

# Git & GitHub

2022-ZJU技能拾遗-@鹤翔万里 授课

## 参考

- [Git 教程 | 菜鸟教程 \(runoob.com\)](#)
- 参考书
  - 《Git 版本控制管理》[ISBN 978-7-115-38243-6](#)
    - *Version Control with Git*, Jon Loeliger
- git book
  - 中文版[Git - Book \(git-scm.com\)](#)
  - 英文版[Git - Book \(git-scm.com\)](#)
- learning git小游戏
  - [Learn Git Branching](#)
  - [azler/githubug](#)
- git reference[Git - Reference \(git-scm.com\)](#)
- 官网- <https://git-scm.com/>

## 相关工具

- gitui: Git TUI 之一 (Rust 实现)
- lazygit: Git TUI 之一 (Go 实现)
- gitoxide: Git 的纯 Rust 实现 (精简、高效、安全)

## 简介

### 发展历史

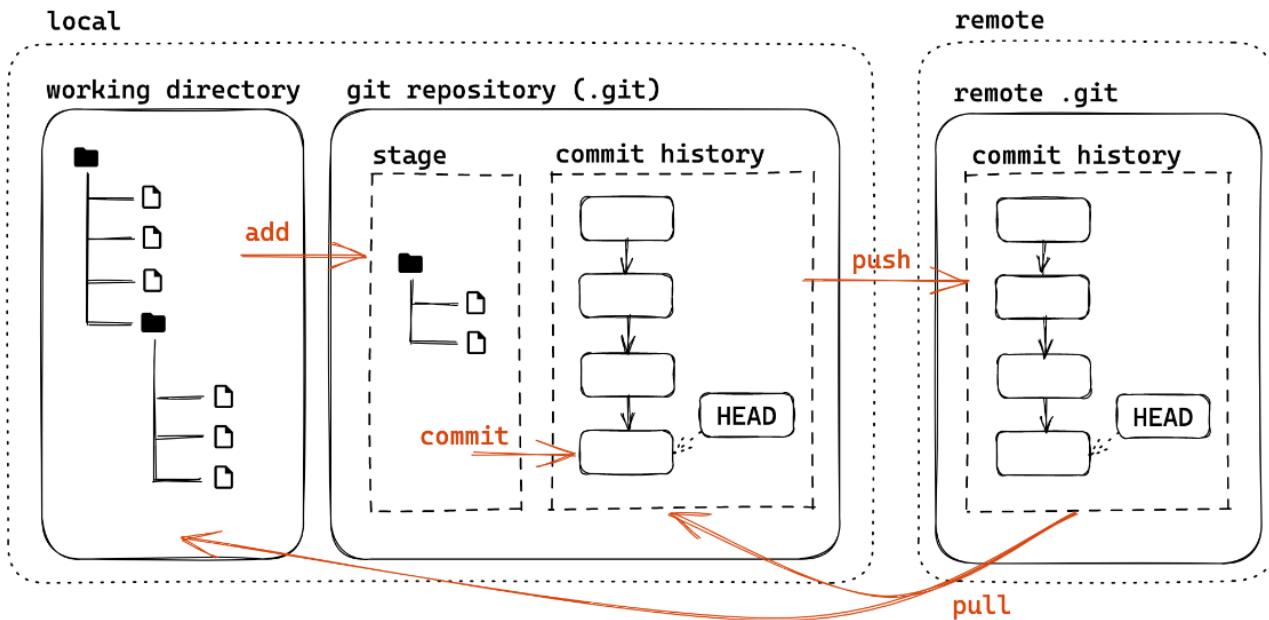
- Linus Torvalds 在开发 Linux 内核时由于当时使用的分布式版本控制系统 BitKeeper 对于免费版本加入了限制，于是开发了一款免费自由而且解决了历代 VCS 缺陷的版本控制系统 Git
- 2005.4.7, [Git 开始自托管](#); 九天后, [Linux 转为使用 Git 作为 VCS](#)

## 什么是Git

- 分布式版本控制系统 (DVCS, Distributed Version Control System)
  - 分布式: 无需联网, 本地
  - 版本控制: 可以回溯历史版本

## Git模型

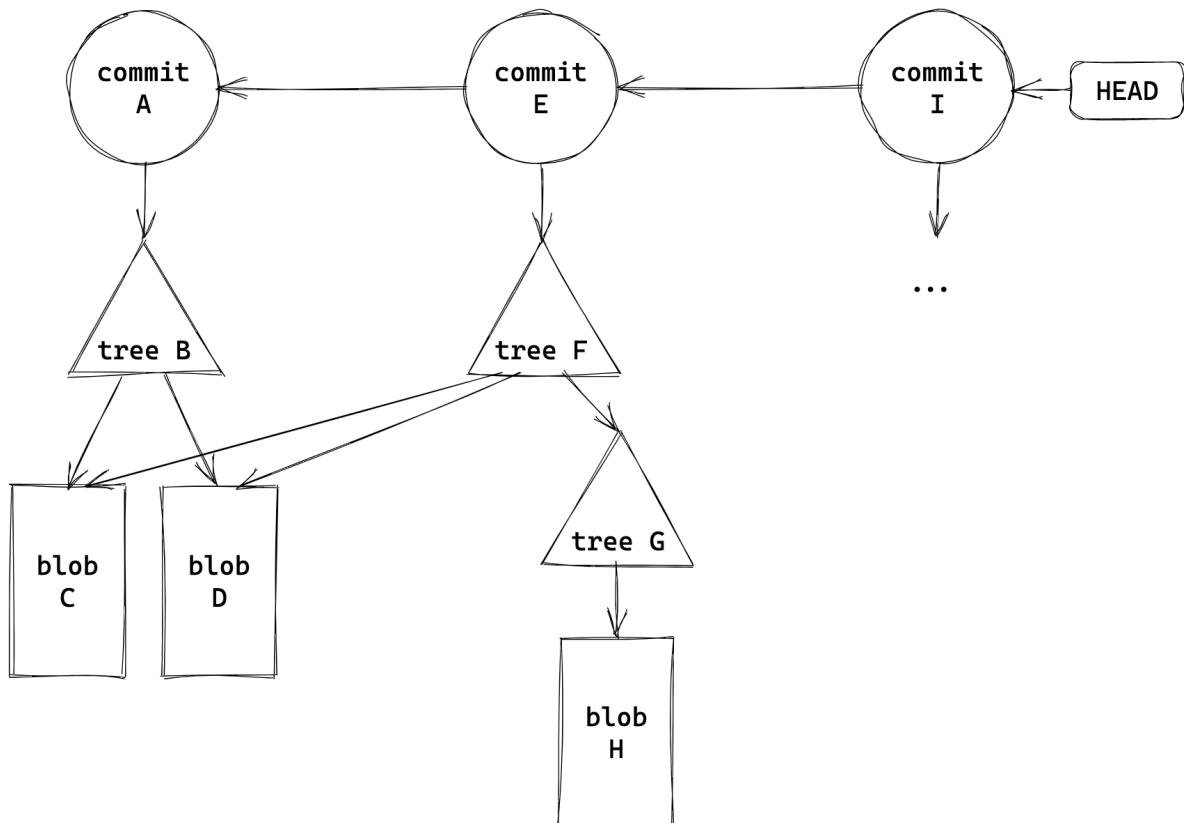
### 图解



## Git 结构

Git 到底是怎么记录的? .git 文件夹里到底都是什么?

- **.git/objects**: 存储的所有东西都在这里 !
- 文件名是对象的 sha1，且头一个字节作为一层目录 (加速文件系统)
- `git cat-file -p id` 查看对象内容 (-t 查看类型 commit/tree/blob)



- 分支指针 (就是内容为 sha1 的文件)
  - **.git/HEAD**: HEAD 指针，指向当前位置
  - **.git/refs**: 各种 ref 指针

- 子目录 heads/remotes/tags
- master -> refs/heads/master
- 因为分支名要作为文件名，所以要求：
  - 可以包含 / (用来分层) 但不能作为开头，/ 后面不能接 . (不能隐藏)
  - 不能包含 .. 不能包括空格或其他空白字符

## 安装与配置

### 安装

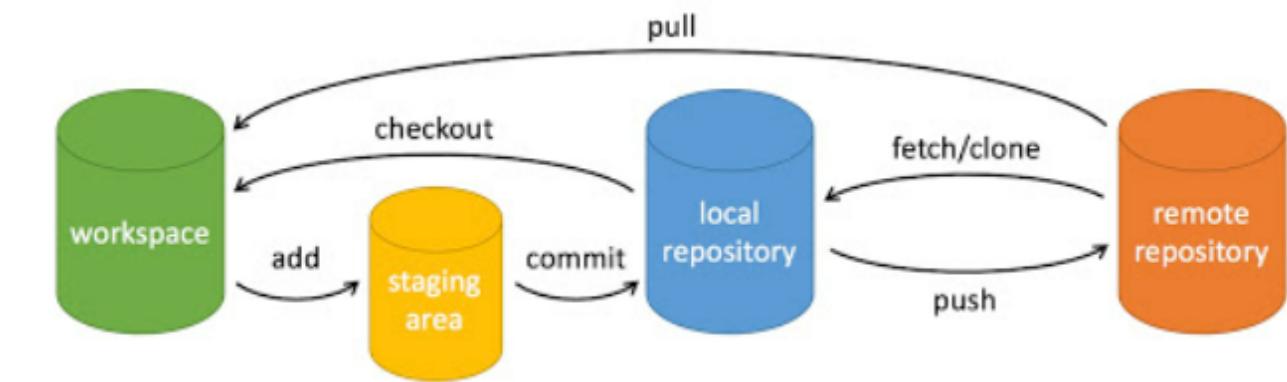
- Unix: 包管理器直接装
  - apt install git / brew install git / ...
- Windows:
  - <https://git-scm.com/download/win>
  - 内带 Git Bash:
    - 一个基于 MinGW 的类 Linux shell 环境
    - 包含了很多常用的但 cmd/powershell 缺少 (或用法差别很大) 的命令
  - 注意看安装时的选项，其中包括了一些使用范围的配置
    - 注意环境变量

### 配置

- 创建一个本地 git 版本库
  - 通过 git init 指令
    - git init: 让当前文件夹变成 git 仓库 (创建 .git 文件夹)
    - git init *folder*: 创建一个新的文件夹并初始化为 git 仓库
- git 账号配置
  - Why? 多人合作区分用户 / 让 GitHub 能够识别出你
  - 全局配置:
    - git config --global user.name "*name*"
    - git config --global user.email "*email*"
  - 针对某一版本库专门设置:
    - 同前，不加 --global

## Git 基础用法

### 常用的6个命令



- `git clone`、`git push`、`git add`、`git commit`、`git checkout`、`git pull`

## Git ignore

- 存放在版本库根目录下的名为 `.gitignore` 的文件，规定忽略哪些文件
- 语法
  - `#` 开头的行为注释
  - `*` 通配多个字符，`**` 通配中间目录（有或无）
    - `*.c` 匹配所有 C 文件，`a/**/b` 匹配 `a/b`、`a/x/b`、`a/x/y/b` 等
  - `/` 开头只匹配根目录，否则匹配所有目录
  - `!` 取消忽略
  - `...`
  - [Git - gitignore Documentation](#)
- `git check-ignore -v file`: 查看某个文件是否被忽略，以及匹配的规则
- 常用语言的 `.gitignore` 模板：[github/gitignore](#)

## Commit message

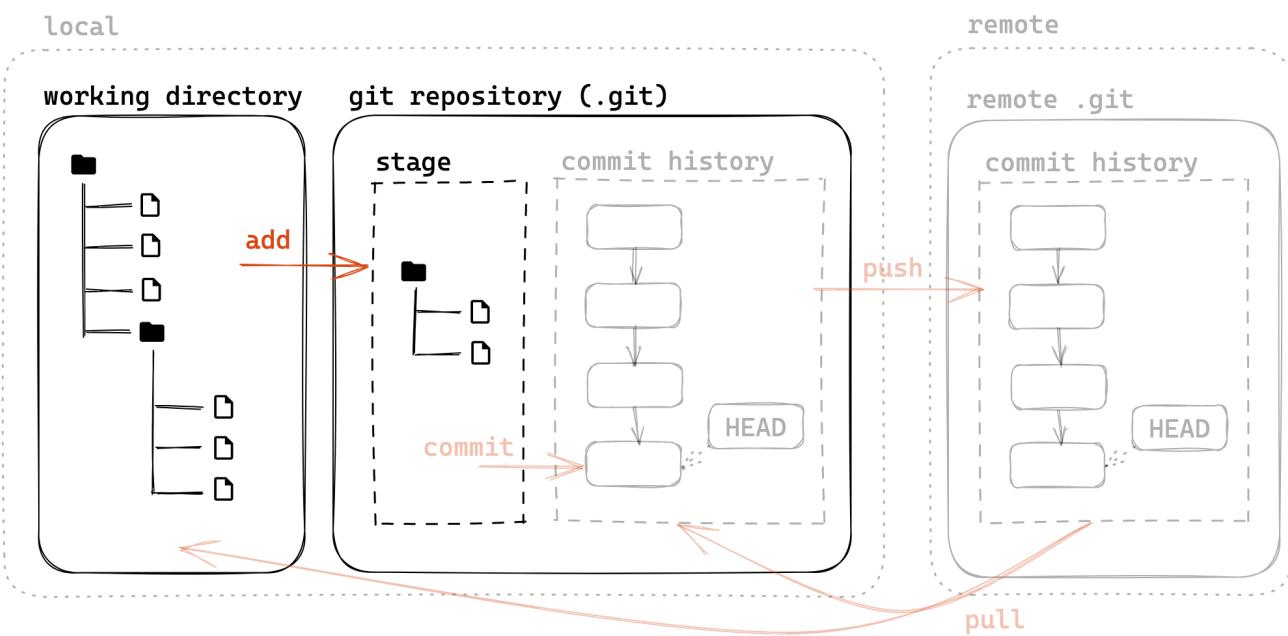
- 意义是什么：记录更改的原因 / 内容，方便定位 / 回溯（特别是合作项目）
- Angular 规范 (来源：[angular/angular:CONTRIBUTING.md](#))

```

<type>([scope]): <summary>
  [body]
  [footer]
  
```

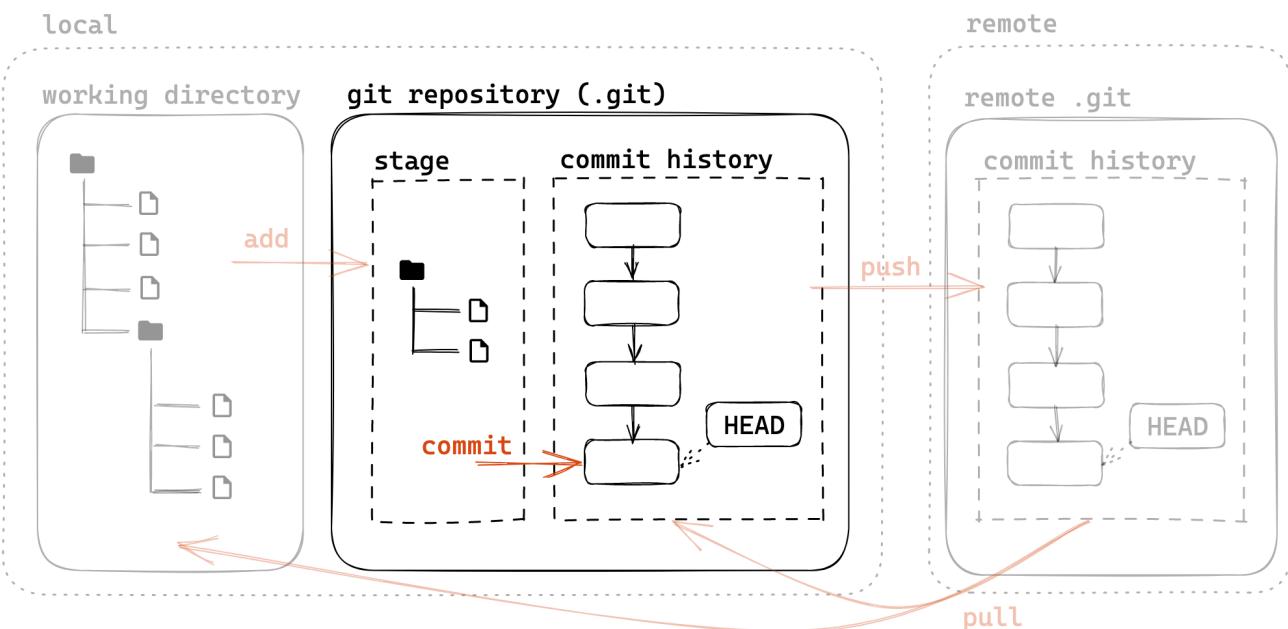
- type: 更改类型 (`fix/feat/docs/refactor/perf/test/ci/...`)
  - 重大更改可以写 `BREAKING CHANGE` 或 `DEPRECATED` (全大写)
- scope: 影响范围 (可选，比如具体影响的模块等)
- summary: 更改的简要描述，英文一般现在时，首字母小写句末无句号
- body: 详细描述，可选
- footer: 解决 issue 了可以写 `Fixes #_id_` 或 `Closes #_id_`

## 文件暂存 add



- 暂存区：已经修改、等待后续提交的文件
- 将文件加入暂存区：
  - `git add file/folder`
  - 只会添加修改过的文件
- 删除文件的几种情况：
  - 只在本地删除版本库中不存在的文件：`rm`
  - 同时删除本地和版本库中的文件：`git rm` / 先 `rm` 再 `add`
  - 将一个已暂存的新文件取消暂存：`git rm --cached`
- 重命名文件：`git mv`（等价于 `mv` + `git rm` + `git add`）
- 查看当前工作区和暂存区状态：`git status`
  - 文件三个类别：未跟踪（Untracked）、已追踪（Tracked）、被忽略（Ignored）

## 提交更改 commit



- 将暂存内容提交到本地仓库，生成一个新节点
  - `git commit`: 默认编辑器编辑提交信息
  - `git commit -m "message"`
  - `-a (--all)` 自动暂存所有更改的文件
- 查看提交历史: `git log`
  - `--oneline`: 每一个提交一行
  - `--graph`: 显示分支结构
  - `--stat`: 显示文件删改信息
  - `-p`: 显示详细的修改内容
- 每个提交都有一个唯一的 sha-1 标识符 (40 位十六进制数)
  - `git show id`: 显示提交详细信息 (*id* 在不重复前提下可以只写前几位)
- 检出之前的某一版本: `git checkout id`

## 版本控制

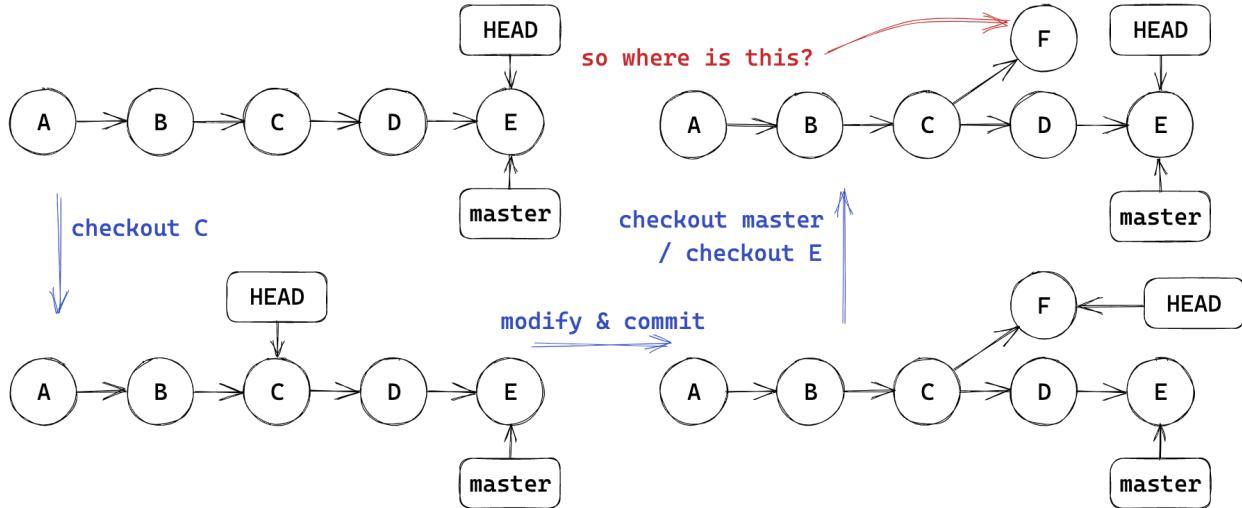
- 创建标签:
  - 轻量标签: `git tag tag id` (*id* 可选, 默认为 HEAD)
  - 附注标签: `git tag -a tag -m "message" id`
- 查看标签: `git tag`
- 版本号命名一般规范: [Semantic Versioning 2.0.0](#)
  - v主版本号.次版本号.修订号[-预发布版本号]
  - 修订号: 兼容修改, 修正不正确的行为
  - 次版本号: 添加新功能, 但是保持兼容
  - 主版本号: 不兼容的 API 修改
    - 且为 0 时表示还在开发阶段, 不保证稳定性
  - 预发布版本号: alpha/beta/rc.1/rc.2/...
  - e.g. v1.0.0-beta < v1.0.0-rc.1 < v1.0.0 < v1.0.1
- Teamwork, 但是要把自己的放在别人的前面
  - `git commit`
  - `git pull --rebase`
  - `git push`

## 分支

### Detached Head 问题

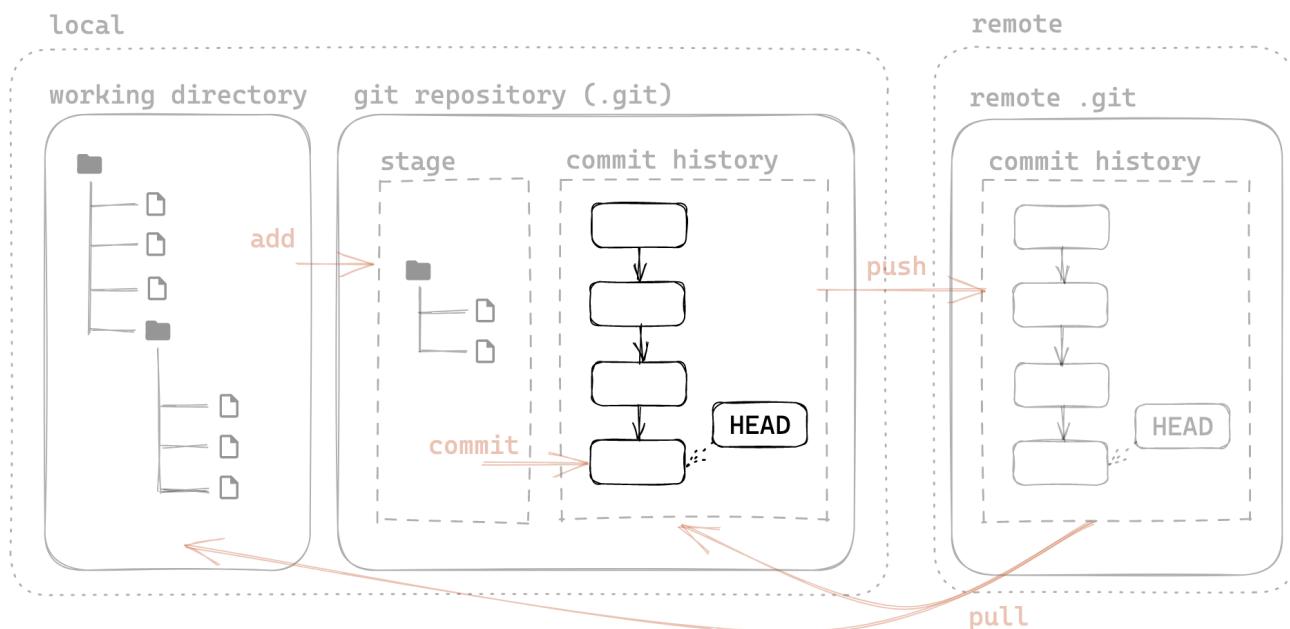
- 什么是 HEAD: 当前工作区在提交历史中的指针
- 什么是 detached HEAD: HEAD 指向某个历史提交, 而不是某个“分支”
- 什么情形会出现 detached HEAD
  - `git checkout id`, 此后的修改不会出现在任何分支

- 切换回 master 后会出现一条不属于任何分支的提交（相当于修改会丢失）



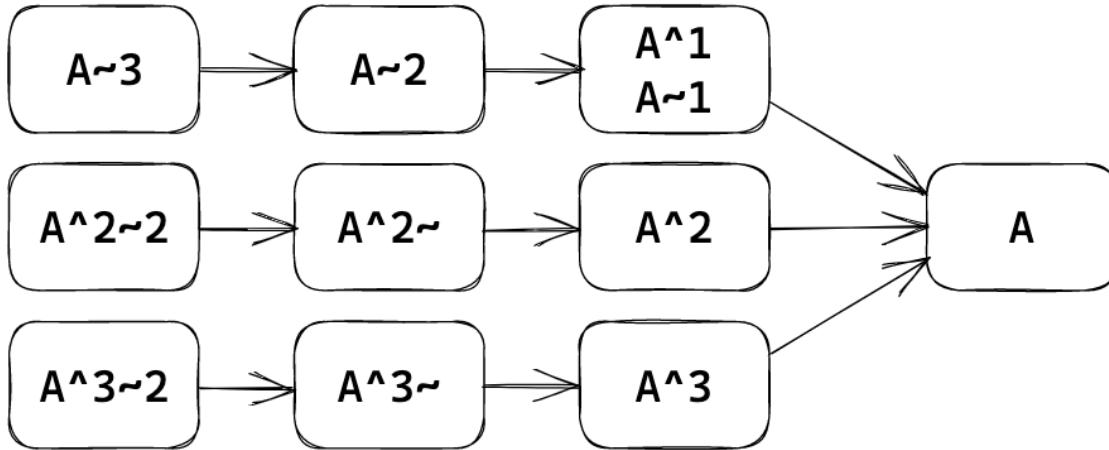
- 如何解决：在 F 的位置上 git checkout -b branch 创建并检出新分支

## 分支操作



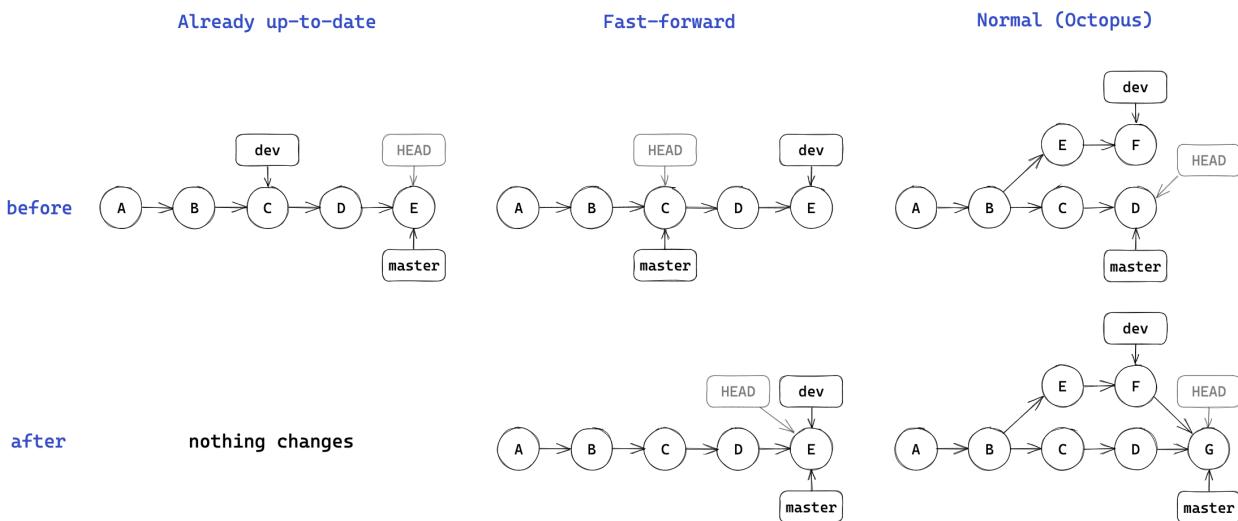
- 创建分支
  - `git branch name`: 基于当前 HEAD
  - `git branch name id`: 基于 id 提交
- 查看分支
  - `git branch` (带 -a 显示远程分支)
  - `git show-branch` 更详细
- 切换分支
  - `git checkout name`
  - `git checkout -b name`: 创建并切换
- 内容比较
  - `git diff branch1 branch2`: 比较两个分支
  - `git diff branch`: 比较工作区和分支
  - `git diff`: 比较工作区和暂存区

- 如何更方便地定位提交
  - 什么是分支名：和 HEAD 一样，也是一个指针（实际上叫引用 ref）
  - 可以基于 ref 使用 ~ 或 ^ 定位父提交
    - ~ 表示第一个父提交，~2 表示第一个父提交的第一个父提交
    - ^ 表示第一个父提交，^2 表示第二个父提交
  - 一个提交可能会有多个父提交（merge commit）



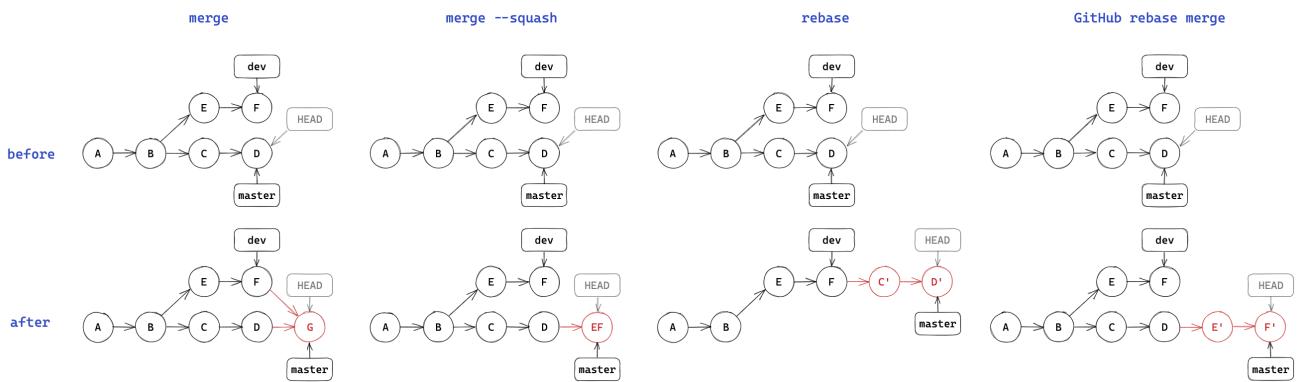
## 合并

- 将多个分支的更改都合并到当前分支：git merge branch1 branch2...
- 几种 merge 的情况
  - 当前分支只比被合并分支多提交：already up-to-date
  - 被合并分支只比当前分支多提交：fast-forward（将 HEAD 指向被合并分支）
  - 都有新的提交：产生一个 merge commit
    - 有冲突需要手动解决冲突（add 后再次 commit 生成 merge commit）



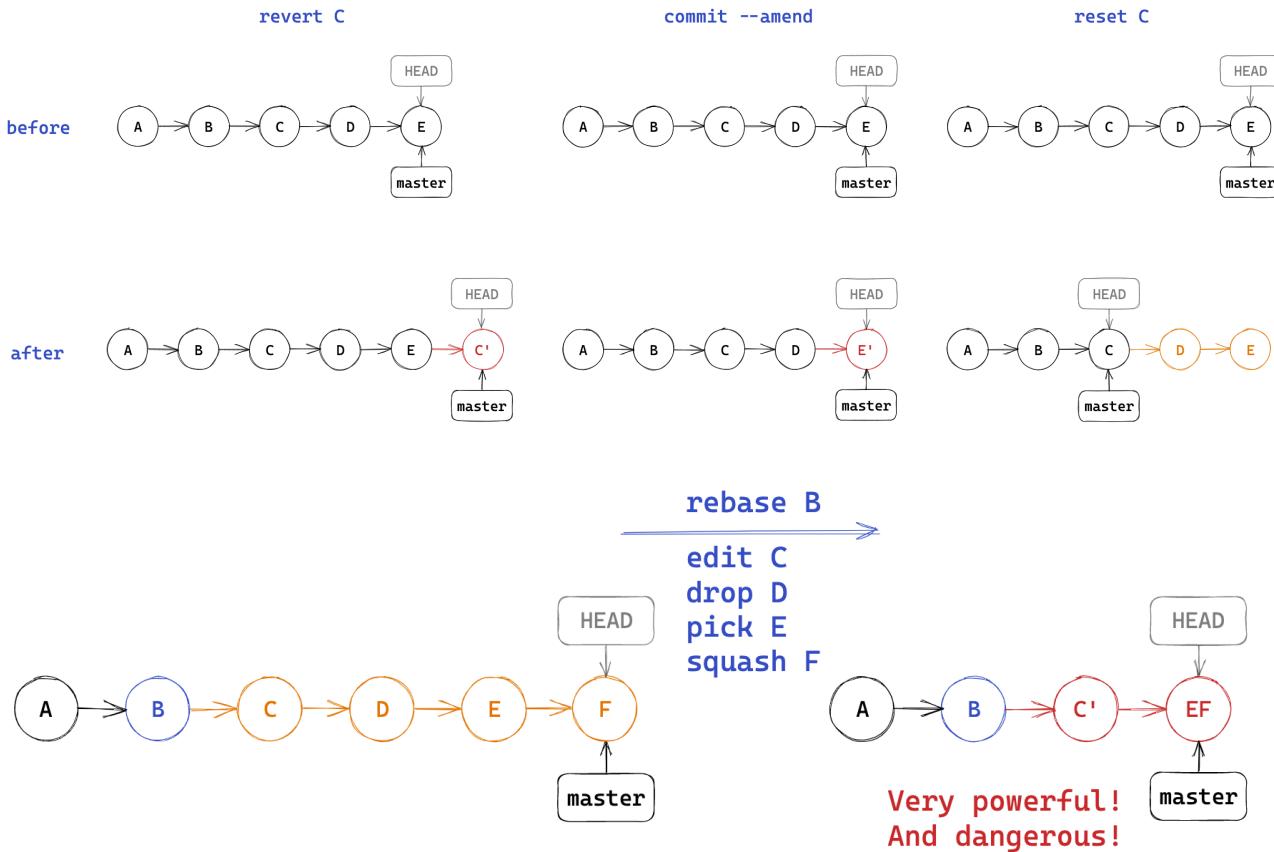
- 实际上 merge 操作一般都在 GitHub 上通过 PR 完成，两种特殊的 merge 方法：
  - squash merge：将目的分支多出的所有提交压缩为一个新提交并入当前分支
  - rebase：变基

- 命令行直接 rebase 会将当前分支接到目标分支后
  - 这种情况会导致提交历史更改，同步会有冲突，合作时不推荐
- 通过 GitHub PR rebase merge 会将目标分支接到当前分支后



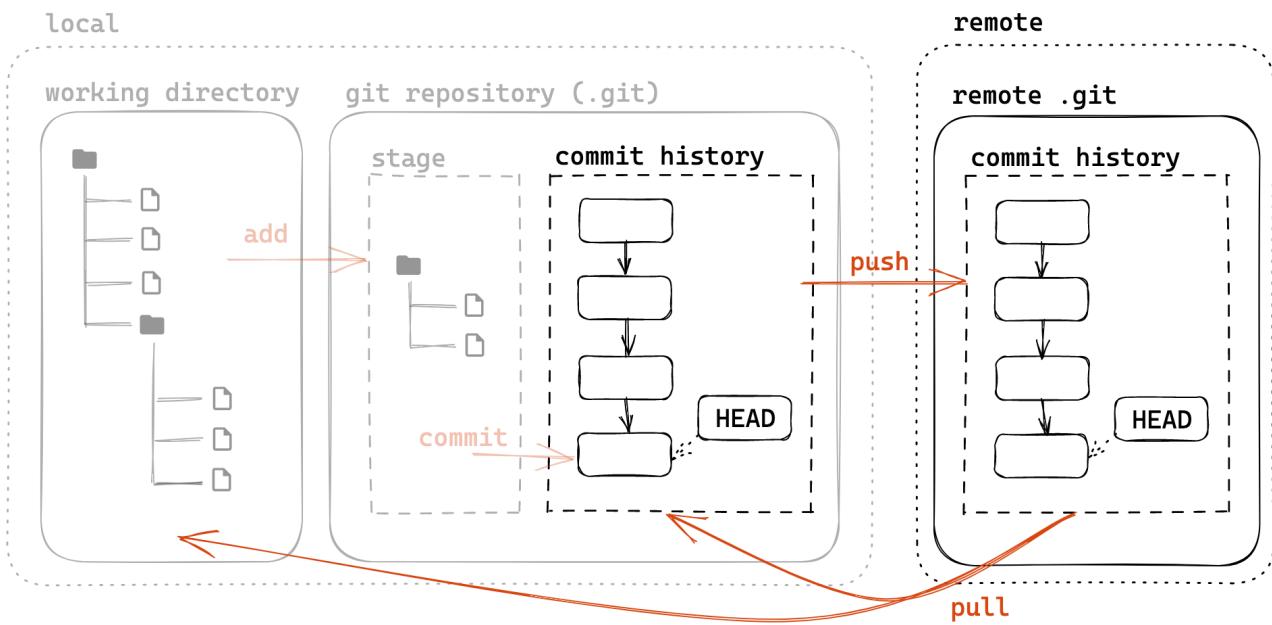
## 修改历史

- git revert *id* (用新提交抹掉以前提交的效果)
- git commit --amend (修改最新提交的 message)
- git reset *id* (回到某一提交的状态)
- git rebase -i *id* (交互式 rebase 变基，修改提交历史)



## Github 基础用法

### 远程版本库

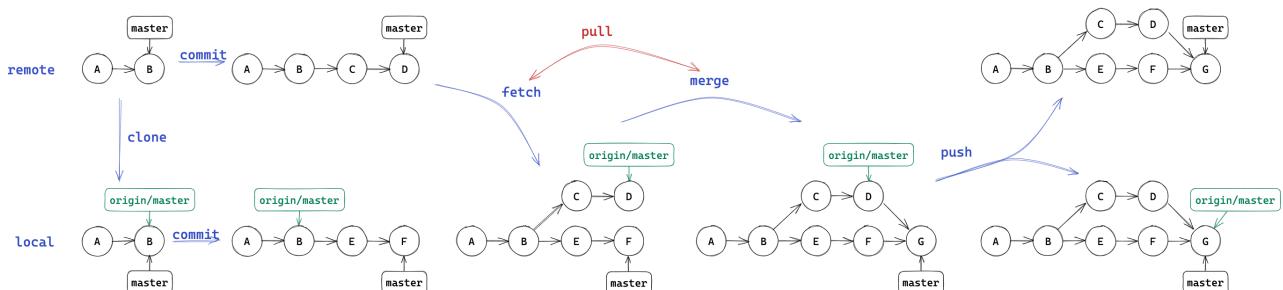


## 作用

- Git 这样的分布式 VCS 如何实现协作?
  - 使用一个远程的“权威”版本库 (remote repository)

## 理解

- 远程版本库也是一个普通的 git 版本库
  - 通过 `git clone src dest` 可以将远程版本库克隆到本地
    - 会自动建立 `remote` 关联，可通过 `git remote` 管理
  - `git push` 会将本地的提交推送到远程版本库
    - 无法直接 `push` 到远程版本库检出的分支中
    - 因此远程一般使用裸版本库 (`--bare`)
  - `git pull` 会将远程版本库的提交拉取到本地
    - 包含 `git fetch` 和 `git merge` 两个步骤
- 理解远程版本库：可以当作本地的一个 `origin/master` 分支
  - 后面会提到，实际上是在另一个命名空间 `remotes` 中
- 多的功能只有 `fetch` 更新这个分支，以及 `push` 推送到远程



## 如何访问

- 如何让合作的人都能访问到远程版本库?
  - 放在服务器上通过 SSH/HTTPS/Git 原生协议等访问

- 更方便的，放在 GitHub/GitLab 等托管网站上

## GitHub 基本操作

- 推荐一个浏览器插件：[Refined GitHub](#)

## 创建

- 创建账号、基本的设置不再赘述
- 一个非常重要的设置：
  - Settings > Access > Emails，一定要设置为 git 配置的邮箱
  - 为什么？想一想 GitHub 作为一个远程版本库的托管平台，它如何将版本库中每个提交的提交者关联到 GitHub 用户？

[Add email address](#)

Email address

Add

### Primary email address

tonycrane@foxmail.com will be used for account-related notifications and can be used for password resets.

tonycrane@foxmail.com 

Save

### Backup email address

Your backup GitHub email address will be used as an additional destination for security-relevant account notifications and can also be used for password resets.

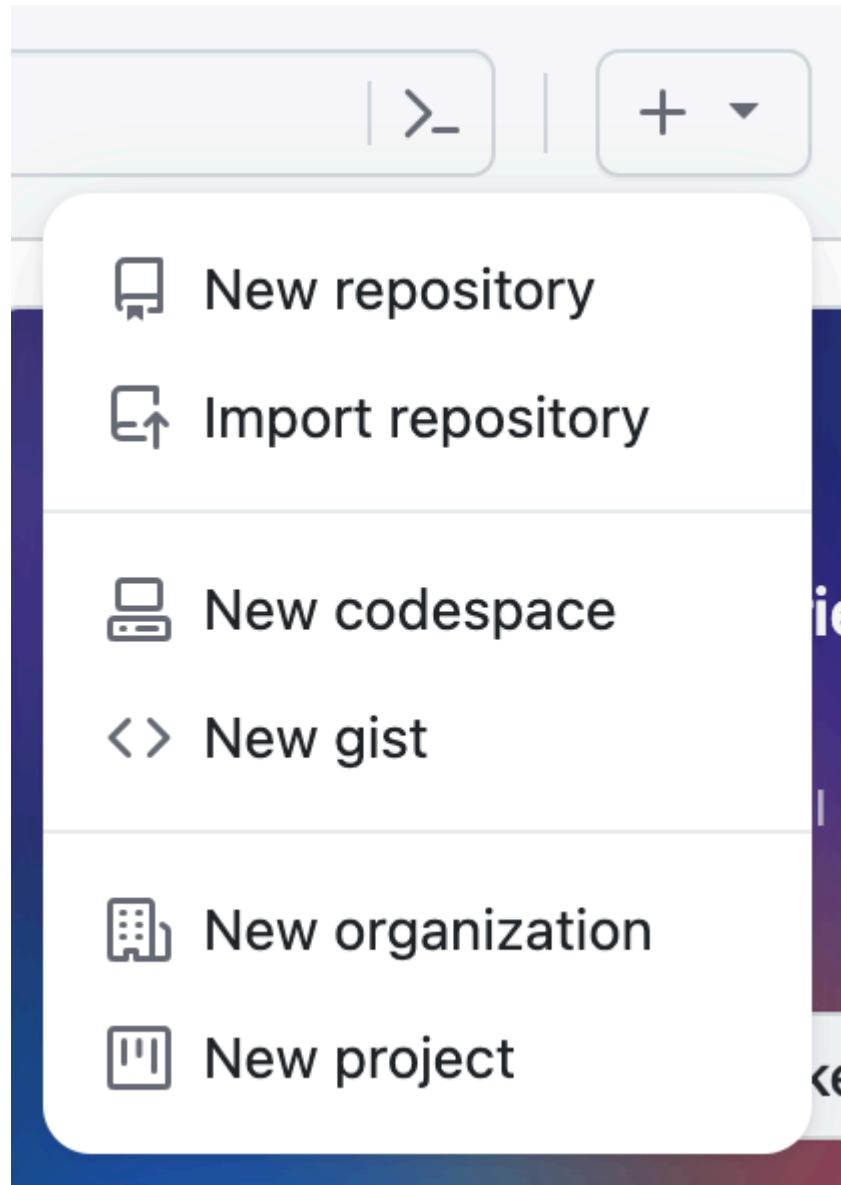
Allow all verified emails 

Save

**Keep my email addresses private**

We'll remove your public profile email and use [44120331+TonyCrane@users.noreply.github.com](mailto:44120331+TonyCrane@users.noreply.github.com) when performing web-based Git operations (e.g. edits and merges) and sending email on your behalf. If you want command line Git operations to use your private email you must [set your email in Git](#).

Commits pushed to GitHub using this email will still be associated with your account.



- 创建 (如图)
  - new repository 新存储库 (import 从链接导入)
  - new codespace 新代码空间 (新功能)
  - new gist 代码片段 (类似剪贴板)
  - organization 组织、project 项目计划面板

- 关于 repo

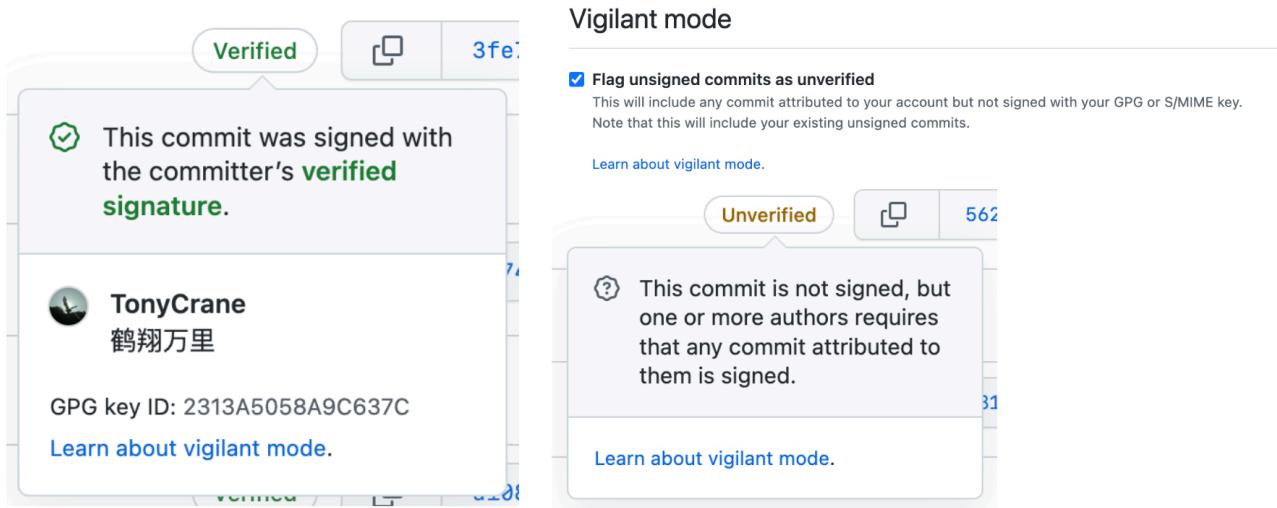
• README.md  
自述文件, README 会自动展示到主页, 一般使用 md (markdown) 格式

## GitHub 基本用法实操

- 新建 repo，基本设置
- 添加代码：
  - 从头开始的空项目：直接 clone
  - 从本地非 git 项目上传：init 后修改 remote
  - 修改、add、commit、push
- 分支、合并：branch / GitHub 上操作
- release：扩展的打 tag
- 小组项目合作：协作者、私有 repo 权限管理
  - pull request、merge、conflict 处理

## 签署 commit

- 为什么建议签署 commit
  - ["delete linux because it sucks"](#)
  - 回忆一下 GitHub 是如何关联 committer 和 GitHub 账号的
  - 只要有了你提交使用的 email，别人就可以伪造你进行 commit
- 如何通过 GPG 签署 commit：[GitHub 文档](#)
  - [GitHub 中提交 commit 时使用 GPG 进行签名](#)
- 带有签名的 commit 在验证后会显示为 Verified，如果开启了 vigilant mode，则没有通过验证的 commit 会标记为 Unverified



## GitHub Pages

- GitHub 会为每个用户 / 组织分配一个二级域名 `username.github.io`
- 可以创建一个名为 `username.github.io` 的 repo，会作为主页，通过 `username.github.io` 即可访问 repo 内存放的静态网页
- 对于其他 repo，也可以开启 Pages 功能，通过 `username.github.io/repo_name` 访问，静态页面来源也需要指定

## Build and deployment

**Source**

Deploy from a branch ▾

设置静态站点来源  
(大部分通过分支来部署, 新功能可以通过 Action 直接部署)

**Branch**

Your GitHub Pages site is currently being built from the gh-pages branch. [Learn more.](#)

gh-pages ▾ / (root) ▾ Save

Learn how to [add a Jekyll theme](#) to your site. 来源的分支中的目录

Your site was last deployed to the [github-pages](#) environment by the [pages build and deployment](#) workflow.  
[Learn more about deploying to GitHub Pages using custom workflows](#)

---

**Custom domain**

Custom domain

Custom domains allow you to serve your site from a domain other than blog.tonycrane.cc. [Learn more.](#)

note.tonycrane.cc Save Remove

✓ DNS check successful

这里有指南  
需要进行一些域名解析的设置

Enforce HTTPS 强制使用 HTTPS, 建议开启

HTTPS provides a layer of encryption that prevents others from snooping on or tampering with traffic to your site.  
When HTTPS is enforced, your site will only be served over HTTPS. [Learn more.](#)

# Git 基础配置

- 创建一个本地 git 版本库
  - 通过 git init 指令
    - git init: 让当前文件夹变成 git 仓库 (创建 .git 文件夹)
    - git init *folder*: 创建一个新的文件夹并初始化为 git 仓库
- git 账号配置
  - Why? 多人合作区分用户 / 让 GitHub 能够识别出你
  - 全局配置:
    - git config --global user.name "name"
    - git config --global user.email "email"
  - 针对某一版本库专门设置:
    - 同前, 不加 --global

# GitHub Actions

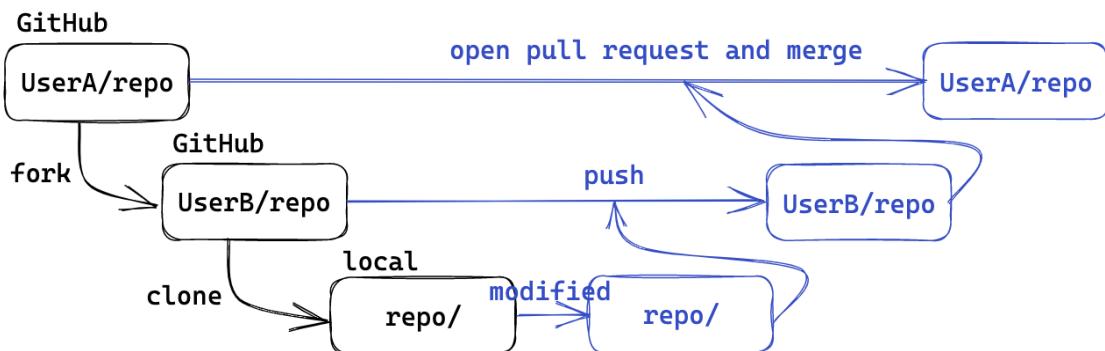
- GitHub 提供的 CI/CD 服务
  - CI (Continuous Integration) : 持续集成
  - CD (Continuous Delivery) : 持续交付
- 即配置一些自动化任务, 在特定事件发生时自动执行
  - 比如说每次 push 后自动测试, release 时自动构建部署
- 如何说明怎么执行任务

- 通过配置文件，文件名为 .github/workflows/workflow\_name.yml
- 如何写这个配置文件：
  - 在 GitHub 上编写会有提示
  - GitHub 文档：<https://docs.github.com/en/actions>
  - [GitHub Action 精华指南](#)、[GitHub Actions 入门教程 - 阮一峰](#)
  - 建议自己建一个 repo，编写一些 workflow，在尝试中学习

## GitHub 项目贡献 issue&discussion

- 先看 README，有没有 CONTRIBUTING、CODE\_OF\_CONDUCT 等文件
- issue
  - 几种内容：反馈 bug、提出新功能、寻求帮助等
  - 可以、而且建议使用 markdown 语法（特别是涉及到代码块的时候）
  - 一些原则：
    - 项目有明确规范 / 模板的时候请按照要求来写
    - 在提 issue 前先搜索有没有已有的类似 issue
    - 反馈 bug 时提供足够的信息，包括代码、错误、环境等
  - 如果自己开的 issue 已经解决或者不存在，请自行关闭
- discussion
  - 类似帖子 / 社区，要比 issue 更随意很多，话题范围也更广
  - 但也同样，请遵守规范，提问请提供足够信息

## Pull Request

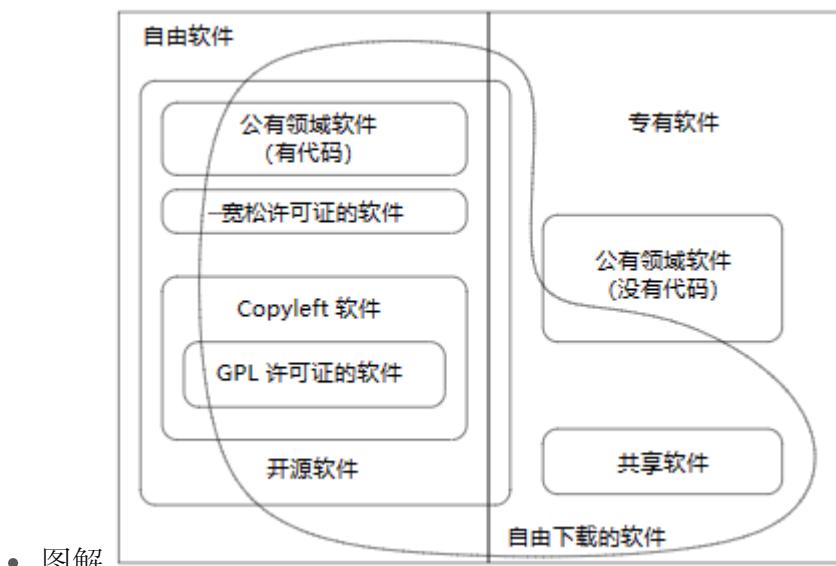


- pull request (PR)
  - 对于他人的 repo，你是没有办法直接 push 的，向其中添加代码更改都是通过 pull request 进行的
  - 在提 pull request 的前一定要阅读贡献守则
  - 提 PR 要按照要求写好标题和描述，修改的内容不要附带无用内容
    - 如果解决了某 issue 的 bug 的话，描述中最好加上 fix/close `#_issue_number_` 这样的内容，会自动链接并在 PR merge 之后自动关闭 issue
  - 一个 PR 中不要包含多个不相关的修改，如果有多个修改，应该分别提 PR
  - 有些项目会自动进行 CI，如果 CI 未通过，请检查错误信息并修改

- 对于自己有权限修改的项目，也建议使用 PR 进行修改，这样更清晰
  - 这时不必通过 fork 的方式，直接新建分支并修改即可
- review
  - 即有权限的人对 PR 进行审查，提出意见
  - review 可以针对某一行 / 几行代码进行
  - 收到 change request 后，请按要求进行修改
  - 一般的项目在 review 通过后才会 merge
- merge
  - 几种方式：merge commit、squash merge、rebase merge
  - 要求线性 log 的项目要使用 squash（一个 PR 就是一个 commit）
  - 可能会出现冲突，需要手动解决（通过 GitHub 或者根据指导在本地命令行进行）
- 向已有 pull request 添加修改的几种常见情况
  - 向自己开启的 PR 中继续添加修改
    - 直接在源分支中继续修改即可同步到 PR 中
    - 所以在开 PR 之后、merge 之前请不要随意删除源分支，也不要继续向其中添加无关修改
  - 向他人开启的 PR 中添加修改
    - 你有目标分支的写（write）权限
      - 可以直接在 GitHub 中进行编辑，这时修改也会同步到 PR 中
      - 也可以本地修改后 push 到源分支
        - 推荐使用 GitHub CLI: `gh pr checkout pr_number`
        - 直接 push 会有错误，但 git 会提示正确方式：`git push source_branch_url HEAD:master`
    - 没有写权限：建议只提出修改建议，或再向源分支发起 PR

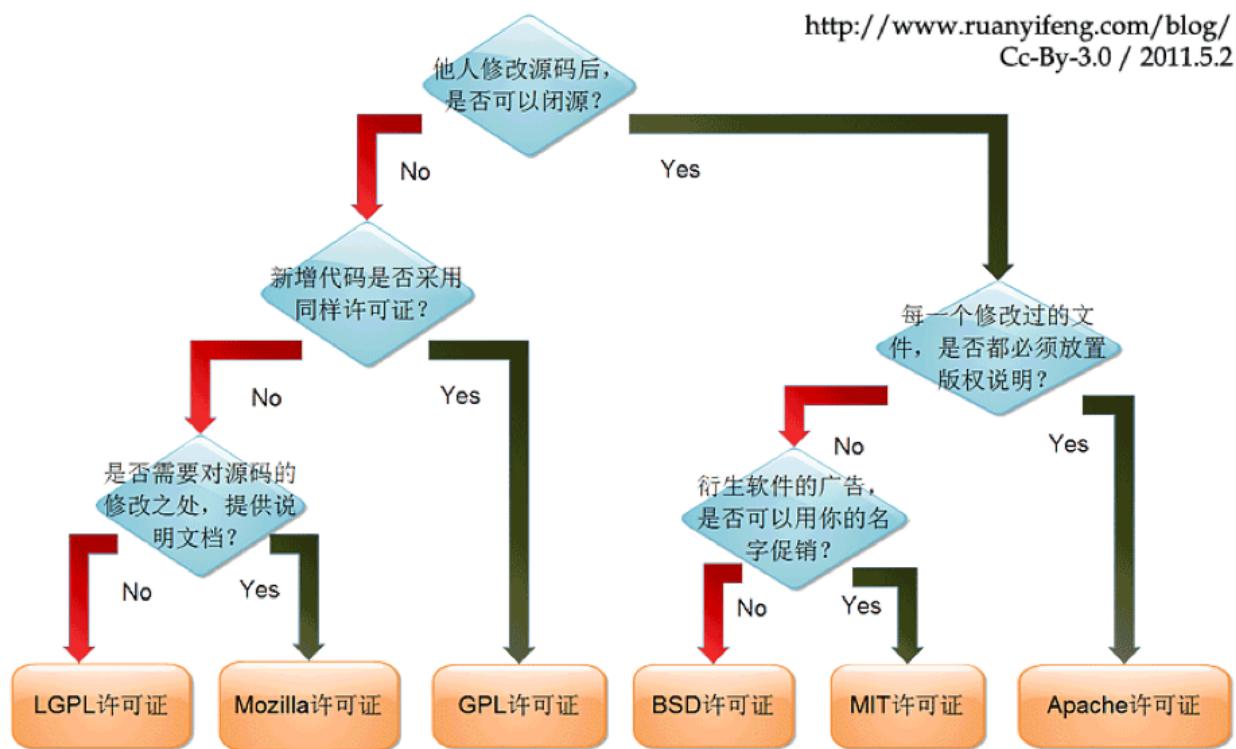
## 开源项目基础 开源/自由软件简介

- 开源（Open Source）：公开源代码
- 自由（Free）：遵循四项自由原则
  - 自由运行、自由修改、自由分发拷贝、自由分发修改
  - 参考: [FSF](#)、[什么是自由软件 - GNU](#)
  - 自由和开源是完全不同的概念
  - 自由软件也并不意味着不是商业软件
- 关于 Copyright 和 Copyleft：
  - Copyright：版权所有，一切权利归软件作者所有
    - Copyright © 最初发表年-最近更新年 所有者. All rights reserved.
  - Copyleft：版权归原作者所有，其他一切权利归任何人所有
    - Copyleft 的一定是自由软件，GPL 是一种 Copyleft 许可证



## 开源协议 / 许可证 (LICENSE)

- 没有许可证? 原作者保留所有权利, 不允许复制、分发、修改
  - 使用的话需要联系原作者, 见 [choosealicense.com/no-permission](http://choosealicense.com/no-permission)



- 常见软件开源许可证
  - GPL (GNU General Public License)
    - Copyleft、有“传染性”
    - GPLv3、AGPLv3、LGPLv3
  - Unlicense: 放弃权利, 进入公共领域
    - 详见 [choosealicense.com \(appendix\)](http://choosealicense.com/appendix)
- 在开源项目中使用许可证
  - 根目录下包含文件 LICENSE, 其中附上许可证内容
  - GitHub 可以从模板生成一些 LICENSE, 也会根据内容识别并显示许可证

- 采取多个许可证：都要放，并说明许可证作用范围

## 非软件类许可证

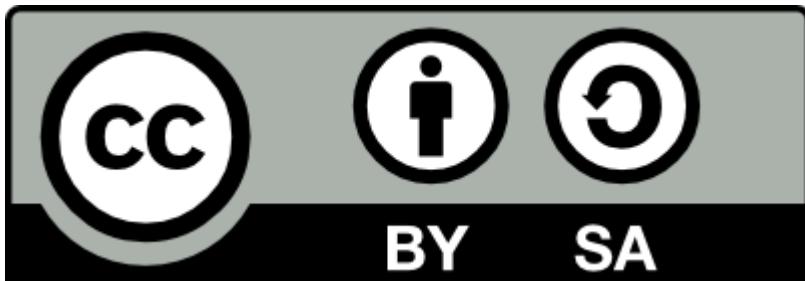
- 主要是 CC (Creative Commons) 系列许可证，用于“知识共享”，不用于软件
  - 官网：<https://creativecommons.org/share-your-work/cclicenses/>
  - CC 0: Public Domain, 进入公共领域



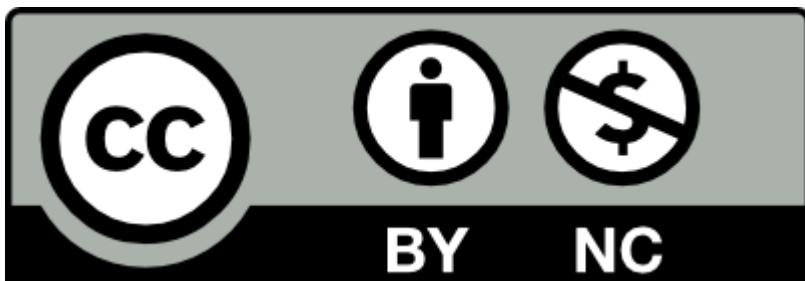
- CC BY: Attribution, 需要标明原作者



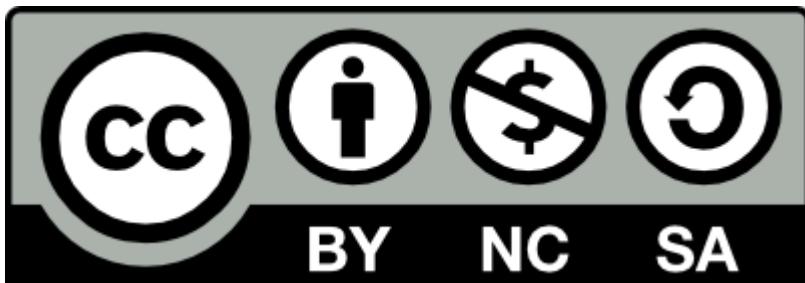
- CC BY-SA: \*ShareAlike, 需要采用相同许可证



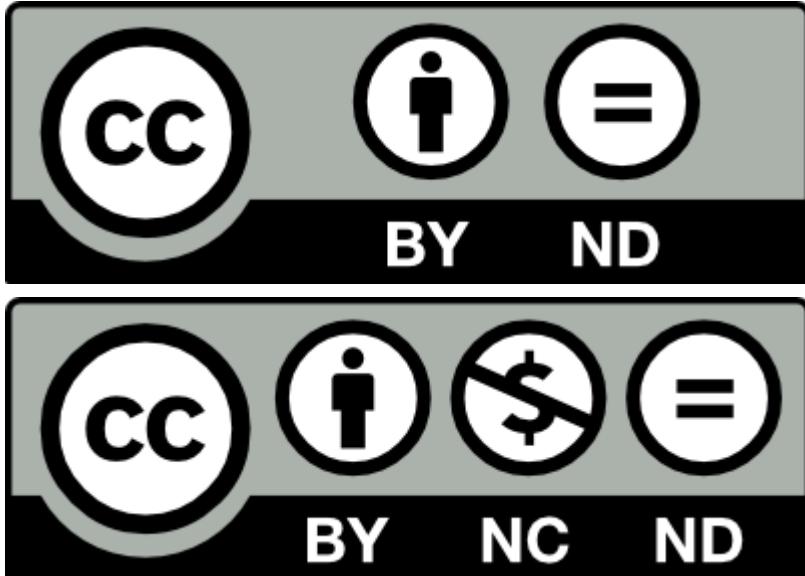
- CC BY-NC: \*NonCommercial, 禁止用于商业用途



- CC BY-NC-SA: 三个要求均有



- CC BY-ND / BY-NC-ND: \*NoDerivs, 禁止分发、修改



- 带有 NC/ND 的就不是自由协议，目前使用的都是 4.0 版本
- 使用
  - 同样把内容写在 LICENSE 里，官网找到对应许可证，进入 /legalcode.txt
    - GitHub 目前只会识别 CC 0 / CC BY / CC BY-SA
    - "... is licensed under a Creative Commons ... 4.0 License"