Learn Git in 30 days

资源来自于 Learn Git in 30 days

记录学习中的一些笔记，由于当前认知有限，可能会自以为不重要忽略掉一些内容，所以对于读者来说权当是个类似于大纲或者关键词索引之类的东西（当然忽略掉的东西以后会补起来的）

[Day 1] 认识 Git 版本控管

> 分散式版本控制管理工具

学习方法

- 多使用命令行
- 找多一点的人一起学（体验多人协作）
- 分散式版本控管，每个人都有一份完整的 Repository (下面简写为 `repo`)，所以经常要合并
- 有合并就有冲突，学会解决

Git 的几个重要设计

- 非线性开发模式（分散开式发模式）
  - 分支与合并机制
  - 历史变更路径
- 分散式开发模型
  - 每个人都有完整的开发历史记录
  - 每个人都有完整的 repo
  - 变更过的档案与历史都仅在本地 `repo`
- 兼容现有操作系统
  - Git 其实就是个文件夹加上一些配置
  - Git 发布方式：HTTP，FTP，rsync，SSH 等
- 高效处理大型文件
- 历史记录保护
  - hash id
- 以工具集为主的设计
- 弹性的合并策略
- 被动的垃圾回收机制
  - 可以中断当前操作或者恢复上一个操作
  - 手动清理无用文件：`git gc --prune`
- 定期封装物件
  - 可以将老旧的档案自动封装进一个封装档(packfile)中，以改善档案储存效率
  - 手动封装：`git gc`
  - 检查档案系统是否完整：`git fsck`

[Day 2] 在 Windows 平台必装的三套 Git 工具

- Git for Windows
- GitHub for Windows

- SourceTree

Git for Windows

- 查看 Git 版本：`git --version`

GitHub for Windows

- GitHub SSH key

注意：

- 在 PowerShell 中 `{}` 有特殊的意义，所以如果 git 参数用到 `{}` 的话，要在该参数的前后加上'单引号'

SourceTree

这个的界面真实相当不错呀

TortoiseGit

[Day 3] 建立储存库

- local repo
- shared repo
- remote repo

本机建立本地储存库(local repo)

- 预设资料夹 `%USERPROFILE%\Documents\GitHub`
- 工作目录 `working directory`
- 建立储存库 `git init`

本机建立共用储存库(shared repo)

> 建立一个 Git 储存库，但是不包含工作目录，别名叫做 “裸储存库(bare repo)”

- 建立共用 repo：`git init --bare`

注意，这个资料夹不能直接拿来做开发用途，只能用来储存 Git 相关资讯，同时最好不要手工遍历这个资料夹下的文件

[不是很理解这句话的意思]当你想要建立一个工作目录时，必须先取得这个裸储存库的内容回来，这时必须使用 `git clone [repo_URI]` 复制一份回来才行，而透过 git clone 的过程，不但会自动建立工作目录，还会直接把这个裸储存库完整复制过来

使用 bare repo 的情况：

- 在一台多人使用的机器上进行协同开发，可开放大部分人对这个 bare repo 资料夹的只读权限，只让一个人或少许人才有写入权限
- 有些人会把 bare repo 放到 Dropbox 上跟自己的多台电脑同步

在远端建立远端储存库(remote repo)

> remote repo 和 bare repo 几乎是一样的

- 复制远端储存库：`git clone [repo_URL]`

[Day 4] 常用的 Git 版本控管指令

- 新增 `git init`
- 删除 `git rm file`
- 重新命名 `git mv curr target`
- 提交变更 `git commit -m 'msg'`
- 查询历史记录 `git log`

新增

- 全部：`git add .`
- 部分：`git add file`

颜色状态：

```
- Untracked 未追踪
- Staged 准备好的档案
```

提交变更

- 提交变更：`git commit -m 'msg'`

必须拥有版本记录说明文字

查询历史

- 限定输出的历史记录数量：`git log -10`

删除

- 删除：`git rm 'file'`

注意：

```
- 删除工作目录快取的 file 这个档案
- 删除工作目录下的 file 这个**实体**档案
```

重新命名档案或目录

- 更名：`git mv currName targetName`

显示工作目录索引状态

- 详情：`git status`

- 精简：`git status -s`

重置目前工作目录

- 重置目前工作目录索引状态：`git reset`
- *工作目录*还原到最新版本：`git reset --hard`

还原某一个档案

把 master 分支中最新的 file 还原，同时也会复原工作目录的索引状态

当档案编辑到一般时，发现改坏了，要恢复到修改前的版本：

- 还原档案为未修改前的版本：`git checkout master file`

[Day 5] 了解储存库、工作目录、物件与索引之间的关系

- 储存库
- 工作目录
- 资料结构
    - 物件
        - blob 物件
        - tree 物件
    - 索引

储存库

`.git` 文件夹就是一个完整的 Git 储存库，未来所有版本的变更都会自动存储在这个资料夹中

工作目录

工作目录 就是我们正在准备开发的文件夹，以后所有的事情都会在这个文件夹下进行

- 有些操作比如 `git checkout` 会更新工作目录下的档案
- 适时保持工作目录干净

Git 资料结构

Git 物件

> 保存版本库中所有 档案与版本记录

物件是一个特别的档案，是将一个档案的内容取出，通过内容产生出一组 SHA1 值，然后依照这个 SHA1 值来命名一个档案

所有要进行控管的目录和档案都会先区分 目录资讯 和 档案内容，我们称为 `tree` 物件 与 `blob` 物件

blob 物件就是档案内容当成 blob 档案内容，然后进行 SHA1 计算吃 hash id，再把这个 id 当作 blob 文件名

tree 物件就是储存特定资料夹下包含哪些档案，以及该档案对应的 blob 物件的档案名为什么，tree 物件可以包含 blob 物件的档案名和相关资讯，同时也可以包含其他的 tree 物件

这些物件都储存在物件储存区，也就是储存库的 `object` 目录

索引

> 保存当下 要进版本库之前的目录状态

索引档案为 .git 文件夹下的 `index` 文件，保存着 Git 储存库中特定版本的状态。其目的主要用来记录哪些文件要被提交到下一个 commit 中。

> Git 索引是介于物件储存区和工作目录之间的媒介

[Day 6] 解析 Git 资料结构 - 物件结构

> 物件是一种不可变的档案类型，物件储存区的档案通常只进不出。每个物件都是以档案内容进行 SHA1 计算出的一个 hash 值。

物件类型

- blob 物件：工作目录中某个档案的内容
- tree 物件：储存特定目录下的所有资讯，包含该目录下的档案名，对应的 blob 物件名称，档案连接或其他 tree 物件
- commit 物件：记录哪些 tree 物件包含在版本中，一个 commit 物件代表 Git 一次提交
- tag 物件：关联一个特定 commit 物件

[Day 7] 解析 Git 资料结构 - 索引结构

> 用来记录有哪些档案即将要被提交到下一个 commit 版本中

别名： - Index - Cache - Directory cache - Current directory cache - Staging area - Staged files

阶段： - untracked - unmodified - modified - staged

> Git 储存库的运作，是将工作目录里的变化，透过更新索引的方式，将资料写成 Git 物件

`git add -u`：仅将更新或删除的档案变更写入到索引

`git rm fileName`：删除工作目录下的档案并更新索引

`git rm --cached fileName`：仅删除索引档中的文件，不会删除工作目录下的文件

`git ls-files`：列出所有目前已经储存在索引档中的那些档案路径

[Day 8] 关于分支的基本观念与使用方式

分支机制就是为了解决开发中的版本冲突问题，同时也会引发冲突

- 建立分支：
  - `git branch name`
  - `git checkout -b name`
- 切换分支：
  - `git checkout name`
- 删除分支：

    - `git branch -d name`

[Day 9] 对比档案与版本差异

> `git diff old new` 以 tree 物件为比较单位

四种基本比较方式

- `git diff` 比较工作目录与索引
- `git diff commit` 比较工作目录与指定 commit 物件里的 tree 物件
    - `git diff HEAD` 比较工作目录与当前分支最新版
- `git diff --cached commit` 比较索引与指定 commit 物件里的 tree 物件
    - `git diff --cached HEAD` 比较索引与当前分支最新版
- `git diff commit1 commit2` 比较指定两个 commit
    - `git diff HEAD^ HEAD` 比较最新版的前一版与最新版

三种 tree 物件来源

- 任意版本中任意 tree
- 索引
- 目前工作目录

[Day 10] 认识 Git 物件的绝对名称

> 物件 id 就是绝对名称

- `git cat-file - commitid` 查看 commit 物件内容
- `git log --pretty=oneline` 查看精简版本历史记录
- `git log --pretty=onelien --abbrev-commit` 精简版部分绝对路径

[Day 11] 认识 Git 物件的 一般参照 与 符号参照

> 参照名称：用一个预先定义或者自行定义的名称来代表一个 Git 物件（类似于 Git 物件的别名），通常指向一个 commit 物件，但也可以指向 blob/tree/tag 物件。所有参照名称都是个纯文字档案，指向版本历史记录中的最新版

参照名称位置：

- 本地分支：`.git/refs/heads/`
- 远程分支：`.git/refs/remotes/`
- 标签：`.git/refs/tags/`

`git show commitid` 取得该版本的变更记录

一般参照（指向一个物件的绝对路径）

如果 newbranch 是 0bd0 的参照名称，那么：

`git cat-file -p newranch` 和 `git cat-file -p 0bd0` 的结果是一样的

符号参照（指向另一个一般参照）

> 符号参照会指向另一个参照名称，内容以 `ref:` 开头

- HEAD 永远指向工作目录中所设定的分支的最新版本
- ORIG_HEAD HEAD 这个 commit 物件的前一版
- FETCH_HEAD 记录远端储存库中每个分支的 HEAD 最新版的绝对路径
- MERGE_HEAD 当执行合并工作时，合并来源的 commit 物件绝对名称会被记录在 MERGE_HEAD 这个符号参考中

使用方式

`git update-ref 一般参照名称 绝对路径` 自由建立一般参照。此时，绝对名称和参照名称都能存取特定物件内容

如果要建立较为正式的参照名称，最好使用 `refs/` 开头

`git update-ref -d 一般参照名称` 删除一般参照

`git show ref` 显示所有参照

[Day 12] 认识 Git 物件的相对名称

没有分支与合并的储存库中

`^` 和 `~` 符号表示前一版

存在分支与合并的储存库中

- `^` 拥有多个上层 commit 物件时，要代表的第几个第一代上层物件
- `~` 代表第一个上层 commit 物件

`git rev-parse` 把任意参考名称或相对名称解析为绝对名称

[Day 13] 暂存工作目录与索引的变更状态

> 透过 `git stash` 将改写到一半的档案建立一个特殊版本（也是一个 commit 物件）。储存完毕后这些变更都会被重置，返回到 HEAD 状态

建立暂存版本

- `git stash` 所有已列入追踪的档案建立暂存版
- `git stash -u` 所有已追踪或未追踪的档案全部建立暂存版

它具有三个 parent：

- 原本工作目录的 HEAD 版本
- 原本工作目录理所有追踪中的版本（在索引中）
- 原本工作目录理所有未追踪的内容（不在索引中）

取回暂存版本

`git stash pop` 取回最近一笔暂存版并合并，暂存版删除

建立多重暂存版

`git stash list` 目前所有 stash 清单 `git stash --save -u <msg>` 自定义信息 `git stash apply` 取回版本，暂存版还在清单里 `git stash apply "stash@{1}"` 取回特定版本 `git stash drop "stash@{1}"` 删除特定暂存版

[Day 14] Git for Windows 选项设定

储存 Git 选项设定的三个地方

- 系统级：`--system` 设定于整台电脑
- 用户级：`--global` 设定于当前用户
- 储存区级：`--local` 设定于当前储存区

选项应用顺序：

- 先应用系统层级（优先级低）
- 再应用使用者层级
- 再应用储存区层级（优先级高）

常用选项设定：

- 别名：`git config --global alias.co commit`
- 预设编辑器：`git config --global core.editor notepad.exe`
- 直接编辑：`git config --edit --system`
- 自动修正错误参数：`git config --global help.autocorrect 1`
- 自定义讯息范本：`git config --local commit.template 'G:\git-commit-template.txt'`

常用别名：

- git config --global alias.co checkout
- git config --global alias.cm commit
- git config --global alias.st status
- git config --global alias.sts "status -s"
- git config --global alias.br branch
- git config --global alias.re remote
- git config --global alias.df diff
- git config --global alias.type "cat-file -t"
- git config --global alias.dump "cat-file -p"
- git config --global alias.lo "log --oneline"
- git config --global alias.ll "log --pretty=format:'%h %ad | %s%d [%Cgreen%an%Creset]' --graph --date=short"
- git config --global alias.lg "log --graph --pretty=format:'%Cred%h%Creset %ad |%C(yellow)%d%Creset %s %Cgreen(%cr)%Creset [%Cgreen%an%Creset]' --abbrev-commit --date=short"

[Day 15] 标签 - 标记版本控制过程中的重要事件

> 以一个“好记”的名称来帮助我们标记某一个版本（commit 物件）

- 轻量标签：某个 commit 版本的别名，是一种相对名称
  - 建立轻量标签：`git tag tagName`
  - 删除轻量标签：`git tag -d tagName`
- 标识标签：是 Git 物件的一种，即 Tag 物件
  - 建立标识标签：`git tag 1.0.0-beta -a -m "Beta version"`，预设会将当前的 HEAD 版本建立成标签物件

[Day 16] 善用版本日志 git reflog 追踪变更轨迹

> 只要修改了任何参照（ref）的内容，或是变更任何分支的 HEAD 参照内容，就会建立历史记录

修改参照的命令，即导致产生新 reflog 的命令有：commit, checkout, pull, push, merge, reset, clone, branch, rebase, stash，每个分支，每个暂存版都有自己的 reflog

可以用 git reset 复原变更：`git reset --hard HEAD@{1}`

显示 reflog 的详细版本记录：`git log -g`

删除特定版本历史记录：`git reflog delete ref@{specifier}`，不会影响到储存库中的任何内容

设定历史记录过期时间：

- `git config --global gc.reflogExpire "never"`
- `git config --global gc.reflogExpireUnreachable "never"`

清除历史记录：`git reflog expire --expire=now --all`

[Day 17] 关于合并的基本观念与使用方式

合并之前：

- 查看自己在哪个分支
- 确保工作目录干净

删除分支：

- git branch -d branchName
- git branch -D branchName

救回误删分支：

- git branch branchName objectId

解决冲突的方法？

[Day 18] 修正 commit 过的版本历史记录 Part 1 (reset & amend)

- 做一个小功能修改就建立版本，这样才能容易追踪变更
- 千万不要累积一大堆修改后才建立一个“大版本”
- 有逻辑，有顺序的修正功能，确保相关的版本修正可以按顺序提交（commit），这样才便于追踪

- 删除最近一次版本：git reset --hard "HEAD^"
- 删除最近一次版本但保留变更：git reset --soft "HEAD^"
- 重新提交最后一个版本：git commit --amend msg

[Day 19] 设定 .gitignore 忽略清单

仅限于 Untracked files，对于已经 Staged files 则不受 .gitignore 控制

[Day 20] 修正 commit 过的版本历史记录 Part 2 (revert)

> `git revert` 把某个版本的变更，透过相反的步骤把变更还原回来，实际上就是合并，在没有冲突的情况下会自动 commit

- git revert commitId

revert 失败：

处理方式类似于合并分支失败情况

使用 git revert 命令执行变更，但不执行 commit 动作：

- git revert -n commitId
- git revert --continue
- git revert --abort

[Day 21] 修正 commit 过的版本历史记录 Part 3 (cheery-pick)

> 当你在一个分支中开发了一段时间，但后来决定整个分支都不要了，不过当中却有几个版本想要留下来。挑选一个或者多个版本，然后套用在目前分支的最新版本上

- git cherry-pick commitId，如果成功，则会在目前 master 上建立一个新版本，但是版本的 Author 和 Date 信息和以前一样
- git cherry-pick -x commitId，添加 `cherry picked from ...` 信息，类似于 revert
- git cherry-pick -e commitId，建立版本之前先编辑信息
- git cherry-pick -n commitId，不建立版本，然后自行 commit

[Day 22] 修正 commit 过的版本历史记录 Part 4 (rebase)

> Base 代表基础版本的意思，表示你想要重新修改特定分支的基础版本，把另外一个分支的变更，当成我这个分支的基础

注意：

- 工作目录必须是干净的
- 不是远端分支
- git rebase master：将 branch1 分支的最新版本(HEAD)倒带(revind)到跟 master 一样的分支起点(revinding head)，然后再重新套用(replay)指定 master 分支中所有版本。先套用 master 然后再套用 branch1。（将 branch1 倒带，然后在 master 的 HEAD 上套用 branch1 的每个版本？）

合并方式：

- 一般指令（Fast-forward）：git merge branch1，直接修改 master 分支的 HEAD 到 branch1 的 HEAD，此时 master 和 branch1 一样。
- 停用 Fast-forward（--no-ff）：git merge branch1 --no-ff，要合并的分支 branch1 先建立一个分支，然后再合并回 master。

注意通过 rebase 之后，分支的起点就不一样了。

[Day 23] 修正 commit 过的版本历史记录 Part 4 (rebase 2)

- git rebase -i commitID

rebase 能做的：

- 调换 commit 顺序：直接修改 pick 的顺序
- 修改 commit 讯息：pick 修改为 reword
- 插入 commit：pick 修改为 edit，然后 git commit --amend，git rebase --continue
- 编辑 commit：类似于 插入 commit，然后 git commit --amend
- 拆解 commit：类似于 插入 commit，然后将某些档案从索引状态移除，git commit --amend 建立新版本，然后 git add. git commit 等
- 压缩 commit（保留讯息）：pick 修改为 squash
- 压缩 commit（丢失版本记录）：pick 修改为 fixup
- 删除 commit：删除对应的 pick 行

[Day 24] 使用 GitHub 远端储存库

- 建立无版本库（无分支）：
    - 先 git clone 再建立版本上传：git push -u origin master
    - 直接上传现有本地库： - git remote add origin URL - 然后： - git pull origin master --allow-unrelated-histories 或者 - git fetch 再 git merge origin/master
- 建立有版本库（会冲突）：
    - 先 git clone 再建立版本上传：git push origin master
    - 直接上传现有本地库： - git remote add origin URL - 然后： - git pull origin master --allow-unrelated-histories 或者 - git fetch 再 git merge origin/master

[Day 25] 使用 GitHub 远端储存库 —— 观念篇

- git clone
- git pull
    - git fetch
    - git merge origin/master
- git push
- git fetch

- git ls-remote
- git remote add origin https://github.com/example.git
- git remote -v
- origin：预设远端分支的参照名称，代表一个远端储存库的 URL 位置

[Day 26] 多人在同一个远端储存库中进行版控

注意在每次 push 之前先 pull 或者 fetch 一次，以免与远端储存库不一致导致 push 失败

如果 push/pull 失败，可以使用 `git reset --hard HEAD` 来返回之前的最新版 如果合并成功了，但是又不想这次合并，可以使用 `git reset --hard ORIG_HEAD` 来重置合并之前的状态

[Day 27] 透过分支在同一个远端储存库中进行版控

常见分支：

- master
- develop
- feature/...
- hotfix/...

使用 `git push --all --tags` 推送所有本地建立的分支/标签 使用 `git fetch --all --tags` 取回所有远端的分支/标签

推送特定的分支：

- `git push origin hotfix/...`

[Day 28] 了解 GitHub 的 fork 与 pull request 版控

> fork 和 pull request 的存在主要是权限以及版本库隔离的需求

[Day 30] Git 操作小技巧

- 使用 ssh key 免输入账号密码
- 还原 rebase 变动
  - 找到 reabse 发生在哪个版本上，然后 reset 到那个版本前一版
- 取得远端储存库统计资讯
  - `git shortlog -sne` 详细列出每个人的 commit 次数
  - `git shortlog -sne -n` 按 commit 数量降幂排序
  - `git shortlog` 显示最近 commit 过的历史记录
- 从工作目录清除不在版本库中的档案
  - `git clean -n` 列出预期会删除的档案
  - `git clean -f` 删除不在版本库的档案
- 删除远端分支
  - `git push origin :RemoteBranch`
  - `git push origin --delete RemoteBranch`
- 找出改坏程式的凶手

    - `git blame fileName`
    - `git blame -L 开始行, 结束行 fileName`