用
- 存在 `.git/ref`
- 两种 ref
  - 直接 ref
  - 简介 ref, e.g. `HEAD` (常见情况)
- 今天将会介绍两个常见 ref
  - heads: 本地分支
  - remotes: 远端分支

local branch and direct ref

- 直接创建文件 (由于没有 reflog 所以不推荐)

```
mkdir -p .git/refs/heads/
echo d4dafde7cd9248ef94c0400983d51122099d312a > .git/refs/heads/br1
```

local branch and direct ref

- 直接创建文件 (由于没有 reflog 所以不推荐)

```
mkdir -p .git/refs/heads/
echo d4dafde7cd9248ef94c0400983d51122099d312a > .git/refs/heads/br1
```

- 下面的命令会在 `.git/log/refs/heads/br1` 留下 reflog
- `git update-ref --no-deref -m 'Reason for update' refs/heads/br1 d4da`
- `git branch -f br1 d4da`

关于 reflog

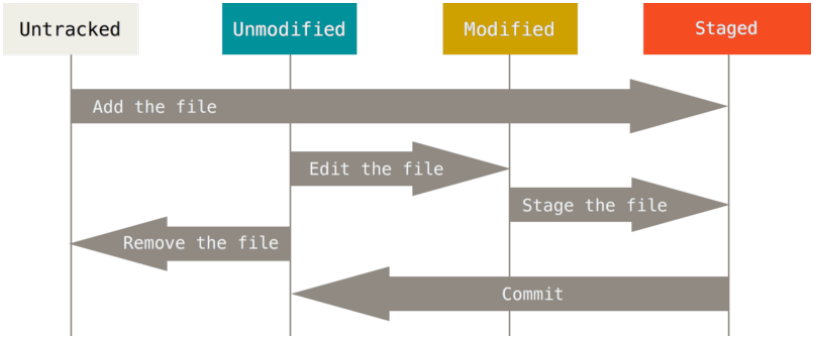
- 记录你的 ref 的一切变化
- 在你一不小心 swich 到其他地方时很有用
  - `git rebase master`
  - `git checkout -B master origin/master`
  - 之后你由于某种原因想要 switch 到之前的 tree
  - reflog 会给出一个 ref **曾经的** commit
- 一个栗子: lots of reflogs

indirect ref

- 还记得你初始化 git repo 的命令吗？
  - `echo 'ref: refs/heads/master' > .git/HEAD`
- 这种格式就是 indirect ref

index

index



- index 储存你想要在 `git commit` commited 的内容
- 文件是 `.git/index`
- 通常叫在 index 中的文件为 staged (如上文所说)
- 一个复杂的数据库
- 包含很多东西, 如 filename, mode, hash, mtime, 等

操纵 index

- 这很困难
- 我们可以学习一些常规情况
- `git add` 将内容存入 index
- 将他们标记为将被 commit

操纵 index

- 这很困难
- 我们可以学习一些常规情况
- `git add` 将内容存入 index
- 将他们标记为将被 commit
- 1. `git add; git status`
  - 那些已经被你加入 index 的文件将被 commit

操纵 index

- 这很困难
- 我们可以学习一些常规情况
- `git add` 将内容存入 index
- 将他们标记为将被 commit
- 1. `git add; git status`
  - 那些已经被你加入 index 的文件将被 commit
- 2. `git add; modify; git status`
  - 你已经 added 的文件内容将被 commit
  - 新的文件内容尚未被 add，并不在 index 里
  - `modify` 将不会被 commit

操纵 index

- 这很困难
- 我们可以学习一些常规情况
- `git add` 将内容存入 index
- 将他们标记为将被 commit
- 1. `git add; git status`
  - 那些已经被你加入 index 的文件将被 commit
- 2. `git add; modify; git status`
  - 你已经 added 的文件内容将被 commit
  - 新的文件内容尚未被 add，并不在 index 里
  - `modify` 将不会被 commit
- 3. `git add; rm; git status; git restore`
  - 景观文件已经被删除，但是他仍在 `index` 中有一份拷贝
  - 如果你不慎 `rm -rf *`，你可以恢复你的文件!

switch/checkout

switch/checkout

- 回忆一下，`.git/HEAD` 是一个 ref
- 这是你 worktree 的 ref
- 回忆一下，你的 worktree 是你实际看到的内容
- 我们可以通过操纵 HEAD 来修改你的 worktree

switch/checkout (cont'd)

- 最著名的: `git checkout master`
  - 让 HEAD 指向 `refs/heads/master`
  - 然后 checkout 你 worktree 的内容
  - 这就是他为什么叫 checkout
  - 其实是一个就语法了，现在推荐使用 switch
  - `git switch master`

switch/checkout (cont'd)

- 最著名的: `git checkout master`
  - 让 HEAD 指向 `refs/heads/master`
  - 然后 checkout 你 worktree 的内容
  - 这就是他为什么叫 checkout
  - 其实是一个就语法了，现在推荐使用 switch
  - `git switch master`
- 也很著名的: `git reset --hard HEAD~1`
  - 将 HEAD 指向 HEAD~1 (the former commit of HEAD)
  - checkout 你 worktree 的内容
  - 注: 还有 `reset --soft/--mixed`, 请自行研究

pull/clone/push

remotes

- 我们已经提到过 `.git/refs/remotes`
- 既然我们有 local ref(branch), 我们也会有 remote ref(branch)
- 没有 remote branch, 算什么分布式版本控制系统（distributed version control system）
- 如果同步两端呢？
- pull commit 从 remote 到 local
- push commit 从 local 到 remote
- 所以 commit 的概念是非常有用的!

config remote

- 如果你还没有 remote，那么添加 remote 是必要的
- 编辑 `.git/config` 来添加
- 或者 `git remote add origin git@github.com:xxx/yyy`
  - `origin` 是一个惯用名，你可以取一个自己喜欢的名字。
  - 你可以有多个 remote
- Demo of my repo

fetch remote

- `git fetch origin master`
  - 从 `origin` 获取 `master` ref
  - 你现在可以看看 `.git/refs/remotes/origin/master` 了

fetch remote

- `git fetch origin master`
  - 从 `origin` 获取 `master` ref
  - 你现在可以看看 `.git/refs/remotes/origin/master` 了
- `git pull origin master`
  - 不同于 `git fetch`, 他会尝试更新你的 local ref
  - 更新 `.git/refs/remotes/origin/master`
  - 并同时更新 `.git/refs/heads/master`
  - 之间的关于会在 `.git/config` 中记录

fetch remote

- `git fetch origin master`
  - 从 `origin` 获取 `master` ref
  - 你现在可以看看 `.git/refs/remotes/origin/master` 了
- `git pull origin master`
  - 不同于 `git fetch`, 他会尝试更新你的 local ref
  - 更新 `.git/refs/remotes/origin/master`
  - 并同时更新 `.git/refs/heads/master`
  - 之间的关于会在 `.git/config` 中记录
- `git pull`
  - 上个命令的缩写，会根据 `.git/config` 中的记录执行

fetch remote

- `git fetch origin master`
  - 从 `origin` 获取 `master` ref
  - 你现在可以看看 `.git/refs/remotes/origin/master` 了
- `git pull origin master`
  - 不同于 `git fetch`, 他会尝试更新你的 local ref
  - 更新 `.git/refs/remotes/origin/master`
  - 并同时更新 `.git/refs/heads/master`
  - 之间的关于会在 `.git/config` 中记录
- `git pull`
  - 上个命令的缩写，会根据 `.git/config` 中的记录执行
- `git clone`
  - 事实上是下面命令的缩写
  - `git init`
  - `git remote add`
  - `git pull`

push remote

- `git push origin master`
  - 将 local branch master 同步到 remote branch master

push remote

- `git push origin master`
  - 将 local branch master 同步到 remote branch master
- `git push`
  - 上个命令的缩写，遵循 `.git/config`

push remote

- `git push origin master`
  - 将 local branch master 同步到 remote branch master
- `git push`
  - 上个命令的缩写，遵循 `.git/config`
- 一个新的 branch 并执行 `git push -u origin :new-branch`
  - 在 remote 添加一个新的 ref
  - 同时设为该 branch 的 upstream
  - 现在可以看看你的 `.git/config`

merge

merge

- 你有 commits 了，你有 refs 了
- 你如何 merge（合并）refs/branches 到一起呢？
- 会议：branch 指向一个 commit，一个 commit 包含一个指定的 tree
- 即我们需要 merge tree，也就是我们需要先 merge blob
- 如何 merge blob?

two way merge

- Two way 意味着算法只关心两个部分 (自己和其他)
- 以 `chapter6.md` 文件为例
- `fileB` 和 `fileC` 的 two way merge
  - fileC 在第一行没有 B 并在最后一行有 C
  - fileB 有第一行的 B 且最后一行没有 C
  - 不知道如何合并，终止
- 这并不实用

three way merge

- Three way merge 意味着算法关心三个部分 (base, our and their)
- `fileB`, `fileC` 和以 `fileA` 为 base 的 three way merge
  - 相较于 `fileA`, `fileB` 在第一行加了 B
  - 相较于 `fileA`, `fileC` 在最后一行加了 C
  - 变化并无冲突
  - `git merge-file --stdout <our> <base> <their>`
  - `git merge-file --stdout fileC fileA fileB`

```
lineBB
...some stuff...
lineCC
```

three way merge (cont'd)

- 那么如果两段都对同一行有修改呢？冲突!
- 通常需要手动干涉
- 栗子 `git merge-file --stdout fileD fileA fileB`
  - 相较于 `fileA`, `fileD` 在第一行加了 D
  - 相较于 `fileA`, `fileB` 在第一行加了 B
  - Output

```
<<<<<<< fileD
lineBD
=======
lineBB
>>>>>>> fileB
...some stuff...
lineC
```

如何解决冲突

- 移除 healper line
- 保留该保留的内容

```
lineBBD
...some stuff...
lineC
```

- 或者，如果你清楚你在做什么
  - `git merge-file --ours --stdout fileD fileA fileB`
  - 保留我们的（our）的修改，移除他们的（their）的修改
  - `git merge-file --theirs --stdout fileD fileA fileB`
  - 保留他们的修改，移除我们的修改
  - `git merge-file --union --stdout fileD fileA fileB`
  - 保留两段的修改，并连接到一起