生物信息学系列教程

生物信息学 Git 基础教程

Git Essentials for Bioinformatics

版本控制 • 团队协作 • 可重复分析

面向生信工作者的系统入门与实践指南

第 1 版　2025 年

目录

前言 **5**

前言 **5**

第 1 章　第 1 章为什么生信需要版本控制 **6**

1.1　1.1 可重复性危机 6

1.2　1.2 版本控制解决的三类问题 6

1.3　1.3 Git 能做什么、不能做什么 7

1.4　1.4 生信项目中需要” 版本化” 的对象 7

第 2 章　第 2 章 Git 核心概念 **9**

2.1　2.1 三个区域：工作区、暂存区、版本库 9

2.2　2.2 提交是快照，不是差异 9

2.3　2.3 分支：可移动的指针 9

2.4　2.4 对象模型速览 10

2.5　2.5 引用与 reflog：一切皆可恢复 10

第 3 章　第 3 章安装与初始化配置 **11**

3.1　3.1 安装 Git 11

3.2　3.2 首次配置：身份 11

3.3　3.3 跨平台三件套：换行符、.gitattributes、编码 12

3.4　3.4 配置 SSH 密钥：免密连接 GitHub/GitLab 13

3.5　3.5 配置层级 13

3.6　3.6 实用的全局配置 14

第 4 章　第 4 章日常基本操作 **15**

4.1　4.1 初始化与克隆 15

4.2　4.2 查看状态与差异 15

4.3　4.3 暂存与提交 16

4.4　4.4 查看历史 16

4.5　4.5 撤销与回退 17

4.6　4.6 .gitignore：生信仓库的第一道防线 17

4.7　4.7 标签：标记版本 18

第 5 章　第 5 章分支与协作 **20**
5.1　5.1 分支的基本操作 20
5.2　5.2 合并：merge 20
5.3　5.3 rebase 与 merge 的取舍 21
5.4　5.4 冲突的解决 21
5.5　5.5 远程仓库：remote、push、pull、fetch 22
5.6　5.6 Pull Request / Merge Request 流程 22
5.7　5.7 团队工作流选型 23

第 6 章　第 6 章生信项目的 Git 实战 **24**
6.1　6.1 项目目录结构设计 24
6.2　6.2 工作流脚本入库：Snakemake / Nextflow 24
6.3　6.3 环境与配置入库 25
6.4　6.4 结果何时入库 26
6.5　6.5 与 R / Python 工具集成 26
6.6　6.6 Git LFS 与大规模数据 26
6.7　6.7 论文配套代码发布 27

第 7 章　第 7 章高级技巧与踩坑指南 **28**
7.1　7.1 生信常见坑 28
7.2　7.2 reflog：找回”丢失”的提交 28
7.3　7.3 历史重写：谨慎行事 29
7.4　7.4 效率技巧：别名与补全 29
7.5　7.5 pre-commit：自动化质量检查 29
7.6　7.6 子模块：慎用 30
7.7　7.7 Git 与 conda 环境的配合 30

第 8 章　第 8 章开源参与与协作规范 **32**
8.1　8.1 在 GitHub 上参与生信开源项目 32
8.2　8.2 提交信息规范：Conventional Commits 32
8.3　8.3 issue、review 与文档文化 33
8.4　8.4 许可证与署名 33

附录 A　常用命令速查表 **34**
A.1　A.1 配置 34
A.2　A.2 仓库操作 34
A.3　A.3 提交 34
A.4　A.4 撤销与回退 35
A.5　A.5 分支与合并 35
A.6　A.6 远程协作 35

附录 B 生信.gitignore 完整模板 **36**

附录 C 最佳实践清单 **38**

参考文献 **39**

参考文献 **39**

前言

生物信息学是” 脚本驱动” 的学科：我们的分析成果最终体现为一套代码、一组配置、一份工作流和若干结果文件。然而，大多数生信工作者的代码管理仍停留在” 文件夹 + 日期命名” 的阶段：`pipeline_v2_final.sh`、`analysis_20250101_ 改.py`、`论文终稿 _ 真终稿.docx`。当分析迭代十几次、合作者反复修改脚本、服务器上的环境和本地的环境不一致时，这种管理方式会迅速失控，甚至让几个月的工作无法重现。

Git 是目前事实标准的版本控制系统（Version Control System, VCS），也是 GitHub、GitLab、Gitee 等协作平台的基础。它解决的不是” 写代码” 的问题，而是” 让代码的历史可追溯、可协作、可恢复” 的问题——这正是可重复研究（reproducible research）的第一块基石。

本书面向有基本命令行经验、正在从事或即将从事生物信息学分析工作的读者，系统讲解 Git 的核心概念、日常操作与生信场景的最佳实践。全书不假设读者有编程经验，只要求能在终端中执行命令。

如何使用本书

- 第 1–2 章建立” 为什么” 与” 是什么”，建议通读；
- 第 3–5 章是动手操作的主体，建议边读边在练习仓库中敲命令；
- 第 6 章是本领域的重点应用，结合工作流（Snakemake/Nextflow）、环境管理（conda）与数据管理讲解；
- 第 7 章是踩坑与进阶，遇到问题时回来查阅；
- 第 8 章与附录作为协作规范和速查工具。

版本说明：本书示例基于 Git 2.23 及以上版本（命令 `git --version` 可查看）。书中使用 Git Bash（Windows）或标准终端（macOS/Linux）执行命令，`$` 为提示符，`#` 开头为注释。

第 1 章　第 1 章为什么生信需要版本控制

1.1　1.1 可重复性危机

2016 年《Nature》的调查显示，超过 70% 的受访科研人员无法重现他人的实验，超过一半无法重现自己的实验。在生信领域，问题尤为突出：分析流程环节多、依赖版本敏感、数据体积大。一项 RNA-seq 分析可能涉及 20 多个工具、几十个参数、若干版本的参考基因组，任何一个环节失忆，结果都难以复原。

可重复性的敌人，正是” 不可追溯的变化”：

- 脚本被反复覆盖，“上一版还能跑通的代码” 找不回来；
- 合作者各自修改后互相覆盖文件，不知道谁改了什么；
- 发表论文几个月后，审稿人或读者索要代码，已无法复原当时的版本；
- 服务器清理、硬盘损坏、误删目录，数年积累付之东流。

版本控制并不能自动消除这些问题，但它提供了一套机制，让**每一个变化都被记录、可追溯、可回退**，从而把” 凭记忆管理” 变成” 凭记录管理”。

1.2　1.2 版本控制解决的三类问题

问题	没有版本控制	使用 Git 之后
追溯	“这个结果是谁、在什么时候、用什么参数跑出来的？” ——靠聊天记录和文件名猜	`git log` 查看每次提交的作者、时间与说明；`git blame` 定位每一行的来源
协作	文件互相覆盖，合并靠手工比对	分支并行开发，`merge` 自动三方合并，冲突逐处解决
恢复	误删/改坏后束手无策	`git restore`、`git reset`、`git reflog` 可回到任意历史状态

1.3 1.3 Git 能做什么、不能做什么

能：追踪文本类文件的全部历史——脚本（`.py`/`.R`/`.sh`）、配置文件（`.yaml`/`.json`）、文档（Markdown/LaTeX/`README`）、小型数据表（`.csv`/`.tsv`）、工作流定义（Snakefile/`nextflow.config`）。

不能（也不该）：

- **管理大型二进制数据**。测序原始数据（`.fastq.gz`，动辄数十 GB）、比对结果（`.bam`）、中间产物不应直接放入 Git 仓库——Git 对二进制文件只能整份存储差异，仓库会迅速膨胀到无法使用。数据应放在独立的存储（服务器磁盘、云端对象存储、数据管理工具），详见第 6 章。
- **替代数据备份**。Git 记录的是" 版本历史"，不是" 异地容灾"。仍需要定期将仓库推送到远程（GitHub/GitLab 或自建服务器）。
- **替代科学记录**。版本控制记录" 做了什么"，而实验设计、参数选择的原因等" 为什么" 仍应写入实验记录与文档。

一句话原则：**代码入库，数据入库外，环境可复现。**

1.4 1.4 生信项目中需要" 版本化" 的对象

一个典型的生信分析项目包含多类对象，它们的版本管理策略各不相同：

对象	示例	是否入库	策略
分析代码	脚本、工作流、函数库	是	Git 全文追踪
配置文件	`config.yaml`、`environment.yml`、Dockerfile	是	Git 追踪，改动即记录
文档	README、实验记录（Markdown）	是	Git 追踪
小数据	样本表（`samples.tsv`）、临床注释	是（体积小、文本）	Git 追踪
参考数据	参考基因组、注释库	否（大、不可变）	单独管理，记录版本号与下载命令
中间结果	比对、定量、质控输出	否	可再生成，不入库；必要时用 Git LFS
最终结果	图表、报告	视情况	论文图表可入库；中间表格只留可再生成的脚本

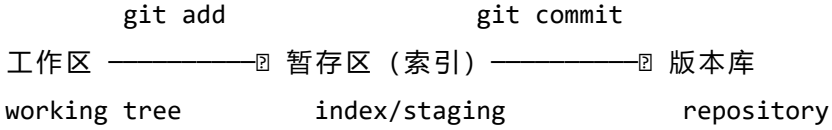
对象	示例	是否入库	策略
计算环境	conda 环境本身	否	用 `environment.yml` 描述，环境本身不入库

理解” 什么该入库”，是生信使用 Git 与普通程序员最大的不同。本书第 6 章将详细展开。

第 2 章　第 2 章 Git 核心概念

2.1　2.1 三个区域：工作区、暂存区、版本库

Git 把一次” 保存” 拆成两步，对应两个区域：



- **工作区（working tree）**：你当前看到的、正在编辑的文件。
- **暂存区（index，又称 staging area）**：`git add` 把文件从工作区放入暂存区，表示” 我确认要提交这些改动”。暂存区允许你把一次任务的多处修改拆成多个逻辑清晰的提交。
- **版本库（repository）**：`git commit` 把暂存区内容固化为一次提交，永久记录进历史。

理解这个三步流程，就能解释 Git 中最常见的困惑：“我明明保存了文件，为什么 `git status` 还说有改动？”——因为 Git 只关心**你明确告诉它**的变化。

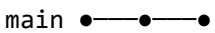
2.2　2.2 提交是快照，不是差异

Subversion 等早期版本控制记录的是” 文件间的差异”。Git 相反：每次提交保存的是**整个项目在该时刻的完整快照**（技术上通过内容寻址与压缩去重实现，磁盘开销远小于直觉）。这意味着：

- 任意两次提交之间都可直接对比，无需从起点逐次” 叠加”；
- 历史中的任何一点都可以完整恢复；
- 提交内容由哈希值（SHA-1，40 位十六进制）唯一标识，如 `3a2f5c…`，无法篡改而不被发现。

2.3　2.3 分支：可移动的指针

分支（branch）是 Git 最强大的概念，也是最常被误解的概念。本质上，**分支只是一个指向某次提交的、可移动的指针**。



创建新分支只是新建一个指针；切换分支（`git switch`）只是让 `HEAD` 指向另一个指针。分支的成本几乎为零，因此 Git 鼓励” 一任务一分支” 的轻量工作流。第 5 章将详细讨论分支的协作用法。

2.4　2.4 对象模型速览

了解 Git 的底层对象模型，能消除大量” 黑盒恐惧”。Git 仓库本质上是一个内容寻址的对象库：

对象	内容	说明
blob	一个文件的全部内容	按内容哈希命名，内容相同则去重
tree	目录清单	记录每个文件名与对应的 blob/tree 哈希
commit	快照的元信息	指向一棵 tree、父提交、作者、时间、提交说明

用 `git cat-file -p <哈希>` 可以查看任意对象。例如：

```
$ git log --oneline
abc1234 更新质控阈值
$ git cat-file -p abc1234
tree 8f2a3b...
parent 9e1c0d...
author Kael <kael@example.org> 1735689600 +0800
committer Kael <kael@example.org> 1735689600 +0800

更新质控阈值
```

2.5　2.5 引用与 reflog：一切皆可恢复

`main`、`HEAD`、`v1.0` 这些名字统称**引用（reference）**，它们是指向提交的符号名。Git 还维护一份 **reflog**（引用日志），记录 `HEAD` 和分支指针每一次移动的历史——即使你误删分支、`reset` 回退过头，只要 reflog 中还有记录，就能找回。这给 Git 带来一个重要特性：**常规操作几乎不会永久丢失数据**（详见 7.2 节）。

第 3 章　第 3 章安装与初始化配置

3.1　3.1 安装 Git

Windows：推荐从 https://git-scm.com/download/win 下载安装（自带 Git Bash、Git GUI 与 credential manager）。也可通过 conda 安装：

```
conda install -c conda-forge git
```

macOS：`brew install git`，或安装 Xcode Command Line Tools（自带 git）。

Linux（Ubuntu/Debian）：

```
sudo apt update && sudo apt install -y git
```

安装后验证：

```
$ git --version
git version 2.47.1
```

> **提示**：Windows 用户建议使用 Git Bash（而非 PowerShell/cmd）执行本书命令，路径与转义规则与 Linux/macOS 一致，减少跨平台差异。

3.2　3.2 首次配置：身份

Git 每次提交都要记录作者，因此必须先配置姓名与邮箱：

```
git config --global user.name "Kael Zhang"
git config --global user.email "kael@example.org"
```

生信实战要点：邮箱务必与你的 GitHub/GitLab 账号绑定邮箱一致——否则提交无法关联到你的账号，贡献图与署名都会错乱。常用做法是用 GitHub 提供的隐私邮箱（`<id>+<用户名>@users.noreply.github.com`）。

查看配置：

```
git config --global --list
```

3.3　3.3 跨平台三件套：换行符、.gitattributes、编码

生信环境横跨 Windows 本机与 Linux 服务器，换行符差异是经典的”幽灵 bug”源头：Windows 用 `CRLF`，Linux/macOS 用 `LF`，混用会导致脚本报错、`diff` 显示整文件被改、`.sh` 脚本在服务器上出现 `\r` 错误。

第一件：设置全局换行符策略

```
git config --global core.autocrlf input      # Windows 提交时转为 LF
```

- Windows 推荐 `input`（检出时保持 LF 或按文件属性处理）或 `true`（检出时转 CRLF，提交时转 LF）；
- macOS/Linux 无需设置（默认 LF）。

第二件：用.gitattributes 显式声明规则。在仓库根目录创建.gitattributes，比全局配置更可靠，且随仓库分发给所有协作者：

```
# 所有文本文件统一用 LF 入库
* text=auto
*.sh text eol=lf
*.py text eol=lf
*.R text eol=lf
*.yaml text eol=lf
*.md text eol=lf
# 二进制文件（即使扩展名是文本也可能包含二进制，如 xlsx）
*.xlsx binary
*.bam binary
*.fastq.gz binary
```

第三件：统一 UTF-8 编码。所有脚本与文档保存为 UTF-8（无 BOM）。Windows 记事本默认编码容易引入 BOM 或 GBK，建议用 VS Code、Vim 或 RStudio 编辑并显式选择 UTF-8。

3.4 3.4 配置 SSH 密钥：免密连接 GitHub/GitLab

生信工作者常在服务器上操作仓库，SSH 比 HTTPS 更适合（免去每次输密码，且支持代理）。生成密钥：

```
ssh-keygen -t ed25519 -C "kael@example.org"
# 一路回车使用默认位置 ~/.ssh/id_ed25519
```

查看公钥并添加到 GitHub（Settings → SSH and GPG keys → New SSH key）：

```
cat ~/.ssh/id_ed25519.pub
```

测试连接：

```
ssh -T git@github.com
# Hi Kael! You've successfully authenticated...
```

之后克隆使用 SSH 地址：

```
git clone git@github.com:kael/awesome-nf.git
```

> **服务器场景：**在计算服务器上配置 SSH 密钥后，即可直接 `git push/pull`。注意不要把私钥（`id_ed25519`）泄露或入库。

3.5 3.5 配置层级

Git 配置分四个层级，后者覆盖前者：

层级	命令	作用范围
system	`git config --system`	整台机器所有用户
global	`git config --global`	当前用户所有仓库
local	`git config --local`（默认）	当前仓库
worktree	`git config --worktree`	当前工作树

身份、编辑器等放 global；仓库特有的设置（如某个仓库使用不同的用户名）放 local。

3.6　3.6 实用的全局配置

```
git config --global init.defaultBranch main        # 默认分支名 main
git config --global core.editor "code --wait"      # 默认编辑器 VS Code
git config --global pull.rebase false              # pull 默认 merge（保守）
git config --global push.default simple            # 只推送当前分支
git config --global color.ui auto                  # 彩色输出
git config --global diff.tool vscode               # diff 工具
git config --global core.quotepath false           # 中文文件名正常显示
```

其中 `core.quotepath false` 对中文路径/文件名尤为重要，否则 `git status` 会显示转义字符。

第 4 章　第 4 章日常基本操作

本章用一个虚拟的生信小项目走完基本流程。先建立练习目录（以下命令在 Git Bash 或终端中执行）：

```
mkdir -p ~/projects/rnaseq-demo && cd ~/projects/rnaseq-demo
```

4.1　4.1 初始化与克隆

初始化：把当前目录变成 Git 仓库。

```
git init
# Initialized empty Git repository in .../rnaseq-demo/.git/
```

克隆：从远程复制仓库（含全部历史）到本地。克隆生信工具仓库示例：

```
git clone https://github.com/snakemake/snakemake.git
git clone git@github.com:kael/rnaseq-demo.git
```

4.2　4.2 查看状态与差异

创建一个脚本：

```
echo '#!/bin/bash
# RNA-seq 质控脚本 v1
echo ”运行 FastQC...”
fastqc data/sample1.fastq.gz -o results/qc/' > run_qc.sh
```

随时查看仓库状态：

```
$ git status
On branch main
```

```
No commits yet

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

untracked（未跟踪）意味着 Git 还不管理这个文件。`git diff` 查看**已跟踪文件**工作区与暂存区的差异；`git diff --staged` 查看暂存区与上次提交的差异。

4.3　4.3 暂存与提交

```
git add run_qc.sh          # 加入暂存区
git status                 # 文件变为绿色 "new file"
git commit -m "添加 RNA-seq 质控脚本 v1"
```

提交信息规范：单行提交信息应是一句祈使句，说明"做了什么"，而非"改了哪些文件"。生信项目推荐在提交信息中附上关键参数或链接（如数据库版本）：

```
git commit -m "更新 HISAT2 索引为 GRCh38 最新版（2024-10 发布）"
```

常见格式为 `<类型>: <简要说明>`，如 `fix: 修正质控阈值`, `feat: 新增差异分析模块`, `docs: 更新 README`（详见 8.2 节）。

只提交部分改动：一次分析常混着多个逻辑改动，用 `git add -p` 交互式选择要暂存的片段，保持每个提交单一职责：

```
git add -p run_qc.sh
```

4.4　4.4 查看历史

```
git log --oneline                    # 简洁列表
git log --oneline --graph --all      # 分支拓扑图
git log -3                           # 最近 3 条
git log --follow -p run_qc.sh        # 追踪某文件的完整演化
git show <commit>                    # 查看某次提交详情
```

```
git blame run_qc.sh                    # 每一行最后是谁改的（排查参数来源神器）
```

4.5 4.5 撤销与回退

Git 的撤销操作按影响范围从小到大排列：

命令	场景	影响
`git restore <file>`	丢弃工作区未暂存的修改	工作区文件回到暂存区/HEAD 状态
`git restore --staged <file>`	把文件移出暂存区（取消 add）	暂存区回退，工作区保留
`git reset --soft HEAD~1`	撤销上次提交，但保留改动在暂存区	只移动分支指针
`git reset --mixed HEAD~1`	撤销提交并取消暂存	改动保留在工作区
`git reset --hard HEAD~1`	彻底回退并丢弃改动	**危险**，改动丢失
`git revert <commit>`	撤销某次已推送的提交	生成一个”反向提交”，历史保留
`git stash push` / `git stash pop`	临时搁置未提交的改动	保存到 stash，可随时取回

生信实战：`reset --hard` 是误删数据的常见元凶。默认分支未推送前可用 `reset`；**已推送的提交请用 `revert`**，避免与协作者的历史分叉。

4.6 4.6 .gitignore：生信仓库的第一道防线

`.gitignore` 声明哪些文件 Git 永不追踪。生信项目的典型模板：

```
# 数据文件（大文件不入库）
data/
*.fastq
*.fastq.gz
*.fq.gz
*.bam
*.bam.bai
*.sam
*.vcf.gz
*.bcf
```

```
# 中间结果与缓存
results/
outputs/
.snakemake/
.nextflow/
__pycache__/
*.pyc
.ipynb_checkpoints/

# R
.Rhistory
.RData
.Rproj.user/

# 环境与凭据
envs/*/
!envs/*.yml
.env
*.key
*.pem

# 系统文件
.DS_Store
Thumbs.db
```

规则要点：

- 每行一个模式；`/` 开头表示仅匹配仓库根；`目录/` 表示整目录；
- `!` 表示排除（重新纳入），且不能越过其父目录的忽略；
- 目录中已有`.gitignore` 的，子目录规则独立生效；
- 用 `git check-ignore -v results/x.txt` 调试某文件为何被忽略。

  提示：`git add .` 前务必 `git status` 确认没有把 `data/`、`.bam` 等大文件误入暂存区。

4.7　4.7 标签：标记版本

标签（tag）为历史打上里程碑，常用于标记论文提交版本或软件发布：

```
git tag -a v1.0 -m ” 论文投稿配套代码版本”      # 附注标签（推荐）
git tag v1.0-lite                                # 轻量标签
git push origin v1.0                             # 推送标签
git tag -l ”v*”                                  # 列出标签
```

配合 GitHub Release / Zenodo 可以生成 DOI，详见 6.7 节。

第 5 章　第 5 章分支与协作

5.1　5.1 分支的基本操作

```
git branch                    # 列出分支
git branch feature/fastqc     # 创建分支
git switch feature/fastqc     # 切换分支（老版本：git checkout）
git switch -c feature/fastqc  # 创建并切换
git branch -d feature/fastqc  # 删除已合并分支
git branch -D feature/fastqc  # 强制删除（丢弃改动）
```

生信最佳实践：每个分析任务开一个分支（如 `feature/deg`、`fix/ref-genome`），任务完成、结果确认后再合并回 `main`。这样 `main` 始终处于”可发布”状态，实验性改动不会污染主线。

5.2　5.2 合并：merge

```
git switch main
git merge feature/fastqc
```

fast-forward 与三方合并：若 `main` 在分支创建后没有新提交，合并只是指针前移（fast-forward）；若两侧都有新提交，Git 执行三方合并并生成一个合并提交：

```
     A---B---C  feature/fastqc
    /
main---D---E---M   (合并提交 M)
```

保留合并拓扑、便于团队理解时使用 `--no-ff`：

```
git merge --no-ff feature/fastqc -m ”合并质控模块”
```

5.3 5.3 rebase 与 merge 的取舍

rebase 把分支上的提交”搬运”到另一条分支的顶端，得到线性历史：

```
git switch feature/fastqc
git rebase main          # 把 feature 的提交变基到 main 之上
```

```
          A'---B'---C'  feature/fastqc（重写后的提交）
         /
main---D---E
```

- 什么时候用 **merge**：共享分支（如 `main`）的常规合入、需要保留合并历史的场景。对新手最安全。
- 什么时候用 **rebase**：整理本地未推送的提交、让 feature 分支保持最新。**绝不要对已推送的共享提交 rebase**，否则会重写历史，导致协作者仓库分叉（7.3 节详述）。
- 折中方案：`git pull --rebase` 拉取远端更新，把本地提交置于远端提交之上，避免产生多余的合并提交。

生信团队建议：小团队（1–5 人）使用 `main` + 短生命周期 feature 分支 + merge（必要时 `--no-ff`）即可，简单清晰；不要过早引入复杂的 Git Flow。

5.4 5.4 冲突的解决

当两侧修改了同一文件的同一区域时，合并/变基会产生冲突：

```
$ git merge feature/deg
Auto-merging scripts/deg.R
CONFLICT (content): Merge conflict in scripts/deg.R
```

打开文件，Git 用标记标出冲突：

```
<<<<<<< HEAD
padj_cutoff <- 0.05
=======
padj_cutoff <- 0.01
>>>>>>> feature/deg
```

解决步骤：

1. 手动编辑文件，保留正确版本，删除 `<<<<<<<`、`=======`、`>>>>>>>` 三行标记；

2. `git add scripts/deg.R` 标记为已解决；
3. `git commit` 完成合并（rebase 冲突则 `git rebase --continue`）。

> **注意**：冲突解决需要你理解上下文，不能盲目”两边都留”。涉及参数阈值的冲突，务必回到实验设计确认，必要时咨询合作者。

5.5　5.5 远程仓库：remote、push、pull、fetch

```
git remote add origin git@github.com:kael/rnaseq-demo.git   # 关联远程
git remote -v                                               # 查看远程
git push -u origin main                                     # 首次推送并绑定
git push                                                    # 之后直接推送
git fetch                                                   # 只下载远程更新
git pull                                                    # fetch + merge
```

push 被拒绝怎么办：远程已有别人推送的提交时，Git 会拒绝。正确流程：

```
git pull --rebase        # 先把远端提交并入本地
# 解决冲突（如有）
git push
```

pull 与 fetch 的区别：`fetch` 只更新本地远程跟踪分支（`origin/main`），不动工作区；`pull` 还会尝试合并，可能产生意外冲突。**先 fetch 再决定如何合并**，是更可控的习惯。

5.6　5.6 Pull Request / Merge Request 流程

在 GitHub（Pull Request）或 GitLab（Merge Request）上的标准协作流程：

1. 在组织仓库点 **Fork** 或新建 feature 分支；
2. 本地 `git clone` / `git switch -c feature/xxx`；
3. 提交并推送：`git push -u origin feature/xxx`；
4. 在网页上发起 PR/MR，写清改动目的、分析结果截图、复现命令；
5. 协作者 review、留言、请求修改，你在 feature 分支继续提交（PR 会自动更新）；
6. 审核通过后合并，删除 feature 分支。

生信项目 PR 的加分项：附上关键结果对比（如质控前后指标表）、运行环境说明、预期运行时长，便于 reviewer 评估。

5.7　5.7 团队工作流选型

工作流	特点	适用场景
GitHub Flow	一个 `main` + 短 feature 分支 + PR	小型生信团队、个人项目（推荐）
Git Flow	`main`/`develop` + `feature`/`release`/`hotfix` 多分支	需要正式版本发布的软件项目
GitLab Flow	环境分支（`staging`/`production`）与功能分支	有部署环节的流程型项目

生信分析项目（而非软件产品）通常选择 GitHub Flow：分支少、合入门槛低、迭代快。

第 6 章　第 6 章生信项目的 Git 实战

6.1　6.1 项目目录结构设计

“代码与数据分离” 是生信 Git 项目的第一原则。推荐结构：

```
rnaseq-project/
├── README.md                # 项目说明：目标、方法、快速开始（入库）
├── LICENSE                  # 许可证（入库）
├── .gitignore               # 忽略数据/结果/缓存（入库）
├── .gitattributes           # 换行符与二进制声明（入库）
├── config/
│   ├── config.yaml          # 分析参数（入库）
│   └── samples.tsv          # 样本清单（入库，小文本）
├── workflow/
│   ├── Snakefile            # 工作流定义（入库）
│   └── rules/               # 规则脚本（入库）
│       ├── qc.smk
│       └── quant.smk
├── scripts/                 # 独立脚本（入库）
│   ├── run_qc.sh
│   └── plot_dge.R
├── envs/
│   └── env.yaml             # conda 环境定义（入库）
├── data/                    # 原始数据（不入库，见 .gitignore）
│   └── raw/                 # 只读，禁止修改
├── results/                 # 分析结果（不入库）
└── docs/                    # 文档（入库）
```

6.2　6.2 工作流脚本入库：Snakemake / Nextflow

工作流文件是纯文本，是 Git 追踪的理想对象。Snakemake 项目示例：

```
# Snakefile（入库）
rule fastqc:
    input: "data/sample1.fastq.gz"
    output: "results/qc/sample1_fastqc.html"
    conda: "envs/env.yaml"
    shell: "fastqc {input} -o results/qc/"
```

配套.gitignore 需要忽略工作流运行时产物：

```
.snakemake/
.nextflow/
work/
.launch/
```

关键实践：

- 用 `git tag v1.0` 标记"本次分析使用的完整工作流版本"，与论文方法部分对应；
- 每次运行前 `git status` 确认工作流与参数文件已提交（可在 Snakefile 中加 `version:` 字段）；
- 参数变更（如参考基因组版本、阈值）应伴随独立的提交，便于事后定位"哪个参数对应哪个结果"；
- Nextflow 项目同理：`nextflow.config`、`main.nf`、`modules/` 入库，`.nextflow/` 与 `work/` 忽略。

6.3　6.3 环境与配置入库

可重复分析 = 代码 + 配置 + **环境**。环境本身（conda 环境目录、Docker 镜像）不入库，但**环境定义文件**必须入库：

```
# envs/env.yaml（入库）
name: rnaseq
channels:
  - conda-forge
  - bioconda
dependencies:
  - python=3.11
  - fastqc=0.12.1
  - hisat2=2.2.1
  - snakemake>=7.0
```

```
# 在其他机器/服务器复现环境
conda env create -f envs/env.yaml
```

同理，`requirements.txt`、`Dockerfile`、`.Rprofile` 都入库。**生信环境应尽量固定版本号**（`fastqc=0.12.1` 而非 `fastqc`），这比代码本身更常导致不可复现。

6.4　6.4 结果何时入库

- **不入库**：中间结果（比对、定量输出、临时表格）、超大文件、可再生成的任何东西——它们会撑爆仓库，且没有版本管理价值。
- **入库**：论文图表、最终报告、需要与代码一起发布的精简表格（如差异基因列表`.tsv`，若体积在几十 MB 以内）。
- **折中**：把”生成结果的命令”入库（工作流/脚本），结果本身留在服务器或对象存储；在 `README` 中写清楚如何重新生成。

判断标准一句话：**如果重跑工作流能原样得到它，就不该入库。**

6.5　6.5 与 R / Python 工具集成

- **RStudio**：Git 面板可视化暂存/提交/历史，适合 R 用户；`.Rproj` 入库，`.Rhistory`、`.RData`、`.Rproj.user/` 忽略。
- **VS Code**：内置 Git 侧边栏 + diff 视图，支持冲突解决界面；配合 GitLens 插件查看 blame/历史。
- **Jupyter**：`.ipynb` 是 JSON 文本，可以入库，但输出（base64 图片、大数据）会造成 diff 噪音。建议：`pip install nbstripout`，用 pre-commit 钩子自动清空输出（7.5 节），或使用 `jupytext` 同时保存`.py` 纯文本版本。
- **服务器（无图形界面）**：命令行操作 + `git` 别名（7.4 节）提高效率。

6.6　6.6 Git LFS 与大规模数据

当某些”半大”文件（几十 MB 到几 GB）确实需要版本管理时（如论文配套的比对结果、VCF），使用 **Git LFS**（Large File Storage）：文件本体存在 LFS 服务器，仓库只存指针。

```
git lfs install
git lfs track "*.bam" "*.vcf.gz" "*.bigWig"
git add .gitattributes     # LFS 规则写入 .gitattributes，随仓库分发
```

```
git add results/ && git commit -m "添加比对结果（LFS）"
```

注意事项：

- LFS 有存储与流量配额（GitHub 免费额度约 1 GB 存储 + 每月流量），**原始测序数据仍不应入库**；
- 拉取 LFS 仓库需安装 `git-lfs`，否则只会得到指针文件；
- 替代方案：数据放服务器目录或对象存储（AWS S3、阿里云 OSS、Zenodo），仓库只记录校验值（`md5sum`）与下载命令——这通常是生信项目更合适的方案。

6.7　6.7 论文配套代码发布

论文投稿与发表时，配套代码的发布流程：

1. 在 GitHub 建立公开仓库，`main` 分支处于可复现状态；
2. 打标签：`git tag -a v1.0 -m "论文配套版本" && git push origin v1.0`；
3. 关联 **Zenodo**（https://zenodo.org）：Zenodo 会在每次 release 时自动归档并分配 **DOI**，论文引用该 DOI 即可满足"代码可访问"要求；
4. 仓库包含 `README.md`（快速开始、环境、数据获取方式）、`LICENSE`、`environment.yml`；
5. 在论文方法部分写明：仓库地址、DOI、运行环境与关键参数。

示例 README 骨架：

```
# 论文配套分析代码：XXX 项目
- 论文：<DOI>
- 数据：GEO GSE123456（预处理后数据见 data/README）
- 环境：conda env create -f envs/env.yaml
- 复现：snakemake -j 8 --use-conda
- 结果：results/（生成后约 2.1 GB）
```

第 7 章　第 7 章高级技巧与踩坑指南

7.1　7.1 生信常见坑

坑 1：二进制文件入库后仓库失控。`.bam`、`.xlsx`、`.ipynb` 输出等一旦入库，Git 每次都整份保存新版本，仓库体积随提交数线性膨胀。补救：用 `git lfs migrate import`（仅限未共享的历史），或使用 `git filter-repo` 重写历史（7.3 节）。**预防远胜于补救。**

坑 2：行尾符导致”整文件被修改”。`git diff` 显示某个`.sh` 文件每行都被改——几乎可以确定是 `CRLF/LF` 问题。修复：按 3.3 节配置`.gitattributes` 后执行：

```
git add --renormalize .
```

坑 3：文件名大小写。macOS/Linux 区分大小写，Windows 默认不区分。`Samples.tsv` 改名为 `samples.tsv` 在 Windows 上可能不被 Git 察觉。建议：核心文件统一小写下划线命名（`samples.tsv`、`config.yaml`），并设置 `git config core.ignorecase false`。

坑 4：符号链接（symlink）。`data -> /mnt/rawdata` 这类软链接，Git 存储的是链接本身而非目标内容。若目标路径在别的机器上不存在，检出即断链。生信项目尽量避免软链接入库，或在 README 中说明路径假设。

坑 5：把密钥/凭据入库。`.env`、数据库密码、`id_rsa` 一旦推送到公共仓库即视为泄露。用`.gitignore` 排除；已在历史中的，用 `git filter-repo` 清除，并立即轮换密钥（注意 GitHub 会扫描并警告）。

7.2　7.2 reflog：找回”丢失”的提交

误 `reset --hard`、误删分支后不要慌：

```
git reflog
# abc1234 HEAD@{0}: reset: moving to HEAD~2
# 9e1c0d8 HEAD@{1}: commit: 更新质控阈值
git branch rescue 9e1c0d8     # 用分支指针救回该提交
git switch rescue             # 回到被误删的状态
```

reflog 默认保留 90 天（`gc.reflogExpire`），在此期间几乎所有误操作都可恢复。**重要提示：**reflog 是本地的，克隆出来的新仓库看不到原仓库的 reflog；误删后不要立即执行 `git gc`。

7.3 7.3 历史重写：谨慎行事

操作	命令	风险
修改上次提交	`git commit --amend`	只改未推送的提交
交互式变基	`git rebase -i HEAD~3`	合并/重排/改写多个提交
批量改写	`git filter-repo`	全历史重写，**只用于未共享的仓库**

铁律：已推送到共享分支的历史不要重写。重写会导致协作者下次 `pull` 时出现无法自动解决的冲突，需要 `git pull --force` 等危险操作。生信项目如需修改历史，请确认所有协作者都知情并同步。

7.4 7.4 效率技巧：别名与补全

```
git config --global alias.st ”status -sb”
git config --global alias.lg ”log --oneline --graph --all --decorate”
git config --global alias.unstage ”restore --staged”
git config --global alias.last ”log -1 HEAD”
git config --global alias.ds ”diff --stat”

git st          # 一行状态
git lg          # 拓扑历史图
```

安装 Bash 补全（Linux/macOS）：`source /usr/share/bash-completion/completions/git`；Windows Git Bash 默认已启用。

7.5 7.5 pre-commit：自动化质量检查

pre-commit 框架在每次提交前自动运行检查，是生信项目低成本提质的利器：

```
# .pre-commit-config.yaml（入库）
repos:
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v5.0.0
```

```
    hooks:
      - id: trailing-whitespace
      - id: end-of-file-fixer
      - id: check-yaml
  - repo: https://github.com/psf/black
    rev: 24.10.0
    hooks:
      - id: black
  - repo: https://github.com/kynan/nbstripout
    rev: 0.8.1
    hooks:
      - id: nbstripout
```

安装启用：

```
pip install pre-commit
pre-commit install
```

之后每次 `git commit` 自动执行检查，失败则阻止提交。Jupyter 用户强烈建议启用 `nbstripout`，避免 `.ipynb` 输出入库。

7.6　7.6 子模块：慎用

`git submodule add <url> path/` 可以把另一个仓库嵌套进当前仓库。生信场景（如引用某个工具仓库）看似合理，但子模块的更新/冲突管理复杂，容易让团队陷入” 更新了子模块但别人不知道” 的混乱。

替代方案：工具依赖用 conda/pip 管理版本（记录在 `environment.yml`），分析流程依赖用 Snakemake 的 `container:` 或 Nextflow 的 module 体系，通常都比子模块更适合生信项目。只有需要精确定位” 某个工具某次提交版本” 时，才考虑子模块（或更简单的：记录 commit hash 到 README/配置中）。

7.7　7.7 Git 与 conda 环境的配合

原则：环境不入库，环境定义入库。 conda 环境目录体积大、路径含机器信息（`/home/kael/anaconda3/envs/rnaseq`），入库毫无价值且会破坏可移植性。

推荐组合拳：

```
conda env create -f envs/env.yaml            # 建环境
conda env export -n rnaseq > envs/env.lock.yaml   # 导出精确版本（可选入库）
```

`env.lock.yaml` 记录全部包的精确版本，作为” 复现基准”；日常开发修改 `env.yaml` 并提交。Snakemake 的 `--use-conda` 会自动按 rule 级 `conda:` 声明创建独立环境，配合 Git 追踪工作流，即为完整的可复现方案。

第 8 章　第 8 章开源参与与协作规范

8.1　8.1 在 GitHub 上参与生信开源项目

生信生态（Bioconda、samtools、bedtools、Snakemake、Nextflow、Seurat 等）高度依赖开源协作。参与流程：

1. **Fork** 目标仓库到自己的账号；
2. 克隆并关联上游：

```
git clone git@github.com:< 你 >/< 仓库 >.git
cd < 仓库 >
git remote add upstream git@github.com:< 官方 >/< 仓库 >.git
git remote -v
```

3. 同步上游最新代码：

```
git fetch upstream
git switch -c fix/xxx upstream/main    # 从上游最新状态开分支
```

4. 开发、提交、推送后发起 Pull Request；
5. 按 reviewer 意见修改：继续在 feature 分支提交推送，PR 自动更新。

参与建议：先从文档修正、bug 报告、测试补全入手；PR 前阅读该项目的 `CONTRIBUTING.md`，运行其规定的测试。

8.2　8.2 提交信息规范：Conventional Commits

生信团队统一提交信息格式，可让历史日志变得可检索、可自动化：

```
<type>(<scope>): <subject>

<body（可选）>
```

type	含义	生信示例
`feat`	新功能	`feat: 新增 DESeq2 差异分析规则`
`fix`	修复	`fix: 修正样本名去重 bug`
`docs`	文档	`docs: 补充复现步骤`
`refactor`	重构	`refactor: 拆分质控规则为独立模块`
`chore`	杂务	`chore: 升级 snakemake 至 8.x`
`perf`	性能	`perf: 并行化比对步骤`

8.3 8.3 issue、review 与文档文化

- **issue 是科学记录的一部分**：写清复现命令、环境、期望与实际输出，附 `git log` 与配置文件。
- **review 对事不对人**：审查关注" 能否复现、是否合理"，建议引用具体行号（`git blame` 定位）。
- **README 是项目的门面**：包含项目目的、快速开始、数据获取、环境安装、许可证。

8.4 8.4 许可证与署名

发布代码前选择许可证：MIT/Apache-2.0（宽松）、BSD-3-Clause（学术常用）、GPL-3.0（copyleft，要求衍生作品开源）。生信常用工具多为 MIT 或 GPL。论文配套代码建议 MIT/BSD 以最大化复用。署名问题（作者顺序、贡献者列表）应写入 `README` 或 `CONTRIBUTORS` 文件，Git 的提交历史恰好提供了客观的贡献记录。

附录 A　常用命令速查表

A.1　A.1 配置

命令	作用
`git config --global user.name ”名字”`	配置用户名
`git config --global user.email ”邮箱”`	配置邮箱
`git config --global --list`	查看全局配置
`git config core.quotepath false`	中文文件名正常显示

A.2　A.2 仓库操作

命令	作用
`git init`	初始化仓库
`git clone <url>`	克隆远程仓库
`git status`	查看状态
`git remote -v`	查看远程
`git remote add origin <url>`	关联远程仓库

A.3　A.3 提交

命令	作用
`git add <file>`	暂存文件
`git add -p`	交互式暂存片段
`git commit -m ”说明”`	提交
`git commit --amend`	修改上次提交
`git diff` / `git diff --staged`	查看工作区/暂存区差异
`git log --oneline --graph`	查看历史拓扑

A.4 A.4 撤销与回退

命令	作用
`git restore <file>`	丢弃工作区修改
`git restore --staged <file>`	取消暂存
`git reset --soft/mixed/hard HEAD~1`	按程度回退
`git revert <commit>`	反向提交（安全撤销已推送提交）
`git stash push / pop / list`	暂存/取回/查看搁置改动
`git reflog`	查看引用历史（救援）

A.5 A.5 分支与合并

命令	作用
`git branch`	列出分支
`git switch -c <name>`	创建并切换分支
`git switch -`	切回上一个分支
`git merge <branch>`	合并分支
`git rebase <branch>`	变基
`git merge --abort / git rebase --abort`	放弃合并/变基

A.6 A.6 远程协作

命令	作用
`git fetch`	下载远程更新（不合并）
`git pull --rebase`	拉取并变基（推荐）
`git push -u origin <branch>`	推送并绑定上游
`git push origin --delete <branch>`	删除远程分支
`git tag -a v1.0 -m ” 说明”`	打附注标签

附录 B　生信.gitignore 完整模板

```
# ========== 数据与原始文件（不入库） ==========
data/
raw/
*.fastq
*.fastq.gz
*.fq.gz
*.bam
*.bam.bai
*.sam
*.cram
*.vcf
*.vcf.gz
*.bcf
*.bw
*.bigWig
*.bigBed

# ========== 中间结果与运行产物 ==========
results/
outputs/
work/
.snakemake/
.nextflow/
*.log
*.tmp
*.temp

# ========== Python ==========
__pycache__/
*.py[cod]
.venv/
venv/
.ipynb_checkpoints/

# ========== R ==========
```

```
.Rhistory
.RData
.Rproj.user/
.Rcheck/

# ========== 环境与凭据（定义文件保留） ==========
envs/*/
!envs/*.yaml
!envs/*.yml
.env
*.key
*.pem

# ========== 编辑器与系统 ==========
.vscode/
.idea/
.DS_Store
Thumbs.db
```

附录 C　最佳实践清单

入门阶段

- ☐ 配置 `user.name` 与 `user.email`（与 GitHub 账号一致）
- ☐ 设置 `core.autocrlf` 并添加`.gitattributes`
- ☐ 新建仓库时先写`.gitignore` 再 `git add .`
- ☐ 提交信息用祈使句、一事一提交

项目阶段

- ☐ 采用” 代码入库、数据在外、环境入定义” 的结构
- ☐ 工作流与配置文件入库，中间结果不入库
- ☐ 每次分析运行前 `git status` 确认状态干净、参数已提交
- ☐ 用标签标记论文/发布版本
- ☐ 大文件用 Git LFS 或对象存储，绝不直接入库

协作阶段

- ☐ 一任务一分支，完成后合并回 `main`
- ☐ 已推送的提交用 `revert` 撤销，不用 `reset`
- ☐ 拉取用 `git pull --rebase`
- ☐ PR/MR 附复现命令与结果截图
- ☐ 发布前确认 README、LICENSE、环境文件齐全，关联 Zenodo DOI

参考文献

1. Chacon S, Straub B. *Pro Git* (2nd ed.). Apress, 2014. 开源在线版：https://git-scm.com/book/zh/v2
2. Git 官方文档. https://git-scm.com/doc
3. Bryan J. *Happy Git and GitHub for the userR*. https://happygitwithr.com
4. Sandve GK, et al. Ten simple rules for reproducible computational research. *PLoS Comput Biol*, 2013, 9(10): e1003285.
5. Köster J, Rahmann S. Snakemake—a scalable bioinformatics workflow engine. *Bioinformatics*, 2012, 28(19): 2520–2522.
6. Di Tommaso P, et al. Nextflow enables reproducible computational workflows. *Nat Biotechnol*, 2017, 35: 316–319.
7. Git Large File Storage 文档. https://git-lfs.com
8. Zenodo——研究输出归档服务. https://zenodo.org
9. pre-commit 框架文档. https://pre-commit.com
10. Conventional Commits 规范. https://www.conventionalcommits.org/zh-hans