Git 变基（Rebase）基础教程

叶家炜

Jan 13, 2026

Git 变基是一种强大的工具，它将一个分支的提交「重新应用」到另一个分支上，从而实现干净的线性历史记录。这种操作不同于传统的合并，它能避免多余的合并提交，让项目历史看起来更加简洁明了。变基的核心在于重新排列提交序列，使分支间的关系更直观。

相比于 merge 操作，变基的主要优势在于保持历史线性。Merge 会创建一个额外的合并提交，记录两个分支的融合过程，这在多人协作时可能导致历史记录变得杂乱。而变基则将 feature 分支的变更「移植」到主分支顶端，形成一条平滑的直线。这种方式特别适合个人开发或清理提交历史，但需要注意它会改写提交历史，因此不适用于已共享的分支。

变基与 merge 的直观对比可以想象为：merge 像两条河流汇合形成一个分叉口，而变基则像将一条支流顺直地接续到主河道上。前者保留了所有历史痕迹，后者追求简洁的单一线性路径。这种差异在长期项目中尤为明显，线性历史更容易 bisect 查找问题。

本文面向 Git 新手到中级用户，如果你已经掌握基本的 commit、branch 和 checkout 命令，就可以轻松跟进。阅读前提包括理解 Git 的基础工作流，如创建分支和切换分支。没有这些基础，建议先复习 Git 官方入门文档。

文章结构从基础概念入手，逐步深入到实际操作、高级技巧、问题排查和最佳实践，最后提供快速参考和练习建议。通过层层递进，你将掌握变基的全貌。

1　变基基础概念

变基的工作原理本质上是将当前分支的提交从其原有基点「剥离」，然后逐一重新应用到目标分支的顶端。具体过程是：Git 首先找到两个分支的共同祖先提交，然后将当前分支独有的提交「暂停」，切换到目标分支，再按顺序「replay」这些提交。每个 replay 过程相当于 cherry-pick 一个提交，如果有冲突则暂停等待用户解决。这种「移植」机制确保了提交内容的完整性，但会生成全新的提交哈希值。

常见变基场景包括当前分支变基到目标分支，使用命令 `git rebase <target>`，这会将当前分支的变更叠加到 target 分支上。另一个场景是交互式变基，通过 `git rebase -i` 可以编辑提交序列，比如合并或删除提交。这两种场景覆盖了 90% 的使用需求。

变基过程中有三种状态：正在进行时，Git 会标记 rebase 状态文件；已暂停状态通常因冲突发生，需要手动干预；已完成状态则一切顺畅，历史已重写。理解这些状态有助于诊断问题。

关键术语中，Base Commit 是变基的基准提交，即目标分支的顶端；Replay 表示重新应用提交的过程；Pick 是交互式变基中的默认动作，意为保持原样；Squash 则将当前提交合并到上一个提交中，结合它们的变更和日志。

2 环境准备

要开始学习变基，首先创建示例仓库。执行以下命令序列：`git init rebase-demo`，然后 `cd rebase-demo`。接下来创建初始提交，例如触碰一个文件 `echo Initial commit > README.md` 并 `git add .`，最后 `git commit -m Initial commit`。这个仓库将作为所有演示的基础。
在仓库中创建测试分支结构：在 main 分支上添加几个提交，如 `echo Main change 1 >> README.md`、`git add .`、`git commit -m Main change 1`，重复几次。然后创建 feature 分支 `git checkout -b feature`，并在其上添加独有提交，如 `echo Feature change >> README.md`、`git commit -m Feature change`。现在你有 main 和 feature 两条平行分支，完美模拟真实开发场景。
为提升体验，配置 `git config --global rebase.autoStash true`，这会在变基时自动暂存未提交变更，避免手动 stash。推荐工具包括 Git GUI 用于可视化历史，以及 VS Code 的 Git Graph 扩展来观察分支变化。

3 基本变基操作

3.1 简单变基

简单变基是最基础的操作，假设你在 feature 分支上，执行 `git checkout feature`，然后 `git rebase main`。这个命令的解读如下：首先切换到 feature 分支，确保它是干净的；然后 rebase main 告诉 Git 将 feature 的提交从 main 的顶端重新应用。Git 会找到 main 和 feature 的分叉点，将 feature 之后的提交逐一 replay 到 main 顶端。如果无冲突，feature 分支现在「骑」在 main 上，形成线性历史。
预期结果是：变基前，main 和 feature 平行；变基后，feature 的提交直接接在 main 末尾，原有 feature 基点被遗弃。新提交有全新哈希，但内容相同。这种操作常用于将本地 feature 同步到远程 main 前，保持干净历史。

3.2 处理变基冲突

变基冲突发生在 replay 提交时，变更与目标分支重叠。机制是 Git 尝试应用补丁，如果文件行冲突则标记 `<<<<<` 等符号。解决步骤：先 `git status`，它会显示「rebase in progress」和冲突文件；编辑冲突文件，手动选择保留哪部分代码；然后 `git add .` 标记已解决；最后 `git rebase --continue` 继续下一个提交。
完整示例：假设 main 有 `echo foo > file.txt`，feature 有 `echo bar >> file.txt`，变基时冲突。编辑后文件可能成 `foo\nbar`，add 并 continue。整个过程确保变更不丢失，但需仔细审查。

3.3 中止变基

如果冲突太棘手，使用 `git rebase --abort`。这个命令解读为：中止当前变基，恢复到 rebase 开始前的分支状态，包括 HEAD 和索引。它会删除 rebase 状态文件，一切如初。使用时机是当你不确定如何解决冲突，或变基策略错误时；必须使用则是如果误操作导致不可逆混乱。

4 交互式变基

4.1 基本语法

交互式变基通过 `git rebase -i HEAD~3` 编辑最近 3 个提交。这个命令解读：`-i` 启用交互模式，HEAD~3 指定从倒数第三个提交开始的范围。Git 会弹出编辑器，显示提交列表，默认全为 pick。保存退出后，Git 按指令执行。
另一种是 `git rebase -i main`，将当前分支变基到 main 前，同时交互编辑。这适合将 feature 的提交精简后叠加到 main。

4.2 常用操作命令详解

交互式变基的核心是编辑器中的命令。`pick` 保持提交不变，是默认选项，用于正常保留。`reword` 只修改提交信息，如修正拼写，Git 会暂停让你编辑消息后继续。`edit` 在该提交处暂停，允许修改代码或作者，然后 `git rebase --continue`。
`squash` 将当前提交合并到上一个，结合变更并让你编辑合并消息，常用于清理小修复。`fixup` 类似 squash 但丢弃当前提交信息，直接融入上一个，适合临时提交。`drop` 完全删除提交，用于移除错误。

4.3 实战案例

修改最近提交信息：`git rebase -i HEAD~1`，将 pick 改为 reword，保存后编辑消息如从「Fix bug」改为「修复登录验证 bug」，继续即可。
合并多个小提交：`git rebase -i HEAD~3`，将后两个改为 squash，编辑器出现合并消息界面，合成「feat: 添加用户模块」。
删除错误提交：`git rebase -i HEAD~4`，将目标行改为 drop，保存后该提交消失。
分离大提交：先 `git reset HEAD~1`，然后重新 commit 分拆，最后 `git rebase -i HEAD~n` 调整顺序。

5 高级变基技巧

5.1 变基到上游分支

`git rebase --onto main featureA featureB` 将 featureB 从 featureA 之后的提交变基到 main 上。解读：`--onto main` 指定新基点，featureA 是旧基点分界，featureB 是目标分支。这常用于将变更从一个分支「移植」到另一个上游，常在多分支协作中应用。

5.2 保留特定提交

`git rebase --onto new-base old-base` 将当前分支从 old-base 之后的提交应用到 new-base。解读：old-base 是保留前缀的分界，新提交只 replay old-base 之后部分。这用于精确控制历史片段。

5.3 批量修改提交作者

`git rebase -i --exec git log --oneline -1 HEAD~5` 在每个 pick 后执行命令。解读：`-i` 交互，`--exec` 指定每次暂停运行 `git log --oneline -1` 查看最新提交，HEAD~5 范围为最近 5 个。实际中可换成 `git commit --amend --author=New Author` 批量改作者。

5.4 变基公共分支的最佳实践

绝对不要对已推送公共分支变基，因为它改写历史会导致他人拉取混乱。如果必须推送，使用 `git push --force-with-lease`，它检查远程是否变化，安全覆盖。

6 常见问题与解决方案

遇到「Cannot rebase: already in progress」是因为变基未完成，使用 `git rebase --abort` 清理或 `--continue` 推进。冲突解决后提交丢失可能是未正确 continue，检查 `git reflog` 找到旧 HEAD 并 reset 恢复。变基后历史混乱通常是对公共分支操作，重置 `git reset --hard origin/main`。交互式保存失败源于编辑器，配置 `git config --global core.editor code --wait` 解决。

7 最佳实践与注意事项

变基推荐用于个人分支和清理历史，如 feature 分支变基前推 main。禁止用于已推送公共分支或共享分支，以免团队冲突。在团队中，策略是 feature 变基到 main 后 merge。变基前检查清单：确认分支干净、无未推提交、备份 reflog。

与 Git Flow 结合，在 release 前变基 feature；GitHub Flow 中，PR 前变基保持线性。

8 快速参考命令表

基础变基：`git rebase main` 将当前变基到 main；`git rebase --abort` 中止；`git rebase --continue` 继续。

交互式：`git rebase -i HEAD~n` 编辑最近 n 个；`git rebase -i --autosquash` 自动处理 fixup。

高级：`git rebase --onto A B C` 将 C 从 B 到 A；`git push --force-with-lease` 安全推送。

9 实践练习

练习 1：基础变基，在示例仓库 `git checkout feature`、`git rebase main`，观察 `git log --oneline --graph`。

练习 2：交互式合并，在 feature 添加 3 小提交，`git rebase -i HEAD~3` squash 后两个。

练习 3：制造冲突，编辑相同行后解决并 continue。

练习 4：故意出错，用 `git reflog` 恢复。完整仓库可在 GitHub rebase-demo 下载实践。

10 结论

变基关键要点：线性历史、交互编辑、冲突处理、安全推送。它的价值在于干净历史促进高效协作。下步学习 Git LFS 或 Submodules。鼓励立即实践，形成肌肉记忆。

11 附录

图形工具如 GitKraken 可视化变基。官方文档：`git rebase --help`。常见错误：NO-REBASE-OPTION 用 abort；FAQ 示例：变基是否改哈希？是，新提交全新 ID。