Git 全操作指南（VSCode 远程 Linux 版）

作者：Mr-Auqin 联系方式：2480015679@qq.com

适配场景

Windows 端 VSCode 通过 SSH 连接 Linux 虚拟机，操作 Linux 内代码，**所有 Git 操作依赖 Linux 端已安装配置的 Git**（已配置 user.name/email）。

文档说明

纯基础 Markdown 格式，代码块语法 100% 兼容所有编辑器，复制后表格、代码块均正常显示。

前置准备：Git 下载安装与用户信息配置

一、Linux 端（虚拟机 / 服务器）安装 Git

场景 1：Ubuntu/Debian 系列（如 Ubuntu 20.04/22.04）

bash

运行

```
# 1. 更新软件源（可选，确保下载最新版本）
sudo apt update
# 2. 安装Git
sudo apt install git -y
# 3. 验证安装成功（显示版本号即生效）
git --version
```

场景 2：CentOS/RHEL 系列（如 CentOS 7/8/9）

bash

运行

```
# 1. 安装Git（CentOS 7需先装epel源，CentOS 8/9可直接装）
# CentOS 7：
sudo yum install epel-release -y
sudo yum install git -y
# CentOS 8/9：
sudo dnf install git -y
# 2. 验证安装成功
git --version
```

二、Windows 端（本地）安装 Git（可选，如需本地同步代码）

1. 下载安装包：访问 Git 官方下载页 → https://git-scm.com/download/win

2. 安装步骤（无脑下一步即可，关键选项建议）：

   - 选择安装路径（默认 C:\Program Files\Git 即可）
   - 选择编辑器：推荐「Visual Studio Code」（与你的开发工具一致）
   - 其余选项保持默认，点击「Next」直到安装完成

3. 验证安装：打开 Windows 终端 / CMD，输入`git --version`，显示版本号即成功

三、核心配置：Git 用户信息（Linux/Windows 端都要配置，必做！）

Git 需要绑定用户名和邮箱，用于标识提交记录的作者，**不配置会导致提交失败。**

1. 全局配置（推荐，所有项目共用）

bash

运行

```
# 配置用户名（替换为你的自定义名称，如"zhangsan"）
git config --global user.name "你的用户名"
# 配置邮箱（替换为你的常用邮箱，如"zhangsan@xxx.com"）
git config --global user.email "你的邮箱地址"
```

2. 检查配置是否生效

bash

运行

```
# 查看所有全局配置信息
git config --global --list
# 单独查看用户名/邮箱
git config --global user.name
git config --global user.email
```

3. 修改配置（如需调整）

bash

运行

```
# 重新执行配置命令即可覆盖原有信息
git config --global user.name "新的用户名"
git config --global user.email "新的邮箱地址"
```

4. 局部配置（可选，仅当前项目生效）

如果需要为单个项目配置不同的用户信息，进入项目目录后执行：

bash

运行

```
# 进入项目文件夹
cd /你的项目路径（Linux）或 cd D:\你的项目路径（Windows）
# 配置局部用户信息（去掉--global参数）
git config user.name "项目专属用户名"
git config user.email "项目专属邮箱"
```

四、常见问题排查

1. 安装失败：Linux 端确保 sudo 权限，Windows 端以管理员身份运行安装包；
2. 配置不生效：检查命令是否输错（无拼写错误、引号完整），重启终端后重试；
3. 提交时报错「Please tell me who you are」：未配置用户信息，重新执行全局配置命令即可。

一、核心概念速览

概念	通俗解释	核心作用	操作优先级
拉取（Pull）	下载远程最新代码 + 合并到当前分支（=fetch+merge）	同步远程代码，避免协作冲突	开工必做
分支（Branch）	独立的动态开发线，可单独提交，不影响主分支	隔离功能开发 / BUG 修复，保护主分支稳定	开发必用
标签（Tag）	指向特定提交的静态标记，不可修改 / 提交	标记正式发布版本（如 v1.0.0）	版本发布用
暂存（Add）	告知 Git「将这些修改纳入下一次提交」	筛选提交内容，衔接代码修改与提交	提交前必做
提交（Commit）	记录暂存的修改，生成本地版本记录，关联作者信息	保存本地代码修改状态	改完必做
推送（Push）	将本地提交记录上传到远程仓库	同步本地代码到远程，支持团队协作	提交后按需做
暂存栈（Stash）	临时保存未提交的修改，清空工作区干净状态	切换分支 / 回退版本时保留未完成修改	应急操作必用
操作日志（Reflog）	记录 HEAD 指针所有移动轨迹（提交 / 切换 / 回退等）	恢复误回退 / 误删除的版本 / 分支	数据恢复必用
版本回退（Reset）	将代码库 / 工作区回退到指定历史版本	撤销错误修改，恢复到稳定版本	应急修复必用

二、完整项目 Git 实操流程

阶段 1：项目初始化（二选一，首次操作）

情况 A：本地 Linux 新建项目，关联远程仓库（Gitee/GitHub/GitLab）

1. VSCode 打开 Linux 内项目文件夹 → 左侧「源代码管理（Ctrl+Shift+G）」→ 点击「初始化仓库」→ 选择当前项目文件夹；
2. 远程平台创建**空仓库**，复制仓库地址（SSH/HTTPS 均可）；
3. VSCode 远程 Linux 终端执行以下命令，完成关联：

bash

运行

```
# origin为远程仓库默认别名，替换为你的实际远程地址
git remote add origin 你的远程仓库地址
git remote -v  # 验证关联成功（输出远程仓库地址即生效）
```

情况 B：已有远程仓库，直接克隆到 Linux

1. VSCode 左侧 Git 图标 → 点击「克隆仓库」→ 粘贴远程仓库地址 → 选择 Linux 内代码保存路径 → 等待克隆完成；
2. 克隆后 VSCode 自动打开项目，

   已默认关联远程仓库

，可直接进入开发。

> 扩展：如需自定义克隆后的文件夹名称，终端执行：`git clone 远程仓库地址 自定义文件夹名`

阶段 2：日常开发核心流程（每日必用，口诀：拉取→建分支→开发→暂存→提交→推送）

1. 拉取最新代码（开工第一步，避免冲突）
   - VSCode 界面操作：Git 面板 → 右上角「…」→ 选择「拉取」；
   - 终端命令（推荐，更稳定）：

bash

运行

```
git switch main/dev  # 先切换到主分支/开发分支
git pull origin main/dev  # 拉取对应远程分支最新代码
# 若拉取时提示"refusing to merge unrelated histories"，执行：
git pull origin main/dev --allow-unrelated-histories
```

1. 创建并切换功能分支（禁止直接在 main/dev 分支开发！）
   - （推荐我个人偏好的 git checkout 方式，简洁高效）
   - 终端命令：

bash

运行

```
# 方式1：git checkout 一键创建+切换（推荐）
git checkout -b feature/你的功能名
# 方式2：git switch 方式（兼容参考）
git switch -c feature/你的功能名
```

✅ 分支命名规范（新手直接照用）：

- 功能开发：feature / 功能名（如 feature/uv-tia）
- 紧急修复：hotfix/BUG 描述（如 hotfix/uart-error）

1. **开发代码**：在新建的功能分支中编写 / 修改代码，所有操作均与主分支隔离，无任何影响；

2. **临时暂存未完成修改**（如需切换分支 / 回退版本时用）

   - 终端命令：

bash

运行

```
git stash  # 暂存所有未提交修改，清空工作区
git stash list  # 查看暂存栈中的所有记录
```

1. 暂存修改（提交前必做）
    - VSCode 界面操作：Git 面板「更改」（红色标识）→ 点击「+」（暂存所有）/ 右键单个文件「暂存更改」；
    - 终端命令：

bash

运行

```
git add .  # 暂存所有修改（推荐）
# 若仅暂存单个文件：git add 文件名/文件路径
```

1. 提交修改（新手仅用「普通提交」）
    - VSCode 界面操作：Git 面板输入框填写**有意义的提交说明**（如 “完成 TIA 电路参数配置”）→ 点击「提交」/ 按 Ctrl+Enter；
    - 终端命令：

bash

运行

```
# -m 后跟提交说明，必须填写，不可省略
git commit -m "完成TIA电路参数配置"
```

1. 推送分支到远程（让团队看到你的代码）
    - VSCode 界面操作：Git 面板 → 右上角「…」→ 「推送」→ 首次推送提示「设置上游分支」，点击「确定」即可；
    - 终端命令（首次需加`-u`绑定本地与远程分支，后续直接 git push）：

bash

运行

```
# 替换为你的实际功能分支名
git push -u origin feature/你的功能名
git push  # 后续推送无需加-u，直接执行
```

阶段 3：功能完成，合并到主 / 开发分支（协作关键）

1. 切换到**目标合并分支**（如 dev / 开发分支，无 dev 则用 main），并拉取最新代码：

bash

运行

```
git switch dev  # 切换到目标分支
git pull origin dev  # 拉取远程最新，避免合并冲突
```

1. 合并功能分支
    - VSCode 界面操作：Git 面板 → 底部「分支」→ 右键要合并的功能分支 → 「合并到当前分支」；
    - 终端命令：

bash

运行

```
# 在dev/main分支执行，替换为你的功能分支名
git merge feature/你的功能名
```

1. **处理冲突（如有）**：VSCode 会自动高亮冲突区域，手动选择「保留当前代码 / 保留传入代码 / 合并两者」→ 保存文件 → 重新暂存 → 提交；

2. **推送合并后的代码到远程**：

bash

运行

```
git push origin dev  # 推送到远程对应分支
```

1. **（可选）删除本地功能分支**（用完即删，保持分支整洁）：

bash

运行

```
git branch -d feature/你的功能名
```

阶段 4：版本发布，打标签（项目里程碑）

仅当 main 分支代码**测试通过、可正式发布**时操作，标签为**永久版本标记**，不随意创建。

1. 切换到 main 主分支，拉取最新代码：

bash

运行

```
git switch main
git pull origin main
```

1. 打标签（推荐「附注标签」，带版本说明，更规范）
    - VSCode 界面操作：Git 面板 → 右上角「…」→ 「标签」→ 「创建标签」→ 输入标签名（如 v1.0.0）→ 填写标签说明 → 点击「创建」；
    - 终端命令（推荐）：

bash

运行

```
# -a=附注标签，-m=标签说明，替换为你的版本号和说明
git tag -a v1.0.0 -m "V1.0.0正式版：支持紫外局放检测核心功能"
```

1. **推送标签到远程**（关键！否则远程无此版本标记）：

bash

运行

```
git push origin v1.0.0  # 推送单个标签（推荐）
# git push origin --tags  # 批量推送所有本地标签
```

1. 找回历史标签版本（如需回滚 / 查看旧版本代码）：
    - VSCode 界面操作：Git 面板 → 展开「标签」→ 右键目标标签 → 「检出标签」；
    - 终端命令：

bash

运行

```
git checkout v1.0.0  # 切换到对应版本（仅查看，修改需新建分支）
```

阶段 5：版本回退与恢复（应急处理）

1. 查看版本操作日志（找回所有历史版本）：

bash

运行

```
git reflog  # 查看HEAD指针所有操作轨迹，获取目标版本哈希值
```

1. 回退到指定历史版本：

bash

运行

```
# 先暂存未提交修改（避免丢失）
git stash
# 硬回退到目标版本（替换为实际哈希值，如d48ed85）
git reset --hard 目标版本哈希值
```

1. 恢复到最新版本（误回退后补救）：

bash

运行

```
# 先查reflog获取最新版本哈希值
git reflog
# 切回最新版本（替换为实际哈希值，如e350f50）
git reset --hard 最新版本哈希值
# 恢复暂存的修改
git stash pop
```

三、易混提交选项区别（避坑重点）

VSCode 提交时出现的 3 个选项，核心差异及使用场景明确区分，新手 90% 场景仅用**普通提交**。

选项名称	底层命令	核心效果	适用场景	严格注意事项
普通提交	git commit -m "说明"	新建一条本地版本记录，记录暂存的所有修改	日常开发、改 BUG、加功能（90% 场景）	1. 必须先暂存文件；2. 必须填写提交说明
提交（修改）	git commit --amend	不新建记录，修改上一次的提交（合并新暂存内容 / 修改提交说明）	1. 上一次提交漏加文件；2. 提交说明写错；3. 小修改不想多一条记录	已推送到远程的提交，禁止使用！会导致本地与远程记录不一致
提交（已署名）	git commit -S -m "说明"	提交时附加 GPG 加密签名，验证提交者真实身份	1. 开源项目强制要求；2. 企业高安全要求项目；3. 核心代码提交	需提前在 Linux 配置 GPG 密钥并上传公钥到远程平台，新手暂不用

四、按场景分类 Git 命令速查表

场景 1：仓库基础管理（首次操作 / 仓库配置）

功能需求	终端命令	VSCode 界面操作	备注
初始化本地仓库	git init	源代码管理 → 「初始化仓库」	仅 Linux 本地新建项目用
克隆远程仓库到 Linux	git clone 远程仓库地址	源代码管理 → 「克隆仓库」→ 粘贴地址 + 选路径	已有远程仓库直接用，自动关联远程
克隆远程仓库并自定义文件夹名称	git clone 远程仓库地址 自定义名	无直接界面操作，需终端执行	需自定义仓库文件夹名时使用
关联本地仓库与远程仓库	git remote add origin 远程地址	无直接界面操作，需终端执行	仅本地初始化后用
查看远程仓库关联信息	git remote -v	无直接界面操作，需终端执行	验证是否关联成功
查看 Git 全局配置（name/email）	git config --global --list	无直接界面操作，需终端执行	验证 Linux 端 Git 配置是否就绪

场景 2：分支操作（开发核心，高频使用）

功能需求	终端命令	VSCode 界面操作	备注
查看所有本地分支	git branch	底部状态栏 → 点击分支名 → 查看列表	带 * 的为当前分支
查看所有远程分支	git branch -r	无直接界面操作，需终端执行	查看远程仓库的所有分支
新建并切换分支（checkout）	git checkout -b 分支名	底部状态栏 → 「创建新分支」→ 输入名称	你偏好的方式，简洁一步到位
新建并切换分支（switch）	git switch -c 分支名	底部状态栏 → 「创建新分支」→ 输入名称	Git 2.23+ 新增，语义更清晰
切换到已有分支（checkout）	git checkout 分支名	底部状态栏 → 选择目标分支	切换前建议暂存本地修改
切换到已有分支（switch）	git switch 分支名	底部状态栏 → 选择目标分支	Git 2.23+ 新增，语义更清晰
切换到远程分支（本地未创建）	git checkout 远程分支名	底部状态栏 → 选择远程分支 → 自动创建本地分支并切换	自动关联远程分支，无需额外配置
合并指定分支到当前分支	git merge 待合并分支名	源代码管理 → 底部分支 → 右键目标分支 → 「合并到当前分支」	合并前先拉取最新代码
删除本地已合并分支	git branch -d 分支名	底部状态栏 → 右键目标分支 → 「删除分支」	-d 为安全删除，未合并分支需用 -D

场景 3：提交 / 拉取 / 推送（日常开发，最高频）

功能需求	终端命令	VSCode 界面操作	备注
暂存所有修改	git add .	源代码管理 → 「更改」 → 点击「+」	推荐使用，高效便捷
暂存单个文件	git add 文件名 / 路径	源代码管理 → 右键单个文件 → 「暂存更改」	仅需提交部分修改时用
普通提交	git commit -m "提交说明"	输入说明 → 点击「提交」/Ctrl+Enter	新手核心提交方式
修正上一次提交	git commit --amend	提交框旁 → 选择「提交（修改）」	未推送远程时可用
拉取远程分支最新代码	git pull origin 分支名	源代码管理 → 「…」→ 「拉取」	开工必做，避免冲突
拉取时允许合并无关历史	git pull 分支名 --allow-unrelated-histories	无直接界面操作，需终端执行	解决 "refusing to merge unrelated histories" 错误
首次推送本地分支到远程	git push -u origin 分支名	源代码管理 → 「…」→ 「推送」→ 确认上游分支	-u 绑定本地与远程分支
后续推送本地分支	git push	源代码管理 → 「…」→ 「推送」	绑定上游后直接使用

场景 4：标签操作（版本发布，低频使用）

功能需求	终端命令	VSCode 界面操作	备注
创建附注标签（推荐）	git tag -a 标签名 -m "说明"	源代码管理 → 「…」→ 「标签」→ 「创建标签」	带说明的标签更易维护
查看所有本地标签	git tag	源代码管理 → 展开「标签」列表	按创建顺序排列
推送单个标签到远程	git push origin 标签名	无直接界面操作，需终端执行	推荐单个推送，精准可控
推送所有标签到远程	git push origin --tags	无直接界面操作，需终端执行	批量发布多个版本时用
检出标签版本（查看旧版本）	git checkout 标签名	源代码管理 → 右键标签 → 「检出标签」	处于 “分离头指针” 状态，不可直接修改
删除本地标签	git tag -d 标签名	无直接界面操作，需终端执行	打错标签时使用

场景 5：临时暂存（Stash，应急操作）

功能需求	终端命令	VSCode 界面操作	备注
暂存所有未提交的修改	git stash	无直接界面操作，需终端执行	清空工作区，切换分支 / 回退版本前必用
查看暂存栈所有记录	git stash list	无直接界面操作，需终端执行	查看所有暂存的修改记录
恢复最新的暂存修改	git stash pop	无直接界面操作，需终端执行	恢复后从暂存栈删除该记录
恢复指定的暂存修改	git stash pop stash@{n}	无直接界面操作，需终端执行	n 为暂存序号，如 stash@{1}
清空所有暂存记录	git stash clear	无直接界面操作，需终端执行	确认暂存修改无用后再执行，不可恢复

场景 6：版本回退与恢复（应急修复）

功能需求	终端命令	VSCode 界面操作	备注
查看提交历史（版本记录）	git log	源代码管理 → 「…」→ 「查看日志」	查看提交时间 / 作者 / 哈希值 / 说明
查看所有操作日志（找回版本）	git reflog	无直接界面操作，需终端执行	记录所有 HEAD 移动轨迹，恢复误操作必用
硬回退到指定版本	git reset --hard 版本哈希值	无直接界面操作，需终端执行	彻底回退代码库 + 工作区，需先 stash
放弃本地未暂存的修改	git checkout -- 文件名	源代码管理 → 右键未暂存文件 → 「放弃更改」	修改不可恢复，谨慎操作
取消单个文件的暂存	git reset HEAD 文件名	源代码管理 → 右键已暂存文件 → 「取消暂存」	仅取消暂存，不删除代码修改
取消所有文件的暂存	git reset HEAD .	源代码管理 → 右键「暂存的更改」→ 「取消暂存所有更改」	仅取消暂存，不删除代码修改

场景 7：其他高频实用操作（问题排查 / 应急处理）

功能需求	终端命令	说明	风险提示
查看工作区状态	git status	区分未暂存（红）/ 已暂存（绿）/ 干净状态，验证 stash 效果	无风险，日常操作后可执行验证
查看分支跟踪关系	git branch -vv	查看本地分支关联的远程分支	无风险，验证分支关联是否正确
强制删除未合并分支	git branch -D 分支名	非安全删除，未合并的修改可能丢失	确认分支无用后再执行，谨慎操作