Git 基本概念

Git 有三个主要的工作区域:

1. 工作区（Working Directory）

- 就是你在电脑里能看到的目录
- 你直接编辑的文件都在工作区
- 对文件的修改首先发生在工作区

2. 暂存区（Staging Area / Index）

- 是一个临时存储区域
- 用于存放即将提交到版本库的修改
- 使用 `git add` 命令将工作区的修改添加到暂存区
- 可以选择性地添加文件，灵活控制每次提交的内容

3. 版本库（Repository）

- 是 Git 用来保存项目元数据和对象数据库的地方
- 使用 `git commit` 命令将暂存区的内容提交到版本库
- 提交后会生成一个永久的快照

工作流程：

```
工作区 --git add--> 暂存区 --git commit--> 版本库
```

Git 常用命令

配置相关

```
# 设置用户名和邮箱
git config --global user.name "你的名字"
git config --global user.email "你的邮箱"

# 查看配置信息
git config --list
```

仓库初始化

```
# 初始化本地仓库
# ⚠️ 谨慎使用：确保在正确的目录下执行
git init

# 克隆远程仓库
git clone <仓库地址>

# 克隆指定分支
git clone -b <分支名> <仓库地址>
```

基本操作

```
# 查看状态
git status

# 查看简洁状态
git status -s

# 添加文件到暂存区
git add <文件名>
git add .                  # 添加所有文件
git add *.js               # 添加所有js文件

# 提交更改
git commit -m "提交说明"

# 添加并提交（跳过暂存区）
git commit -am "提交说明"

# 修改最后一次提交
git commit --amend

# 取消跟踪文件（保留本地文件）
git rm --cached <文件名>

# 取消跟踪文件夹（保留本地文件夹）
git rm --cached -r <文件夹名>

# 取消跟踪并删除文件
git rm <文件名>
```

查看历史

```
# 查看提交历史
git log

# 查看简洁历史
git log --oneline

# 查看图形化历史
git log --graph --oneline --all

# 查看最近n条记录
git log -n <数量>

# 查看某个文件的历史
git log <文件名>

# 查看详细改动
git log -p
```

分支操作

```
# 查看所有分支
git branch -a

# 查看本地分支
git branch

# 创建新分支
git branch <分支名>

# 切换分支
git checkout <分支名>

# 创建并切换到新分支
git checkout -b <分支名>

# 使用switch命令切换分支（推荐）
git switch <分支名>
git switch -c <分支名>    # 创建并切换

# 合并分支
git merge <分支名>

# 删除本地分支
git branch -d <分支名>

# 强制删除分支
git branch -D <分支名>

# 重命名分支
git branch -m <旧名称> <新名称>
```

远程仓库

```
# 查看远程仓库
git remote -v

# 添加远程仓库
git remote add origin <仓库地址>

# 删除远程仓库
git remote remove <远程名>

# 重命名远程仓库
git remote rename <旧名称> <新名称>

# 拉取远程更新
```

```
git fetch origin

# 拉取并合并
git pull origin <分支名>

# 推送到远程
git push origin <分支名>

# 推送所有分支
git push --all origin

# 推送标签
git push origin <标签名>
git push --tags              # 推送所有标签

# 删除远程分支
git push origin --delete <分支名>

# 设置上游分支
git push -u origin <分支名>
```

撤销操作

```
# 撤销工作区的修改
git checkout -- <文件名>
git restore <文件名>          # 推荐使用

# 撤销暂存区的文件
git reset HEAD <文件名>
git restore --staged <文件名>  # 推荐使用

# 回退到上一个版本
git reset --soft HEAD^        # 只撤销commit，保留修改在暂存区（已add状态）
git reset --mixed HEAD^       # 撤销commit和add，保留修改在工作区（未add状态）
git reset --hard HEAD^        # ⚠️ 撤销所有修改，恢复到上一版本（危险操作）

# 回退到指定版本
git reset --hard <commit-id>

# 查看操作历史（用于恢复误删的提交）
git reflog

# 撤销某次提交（创建新提交）
git revert <commit-id>

# reset 和 revert 的区别：
# reset：时光倒流，直接回到过去某个版本，中间的提交历史会消失
#         适合：本地还没推送的提交
# revert：撤销操作，通过新增一个"反向提交"来抵消之前的修改，历史记录完整保留
#         适合：已经推送到远程的提交
```

差异比较

```
# 查看工作区与暂存区的差异
git diff

# 查看暂存区与最新提交的差异
git diff --cached
git diff --staged

# 查看工作区与最新提交的差异
git diff HEAD

# 查看两个提交之间的差异
git diff <commit-id1> <commit-id2>

# 查看某个文件的差异
git diff <文件名>

# 查看分支间的差异
git diff <分支1> <分支2>
```

暂存操作（Stash）

> **注意：** 这里的"暂存操作"和前面提到的"暂存区"是两个完全不同的概念！
>
> - **暂存区（Staging Area）**：git add 后文件存放的地方，是提交前的准备区域
> - **Stash（临时存储）**：临时保存工作进度的独立存储栈，与暂存区无关

什么是 stash？ stash 是一个临时存储区，用来保存当前工作区和暂存区的修改，然后恢复到干净的工作状态。

使用场景：

- 正在开发功能，突然需要切换分支处理紧急问题
- 想拉取远程更新，但本地有未提交的修改
- 临时保存工作进度，但还不想提交

工作原理：

```
当前修改 --git stash--> 临时保存 --git stash pop--> 恢复修改
```

```
# 暂存当前修改
git stash

# 暂存时添加说明
git stash save "说明信息"

# 查看暂存列表
git stash list

# 恢复最近的暂存
git stash pop

# 恢复指定的暂存
git stash apply stash@{n}

# 删除暂存
git stash drop stash@{n}

# 清空所有暂存
git stash clear
```

标签管理（Tag）

什么是标签？ 标签是指向特定提交的固定引用，相当于给某个提交起了一个容易记住的名字。

使用场景：

- 标记项目的发布版本（如 v1.0.0、v2.0.0）
- 标记重要的里程碑节点
- 方便快速回到某个重要的历史版本

两种标签类型：

- **轻量标签**：只是一个简单的引用，类似书签
- **附注标签**：包含标签信息、打标签者、日期等完整信息（推荐使用）

```
# 查看所有标签
git tag

# 创建轻量标签
git tag <标签名>

# 创建附注标签
git tag -a <标签名> -m "标签说明"
```

```
# 给指定提交打标签
git tag -a <标签名> <commit-id>

# 查看标签信息
git show <标签名>

# 删除本地标签
git tag -d <标签名>

# 删除远程标签
git push origin :refs/tags/<标签名>
```

实际使用示例：

```
# 场景：项目开发完成，准备发布 1.0.0 版本

# 1. 确保代码已提交
git add .
git commit -m "完成 1.0.0 版本开发"

# 2. 创建标签
git tag -a v1.0.0 -m "发布 1.0.0 版本"

# 3. 推送标签到远程
git push origin v1.0.0

# 4. 以后需要回到这个版本
git checkout v1.0.0
```

其他常用命令

```
# 查看某个文件的每一行是谁修改的
git blame <文件名>

# 清理未跟踪的文件
git clean -n                # 预览将要删除的文件（安全，不会真的删除）
git clean -f                # 删除未跟踪的文件
git clean -fd               # 删除未跟踪的文件和目录

# 推荐使用流程：先预览，确认后再删除
# 1. git clean -n    查看哪些文件会被删除
# 2. git clean -f    确认无误后执行删除

# 变基操作（Rebase）
# 作用：将当前分支的提交"移动"到目标分支的最新提交之后，形成线性历史
git rebase <分支名>

# 交互式变基：可以修改、合并、删除提交
```

```
git rebase -i HEAD~3       # 编辑最近3次提交
git rebase -i HEAD~n       # 编辑最近n次提交

# ⚠️ 重要提示：不要对已推送到远程的提交进行 rebase，会导致历史冲突

# 交互式变基的操作说明：
# 执行 git rebase -i 后，会打开编辑器显示提交列表，每行一个提交
# 可以使用以下命令对提交进行操作：
#   pick   - 保留该提交（默认）
#   reword - 保留提交，但修改提交信息
#   edit   - 保留提交，但暂停以便修改内容
#   squash - 将该提交合并到前一个提交，会打开编辑器让你编辑合并后的提交信息
#   fixup  - 将该提交合并到前一个提交，只保留前一个提交的信息（丢弃当前提交信息）
#   drop   - 删除该提交
#
# 示例：执行 git rebase -i HEAD~3 后会看到类似内容：
#   pick abc123 添加登录功能
#   pick def456 修复登录bug
#   pick ghi789 优化登录性能
#
# 修改为（合并后两个提交）：
#   pick abc123 添加登录功能
#   squash def456 修复登录bug
#   squash ghi789 优化登录性能
#
# 执行过程：
#   1. 保存退出后，Git 会将三个提交合并成一个
#   2. 自动打开编辑器，显示所有提交信息供你编辑：
#      添加登录功能
#
#      修复登录bug
#
#      优化登录性能
#   3. 你可以编辑成想要的格式，保存后完成合并
#   4. 最终结果：1个提交，包含所有修改和你编辑后的提交信息


# 查看文件的修改记录
git show <commit-id>

# 搜索提交信息
git log --grep="关键词"

# 搜索代码内容
git log -S "代码内容"

# 忽略文件权限变化
# 作用：让 Git 不跟踪文件的可执行权限变化（chmod +x/-x）
# 使用场景：跨平台开发（Windows/Linux/Mac）时，避免权限变化导致的误提交
git config core.fileMode false
```

.gitignore 文件

```
# .gitignore 文件用于忽略不需要版本控制的文件
# 常见配置示例：

# 忽略所有 .log 文件
*.log

# 忽略 node_modules 目录
node_modules/

# 忽略所有 .env 文件
.env
.env.local

# 忽略 IDE 配置
.vscode/
.idea/

# 忽略编译输出
dist/
build/
```