Learn Git The Not So Super Hard Way[1]

Zenithal

2022-03-31

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

Why this

- Learning git is painful
  - Too many concepts (commit, branch, stage, index)
  - Too many commands (clone, pull, push)

Why this

- Learning git is painful
  - Too many concepts (commit, branch, stage, index)
  - Too many commands (clone, pull, push)
- State machine too complex
  - You often do not know what state you are in
  - Conflict! Help me ERIN!

Why this

- Learning git is painful
  - Too many concepts (commit, branch, stage, index)
  - Too many commands (clone, pull, push)
- State machine too complex
  - You often do not know what state you are in
  - Conflict! Help me ERIN!
- Learning about commands is not enough
  - You often do not know what's going on
  - Let's break it down to basic elements

Why this

- Learning git is painful
  - Too many concepts (commit, branch, stage, index)
  - Too many commands (clone, pull, push)
- State machine too complex
  - You often do not know what state you are in
  - Conflict! Help me ERIN!
- Learning about commands is not enough
  - You often do not know what's going on
  - Let's break it down to basic elements
- The super hard way is super easy
  - You change the file, you know what's going on

init

git repo structure

- git repo
  - often the `.git`
  - often contains HEAD, config
- worktree
  - the file
  - often contains README.md, main.c, main.h
  - worktree is just a checkout of the git repo
  - you can re-contruct your worktree from the git repo
  - the git repo is essential, but worktree is not

git init

- `mkdir .git`
- `mkdir .git/objects`
  - Must have
- `mkdir .git/refs`
  - Must have
- `echo 'ref: refs/heads/master' > .git/HEAD`
  - Establish HEAD ref
  - HEAD points to `.git/refs/heads/master` (Even though it does not exist now)
  - Side note: `refs/heads/main`
- config, hooks, info, etc are not necessary
- Now you can `git status` to check the status

objects

objects

- You have created `.git/objects`, then what are objects

objects

- You have created `.git/objects`, then what are objects
- Four types of objects

objects

- You have created `.git/objects`, then what are objects
- Four types of objects
  - blob: file content

objects

- You have created `.git/objects`, then what are objects
- Four types of objects
  - blob: file content
  - tree: folder
    - Side note: what's in folder in file system
    - filename (stored here instead of in blob!)
    - hash of blobs/trees (folder structure!)

objects

- You have created `.git/objects`, then what are objects
- Four types of objects
  - blob: file content
  - tree: folder
    - Side note: what's in folder in file system
    - filename (stored here instead of in blob!)
    - hash of blobs/trees (folder structure!)
  - commit: a state of the root folder
    - contains one specific tree
    - parent(s): other commit(s)
    - author/committer/commit message: meta data

objects

- You have created `.git/objects`, then what are objects
- Four types of objects
  - blob: file content
  - tree: folder
    - Side note: what's in folder in file system
    - filename (stored here instead of in blob!)
    - hash of blobs/trees (folder structure!)
  - commit: a state of the root folder
    - contains one specific tree
    - parent(s): other commit(s)
    - author/committer/commit message: meta data
  - tag: will not introduce today

blob

- blob: file content

blob

- blob: file content
- `echo 'hello' | git hash-object -t blob --stdin -w`
  - Write a blob/file whose content is `'hello'`
  - hash that content to an object in type `blob` from `stdin` then write to the object database
  - Output: `ce013625030ba8dba906f756967f9e9ca394464a`, the hash of the object

blob

- blob: file content
- `echo 'hello' | git hash-object -t blob --stdin -w`
  - Write a blob/file whose content is `'hello'`
  - hash that content to an object in type `blob` from `stdin` then write to the object database
  - Output: `ce013625030ba8dba906f756967f9e9ca394464a`, the hash of the object
- `cat .git/objects/ce/013625030ba8dba906f756967f9e9ca394464a`
  - Output: `xKORO0cH`, compressed content of hello
  - note the object path!

blob

- blob: file content
- `echo 'hello' | git hash-object -t blob --stdin -w`
  - Write a blob/file whose content is `'hello'`
  - hash that content to an object in type `blob` from `stdin` then write to the object database
  - Output: `ce013625030ba8dba906f756967f9e9ca394464a`, the hash of the object
- `cat .git/objects/ce/013625030ba8dba906f756967f9e9ca394464a`
  - Output: `xKOR0cH`, compressed content of hello
  - note the object path!
- Check the actual content

```
$ 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)

- Painful using raw command? Of course we have higher level instructions
- `git cat-file blob ce01`
  - Output: `hello`
- `git show ce01`
  - Output: `hello`

tree

- tree: folder

tree

- tree: folder
- Create a 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: folder
- Create a 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
```

- You can also (another format)

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

tree (cont'd)

- Directly inspect file content

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

tree (cont'd)

- Directly inspect file content

```
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)

- Directly inspect file content

```
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` (Compare with mktree above)

tree (cont'd)

- Directly inspect file content

```
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` (Compare with mktree above)
- `git show 5841` (A more simple version)

commit

Directly create file

```
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)

Or from high level command

```
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
```

That's why you need to `git config --global user.email` and `user.name`

commit (cont'd)

- Directly inspect file content

```
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)

- Directly inspect file content

```
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)

- Directly inspect file content

```
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` (A more simple version, in diff format)
- Note: commits are snapshots, not diffs/patchs[2]

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

Lucky commit

- Feeling hash too boring?
- Try lucky commit![3]
- ```
  $ git log
  1f6383a Some commit
  $ lucky_commit
  $ git log
  0000000 Some commit
  ```
- Note the commit msg in the prev slide, we can change it to mine a lucky hash

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

ref

ref

- ref is a convenient reference to one specific commit/other ref
- in `.git/ref`
- two types of ref
  - direct ref
  - indirect ref, e.g. `HEAD` (often the case)
- two common refs we will introduce today
  - heads: local branch
  - remotes: remote branch

local branch and direct ref

- Create file (not recommended as no reflog)

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

local branch and direct ref

- Create file (not recommended as no reflog)

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

- The following command will leave reflog in `.git/log/refs/heads/br1`
- `git update-ref --no-deref -m 'Reason for update' refs/heads/br1 d4da`
- `git branch -f br1 d4da`

about reflog

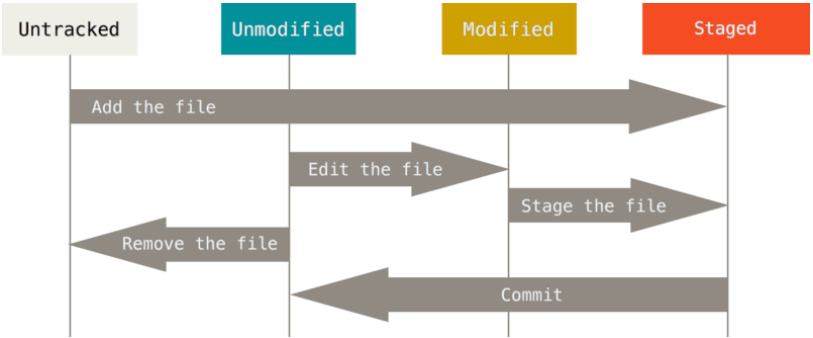
- Record all the changes to your ref
- Useful when you accidently switch to another place
  - `git rebase master`
  - `git checkout -B master origin/master`
  - then you want to switch to old tree for some reason
  - reflog shows the commit that one ref **was**
- Demo of my working dir: lots of reflogs

indirect ref

- Remember when you init
  - `echo 'ref: refs/heads/master' > .git/HEAD`
- This format is indirect ref

index

index



- index stores what to be commited when you `git commit`
- file at `.git/index`
- often we call things in index as staged (the figure above)
- a complex database
- contains many things, like filename, mode, hash, mtime, etc

manipulate index

- it is hard to manupulate index
- we study common cases here
- `git add` stores the content into index
- mark them ready for commit

manipulate index

- it is hard to manupulate index
- we study common cases here
- `git add` stores the content into index
- mark them ready for commit
- 1. `git add; git status`
  - the file you added is ready for commit

manipulate index

- it is hard to manupulate index
- we study common cases here
- `git add` stores the content into index
- mark them ready for commit
- 1. `git add; git status`
  - the file you added is ready for commit
- 2. `git add; modify; git status`
  - the file content you added is ready for commit
  - the new file content you did not add is not visible to index
  - `modify` will not be contained in commit

manipulate index

- it is hard to manupulate index
- we study common cases here
- `git add` stores the content into index
- mark them ready for commit
- 1. `git add; git status`
  - the file you added is ready for commit
- 2. `git add; modify; git status`
  - the file content you added is ready for commit
  - the new file content you did not add is not visible to index
  - `modify` will not be contained in commit
- 3. `git add; rm; git status; git restore`
  - even though file is deleted, it has a copy in `index`
  - if you accidently `rm -rf *`, you can restore your file!

switch/checkout

switch/checkout

- Recall that `.git/HEAD` is a ref
- This ref is for your worktree
- Recall your worktree is your actual content
- Change the content of your worktree by manipulating HEAD

switch/checkout (cont'd)

- Most famous: `git checkout master`
  - Make HEAD point to `refs/heads/master`
  - Then checkout the content to your worktree
  - That's why it is named checkout
  - Actually an old syntax, recommend using switch now
  - `git switch master`

switch/checkout (cont'd)

- Most famous: `git checkout master`
  - Make HEAD point to `refs/heads/master`
  - Then checkout the content to your worktree
  - That's why it is named checkout
  - Actually an old syntax, recommend using switch now
  - `git switch master`
- Yet most famous: `git reset --hard HEAD~1`
  - Change HEAD to HEAD~1 (the former commit of HEAD)
  - Checkout the content to your worktree
  - Note: there are `reset --soft/--mixed`, learn them by yourself

pull/clone/push

remotes

- Recall that we have talked about `.git/refs/remotes`
- Since we have local ref(branch), we can also have remote ref(branch)
- If no remote branch, it is not a distributed version control system
- How to sync them?
- pull commit from remote to local
- push commit from local to remote
- So the concept of commit is very useful

config remote

- If you want to have remote branch, you must have a remote first
- edit `.git/config` to add them
- or `git remote add origin git@github.com:xxx/yyy`
  - `origin` is a convention, you can use other name
  - You can have multiple remote
- Demo of my repo

fetch remote

- `git fetch origin master`
  - Fetch the `master` ref from `origin`
  - You can check `.git/refs/remotes/origin/master` now

fetch remote

- `git fetch origin master`
  - Fetch the `master` ref from `origin`
  - You can check `.git/refs/remotes/origin/master` now
- `git pull origin master`
  - despite `git fetch`, it tries to update your local ref
  - Update `.git/refs/remotes/origin/master`
  - and update `.git/refs/heads/master` accordingly
  - The relationship is recorded in `.git/config`

fetch remote

- `git fetch origin master`
  - Fetch the `master` ref from `origin`
  - You can check `.git/refs/remotes/origin/master` now
- `git pull origin master`
  - despite `git fetch`, it tries to update your local ref
  - Update `.git/refs/remotes/origin/master`
  - and update `.git/refs/heads/master` accordingly
  - The relationship is recorded in `.git/config`
- `git pull`
  - short hand for the above, according to your `.git/config`

fetch remote

- `git fetch origin master`
  - Fetch the `master` ref from `origin`
  - You can check `.git/refs/remotes/origin/master` now
- `git pull origin master`
  - despite `git fetch`, it tries to update your local ref
  - Update `.git/refs/remotes/origin/master`
  - and update `.git/refs/heads/master` accordingly
  - The relationship is recorded in `.git/config`
- `git pull`
  - short hand for the above, according to your `.git/config`
- `git clone`
  - Actually a short hand for
  - `git init`
  - `git remote add`
  - `git pull`

push remote

- `git push origin master`
  - Sync your local branch master to remote branch master

push remote

- `git push origin master`
  - Sync your local branch master to remote branch master
- `git push`
  - short hand for the above, according to your `.git/config`

push remote

- `git push origin master`
  - Sync your local branch master to remote branch master
- `git push`
  - short hand for the above, according to your `.git/config`
- New branch then `git push -u origin :new-branch`
  - Add a new ref in the remote
  - At the same time set the upstream to new-branch
  - Check your `.git/config` now

merge

merge

- Now you have commits, you have refs
- How do you merge refs/branches together?
- recall that a branch points to a commit, a commit contains a specific tree
- Namely we need to merge tree, then we need to merge blob first
- How to merge blob?

two way merge

- Two way means the algo can only see two files (our and their)
- Let's setup the file as `chapter6.md`
- Two way merge of `fileB` and `fileC`
  - The change can be fileC has removed B in the first line and added C in the last line
  - The change can be fileB has added B in the first line and deleted C in the last line
  - Do not know how to merge, abort
- It is not useful

three way merge

- Three way merge means the algo can see three files (base, our and their)
- Three way merge of `fileB` and `fileC` with `fileA` as base
  - Compared with `fileA`, `fileB` added B in the first line
  - Compared with `fileA`, `fileC` added C in the last line
  - No conflict in changes
  - `git merge-file --stdout <our> <base> <their>`
  - `git merge-file --stdout fileC fileA fileB`

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

three way merge (cont'd)

- What if they both modify the same line? Conflict!
- Usually need manual involvement
- E.g. `git merge-file --stdout fileD fileA fileB`
  - Compared with `fileA`, `fileD` added D in the first line
  - Compared with `fileA`, `fileB` added B in the first line
  - Output

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

How to resolve conflict

- Remove all the helper line
- Leave the actual content

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

- Or if you are aware of what you are doing
  - `git merge-file --ours --stdout fileD fileA fileB`
  - Keep our change, discard theirs
  - `git merge-file --theirs --stdout fileD fileA fileB`
  - Keep their change, discard ours
  - `git merge-file --union --stdout fileD fileA fileB`
  - Keep both changes, concat them