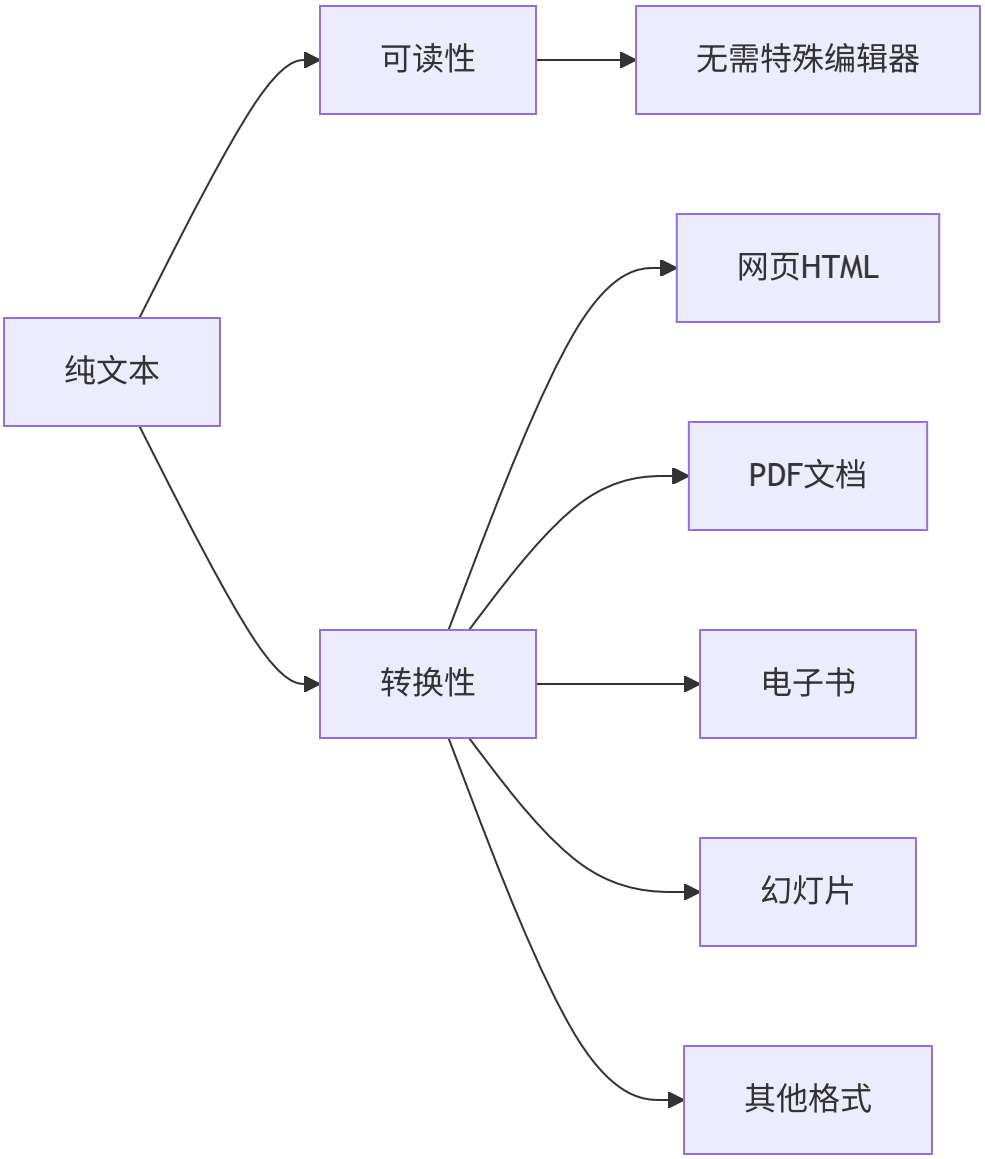
# 开发文档编写与版本控制基础

## 1. Markdown编写开发文档

Markdown 是一种用简单语法代替复杂排版的标记语言，专为网络写作而设计。它能让你专注于内容创作，而无需分心调整格式。Markdown文档用纯文本编写，能自动转换成精美排版的格式（如HTML/PDF）。

# 1.1. Markdown的基本特征

1. **诞生背景**
   - 2004年由John Gruber创建
   - 目标：实现「易读易写」的纯文本格式
   - 哲学：内容与呈现分离
2. **特点**：
   - 文件扩展名为 `.md` 或 `.markdown`
   - 兼容所有文本编辑器
   - 被GitHub、知乎、简书等平台原生支持
   - 比Word减少80%的格式调整时间
   - 代码与文档混合编写时尤其高效
   - GitHub/Gitee的README文件标准格式
   - 技术文档的首选（如Vue.js官方文档）
3. **技术特征**



4. **标准体系**
    - **CommonMark**：标准化版本
    - **GFM**：GitHub Flavored Markdown
    - **扩展语法**：各平台定制（如表格、任务列表）

## 1.2. 核心应用场景

1. **技术文档编写**
    - **代码项目文档**：GitHub/Gitee 的 `README.md` 文件
    - **API文档**：结合 Swagger 或 MkDocs 生成交互式文档
2. **知识管理与笔记**
    - **个人知识库**：Obsidian/Notion 等工具的原生格式
    - **团队协作**：语雀/飞书文档的底层存储格式
    - **优势**：纯文本易于版本控制（Git友好）
3. **学术写作**
    - **论文草稿**：通过 Pandoc 转换为 LaTeX 或 Word
    - **数学公式支持**：

```
$$
\frac{\partial f}{\partial t} = \nabla^2 f
$$
```

$$\frac{\partial f}{\partial t} = \nabla^2 f$$

4. **网络内容发布**
    - **博客平台**：Hexo/Hugo 静态网站生成器
    - **CMS系统**：WordPress 等支持 Markdown 插件

## 1.3. 企业级应用价值

1. **文档即代码（Docs as Code）**
    - 与开发工具链深度集成
    - 示例架构：

```
docs/
├── README.md
├── api/
│   ├── rest.md
│   └── graphql.md
└── Makefile  # 自动化构建
```

2. **成本效益分析**

| 指标 | Markdown方案 | 传统WYSIWYG方案 |
| --- | --- | --- |
| 维护成本 | 低 | 高 |
| 协作效率 | 高 | 中 |
| 长期可读性 | 极高 | 依赖特定软件 |

# 1.4. Markdown基本语法

| 形式 | 或者 | 或者 |
| --- | --- | --- |
| `*Italic*` | `_Italic_` | *Italic* |
| `**Bold**` | `__Bold__` | **Bold** |
| `# Heading 1` | `Heading 1`<br>`=========` | # Heading 1 |
| `## Heading 2` | `Heading 2`<br>`---------` | ## Heading 2 |
| `[Link](http://a.com)` | [Link][1]<br>⋮<br>[1]: http://b.org | Link |
| `![Image](http://url/a.png)` | `![Image][1]`<br>⋮<br>[1]: http://url/b.jpg |  |
| `> Blockquote` | | > Blockquote |

| 形式 | 或者 | 或者 |
| --- | --- | --- |
| `* List`<br>`* List`<br>`* List` | `- List`<br>`- List`<br>`- List` | • List<br>• List<br>• List |
| `1. One`<br>`2. Two`<br>`3. Three` | `1) One`<br>`2) Two`<br>`3) Three` | 1. One<br>2. Two<br>3. Three |
| `Horizontal rule:`<br>`---` | `Horizontal rule:`<br>`***` | Horizontal rule: |
| `Inline code` with backticks | | `Inline code` with backticks |
| ```<br># code block<br>print '3 backticks or'<br>print 'indent 4 spaces'<br>``` | ····# code block<br>····print '3 backticks or'<br>····print 'indent 4 spaces' | # code block<br>print '3 backticks or'<br>print 'indent 4 spaces' |

# 1.5. Markdown入门

1. **学习资源**
   - CommonMark教程
   - 交互式教程：Markdown Tutorial
   - 速查表：Markdown Cheatsheet
   - 增强Markdown语法：Markdown Preview Enhanced
   - Mermaid学习：Mermaid
   - Katex参考：Katex
2. **工具推荐**
   - 平台：VS Code
   - 插件：Markdown Preview Enhanced

# 1.6. 最佳实践建议

1. **风格指南**
   - 标题层级不超过3级

- 中英文混排留空格
- 列表项统一使用 -

2. **版本控制友好**
   - 每行不超过80字符
   - 用 `<!-- comment -->` 添加注释
3. **扩展语法慎用**
   - 优先使用CommonMark标准
   - 平台特有语法需显式标注

# 2. 版本控制工具

版本控制（Version Control）是一种记录和管理文件或代码变更的系统，允许用户追踪历史修改、协作开发、回滚到旧版本等。它是软件开发、文档管理等领域的重要工具。

## 2.1. 版本控制的类型

版本控制系统（VCS）主要分为以下三类：

1. **本地版本控制系统（Local VCS）**
   - **特点**：所有版本数据存储在本地计算机上。
   - **例子**：RCS（Revision Control System）。
   - **缺点**：无法团队协作，存在单点故障风险（如硬盘损坏）。
2. **集中式版本控制系统（Centralized VCS, CVCS）**
   - **特点**：
     - 单台中央服务器存储所有版本历史。
     - 用户通过客户端从服务器获取最新代码或提交更改。
   - **例子**：SVN（Subversion）、CVS（Concurrent Versions System）。
   - **优点**：支持团队协作，权限管理方便。
   - **缺点**：
     - 中央服务器是单点故障（若宕机则无法协作或恢复历史版本）。
     - 所有操作依赖网络连接。
3. **分布式版本控制系统（Distributed VCS, DVCS）**
   - **特点**：
     - 每个用户的本地仓库都是完整的副本（包含全部历史记录）。
     - 无需始终连接中央服务器（支持离线操作）。
     - 通过“推送”（push）和“拉取”（pull）同步变更。
   - **例子**：Git、Mercurial、Bazaar。

- **优点**：
  - 去中心化，无单点故障。
  - 分支管理高效，适合大规模协作（如开源项目）。
- **缺点**：学习曲线较陡（尤其是Git）。

**关键区别对比**

| 特性 | 本地VCS | 集中式VCS（如SVN） | 分布式VCS（如Git） |
| --- | --- | --- | --- |
| **数据存储位置** | 本地 | 中央服务器 | 每个用户都有完整副本 |
| **是否需要网络** | 否 | 提交/更新时需要 | 仅同步时需要 |
| **单点故障风险** | 有（本地损坏） | 有（服务器宕机） | 无 |
| **分支/合并效率** | 低 | 低（分支代价高） | 高（轻量级分支） |
| **典型工具** | RCS | SVN、CVS | Git、Mercurial |

**补充说明**

- **Git**：目前最流行的DVCS，由Linus Torvalds在 2005 年开发，用于管理Linux内核代码。支持非线性开发（如多分支并行），彻底改变了软件开发中的协作方式。
- **SVN**：集中式代表工具，适合需要严格权限控制的企业环境。
- **现代趋势**：分布式系统（如Git）已成为主流，尤其配合GitHub、GitLab等平台，极大提升了协作效率。

# 2.2. git 工作基础概念

Git 是一个分布式版本控制工具，用于管理代码仓库，配合代码托管平台（如 Gitee 或 GitHub）能高效管理代码协作。作为版本控制系统，它可以记录文件所有历史变更，支持多人并行开发，可回溯到任意历史版本。据 2023 年 Stack Overflow 调查，**87%** 的开发者使用 Git，它已成为软件开发的空气和水般的存在。
它的工作主体分为**本地仓库**和**远程仓库**。

## 2.2.1. 本地仓库

**本地仓库**是 Git 在**你的计算机上**存储项目完整历史记录的地方，位于项目根目录的隐藏文件夹 `.git` 中。它包含所有版本信息、分支、提交记录，并允许你在离线状态下工作。

- **广义的“本地仓库”**：指的是整个项目文件夹，它包括 工作区、暂存区 和 版本库（.git目录）。
- **狭义的“仓库”**：特指那个存储历史的 .git 目录。

**本地仓库的核心组成**


- `HEAD`：指向当前所在的**分支或提交**（如 `refs/heads/main`）。
- **对象数据库（Objects）**：
  - `Blob`：存储文件内容。
  - `Tree`：记录目录结构和文件名。
  - `Commit`：保存提交信息、作者、时间戳等。
- **引用（Refs）**：
  - 分支（`refs/heads/`）：如 `main`、`feature`。
  - 标签（`refs/tags/`）：如 `v1.0`。
- **暂存区（Index）**：临时存储准备提交的改动。

## 2.2.2. 远程仓库

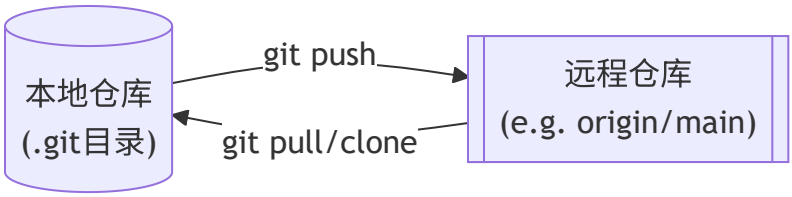
远程仓库是存储在云端（如 GitHub、GitLab、Gitee）或局域网服务器上的中央代码库，用于团队协作和备份。

**远程仓库的核心作用**

1. **备份代码**：防止本地数据丢失。

2. **团队协作**：多人通过同一个远程仓库同步代码。
3. **版本管理**：统一保存所有分支和提交历史。

它与本地仓库的关系如下：




## 2.2.3. 工作区、暂存区、分支、HEAD及分支指针

当用户在自己的计算机上工作时，需要了解 Git 的三种核心概念，它们共同构成了 Git 的工作流程。暂存区和分支由本地仓库进行管理。理解这三者的关系，是掌握 Git 的基础！

### 1. 工作区（Working Directory）

- **定义**：工作区是你的**项目目录**，也就是你在本地电脑上直接编辑的文件和文件夹。
- **特点**：
  - 所有修改（新增、编辑、删除文件）最初都发生在工作区。
  - 工作区的改动不会自动被 Git 记录，必须手动通过 `git add` 将工作区的修改添加到**暂存区**。
- **常用命令**：

```
git status       # 查看工作区的文件状态（未跟踪/已修改）
git add <file>   # 将工作区的改动添加到暂存区
git restore <file> # 丢弃工作区的修改（恢复到上次提交状态）
```

### 2. 暂存区（Staging Area / Index）

- **定义**：暂存区是一个中间区域，用于临时存放你**准备提交**的改动（类似“购物车”）。
- **作用**：
  - 允许你选择性地提交部分修改（而不是所有工作区的改动）。
  - 类似于“购物车”，你可以先挑选要提交的文件，再统一提交。
- **特点**：
  - 使用 `git add` 将工作区的改动放入暂存区。
  - 只有暂存区的内容才会被 `git commit` 记录到版本历史。
- **常用命令**：

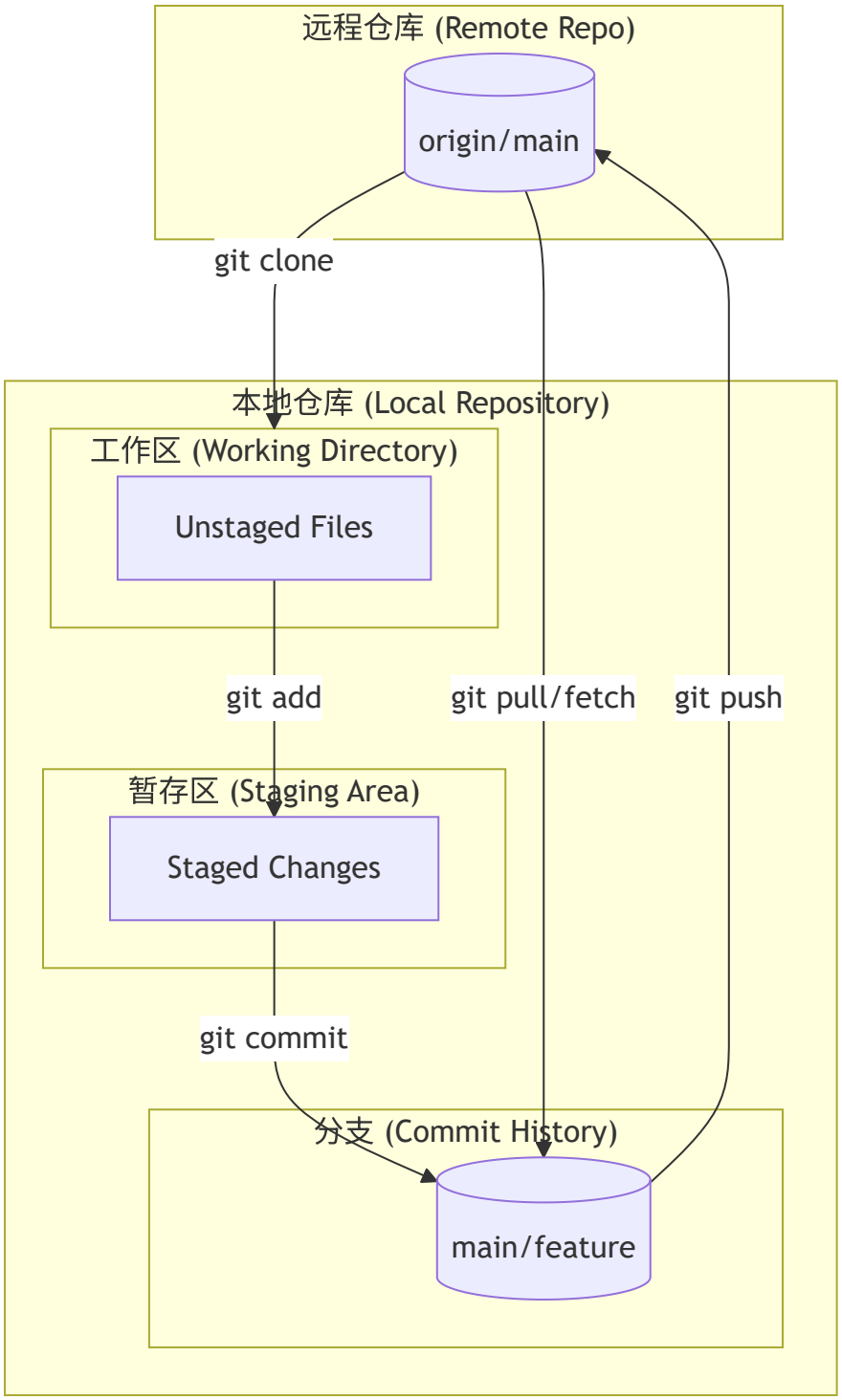
```
git add <file>       # 将文件添加到暂存区
git add .            # 添加所有改动到暂存区
git reset <file>     # 从暂存区移除文件（但保留工作区修改）
git diff --cached    # 查看暂存区和最新提交的差异
```

### 3. 分支（Branch）

- **定义**：分支是 Git 中独立的开发线，用于隔离不同的开发任务（如新功能开发、Bug 修复）。
- **作用**：
  - 允许多人协作，互不干扰。
  - 可以轻松切换、合并不同的代码版本。
- **特点**：
  - 默认分支通常是 `main` 或 `master`。
  - 每次提交（`git commit`）都会在当前分支上生成一个新的版本快照。
  - 分支之间可以合并（`git merge`）或变基（`git rebase`）。
- **常用命令**：

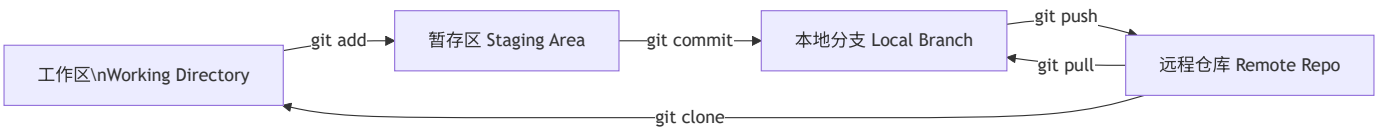
```
git branch              # 查看所有分支
git branch <name>       # 创建新分支
git checkout <branch>   # 切换到指定分支
git merge <branch>      # 合并分支
git rebase <branch>     # 变基（重写提交历史）
```

### 4. Git 核心结构示意图




## 5. 三者的关系

1. **最简工作流程**：

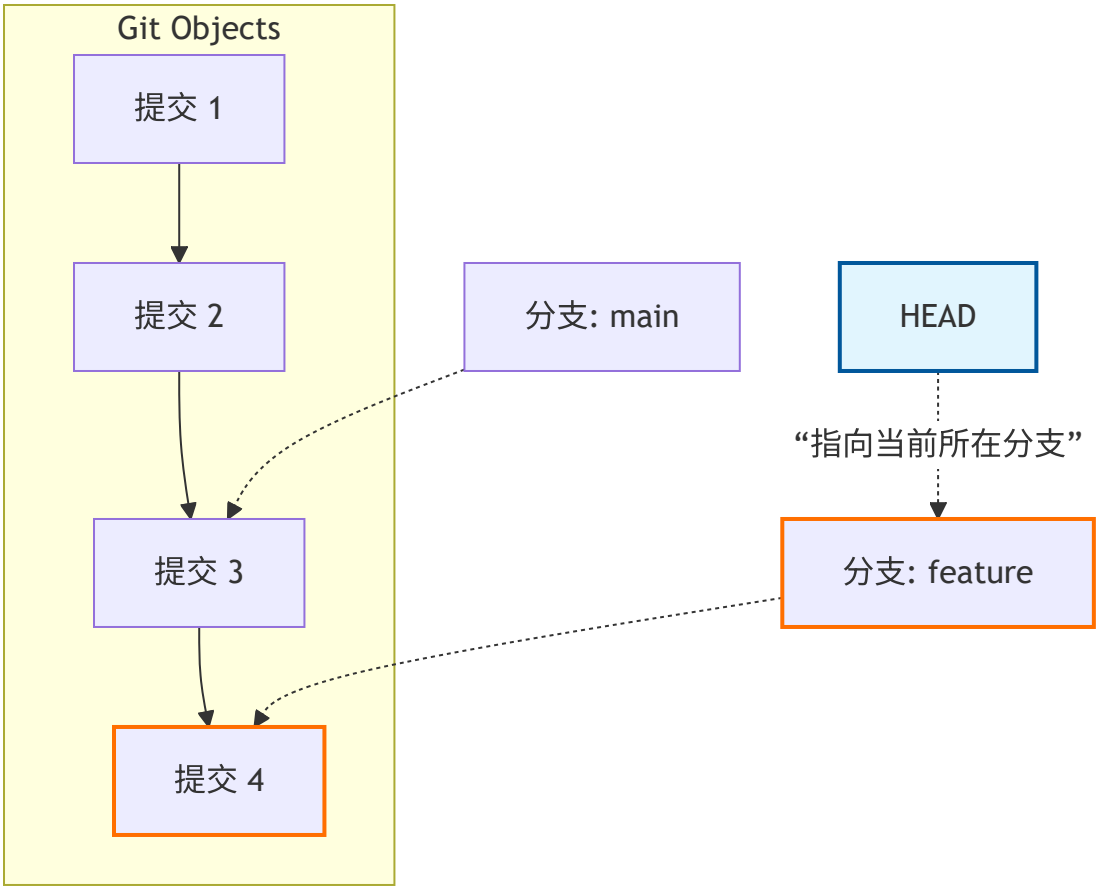



2. **示例**：

```
# 1. 在工作区修改文件
echo "Hello Git" > README.md

# 2. 将修改添加到暂存区
git add README.md

# 3. 提交到当前分支
git commit -m "Update README"

# 4. 推送分支到远程仓库（如GitHub）
git push origin main
```

3. **`HEAD` 及分支指针**
   i. **HEAD指针**

   `HEAD` 指向当前分支指针。 `HEAD` 只有一个，它是一个 **“指针中的指针”**，它会随着你的切换而指向当前所在分支的最新提交。
   为了更直观地理解 `HEAD` 、分支指针（如 `main` ， `feature` ）和提交之间的关系，请看下面的图示：




   a. **HEAD 是什么?**

   - `HEAD` 是 Git 仓库中一个特殊的**引用文件**（位于 `.git/HEAD` ），它永远指向**你当前所在的位置**。

- 你可以把它理解成你电脑上的“当前浏览器窗口”或“你正在看的书页”。
- **关键：在整个仓库中，有且只有一个 `HEAD` 。**

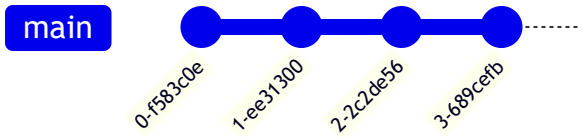
b. **HEAD 的两种状态**

`HEAD` 可以以两种方式指向提交：

**状态一：指向一个分支指针（通常状态）**

这是最常见的情况。当你处在一个分支上时（例如 `git checkout main` 或 `git switch feature` ）， `HEAD` 并不直接指向一个提交，而是**指向一个分支引用**。

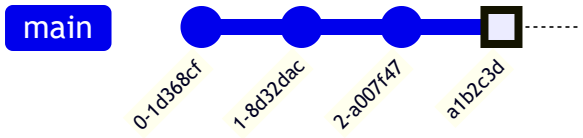
- **命令：** `git checkout main`
- **结果：** `HEAD -> refs/heads/main ->提交 a1b2c3d`
- **含义：**“HEAD 附加在 main 分支上”。此时，任何新的提交都会让 `main` 分支指针向前移动， `HEAD` 也会随之移动，因为它指向的是 `main` 这个指针，而不是某个固定的提交。




**状态二：直接指向一个提交（“分离头指针”状态）**

如果你直接切换到一个提交的哈希值或标签，而不是分支名， `HEAD` 就会直接指向那个具体的提交。

- **命令：** `git checkout a1b2c3d` 或 `git checkout v1.0.0`
- **结果：** `HEAD -> 提交 a1b2c3d`
- **含义：**“HEAD 直接指向了提交 a1b2c3d”。Git 会警告你处于 "detached HEAD state"（分离头指针状态）。




**为什么“分离头指针”是危险的?**

因为如果你在这个状态下创建新的提交，这个新提交将**不属于任何分支**。当你切换回其他分支时，这个没有分支指向的提交很可能会被 Git 的垃圾回收机制最终删除，导致你的工作丢失。

**（安全做法：如果你在分离头指针状态下做了修改，记得创建一个新分支来指向它： `git switch -c new-branch-name` ）**

**总结与类比**

| 概念 | 类比 | 关系 |
| --- | --- | --- |
| **提交** | 书中的**某一页** | 存储项目的具体快照 |
| **分支指针** ( `main` , | 书的**目录**里的一个章节名 | 指向一个章节的最后一项（最新提交） |

| 概念 | 类比 | 关系 |
| --- | --- | --- |
| feature ） | | |
| **HEAD** | 你**当前正在阅读的章节** | 指向**目录中的一个章节名**（通常），<br>或者直接翻到**某一页**（分离头指针） |

所以：

- **每个分支都有一个 HEAD 吗？ 不是的**。 HEAD 只有一个。
- **HEAD 会随着切换指向不同的分支吗？ 是的，完全正确**。当你切换分支时， HEAD 这个唯一指针的内容会被更新，改为指向你刚切换过去的那个分支指针。

你可以用一个简单的命令来验证：

```
git symbolic-ref HEAD
```

- 如果它输出 refs/heads/main 或类似内容，说明 HEAD 正指向 main 分支。
- 如果它报错 fatal: ref HEAD is not a symbolic ref ，说明你处于“分离头指针”状态， HEAD 直接指向了一个提交。

ii. **分支指针**

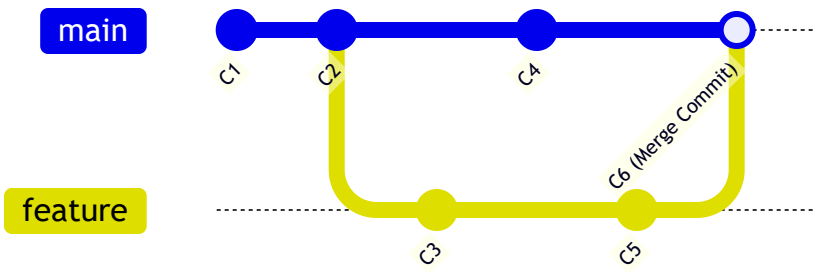
**分支指针是 Git 版本控制的核心机制之一**，本质上是一个**轻量级的、可移动的标签**，它指向某个特定的提交（commit）。

a. **本质**：它是一个保存在 .git/refs/heads/ 目录下的简单文本文件。文件的名字就是分支的名字（如 main ， feature ），文件的内容就是它指向的那个提交的哈希值（SHA-1）。
   - 例如： .git/refs/heads/main 这个文件里可能存着 a1b2c3d4e5f... 这样一个哈希值。

b. **作用**：**标记一条开发线的“最新进展”**。它始终指向该分支上最新的那次提交。

**可视化理解分支指针**

让我们通过一个图表来可视化分支指针的工作方式。下图展示了一个典型的开发流程：从主分支创建特性分支，并在两个分支上分别进行提交，最终通过合并将工作整合。




上图清晰地展示了：

a. **指针的创建**：在 C2 提交处，创建了 feature 分支指针（基于 main 分支）。

b. **指针的移动**：

- 在 feature 分支上提交 C3 和 C5 后， feature 指针移动到最新的 C5 。

- 在 `main` 分支上提交 `C4` 后， `main` 指针移动到最新的 `C4` 。

c. **指针的合并**：执行合并操作后，创建了一个新的提交 `C6` ， `main` 指针随即移动到了这个新的合并提交上。

iii. **HEAD 与分支指针的关系**

这是理解 Git 的关键。 `HEAD` 和分支指针是两种不同的指针，它们协同工作：

- **分支指针**：标记分支的“终点”。
- **`HEAD` 指针**：标记**你当前正在哪个“终点”上工作**。

**绝大多数情况下， `HEAD` 并不直接指向提交，而是指向一个分支指针（这被称为“附加状态”）。**

**关系比喻：**

- **分支指针**像是**火箭的当前轨道**。
- **`HEAD`** 像是 **“当前正在操作中”的轨道指示牌**。
- **新的提交**就像是沿着当前轨道又发射了一节新的火箭舱段。

**命令示例：**

```
git checkout main   # 操作：将HEAD指向main分支指针
# 状态：HEAD -> ref:heads/main -> Commit C2

git commit -m "new commit" # 操作：创建一个新提交C3
# 结果：
# 1. 基于C2创建新的提交C3
# 2. main分支指针自动移动到C3
# 3. 因为HEAD指向main分支，所以HEAD自然也“看到”了C3
# 状态：HEAD -> ref:heads/main -> Commit C3
```

**总结**

- **有分支指针吗？ 有的，而且这是 Git 的核心设计。**
- 每个分支都对应一个指针文件，存储着该分支最新提交的哈希值。
- 分支指针是**轻量的、可移动的**。
- `HEAD` 通常**指向一个分支指针**，而不是直接指向提交。这使得 `HEAD` 可以“骑在”分支指针上，随着分支的增长而自动前进。
- 正是这种“指针链”（`HEAD` -> 分支指针 -> 提交）的设计，使得 Git 的分支模型如此强大和高效。

### 7. 总结

| 概念 | 作用 | 关键命令 |
| --- | --- | --- |
| **工作区** | 直接编辑文件的地方 | `git status` , `git restore` |
| **暂存区** | 暂存待提交的改动 | `git add` , `git reset` |

| 概念 | 作用 | 关键命令 |
| --- | --- | --- |
| **分支** | 管理不同的开发线（版本历史） | `git branch`，`git commit` |

## 2.2.4. Git 基本操作流程

**一、下载安装Git**

在Git官网下载并安装Git。

**二、Git环境配置**

```
git config --global user.name "YourName"
git config --global user.email "your@email.com"

# 查看配置
git config --list
```

这一步是设置你的用户名和邮件地址。 这两个信息在你以后的每一次提交Git都会使用它们，用于识别工作主体是谁。配置完成后，Git会自动使用这些配置信息，比如用户名、邮箱、编辑器。当你想针对特定项目使用不同的用户名称与邮件地址时，可以在那个项目目录下运行没有 `--global` 选项的命令来配置。

如果想要检查你的配置，可以使用 git config --list 命令来列出所有 Git 当时能找到的配置：

```
$ git config --list
user.name=Your Name Here
user.email=your_email@example.com
core.editor=vim
color.ui=auto
alias.st=status
...
```

可以通过输入 `git config <key>`： 来检查 Git 的某一项配置：

```
$ git config user.name
Your Name Here
```

**三、创建仓库**

创建本地仓库有两种方式，创建或切换到某个文件夹，进入该文件夹后：

1. 直接将某个文件夹下的所有文件都添加到 Git 中：
   在该目录下执行 `git init`，该命令会在该目录创建一个 `.git` 的隐藏目录，该目录中保存了 Git 的所有信息，暂存区、分支等均在此目录中进行跟踪管理，它是 Git 的核心，不可以删除该目录，否则无法恢复仓库，含有 `.git` 隐藏目录的该目录成为你的项目的工作区。
2. 克隆某个远程仓库：
   获取某个远程git仓库的地址，比如：https://gitee.com/yourname/project ，然后执行：

   ```
   git clone https://gitee.com/yourname/project.git
   ```

   这会在当前目录下创建一个名为 `project` 的目录，同样，在这个目录下还会有一个隐藏的 `.git` 文件夹。同时，将所有的远程项目文件复制到这个新建的 `project` 文件夹，这个 `project` 文件夹就是你的项目目录，既是你的工作区。
   如果你想在克隆远程仓库的时候，将本地仓库的名字改为自定义的名字，你可以通过指定新的目录名：

   ```
   $ git clone https://gitee.com/yourname/project.git myproject
   ```

   那么，工作目录的名字就是 `myproject`。

本地仓库通过上面两种方式之一创建完成之后就可以等待后续的开发和使用了。

**注意：**

> 要想Git能跟踪到你的文件，你的所有工作必须在工作区目录中进行。

**四、文件管理**

1. 工作区中文件的状态
   在Git中，工作区目录就是你的项目目录。里面的文件要么是**被跟踪的文件（tracked files）**，要么是**未被跟踪的文件（untracked files）**。**已跟踪的文件**是指那些被纳入了版本控制的文件，它们的状态可能是**未修改（Unmodified）**，**已修改（Modified）**或**已放入暂存区（Staged）**。简而言之，已跟踪的文件就是 Git 已经知道的文件。其余文件均可视为未跟踪的文件。
2. 添加文件到暂存区
   未跟踪的文件，通过添加文件到暂存区： `git add filename` 命令，将文件添加到暂存区，即可成为被跟踪的文件。
   被跟踪的文件，后续经过修改（Modified），在需要的时候，依然也是通过 `git add filename` 命令，将文件添加到暂存区。
3. 提交暂存区到分支
   在暂存区的文件，可以通过 `git commit -m "commit message"` 命令，将文件提交到本地仓库。此时，既是对你的工作区被跟踪文件做了一次**快照（Snapshot）**。所谓快照，就是Git 记录下在**commit的那个精确的时刻**，你的**整个项目**在暂存区（Staging Area）里的状态。就如同你打游戏的

一次进度暂存。

被跟踪的文件，后续经过修改（Modified），在需要时，可以通过 `git commit -a` 命令（相当于先执行 `git add` 将所有已跟踪文件的修改添加到暂存区，然后立即执行 `git commit`），将文件添加到暂存区并提交。

4. 状态的查看

   后续可以通过 `git status` 命令查看工作区目录中的文件状态。使用 `git status -s` 命令或 `git status --short` 命令，将得到一种格式更为紧凑的输出。

   使用 `git log` 可以查看提交历史。

5. 查看尚未暂存的文件更新情况

   两种情况：

   i. 尚未暂存

      `git diff` 命令，此命令比较的是**工作目录**中当前文件和**暂存区**域快照之间的差异。 也就是修改之后还没有暂存起来的变化内容。

   ii. 已暂存

      若要比对**已暂存文件**与**最后一次提交**的文件差异，可以用 `git diff --staged` 命令。

      `--staged` 可以换成 `--cached` 。

6. 删除文件

   三种情况：

   i. 直接通过操作系统删除文件，这种情况下，文件会从工作目录中删除，但文件不会从暂存区中删除。需要再运行 `git rm filename` 将该文件从暂存区中删除，并记录此次移除文件的操作。

   ii. 将文件从暂存区中删除，但仍保留在工作目录中，可以使用 `git rm --cached filename` 。

   iii. 同时从工作区和暂存区中删除文件，可以使用 `git rm filename` 。

7. 移动或重命名文件

   `git mv file_from file_to` 将文件从 `file_from` 移动或重命名为 `file_to` 。

8. 查看提交历史

   `git log` 命令可以查看以往的提交历史记录。

   使用选项 `-p` 或 `--patch` ，会显示每次提交所引入的差异。 也可以限制显示的日志条目数量，例如使用 -2 选项来只显示最近的两次提交。

   `git log --pretty=oneline` 会将每个提交放在一行显示，会缩短显示的行数，比较清爽。

9. 修改的撤销

   i. 修改尚还在工作区： `git checkout -- file`

   ii. 修改已经添加在暂存区： `git reset HEAD <file>` 可以把暂存区的修改撤销掉（unstage），重新放回工作区

   iii. 修改提交以后： `git reset --hard HEAD^` 或者 `git reset --hard HEAD~1` ，再或者 `git reset --hard commit_id` ，彻底丢弃指定的提交。 `git reset --soft HEAD^` ，撤销提交但保留改动到暂存区。

## 五、分支管理

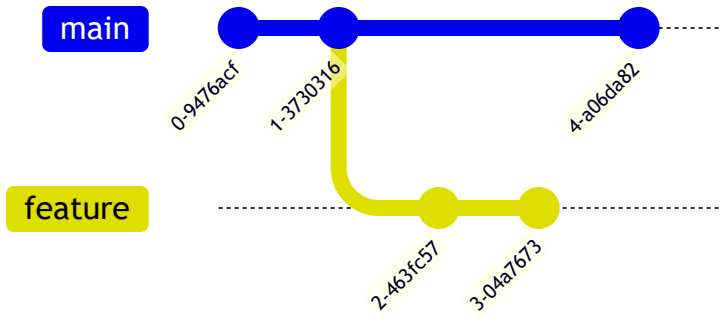
分支是 Git 的**杀手级特性**，也是其最强大的核心概念。

1. 什么是分支?

   想象一下，你的项目开发是一条**时间线**，这条线由一系列提交（快照）构成。默认情况下，这条线叫做 `main` 或 `master` 分支。

   一个**分支**就是这条时间线上的一个**可移动的指针**，它指向某个具体的提交。

   - `HEAD` 是一个特殊的指针，它指向**你当前所在的分支**。
   - 当你进行提交时，当前分支的指针就会自动向前移动，指向最新的提交，而 `HEAD` 也跟着它一起移动。




   **说明**:

   - `HEAD` 指针在图中是隐含的，通过 `checkout` 命令来移动。
   - 当使用 `checkout feature` 或 `switch feature` 后， `HEAD` 就指向了 `feature` 分支- 最新的提交（如 `main` 分支上的提交）就是该分支指针所指向的位置。

2. 为什么需要分支？（分支的应用场景）

   分支的存在是为了实现**并行开发**，互不干扰。经典场景包括：

   - **开发新功能**：为每个新功能（如 `feature/login` ）创建一个分支，功能测试无误后再合并回主分支。
   - **修复线上Bug**：从主分支拉一个 `hotfix/xxx` 分支紧急修复，确保主分支始终稳定。
   - **尝试性工作**：在分支上试验一些可能失败的想法，失败了直接删除分支即可，不会影响主代码。
   - **协同工作**：团队中每个成员都可以在自己的分支上工作，完成后统一合并。

3. 分支的基本管理命令

   i. 查看分支

```
git branch              # 列出所有本地分支，当前分支前有 *
git branch -v           # 查看每个分支的最后一次提交
git branch -a           # 列出所有分支（包括远程分支）
git branch -r           # 只查看远程分支
```

   ii. 创建分支

```
git branch <branch-name>     # 基于当前提交创建一个新分支
git branch <branch-name> <commit-hash> # 基于特定提交创建分支
```

iii. 切换分支

```
git checkout <branch-name>      # 切换到指定分支
git switch <branch-name>        # 更语义化的新命令（推荐）
```

iv. 创建并立即切换到新分支

```
git checkout -b <new-branch-name>     # 旧方式
git switch -c <new-branch-name>       # 新方式（推荐）
```

v. 删除分支

```
git branch -d <branch-name>     # 安全删除（只能删除已合并的分支）
git branch -D <branch-name>     # 强制删除（无论是否合并）
```

vi. 合并分支

这是最核心的操作，将指定分支的更改整合到**当前分支**。

```
git checkout main               # 首先，切换到你要接受合并的目标分支（如main）
git merge <branch-name>         # 将 <branch-name> 分支合并到 main
```

vii. 重命名分支

```
git branch -m <old-name> <new-name> # 重命名分支
```

4. **分支的合并（Merge）与变基（Rebase）及冲突解决**

这是整合分支历史的两种主要方式。

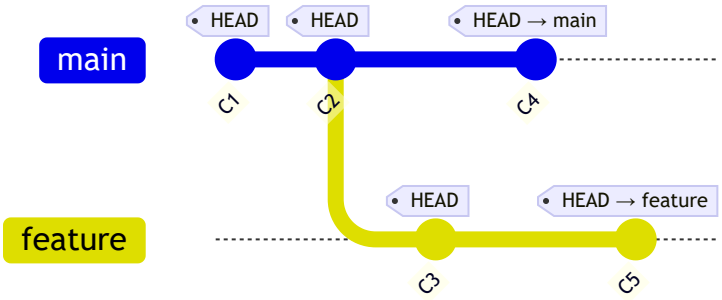
i. 合并（Merge）

**合并的实现方式是通过创建一个新的提交来整合两条时间线的历史，而不是简单地拼接。准确的比喻：创造一个新的“汇合点”。**

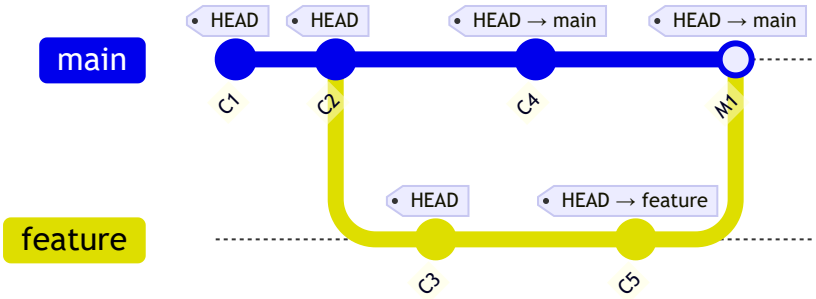
通过分支的合并，Git 会创建一个新的提交节点，这个节点有两个父节点，像一个岔路口，清晰地记录了历史的汇合点，从而保留了完整的分支拓扑结构，而不是将其压扁成一条直线。合并保留了完整的真实历史记录，但是历史记录可能会变得比较复杂，尤其是当分支很多时。

想象一下两条并行的道路（分支 `main` 和 `feature`），它们从同一个起点（共同祖先提交）分道扬镳。

- **合并前**：你有两条独立发展的历史线。




- **合并后**：Git 不会抹去任何一条路，也不会简单地把一条路接到另一条的末尾。相反，它在两条路的前方**创造了一个新的十字路口（合并提交 `M1`）**，这个路口同时连接着两条路，标志着“从这里开始，两条线的历史又汇合了”。




**这个新的“汇合点”（合并提交 `M1`）有两个父提交：**

a. 第一个父提交是原来 `main` 分支的最新提交（`C4`）。

b. 第二个父提交是你要合并进来的 `feature` 分支的最新提交（`C5`）。

这样，整个项目历史就变成了一张**图**（有向无环图），而不是一条简单的直线。从这个新的合并提交 `M` 开始，你可以向左走到 `main` 的历史，也可以向右走到 `feature` 的历史。**所有历史都被完整地保留了下来。**

**关键点**：

- **`git merge feature` 是在 `main` 分支上执行的**，所以执行合并命令前，必须先 `checkout main`，此时 `HEAD` 指向 `main`。
- 合并完成后，会创建一个新的合并提交（M1），**`HEAD` 指针和 `main` 分支指针**都会一起移动到这个新的提交上。
- `feature` 分支的指针保持不变，仍然指向 C5。

**两种合并方式，同一种理念**

Git 主要通过两种方式来实现这种“汇合”，它们都创建新的提交点，但过程不同：

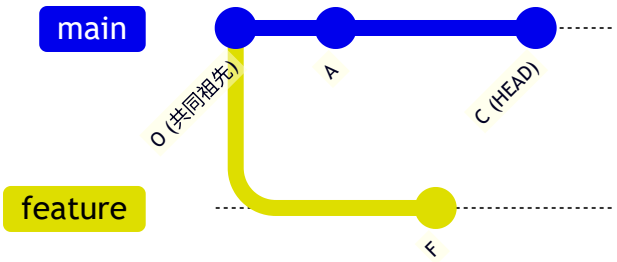
a. **三方合并**

这是最常见的情况。当两个分支都有新的提交时，Git 会进行一次**三方合并**。它需要三个提交来计算合并结果：

a. **合并基础**：两个分支最近的共同祖先（`O`）。

b. **当前分支的末端**：例如 `main` 分支的 `C`。

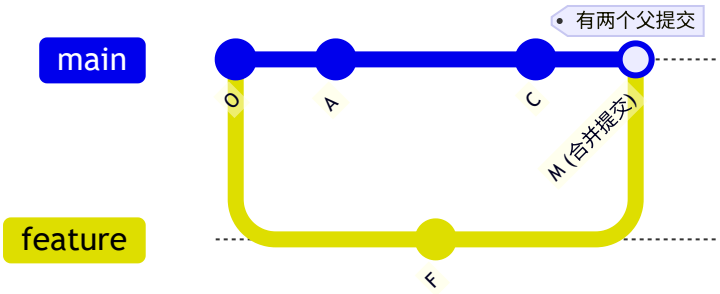
c. **要合并的分支的末端**：例如 `feature` 分支的 `F`。




Git 会比较：

- `O` → `C`：我们在 `main` 分支上做了什么改动？
- `O` → `F`：在 `feature` 分支上又做了什么改动？

然后尝试将这两组改动**整合**起来，应用到一个新的提交上。如果改动没有冲突，Git 会自动完成这个整合，并创建一个新的**合并提交** `M`。

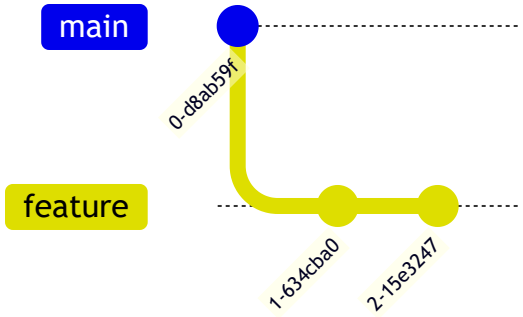



b. **快进合并**

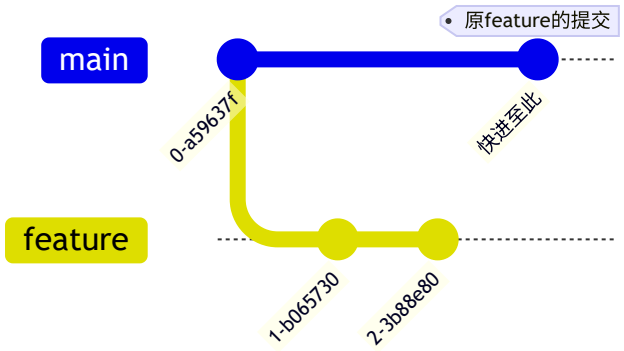
这是一种特殊情况。如果自分支分离后，**当前分支（如 `main`）没有任何新的提交**，那么 `main` 的末端直接就是共同祖先。

这时，Git 不需要创建一个新的合并提交。它可以直接将 `main` 分支的指针**快进**到 `feature` 分支的最新提交，使历史保持线性。

- **合并前**：`main` 停留在老地方，`feature` 前进了。




- **合并后（快进）**：`main` 指针直接移动到 `feature` 的位置。




**注意**：可以使用 `git merge --no-ff` 来禁止快进合并，强制创建一个合并提交，即使满足快进条件。这在团队协作中非常有用，因为它能在历史中清晰地保留一次合并操作的记录。

**总结**

所以，**Git 是把两个时间线合并到一条时间线了吗？**

- **从结果上看，是的**：合并后，你得到了一条新的、统一的时间线（当前分支），它包含了之前两条时间线的所有工作内容。
- **从实现上看，不是简单的拼接**：Git 通过**创建一个新的提交节点**来实现合并。这个节点有两个父节点，像一个岔路口，清晰地记录了历史的汇合点，从而保留了完整的分支拓扑结构，而不是将其压扁成一条直线。

ii. **冲突**

当你在 `main` 分支和 `feature` 分支上修改了**同一个文件的同一个区域**时，Git 无法自动决定应该保留哪个修改。这时，就会发生**冲突**，需要你手动解决。

**冲突是如何发生的？**

Git 的合并是基于**行**的。如果两个分支对同一文件的**不同部分**进行了修改，Git 通常能聪明地自动合并。但一旦修改了**相同或相邻的行**，Git 就会不知所措，因为它无法判断作者的意图。

**冲突示例场景：**

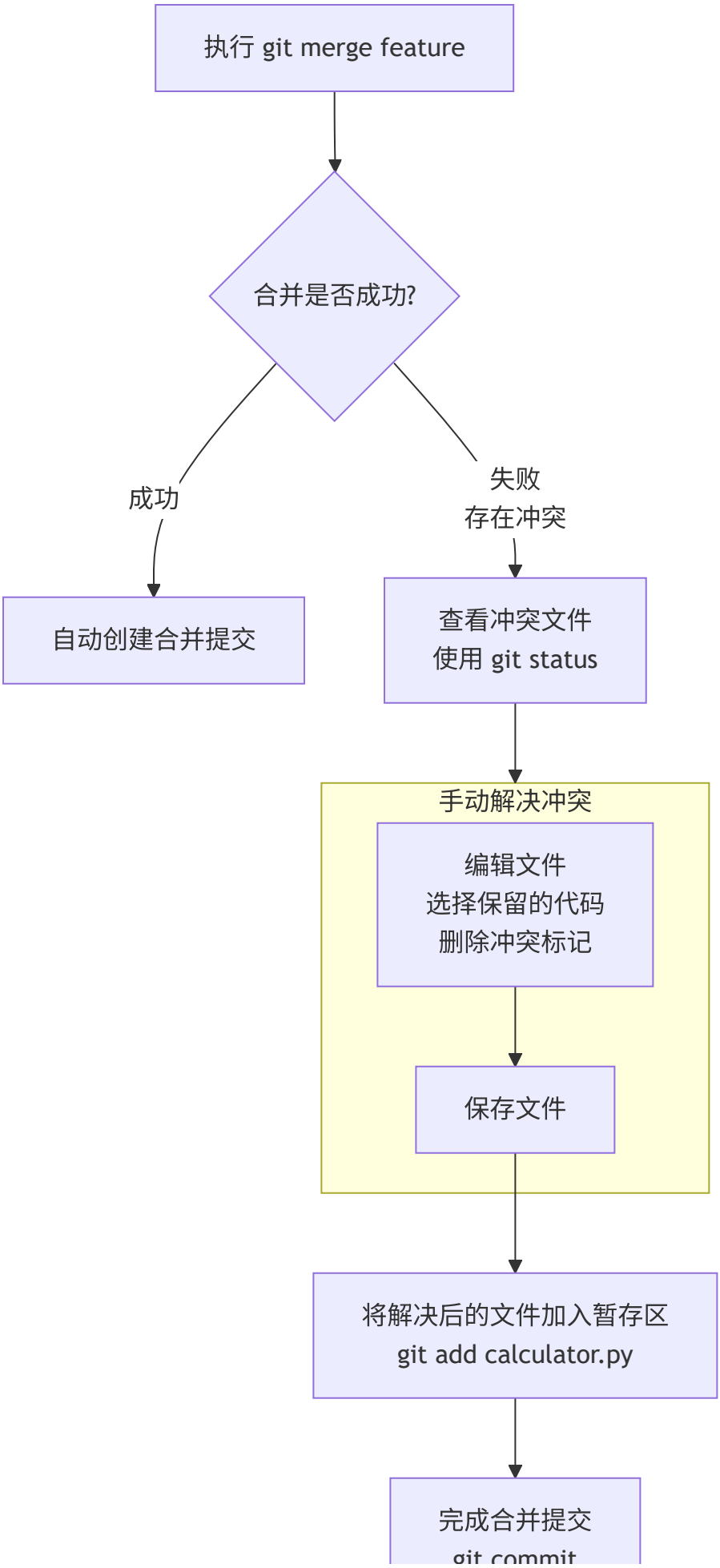
a. 在 `main` 分支上，你修改了 `calculator.py` 文件中的一行代码。

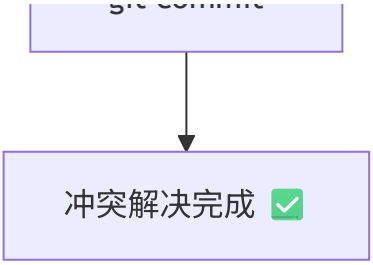
b. 在 `feature` 分支上，你的同事也修改了 `calculator.py` 文件的同一行代码。

c. 当你尝试将 `feature` 合并到 `main` 时，Git 会报告冲突。

**解决冲突的完整工作流**

当 `git merge` 命令因冲突而中止后，整个解决流程如下所示：


执行 git merge feature
合并是否成功?
成功
失败
存在冲突
自动创建合并提交
查看冲突文件
使用 git status
手动解决冲突
编辑文件
选择保留的代码
删除冲突标记
保存文件
将解决后的文件加入暂存区
git add calculator.py
完成合并提交
git commit




下面是图中每个步骤的详细说明：

**第1步：识别冲突**

合并命令失败后，Git 会明确告诉你哪些文件有冲突。

```
$ git merge feature
Auto-merging calculator.py
CONFLICT (content): Merge conflict in calculator.py
Automatic merge failed; fix conflicts and then commit the result.
```

使用 `git status` 查看详细状态：

```
$ git status
On branch main
You have unmerged paths.
(fix conflicts and run "git commit")
(use "git merge --abort" to abort the merge)

Unmerged paths:
(use "git add <file>..." to mark resolution)
        both modified:   calculator.py  <--- 这就是冲突文件！
```

**第2步：手动编辑文件，解决冲突**

打开有冲突的文件（例如 `calculator.py`），你会看到 Git 留下的特殊冲突标记：

```
def add(a, b):
<<<<<<< HEAD
    result = a + b  # 这是当前分支 (main) 的修改
    return result
=======
    sum = a + b     # 这是要合并的分支 (feature) 的修改
    return sum
>>>>>>> feature
```

- `<<<<<<< HEAD`：标记冲突开始。之后的内容是**当前所在分支**（你执行 `git merge` 时所在的分支，这里是 `main`）的代码。
- `=======`：分隔符。分隔两个冲突的修改。

- `>>>>>>> feature`：标记冲突结束。之前的内容是**你要合并进来的分支**（这里是 `feature`）的代码。

**你的任务就是手动编辑这个文件，决定最终要保留的代码。** 你有几种选择：

a. **保留 `main` 的代码**：删除 `feature` 的代码和所有冲突标记。
b. **保留 `feature` 的代码**：删除 `main` 的代码和所有冲突标记。
c. **保留两者的精华**：手动编写一段**全新的代码**，整合两者的修改。
d. **完全重写**：如果都不满意，可以完全重写这一部分。

**解决后的文件示例：**

假设我们决定采用 `feature` 的变量名，但加上 `main` 的注释。

```
def add(a, b):
    sum = a + b     # 这是要合并的分支 (feature) 的修改
    return sum
```

**第3步：标记冲突已解决并完成合并**

a. **将解决后的文件加入暂存区**：这告诉 Git，“这个文件的冲突我已经处理好了”。

```
git add calculator.py
```

（`git status` 此时会从 "Unmerged paths" 变成 "Changes to be committed"）

b. **完成合并提交**：

```
git commit -m "fix: merge conflict in calculator.py by preferring feature's implementat
```

Git 会为你预先填充一个提交信息，通常直接使用即可。

**中止合并**

如果合并变得一团糟，你想重来，可以中止这次合并，回到合并前的状态。

```
git merge --abort
```

**总结**

解决合并冲突的核心步骤是：

a. **理解**：通过冲突标记，看懂两个分支分别做了什么修改。
b. **决策**：基于业务逻辑，决定是保留一方、合并双方还是完全重写。
c. **编辑**：动手修改文件，**删除所有冲突标记**，留下你想要的最终代码。
d. **完结**：`git add` 告诉 Git 冲突已解决，然后 `git commit` 完成合并。

iii. 变基（Rebase）

**核心比喻：就是整理时间线**

`git rebase`（变基）是 Git 中一个强大但略有挑战的概念。想象你的项目历史是一部正在拍摄的**电影胶片**。

- `merge` **（合并）**：像是把两条并行的胶片（分支）**剪下来，并排地粘**到一个新的画面上。它保留了所有原始历史，但会创建一个新的“合并”节点，历史线变成一个有岔路的图。
- `rebase` **（变基）**：像是把其中一条胶片（特性分支）**剪下来，接到另一条胶片（主分支）的最新片尾之后**。结果是得到一条**完全线性、整洁**的历史线，仿佛所有工作都是按顺序完成的。

a. **Rebase 是什么?**

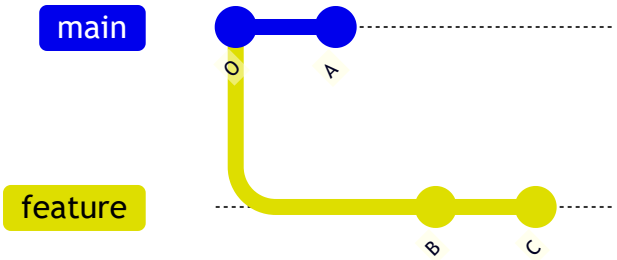
**变基**的本质是**改变一个分支的“基础”提交**。

默认情况下，当你从 `main` 分支切出一个 `feature` 分支后，两个分支都会独立发展。 `feature` 分支的“基础”是它从 `main` 分出去的那个旧提交，这个处于 `main` 分支上的那个分裂点提交就是 `**基点(base)**` 。所以**变基**就是将这个分支的 ***“基础”**提交改为 `main` 分支 **最新的提交**。

`git rebase` 所做的是：**“嘿， `feature` 分支，别以那个旧的 `main` 提交为基础了。请把你所有的新提交重新‘播放’在 `main` 分支最新的提交之上。”**

b. **工作流程对比：Merge vs. Rebase**

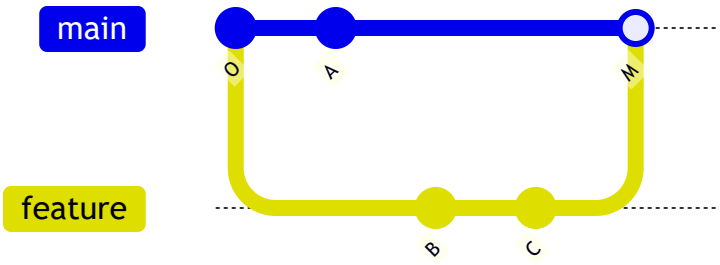
假设我们有如下初始状态，并分别在两个分支上进行了开发：




a. **方案一：使用 `git merge` （合并）**

```
# 在 main 分支上执行
git merge feature
```

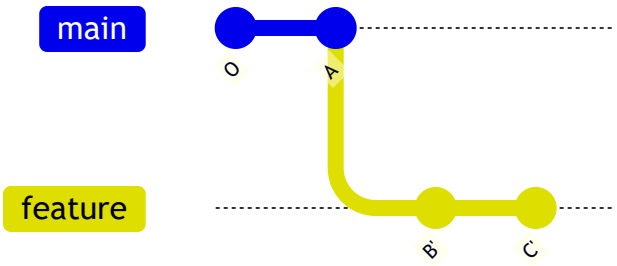
合并后的历史会创建一个新的合并提交（`M`），它有两个父提交，历史轨迹清晰但出现了分叉与交汇。




b. **方案二：使用 `git rebase` （变基）**

```
# 1. 先切换到 feature 分支
git checkout feature

# 2. 执行变基，将 feature 的提交“重新嫁接”到 main 的最新提交之后
git rebase main
```

变基后的历史变为一条直线，仿佛工作 `B` 和 `C` 是在 `main` 分支的 `A` 提交完成之后才顺序进行的。




**关键点**：

- **`git rebase main` 是在 `feature` 分支上执行的**，所以执行命令前， `HEAD` 指向 `feature` （在 C）。
- 变基后， `B` 和 `C` 提交的哈希ID会改变，但提交内容不变，因为它们现在是基于新的“基础”(提交 `A` ）重新计算出来的新提交（可看作是 `B'` 和 `C'` )。
- 变基完成后， **`HEAD` 指针和 `feature` 分支指针**会一起移动到新的提交链顶端（C')。
- 原来的提交 `B` `C` 如果没有被其他分支引用，最终会被 Git 的垃圾回收机制清除。

c. **Rebase 的两种主要用法**

**用法 1：整理本地分支（最安全，最推荐）**

在将本地分支推送到远程之前，用它来整理提交历史，使其更清晰。

```
# 假设你在 feature 分支上工作了几天
# 期间，main 分支也有其他人推送了更新

# 1. 获取远程最新main分支代码，但不合并
git fetch origin

# 2. 在feature分支上，变基到最新的origin/main
git rebase origin/main

# 这个过程可能发生冲突，需要你一步步解决（类似合并冲突）
# 解决后，用 git rebase --continue 继续
```

**好处**：你的 `feature` 分支历史看起来就像是基于最新的 `main` 开发的，没有多余的合并提交，历史线非常干净。

**用法 2：交互式变基（ `git rebase -i` ）（超级强大）**

这允许你在“重新嫁接”提交时，对提交历史进行**编辑、整理、美化**。

```
# 变基当前分支的最后3个提交
git rebase -i HEAD~3
```

执行后会打开一个文本编辑器，你可以**重新排序、压缩（squash）、修改（edit）提交信息**等。

```
pick a1b2c3d Added new feature X
pick b2c3d4e Fixed a typo
pick c3d4e5f Added new feature Y

# 你可以将其改为：
pick a1b2c3d Added new feature X
fixup b2c3d4e Fixed a typo  # 将"Fixed a typo"合并到上一个提交中，并丢弃其提交信息
pick c3d4e5f Added new feature Y

# 或者：
reword a1b2c3d Added an awesome new feature X  # 修改提交信息
pick b2c3d4e Fixed a typo
pick c3d4e5f Added new feature Y
```

这是整理凌乱的本地提交历史的终极武器，可以让你在推送前生成清晰、有逻辑的提交历史。

d. **黄金法则：何时用，何时不用**

- ✅ **DO 可以/应该使用 Rebase 的情况**：
  - 整理**尚未推送**到远程仓库的**本地提交历史**。
  - 在合并到 `main` 之前，使你的特性分支与主分支保持同步。
- ❌ **DO NOT 绝对不要使用 Rebase 的情况**：
  - **不要对已经推送到远程仓库（尤其是共享仓库）的提交进行变基**。
  - **为什么？** 因为变基会改变提交的哈希ID（重写历史）。你的同事本地仓库里的历史还是旧的，当你把重写后的历史强制推送（`git push --force`）后，会造成巨大的混乱和同步困难。这会成为团队协作的噩梦。

e. **总结**

- `merge`：**合并**。保留完整历史，创建一个合并提交。历史是真实的，但可能会显得杂乱。适合将公共分支的更改整合到一起。
- `rebase`：**变基**。重写历史，创造一条更线性、整洁的提交线。历史被美化过，仿佛一切井然有序。**仅推荐用于整理本地分支。**

**一个很好的工作流：**

> **在本地用 `rebase` 来整理历史，在共享时用 `merge` 来集成工作。**
> 即：自己开发时用 `git rebase` 保持分支更新和整洁，完成后发起一个 Pull Request，然后由管理员用 `git merge` 合并到 `main` 分支。

iv. **变基时产生冲突**

**为什么变基容易产生冲突?**

想象一下， `git rebase` 的工作是：**将你分支上的提交一个一个地“重新播放”到新的基点上**。

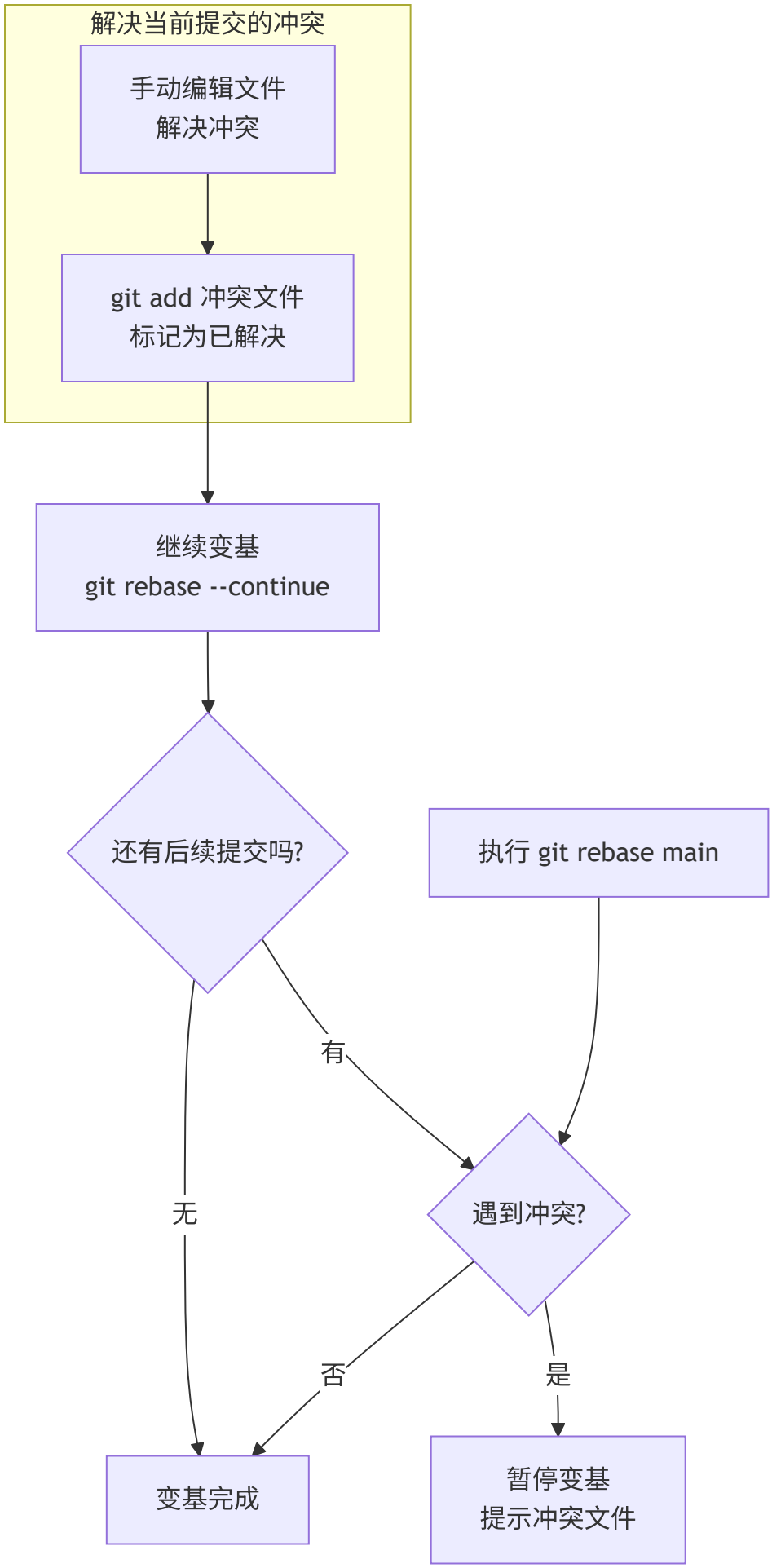
- **合并**：比较的是“分岔点”和“两个终点”的差异，然后一次性生成一个合并结果。
- **变基**：是将每个提交**按顺序**应用上去。任何一个提交在应用时如果和新的基础有冲突，整个过程就会暂停。

这就好比：

- **合并**：是直接把两本书的最终章节拼在一起。
- **变基**：是把你写的每一页手稿，重新抄写到一本别人已经修改过的新书稿的对应位置。在抄写某一页时，如果发现那一页的位置已经被别人写了别的内容，你就不知道该怎么办了。

**变基时解决冲突的完整工作流**

当 `git rebase` 过程中发生冲突时，整个过程会暂停，并提示你解决。其解决流程是一个循环，需要为每一个产生冲突的提交重复操作，如下图所示：


解决当前提交的冲突
手动编辑文件
解决冲突
git add 冲突文件
标记为已解决
继续变基
git rebase --continue
还有后续提交吗?
有
无
执行 git rebase main
遇到冲突?
否
是
变基完成
暂停变基
提示冲突文件

下面是每个步骤的详细说明：

**第1步：冲突发生，变基暂停**

你执行 `git rebase main`，然后可能会看到这样的输出：

```
Auto-merging calculator.py
CONFLICT (content): Merge conflict in calculator.py
error: could not apply abc1234... Added new feature  # 在应用某个提交时失败了
Resolve all conflicts manually, mark them as resolved with
"git add/rm <conflicted_files>", then run "git rebase --continue".
You can instead skip this commit: run "git rebase --skip".
To abort and get back to the state before "git rebase", run "git rebase --abort".
```

**第2步：手动解决冲突**

此时，**不要执行 `git commit`**。你需要做的是：

a. **编辑有冲突的文件**：文件里会有和 `git merge` 一样的冲突标记（`<<<<<<<`，`=======`，`>>>>>>>`）。

- **注意**：此时的冲突标记略有不同。`HEAD` 表示你**正在 rebase 到的目标分支的新基础**（例如 `main` 的最新提交），而 `abc1234`（提交哈希）才是你正在应用的、来自旧分支的提交。

```
<<<<<<< HEAD
# 这是main分支上的最新代码
result = a + b
=======
# 这是你特性分支上的修改
sum = a + b
>>>>>>> abc1234... Added new feature
```

b. **做出决策**：和解决合并冲突一样，你需要编辑文件，保留想要的代码，并**完全删除所有冲突标记**。

**第3步：继续变基**

解决完冲突后，**这是最关键的一步**：

a. **将解决后的文件加入暂存区**：这告诉 Git “这个文件的冲突我已经解决了”。

```
git add calculator.py
```

b. **继续变基流程**：使用 `git rebase --continue` 告诉 Git：“我处理完这个提交的冲突了，请继续应用下一个提交吧。”

```
git rebase --continue
```

**第4步：循环（很可能发生）**

Git 会继续尝试应用下一个提交。**如果下一个提交也修改了同一区域，很可能再次发生冲突。**你需要重复第2步和第3步，直到所有提交都成功应用完毕。

**其他选项：放弃或跳过**

在解决冲突时，你还有两个选择：

- **`git rebase --skip`：**
  **跳过当前正在应用的提交**。这意味着你会完全丢弃这个提交引入的所有更改。**非常危险**，除非你确定这个提交完全没必要了，否则不要使用。
- **`git rebase --abort`：**
  **完全中止变基操作**。这是最安全的后备选项。当你觉得冲突解决起来太复杂，想把一切恢复到你执行 `rebase` 之前的状态，然后换个策略（比如直接 `merge`）时，就使用这个命令。

```
git rebase --abort # 一切回到原点，就像什么都没发生过
```

**为什么变基解决冲突更麻烦？**

因为这是一个**迭代过程**。你可能需要为多个提交重复解决相似甚至相同的冲突。例如，如果你在旧分支的早期提交中引入了一个错误写法，并在后续提交中修复了它，那么在变基时，你可能会在“引入错误”的那个提交处解决一次冲突，然后在“修复错误”的提交处再解决一次。

**最佳实践与建议**

a. **在变基前先同步**：在执行 `git rebase main` 之前，先确保你的 `main` 分支是最新的（`git fetch origin && git checkout main && git pull`）。
b. **一个一个提交处理**：变基冲突需要耐心，一次处理一个提交。
c. **使用工具**：`git mergetool` 在解决变基冲突时同样有效，可以可视化地帮助你。
d. **理解你在重写历史**：记住，变基会改变提交的哈希值。**只对你本地、尚未推送的分支执行变基**。如果分支已经推送共享，就不要变基了，否则会给团队协作带来灾难。

**总结：变基肯定会产生冲突，其解决过程是交互式的、一步一步的。核心命令是：解决冲突 -> `git add` -> `git rebase --continue`，直到完成。如果搞不定，就用 `git rebase --abort` 安全退出。**

5. **总结**

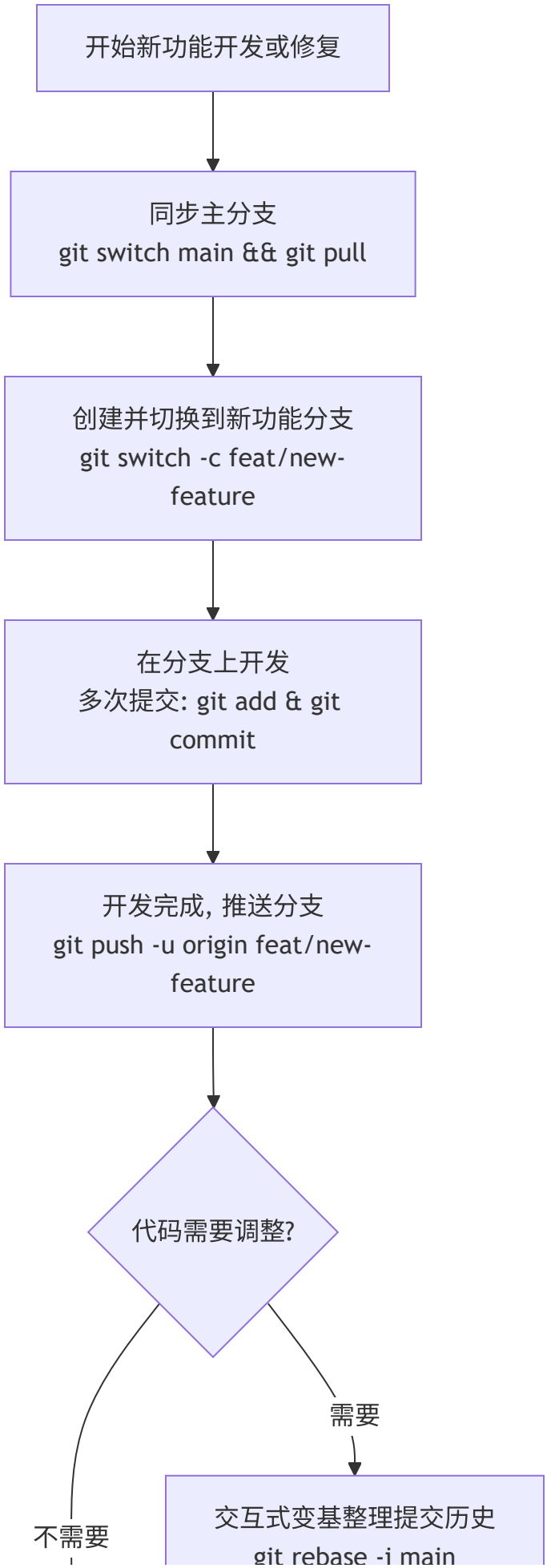
| **操作** | **常用命令** |
| --- | --- |
| **查看/创建/删除** | `git branch`，`git switch -c`，`git branch -d` |
| **切换** | `git switch` 或 `git checkout` |
| **合并** | `git merge` (保留历史) |
| **变基** | `git rebase` (整理历史) |

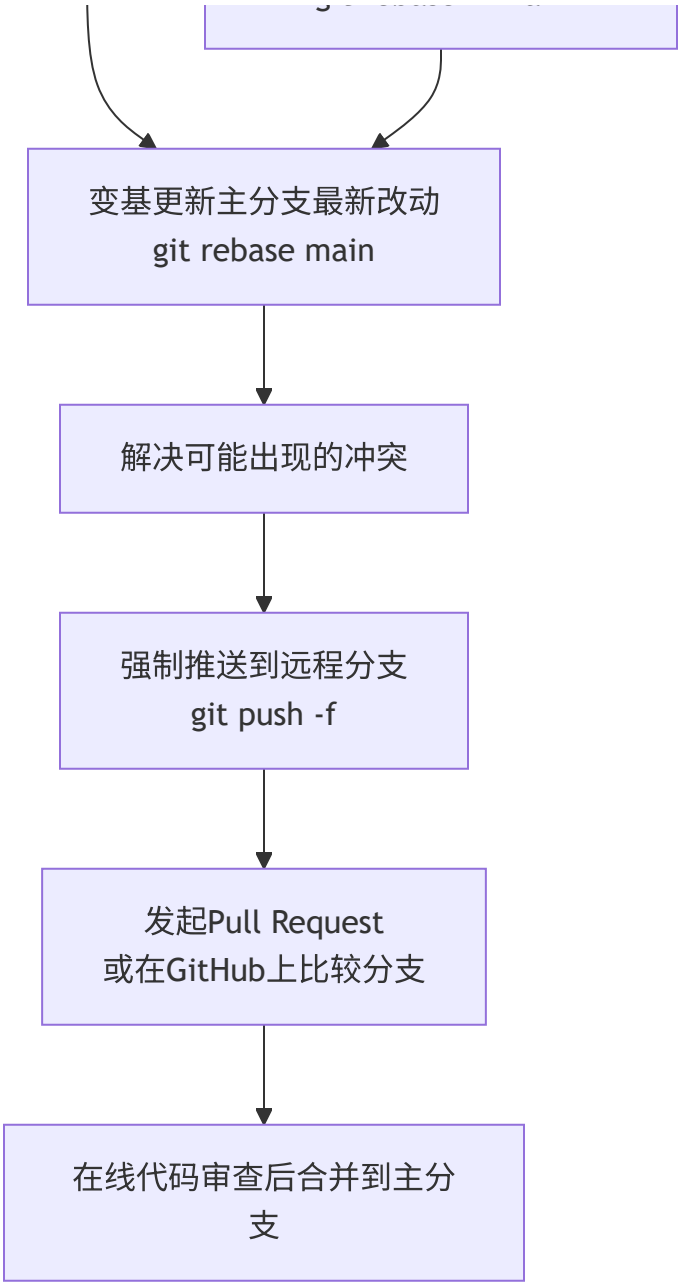
| 操作 | 常用命令 |
| --- | --- |
| **同步远程** | `git push`，`git fetch` |

6. **最佳实践：**
   - **主分支保护**：保持 `main` 分支的清洁和稳定。
   - **分支粒度要小**：一个分支只做一件事（一个功能或一个修复）。
   - **频繁提交**：在分支上频繁提交，保持提交记录的原子性。
   - **频繁同步**：经常从 `main` 分支拉取更新到你的功能分支，减少最终合并时的冲突。
   - **先 `fetch` 后操作**：在操作远程分支前，先 `git fetch` 一下，获取最新信息。

## 六、工作流程

1. **个人开发工作流**
   下面介绍一种非常经典且强大的个人工作流程：**基于功能分支的流程**。它非常适合个人项目、练习或是在团队项目中的个人开发部分。
   为了更直观地理解这个流程，下图展示了从开始工作到最终合并的完整闭环：


开始新功能开发或修复
同步主分支
git switch main && git pull
创建并切换到新功能分支
git switch -c feat/new-feature
在分支上开发
多次提交: git add & git commit
开发完成, 推送分支
git push -u origin feat/new-feature
代码需要调整?
需要
不需要
交互式变基整理提交历史
git rebase -i main




i. **核心原则**

a. **`main` 分支是神圣的**：始终保持主分支（`main` 或 `master`）的清洁和可运行状态。你永远不会直接在主分支上提交代码。

b. **功能驱动**：每一项新功能、每一个错误修复，都应在**单独的分支**上完成。

c. **频繁提交**：在功能分支上，频繁地提交小更改。提交信息要清晰明了。

d. **保持同步**：经常将主分支的更新拉取并合并到你的功能分支上，避免最后集成时出现大量冲突。

ii. **详细步骤分解**

**第 0 步：准备工作 (Always Start Here)**

在开始任何新工作之前，确保你的本地主分支是最新的。

```
git switch main       # 切换到主分支
git pull origin main # 从远程仓库拉取最新代码
```

**第 1 步：创建功能分支**

为新功能或修复创建一个**描述性的分支名**。

```
# 语法: git switch -c <branch-name>
git switch -c feat/add-user-auth    # 新功能: feat/
git switch -c fix/header-bug        # 修复bug: fix/
git switch -c docs/update-readme    # 更新文档: docs/
```

**第 2 步：在分支上开发**

这是你的主要工作阶段。在此分支上，你可以自由地：

- **添加文件**
- **修改代码**
- **频繁提交**

```
# ... 编写一些代码 ...
git add .                                          # 暂存所有更改
git commit -m "feat: implement user login API"     # 提交

# ... 编写更多代码 ...
git add index.js
git commit -m "fix: validate email format"         # 再次提交
```

**提交信息规范**：推荐使用类似 `feat:`，`fix:`，`docs:`，`style:` 等前缀，让历史更清晰。

**第 3 步：定期变基（Rebase）**

这是保持历史线性和整洁的**关键步骤**。当 `main` 分支有更新时，将它们“合并”到你的功能分支上的最佳方式是使用变基。

```
# 1. 确保main是最新的 (如果别人提交了代码)
git switch main
git pull origin main

# 2. 回到你的功能分支并变基
git switch feat/add-user-auth
git rebase main
```

- **如果发生冲突**：解决冲突，然后 `git add .`，接着执行 `git rebase --continue`。
- **变基的好处**：它会让你的所有提交都“仿佛”是基于最新的 `main` 分支开发的，使得最终合并时历史是一条直线，非常清晰。

**第 4 步：完成开发，推送分支**

当功能完成并通过测试后，将分支推送到远程仓库（如 GitHub/GitLab）。

```
git push -u origin feat/add-user-auth # -u 设置上游分支，后续push简化
```

**第 5 步：发起拉取请求 (Pull Request / Merge Request)**

a. 在 GitHub/GitLab 上，针对你的 `feat/add-user-auth` 分支创建一个 Pull Request (PR)。

b. **即使是一个人，也强烈建议创建 PR**。这为你提供了一个：

- **代码审查的机会**：自己Review一下自己的代码，常常能发现之前忽略的问题。
- **自动化的检查**：可以集成 CI/CD（如 GitHub Actions），自动运行测试、检查代码风格。
- **合并的仪式感**：明确标记一个功能的完成。

**第 6 步：合并并清理**

a. 在 PR 页面确认代码无误后，点击合并（Merge）。

b. 合并成功后，回到本地进行清理：

```
git switch main                  # 切换回主分支
git pull origin main             # 拉取合并后的最新代码（现在包含你的工作了！）
git branch -d feat/add-user-auth # 删除本地的功能分支
```

**(可选) 第 7 步：整理提交历史（交互式变基）**

在推送前，如果你觉得提交历史太乱（比如有很多临时提交），可以使用交互式变基来整理，这是**打造完美提交历史的利器**。

```
git rebase -i main
```

然后你可以：

- `squash(s)`：将多个小提交合并成一个有意义的提交。
- `reword(r)`：修改某条提交信息。
- `drop(d)`：删除某个提交。

**注意：交互式变基会重写历史，只适用于尚未推送到远程的分支。如果已推送，需要强制推送 `git push -f` ，但要谨慎使用。**

**总结**

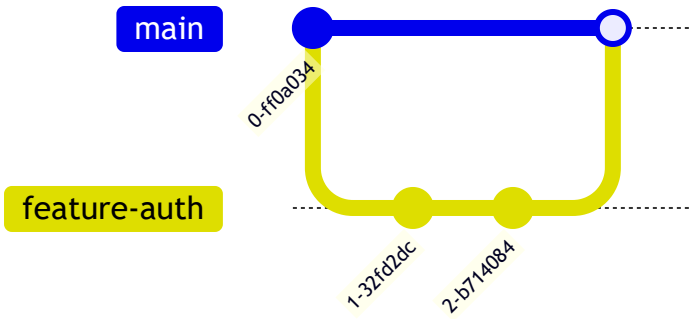
这个个人工作流程的核心是：**功能分支 + 频繁提交 + 变基同步 + PR自检**。

它帮助你：

- **保持代码库整洁**。
- **上下文切换自如**（可以随时中断一个功能去修复另一个紧急bug）。
- **形成清晰、有意义的版本历史**。
- **为团队协作打下坚实基础**。

2. **团队协作工作流**

i. **功能分支策略**




- **操作步骤**：

```
# 1. 创建功能分支
git checkout -b feature-auth

# 2. 开发完成后推送到远程
git push -u origin feature-auth

# 3. 在GitHub创建Pull Request
# 4. 代码审核通过后合并到main分支
```

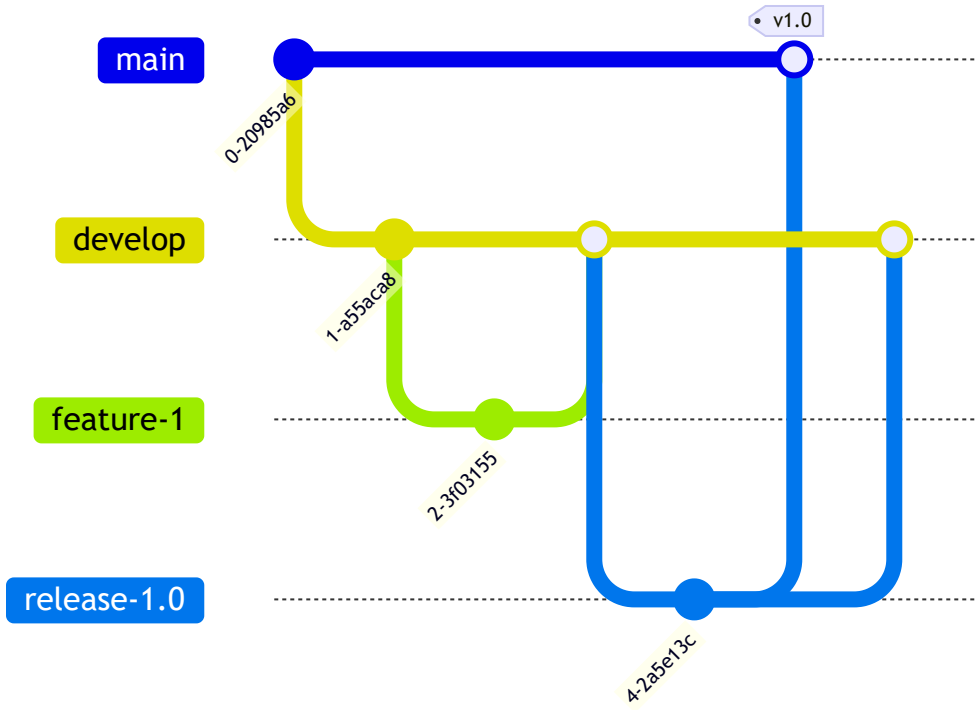
ii. **解决合并冲突**

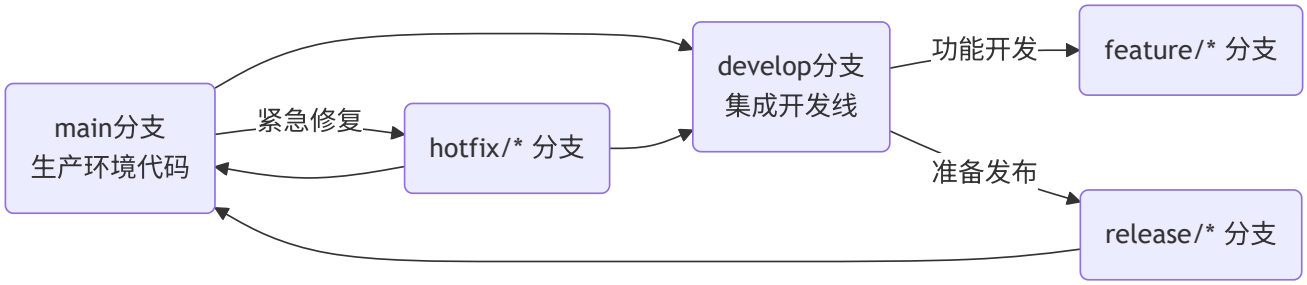
- 当多人修改同一文件时发生
- **解决步骤**：

```
git fetch origin
git merge origin/main  # 出现冲突
# 手动编辑冲突文件（搜索 >>>>>> 标记）
git add resolved_file.py
git commit -m "解决合并冲突"
```

3. **企业级工作流方案**

i. **Git Flow（经典模型）**




- **分支类型**：
  - `main`： 生产环境代码
  - `develop`： 集成测试分支
  - `feature/*`： 功能开发分支
  - `release/*`： 版本发布分支
  - `hotfix/*`： 紧急修复分支

ii. **GitHub Flow（简化版）**

a. `main` 分支永远是可部署的状态。

b. 从 `main` 拉取一个新分支进行功能开发或修复。

c. 在本地进行提交，并定期推送到远程的同一分支。

d. 开发完成后，发起 **Pull Request**（PR）。

e. 代码审查通过后，合并到 `main` 分支，并立即部署。

4. **Git 工作流最佳实践**

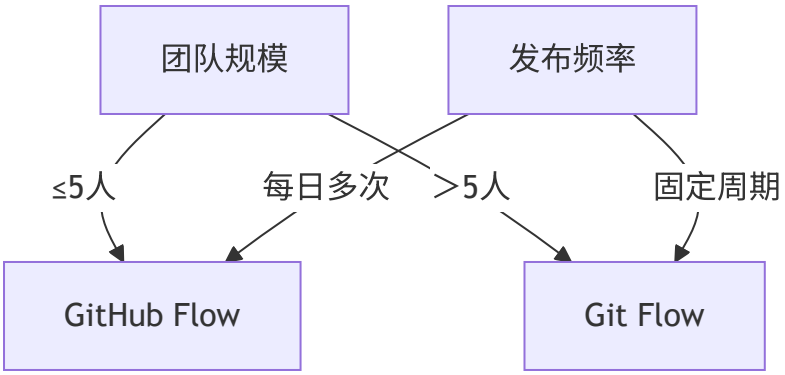
i. **分支命名约定**

- `feature/user-auth` （功能开发）
- `bugfix/header-style` （问题修复）
- `hotfix/order-404` （紧急修复）

ii. **.gitignore 文件**

```
# 忽略系统文件
.DS_Store
# 忽略依赖目录
node_modules/
# 忽略敏感信息
*.env
```

5. **工作流选择建议**




**七、远程仓库的操作**

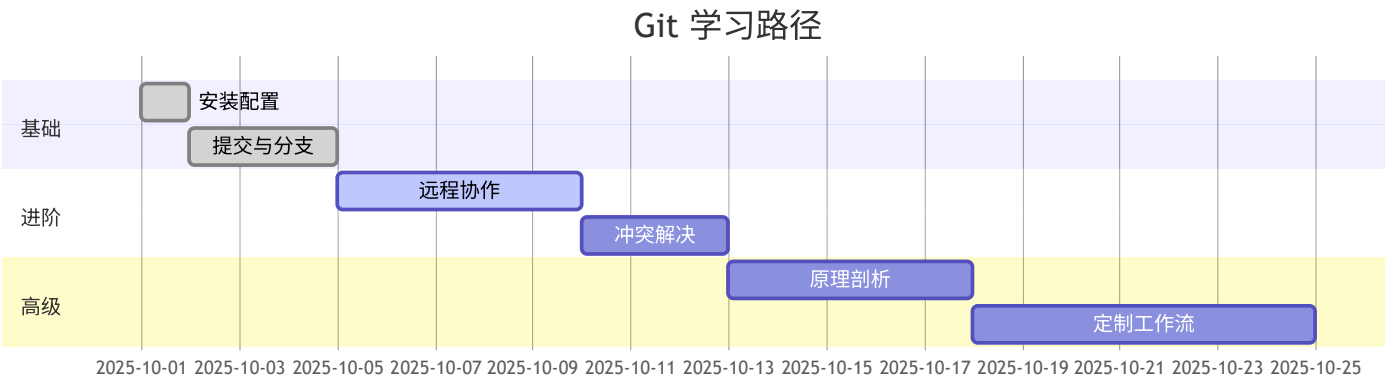
远程分支是你本地仓库里指向远程仓库（如 GitHub）分支的指针，格式为 `<remote>/<branch>` （如 `origin/main` ）。

1. 显示远程仓库： `git remote -v` ，该命令会显示所有远程仓库，以及它们对应的网址。
2. 添加远程仓库： `git remote add <shortname> <url>` 添加一个远程仓库，并设置它的别名为 `<shortname>` 。
3. 推送到远程仓库： `git push -u origin master` ，将本地 `master` 分支的数据推送到远程 `origin` 的 `master` 分支， 参数 `-u` 表示首次推送。
4. 拉取远程仓库： `git pull origin master` ，将远程仓库的数据拉到本地 `master` 分支，并完成合并； `git fetch <remote>` 只拉取远程仓库的数据，但不会合并到本地。
5. 修改远程仓库的简写名： `git remote rename` 。 例如，想要将 pb 重命名为 paul，可以用 git remote rename 这样做：

```
bash     $ git remote rename pb paul
```

6. 删除远程仓库： `git remote rm origin` 或者 `git remote remove origin`
7. 显示远程仓库的配置信息： `git remote show origin` 。

## 2.3. 学习路线图




**推荐学习资源**

1. Git 官方文档
2. 廖雪峰Git教程
3. Gitee 帮助中心
4. 在线学习Git

# 3. 软件许可证协议

**软件许可证协议**，是一份具有法律效力的合同，规定了用户在使用、修改、分发软件时必须遵守的规则和条件。它定义了软件的版权所有者（通常是开发者或公司）和用户之间的权利与责任关系。
理解软件协议是参与软件开发（无论是使用还是创作）的必备知识。在使用任何第三方代码前，请务必检查其许可证协议，确保合规，避免法律纠纷。

简单来说，它回答了以下几个核心问题：

- **我能用这个软件做什么？**（个人使用？商业使用？）
- **我能看和修改它的源代码吗？**
- **我能把它卖钱吗？**
- **如果我修改了代码，我必须把我修改后的版本也开源吗？**
- **我需要给我的产品起另一个名字吗？**
- **如果软件出问题导致损失，谁来负责？**

**它不是**：软件的功能说明书或使用教程。

## 3.1. 为什么软件协议如此重要？

1. **保护开发者权益**：明确版权归属，防止他人盗用代码并声称是自己的。
2. **约束用户行为**：防止软件被滥用，比如用于非法用途。
3. **促进协作与共享**（特别是开源协议）：通过明确规则，鼓励更多人放心地使用、贡献和分发软件，从而形成繁荣的生态。
4. **规避法律风险**：无论是使用者还是开发者，不遵守软件协议都可能面临被起诉的风险。

## 3.2. 软件协议的主要类型

软件协议总体上可以分为两大阵营：**开源协议**和**商业闭源协议**。它们之间的核心区别如下表所示：

| **特性** | **开源协议** | **商业闭源协议** |
| --- | --- | --- |
| **获取源代码** | **可以** | **不可以** |
| **修改源代码** | **可以** | **不可以** |
| **再分发** | **可以**（需遵守协议） | **严格限制** |
| **收费** | 可以（卖服务、支持等） | 通常需要购买许可证 |
| **主要目的** | 促进协作、共享、透明 | 保护知识产权、盈利 |
| **例子** | GPL, MIT, Apache | Windows 许可证, Photoshop 许可证 |

## 3.3. 常见开源协议详解（按限制严格程度排序）

1. **GPL (GNU General Public License)**
   - **核心要求**：**病毒式传染性**。只要你使用了任何GPL协议的代码，你的整个项目也必须以GPL协议开源。
   - **适用场景**：希望确保所有衍生作品都保持开源的开源项目。如Linux内核、Git。
   - **风险**：商业公司通常避免使用GPL库，以免被迫开源自己的专有代码。
2. **LGPL (GNU Lesser General Public License)**
   - **核心要求**：GPL的宽松版。如果只是**动态链接**（如作为一个库）使用LGPL代码，你的专有代码可以不用开源。但如果你**修改了LGPL库本身**，则必须开源。
   - **适用场景**：开源库，希望被更多项目（包括闭源项目）使用。
3. **Apache 2.0**
   - **核心要求**：允许自由使用、修改、分发，甚至销售。要求包含原始版权和协议声明，并提供修改说明。**特别提供了专利授权**，保护使用者不会被原作者用专利起诉。
   - **适用场景**：大型项目，涉及专利保护需求。如Android、Apache项目。

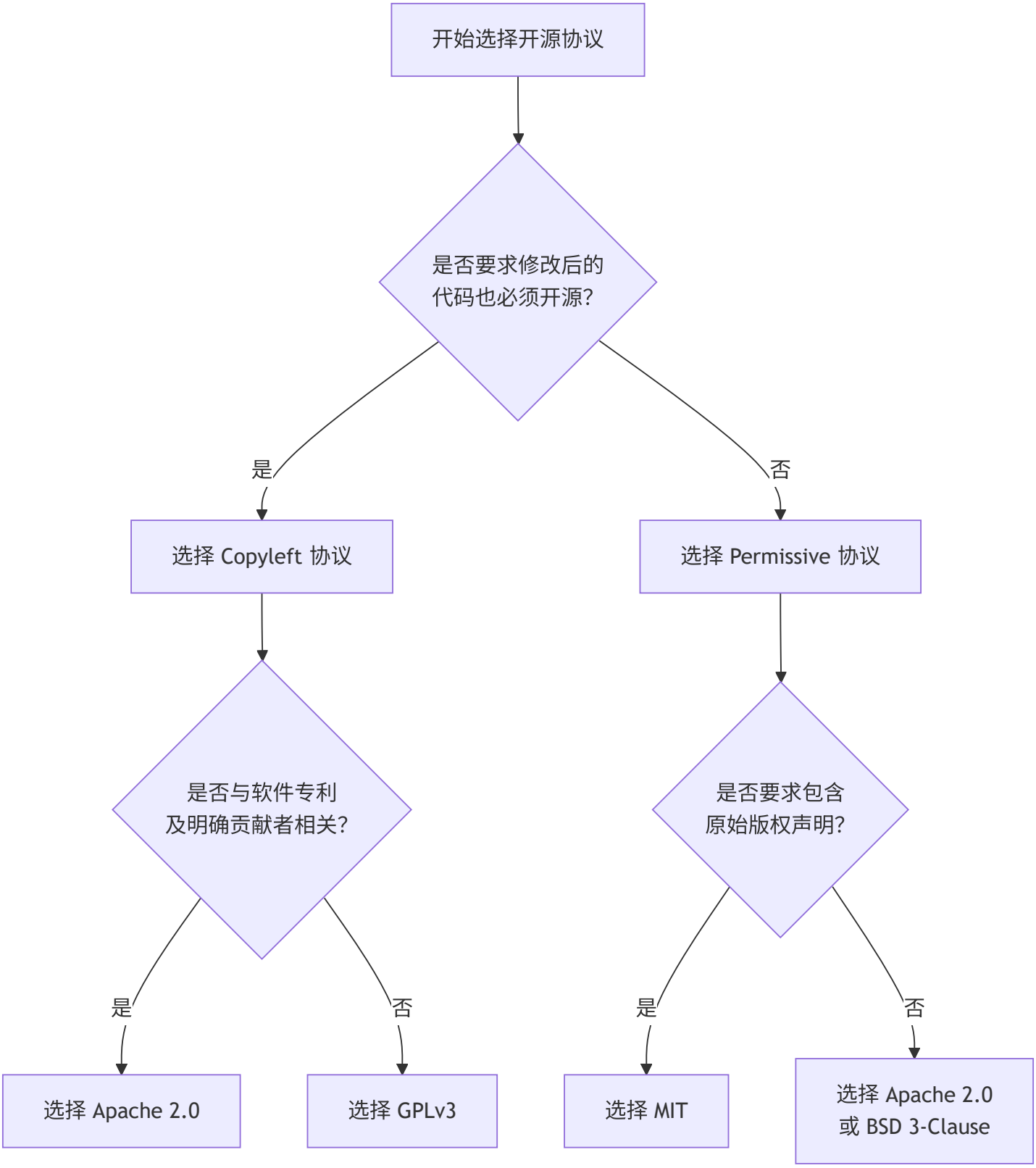
4. **MIT**
    - **核心要求**：非常宽松，几乎只有一点：在你的软件副本中包含原始版权和协议声明即可。然后你想干嘛就干嘛（私用、商用、修改、销售、闭源都可以）。
    - **适用场景**：希望被广泛采用、对用户几乎无限制的项目。如jQuery、React、Rails。
5. **BSD (2-Clause/3-Clause)**
    - **核心要求**：与MIT类似，非常宽松。BSD 3-Clause 多了一条“**禁止用作者的名字为衍生作品背书或促销**”。
    - **适用场景**：同MIT。如FreeBSD、NetBSD。

## 3.4. 如何为你的项目选择协议?

1. **我想让所有人都能用，包括闭源商业项目** -> 选择 **MIT** 或 **BSD**。
2. **我想保护我的专利，并且要求使用者注明变更** -> 选择 **Apache 2.0**。
3. **我开发的是一个库，希望闭源项目能用，但保证库本身的改进要开源** -> 选择 **LGPL**。
4. **我坚信开源精神，希望所有基于我代码的项目都必须开源** -> 选择 **GPL**。
5. **我的代码是公司的核心资产，不想让人看到** -> **不开源**，使用商业许可。

**重要提示**：一旦为项目指定了开源协议，通常不可撤销。因此选择时需要慎重。

为了更直观地理解不同类型开源协议的限制和要求，请参考下面的流程图，它可以帮助你根据项目目标选择合适的协议：




## 3.5. 总结

| 协议 | 简单要求 | 宽松程度 |
| --- | --- | --- |
| **GPL** | 用了我的代码，你的也得开源 | ⚠️ 严格 |

| 协议 | 简单要求 | 宽松程度 |
| --- | --- | --- |
| **LGPL** | 动态链接我没事，改我就得开源 | 🔸 较严格 |
| **Apache 2.0** | 保留我的声明，别用专利告我 | ✅ 宽松 |
| **MIT** | 保留我的声明，然后随你便 | ✅ ✅ 非常宽松 |
| **BSD** | 保留我的声明，别用我名字打广告 | ✅ ✅ 非常宽松 |