

???

- [HEAD vs main \(+HEAD branch\)](#)
- [Git workflow](#)
- [git describe](#)
- [git merge](#)
- [cherry-pick](#)
- [git pull --allow-unrelated-histories](#)
- [git reset](#)
- [revert](#)
- [git tag](#)
- [git reflog \(git reset --hard\)](#)

HEAD vs main (+?? HEAD ?? ??)

1. ??

??	??	??
HEAD	?? ?? ?? ?? ?? ??	HEAD → main ?? HEAD → <?? ??>
main	?? ?? ?? (Git 2.28+ ?? ??)	main → C3 ??

2. ?? ??

??	HEAD (??)	main (??)
??	?? ??	?? ?? ??
??	"?? ?? ?? ?? ?? " ??	???? ???? ?
?? ??	???? , ?? ??	?? ?? ? ?? , ?? ?? ? ?
Git ?? ??	reset, checkout, reflog	merge, push, checkout ??
?? ?? ??	.git/HEAD	.git/refs/heads/main

3. ??? ??

A---B---C---D

↑

main

↑

HEAD

- HEAD → main → D ??

- `git checkout main` → `HEAD` `main` `main`
- `git checkout <sha>` → `HEAD` **Detached**

4. HEAD? ? ?? ??

상태	HEAD	내용
Attached HEAD	HEAD <code>main</code>	<code>git checkout main</code> (HEAD → main)
Detached HEAD	HEAD <code>a1b2c3d</code>	<code>git checkout a1b2c3d</code> (HEAD detached at a1b2c3d)

```
git checkout main      # HEAD → main
git checkout a1b2c3d   # HEAD → a1b2c3d (Detached)
```

5. ?? ???

명령어	내용
<code>git symbolic-ref HEAD</code>	HEAD <code>refs/heads/main</code>
<code>cat .git/HEAD</code>	HEAD <code>ref: refs/heads/main</code>
<code>git log</code>	HEAD <code>main</code>
<code>git reflog</code>	HEAD <code>main</code> (HEAD detached at a1b2c3d)

6. ?? ???

상태	HEAD	main
현재	<code>a1b2c3d</code> (HEAD)	<code>a1b2c3d</code>
작업	<code>a1b2c3d</code> 1	<code>a1b2c3d</code>
브랜치	<code>a1b2c3d</code> <code>main</code> / <code>main</code>	<code>a1b2c3d</code> (HEAD)
브랜치 생성	<code>a1b2c3d</code> <code>main</code>	<code>a1b2c3d</code> (HEAD)

7. ?? HEAD ?? ??

??	??
HEAD	?? ?? ?? ?? (or ???) ???
FETCH_HEAD	git fetch ?? ??? ?? ???
MERGE_HEAD	git merge ?? , ?? ?? ??
ORIG_HEAD	??? HEAD ?? ??? ?? ?? (ex. reset ?? ??)
CHERRY_PICK_HEAD	cherry-pick ?? ?? ? ??? ?? ??
REBASE_HEAD	rebase ?? ??? ?? ?? ??

8. ?? ??

```
git reset --hard ORIG_HEAD    # ?? ??? ??

git merge feature             # ?? ? MERGE_HEAD ???
cat .git/MERGE_HEAD           # ?? ?? ?? ??

git fetch origin               # FETCH_HEAD ??
cat .git/FETCH_HEAD
```

??

.git/HEAD	?? ?? ??
.git/ORIG_HEAD	reset/merge ? ?? HEAD
.git/FETCH_HEAD	fetch? ??? ?? ?? ?? ??
.git/MERGE_HEAD	merge ?? ?? ??

Git ??? ??

1. Git Commit

1.1 ??

- Git 使用 快照 的方式 记录 文件 的 变化 , 快照 使用 增量 (delta) 的方式 存储 数据 .
- 每次 提交 都会 生成 一个 唯一的 哈希 值 .

1.2 ???

```
git commit
```

2. Git Branch

2.1 ??

- Git 使用 分支 来 管理 不同的 开发 任务 (reference) 指向 不同的 提交 .
- “每个 分支 都是一个 指向 提交 的 指针 ” 每个 分支 都是一个 指向 提交 的 指针 .
- 默认 分支 是 master 分支 , 用于 存放 生产 环境 的 代码 .

2.2 ???

```
git branch -f <branch>  
git checkout <branch> # 切换 分支
```

3. Git Merge

3.1 ??

- Merge 分支到分支，将分支合并到当前分支。
- 分支合并后，分支仍然存在。

3.2 ???

```
git merge <branch>
```

4. Git Rebase

4.1 ??

- 将分支合并到当前分支，并更新分支指针。
- 分支合并后，分支仍然存在。
- A 分支 B 分支 → A 分支 (base) B 分支。

4.2 ???

```
git rebase <branch>
```

5. Git Checkout

5.1 ??

- HEAD 分支，指向当前分支。
- checkout 分支，将分支切换到当前分支。
- HEAD 分支，指向当前分支。

5.2 ???

```
git checkout <branch or commit>
```

6. ?? ?? (Relative Reference)

6.1 ?? ??














- :   
- ~<n>:   

6.2 ??

```
git HEAD^  
git HEAD~2
```

7. Git Reset & Revert

7.1 git reset

-          .
-   ,      .

```
git reset <commit>
```

7.2 git revert

-        .

```
git revert <commit>
```

8. Git Cherry-pick

8.1 ??

-     HEAD   .
-      .

8.2 ???

```
git cherry-pick <commit1> <commit2> ...
```

9. Git Interactive Rebase

9.1 ??

- 交互式变基 允许你重新排列提交顺序，并可以删除或添加提交。

9.2 ???

```
git rebase -i <base>
```

10. Git Tag

10.1 ??

- 为提交创建永久性的标识符，通常用于发布新版本。
- 可以创建轻量级标签或附属于提交的 annotated tag。

10.2 ???

```
git tag <tag> <commit>
```

11. Git Describe

11.1 ??

- 在分支上生成描述符，格式为 `*(tag)*`，表示距离上一个标签的提交数。

11.2 ???


```
git describe <ref>
```

11.3 ?? ??

```
<tag>_<numCommits>_g<hash>
```

12. Git Clone

12.1 ??

- 在本地克隆远程仓库。

12.2 ???

```
git clone <repo_url>
```

13. Git Remote

13.1 ??

- 在本地克隆远程仓库。
- clone 时指定 origin。

13.2 ??

- origin/main 分支，HEAD 指向。

13.3 ??

```
git remote  
git checkout origin/main
```

14. Git Fetch

14.1 ??

- `git fetch` 从远程仓库获取更新, 但不会合并到本地分支。

14.2 ???

```
git fetch <remote>
```

15. Git Pull

15.1 ??

- `git fetch` + `git merge` 从远程仓库获取更新并合并到本地分支。
- `git pull` 是 `git fetch` 和 `git merge` 的快捷方式。

15.2 ???

```
git pull origin main
git diff HEAD origin/main # 查看差异
```

16. Git Push

16.1 ??

- `git push` 将本地分支的更新推送到远程仓库。

16.2 ???

```
git push origin <branch>
```

17. Git fakeTeamwork

17.1 ??

-           .

-

git describe

1. Git Describe??

- `git describe` 在 当前 分支 (ref) 的基础上 向前 追溯 最近的 标签 (tag) , 并 返回 该 标签 的 名称 和 距离 当前 分支 的 步数 .
- 如果 当前 分支 正好 在 一个 标签 上 , 则 返回 该 标签 的 名称 .

2. ?? ?? ??

```
<tag>_<N>_g<short-hash>
```

部分	说明
<tag>	最近的 标签 名称
<N>	距离 最近 标签 的 步数 (0 表示 当前 分支 正好 在 一个 标签 上)
g<短哈希>	Git short hash

3. ?? ?? ??

3.1 --tags

```
git describe --tags
```

- 返回 最近 的 **annotated tag** 的 名称 .
- `--tags` 返回 最近 的 **lightweight tag** 的 名称 .

3.2 --abbrev=0

```
git describe --abbrev=0
```

- 在本地仓库中，使用 `git describe` 命令，可以获取当前 commit 的最近一个 tag 的哈希值。
- 在远程仓库中，使用 `git describe` 命令，可以获取当前 commit 的最近一个 tag 的哈希值。

3.3 --exact-match

```
git describe --exact-match
```

- 在本地仓库中，使用 `git describe --exact-match` 命令，可以获取当前 commit 的最近一个 tag 的哈希值。
- 在远程仓库中，使用 `git describe --exact-match` 命令，可以获取当前 commit 的最近一个 tag 的哈希值。

4. ?? ??

```
git describe          # v2.0.0-2-gabc1234
git describe --tags    # lightweight tag
git describe --abbrev=0 # v2.0.0
git describe --exact-match # v2.0.0 (tag is exact match)
```

5. ????

- `git describe` 命令可以获取当前 commit 的最近一个 tag 的哈希值。
- `lightweight tag` 是指没有指向任何 commit 的 tag。
- `--tags` 选项用于指定要匹配的 tag 的哈希值。
- `--abbrev=0` 选项用于指定要匹配的 tag 的哈希值的位数。

git merge

1. Git Merge??

`git merge` 是 Git 中用于合并分支的命令。它通常用于将开发分支合并回主分支，或者将功能分支合并回开发分支。

2. ?? ??

```
git merge <branch>
```

命令	描述
<code>git merge main</code>	将 main 分支合并到当前分支
<code>git merge <branch></code>	将 <branch> 分支合并到当前分支

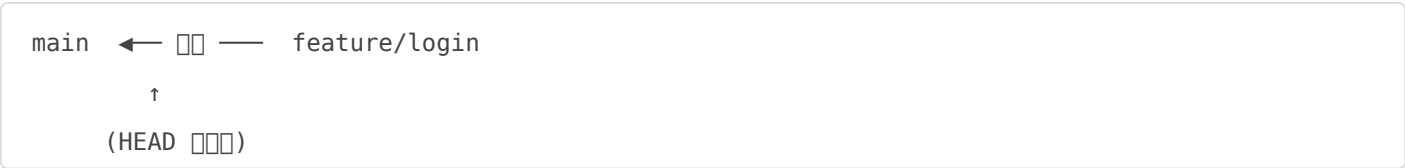
合并后，主分支的指针会指向合并后的新提交，而分支的指针也会指向该提交。

3. ?? ??

```
git checkout main # main 分支 (HEAD 指向)
git merge feature/login # feature/login 分支合并到 main 分支
```

合并后，主分支的指针会指向合并后的新提交，而分支的指针也会指向该提交。

4. ?? ????



5. ?? ??

5.1 ?? ?? ????? ??

```
git checkout main
```

5.2 ?? ??

```
git merge feature/login
```

6. ?? ??

- **Fast-forward merge:** 本地分支 合并 远程分支 更新 本地分支
- **Merge commit:** 本地分支 合并 远程分支 创建 新的 commit

7. ?????

本地分支	远程分支
HEAD 指向	指向 本地分支 或 远程分支 (git checkout <远程分支>)
本地分支	本地分支 与 远程分支 发生 conflict 时 本地分支 → 远程分支
本地分支 与 远程分支	本地分支 与 远程分支 发生 冲突 时 本地分支 与 远程分支

8. ?? ?

- 本地分支 与 远程分支 发生 冲突 时 git pull 本地分支 与 远程分支
- 本地分支 git log 本地分支 gitk 本地分支 与 远程分支
- 本地分支 与 远程分支 发生 冲突 时 --no-ff 本地分支 与 远程分支 发生 冲突 时 :

```
git merge --no-ff feature/login
```


cherry-pick

1. Git Cherry-pick???

```
git cherry-pick 1 2 3 4 5 6 7 8 9 10 .
```

[illegible]

2. ?? ??

```
git cherry-pick <[] []>
```

2

```
git cherry-pick a1b2c3d
```

3. ??? ??

? ??: feature ???? ?? ?? C3? main ???? ???? ?

□□□□ □□

```
main:  A - - - B
```

\

```
feature:      C1---C2---C3---C4
```

```
git checkout main
git cherry-pick C3
```

```
main:      A---B---C3'
           \
feature:    C1---C2---C3---C4
```

```
feature:      C1---C2---C3---C4
```

“C3” C3

4. ?? ?? cherry-pick

```
git cherry-pick <commit>^..<commit>
```

```
git cherry-pick C2^..C4
```

$\rightarrow C_2, C_3, C_4$

5. ?? ?? ?

• ☐ ☐ ☐ ☐ ☐ ☐

-

```
git add .  
git cherry-pick --continue
```

- `git cherry-pick --abort`

`git cherry-pick --abort`

6. ?? ?? ??

??	??
?? ??	<code>main</code> ?? ?? ?? <code>release</code> ???
?? ??	?? ?? ? ?? ?? ??
?? ??	??? ??? ?? ?? ???

7. cherry-pick vs merge vs rebase

??	?? ??	??? ??	??
<code>merge</code>	?? ??	$O(\text{?? ?? ??})$?? ??? ??
<code>rebase</code>	?? ??	$O(\text{??? ??})$??? ?? ???
<code>cherry-pick</code>	?? ??	$O(\text{?? ?? ??})$?? ?? ??

8. ?????

- ?? ?? ?? ?? ??? `(C3')` ?? ??
- ?? `cherry-pick` ?? ?? ?? ?? ??
- ?? ?? ?? ??? ?? ? ??? ?? ?? ??

git pull --allow-unrelated-histories

1. git pull --allow-unrelated-histories ??

`--allow-unrelated-histories` Git 的 一个 选项，用于 允许 合并 不相关 的历史记录。

通常，Git 要求 合并 的 分支 必须 有 共同的 祖先。但是，如果你 想要 合并 两个 没有 共同 祖先 的 分支，就需要 使用 `--allow-unrelated-histories` 选项。

2. ?? ?? ??

```
fatal: refusing to merge unrelated histories
```

? ?? ??????

- 初始化 `git init` 并 添加 文件 到 仓库
- 推送到 `(GitHub)` 仓库 (文件 : README.md)
- 使用 `git pull origin main` 拉取 更新

3. ?? ??

```
git pull origin main --allow-unrelated-histories
```

- 如果 遇到 冲突，使用 `git pull origin main --allow-unrelated-histories` 解决
- 如果 遇到 冲突 (conflict)，使用 `git pull origin main --allow-unrelated-histories` 解决

4. ?? ?

```
# 本地分支
git init
git remote add origin https://github.com/user/repo.git

# 远程分支
git pull origin main --allow-unrelated-histories
```

5. ?????

- 本地分支与远程分支同步 : 使用 `git pull` 命令
- 本地分支与远程分支同步 : 使用 `git pull` 命令

6. ?? ??

命令	说明
<code>--rebase</code>	在合并时，将本地分支的提交重新基于远程分支的提交
<code>--no-commit</code>	在合并时，不提交本地分支的提交
<code>--strategy=ours / theirs</code>	在合并时，选择使用哪一方的提交

git reset

1. Git Reset ???

git reset Git 的 命令 可以 (HEAD) 的 状态 重置 , 但是 不会 删除 文件 , 只 会 删除 目录 下 的 文件 .

“ 在 本地 仓库 中 的 文件 被 删除 后 , 在 下次 提交 时 会 被 删除 . ”

2. reset ?? ?? ??

命令	HEAD 指针	Staging Area 指针	工作区 文件 状态
--soft	指向 O	指向 空	保留 文件
--mixed	指向 O	指向 空	保留 文件
--hard	指向 O	指向 空	删除 文件

3. ??? ??

命令 示例:

A---B---C---D (HEAD)
 ↑
 git reset

git reset --soft B

命令 示例:

A---B (HEAD)

↑

현재 커밋 C, D로 리셋된 상태

```
git reset --hard B
```

HEAD:

A---B (HEAD)

↑

C, D 커밋은 현재 HEAD에서 삭제된 상태 (리셋된 상태)

4. ??? ??

? --soft

```
git reset --soft <커밋>
```

- 커밋된 파일은 유지
- 커밋된 파일은 현재 HEAD에서 삭제된 상태

현재 HEAD :

```
“커밋된 파일은 현재 HEAD에서 삭제된 상태, 커밋된 파일은 현재 HEAD에서 삭제된 상태”
```

? --mixed (?? ??)

```
git reset --mixed <커밋>
```

- 커밋된 파일 & 커밋된 파일
- 커밋된 파일은 현재 HEAD에서 삭제된 상태

현재 HEAD :

```
“add된 파일은 현재 HEAD에서 삭제된 상태, git add된 파일은 현재 HEAD에서 삭제된 상태”
```

? --hard

```
git reset --hard <commit>
```

- 1. 2. 3. 4. 5. 6. 7. 8. 9. 10.

11. 12. :

“ 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11. 12. ”

△ 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11. 12. !

5. ?? ??

```
# 1. 2. 3. 4.
echo "a" > a.txt && git add . && git commit -m "A"
echo "b" > b.txt && git add . && git commit -m "B"
echo "c" > c.txt && git add . && git commit -m "C"

git log --oneline
```

```
# 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11. 12.
git reset --soft HEAD~1 # 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11. 12.
git reset --mixed HEAD~1 # 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11. 12.
git reset --hard HEAD~1 # 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11. 12.
```

6. git reset vs git revert ??

| 1. | 2. | 3. | 4. | 5. |
|--------|----|------|----|-------------------------------------|
| reset | 1. | 2. X | 3. | 4. 5. 6. 7. 8. 9. 10. , 11. 12. |
| revert | 1. | 2. O | 3. | 4. 5. 6. 7. 8. 9. 10. , 11. 12. 13. |

? ??

| ?? | ?? | ?? ?? |
|---------|--------------------------|---------------|
| --soft | ??? ?? , staging? ??? | ??? ?? ??? ?? |
| --mixed | ?? + staging ?? , ??? ?? | ??? |
| --hard | ?? + staging + ??? ?? ?? | ??? ? ?? *?? |

revert

git tag

1. Git 什么?

- **Git 的 (Tag)** 是 一个 特殊的 分支 名称
- 通常 用于 标记 重要的 版本 或 发布 版本
- 与 分支 不同, 标签 是 不可变 的

2. 什么 什么 什么

| 名称 | 描述 | 用途 |
|------|-------------------|-------------|
| 轻量级 | 简单 快速 创建 标签 | 快速 标记 版本 |
| 标注式 | 包含 提交 信息 和 元数据 | 详细 记录 版本 信息 |
| 分支 | 用于 开发 新功能 或 修复 问题 | 管理 代码 分支 |
| HEAD | 指向 当前 分支 的 指针 | 跟踪 当前 分支 |

3. 什么 什么

3.1 Lightweight Tag (什么 什么)

- 简单 快速 创建 标签
 - 不包含 提交 信息 和 元数据 (X)
- git tag v1.0.0

3.2 Annotated Tag (什么 什么)

- 包含 提交 信息 和 元数据

- 创建分支，并推送到远程仓库。GPG 签名

```
git tag -a v1.0.0 -m "Release version 1.0.0"
```

4. 创建新分支并推送到远程仓库

4.1 创建新分支

```
git tag
```

4.2 创建新分支并推送到远程仓库

```
git tag v1 <commit-id>
```

4.3 创建新分支并推送到远程仓库

```
git checkout v1
```

- 当前分支：HEAD(detached HEAD) 指向 commit
- → 创建新分支，并推送到远程仓库

```
git checkout -b new-branch-from-v1
```

4.4 创建新分支并推送到远程仓库

```
git tag -d v1
```

4.5 创建新分支并推送到远程仓库

```
git push origin v1
```

- 创建分支并推送到远程仓库

```
git push origin --tags
```

4.6 ????? ?? ??

```
git push origin --delete tag v1
```

5. git describe: ?? ??? ?? ??

5.1 ???

```
git describe
```

- HEAD 指向 2 个 commit 的 父 commit

```
v1.0.0-2-g9f23a1c
```

- v1.0.0 → 1 个 commit 的 父 commit
- -2 → 2 个 commit 的 父 commit
- g9f23a1c → 2 个 commit 的 父 commit

6. ?????

- 1.0.0 版本 1.0.0 的 父 commit, 1.0.0 的 父 commit
- 1.0.0 的 父 commit 的 父 commit : 1.0.0

```
git checkout -b fix-hotbug v1.0.0
```

7. ?? ?? ??

```
# 1.0.0 1.0.0 1.0.0  
git tag v2.0.0
```

```
# 1.0.0 1.0.0 1.0.0
```

```
git tag -a v1.0.0 <tag name> -m "Initial release"
```

创建 tag

```
git tag
```

推送 tag

```
git push origin --tags
```

? ??? ??

| 操作 | 命令 |
|-----------|--|
| 创建 tag | <code>git tag v1.0.0</code> |
| 带消息创建 tag | <code>git tag -a v1.0.0 -m "tag message"</code> |
| 列出 tag | <code>git tag</code> |
| 推送 tag | <code>git push origin v1.0.0</code> |
| 推送所有 tag | <code>git push origin --tags</code> |
| 删除 tag | <code>git tag -d v1.0.0</code> |
| 删除并推送 tag | <code>git push origin --delete tag v1.0.0</code> |
| 生成可发布的描述 | <code>git describe</code> |

git reflog (git reset ? ?? ??)

1. ?? ??

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

❌ ❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽ ❾ ❿ ❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽ ❾ ❿ .

2. git reflog??

“Git❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽ ❾ ❿ .

```
git reflog
```

❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽ ❾ ❿ .

```
c3d9f7b HEAD@{0}: reset: moving to HEAD~1
e7a1b2a HEAD@{1}: commit: ❶ ❷
...
```

→ ❶ HEAD@{1}❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽ ❾ ❿ .

3. ?? ??

```
git reset --hard HEAD@{1}
```

❶ ❷ ❸ ❹ :

```
git reset --hard e7a1b2a
```

→ ❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽ ❾ ❿ !

?? ? : ???? ????

git --hard , , .

```
git checkout -b recovery HEAD@{1}
```

? ??

| 命令 | 作用 |
|-------------------------------|-------------------|
| git reflog | HEAD, 分支 分支 分支 |
| git reset --hard HEAD@{n} | 分支 分支 分支 分支 |
| git checkout -b <分支> HEAD@{n} | 分支 分支 分支 (分支 分支) |