

Git 使用教程：30 分钟必学与科研进阶参考

2026-10-18

目录

第一部分：30 分钟必学	2
第二部分：科研进阶参考	5
课程首页 · English · PDF	

项目	说明
适用范围	Git 2.x; Ubuntu 终端; GitHub 协作示例
文档核验日期	2026-10-18
许可	CC BY-NC-SA 4.0 · Xiong Zeng and contributors

第一部分：30 分钟必学

时间	内容
0-5 分钟	四个区域与常用命令
5-10 分钟	安装及身份配置
10-22 分钟	一条完整的 GitHub 工作流
22-26 分钟	检查 Diff 和历史
26-30 分钟	同步远程与安全习惯

1. Git 是什么：四个区域与基本命令

Git 是分布式版本控制系统（Version Control System, VCS）。它记录代码的修改历史，使你能够比较版本、恢复旧版本、在独立分支上尝试新算法，并与团队成员合并工作。

1.1 Git 能解决哪些问题？

- **版本管理**：每次提交都对应一个可追溯的代码状态。
- **安全试验**：在实验分支上尝试新奖励函数或训练算法，不干扰稳定版本。
- **错误恢复**：找回被误改的代码，或撤销引入 Bug 的提交。
- **团队协作**：独立开发 PPO、SAC、视觉感知和 Sim2Real 模块，再合并。
- **跨机器同步**：借助 GitHub/GitLab 在本地电脑与 GPU 服务器同步源码。
- **实验复现**：记录某次训练对应的 Commit ID，与配置、随机种子、数据版本关联。

1.2 Git、GitHub、GitLab 和 VS Code

工具	作用
Git	在本地管理版本、提交、分支和合并；大部分操作无需联网。
GitHub	在线托管 Git 仓库，提供 Pull Request、代码审查和自动化功能。
GitLab	Git 托管与协作平台的另一种选择。
VS Code	编辑器；内置 Source Control 界面，可操作 Git。

Git 不是 GitHub。即使没有 GitHub 账号，你仍能运行 `git init`、`git add`、`git commit`、`git branch` 等命令。

1.3 理解工作区、暂存区、本地仓库、远程仓库



三条最常用的命令含义不同：

```
git add .
git commit -m "Implement PPO training"
git push
```

- **git add**: 将选定的文件变化放入暂存区。
- **git commit**: 将暂存内容记录为本地提交。
- **git push**: 将本地提交发送到远程仓库。

一个 Commit 可以理解为项目在某个时刻的一份版本快照；其哈希 ID 用于唯一标识该提交。工作区修改若未提交，不会因为执行 **git push** 就自动上传。

1.4 最先记住的 10 个命令

命令	用途
<code>git init</code>	初始化本地仓库
<code>git clone URL</code>	克隆远程仓库
<code>git status</code>	查看当前状态
<code>git add PATH</code>	暂存文件变化
<code>git commit -m "..."</code>	创建本地提交
<code>git push</code>	推送提交
<code>git pull</code>	获取并整合远程变更
<code>git log --oneline</code>	查看简短提交历史
<code>git switch -c NAME</code>	创建并切换分支
<code>git merge NAME</code>	合并指定分支

2. 安装与最小配置

2.1 安装 Git

打开 Ubuntu 终端：

```
sudo apt update
sudo apt install git
git --version
```

安装成功后会输出类似 `git version 2.x.y` 的版本信息；具体版本取决于 Ubuntu 发行版和软件源。

2.2 设置提交者身份

```
git config --global user.name "Your Name"
git config --global user.email "your_email@example.com"
```

这两个值写入新提交的作者身份；建议邮箱与你希望关联的 GitHub 邮箱一致。查看全局配置：

```
git config --global --list
```

也可以只给当前仓库设置不同的身份（移除 `--global`）：

```
git config user.name "Lab Account"
git config user.email "lab@example.com"
```

2.3 统一新仓库默认分支名

```
git config --global init.defaultBranch main
```

它影响之后通过 `git init` 创建的新仓库，不会自动重命名已有仓库。

3. 唯一完整工作流：从 Clone 到 Pull Request

这是整份教程唯一按操作顺序完整展开的日常工作流，替代旧版中“本地项目”“多人协作”“完整科研工作流”和“新手路线”的重复示例。

3.1 开始前确认

假设 GitHub 上已存在 `robot-policy` 仓库，你有推送分支的权限；默认主分支是 `main`。若尚无远程仓库，请在第二部分第 6 章学习 `git init` 和连接 GitHub。克隆公共仓库可使用 HTTPS；推送时需完成 GitHub 认证（Credential Manager、合适的令牌或配置 SSH）。

3.2 克隆、修改、检查、提交并发起 PR

```
# 第一次下载仓库（只执行一次）
git clone https://github.com/YOUR_USERNAME/robot-policy.git
cd robot-policy

# 从最新 main 开始一个独立任务
git switch main
git pull --ff-only
git switch -c feature/pose-estimation

# 修改已有源码后，检查即将提交的内容
git status
git diff
git add train.py
git diff --staged
git commit -m "Improve visual pose estimation"

# 首次推送该功能分支
git push -u origin feature/pose-estimation
```

然后在 GitHub 页面从 `feature/pose-estimation` 向 `main` 发起 Pull Request，等待检查与审阅；合并后，在本地执行 `git switch main` 与 `git pull --ff-only` 获取更新。不要用没有写入权限的开源上游仓库练习 `push`；必要时先 Fork。

3.3 分清哪些步骤是本地的

`git add` 只更新暂存区，`git commit` 只创建本地历史，`git push` 才向远程上传提交；GitHub PR 是审阅与合并请求，不等于本地 `git commit`。未提交的工作区改动不会被 `push` 发送。

4. 用 Status、Diff 与 Log 检查改动

调试模型训练问题时，知道代码实际改了什么往往比记住“我应该改了什么”更重要。

4.1 查看工作区状态

```
git status
```

可以看到当前分支、已暂存修改、未暂存修改和未跟踪文件。

4.2 比较代码

```
git diff          # 工作区 vs 暂存区
git diff --staged # 暂存区 vs 最近提交
git diff -- train.py # 只看某个文件
```

差异示例：

```
- learning_rate = 3e-4
+ learning_rate = 1e-4
```

- 是旧内容，+ 是新内容。

4.3 查看提交图

```
git log --oneline
git log --oneline --graph --all
```

例如：

```
* a34bd21 (HEAD -> main) Merge SAC implementation
|\
| * d72f8a1 Implement SAC algorithm
| * f61b2c9 Add replay buffer
|/
* b83fc92 Implement PPO
* 28ad613 Initial commit
```

建议经常组合使用 `git status`、`git diff`、`git log --oneline --graph --all`。每次提交前，先确认没有误加入临时文件、模型权重或敏感信息。

5. 更新远程代码与提交前安全检查

5.1 更新本地仓库：Fetch 与 Pull

```
cd robot-policy
git fetch origin
```

`fetch` 只获取远程最新对象和远程跟踪分支，不会直接合并到当前工作分支。

如果当前位于跟踪 `origin/main` 的 `main` 分支，可执行：

```
git merge origin/main
```

把更新合并到当前分支。另一种常见写法：

```
git pull
```

`pull` 相当于先 `fetch`，再按配置采用 `merge` 或 `rebase` 整合。为了避免不期望的自动合并，新手在更新稳定分支时可以用：

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

它只接受快进更新；若本地和远程历史已分叉，会停止并要求你明确处理。

注意：在开源项目原仓库上通常没有推送权限。想提交自己的修改，可以先 Fork 到个人 GitHub，再克隆自己的 Fork，或者在团队仓库中通过授权分支发起 Pull Request。

5.2 提交前的三项检查

1. `git status`：确认所在分支、待提交文件与未跟踪文件。
2. `git diff --staged`：核对实际进入本次提交的改动。
3. 不提交密码、私钥、模型权重、大规模数据或训练输出；`.gitignore` 与 Git LFS/DVC 详见第 9 章。

5.3 最小恢复动作

误将文件加入暂存区时，可执行 `git restore --staged FILE`，它只取消暂存、不删除工作区改动。需要放弃修改或撤销共享提交时，先看第 8 章的风险对照表。

第二部分：科研进阶参考

这部分按问题检索即可，无需在第一次接触 Git 时全部掌握。

6. 新建仓库、SSH 与 GitHub 认证

6.1 从零初始化一个科研项目

如果还没有远程仓库，可以先在本地建立项目：

```
mkdir robot-policy && cd robot-policy
git init
echo '# Robot Policy' > README.md
```

```
git add README.md
git commit -m "Initial commit"
```

在 GitHub 创建不带初始化文件的空仓库，然后参照 6.3 配置 origin 并首次推送。已有仓库使用 `git clone`，不要重复 `git init`。

6.2 配置 SSH 密钥

在 Ubuntu 上长期开发，推荐为自己的 GitHub 账号配置 SSH。

先检查是否已有密钥：

```
ls -al ~/.ssh
```

如果没有可用的 Ed25519 密钥，创建一个：

```
ssh-keygen -t ed25519 -C "your_email@example.com"
```

按照提示保存密钥并设置口令。若密钥文件已存在，**不要不加判断地覆盖原文件**。

启动 SSH Agent，添加私钥：

```
eval "$(ssh-agent -s)"
ssh-add ~/.ssh/id_ed25519
```

输出并复制公钥：

```
cat ~/.ssh/id_ed25519.pub
```

在 GitHub 网页进入 Settings → SSH and GPG keys → New SSH key，粘贴公钥并保存。注意：id_ed25519 是私钥，**绝不能上传或公开**；只能分享 id_ed25519.pub。

测试 SSH 连接：

```
ssh -T git@github.com
```

首次连接时要核对 GitHub 服务器指纹并确认。连接成功后通常会看到认证成功但不提供 Shell 访问的提示。

参考：[GitHub 官方 SSH 配置文档](#)。

6.3 将本地项目上传到 GitHub

在 GitHub 网站创建一个名为 robot-policy 的空仓库；如果本地已经提交过代码，先不要在网页上添加 README、.gitignore 或 License，避免产生两份独立初始历史。

```
cd robot-policy
git remote add origin git@github.com:YOUR_USERNAME/robot-policy.git
git remote -v
git branch -M main
git push -u origin main
```

解释：

- origin 是远程仓库的常用别名，不是特殊服务器。
- git remote -v 显示读取与推送地址。
- git branch -M main 将当前分支命名为 main。
- -u 为本地分支建立上游跟踪关系，以后通常可以直接运行 git push。

6.4 从 GitHub 下载开源项目

例如：

```
git clone https://github.com/ajwagen/dsrl.git
cd robot-policy
```

如果你使用 SSH 且具有相应访问权限，也可以克隆仓库提供的 SSH 地址：

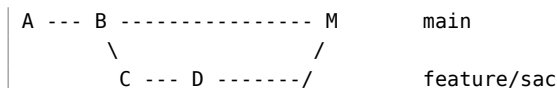
```
git clone git@github.com:ajwagen/dsrl.git
```

clone 会下载仓库历史、创建项目目录，并默认配置 origin 远程仓库。克隆后无需再次 git init。

HTTPS	SSH
公共仓库一般可匿名克隆	通常需要配置 SSH 身份验证
使用 <code>https://... URL</code>	使用 <code>git@github.com:... URL</code>
推送时需要适当的账号认证	配置密钥后推送较方便

7. 分支、团队协作与冲突处理

分支允许你同时维护稳定版本和实验性代码。例如 `main` 保存稳定 PPO，`feature/sac` 用于新的 SAC 实现：



其中 `M` 代表一次可能的合并提交；如果满足快进条件，Git 也可能不创建 `M`。

7.1 查看、创建与切换分支

```

git branch          # 列出本地分支
git branch feature/sac  # 只创建分支
git switch -c feature/sac  # 创建并切换（择一使用）
git switch main      # 切回 main
git switch feature/sac  # 切回实验分支

```

如果已经用 `git branch feature/sac` 创建了分支，就用 `git switch feature/sac` 切换，而不是再次运行同名的 `switch -c`。

Git 在切换分支时会更新工作区文件；如果未提交的修改与目标分支冲突，切换会被阻止。应先提交或使用 `git stash`。

7.2 合并到主分支

```

git switch main
git merge feature/sac

```

若没有冲突即可完成；若有冲突，参见下文 7.5 节。

7.3 删除已完成的分支

```
git branch -d feature/sac
```

`-d` 会检查分支是否已安全合并。删除对应远程分支：

```
git push origin --delete feature/sac
```

在多人项目中，更推荐通过 GitHub Pull Request 将功能分支合并到受保护的 `main`。

科研团队可按模块建立分支：

```

main
├─ feature/ppo
├─ feature/sac
├─ feature/pose-estimation
└─ feature/sim2real

```

推荐采用 **Feature Branch → Push → Pull Request → Review → Merge** 流程。

7.4 上传分支并创建 PR

```
git push -u origin feature/pose-estimation
```

在 GitHub 仓库页面创建 Pull Request，通常以 `feature/pose-estimation` 为源分支、`main` 为目标分支。团队成员可以在 PR 中检查差异、审阅、讨论、运行 CI 测试，最后合并。

团队协作的基本习惯：**不要在共享主分支直接进行未验证的大型算法改动**；让新特性先在隔离分支上通过基本测试。

两个分支修改了同一处内容时，Git 可能无法自动判断保留哪个版本。

例如同一个 `train.py`:

```
<<<<<<< HEAD
learning_rate = 0.0001
=====
learning_rate = 0.0003
>>>>>>> feature/sac
```

其中:

- <<<<<<< HEAD 到 =====: 当前分支中的版本;
- ===== 到 >>>>>>> feature/sac: 传入分支中的版本。

7.5 解决冲突

1. 运行 `git status`, 定位冲突文件。
2. 打开文件, 理解两边修改的实际意图。
3. 选择一边, 或者编写兼容两边需求的最终代码。
4. 删除所有冲突标记 (<<<<<<<、=====、>>>>>>>)。
5. 运行测试, 确认逻辑正确。
6. `git add` 标记该文件已解决, 完成合并提交。

例如决定使用:

```
| learning_rate = 0.0001
```

随后:

```
| git add train.py
| git commit
```

若想取消当前 `merge`:

```
| git merge --abort
```

如果冲突发生在 `rebase` 中, 使用的是 `git rebase --continue`、`git rebase --abort` 等命令, 而不是 `merge --abort`。

7.6 VS Code 解决冲突

VS Code 的 Merge Editor 可以比较当前版本、传入版本和合并后的结果。无论通过图形界面还是终端解决, 最终都应运行测试, 尤其是奖励函数、动作尺度、机器人坐标系等难以仅凭文本差异判断的代码。

8. 撤销、更正与历史恢复

恢复操作必须区分: 修改是否已暂存、是否已提交、是否已共享。下面的命令不等价。

8.1 尚未暂存: 放弃某个文件的修改

```
| git restore train.py
```

把工作区的 `train.py` 恢复到暂存区对应的内容。未另行保存的工作区修改可能丢失。

8.2 已 `git add`, 尚未 `commit`: 取消暂存

```
| git restore --staged train.py
```

修改仍保留在工作区, 只是不再参与下一次提交。

8.3 已 `commit`, 尚未共享: 撤销提交并保留改动

假设最近两次提交:

```
| a82bc31 Bad reward modification
| b41d2ab Working PPO version
```

撤销最近提交, 但仍保留暂存:

```
| git reset --soft HEAD~1
```

撤销最近提交并取消暂存，但保留工作区改动：

```
git reset HEAD~1
```

连同相应工作区修改一起丢弃：

```
git reset --hard HEAD~1
```

reset --hard 是破坏性操作。执行前务必确认代码、实验结果和未提交内容不需要保留。

8.4 已 push / 其他人可能已拉取：用 Revert

```
git revert a82bc31
```

revert 创建一个新的提交，尝试抵消某个旧提交所引入的改动，保留可审计的共享历史。若修改已经与后续提交交织在一起，可能需要解决冲突。

操作	影响	典型场景
git restore FILE	丢弃文件未暂存修改	放弃本地临时改动
git restore --staged FILE	取消暂存，保留改动	文件被误加到暂存区
git reset --soft HEAD~1	回退分支指针，保留暂存	整理未共享的提交
git reset HEAD~1	回退分支指针，保留工作区	重新选择修改进行提交
git reset --hard HEAD~1	回退并丢弃相应修改	确认无需保留时
git revert HASH	增加反向提交	撤销已经共享的提交

基本原则：本地未共享的历史可以谨慎 **reset**；共享历史优先 **revert**。

9. 忽略文件、Git LFS 与 DVC

机器学习项目常见数 GB 的数据集、模型权重、录像和训练日志，它们通常**不适合直接提交到普通 Git 仓库**。

9.1 推荐模板

在项目根目录创建文件：

```
touch .gitignore
nano .gitignore
```

示例：

```
# Python cache
__pycache__/
*.pyc

# Local environments
.venv/
venv/

# Secrets
.env
*.pem

# Model artifacts
*.pt
*.pth
*.ckpt
checkpoints/

# Training outputs
outputs/
logs/
wandb/
```

```
# Large datasets
data/
datasets/

# ROS recordings
*.bag
*.db3

# Optional IDE state
.vscode/
```

根据项目调整规则：例如某些项目确实需要跟踪小型测试权重，或者共享部分 `.vscode/` 配置，不应机械地全部忽略。

9.2 .gitignore 的作用范围

规则主要影响尚未跟踪的文件。若 `checkpoint.pt` 以前已经被 Git 跟踪，即使现在写了 `*.pt`，它也不会自动退出版本控制。

要停止跟踪、但保留本地文件：

```
git rm --cached checkpoint.pt
git commit -m "Stop tracking checkpoint"
```

这不会清除历史提交中的旧副本。若意外上传 API Key、口令或其他机密，应该立即撤销/轮换凭据，并评估是否需要清理公开历史。

9.3 模型、数据与代码分离

建议：

- Git：源码、配置、脚本、文档、小型测试资源。
- 对象存储/实验平台：模型检查点、视频、训练日志和大型数据集。
- 实验元数据：保存代码 Commit ID、数据集版本、权重文件哈希和训练参数。

需要版本化大文件时，可以另行评估 Git LFS、DVC 或专门的模型/数据管理方案。

9.4 何时选择 Git LFS、DVC 或对象存储？

方案	管理对象	核心注意事项
普通 Git + <code>.gitignore</code>	代码、配置、文档；忽略临时输出	<code>.gitignore</code> 不会删除已上传历史
Git LFS	需要与 Git 版本关联的较大二进制文件	Git 保存指针，二进制内容由 LFS 存储；服务端需支持，且可能有配额
DVC + 外部存储	大规模数据集及其版本/流水线元数据	Git 追踪 <code>.dvc</code> 等元数据；实际数据需配置远程存储并单独同步
对象存储 / 实验平台	海量视频、模型检查点、训练日志	保存版本、路径、校验和，不能只保存一个会变化的 URL

使用 Git LFS 的最小示例（须先安装 Git LFS，且仓库服务支持 LFS）：

```
git lfs install
git lfs track "*.onnx"
git add .gitattributes models/policy.onnx
git commit -m "Track policy model via LFS"
```

DVC 可以对数据目录执行 `dvc add datasets/trajectories`，Git 提交生成的 `.dvc` 元数据及忽略规则；同时必须配置并同步 DVC 远程存储，才能让其他机器获取实际数据。不要误认为安装 LFS 或 DVC 就自动提供无限远程空间。

10. Stash、Rebase、Show 与 Blame

10.1 git stash: 临时收起未提交改动

你正在开发 SAC，却需要立即切换回 main 修复问题：

```
git stash
```

默认主要收起已跟踪文件的改动；若需要一起保存未跟踪文件：

```
git stash push -u -m "WIP SAC training"
```

然后：

```
git switch main
# 修复并提交问题
git switch feature/sac
git stash pop
```

命令	用途
git stash list	查看 Stash 记录
git stash show -p	查看最近 Stash 的差异
git stash pop	恢复修改；成功后移除该条 Stash
git stash apply	恢复修改，但保留该条 Stash

Stash 恢复也可能出现冲突，不能把它当作长期可靠备份。

10.2 git rebase: 把实验分支移到新基线

原始历史：

```
main:    A -- B -- C
feature: A -- B -- D -- E
```

运行：

```
git switch feature
git rebase main
```

概念上变为：

```
main:    A -- B -- C
feature: A -- B -- C -- D' -- E'
```

rebase 会重放提交并生成新的提交 ID；适合整理自己的私有分支，不要随意重写其他人正在使用的共享历史。

10.3 查看某个提交、追踪某行代码

```
git show <commit-id>    # 查看一次提交的详细改动
git blame train.py       # 查看每行最近由哪次提交修改
```

git blame 在追踪“奖励缩放何时被修改”“哪个提交改变了控制频率”时很实用，但它是定位历史的工具，不等于判断责任。

10.4 查看所有分支

```
git branch -a
```

可能显示：

```
* main
  feature/sac
  remotes/origin/main
  remotes/origin/feature/ppo
```

远程跟踪分支（remotes/origin/...）代表本地记录的远程状态；执行 git fetch 可以更新这些记录。

11. 科研版本追踪与实验复现

11.1 按任务创建分支

典型科研场景	推荐方式
新增 PPO / SAC 算法	feature/ppo / feature/sac
探索新的奖励函数	experiment/reward-v2
实现在线 Residual Velocity 适配	experiment/residual-velocity
修改位姿估计或相机标定	feature/pose-estimation
修复线上机器人控制 Bug	fix/control-latency
论文复现实验	用 Commit ID 或 Tag 固定发布代码

11.2 每次训练记录 Commit ID

```
git rev-parse HEAD
```

建议和实验配置一起保存，例如：

```
experiment:
  name: residual_velocity_v3
  git_commit: a82bc31
  random_seed: 42
training:
  learning_rate: 0.0003
  batch_size: 256
```

上述 Commit ID 是演示用短哈希，正式记录最好写入命令输出的**完整哈希**。

重要：Commit ID 只标记已经提交的代码状态。如果实验启动时存在未提交修改，仅保存哈希并不能完整复现。正式运行前请检查 `git status`，必要时记录补丁或先建立实验提交。

11.3 把代码、配置和大文件管理分开

- **Git 仓库**：源码、配置、训练入口、评测脚本、README、环境说明。
- **实验输出目录**：TensorBoard/W&B 日志、视频、权重和大型缓存；通常加入 `.gitignore`。
- **复现记录**：Commit ID、环境与依赖版本、数据集版本、训练随机种子、仿真器版本、GPU/驱动信息。
- **机器人部署**：在发布稳定控制策略时，可创建 Tag（如 `v1.0-deploy`）并记录对应的控制参数和硬件版本。

例如为经过验证的版本添加 Tag：

```
git tag -a v1.0-deploy -m "Validated robot deployment"
git push origin v1.0-deploy
```

11.4 一个实验至少应关联哪些版本信息？

一次仿真或真机实验应绑定：**完整 Commit ID**、**工作区是否干净**、**配置文件**、**随机种子**、**依赖与仿真器版本**、**数据集版本/校验和**、**模型 checkpoint 哈希**。真机还应注明机器人 URDF/固件/相机外参/控制频率等可能影响结果的配置。

Commit ID 只定位已提交的代码；若 `git status` 仍显示修改，必须补充 Patch 或先建立可追踪提交。它不代替数据和环境管理。

12. 常见错误与安全排查

12.1 fatal: not a git repository

当前目录及其父目录都找不到有效 Git 仓库。检查：

```
pwd
ls -a
```

确认是否进入了项目目录；只有确实要创建**新仓库**时才运行 `git init`。

12.2 Permission denied (publickey)

SSH 身份验证失败。检查：

```
ssh -T git@github.com
```

确认本地 SSH Agent 已加载正确密钥，并且公钥已添加到 GitHub 对应账号。

12.3 Updates were rejected

远程通常有本地不存在的提交，直接推送无法快进。先检查：

```
git fetch origin
git status
```

对于你自己的功能分支且当前上游配置正确，可根据团队规范选择：

```
git pull --rebase
# 解决可能出现的冲突并测试
git push
```

不要为了消除报错直接 **git push --force**；这可能覆盖别人的提交。

12.4 Your local changes would be overwritten

当前未提交修改可能被切换/同步操作覆盖。先二选一：

```
# 方案 A: 提交当前工作
git add .
git commit -m "Save work in progress"

# 方案 B: 临时保存（包括未跟踪文件）
git stash push -u
```

然后再重复原操作。

12.5 常见错误处理原则

遇到不熟悉的 Git 报错时，先运行 **git status**，阅读 Git 给出的后续操作提示，再判断是处于 merge、rebase 还是普通状态。不要在不清楚仓库状态时连续执行 **reset --hard**、**clean -fd** 或强制推送。

13. VS Code 中的 Git 界面

在项目目录运行：

```
code .
```

左侧打开 **Source Control（源代码管理）**，快捷键通常是 **Ctrl + Shift + G**。

操作	VS Code 中的位置或方式
查看修改文件	Source Control 文件列表
比较修改前后代码	点击列表中的修改文件
暂存	文件旁的 + / Stage Changes
提交	输入提交信息并选择 Commit
推送	Push 或 Sync Changes
切换分支	状态栏分支名称或命令面板
解决冲突	Merge Editor

假设你将：

```
reward = success_reward
```

改成：

```
reward = success_reward + progress_reward
```

VS Code 的 Diff Editor 会直观显示删除和新增的行。对于涉及多文件配置的强化学习实验，在提交前检查一次完整 Diff，通常可以发现意外的超参数修改。

推荐搭配：终端更适合准确输入和记录 Git 命令；VS Code 更适合比较改动、定位冲突、审查大段代码。

14. 进阶命令速查与官方资料

分类	命令	用途
初始化	<code>git init</code>	创建本地仓库
克隆	<code>git clone URL</code>	下载仓库
状态	<code>git status</code>	查看当前状态
差异	<code>git diff</code>	查看未暂存改动
历史	<code>git log --oneline</code>	查看提交历史
暂存	<code>git add .</code>	暂存当前目录变化
提交	<code>git commit -m "msg"</code>	创建提交
上传	<code>git push</code>	推送本地提交
获取	<code>git fetch</code>	下载远程状态，不自动合并
同步	<code>git pull --ff-only</code>	仅允许快进更新
分支	<code>git branch</code>	列出本地分支
分支	<code>git switch -c name</code>	新建并切换分支
分支	<code>git switch name</code>	切换已有分支
合并	<code>git merge name</code>	合并指定分支
撤销	<code>git restore file</code>	丢弃未暂存的文件改动
撤销	<code>git restore --staged file</code>	取消暂存
撤销	<code>git revert HASH</code>	生成反向提交
暂存	<code>git stash</code>	临时收起改动
暂存	<code>git stash pop</code>	恢复 Stash
信息	<code>git show HASH</code>	查看指定提交
信息	<code>git blame file</code>	追踪文件行的提交历史

14.1 官方参考资料

- [Pro Git \(中文\)](#)
- [Pro Git \(English\)](#)
- [Git 官方命令参考](#)
- [GitHub: 通过 SSH 连接](#)
- [GitHub: Pull Request 指南](#)

结语：对于科研代码，Git 最重要的价值不是“把代码上传到 GitHub”，而是将**源代码版本、实验变更与可复现结果**联系起来。先保持“小步提交、分支实验、审查 Diff、避免上传大文件”的习惯，再逐步扩展到团队协作。