万字长文｜GitHub 从入门到精通

作者：Miles Ma（@miles_mazy）
原文：[https://x.com/miles_mazy/status/2088939439766819162]
(https://x.com/miles_mazy/status/2088939439766819162)
发布时间：Sun Aug 16 10:42:27 +0000 2026
本地保存日期：2026-08-17

如果你想用 GitHub 赚钱，最直接的路并不复杂：在许可证允许的前提下，找到有价值的开源项目，把部署、中文说明和售后做成服务，再去闲鱼等平台卖交付。

但是真正能收钱的是信息筛选和落地能力。想做 AI、转 FDE，或者把自己变成一家 OPC，代码、文档、版本和协作早晚都会落到 GitHub 上。

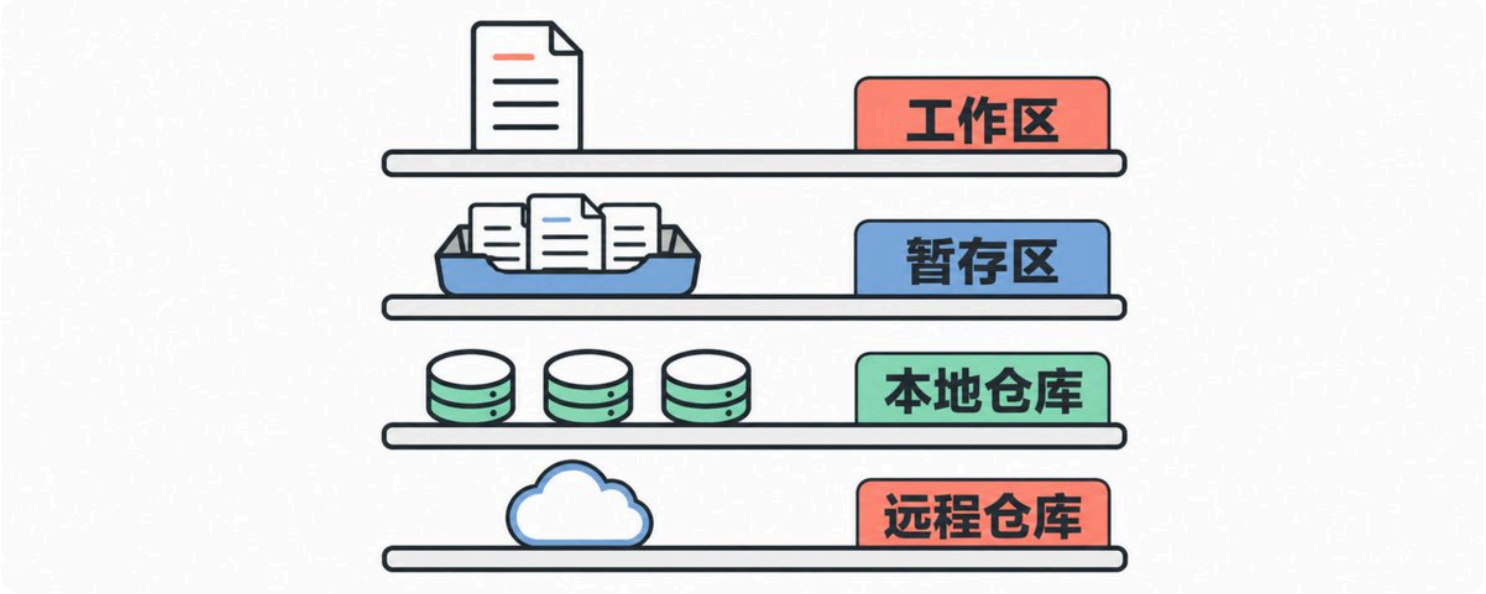
哪怕你做的是创作和自媒体，GitHub 上也有大量选题工具、自动化项目和内容生产流程，可文件一多、AI 一改，没有 Git 管版本，很快就会失控。所以，程序员要学，做项目、做内容的人也一样；它决定你能不能把灵感变成一个可管理、可复用、能交付的项目。

这篇我断断续续磨了大半个月，从 0 到 1 把 Git、GitHub、Commit、分支、PR 和常见报错都实操了一遍，之后的直播也会沿着同一套流程来讲。正式开播前，我先把这份教程开源出来，大家可以先收藏，也可以跟着完整做一遍。

一、Git 和 GitHub，到底谁管什么

Git 是装在电脑里的版本管理工具。断网时，你仍然可以提交、看历史、建分支和合并。GitHub 是远程仓库与协作平台，它接收 Git 推上来的提交，再提供 Issues、Pull Requests、Actions、代码审查和权限管理。

Git 最容易混的，是同一份修改会处在四个不同位置。下面这张信息图把工作区、暂存区、本地仓库和远程仓库拆成了四层。



按下保存，只会把内容写进硬盘。git add 负责挑选，git commit 在本地留下版本，git push 才把这些提交送到 GitHub。

所以提交前要看 diff、实际运行或测试；推送成功后，再去网页回查。这样出了问题，你能马上知道它停在哪一层。

二、动手前，只准备四样东西

你需要 Git、一个 GitHub 账号、一个编辑器和一个练习项目。编辑器用 VS Code 就够了，项目可以是一张网页，也可以是一份 Markdown 文档。

先在终端确认 Git：

```
git --version
```

这次演练使用的是 macOS 和 Git 2.49.0。Windows 用户可以用 Git Bash 或 VS Code 内置终端，下面的 Git 命令相同。

接着配置提交作者：

```
git config --global user.name "你的名字"
git config --global user.email "你的邮箱"
```

这是写进提交记录的作者信息，不负责登录 GitHub。如果你只想给当前练习项目配置，把 --global 换成 --local。

GitHub 的登录是另一件事。命令行常用三种方式：

- GitHub CLI，通过 gh auth login 打开浏览器授权；
- HTTPS，使用 Personal Access Token 或凭据管理器；
- SSH，把公钥添加到 GitHub，后续通过密钥认证。

刚入门可以选 GitHub CLI 或 HTTPS。使用 HTTPS 时，终端要求 Password 就填写 Token，普通账号密码已经不适用。Token 不要写进命令、远程 URL、README、聊天和截图。

三、别急着 init，先确认终端到底在哪

这次练习从一个普通网页开始。它能在浏览器打开，但还没有任何 Git 历史。

项目里有三个文件：

```
index.html
style.css
.gitignore
```

在 VS Code 选择「打开文件夹」，不要只点开一个 HTML 文件。然后在内置终端运行：

```
pwd
ls
```

pwd 显示当前目录，ls 列出文件。看到 index.html 和 style.css 以后再继续。

这个检查看着很笨，却能挡住最麻烦的一类事故：有人在桌面、Documents，甚至用户主目录执行 git init，随后 git add . 把几千个无关文件放进暂存区。Git 没坏，目录选错了。

四、git init 做了什么

现在初始化仓库：

```
git init -b main
git status --short
```

git init -b main 在当前目录创建 .git，并把初始分支命名为 main。.git 是一个隐藏目录，提交、分支、暂存区、远程地址等信息都放在里面。项目文件还在原处，Git 从这一刻开始观察它们。

截图里的 ?? 表示未跟踪。文件已经存在，Git 还没决定要不要记录。

想确认仓库根目录，可以运行：

```
git rev-parse --show-toplevel
```

输出应该是当前项目文件夹。若提示 fatal: not a git repository，先检查目录，再看是否执行过 git init。

五、第一次提交，先留住一个可靠起点

项目还没修改，为什么要先提交？因为后面所有变化都需要一个可比较的起点。先在浏览器打开网页，确认标题、卡片和报名区能显示；把窗口缩窄，看看手机宽度有没有横向滚动。

再看 .gitignore。这次使用的内容是：

```
.env
.env.*
*.log
node_modules/
dist/
build/
```

.gitignore 用来挡住密钥、日志、依赖和构建产物。它主要对尚未跟踪的文件生效。一个密钥如果已经提交过，后来补上 .gitignore，那段历史还在，真正的处理还包括吊销或轮换密钥。

开始挑选第一次提交的文件：

```
git add index.html style.css .gitignore
git status --short
git diff --cached --stat
```

状态中的 A 是 Added，表示文件已经进入暂存区。git diff --cached --stat 会告诉你这次准备提交几个文件、大概改了多少行。想看具体内容，运行：

```
git diff --cached
```

确认以后提交：

```
git commit -m "chore: 初始化校园 AI 招新页"
git log --oneline
git status
```

Commit 可以理解成带作者、时间、说明和父提交的项目快照。ffdf4ff 是这次提交哈希的短写，在当前仓库中用它就能准确定位版本。

feat、fix、docs、style、chore 是常见的提交类型，并非 Git 强制语法。比前缀更重要的是后面的中文：做了什么，改了哪个对象，为什么做。

六、第二次提交：把 AI 的修改当成待审稿

接下来给页面加一个「查看报名方式」按钮。使用 AI 编程工具时，我会把边界写进提示词：

```
只修改 index.html，在介绍文字下方增加“查看报名方式”链接，
链接到页面内的 #apply。不要修改 style.css，不要执行 Git 提交。
完成后告诉我改了哪个文件。
```

手工修改也很简单：

```
<a class="cta" href="#apply">查看报名方式</a>
```

AI 说完成了，先别提交。运行：

```
git status --short
git diff -- index.html
git diff --check
```

本地实操 03

先看 diff，再决定这次改动值不值得提交

git diff 显示工作区里尚未暂存的变化。绿色 + 是新增行，红色 - 是删除行。git diff --check 没有输出，说明没有发现明显的尾随空格等格式问题；它不会替你检查按钮能不能点。

回到浏览器刷新，点击按钮，再把窗口缩窄。页面应该滚动到报名区，按钮和卡片在窄屏下仍然正常。

测试通过后才提交：

```
git add index.html
git diff --cached
git commit -m "feat: 新增报名入口，方便快速查看申请方式"
git log --oneline -2
```

这时仓库里有两个能说清楚的版本：起始页和报名按钮。以后按钮出问题，可以直接找到它是哪次加进来的。

顺便把两个 diff 分清：

```
git diff             # 工作区与暂存区之间的差异
git diff --cached    # 暂存区与最近一次提交之间的差异
```

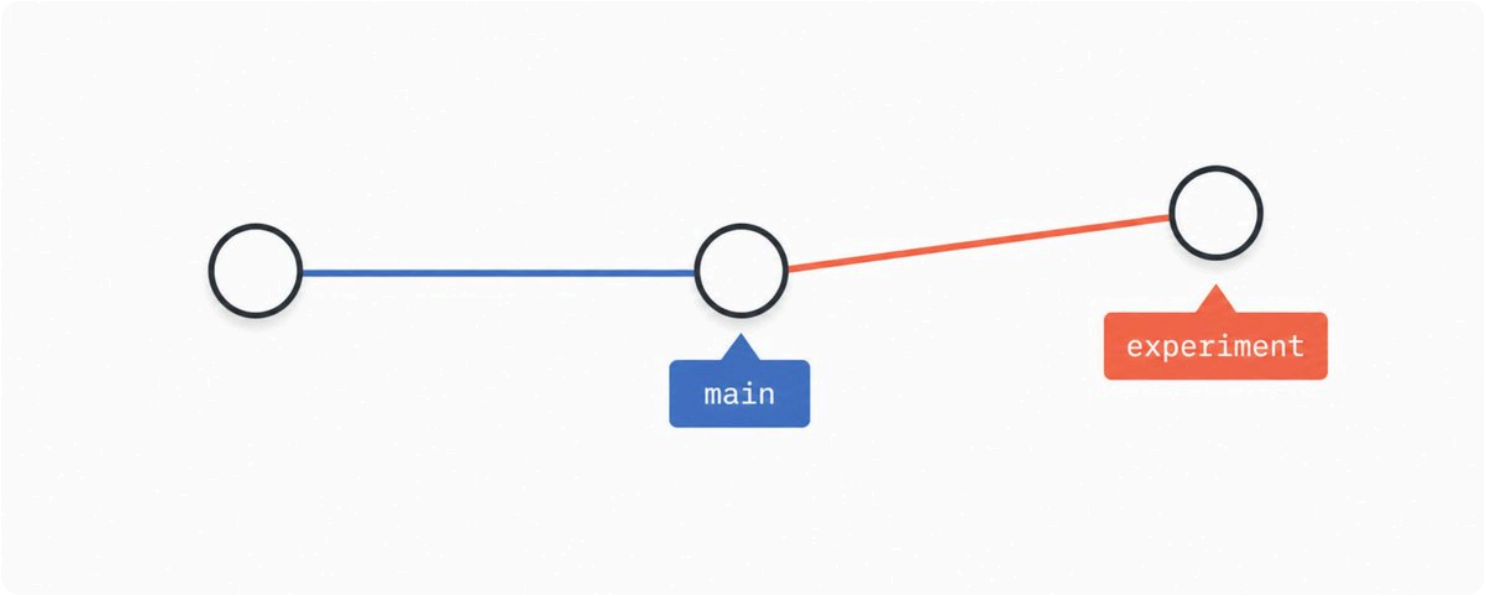
如果 git diff 没输出，文件可能没保存，也可能已经暂存或提交。依次看 git status、git diff --cached 和 git log，比反复输入 git add . 靠谱。

七、分支：给不确定的改动留一个试验位置

按钮只增加一行，风险不大。把整套主题从紫色改成橙色，结果可能好看，也可能很俗，这种修改适合放进分支。

```
git switch -c experiment/warm-theme
git branch --show-current
```

分支在概念上是指向某个提交的名字。新分支刚创建时与 main 指向同一提交，所以文件完全一样。等实验分支产生新提交，两条线才分开。



图里蓝色的 main 仍指着第二次提交，橙色的 experiment 已经指向第三次提交。项目并没有复制出两套文件，变化的只是两个分支名分别指向哪里。

修改 style.css 的颜色变量，刷新页面确认以后提交：

```
git diff -- style.css
git diff --check
git add style.css
git commit -m "style: 在实验分支试用暖色主题"
git log --oneline --graph --decorate --all
```

本地实操 04

分支是提交指针，创建它不会复制整份项目

HEAD 表示你当前站在哪里。截图里 HEAD 指向 experiment/warm-theme，main 仍停在按钮提交。

确定保留暖色主题，切回 main 合并：

```
git switch main
git merge experiment/warm-theme
```

这里出现 Fast-forward，因为 main 在实验期间没有新增提交。Git 直接把 main 指针向前移动到暖色主题的提交，改动已经合并成功。

已合并分支可以安全删除：

```
git branch -d experiment/warm-theme
```

小写 -d 会检查分支是否已经合并。大写 -D 会强制删除，分支中尚未合并的提交可能因此失去引用，别把它当日常清理命令。

八、冲突没有那么神秘，它只是 Git 不敢替你选

为了验证冲突，我又复制了一份仓库。main 把主标题改成「让校园里的创意，被更多人看见」，feature 分支把同一行改成「把一个想法，做成真正能用的作品」。合并时 Git 停了下来：

本地实操 06

两条分支改到同一行，Git 会停下来等人决定

冲突标记分成三段：

```
<<<<<<< HEAD
当前分支的内容
=======
要合进来的分支内容
>>>>>>> feature/rewrite-heading
```

处理方法是编辑文件，留下最终想要的文字，删除三组标记，实际测试，再运行：

```
git add index.html
git commit
```

如果当时不想处理，可以退出这次合并：

```
git merge --abort
```

冲突表示两个人或两个 Agent 对同一位置给出了不同答案，Git 无法擅自替人选择。

九、把本地仓库送到 GitHub

项目已经有本地历史，现在去 GitHub 创建仓库。点击右上角 +，选择 New repository，填写仓库名，例如：

```
campus-ai-demo
```

第一次练习建议设为 Private。因为本地已经有 README、.gitignore 和提交历史，GitHub 新仓库保持空白，不要在网页端初始化 README、许可证或 .gitignore。否则本地和远程会各有一段初始历史，第一次推送就需要先处理两边的关系。GitHub 官方的「Adding locally hosted code」也明确提醒了这一点。

复制 HTTPS 地址：

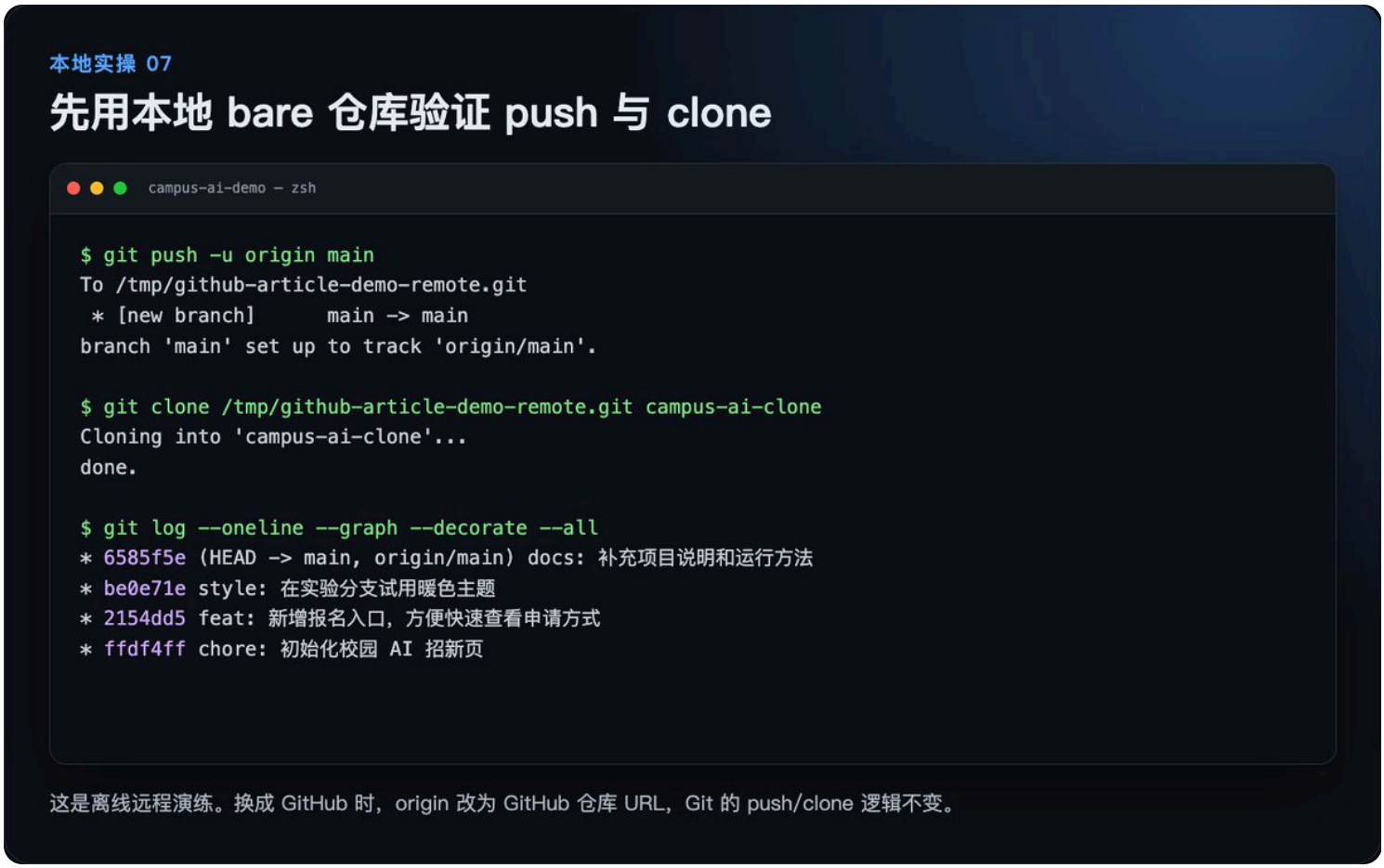
```
https://github.com/你的用户名/campus-ai-demo.git
```

回到项目终端：

```
git remote add origin https://github.com/你的用户名/campus-ai-demo.git
git remote -v
git push -u origin main
```

origin 是远程地址的别名，换成别的名字也能工作，只是社区习惯把主要远程叫 origin。-u 会把本地 main 与 origin/main 建立跟踪关系，后续通常直接运行 git push。

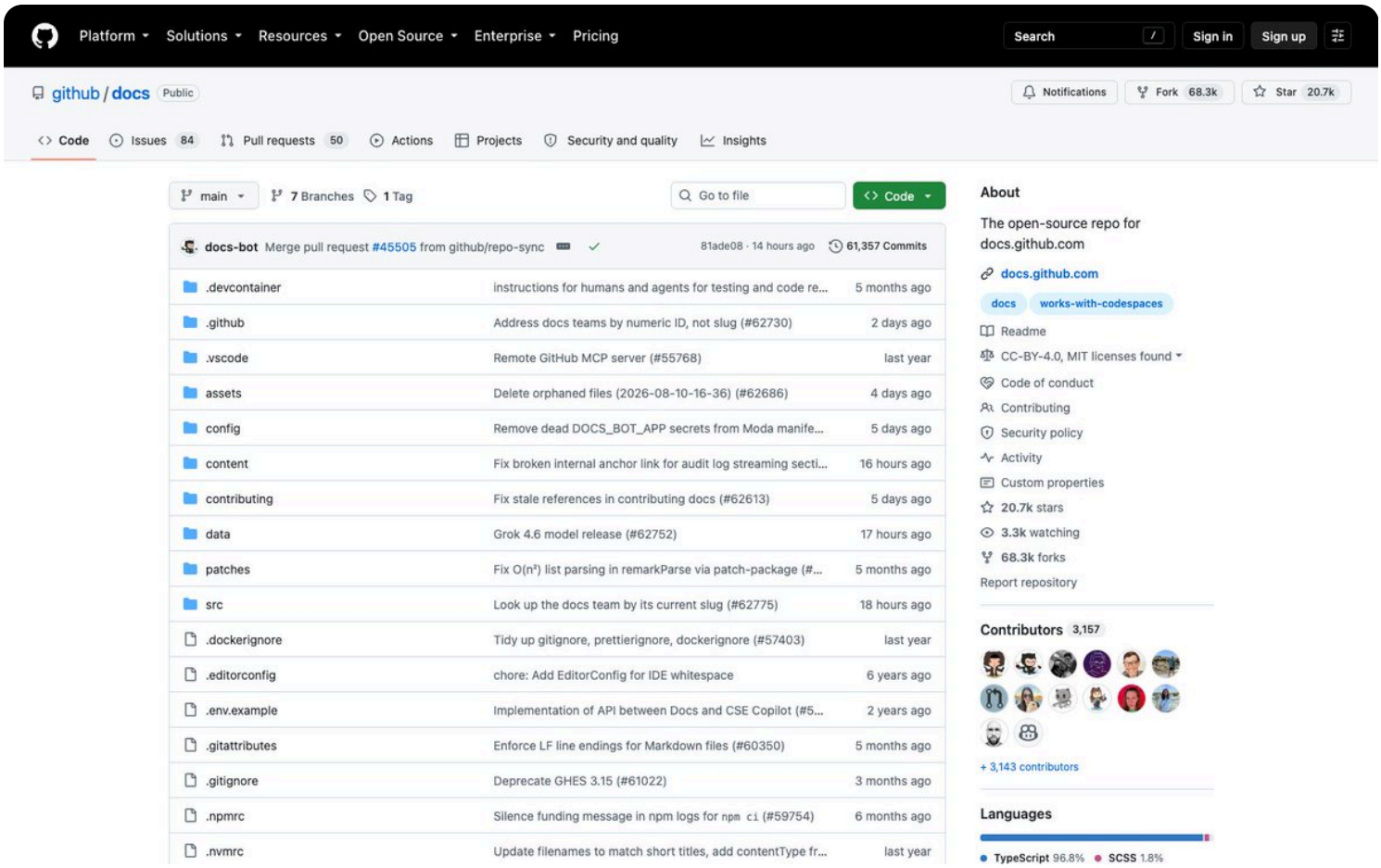
下面这张终端图使用本地 bare 仓库跑通了 push 与 clone，因此没有改动现有 GitHub 账号。换成 GitHub 时只需替换 origin URL，Git 传递提交和建立跟踪关系的逻辑相同。



真实推送完成后，回到 GitHub 网页刷新，确认文件、README、默认分支和提交历史都能看到。终端的成功信息是一层证据，网页回查是另一层。

十、第一次看 GitHub 仓库，页面上这些东西怎么读

下面是 GitHub 官方文档仓库的真实页面，截图时间为 2026 年 8 月 15 日。



打开一个仓库，先看这些位置：

- Code：文件、目录、分支和提交；
- Issues：Bug、需求、任务和讨论；
- Pull requests：等待审查或合并的改动；
- Actions：自动测试、构建和部署；
- Security：安全策略与漏洞相关功能；
- Insights：贡献、流量和仓库活动；
- README：项目介绍与使用入口；
- LICENSE：允许怎样使用、修改和分发。

读陌生项目时别先盯 Star。先回答五个问题：它解决什么问题，怎么运行，依赖什么，最近是否维护，许可证允许我做什么。Star 反映关注度，不替你检查安全、兼容性和授权。

十一、clone、fetch、pull、push，四个方向别搞混

第一次把远程仓库取到电脑：

```
git clone https://github.com/OWNER/REPO.git
```

Clone 会带回文件、提交历史和远程配置，通常自动把远程命名为 origin。下载 ZIP 只有当时的文件快照，没有完整历史，也不会建立远程关系。

之后常用的三个动作是：

```
git fetch origin   # 下载远程信息，不改当前工作文件
git pull           # fetch 后再整合到当前分支
git push           # 把本地提交发送到远程
```

想先看远程发生了什么，可以：

```
git fetch origin
git status -sb
git log --oneline HEAD..origin/main
```

确认本地没有分叉，希望只接受快进更新时：

```
git pull --ff-only
```

pull 会先 fetch，再根据配置做 merge 或 rebase。团队在第一次合作前最好约定整合方式，遇到分叉时也别靠 force push 抹平问题。

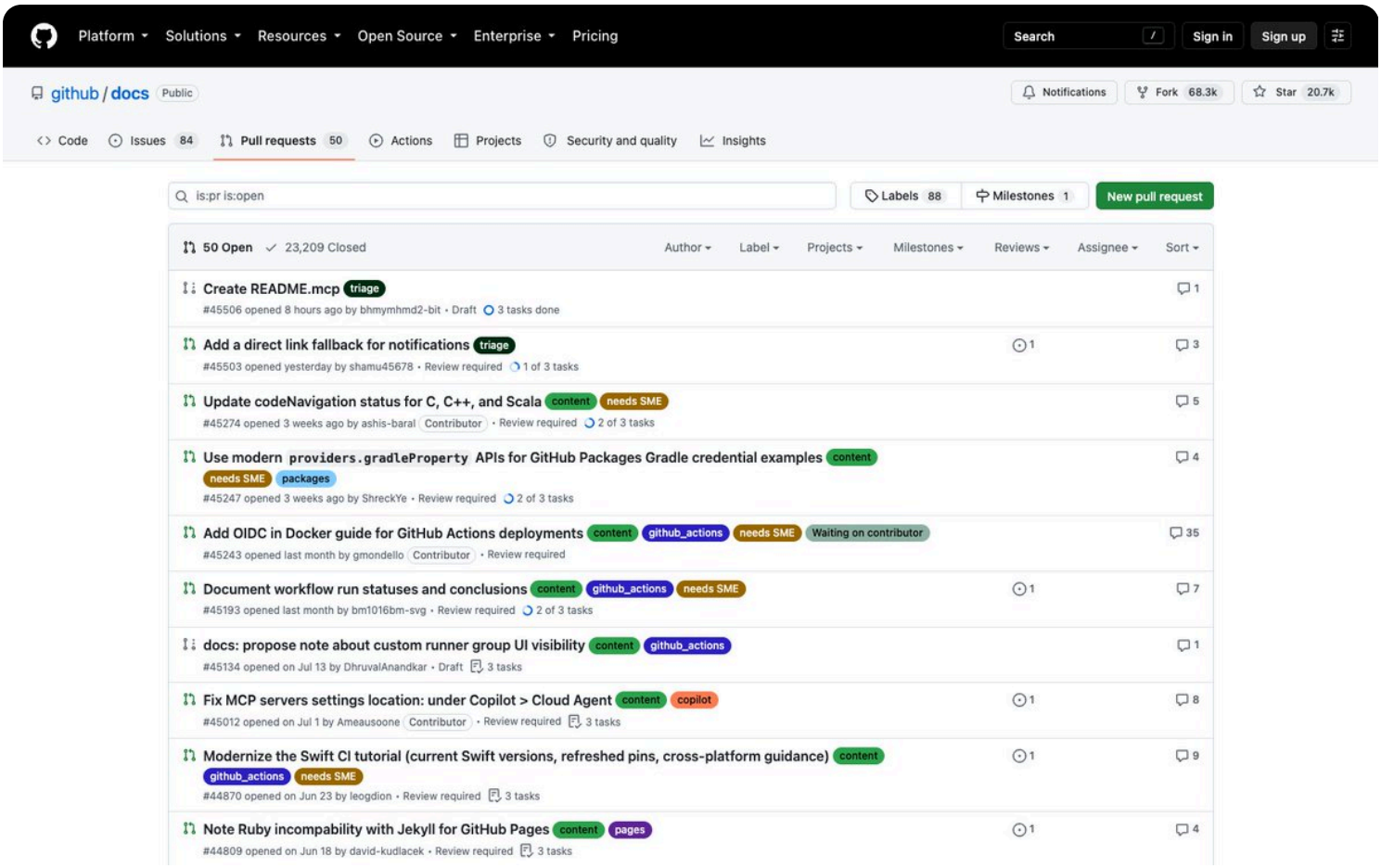
十二、从个人仓库走到 GitHub 协作

Pull Request 是一次合并提案，也是协作发生的地方。讨论、代码审查和自动检查都围绕同一份改动展开，确认以后才合并进 main。

假设 Issue 是「增加活动时间说明」，本地操作可以这样做：

```
git switch -c feat/event-time
# 修改并测试页面
git add index.html
git commit -m "feat: 增加活动时间说明"
git push -u origin feat/event-time
```

推送后，GitHub 通常会提示创建 Pull Request。PR 是一次合并提案，里面会展示描述、提交、文件差异、评论、审查和自动检查。它不会因为被创建就自动进入 main。



一个让人愿意审的 PR，至少说明三件事：改了什么，为什么改，怎么验证。改动越聚焦，审查者越容易看出问题。

同一团队中，你有仓库写权限，可以直接从分支发 PR。给陌生开源项目贡献时，常见做法是先 Fork 到自己的账号，再 clone 自己的 Fork：

```
git clone https://github.com/你的用户名/项目名.git
cd 项目名
git remote add upstream https://github.com/原作者/项目名.git
git remote -v
```

这里通常有两个远程：

```
origin    你自己的 Fork
upstream  原作者的仓库
```

同步原项目：

```
git fetch upstream
git switch main
git merge --ff-only upstream/main
git push origin main
```

接着在新分支完成修改，推到自己的 Fork，再向 upstream 发 PR。Fork、clone 和 branch 解决的是三件不同的事：Fork 是 GitHub 上的一套仓库空间，clone 把仓库带到本地，branch 是某个仓库内部的开发线。

十三、README 和 LICENSE，决定别人敢不敢用

README 至少回答这些问题：

1. 项目是什么；
2. 解决什么问题；
3. 怎样安装或运行；
4. 现在完成到什么程度；
5. 主要文件放在哪里；
6. 作者、素材和引用来源是谁。

代码能运行，README 写得含糊，三个月后的自己也可能接不起来。最小 README 不必漂亮，先把项目、运行方法和状态写清楚。

公开仓库也不自动等于获得开源许可。GitHub 官方许可证说明写得很明确：没有许可证时，默认版权规则仍然适用，作者保留对复制、分发和衍生作品的权利。公开意味着别人能看，也能按 GitHub 服务条款进行 Fork；把代码拿进自己的公开或商业项目，还要看仓库里的 LICENSE。

MIT、Apache-2.0、GPL 等许可证的义务不同。遇到商业使用、再分发或混合许可证，读完整文件，必要时请专业人士判断，别只问 AI 一句「能不能商用」。

十四、改错了以后，先判断改动在哪一层

后悔药要按状态选。

暂存了错误文件，但想保留文件内容：

```
git restore --staged 文件名
```

最近一次提交信息写错，而且还没推送：

```
git commit --amend -m "新的提交信息"
```

共享分支上某次提交需要撤销：

```
git revert 提交哈希
```

Revert 会产生一个新的反向提交，旧历史仍然可见，比较适合已经推送和多人使用的分支。

git restore 文件名 会丢弃尚未提交的修改；git reset --hard 会让提交、暂存区和工作区一起回到指定位置；git push --force 可能覆盖远端提交。这三类操作执行前要确认目标和备份，零基础阶段不要把它们当通用修复按钮。

十五、八个最常见的报错，按这个顺序查

1. fatal: not a git repository

```
pwd
ls
git status
```

通常是目录不对，或当前项目还没 git init。

2. Author identity unknown

```
git config --local user.name "你的名字"
git config --local user.email "你的邮箱"
```

3. nothing to commit

检查文件是否保存，是否改了另一个副本，以及改动是否已经提交：

```
git status
git log --oneline -3
```

4. remote origin already exists

```
git remote -v
git remote set-url origin 正确的GitHub地址
```

5. src refspec main does not match any

仓库可能还没有提交，或者当前分支不叫 main：

```
git log --oneline
git branch --show-current
```

6. Authentication failed 或 403

检查远程 URL、仓库归属、账号权限和认证方式。不要把 Token 发给别人排查。

7. rejected non-fast-forward

远程存在本地没有的提交。先 fetch 和查看差异，别直接强推：

```
git fetch origin
git status -sb
git log --oneline --graph --decorate --all -10
```

8. 合并冲突

运行 git status 找出 UU 文件，人工确定最终内容，测试后 add 和 commit；暂时不处理就 git merge --abort。

学生或同事只说「Git 坏了」时，让他提供这五条输出：

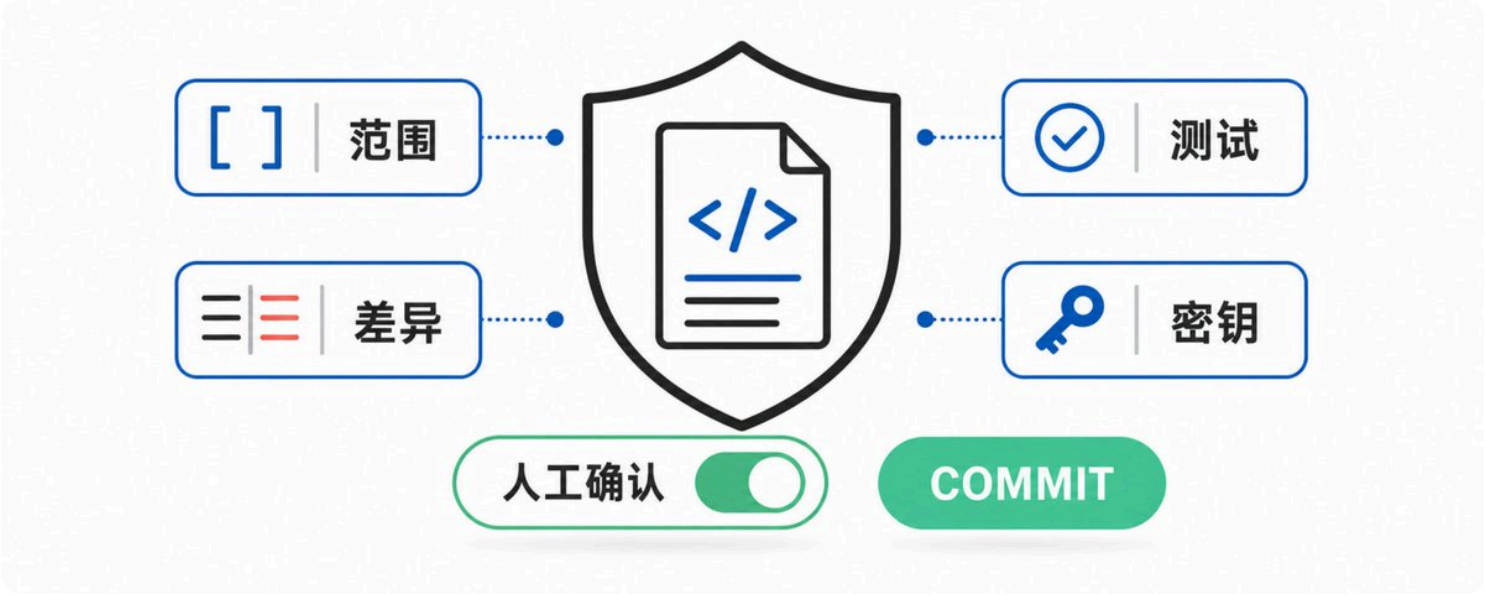
```
pwd
git status
git branch --show-current
git log --oneline -5
git remote -v
```

再补上操作系统、刚执行的完整命令和完整报错。大多数问题会很快落到目录、状态、身份、远程地址或权限中的某一层。

十六、AI 时代，Git 更像一个验收系统

AI 可以替你敲命令，但它无法自动知道哪些修改符合业务意图。一次提示词改了 20 个文件，你没看 diff、没运行项目、没检查密钥，Git 只会忠实记录这批混乱。

更稳的做法是把任务缩小，把人放在验收位置。范围、差异、测试和密钥检查都过关以后，再由人决定这批修改能不能成为一次 commit。



让 AI 操作 Git 时，也要给边界：

```
请先查看 git status 和 git diff，只总结当前变化。
不要丢弃任何未提交内容，不要执行 reset --hard、clean、force push。
完成修改后先给出验证结果，不要自动 commit 或 push。
```

会不会背命令已经没那么重要。你要能读懂状态，知道 AI 动了什么，判断验证是否足够，并在危险操作出现时叫停。

十七、把整套流程再跑一遍

```
# 1. 确认位置
pwd
ls

# 2. 初始化
git init -b main
git status

# 3. 第一次提交
git add index.html style.css .gitignore
git diff --cached
git commit -m "chore: 初始化项目"

# 4. 修改、检查、测试、再提交
git status --short
git diff
git diff --check
git add index.html
git commit -m "feat: 新增报名入口"

# 5. 分支试验
git switch -c experiment/warm-theme
git add style.css
git commit -m "style: 试验暖色主题"
git switch main
git merge experiment/warm-theme

# 6. 连接 GitHub
git remote add origin https://github.com/你的用户名/仓库名.git
git remote -v
git push -u origin main

# 7. 最后检查
git status
git log --oneline --graph --decorate --all
git remote -v
git diff --check
git ls-files
```

当你能解释每条命令改变了哪一层，也能独立处理一次错误目录、一次暂存错误和一次合并冲突，GitHub 就不再只是存代码的网站。你已经能把个人项目做成可回看、可审查、可协作的仓库。

下一步不需要继续收集命令。找一个真实的小项目，连续做 7 天：每天只完成一个小改动，看 diff，测试，提交，再推到 GitHub。提交历史会把这套东西慢慢变成你的工作习惯。

我是 Miles，一名从大厂转型 FDE 的 AI 算法专家，做过算法研发、优化部署，也做过企业培训。关注我 @miles_mazy **一起成长，一起赚钱**。