

Git 实践

这篇文章是 git 的常用使用方法。

目录

- [Git 实践](#)
 - [目录](#)
 - [安装 Git](#)
 - [Windows](#)
 - [Linux](#)
 - [方法一：使用系统命令安装 git](#)
 - [方法二：使用 Linuxbrew 安装最新版本的 git](#)
 - [方法三：源码编译安装](#)
 - [配置 Git](#)
 - [用户信息和推送配置](#)
 - [代理配置](#)
 - [HTTP 代理](#)
 - [SSH 代理](#)
 - [从 Windows 凭证中移除错误的凭证](#)
 - [开发人员](#)
 - [克隆和检出 develop 分支](#)
 - [克隆远程仓库](#)
 - [检出 develop 分支](#)
 - [分支操作](#)
 - [创建新分支](#)
 - [切换分支](#)
 - [查看全部分支](#)
 - [查看分支关联](#)
 - [推送分支到远程仓库](#)
 - [合并功能分支到 develop 分支](#)
 - [方法一：通过 merge 合并代码](#)
 - [方法二：通过 rebase 合并代码](#)
 - [推送 develop 分支](#)
 - [删除分支](#)
 - [提交代码](#)
 - [检查工作区状态](#)
 - [提交变更](#)
 - [修正最近一次提交变更的注释](#)
 - [查看提交历史](#)
 - [查看某个文件的提交历史](#)
 - [解决合并冲突](#)
 - [高级操作](#)
 - [查看远程仓库地址](#)
 - [修改远程仓库地址](#)
 - [贮藏和恢复](#)
 - [比较代码](#)
 - [放弃对某个文件的修改](#)
 - [从提交历史中获取文件](#)
 - [放弃全部修改](#)
 - [重置到某次提交](#)
 - [检出某次提交](#)
 - [维护人员](#)
 - [创建新项目](#)
 - [在 GitLab 中创建一个项目](#)
 - [克隆到本地](#)
 - [添加代码文件](#)
 - [添加 README.md 文件](#)
 - [添加 .gitignore 文件](#)
 - [忽略已被跟踪的文件或目录](#)
 - [Git LFS](#)
 - [提交并推送 master 分支](#)
 - [创建并推送 develop 分支](#)
 - [配置 GitLab 项目权限](#)
 - [打 tag](#)
 - [添加 tag](#)
 - [推送 tag](#)
 - [查看 tag 列表](#)
 - [删除 tag](#)
 - [找到错误的提交](#)
 - [回滚到某次提交](#)
 - [统计贡献情况](#)
 - [Git 图形化工具](#)
 - [SourceTree 工具](#)
 - [VSCode 的 GitLens 扩展](#)
 - [Git 工具自带的图形工具](#)

安装 Git

Windows

1. 到这个页面下载安装程序：<http://www.git-scm.com/download/win>
2. 运行安装程序，在安装向导中，请确定下面配置
 - 在 Select Components 页面勾选下面选项：
 - [x] Git Bash here

- [x] Git LFS (Large File Support)
 - 在 Configuring the line ending conversions 页面选择:
 - [x] Checkout as-is, commit as-is
3. 安装完成后, 在资源管理器的文件夹右键菜单中会包含 Git Bash here 菜单项, 执行它即可进入 Git Bash 命令行界面。

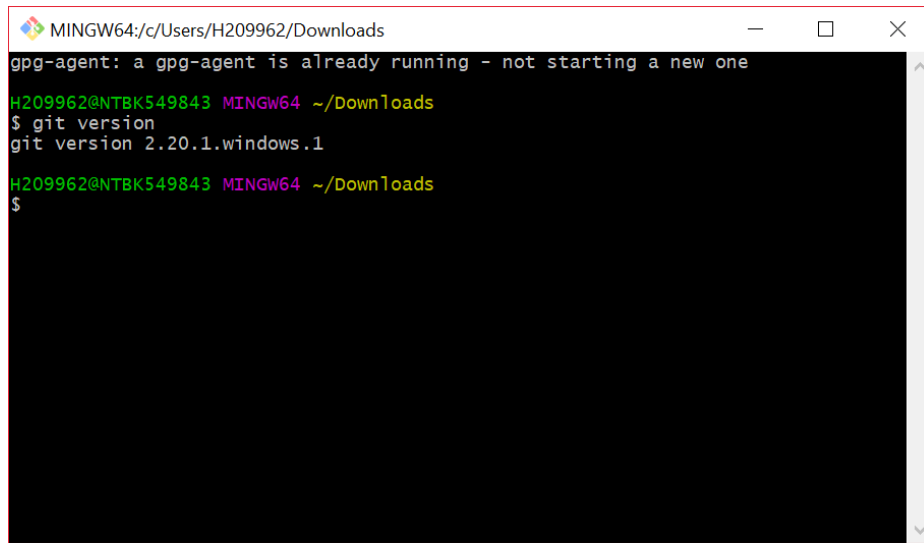


图1 在 Downloads 目录打开 Git Bash, 并执行了 git version

Linux

方法一: 使用系统命令安装 git

- Fedora, CentOS, or Red Hat

```
sudo yum install git
```
- Debian, or Ubuntu

```
sudo apt-get install git
```

方法二: 使用 Linuxbrew 安装最新版本的 git

1. 安装开发工具集
 - Fedora, CentOS, or Red Hat

```
sudo yum groupinstall 'Development Tools' && sudo yum install curl file git
```
 - Debian, or Ubuntu

```
sudo apt-get install build-essential curl file git
```
2. 安装 linuxbrew

```
sudo sh -c "$(curl -fsSL https://raw.githubusercontent.com/Linuxbrew/install/master/install.sh)"
```

```
test -d ~/.linuxbrew && eval "$(~/.linuxbrew/bin/brew shellenv)
```

```
test -d /home/linuxbrew/.linuxbrew && eval "$(home/linuxbrew/.linuxbrew/bin/brew shellenv)
```

```
test -r ~/.bash_profile && echo "eval \"\$(brew --prefix)/bin/brew shellenv\"" >> ~/.bash_profile
```

```
echo "eval \"\$(brew --prefix)/bin/brew shellenv\"" >> ~/.profile
```
3. 通过 linuxbrew 安装 git

```
brew install git
```
4. 检查是否安装成功, 查看 git 版本

```
git version
```

方法三: 源码编译安装

参见: [安装 Git、gitflow 和 gitls](#)

配置 Git

第一次安装 git, 请进行全局配置

用户信息和推送配置

```
# 请更改为你的用户名和邮箱
git config --global user.name "yourname"
git config --global user.email "yourname@domain.com"
```

```
# 配置仅推送当前分支
git config --global push.default simple
```

```
# 配置 gui 文本编码 (Windows 使用)
git config --global gui.encoding utf-8
```

```
# 查看配置
cat ~/.gitconfig
```

代理配置

HTTP代理

完整格式: `git config --global http.https://domain.com.proxy http://proxyUsername:proxyPassword@proxy.server.com:port`

例子:

```
git config --global http.https://git.openearth.community.proxy http://10.192.124.220:80
```

SSH代理

例子:

```
vim ~/.ssh/config
# 添加内容:
Host git.openearth.community
    ProxyCommand connect -H 10.192.124.220:80 %h %p
    ServerAliveInterval 30
```

从 Windows 凭证中移除错误的凭证

Git 远程仓库可以通过 ssh 或者 http(s) 协议下载。当使用 http(s) 协议时, Windows 版本的 git 工具, 会在操作系统的“凭证管理器”中记住 URI 地址和用户录入的用户名、密码。如果更改了用户名和密码, 需要手动到“凭证管理器”中移除。

1. 打开“控制面板 - 凭证管理器”Control Panel\All Control Panel Items\Credential Manager
2. 选择“Windows 凭证”Windows Credentials
3. 从列表中找到 git 的凭证, 删除 Remove 它

当重新用 git push 或者 pull 时, 如果要求用户密码, 会重新出现登录窗口。

开发人员

克隆和检出 develop 分支

克隆远程仓库

当开发一个项目前, 先将远程仓库克隆到本地目录, 会得到一个以仓库名命名的新目录。所有的其他 git 操作, 都应该在这个新目录内使用。这个新目录中会包含一个隐藏的 .git 目录, 保存了这个项目的全部 git 数据。

格式: `git clone [--recursive] 远程仓库地址 [本地路径]`

```
# 在当前目录下克隆 blog 仓库, 然后进入 blog 目录, 用 vscode 打开
# 当前默认分支是 master
# 注意: 不要在 master 分支上修改代码
git clone https://git.openearth.community/weirongbao/blog.git
cd blog
code .
```

检出 develop 分支

```
# 检出远程 develop 分支到本地 develop
# 执行后, 当前分支为 develop
# 注意: 请在 develop 分支进行推送和拉取代码
git pull
git checkout -b develop origin/develop
```

分支操作

创建新分支

为了方便开发, 建议从 develop 分支检出一个新的分支, 如 feature/myfeature。这样做的好处是, 当有其他任务临时中断当前开发, 可以提交变更 feature/myfeature 后, 切换回 develop 分支或者其他分支上, 进行优先级高的工作任务。完成后再次切换回 feature/myfeature 分支继续开发。

```
# 将当前分支检出到新分支 feature/myfeature
git checkout -b feature/myfeature
```

切换分支

```
# 切换到 develop 分支
# 切换前要保证当前分支的工作空间是干净的
# 用 git branch 命令查看本地分支列表, 当前分支会以 *开头
git checkout develop
git branch
```

```
# 切换到 feature/myfeature 分支
git checkout feature/myfeature
git branch
```

查看全部分支

```
# 会显示本地分支以及远程分支列表
git branch -a
```

查看分支关联

```
# 会显示本地分支及其关联的远程分支详情
git branch -vv
```

推送分支到远程仓库

如果需要多人协作开发某个功能，需要将本地分支推到远程仓库。

```
# 将 feature/myfeature 推送到远程仓库
git push -u origin feature/myfeature
```

其他开发人员可参考[检出 develop 分支](#)的方法检出这个功能分支。

```
# 从远程仓库检出 myfeature 功能分支
git checkout -b feature/myfeature origin/feature/myfeature
```

合并功能分支到 develop 分支

方法一：通过 merge 合并代码

当功能开发完成后，应该将代码合并到 develop 分支。

```
# 切换到 develop 分支，注意：要保证工作空间是清洁的才能成功切换分支
git checkout develop
# 合并 feature/myfeature 分支到当前分支（develop分支）
git merge feature/myfeature
```

如果之前修改或者更新过 develop 分支的代码，可能会出现合并冲突，请阅读[解决合并冲突](#)，然后继续。

可以使用下面命令终止或者继续合并操作。

```
git merge --abort
git merge --continue
```

方法二：通过 rebase 合并代码

注意：rebase 方法将重写提交历史，请不要在公共代码中使用此方法。

```
# 假设当前分支是 feature/myfeature
# 以 develop 分支为基础，重新一步步应用当前分支产生的全部提交，此过程称作“衍合”
git rebase develop
# 完成衍合后，检出 develop 分支，然后合并衍合后的 feature/myfeature
git checkout develop
git merge feature/myfeature
```

衍合产生冲突是正常的，可以手动修改后通过下面命令继续，或者终止衍合过程。

```
git rebase --continue | --skip | --abort | --quit | --edit-todo | --show-current-patch
```

更详细方法[请参考官方文档](#)，或执行用 git help rebase 命令。

推送 develop 分支

先拉取再推送。

```
# 拉取远程 develop 分支，合并到本地 develop
git pull develop
```

如果有 develop 发生更新，会显示 vim 编辑器，提示输入提交合并说明，直接输入 zz 或者输入 :wq，保存并退出 vim 即可。

如果出现合并冲突，请阅读[解决合并冲突](#)，然后继续。

推送 develop 到远程仓库。

```
# 如果pull顺利，推送 develop 代码
git push develop
```

注意：任何情况下，都不要使用强制推送命令！

删除分支

成功推送代码后，需要删除不再需要的功能分支。

删除本地分支

```
# 先切换到 develop 分支，然后再删除 feature/myfeature 分支
git checkout develop
git branch -d feature/myfeature
```

删除远程仓库分支

```
# 删除远程仓库 feature/myfeature 分支
git branch -r -d origin/feature/myfeature
git push origin :feature/myfeature
```

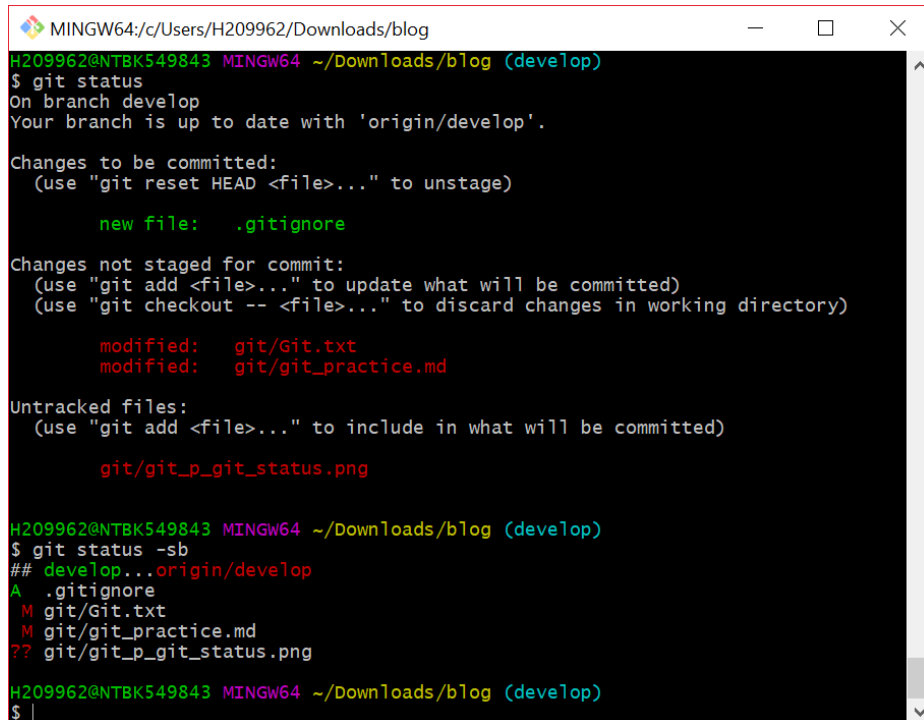
提交代码

检查工作区状态

注意：在编辑代码前，请检查当前所在的分支！

检查工作区状态。

```
# -sb 参数会以简短的方式显示状态信息
git status -sb
```



```
H209962@NTBK549843 MINGW64 ~/Downloads/blog (develop)
$ git status
On branch develop
Your branch is up to date with 'origin/develop'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   .gitignore

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   git/Git.txt
    modified:   git/git_practice.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    git/git_p_git_status.png

H209962@NTBK549843 MINGW64 ~/Downloads/blog (develop)
$ git status -sb
## develop...origin/develop
A .gitignore
M git/Git.txt
M git/git_practice.md
?? git/git_p_git_status.png

H209962@NTBK549843 MINGW64 ~/Downloads/blog (develop)
$
```

图2 git status 与 git status -sb 的区别

简短状态信息标记说明：

- ?? = 未跟踪（可能是需要 git add 命令加入的文件）
- '' = 未修改
- M = 已修改
- A = 已加入
- D = 已删除
- R = 已重命名
- C = 已复制
- U = 已更新但未合并（冲突）

如果发现 U 标记，请阅读[解决合并冲突](#)，然后继续。

提交变更

提示：当提交被推送到远程仓库之前，建议根据实际情况，多次提交变更。

如果有目录结构被修改，如包含状态标记 ??，先用下面命令跟踪更新的目录结构。

```
# 跟踪全部目录结构改变
git add -A
# 检查工作区状态状况
git status -sb
```

提交变更

```
# 如果没有目录结构改变，可以用 -m 代替 -am
git commit -am "add gitignore and png files, update git_practice.md and Git.txt"
```

```
MINGW64:/c/Users/H209962/Downloads/blog
$ git status -sb
## develop...origin/develop
A .gitignore
M git/Git.txt
M git/git_practice.md
?? git/git_p_git_status.png

H209962@NTBK549843 MINGW64 ~/Downloads/blog (develop)
$ git add -A

H209962@NTBK549843 MINGW64 ~/Downloads/blog (develop)
$ git status -sb
## develop...origin/develop
A .gitignore
M git/Git.txt
A git/git_p_git_status.png
M git/git_practice.md

H209962@NTBK549843 MINGW64 ~/Downloads/blog (develop)
$ git commit -am "add gitignore and png files, update git_practice.md and Git.txt"
[develop f2d3c8f] add gitignore and png files, update git_practice.md and Git.txt
4 files changed, 149 insertions(+), 17 deletions(-)
create mode 100644 .gitignore
create mode 100644 git/git_p_git_status.png

H209962@NTBK549843 MINGW64 ~/Downloads/blog (develop)
$ git status -sb
## develop...origin/develop [ahead 1]

H209962@NTBK549843 MINGW64 ~/Downloads/blog (develop)
$
```

图3 提交变更

注释说明有些特殊用法:

- #number: number 可以是 GitLab 的 issue 编号, 比如: fixed #13
- [ci-skip]: 此提交将不会触发 CI (持续集成)
- WIP: 未完成的工作, 在开头加上 WIP (Work in progress) 标记

修正最近一次提交变更的注释

提交变更的注释应该是表达清楚的, 以方便将来追溯提交历史。

```
# 修改上次提交的注释
git commit --amend -m "Add commit command description to git_practice.md"
```

查看提交历史

```
# 查看最近4次提交
git log -n 4
```

格式化的提交历史

```
# 创建别名 glola
alias glola='git log --graph --pretty=\'\'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset\'\'\' --abbrev-c
# 执行 glola
glola
```

```
MINGW64:/c/Users/H209962/Downloads/blog
* e9d4980 - (HEAD -> develop) Add commit command description to git_practice.md (15 minutes ago) <Rongbao WEI>
* 90c140a - (origin/master, origin/develop, origin/HEAD, master) [wip] update (2 hours ago) <Rongbao WEI>
* 2620208 - update (23 hours ago) <Rongbao WEI>
* ec07fad - update use_git_flow.md and clean git_practice.md content (23 hours ago) <Rongbao WEI>
* 7f0eb7c - update (25 hours ago) <Rongbao WEI>
* d14bb6a - update (25 hours ago) <Rongbao WEI>
* 2a2d980 - update (25 hours ago) <Rongbao WEI>
* 0e5fe02 - update (25 hours ago) <Rongbao WEI>
* 497fe42 - update (25 hours ago) <Rongbao WEI>
* 0e93fe9 - add git_practice.md (undone) (25 hours ago) <Rongbao WEI>
* 8914c7e - update dcos.txt (11 days ago) <Rongbao WEI>
* d8526bc - update dcos.txt (12 days ago) <Rongbao WEI>
* ab37dba - add dcos.txt (12 days ago) <Rongbao WEI>
* 7540839 - Update Git.txt (3 weeks ago) <wei rongbao>
* 2d71af4 - Update Git.txt (3 weeks ago) <Rongbao WEI>
* 0dd7aa1 - Update Git.txt (3 weeks ago) <Rongbao WEI>
* b07f9ce - Update save_all_microservice_images.md (4 weeks ago) <wei rongbao>
* 21130c0 - Update save_all_microservice_images.md (4 weeks ago) <wei rongbao>
* d6464dd - Update save_all_microservice_images.md (4 weeks ago) <wei rongbao>
* a056aa1 - update (5 weeks ago) <wei rongbao>
```

图4 格式化的提交日志

如果显示内容超过屏幕，可按下面键操作：

- f 下一屏
- b 上一屏
- j 下一行
- k 上一行
- /add 查找 add
- n 查找下一处
- N 查找上一处
- h 帮助
- q 退出

查看某个文件的提交历史

```
# 查看 git/Git.txt 文件最近4次提交
git log -n 4 -- git/Git.txt
# 查看 git/Git.txt 文件的最近一次提交和修改
git log -n 1 -p -- git/Git.txt
```

解决合并冲突

大多数合并(git merge)操作，git 会自动解决合并冲突，但是，有时候需要手动去解决。

产生冲突后用 git status -sb 查看，会出现如下画面。

```
3 files changed, 52 insertions(+), 3 deletions(-)
create mode 100644 git/git_p_git_add_commit.png
create mode 100644 git/git_p_git_log.png

H209962@NTBK549843 MINGW64 ~/Downloads/blog (develop)
$ git pull
remote: Enumerating objects: 7, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 3), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
From https://git.openeearth.community/weirongbao/blog
 90c140a..74b543f  develop -> origin/develop
Auto-merging git/git_practice.md
CONFLICT (content): Merge conflict in git/git_practice.md
Automatic merge failed; fix conflicts and then commit the result.

H209962@NTBK549843 MINGW64 ~/Downloads/blog (develop|MERGING)
$ git status -sb
## develop...origin/develop [ahead 2, behind 1]
UU git/git_practice.md

H209962@NTBK549843 MINGW64 ~/Downloads/blog (develop|MERGING)
$
```

图5 git 冲突状态

解释：git pull 类似执行了 git fetch 后再执行 git merge，因此 git pull 会导致合并操作。

用文本编辑工具打开冲突文件，搜索 <<<<<< 会找到冲突位置。注意每个文件可能存在多处冲突。

冲突的内容由下面机构组成：

```
<<<<<<< HEAD
合并前的内容（我的）
=====
来自合并的内容（他的）
>>>>>>> 提交编号
```

将这块冲突文本修改正确，记得删除 <<<<<<<、===== 和 >>>>>>> 行。

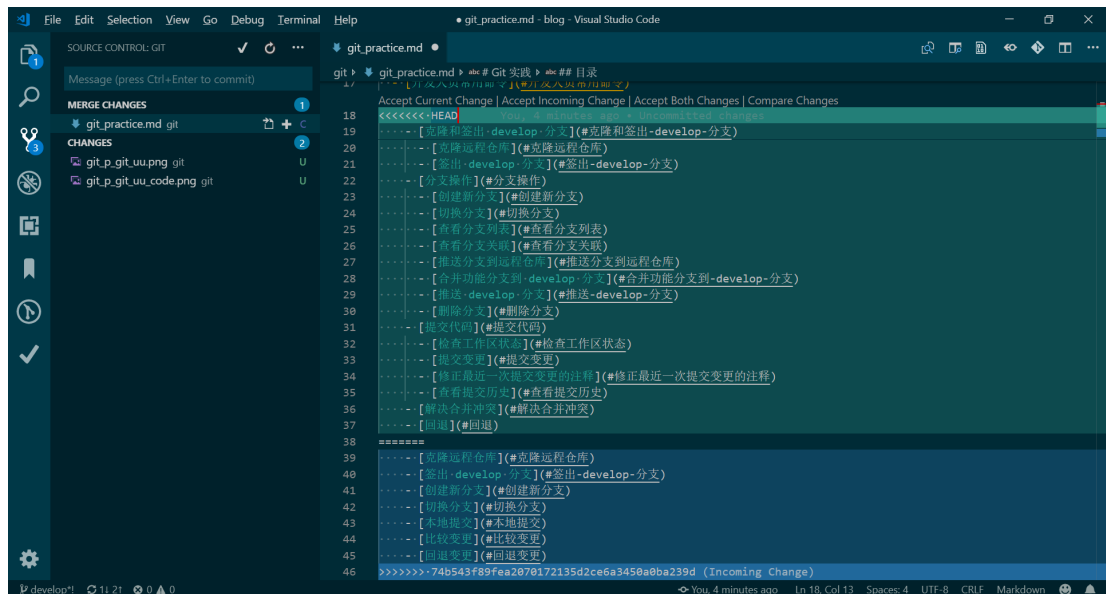


图6 vscode 中显示的冲突内容

然后用 `git add` 命令，跟踪该文件，表示冲突已解决。

```
# 标记解决了 git/git_practice.md 的冲突
git add git/git_practice.md
```

按此方法，处理好全部冲突的文件后，提交变更。

```
git commit -am "resolve conflicts"
```

冲突就解决完了。

高级操作

查看远程仓库地址

```
git remote -v
```

修改远程仓库地址

```
# 修改远程仓库为新的 git 地址
git remote set-url origin <新的 git 地址>
```

贮藏和恢复

如果不想提交变更，又不想抛弃现有的修改，可以利用 `stash` 命令贮藏修改。

```
# 贮藏工作区的修改
git stash

# 恢复贮藏工作，并从贮藏列表中抛弃它
git stash pop

# 显示贮藏列表
git stash list

# 应用 stash@{2} 贮藏
git stash apply stash@{2}

# 删除 stash@{2} 贮藏
git stash drop stash@{2}

# 清空贮藏
git stash clear
```

更多的 `stash` 命令的使用，[请参考官方文档](#) 或者执行 `git help stash` 命令。

比较代码

```
# 比较当前的更改
git diff

# 比较当前与某个历史变更的更改
# 参数可以跟一个历史变更的 id 号
git diff ec07fad
```



```
# 比较任意两个历史版本的更改
# 参数也可以是引用, HEAD指当前提交, HEAD~2指当前提交的爷爷提交
git diff HEAD HEAD~2
```

```
# 可以用 rev-parse 命令解析其变更 id 号
git rev-parse --short HEAD
git rev-parse --short HEAD~2
```

引用说明: HEAD 指当前提交位置, ^ 符号和 ~ 符号的使用, 参见: [“Specifying Revisions”](#), 关系如下:

```

      G   H   I   J
      \   /   \   /
       D   E   F
        \   /   \   \
         \   /     \
          B   C
           \   /
            A

A =          = A^0
B = A^      = A^1      = A~1
C = A^2     = A^2
D = A^^     = A^1^1    = A~2
E = B^2     = A^^2
F = B^3     = A^^3
G = A^^^    = A^1^1^1  = A~3
H = D^2     = B^^2     = A^^^2  = A~2^2
I = F^      = B^3^     = A^^3^
J = F^2     = B^3^2    = A^^3^2
```

放弃对某个文件的修改

注意: 请小心使用下面命令! 会覆盖当前文件的修改!

```
# 放弃对 git/Git.txt 文件的修改
git checkout -- git/Git.txt
```

从提交历史中获取文件

注意: 请小心使用下面命令! 会覆盖同名文件!

```
# 用 ec07fad 提交的 git/Git.txt 代替现在的
git checkout ec07fad git/Git.txt
```

放弃全部修改

注意: 请小心使用下面命令! 全部修改将丢失!

```
# 放弃修改, 注意只针对被跟踪了的文件和目录
git reset HEAD --hard

# 移除未被跟踪的文件, 可以用 -xf 参数代替 -df, 忽略.gitignore
git clean -df
```

说明: .gitignore 文件用于排除跟踪文件或目录, 参考[添加.gitignore文件](#)。

重置到某次提交

注意: 请谨慎使用下面命令! 不可恢复!

注意: git reset --hard 方法将重写提交历史, 请不要在公共代码中使用此方法。

```
# 重置到 ec07fad, 在 ec07fad 后的提交变更将全部丢失!
git reset ec07fad --hard
# 查看最近4次提交日志
git log -n 4
```

检出某次提交

有时候需要看下某个历史版本的效果, 可以检出某个提交到一个新分支上。

注意: 在这个 temp 分支仅用来测试, 不要提交代码。

```
# 将 ec07fad 检出到新分支 temp
# 检出前工作空间应该是干净的, 并且temp分支不存在
# 当前分支 temp
git checkout -b temp ec07fad
```

测试完成后, 请删除 temp 分支, 参见[删除分支](#)。

维护人员

创建新项目

在 GitLab 中创建一个项目

在新建项目页面, 选择 Blank Project, 输入下面内容:

- Project Name: 项目名, 英文+数字+下划线, 英文开头
- Project URL: 可以从下拉框中选择一个组名, 默认是个人名称 (个人项目默认最多10个)
- Project description: 建议写一个项目的简短说明
- Initialize repository with a README: 建议不勾选

点 Create Project 按钮创建项目。

New Project - GitLab

https://git.openearth.community/projects/new

GitLab Projects Groups Activity Milestones Snippets

(repository), plan your work (issues), and publish your documentation (wiki), among other things.

All features are enabled for blank projects, from templates, or when importing, but you can disable them afterward in the project settings.

Tip: You can also create a project from the command line. [Show command](#)

Project name: demo

Project URL: https://git.openearth.community/weirongbao

Project slug: demo

Project description (optional): A demo project

Visibility Level: Private (selected)

Private: Project access must be granted explicitly to each user.

Internal: Internal visibility has been restricted by the administrator.

Public: Public visibility has been restricted by the administrator.

☐ Initialize repository with a README

Allows you to immediately clone this project's repository. Skip this if you plan to push up an existing repository.

Create project Cancel

图7 GitLab 新建项目页面

克隆到本地

在项目首页，获得 `git` 地址，然后克隆到本地。

```
git clone https://git.openearth.community/weirongbao/demo.git
cd demo
```

```
MINGW64:/c:/Users/H209962/Downloads/demo
H209962@NTBK549843 MINGW64 ~/Downloads
$ git clone https://git.openearth.community/weirongbao/demo.git
Cloning into 'demo'...
warning: You appear to have cloned an empty repository.
H209962@NTBK549843 MINGW64 ~/Downloads
$ cd demo
H209962@NTBK549843 MINGW64 ~/Downloads/demo (master)
$
```

图8 克隆新项目仓库

添加代码文件

如果已经有代码文件，将全部代码文件复制到此文件夹内。

Windows 用户可以用 `explorer .` 命令，用资源管理器打开当前项目文件夹。

添加 README.md 文件

README.md 文件会显示在 GitLab 项目页面

```
# 创建 README.md 文件
touch README.md
# 用 vscode 打开当前文件夹，然后编辑 README.md
code .
```

建议格式如下：

```
# Project Name
```

```
description
```

```
## Title 1
```

```
Content
```

添加 .gitignore 文件

.gitignore 文件用来忽略 `git` 对非代码文件的跟踪和提交。你需要根据不同的项目输入不同的内容。

```
# 创建 .gitignore 文件
touch .gitignore
# 用 vscode 打开当前文件夹，然后编辑 .gitignore
code .
```

参考模板如下：

- Java 项目

```
# Linux
*~
.directory
.Trash-*

# Windows
Thumbs.db
ehthumbs.db
Desktop.ini
$RECYCLE.BIN/

# JetBrains
*.iml
.idea/
*.ipr
*.iws
/out/
.idea_modules/
atlassian-ide-plugin.xml
com_crashlytics_export_strings.xml
crashlytics.properties
crashlytics-build.properties

# Eclipse
*.pydevproject
.metadata
.gradle
bin/
tmp/
*.tmp
*.bak
*.swp
*~.nib
local.properties
.settings/
.loadpath
.project
.externalToolBuilders/
*.launch
.cproject
.classpath
.factorypath
.buildpath
.target
.texlipse

# Maven
target/
pom.xml.tag
pom.xml.releaseBackup
pom.xml.versionsBackup
pom.xml.next
release.properties
dependency-reduced-pom.xml
buildNumber.properties
.mvn/timing.properties

# Java
*.class
*.mtj.tmp/
*.jar
*.war
*.ear
hs_err_pid*
```

- Node 项目

```
# Linux
*~
.directory
.Trash-*

# Windows
Thumbs.db
ehthumbs.db
Desktop.ini
$RECYCLE.BIN/

# VSCode
.settings

# Node
logs
*.log
npm-debug.log*
```

```
pids
*.pid
*.seed
lib-cov
coverage
.grunt
.lock-wscript
build/Release
node_modules
```

• C 项目

```
# Object files
*.o
*.ko
*.obj
*.elf

# Precompiled Headers
*.gch
*.pch

# Libraries
*.lib
*.a
*.la
*.lo

# Shared objects (inc. Windows DLLs)
*.dll
*.so
*.so.*
*.dylib

# Executables
*.exe
*.out
*.app
*.i*86
*.x86_64
*.hex

# Debug files
*.dSYM/
```

• C++ 项目

```
# Compiled Object files
*.slo
*.lo
*.o
*.obj

# Precompiled Headers
*.gch
*.pch

# Compiled Dynamic libraries
*.so
*.dylib
*.dll

# Fortran module files
*.mod

# Compiled Static libraries
*.lai
*.la
*.a
*.lib

# Executables
*.exe
*.out
*.app
```

• C# 项目

```
[Bb]in/
[Oo]bj/
TestResults
*.suo
*.user
*.sln.docstates
[Dd]ebug/
[Rr]elease/
x64/
*_i.c
*_p.c
*.ilk
*.meta
*.obj
*.pch
*.pdb
*.pgc
*.pgd
*.rsp
*.sbr
*.tlb
*.tli
*.tlh
*.tmp
*.log
*.vspssc
*.vsssc
.builds
ipch/
*.aps
```

```

*.ncb
*.opensdf
*.sdf
*.psess
*.vsp
*.vspx
*.gpState
ReSharper*
*.ncrunch*
.*crunch*.local.xml
[Be]xpress
DocProject/buildhelp/
DocProject/Help/*.HxT
DocProject/Help/*.HxC
DocProject/Help/*.hhc
DocProject/Help/*.hhk
DocProject/Help/*.hhp
DocProject/Help/Html2
DocProject/Help/html
publish
*.Publish.xml
packages
csx
*.build.csdef
AppPackages/
[Bb]in
[Oo]bj
sql
TestResults
[Tt]est[Rr]esult*
*.Cache
ClientBin
[Ss]tyle[Cc]op.*
~$*
*.dbmdl
Generated Code
UpgradeReport_Files/
Backup*/
UpgradeLog*.XML

```

- 更多模板，请参考 [gogs-gitignore](#)。

忽略已被跟踪的文件或目录

有时候会发现需要忽略掉已经被跟踪的文件，但是它最初并没有标记在 `.gitignore` 文件里。

```

# 从跟踪缓存里，递归移除 logs 目录索引
# 如果移除一个文件索引，不需要 -r 参数
git rm --cached -r logs/
# 将 logs/ 追加到 .gitignore 文件里
# 当然也可以直接用文本编辑工具打开 .gitignore 文件，添加一行 logs/
echo "logs/" >> .gitignore
# 提交变更
git commit -m "We really don't want Git to track this anymore!"

```

Git LFS

LFS (Large File Storage) 是将大文件存储在远程服务器，而 `git` 的变更历史中只保留引用，这样能有效地减少 `git` 仓库。

GitLab 8.2 开始支持 LFS 功能，创建项目后，需要手动在 Settings -> General -> Permissions 中开启 Git Large File Storage。

然后参考下面命令添加某些文件为 LFS

```

# 将 iso 文件添加到 LFS
git lfs track "*.iso"
# 假设添加一个 iso 文件
git add ubuntu.iso
# 加入 lfs 配置文件
git add .gitattributes
# 提交变更
git commit -am "lfs track iso files"
# 查看已加入到 lfs 的文件
git lfs ls-files

```

在 GitLab 10.5 开始支持 LFS 文件锁功能，例如：

```

# 对 png 文件配置为可加锁
git lfs track "*.png" --lockable
# 锁定 images/banner.png 文件
git lfs lock images/banner.png
# 查看已加锁的文件
git lfs locks
# 对 images/banner.png 解锁
# 也可以用上一条命令得到的 id 解锁，如: git lfs unlock --id=123
# 可以加入 --force 强制解锁
git lfs unlock images/banner.png
# 查看失败的日志
git lfs logs last

```

更多 LFS 的使用，参考[官方文档](#)

提交并推送 master 分支

```

git add -A
git commit -am "first commit"
git push -u origin master

```

创建并推送 develop 分支

```

git checkout -b develop
git push -u origin develop

```

备注：维护人员也可以在 GitLab 项目页面上，通过 + 号按钮新建分支。

配置 GitLab 项目权限

1. 打开 GitLab 的项目页面
2. 对分支和版本 tag 进行保护（必须）。进入 “Settings - Repository” 页面
 - 展开 “Protected Branches”，可以看见 master 分支保护已添加，继续添加 develop 分支，然后设置这两个分支的保护内容：Allowed to merge: Developers + Maintainers, Allowed to push: No one。
 - 展开 “Protected Tags”，创建 Tag: v*, Allowed to create: Maintainers，点 “Protect” 按钮创建
3. 配置成员和权限（必须）。进入 “Settings - Members” 页面
 - Invite member: 共享项目给其他人。可以邀请某些人，并且设置每个人权限是 Maintainer（维护人员）或者 Developer（开发人员）或者其他。
 - Invite group: 共享项目给其他组。可以邀请某些组，并且设置组成员的最高级别的权限，至少是 Developer 才能参与开发。
4. 合并请求批准（可选），进入 “Settings - Merge request”，在 “Merge request approvals” 中添加审核人，“Approvals required” 设置要求批准人数，可以是 0，可以勾选 “Enable self approval of merge requests”，最后点 “Save changes” 按钮。
5. 配合 “Jenkins” 等第三方服务（可选），可以在 “Settings - Integrations” 中配置 “WebHook”。
6. 开启 “LFS (Large File Storage)” 功能（可选），可以在 “Settings - General” 页面的 “Permissions” 中启用 “Git Large File Storage”，关于 git lfs 的使用，参见[Git LFS](#)。

打 tag

维护人员应该在 master 分支，对释放的代码打版本 tag。版本 tag 以 v 开始，格式如：v1.0.0，关于版本说明[请参考这里](#)。

备注：维护人员也可以在 GitLab 项目页面上，通过 + 号按钮添加 tag。

添加 tag

```
# 检出 master 分支，并拉取最新的代码，假设这是释放代码
git checkout master
git pull

# 打 v1.0.0 tag，注意 tag 名称不要重复
# -m 参数可选，对 tag 添加说明
# 可以加入 -s 参数，对 tag 进行 pgp 签名
git tag v1.0.0 -m 'version 1.0.0'

# 查看版本 tag
git tag -l 'v*'
```

推送 tag

```
# 将 tag 推到远程仓库
git push origin v1.0.0
```

查看 tag 列表

```
# 查看全部 tag
git tag -l
```

删除 tag

注意：请不要删除已被推送到远程仓库，已作为释放标记的版本 tag。

```
# 删除名称为 deltag 的 tag
git tag -d deltag
# 删除远程仓库名称为 deltag 的 tag
git push origin :refs/tags/deltag
```

找到错误的提交

有时候需要从提交历史中找到某个功能代码运行正常的一次提交。git 提供了二分搜索(Binary Search)功能，从提交历史中定位问题。

```
# 假设提交历史: ... --- 0 --- 1 --- 2 --- 3 --- 4* --- 5 --- current
# 先贮藏当前工作
git stash

# 开始二分搜索
git bisect start

# 标记 current 提交是错误的
git bisect bad

# 标记 0 提交是正确的, git 将会在 0...current 之间二分搜索
git bisect good 0

# 此时, git 会检出 3 提交
# 测试这个提交的代码是否正确, 需要配合你的编译命令, 这里假设使用的 make
make
make test

# 如果判定了这次代码是正确的, 请输入
git bisect good

# 此时, git 会检出 5 提交, 依旧验证其是否正确
make
make test

# 假设测试未通过, 此代码是出错的代码
git bisect bad

# 此时, git 定位到 4 提交 (在最近一次被标记为 good 和 bad 中间的提交)
# 继续验证是否正确
make
make test

# 如果验证失败
git bisect bad

# 最终, 我们找到了错误的代码是来自 4 提交
# 然后通过下面命令, 结束 bisect 任务
git bisect reset

# 恢复贮藏的工作
git stash pop
```

回滚到某次提交

建议开发人员使用[从提交历史中获取文件](#)的方法, 对自己的文件用历史文件逐个替换, 而不是采用回滚的方法。

回滚有多种方法, 对于公共代码, 请使用下面命令。

```
# 假设在 develop 分支上回退到某次 commit
git checkout develop
# 回滚到某个历史 commit
git reset --hard <commit>
git reset --soft HEAD@{1}
git commit -m "Reverting to the state of the project at <commit>"
```

统计贡献情况

```
# 按贡献者名字统计提交数
git shortlog -sn
# 按贡献者名字和邮箱统计提交数
git shortlog -sne
```

Git 图形化工具

SourceTree 工具

[SourceTree](#) 是支持 macOS 和 Windows 的免费图形化 git 工具。Windows 版本下载请进入主页, 点击 [Also available for Windows](#) 下载。

安装前需要需要免费注册一个 [Atlassian 账户](#)。

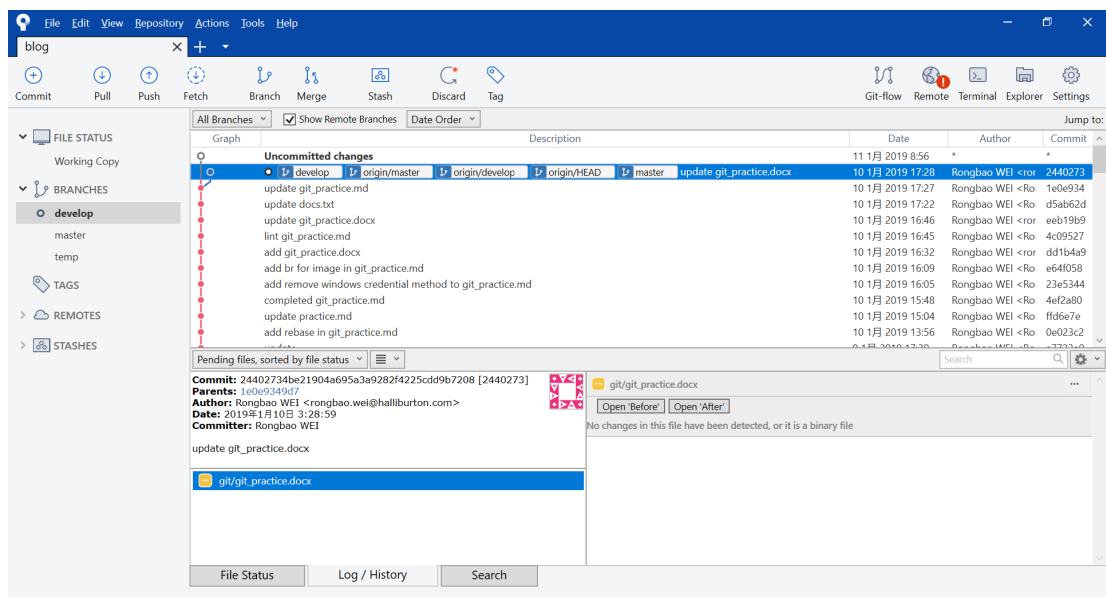


图9 SourceTree for Windows

VSCode 的 GitLens 扩展

打开 VSCode，按 Ctrl+Shift+X 打开“扩展”窗格，在搜索栏里输入 GitLens，点 Install 按钮，安装成功后点 Reload 按钮重启 VSCode。在左侧的会多出一个 GitLens 窗格，点开如下图。

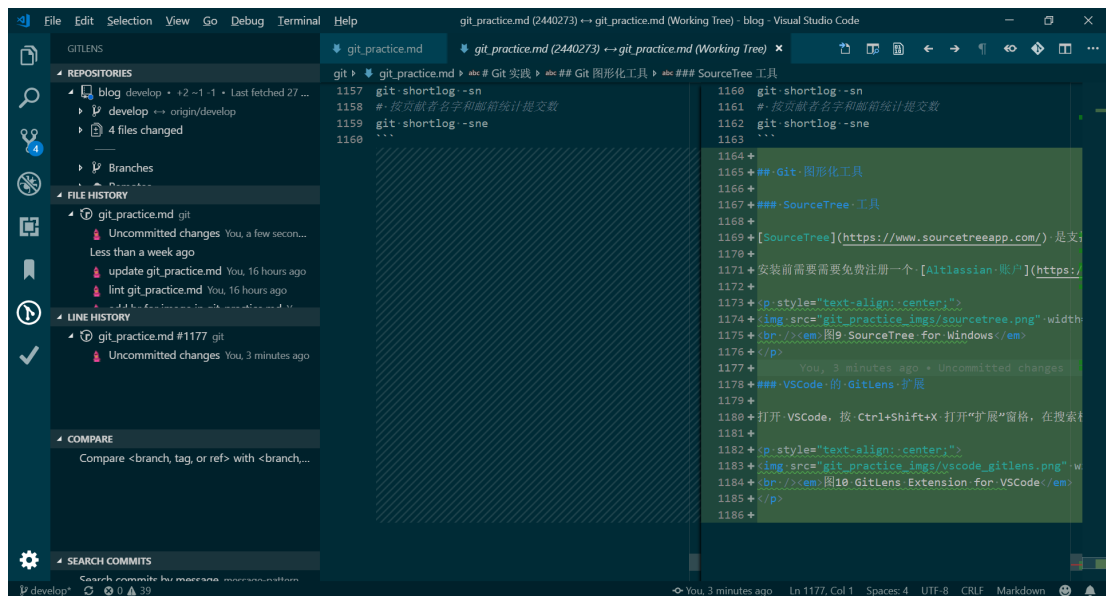


图10 GitLens Extension for VSCode

Git 工具自带的图形工具

在项目目录中直接运行
gitk

Windows 用户还可以运行 git gui 打开另一个界面
git gui

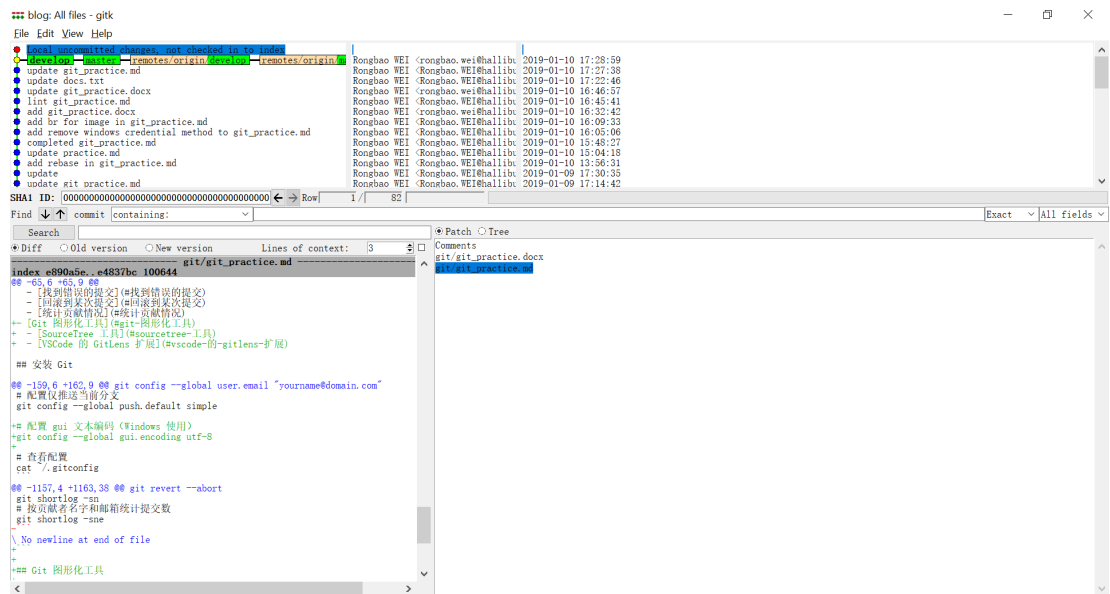


图11 Gitk in Windows