目录

1 基础操作

1.1 操作概览

表 1 Git 常用命令一览

操作	命令示例	说明
初始化仓库	`git init`	初始化本地仓库
克隆仓库	`git clone <仓库地址>`	克隆远程仓库到本地
查看状态	`git status`	查看当前仓库状态
添加到暂存区	`git add <文件名>`	添加文件到暂存区
提交更改	`git commit -m "提交说明"`	提交暂存区内容
查看历史	`git log`	查看提交历史
查看 ref 历史	`git reflog`	查看 HEAD 的所有移动痕迹
推送到远程	`git push`	推送本地提交到远程仓库
拉取更新	`git pull`	拉取并合并远程更新
创建分支	`git branch <分支名>`	创建新分支
切换分支	`git checkout <分支名>`	切换到指定分支
查看帮助	`git --help` 或 `git help <command>`	显示对应命令的帮助
查看远程	`git remote`	查看远程信息

1.2 详细说明

1.2.1 `git log` 和 `git reflog`

- `git log`：查看 *项目提交历史*。它显示的是代码库的 *演进历史*，是你有意识创建的提交记录（commit history）。
- `git reflog`：查看 *引用日志*。它记录的是 *本地仓库中 HEAD 指针和分支指针的移动历史*，是你（或 Git 命令）在本地仓库中 *执行操作的踪迹*。

1.2.2 `git push` 命令格式

推送命令的几种格式：

```
git push <远程主机名> <本地分支名>:<远程分支名>  // 标准格式
git push <远程主机名> <本地分支名>               // 推送到同名分支
git push <远程主机名> :<远程分支名>              // 删除远程分支（不推荐）
git push origin --delete <远程分支名>           // 删除远程分支（推荐）
```

> **注意**
>
> - 第一行是标准写法，明确指定本地和远程分支
> - 第二行是简化写法，推送到*同名远程分支（不存在则创建）*
> - 最后两行用于删除远程分支，推荐使用 `--delete` 的写法

1.2.3 `git pull` 命令格式

```
git pull <远程主机名> <远程分支名>:<本地分支名>    // 标准格式
git pull <远程主机名> <远程分支名>                  // 远程分支与当前分支合并
```

实际上是 `git fecth` 和 `git merge` 的简写。

> **注意**
>
> 当 当前分支 和对应远程主机的某一个分支建立了 `upstream` 关系(追踪关系)，也可以省略远程分支。
>
> 如果当前分支只有一个追踪分支，连远程主机名都可以省略。

> **注意**
>
> 如果不想使用 `git merge` 来合并，而是使用 `git rebase` ，可以使用 `git pull --rebase` 操作。

1.2.4 `git checkout` 命令格式

```
git checkout [-b] <本地分支名>     // 切换到某一个本地分支[-b:不存在则创建]
git checkout <commit-hash>         // 将 HEAD 指向对应的 commit 而非一个分支的尖端
```

> **提示**
>
> `git checkout <commit-hash>` 往往用于基于特定点来新建一个分支。流程如下:
>
> - `git checkout <tag-name>` (这会进入分离头指针状态，因为标签通常不是分支)。
> - `git switch -c <new-branch-name>` (基于当前的分离头指针状态创建新分支并切换到它)。

1.3 演示 1

执行 `git push origin main:main` :

```
➜  Git-Example git:(main) : git push origin main:main

Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 16 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 769 bytes | 769.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/gan-rui-lin/Git-Example.git
   2964a06..fb7803e  main -> main
```

执行 `git branch`:

```
➜  Git-Example git:(main)   git branch
* main
```

执行 `git checkout -b new_branch`:

```
➜  Git-Example git:(main) git checkout -b new_branch
Switched to a new branch 'new_branch'
➜  Git-Example git:(new_branch)
```

来到 new_branch 之后，继续执行 `git push origin new_branch`,将当前分支推到远程的同名分支上去：

```
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
remote:
remote: Create a pull request for 'new_branch' on GitHub by visiting:
remote:      https://github.com/gan-rui-lin/Git-Example/pull/new/new_branch
remote:
To https://github.com/gan-rui-lin/Git-Example.git
 * [new branch]      new_branch -> new_branch
```

最后执行 `git push origin --delete new_branch` 删除这一个新分支:

```
➜  Git-Example git:(new_branch)   git push origin --delete new_branch
To https://github.com/gan-rui-lin/Git-Example.git
 - [deleted]         new_branch
```

执行 `git branch -a`:

```
  main
* new_branch
  remotes/origin/main
```

可以看到远程的 new_branch 分支被删除了。

切换回 main 分支并做一些改动并提交(尚未推送到远程), 执行 `git log`:

```
 Git-Example git:(main)   git log
commit 3e6cca8050f6409f5b9bad5d4b9849bcebffdf49 (HEAD -> main)
Author:
Date:   Thu Aug 21 10:21:11 2025 +0800

    git push/pull example

commit fb7803e0d23e9c3fa6e78eaadc96b159a3f1e293 (origin/main, new_branch)
Author:
```

```
Date:   Thu Aug 21 10:00:42 2025 +0800

    init: typ style

commit 2964a06ec2a6428fa9348b317534047a553ebdbd
Author:
Date:   Wed Aug 20 18:00:43 2025 +0800

    add license
```

可以看到当前的 `HEAD` 、本地 `main` 分支和 `new_branch` 分支、`origin` 远程的 `main` 分支所指向的 commit-hash 值。

执行 `git checkout fb7803e0d23e9c` , 这个 hash 值是第二次提交的 hash 值:

此时再执行 `git reflog | tail -n 2` 命令:

```
fb7803e (HEAD, origin/main, new_branch) HEAD@{0}: checkout: moving from main
to fb7803e0d23e9c
3e6cca8 (main) HEAD@{1}: commit: git push/pull example
```

可以发现 HEAD 指向了 `fb7803e` 的这个游离分支上。并且 `HEAD@{n}` 的 `n` 告诉了这是 HEAD 指针的第 n 次移动。

最后执行 `git checkout main` 切换回本地 `main` 分支并提交更改，推送到对应远程。

2 进阶操作

2.1 操作概览

表 2 Git 进阶命令一览

操作	命令示例	说明
暂存更改	`git stash`	暂时保存当前工作目录的更改
应用暂存	`git stash pop`	应用并删除最近的暂存
查看暂存	`git stash list`	查看所有暂存记录
变基合并	`git rebase <分支名>`	将当前分支变基到指定分支
交互式变基	`git rebase -i <commit>`	交互式修改提交历史
撤销提交	`git revert <commit>`	创建新提交来撤销指定提交
挑选提交	`git cherry-pick <commit>`	将指定提交应用到当前分支
重置提交	`git reset --hard <commit>`	强制重置到指定提交
修改提交	`git commit --amend`	修改最近一次提交
查看差异	`git diff <commit1> <commit2>`	比较两个提交的差异
标签管理	`git tag <标签名>`	为当前提交创建标签
子模块	`git submodule add <仓库>`	添加子模块

2.2 详细说明

2.2.1 `git stash` 暂存操作

暂存功能允许你临时保存当前的工作状态，而不需要创建提交：

```
git stash                      // 暂存当前更改
git stash push -m "描述信息"    // 带描述的暂存
git stash list                 // 查看暂存列表
git stash show stash@{0}       // 查看指定暂存的内容
git stash pop                  // 应用并删除最近的暂存
git stash apply stash@{0}      // 应用指定暂存但不删除
git stash drop stash@{0}       // 删除指定暂存
git stash clear                // 清空所有暂存
```

> 提示
>
> `git stash` 只会暂存已跟踪的文件。如果你有新文件需要暂存，使用 `git stash -u` 包含未跟踪的文件。

2.2.2 `git rebase` 变基操作

变基是一种强大的历史重写工具，可以保持提交历史的线性：

```
git rebase <目标分支>          // 将当前分支变基到目标分支
git rebase -i HEAD~3           // 交互式变基最近3个提交
git rebase --continue          // 解决冲突后继续变基
git rebase --abort             // 中止变基操作
git rebase --onto A B C        // 将C分支从B开始的提交变基到A
```

> 重要
>
> `git rebase` 的黄金法则是 **永远不要在公共分支上使用它**。

2.2.3 `git revert` 撤销操作

`revert` 通过创建新提交来撤销之前的更改，是一种安全的撤销方式：

```
git revert <commit-hash>       // 撤销指定提交
git revert HEAD                // 撤销最近一次提交
git revert HEAD~3..HEAD        // 撤销最近3个提交
git revert -n <commit>         // 撤销但不自动提交
```

> 提示
>
> 与 `git reset` 不同，`git revert` 不会删除历史记录，而是创建新的撤销提交，适合在共享分支上使用。

2.2.4 `git cherry-pick` 挑选提交

挑选特定的提交应用到当前分支：

```
git cherry-pick <commit-hash>     // 挑选单个提交
git cherry-pick A..B              // 挑选A到B之间的提交
git cherry-pick -x <commit>       // 挑选并在提交信息中记录原始提交
git cherry-pick --no-commit <commit>  // 挑选但不自动提交
```

2.2.5 `git reset` 重置操作

```
git reset --soft HEAD~1       // 软重置：保留更改在暂存区
git reset --mixed HEAD~1      // 混合重置：保留更改在工作目录
git reset --hard HEAD~1       // 硬重置：完全删除更改
```

2.2.6 `git diff` 查看差异

```
git diff [<path>]
git diff branch1 branch2 [--stat] //[详细]显示出branch2 相比 branch1 做的改动
git diff [<commit-id>] [<path>...]  // 显示工作区和指定的id 在某一个文件的改动
git diff --cached [<commit-id>] [<path>...] // 比较暂存区和指定的 commit 改动
```

Git 模型

Git 模型的 *专用术语* 如下:

- Workspace：工作区
- Index / Stage：暂存区
- Repository：仓库区（或本地仓库）
- Remote：远程仓库

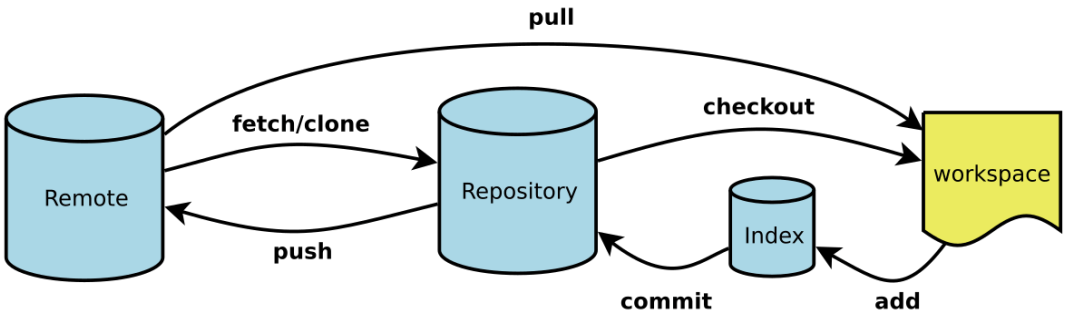

图 1　Git 模型

在使用 `git diff [path]` 命令时，默认是查看工作区(workspace)和暂存区(index)之间的差异。

提示

当需要比较当前本地分支和远程对应分支的差异时，一般按照如下流程来操作：

1. `git fetch <远程>` 获取远程分支的最新改动（移动 /HEAD 指针 ）

2. `git diff <本地分支> <远程>/<远程分支>` （在两个分支之间做比较）

2.3 演示 2

在 `main` 分支添加 `temp.txt`，此时 `temp.txt` 是 `untracked` 状态。尝试执行 `git checkout new_branch`，提示:

```
Please commit your changes or stash them before you switch branches.
Aborting
```

此时可以使用 `git stash push -m "peek at new_branch"` 去保存进度

执行 `git stash list` 命令：

```
➜  Git-Example git:(main)  git stash list
stash@{0}: On main: peek at new_branch
```

执行 `git status` 命令， 发现 temp.txt 仍然是 Untracked 的状态并没有被 stash 进去：

```
On branch main
Your branch is up to date with 'origin/main'.

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

此时执行 `git checkout new_branch` 则能成功切换到 `new_branch` 分支上。

处理完 `new_branch` 上改动, 执行 `git checkout main`, `git stash pop` 恢复到保存前状态：

```
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   "docs/
\345\237\272\347\241\200\346\223\215\344\275\234.typ"

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

此时执行 `git stash list` 显示空。

接下来按如下步骤练习 `git reset` 操作。

1. 创建 feat/a 分支和 feat/b 分支：

2. 在 feat/a 分支上提交 temp.txt 改动（没有提交到远程）：
3. 先尝试使用 `git reset --soft HEAD~1` 来回退到上一次提交：
4. 执行 `git status` 发现两次提交的差异被放到了暂存区：

```
➜  Git-Example git:(feat/a)  git status
On branch feat/a
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   temp.txt
```

再次提交 temp.txt 并使用 `git reset --hard HEAD~1`，此时所有本地改动全部删除。

> 小心
>
> `git reset --hard` 命令会将所有有区别的本地改动都删除（包括提交了的改动和没提交的改动）。误操作的情况下，可以尝试使用 `git reflog` 查找 `git reset` 之前的版本并恢复提交了的改动部分。

在 feat/a 分支上提交所有改动并推送到对应远端。

接下来在 feat/b 分支上按如下操作学习 `git rebase -i`：

切换到 feat/b 分支上，做一些改动并提交；

再做一些改动提交，并 推送到对应远程：

执行 `git log` 显示：

```
Git-Example git:(feat/b)   git log
commit 373509d05c77e0b2c89842a6d508a66368ed2121 (HEAD -> feat/b, origin/
feat/b)
Author:
Date:   Thu Aug 21 11:39:23 2025 +0800

    feat/b change-2

commit 901da01ea6b7db1a3bb33b03631498674b0a755a
Author:
Date:   Thu Aug 21 11:38:20 2025 +0800

    feat/b change-1

commit 97d657766afded606d6102afad68fe5e949ececa (origin/main, main)
```

此时如果我们想要合并这两次远程的提交历史,执行 `git rebase -i HEAD~2` 来处理最近的两次提交历史：

把

```
pick 901da01 feat/b change-1
pick 373509d feat/b change-2
```

改为

```
pick 901da01 feat/b change-1
squash 373509d feat/b change-2
```

表示把第二个改动合并到前一个改动上，最后编辑新的 commit 信息。完成后执行 `git log`：

```
commit 0ff2e03ef271e258b29b4b2a44c6c80694dbf7b1 (HEAD -> feat/b)
Author:
Date:   Thu Aug 21 11:38:20 2025 +0800

    feat: combined b changes

    * Includes change-1 and change-2
    * Unified feature implementation# This is a combination of 2 commits.

commit 97d657766afded606d6102afad68fe5e949ececa (origin/main, main)
```

此时，就完成了本地分支的 commit 的清理工作。

> 注意
>
> 需要使用 `git push -f origin feat/b` 命令来强制推送，因为 `git rebase` 会重写 commit，导致和远端的历史出现不一致的情况。

切换回 main 分支。此时我们想把 feat/a 和 feat/b 的改动合并到 main 分支。就像上面提到的，由于 main 是一个公共分支，在这种分支上我们不能使用 `git rebase` 操作。下面我们使用 `git merge` 来合并。

执行 `git log --oneline -5`：

```
➜  Git-Example git:(main) git log --oneline -5
b3efa5c (HEAD -> main) Merge branch 'feat/b'
8cf4370 (origin/feat/b, feat/b) git rebase -i example
0ff2e03 feat: combined b changes
18e94a4 (origin/feat/a, feat/a) feat/a changes
97d6577 (origin/main, origin/HEAD) git checkout example
```

至此，合并完成。

在 main 分支做一个错误的改动并提交且推送:

```
$1+2 = 3 // '$' 未闭合
```

执行 `git revert HEAD`：

```
➜  Git-Example git:(main) git revert HEAD
[main 4cd8927] Revert "bad commit"
 1 file changed, 2 deletions(-)
```

执行 `git log --oneline -1`：

```
➜  Git-Example git:(main) git log --oneline -1
4cd8927 (HEAD -> main) Revert "bad commit"
```

执行 `git log --oneline | tail -n 3`：

```
➜  Git-Example git:(main) git log --oneline | tail -n 3
3e6cca8 git push/pull example
fb7803e init: typ style
2964a06 add license
```

执行 `git diff fb7803e 3e6cca8` 便可以查看第三次 commit 相对于第二次 commit 多做了什么改动。

随便做一些改动，不需要添加到暂存区，执行 `git diff HEAD`：

就可以看到当前做的一些新改动。