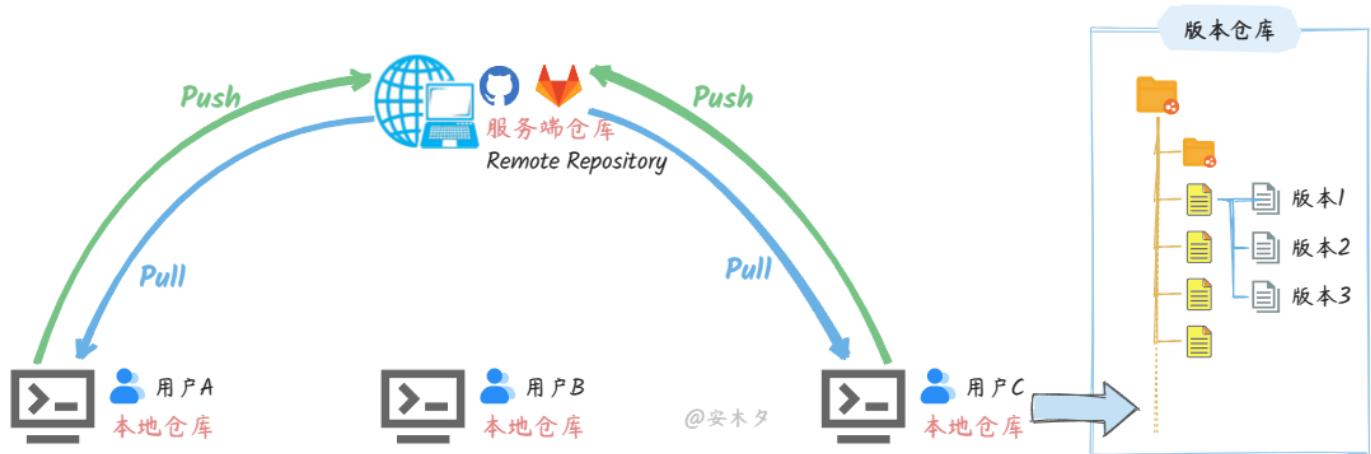


Git 教程

01、认识一下 Git！— 简介

Git 是当前最先进、最主流的分布式版本控制系统，免费、开源！核心能力就是版本控制。再具体一点，就是面向代码文件的版本控制，代码的任何修改历史都会被记录管理起来，意味着可以恢复到以前的任意时刻状态。支持跨区域多人协作编辑，是团队项目开发的必备基础，所以 Git 也就成了程序员的必备技能。



主要特点：

- 开源免费，使用广泛。
- 强大的文档（代码）的历史版本管理，直接记录完整快照（完整内容，而非差异），支持回滚、对比。
- 分布式多人协作的代码协同开发，几乎所有操作都是本地执行的，支持代码合并、代码同步。
- 简单易用的分支管理，支持高效的创建分支、合并分支。

Git 是干什么的？— 基础概念

先了解下 Git 的基本概念，及基本框架、工作流程。

2.1、Git 概念汇总

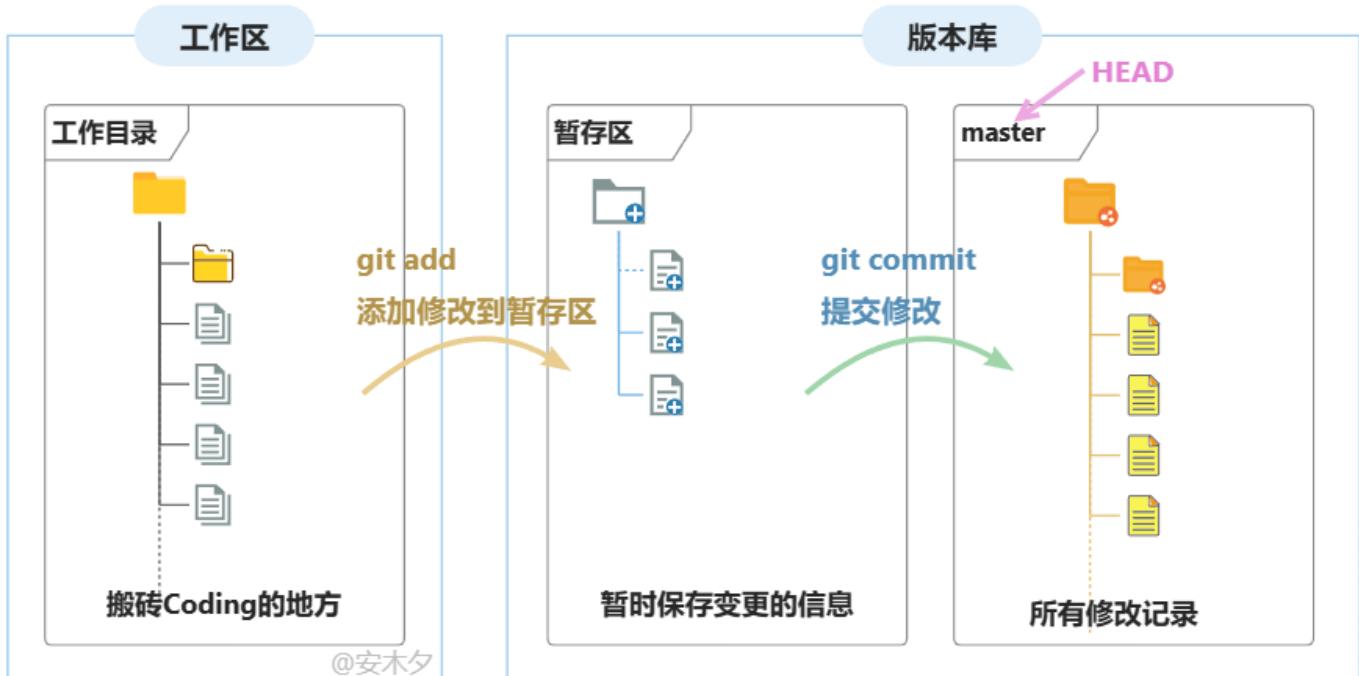
概念名称	描述
工作区 (Workspace)	就是在电脑里能看到的代码库目录，是我们搬砖的地方，新增、修改的文件会提交到暂存区
暂存区 (stage 或 index)	用于临时存放文件的修改，实际上它只是一个文件 (.git/index)，保存待提交的文件列表信息。
版本库 / 仓库 (Repository)	Git 的管理仓库，管理版本的数据库，记录文件 / 目录状态的地方，所有内容的修改记录（版本）都在这里。
服务端 / 远程仓库 (origin 或 remote)	服务端的版本库，专用的 Git 服务器，为多人共享提供服务，承担中心服务器的角色。本地版本库通过 push 指令把代码推送到服务端版本库。
本地仓库	用户机器上直接使用的版本库
分支 (Branch)	分支是从主线分离出去的“副本”，可以独立操作而互不干扰，仓库初始化就有一个默认主分支 master。
头 (HEAD)	HEAD 类似一个“指针”，指向当前活动分支的最新版本。
提交 (Commit)	把暂存区的所有变更的内容提交到当前仓库的活动分支。
推送 (Push)	将本地仓库的版本推送到服务端（远程）仓库，与他人共享。
拉取 (Pull)	从服务端（远程）仓库获取更新到本地仓库，获取他人共享的更新。
获取 (Fetch)	从服务端（远程）仓库更新，作用同拉取（Pull），区别是不会自动合并。
冲突 (Conflict)	多人对同一文件的工作副本进行更改，并将这些更改合并到仓库时就会面临冲突，需要人工合并处理。
合并 (Merge)	对有冲突的文件进行合并操作，Git 会自动合并变更内容，无法自动处理的冲突内容会提示人工处理。
标签 (Tags)	标签指的是某个分支某个特定时间点的状态，可以理解为提交记录的别名，常用来标记版本。
master (或 main)	仓库的“master”分支，默认的主分支，初始化仓库就有了。Github 上创建的仓库默认名字为“main”
origin/master	表示远程仓库（origin）的“master”分支
origin/HEAD	表示远程仓库（origin）的最新提交的位置，一般情况等于“origin/master”

2.2、工作区 / 暂存区 / 仓库

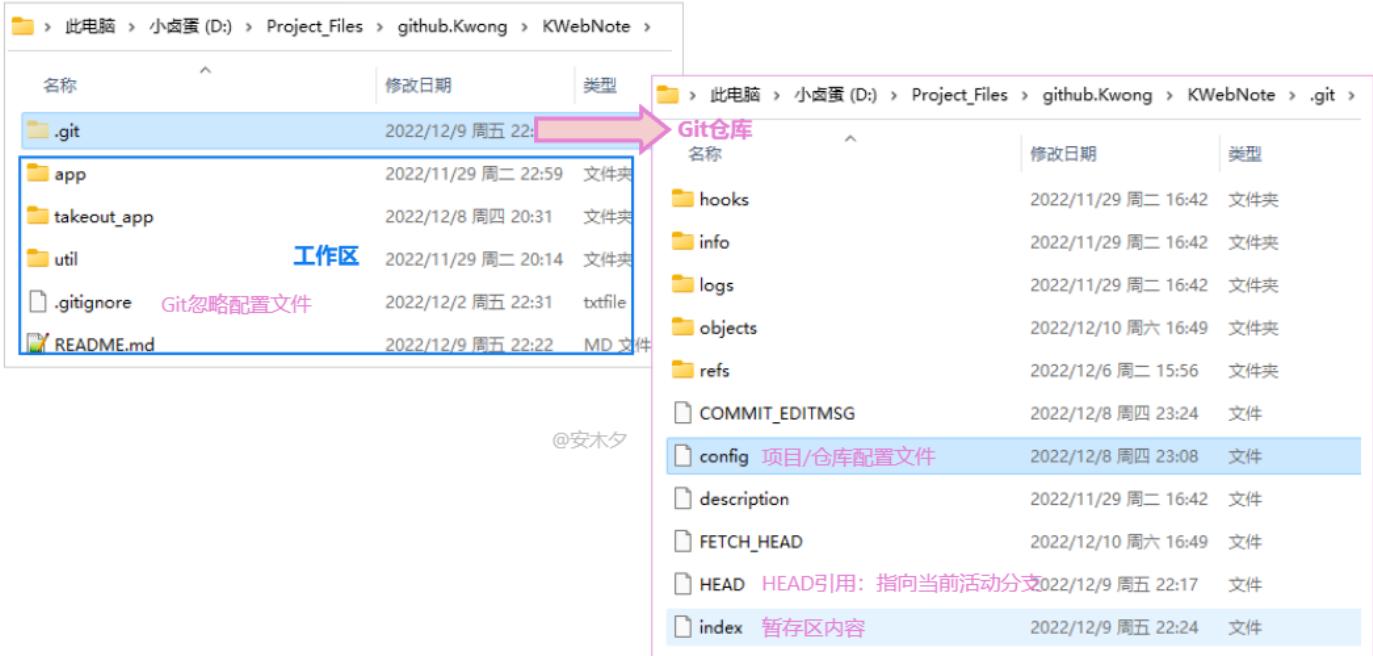


工作区
处理工作的区域
暂存区
已完成工作的临时存放区域，
等待被提交
Git 仓库
最终的存放区域
CSDN @小仓鼠真的好可爱吖

工作区、暂存区、版本库是 Git 最基本的概念，关系如下图：



- **工作区 (Workspace)** 就是在电脑里能看到的代码库目录，是我们搬砖的地方，新增、修改的文件会提交到暂存区。
 - 在这里新增文件、修改文件内容，或删除文件。
- **暂存区 (stage 或 index)** 用于临时存放文件的修改，实际上上它只是一个文件 (.git/index)，保存待提交的文件列表信息。
 - 用 `git add` 命令将工作区的修改保存到暂存区。
- **版本库 / 仓库 (Repository /rɪ'pa:zətɔ:rɪ/ 仓库)** Git 的管理仓库，管理版本的数据库，记录文件 / 目录状态的地方，所有内容的修改记录 (版本) 都在这里。就是工作区目录下的隐藏文件夹 `.git`，包含暂存区、分支、历史记录等信息。
 - 用 `git commit` 命令将暂存区的内容正式提交到版本库。
 - `master` 为仓库的默认分支 `master`， `HEAD` 是一个“指针”指向当前分支的最新提交，默认指向最新的 `master`。



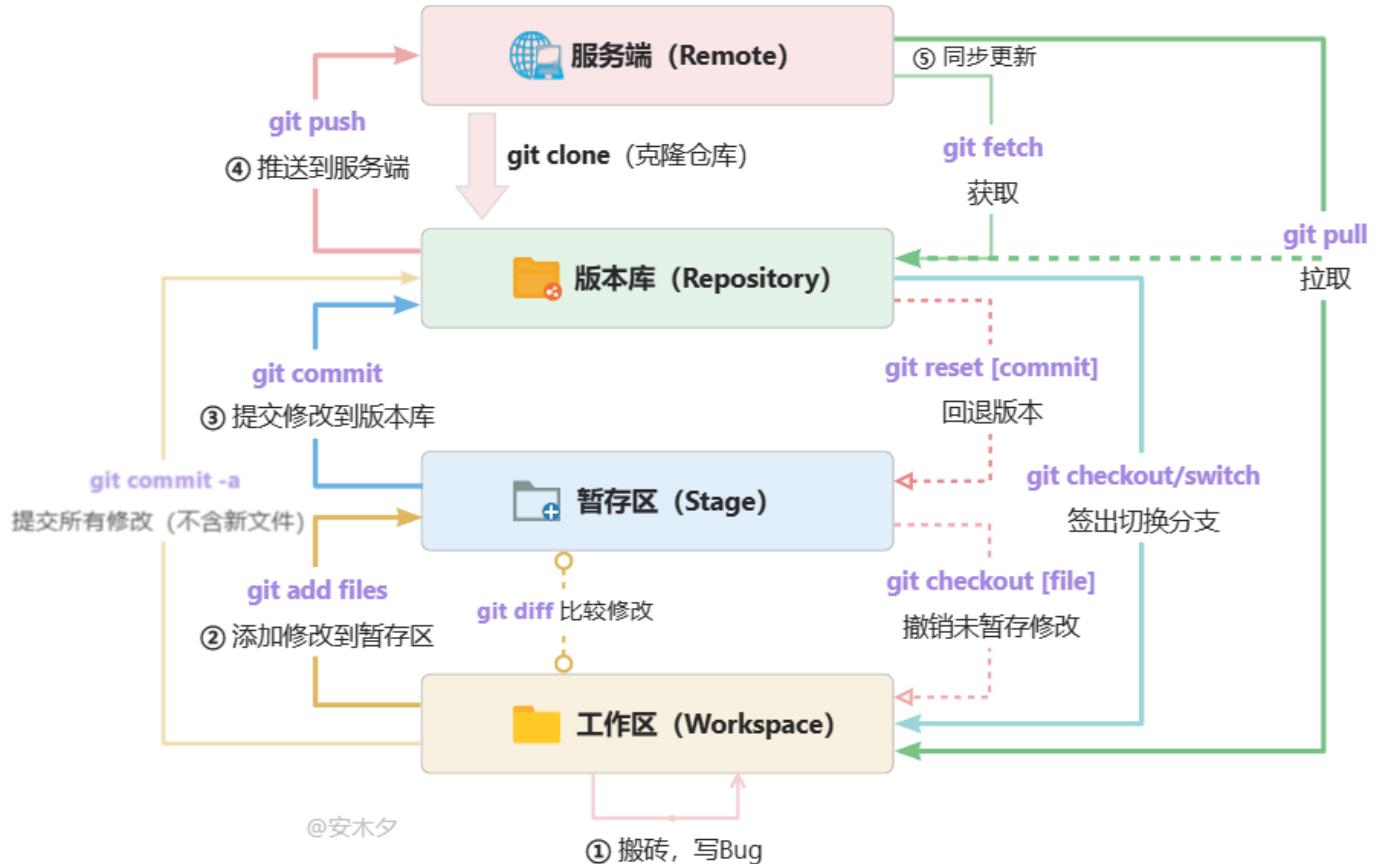
如上图，为对应本地仓库目录的结构关系。

- KWebNote 为项目目录，也就是 Git 工作区。
- 项目根目录下隐藏的 `.git` 目录就是 Git 仓库目录了，存放了所有 Git 管理的信息。
- `.git/config` 为该仓库的配置文件，可通过指令修改或直接修改。
- `index` 文件就是存放的暂存区内容。

2.3、Git 基本流程（图）

Git 的工作流程核心就下面几个步骤，掌握了就可以开始写 Bug 了。

- 0、准备仓库：创建或从服务端克隆一个仓库。
- 1、搬砖：在工作目录中添加、修改代码。
- 2、暂存（`git add`）：将需要进行版本管理的文件放入暂存区域。
- 3、提交（`git commit`）：将暂存区域的文件提交到 Git 仓库。
- 4、推送（`git push`）：将本地仓库推送到远程仓库，同步版本库。
- 5、获取更新（`fetch/pull`）：从服务端更新到本地，获取他人推送的更新，与他人协作、共享。

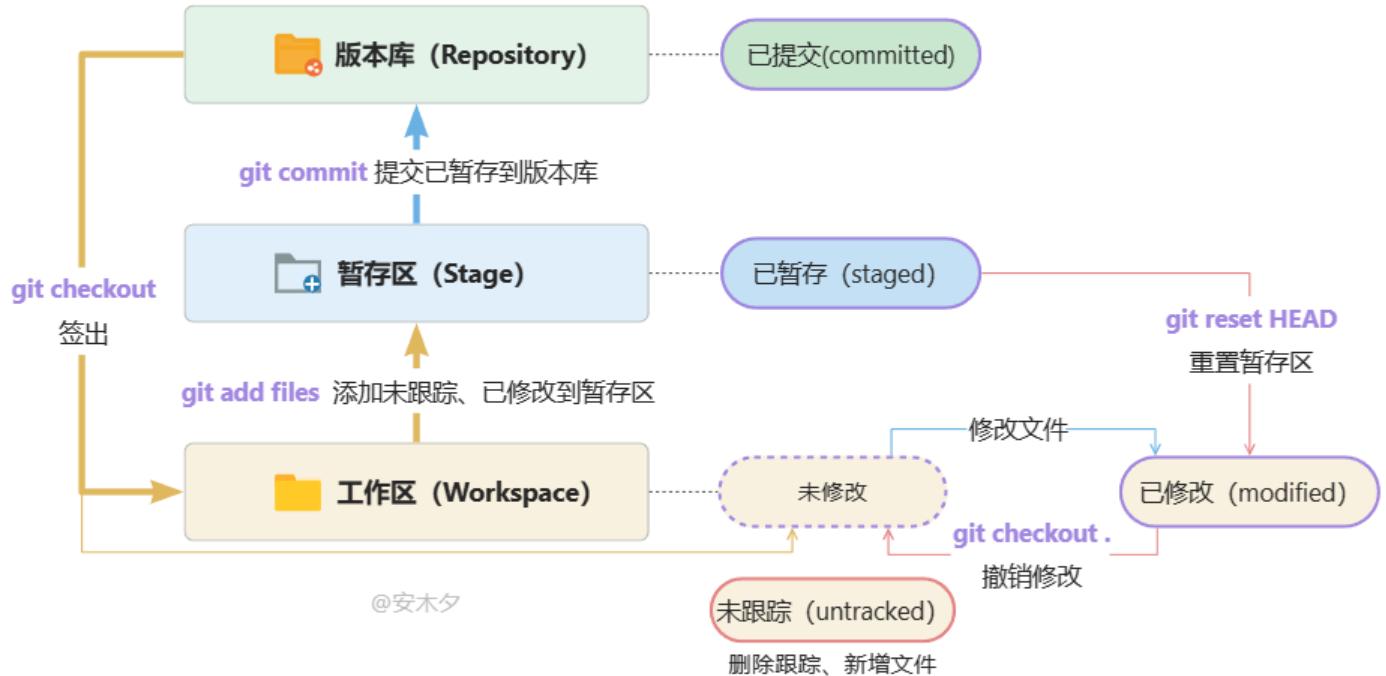


- `git commit -a` 指令省略了 `add` 到暂存区的步骤，直接提交工作区的修改内容到版本库，不包括新增的文件。
- `git fetch`、`git pull` 都是从远程服务端获取最新记录，区别是 `git pull` 多了一个步骤，就是自动合并更新工作区。
- `git checkout .`、`git checkout [file]` 会清除工作区中未添加到暂存区的修改，用暂存区内容替换工作区。
- `git checkout HEAD .`、`git checkout HEAD [file]` 会清除工作区、暂存区的修改，用 HEAD 指向的当前分支最新版本替换暂存区、工作区。
- `git diff` 用来对比不同部分之间的区别，如暂存区、工作区，最新版本与未提交内容，不同版本之间等。
- `git reset` 是专门用来撤销修改、回退版本的指令，替代上面 `checkout` 的撤销功能。

2.4、Git 状态 (图)

Git 在执行提交的时候，不是直接将工作区的修改保存到仓库，而是将暂存区域的修改保存到仓库。要提交文件，首先需要把文件加入到暂存区域中。因此，Git 管理的文件有三 (+2) 种状态：

- **未跟踪 (untracked)**：新添加的文件，或被移除跟踪的文件，未建立跟踪，通过 `git add` 添加暂存并建立跟踪。
- **未修改**：从仓库签出的文件默认状态，修改后就是“已修改”状态了。
- **已修改 (modified)**：文件被修改后的状态。
- **已暂存 (staged)**：修改、新增的文件添加到暂存区后的状态。
- **已提交 (committed)**：从暂存区提交到版本库。



03、起步：Git 安装配置

Git 官网：<https://www.git-scm.com/> 下载安装包进行安装。Git 的使用有两种方式：

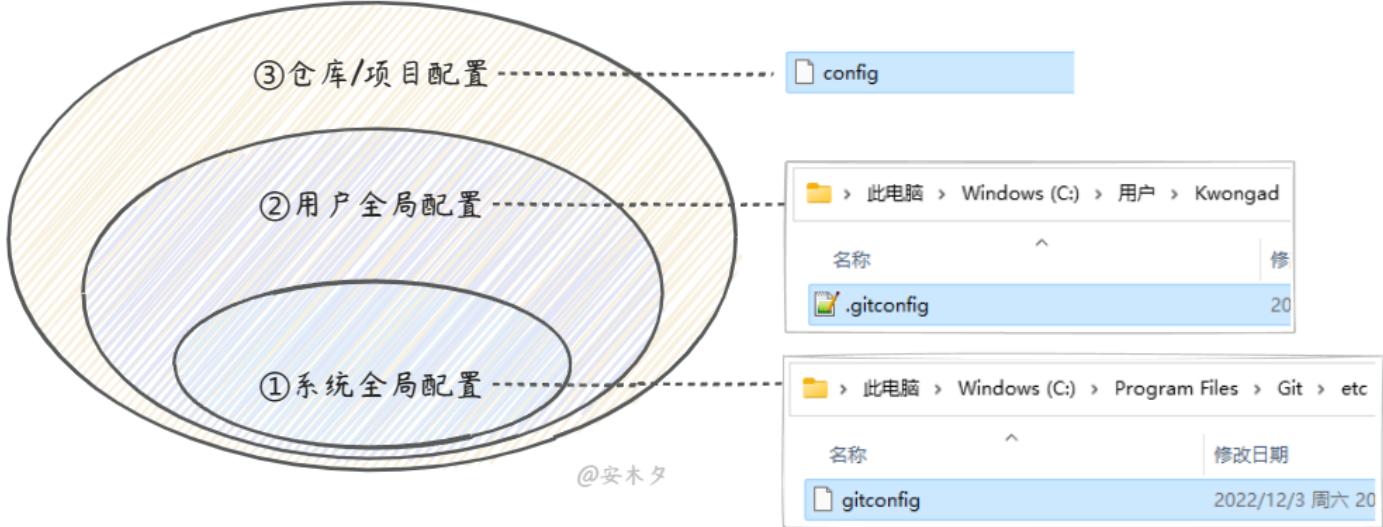
- 命令行：Git 的命令通过系统命令行工具，或 Git 提供的命令行工具运行 (`C:\Program Files\Git\git-bash.exe`)
- GUI 工具：Windows (GUI)、Mac (GUI) 工具，需单独安装，使用更简单、更易上手。指令 `git --version` 查看安装版本号

```
$ git --version
git version 2.33.0.windows.2
```

3.1、Git 的配置文件

Git 有三个主要的配置文件：三个配置文件的优先级是 `system < global < local`

- 系统全局配置 (--system)**：包含了适用于系统所有用户和所有仓库（项目）的配置信息，存放在 Git 安装目录下 `C:\Program Files\Git\etc\gitconfig`。
- 用户全局配置 (--global)**：当前系统用户的全局配置，存放用户目录：`C:\Users\[系统用户名]\.gitconfig`。
- 仓库 / 项目配置 (--local)**：仓库（项目）的特定配置，存放在项目目录下 `.git/config`。



```
#查看git配置
git config --list
git config -l

#查看系统配置
git config --system --list

#查看当前用户 ( global ) 全局配置
git config --list --global

#查看当前仓库配置信息
git config --local --list
```

仓库的配置是上面多个配置的集合：

```
$ git config --list
$ git config -l
diff.astextplain.textconv=astextplain
http.sslbackend=openssl
http.sslcainfo=C:/Program Files/Git/mingw64/ssl/certs/ca-bundle.crt
core.autocrlf=true
core.fscache=true
core.symlinks=false
pull.rebase=false
credential.helper=manager-core
credential.https://dev.azure.com.usehttppath=true
init.defaultbranch=master
user.name=Kanding
user.email=123anding@163.com
```

3.2、配置 - 初始化用户

当安装 Git 后首先要做的事情是配置你的用户信息——告诉 Git 你是谁？配置用户名、邮箱地址，每次提交文件时都会带上这个用户信息，查看历史记录时就知道是谁干的了。

配置用户信息：

```
$ git config --global user.name "Your Name"
$ git config --global user.email "email@example.com"
# 配置完后，看看用户配置文件：
$ cat 'C:\Users\Kwongad\.gitconfig'
[user]
    name = Kanding
    email = 123anding@163.com
```

- `user.name` 为用户名，`user.email` 为邮箱。
- `--global`：`config` 的参数，表示用户全局配置。如果要给特定仓库配置用户信息，则用参数 `--local` 配置即可，或直接在仓库配置文件 `.git/config` 里修改。

3.3、配置 - 忽略.gitignore

工作目录中的文件并不是全都需要纳入版本管理，如日志、临时文件、私有配置文件等不需要也不能纳入版本管理，那该怎么办呢？在工作区根目录下创建“`.gitignore`”文件，文件中配置不需要进行版本管理的文件、文件夹。“`.gitignore`”文件本身是被纳入版本管理的，可以共享。有如下规则：

- `#` 符号开头为注释。
- 可以使用 Linux 通配符。
- 星号 (*) 代表任意多个字符。
- 问号 (?) 代表一个字符。
- 方括号 ([abc]) 代表可选字符范围。
- 大括号 ({string1,string2,...}) 代表可选的字符串等。
- 感叹号 (!) 开头：表示例外规则，将不被忽略。
- 路径分隔符 (/f) 开头：· 表示要忽略根目录下的文件f。
- 路径分隔符 (/f) 结尾：· 表示要忽略文件夹f 下面的所有文件。

```
#为注释
*.txt #忽略所有“.txt”结尾的文件
!lib.txt #lib.txt除外
/temp #仅忽略项目根目录下的temp文件，不包括其它目录下的temp，如不包括“src/temp”
build/ #忽略build/目录下的所有文件
doc/*.txt #会忽略 doc/notes.txt 但不包括 doc/server/arch.txt
```

各种语言项目的常用`.gitignore` 文件配置：<https://github.com/github/gitignore>

04、Git 的 GUI 工具们

如果不想要用命令行工具，完全可以安装一个 Git 的 GUI 工具，用的更简单、更舒服。不用记那么多命令了，极易上手，不过 Git 基础还是需要学习了解一下的。

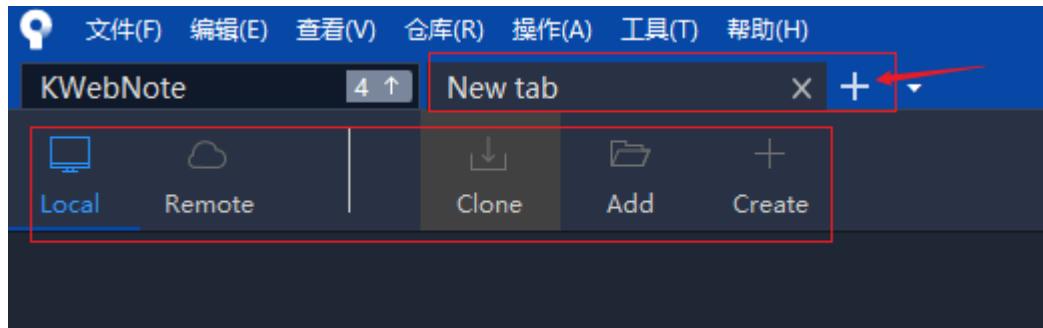
- TortoiseGit：小乌龟，SVN 时代就很流行的代码管理 GUI 利器。

- 只有 Windows 版本，支持中文，需要单独下载安装中文语言包。
- 开源，免费，与文件管理器的良好集成。
- 内置冲突对比解决工具。
- Sourcetree：SourceTree 是老牌的 Git GUI 管理工具了，也号称是最好用的 Git GUI 工具。
 - 适用于 Windows 和 Mac 系统，内置中文版，自动识别语言。
 - 免费、功能强大，使用简单。
 - 功能丰富，基本操作和高级操作都设计得非常流畅，适合初学者上手，支持 Git Flow。
 - 无冲突对比工具，支持配置第三方组件。
- GitHub Desktop：Github 官方出品的 Git 管理工具。
- GitKraken：GitKraken 是一个跨平台 GUI Git 客户端，有免费版，专业版和企业版，这些版本启用了不同的功能。

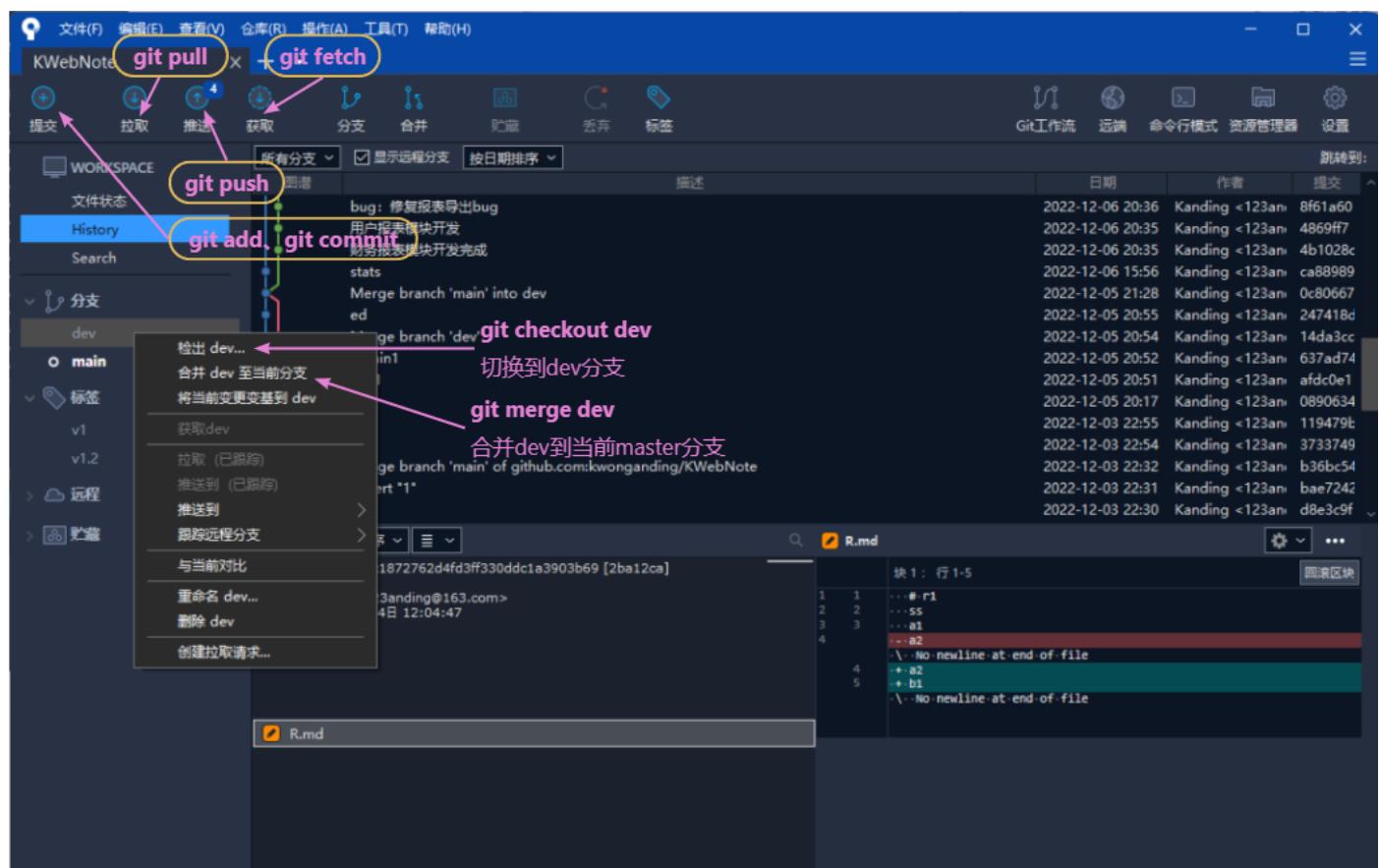
4.1、SourceTree

SourceTree 的官网 下载安装包，支持 Window、Mac 系统，按照提示完成安装。

- SourceTree 支持管理多个仓库，通过 + 按钮，可选择多种方式添加仓库。

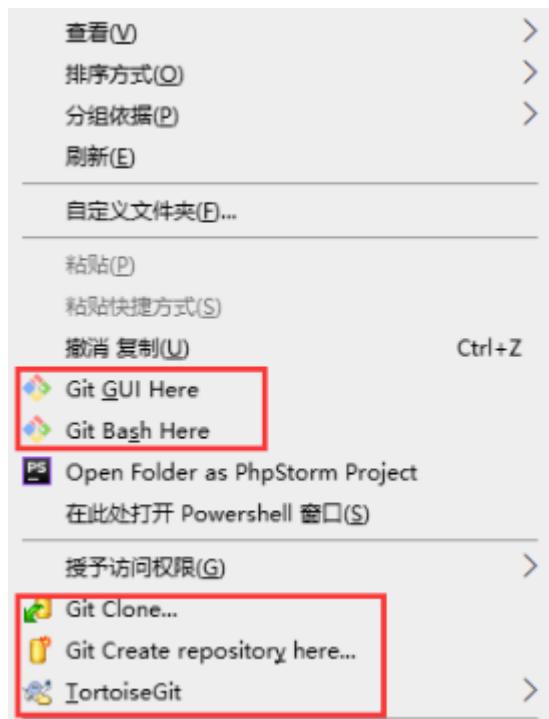


- 然后就是可视化的仓库管理了，不用记住繁琐的指令（参数）了，可视化操作。



4.2、TortoiseGit

TortoiseGit 官网下载安装包，及中文语言包，按照提示完成安装。小乌龟的 Git 是集成到操作系统里的，直接右键文件夹就可以进行 Git 操作了。



- 先进入设置：右键文件夹菜单 --> TortoiseGit --> Settings 进入设置，设置中文语言。
- 小乌龟的各种 Git 操作都在右键菜单了，深度集成到了操作系统的资源管理器中了，文件图标也是有 Git 状态的，比较容易分辨。

勾 .settings	2019/12/4 15:21	文件夹	
勾 src	2019/12/4 15:21	文件夹	
勾 .classpath	2019/12/4 15:21	CLASSPATH 文件	2 KB
勾 .factorypath	2019/12/4 15:21	FACTORYPATH ...	4 KB
勾 .gitignore	2019/12/4 15:21	文本文档	1 KB
勾 .project	2019/12/4 15:21	PROJECT 文件	2 KB
勾 pom	2019/12/4 15:21	XML 文档	2 KB
勾 test1	2019/12/4 17:55	文本文档	1 KB

Path	Extension	Status	Lines added	Lines removed
.gitignore	.gitignore	Deleted	0	1

Showing 24 revision(s), from revision 4d18de58 to revision 135802bb - 1 revision(s) selected, 0 file(s) selected; line: 0(+) 1(-) files: modified = 0 added = 0

Show Whole Project All Branches Refresh Statistics Walk Behaviour View Help OK

4.3、VSCode 中的 Git

VSCode 自带的 Git 工具基本已经可以满足日常使用了，既有可视化功能，也能敲命令，习惯了不就不用安装其他 GUI 工具了。不过还是可以再安装一些 VSCode 插件，来增强 Git 功能。

- **GitLens**：在团队项目开发中非常实用，必备！！！用于快速查看代码提交历史记录，在代码上会显示最近的修改信息，包括提交者，只就这一点就值得推荐了。



- **Git History**：可以轻松快速浏览 Git 文件操作历史记录的工具，可视化展示，操作简单。

05、Git 使用入门

5.1、创建仓库

创建本地仓库的方法有两种：

一种是创建全新的仓库：`git init`，会在当前目录初始化创建仓库。另一种是克隆远程仓库：`git clone [url]`。

```
# 准备一个文件夹“KwebNote”作为仓库目录，命令行进入该文件夹  
Kwongad@Kwongad-T14 MINGW64 ~  
$ cd d:  
Kwongad@Kwongad-T14 MINGW64 /d  
$ cd Project_Files  
Kwongad@Kwongad-T14 MINGW64 /d/Project_Files  
# 多次cd指令进入到仓库目录KwebNote：“cd <目录名称>”指令进入目录，“cd ..”返回上级目录（有空格）  
Kwongad@Kwongad-T14 MINGW64 /d/Project_Files/github.kwong/KwebNote  
  
# 开始初始化项目，也可指定目录：git init [文件目录]  
$ git init  
Initialized empty Git repository in D:/Project_Files/github.Kwong/KwebNote/.git/
```

注意：Git 指令的执行，都需在仓库目录下。

创建完多出了一个被隐藏的 `.git` 目录，这就是本地仓库 Git 的工作场所。

名称	修改日期	类型
📁 .git	2022/12/14 周三 21:49	文件夹

克隆远程仓库，如在 github 上创建的仓库 “<https://github.com/kwonganding/KWebNote.git>”

```
$ git clone 'https://github.com/kwonganding/KWebNote.git'  
Cloning into 'KWebNote'...  
remote: Enumerating objects: 108, done.  
remote: Counting objects: 100% (108/108), done.  
remote: Compressing objects: 100% (60/60), done.  
remote: Total 108 (delta 48), reused 88 (delta 34), pack-reused 0  
Receiving objects: 100% (108/108), 9.36 KiB | 736.00 KiB/s, done.  
Resolving deltas: 100% (48/48), done.
```

会在当前目录下创建 “KWebNote” 项目目录。

名称	修改日期	类型	大小
.git	2022/12/14 周三 22:20	文件夹	
app	2022/11/29 周二 22:59	文件夹	
takeout_app	2022/12/12 周一 19:02	文件夹	
util	2022/11/29 周二 20:14	文件夹	
.gitignore	2022/12/2 周五 22:31	txtfile	1 KE
R.md	2022/12/14 周三 22:20	MD 文件	1 KE
README.md	2022/12/13 周二 22:54	MD 文件	1 KE

5.2、暂存区 add

可以简单理解为，`git add` 命令就是把要提交的所有修改放到暂存区（Stage），然后，执行 `git commit` 就可以一次性把暂存区的所有修改提交到仓库。

指令	描述
<code>git add [file1] [file2]</code>	添加文件到暂存区，包括修改的文件、新增的文件
<code>git add [dir]</code>	同上，添加目录到暂存区，包括子目录
<code>git add .</code>	同上，添加所有修改、新增文件（未跟踪）到暂存区
<code>git rm [file]</code>	删除工作区文件，并且将这次删除放入暂存区

```
# 添加指定文件到暂存区，包括被修改的文件
$ git add [file1] [file2] ...

# 添加当前目录的所有文件到暂存区
$ git add .

# 删除工作区文件，并且将这次删除放入暂存区
$ git rm [file1] [file2] ...

# 改名文件，并且将这个改名放入暂存区
$ git mv [file-original] [file-renamed]
```

修改文件“R.md”，未暂存：

```
$ git status
On branch main
Your branch is ahead of 'origin/main' by 4 commits.
  (use "git push" to publish your local commits)

Changes not staged for commit: 未暂存
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   R.md
```

执行 `git add .` 暂存：

```
$ git status
On branch main
Your branch is ahead of 'origin/main' by 4 commits.
  (use "git push" to publish your local commits)

Changes to be committed: (已暂存) 待提交
  (use "git restore --staged <file>..." to unstage)
    modified:   R.md
```

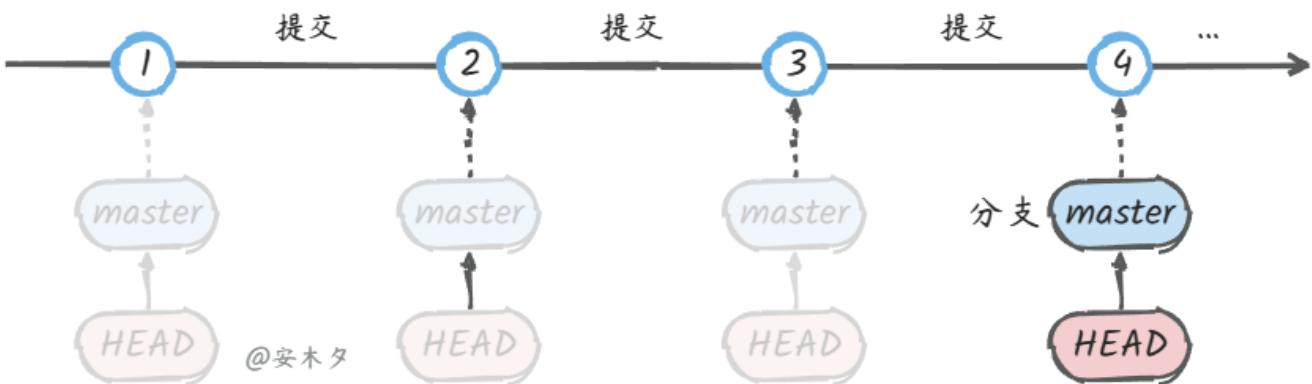
提交 commit - 记录

`git commit` 提交是以时间顺序排列被保存到数据库中的，就如游戏关卡一样，每一次提交（commit）就会产生一条记录：id + 描述 + 快照内容。

- **commit id**：根据修改的文件内容采用摘要算法（SHA1）计算出不重复的 40 位字符，这么长是因为 Git 是分布式的，要保证唯一性、完整性，一般本地指令中可以只用前几位（6）。即使多年以后，依然可通过 id 找到曾经的任何内容和变动，再也不用担心丢失了。
- **描述**：针对本次提交的描述说明，建议准确填写，就跟代码中的注释一样，很重要。
- **快照**：就是完整的版本文件，以对象树的结构存在仓库下 `\.git\objects` 目录里，这也是 Git 效率高的秘诀之一。

- SHA1 是一种哈希算法，可以用来生成数据摘要
- Git 不适合大的非文本文件，会影响计算摘要、快照的性能。

多个提交就形成了一条时间线，每次提交完，会移动当前分支 `master`、`HEAD` 的“指针”位置。



Sourcetree 上的历史记录：

图谱	描述
● main ● origin/main ● origin/HEAD	报表新增导入功能 bug: 修复报表导出bug 用户报表模块开发 财务报表模块开发完成
	一般情况，每完成一个小功能、一个 Bu 就可以提交一次，这样会形成比较清晰的历史记录。
	指令：

指令	描述
git commit -m '说明'	提交变更，参数 <code>-m</code> 设置提交的描述信息，应该正确提交，不带该参数会进入说明编辑模式
git commit -a	参数 <code>-a</code> ，表示直接从工作区提交到版本库，略过了 <code>git add</code> 步骤，不包括新增的文件
git commit [file]	提交暂存区的指定文件到仓库区
git commit --amend -m	使用一次新的 <code>commit</code> ，替代上一次提交，会修改 <code>commit</code> 的 hash 值 (id)
git log -n20	查看日志 (最近 20 条)，不带参数 <code>-n</code> 则显示所有日志
git log -n20 --oneline	参数 “ <code>--oneline</code> ” 可以让日志输出更简洁 (一行)
git log -n20 --graph	参数 “ <code>--graph</code> ” 可视化显示分支关系
git log --follow [file]	显示某个文件的版本历史
git blame [file]	以列表形式显示指定文件的修改记录
git reflog	查看所有可用的历史版本记录 (实际是 HEAD 变更记录)，包含被回退的记录 (重要)
git status	查看本地仓库状态，比较常用的指令，加参数 <code>-s</code> 简洁模式



通过 `git log` 指令可以查看提交记录日志，可以很方便的查看每次提交修改了哪些文件，改了哪些内容，从而进行恢复等操作。

```

# 提交暂存区到仓库区
$ git commit -m [message]
# 提交所有修改到仓库
$ git commit -a -m'修改README的版权信息'

# 提交暂存区的指定文件到仓库区
$ git commit [file1] [file2] ... -m [message]

# 使用一次新的commit，替代上一次提交
# 如果代码没有任何新变化，则用来改写上一次commit的提交信息
$ git commit --amend -m [message]

$ git log -n2
commit 412b56448568ff362ef312507e78797befcf2846 (HEAD -> main)
Author: Kanding <123anding@163.com>
Date:   Thu Dec 1 19:02:22 2022 +0800

commit c0ef58e3738f7d54545d8c13d603cddeee328fcb
Author: Kanding <123anding@163.com>
Date:   Thu Dec 1 16:52:56 2022 +0800

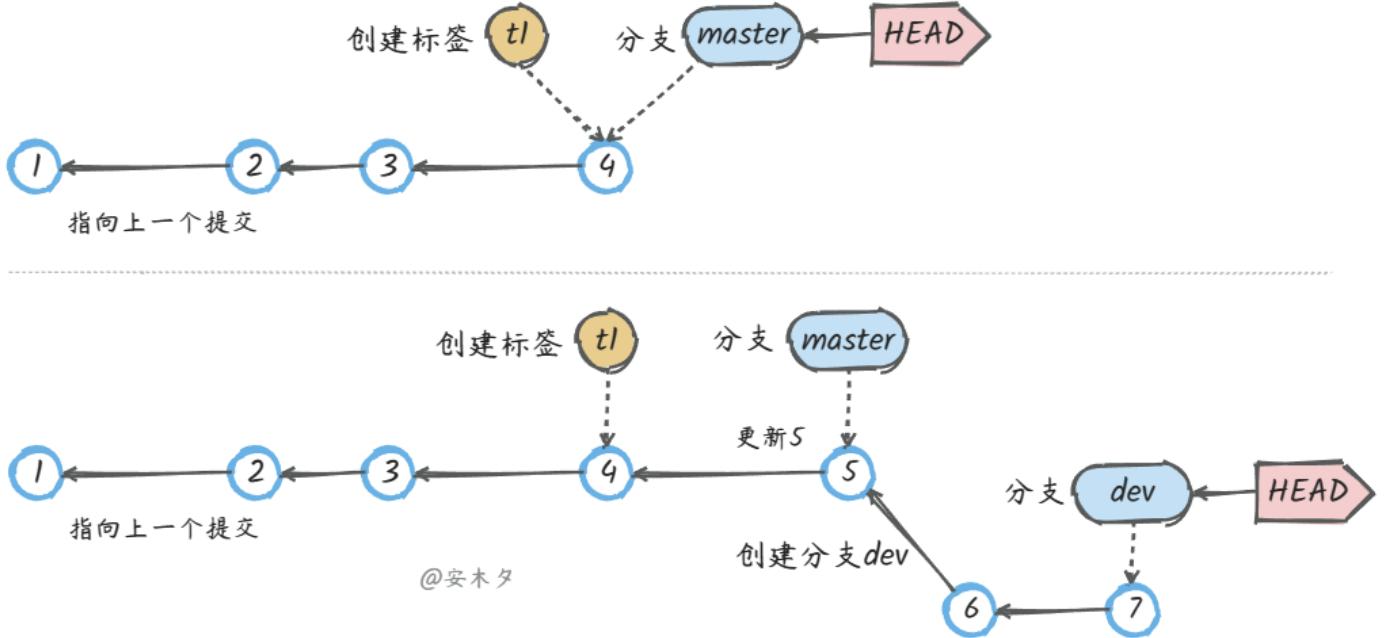
# 用参数“--oneline”可以让日志输出更简洁（一行）
$ git log -n2 --oneline
5444126 (HEAD -> main, origin/main, origin/HEAD) Update README.md
228362e Merge branch 'main' of github.com:kwonganding/KWebNote

```

5.4、Git 的“指针”引用们

Git 中最重要的就是提交记录了，其他如**标签**、**分支**、**HEAD** 都对提交记录的“指针”引用，指向这些提交记录，理解这一点很重要。

- 提交记录之间也存在“指针”引用，每个提交会指向其上一个提交。
- **标签** 就是对某一个提交记录的固定“指针”引用，取一个别名更容易记忆一些关键节点。存储在工作区根目录下 `.git\refs\tags`。
- **分支** 也是指向某一个提交记录的“指针”引用，“指针”位置可变，如提交、更新、回滚。存储在工作区根目录下 `.git\refs\heads`。
- **HEAD**：指向当前活动分支（最新提交）的一个“指针”引用，存在在 `“.git/HEAD”` 文件中，存储的内容为 `“ref: refs/heads/master”`。



上图中：

- **HEAD** 始终指向当前活动分支，多个分支只能有一个处于活动状态。
- 标签 **t1** 在某一个提交上创建后，就不会变了。而分支、**HEAD** 的位置会改变。打开这些文件内容看看，就更容易理解这些“指针”的真面目了。

```
# tag
$ git tag -a 'v1' -m'v1版本'
$ cat .git/refs/tags/v1
a2e2c9cae35e176cf61e96ad9d5a929cfb82461

# main分支指向最新的提交
$ cat .git/refs/heads/main
8f4244550c2b6c23a543b741c362b13768442090

# HEAD指向当前活动分支
$ cat .git/HEAD
ref: refs/heads/main

# 切换到dev分支，HEAD指向了dev
$ git switch dev
Switched to branch 'dev'
$ cat .git/HEAD
ref: refs/heads/dev
```

这里的主分支名字为“**main**”，是因为该仓库是从 Github 上克隆的，Github 上创建的仓库默认主分支名字就是“**main**”，本地创建的仓库默认主分支名字为“**master**”。

“指针”引用：之所以用引号的“指针”，是为了便于统一和理解。和指针原理类似，都是一个指向，只是实际上可能更复杂一点，且不同的“指针”引用会有区别。

5.5、提交的唯一标识 id, HEAD~n 是什么意思 ?

每一个提交都有一个唯一标识，主要就是提交的 hash 值 commit id，在很多指令中会用到，如版本回退、拣选提交等，需要指定一个提交。那标识唯一提交有两种方式：

- 首先就是 commit id，一个 40 位编码，指令中使用的时候可以只输入前几位（6 位）即可。
- 还有一种就是 HEAD~n，是基于当前 HEAD 位置的一个相对坐标。
 - HEAD 表示当前分支的最新版本，是比较常用的参数。
 - HEAD^ 上一个版本，HEAD^^ 上上一个版本。
 - HEAD~ 或 HEAD~1 表示上一个版本，以此类推，HEAD~10 为最近第 10 个版本。
 - HEAD@{2} 在 git reflog 日志中标记的提交记录索引。通过 git log、git reflog 可以查看历史日志，可以看每次提交的唯一编号（hash）。区别是 git reflog 可以查看所有操作的记录（实际是 HEAD 变更记录），包括被撤销回退的提交记录。

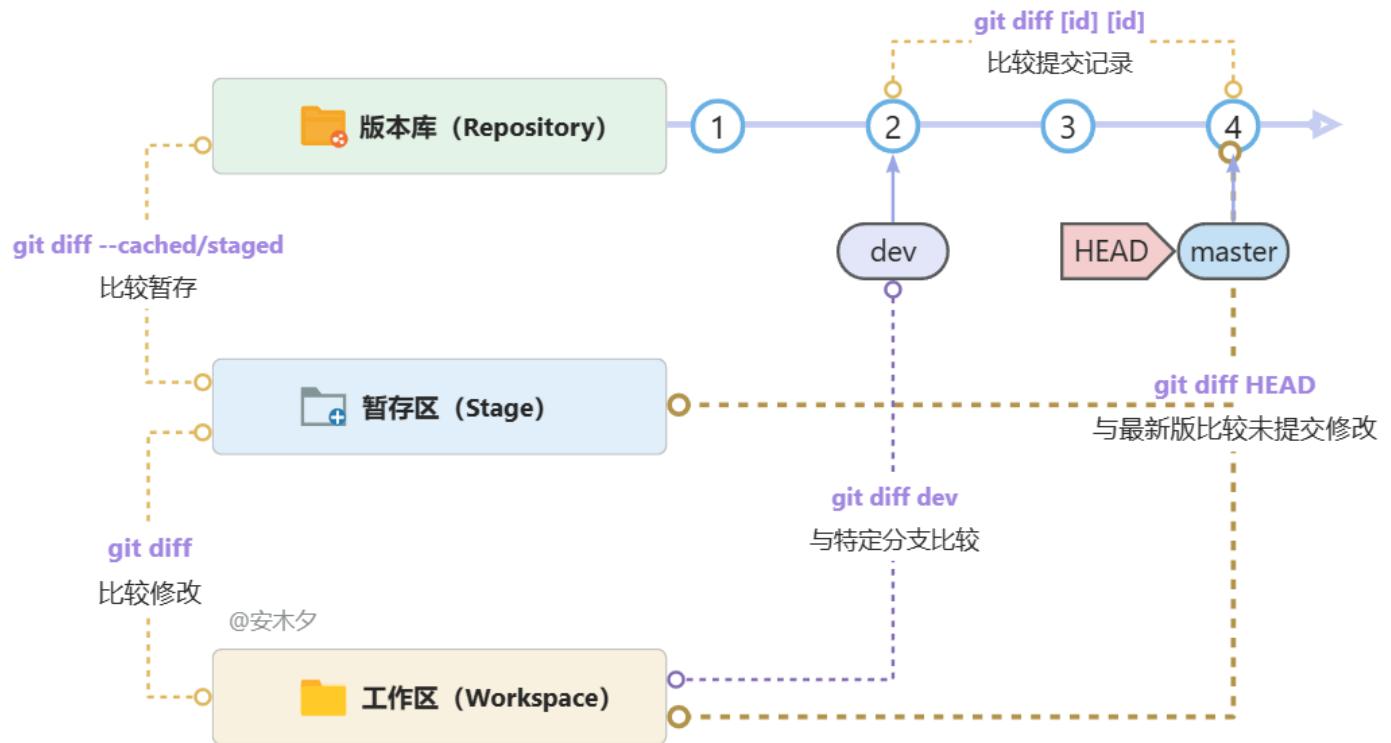
```
$ git reflog -n10
5acc914 (HEAD -> main) HEAD@{0}: reset: moving to HEAD~
738748b (dev) HEAD@{1}: reset: moving to HEAD~
9312c3e HEAD@{2}: reset: moving to HEAD~
db03fcf HEAD@{3}: reset: moving to HEAD~
1b81fb3 HEAD@{4}: reset: moving to HEAD~
41ea423 HEAD@{5}: reset: moving to HEAD~
d3e15f9 HEAD@{6}: reset: moving to d3e15f9
1b81fb3 HEAD@{7}: reset: moving to HEAD~1
41ea423 HEAD@{8}: reset: moving to HEAD~
d3e15f9 HEAD@{9}: reset: moving to HEAD~
```

比较 diff

git diff 用来比较不同文件版本之间的差异。

指令	描述
git diff	查看暂存区和工作区的差异
git diff [file]	同上，指定文件
git diff --cached	查看已暂存的改动，就是暂存区与新版本 HEAD 进行比较
git diff --staged	同上
git diff --cached [file]	同上，指定文件
git diff HEAD	查看已暂存的 + 未暂存的所有改动，就是与最新版本 HEAD 进行比较
git diff HEAD~	同上，与上一个版本比较。HEAD~ 表示上一个版本，HEAD~10 为最近第 10 个版本
git diff [id] [id]	查看两次提交之间的差异
git diff [branch]	查看工作区和分支直接的差异

画个图更清晰些：



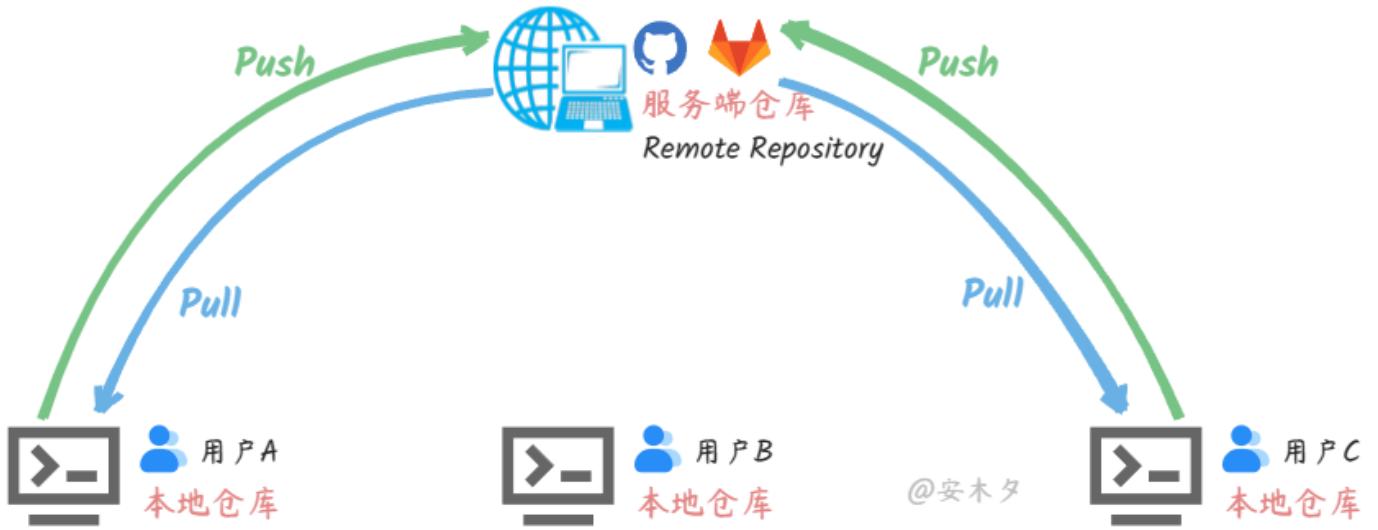
```
# 查看文件的修改
$ git diff README.md

# 查看两次提交的差异
$ git diff 8f4244 1da22

# 显示今天你写了多少行代码：工作区+暂存区
$ git diff --shortstat "@{0 day ago}"
```

06、远程仓库

Git 作为分布式的版本管理系统，每个终端都有自己的 Git 仓库。但团队协作还需一个中间仓库，作为中心，同步各个仓库。于是服务端（远程）仓库就来承担这个职责，服务端不仅有仓库，还配套相关管理功能。



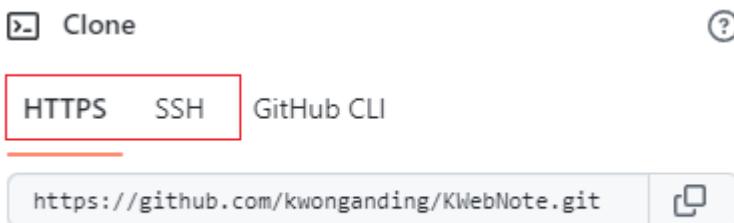
可以用公共的 Git 服务器，也可以自己搭建一套 Git 服务器。

公共 Git 服务器，如 Github、Gitlab、码云 Gitee、腾讯 Coding 等。搭建私有 Git 服务器，如开源的 Gitlab、Gitea、等。

6.1、远程用户登录

Git 服务器一般提供两种登录验证方式：

- **HTTPS**：基于 HTTPS 连接，使用用户名、密码身份验证。
 - 每次都要输入用户名、密码，当然可以记住。
 - 地址形式：<https://github.com/kwonganding/KWebNote.git>
- **SSL**：采用 SSL 通信协议，基于公私钥进行身份验证，所以需要额外配置公私秘钥。
 - 不用每次输入用户名、密码，比较推荐的方法。
 - 地址形式：<git@github.com:kwonganding/KWebNote.git>



```
#查看当前远程仓库使用的那种协议连接：
$ git remote -v
origin  git@github.com:kwonganding/KWebNote.git (fetch)
origin  https://github.com/kwonganding/KWebNote.git (push)
```

```
# 更改为https地址，即可切换连接模式。还需要禁用掉SSL，才能正常使用https管理git
git config --global http.sslVerify false
```

远程用户登录：HTTPS

基于 HTTPS 的地址连接远程仓库，Github 的共有仓库克隆、拉取（pull）是不需要验证的。

 Clone



HTTPS SSH GitHub CLI

<https://github.com/kwonganding/KWebNote.git>



Use Git or checkout with SVN using the web URL.

```
$ git clone 'https://github.com/kwonganding/KWebNote.git'  
Cloning into 'KWebNote'...  
  
# 仓库配置文件“.git/config”  
[remote "origin"]  
  url = https://github.com/kwonganding/KWebNote.git  
  fetch = +refs/heads/*:refs/remotes/origin/*  
  pushurl = https://github.com/kwonganding/KWebNote.git
```

推送 (push) 代码的时候就会提示输入用户名、密码了，否则无法提交。记住用户密码的方式有两种：

- **URL 地址配置**：在原本 URL 地址上加上用户名、密码，<https://> 后加 **用户名:密码@**

```
# 直接修改仓库的配置文件“.git/config”  
[remote "origin"]  
  url = https://用户名:密码@github.com/kwonganding/KWebNote.git  
  fetch = +refs/heads/*:refs/remotes/origin/*  
  pushurl = https://github.com/kwonganding/KWebNote.git
```

- **本地缓存**：会创建一个缓存文件 **.git-credentials**，存储输入的用户名、密码。

```
# 参数“--global”全局有效，也可以针对仓库设置“--local”  
# store 表示永久存储，也可以设置临时存储  
git config --global credential.helper store  
  
# 存储内容如下，打开文件“仓库\\.git\\.git-credentials”  
https://kwonganding:[加密内容可见]@github.com
```

远程用户登录：SSH

HTTPS SSH GitHub CLI

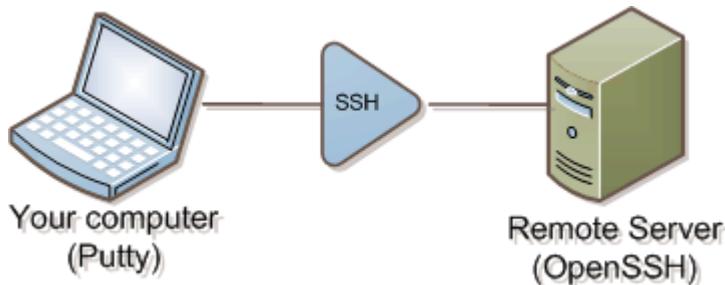
<git@github.com:kwonganding/KWebNote.git>



Use a password-protected SSH key.

SSH (Secure Shell，安全外壳) 是一种网络安全协议，通过加密和认证机制实现安全的访问和文件传输等业务，多用来进行远程登录、数据传输。SSH 通过公钥、私钥非对称加密数据，所以 SSH 需要生成一个公私钥

对，公钥放服务器上，私有自己留着进行认证。



- 生成公私钥：通过 Git 指令 `ssh-keygen -t rsa` 生成公私钥，一路回车即可完成。生成在“`C:\Users\用户名\.ssh`”目录下，文件 `id_rsa.pub` 的内容就是公钥。

A screenshot of a Windows File Explorer window showing the contents of the `.ssh` directory. The files listed are:

名称	修改日期	类型	大小
<code>id_rsa</code>	2022/11/30 周三 16:37	文件	3 KB
<code>id_rsa.pub</code>	2022/11/30 周三 16:37	PUB 文件	1 KB
<code>known_hosts</code>	2022/11/30 周三 21:13	文件	1 KB
<code>known_hosts.old</code>	2022/11/30 周三 21:13	OLD 文件	1 KB

- 配置公钥：打开 `id_rsa.pub` 文件，复制内容。Github 上，打开 Setting > SSH and GPG keys > SSH keys > 按钮 New SSH key，标题 (Title) 随意，秘钥内容粘贴进去即可。

A screenshot of a GitHub account settings page for "Kanding". The sidebar on the left shows options like Public profile, Account, Appearance, Accessibility, Notifications, Access, Billing and plans, Emails, Password and authentication, Sessions, and SSH and GPG keys (which is selected). The main area is titled "SSH keys" and contains the following text: "This is a list of SSH keys associated with your account. Remove any keys that you do not recognize." Below this is a table titled "Authentication Keys" showing one entry:

	<code>id_rsa</code>	SHA256:641CTwUeQSFnPmwAF1UzKZcvGaC3/Zgoh zOHHy+ni90	Added on Nov 30, 2022	Delete
Last used within the last week — Read/write				

At the bottom of the "SSH keys" page, there is a note: "Check out our guide to generating SSH keys or troubleshoot common SSH problems."

SSH 配置完后，可用 `ssh -T git@github.com` 来检测是否连接成功。

```
$ ssh -T git@github.com
Hi kwonganding! You've successfully authenticated, but GitHub does not provide shell
access.
```

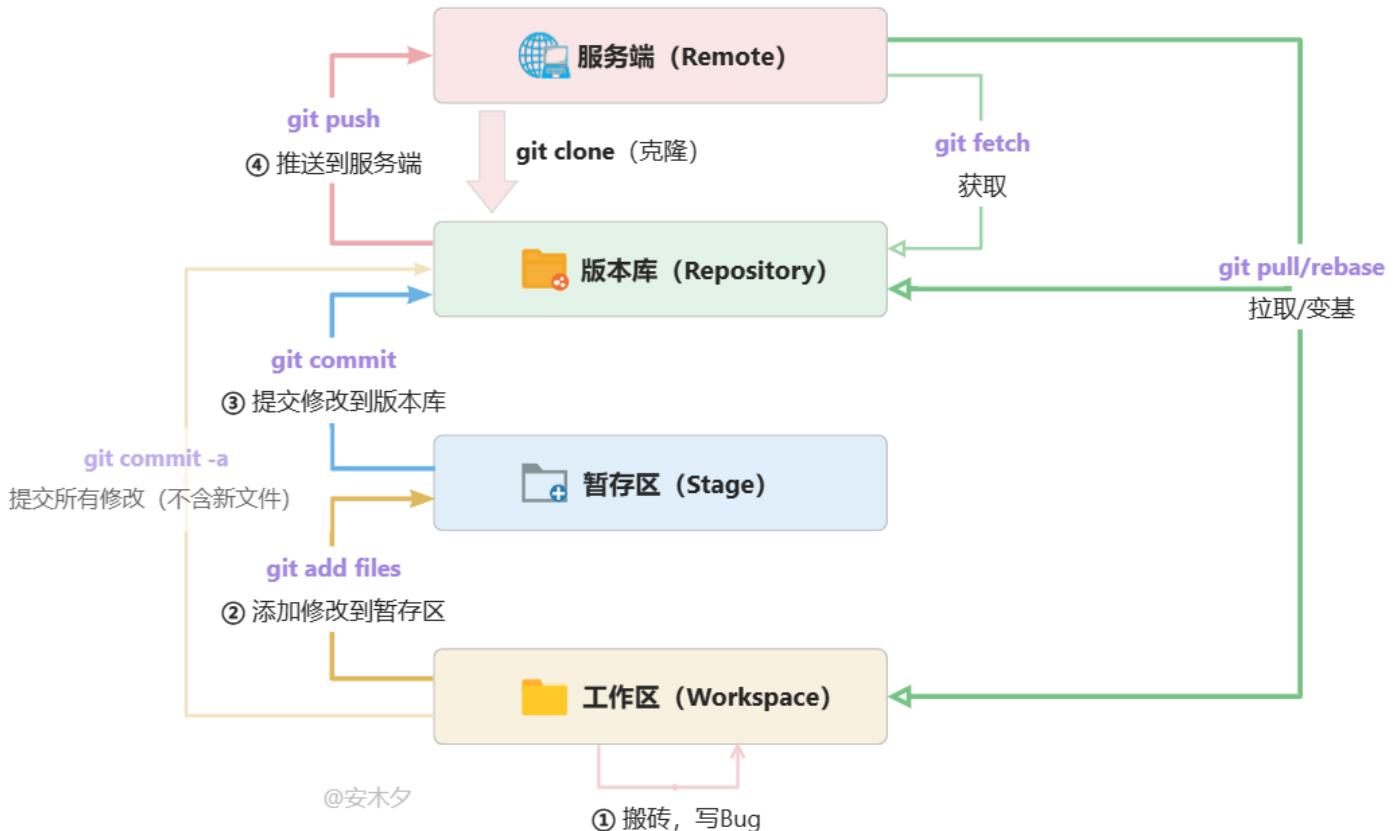
6.2、远程仓库指令

指令	描述
git clone [git 地址]	从远程仓库克隆到本地 (当前目录)
git remote -v	查看所有远程仓库，不带参数 <code>-v</code> 只显示名称
git remote show [remote]	显示某个远程仓库的信息
git remote add [name] [url]	增加一个新的远程仓库，并命名
git remote rename [old] [new]	修改远程仓库名称
git pull [remote] [branch]	取回远程仓库的变化，并与本地版本合并
git pull	同上，针对当前分支
git fetch [remote]	获取远程仓库的所有变动到本地仓库，不会自动合并！需要手动合并
git push	推送当前分支到远程仓库
git push [remote] [branch]	推送本地当前分支到远程仓库的指定分支
git push [remote] --force/-f	强行推送当前分支到远程仓库，即使有冲突，⚠ 很危险！
git push [remote] --all	推送所有分支到远程仓库
git push -u	参数 <code>-u</code> 表示与远程分支建立关联，第一次执行的时候用，后面就不需要了
git remote rm [remote-name]	删除远程仓库
git pull --rebase	使用 rebase 的模式进行合并

6.3、推送 push / 拉取 pull

`git push`、`git pull` 是团队协作中最常用的指令，用于同步本地、服务端的更新，与他人协作。

- **推送 (push)**：推送本地仓库到远程仓库。
 - 如果推送的更新与服务端存在冲突，则会被拒绝，`push` 失败。一般是有其他人推送了代码，导致文件冲突，可以先 `pull` 代码，在本地进行合并，然后再 `push`。
- **拉取 (pull)**：从服务端（远程）仓库更新到本地仓库。
 - `git pull`：拉取服务端的最新提交到本地，并与本地合并，合并过程同分支的合并。
 - `git fetch`：拉取服务端的最新提交到本地，不会自动合并，也不会更新工作区。

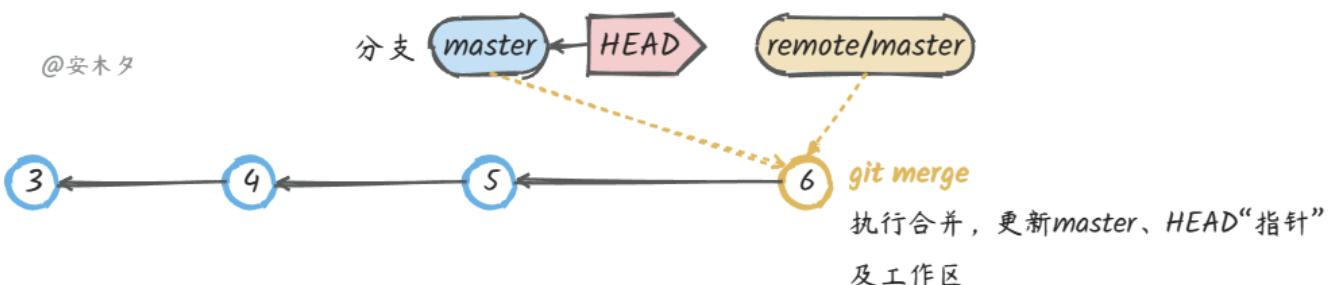
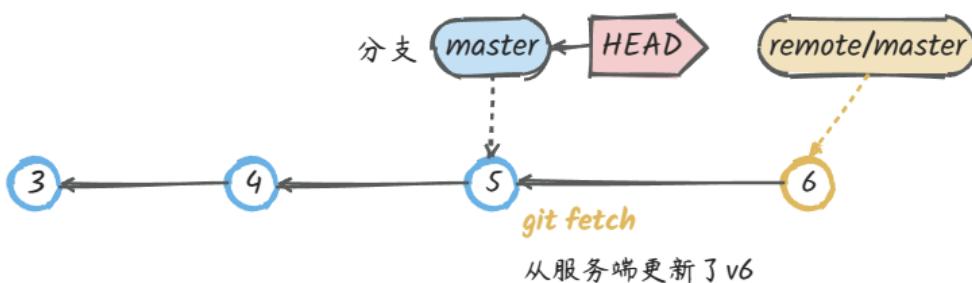


6.4、fetch 与 pull 有什么不同？

两者都是从服务端获取更新，主要区别是 **fetch** 不会自动合并，不会影响当前工作区内容。

git pull = git fetch + git merge

如下面图中，**git fetch** 只获取了更新，并未影响 **master**、**HEAD** 的位置。要更新 **master**、**HEAD** 的位置需要手动执行 **git merge** 合并。



```
# fetch只更新版本库
$ git fetch
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 663 bytes | 44.00 KiB/s, done.
From github.com:kwonganding/KWebNote
  2ba12ca..c64f5b5  main      -> origin/main

# 执行合并，合并自己
$ git merge
Updating 2ba12ca..c64f5b5
Fast-forward
 README.md | 2 ++
 1 file changed, 1 insertion(+), 1 deletion(-)
```

6.5、将本地创建的 git 关联到远程仓库，并推送到远程仓库

如果是用 `git init` 创建本地仓库，首先需要关联到远程仓库。

```
git remote add origin git@git.oschina.net:yourname/demo.git    # 添加远程仓库 origin
```

如果写错地址或者移除远程仓库：

```
git remote rm origin    # 删除错误的远程仓库 origin
```

获取远程库与本地同步合并（如果远程库不为空必须做这一步，否则后面的提交会失败）：

```
git pull --rebase origin master
```

将最新的修改推送到远程仓库：

```
git push -u origin master      # 第一次 push
git push origin master
```

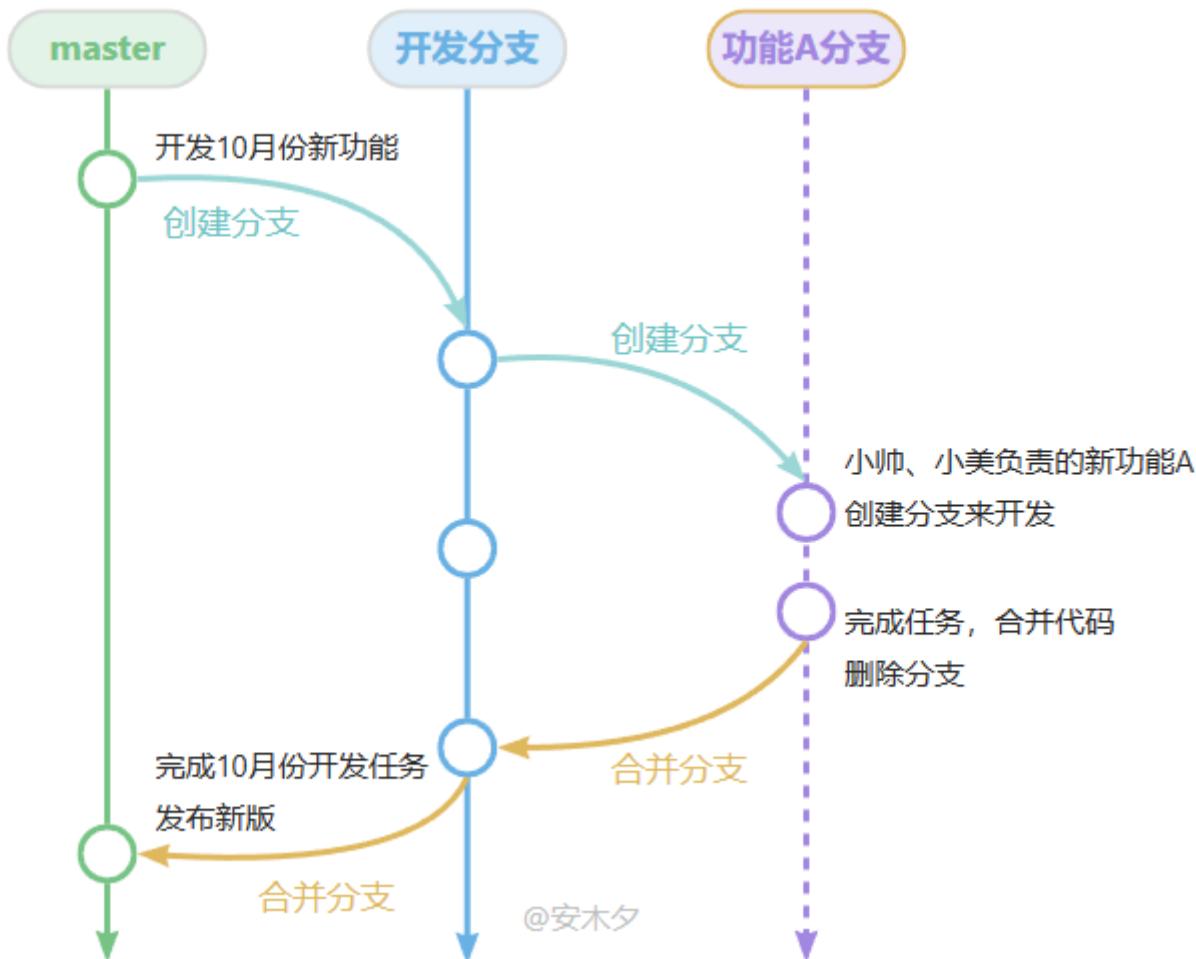
第一次 `push` 的时候，加上 `-u` 参数，Git 就会把本地的 `master` 分支和远程的 `master` 分支进行关联起来，以后的 `push` 操作就不再需要加上 `-u` 参数了

07、Git 利器 - 分支

分支是从主线分离出去的“副本”，分支就像是平行宇宙，可独立发展、独立编辑、提交，也可以和其他分支合并。分支是 Git 的核心必杀利器之一，分支创建、切换、删除都非常快，他非常的轻量。所以，早建分支！多用分支！

7.1、分支 Branch

比如有一个项目团队，准备 10 月份发布新版本，要新开发一堆黑科技功能，占领市场。你和小伙伴“小美”一起负责开发一个新功能 A，开发周期 2 周，在这两周你们的代码不能影响其他人，不影响主分支。这个时候就可以为这个新功能创建一个分支，你们两在这个分支上干活，2 周后代码开发完了、测试通过，就可以合并进要发版的开发分支了。安全、高效，不影响其他人工作，完美！



在实际项目中，一般会建几个主线分支。

- **master**：作为主分支，存放稳定的代码，就是开发后测试通过的代码，不允许随便修改和合并。
- **开发分支**：用于团队日常开发用，比如团队计划 10 月份开发 10 个功能并发版，则在此分支上进行，不影响主分支的稳定。
- **功能 A 分支**：开发人员根据自己的需要，可以创建一些临时分支用于特定功能的开发，开发完毕后再合并到开发分支，并删除该分支。分支就是指向某一个提交记录的“指针”引用，因此创建分支是非常快的，不管仓库多大。当我们运行 `git branch dev` 创建了一个名字为 `dev` 的分支，Git 实际上是在 `.git\refs\heads` 下创建一个 `dev` 的引用文件（没有扩展名）。

```
$ git branch dev
$ cat .git/refs/heads/dev
ca88989e7c286fb4ba56785c2cd8727ea1a07b97
```

7.2、分支指令

指令	描述
git branch	列出所有本地分支，加参数 <code>-v</code> 显示详细列表，下同
git branch -r	列出所有远程分支
git branch -a	列出所有本地分支和远程分支，用不同颜色区分
git branch [branch-name]	新建一个分支，但依然停留在当前分支
git branch -d dev	删除 <code>dev</code> 分支， <code>-D</code> (大写) 强制删除
git checkout -b dev	从当前分支创建并切换到 <code>dev</code> 分支
git checkout -b feature1 dev	从本地 <code>dev</code> 分支代码创建一个 <code>feature1</code> 分支，并切换到新分支
git branch [branch] [commit]	新建一个分支，指向指定 <code>commit id</code>
git branch --track [branch] [remote-branch]	新建一个分支，与指定的远程分支建立关联
git checkout -b hotfix remote hotfix	从远端 <code>remote</code> 的 <code>hotfix</code> 分支创建本地 <code>hotfix</code> 分支
git branch --set-upstream [branch] [remote-branch]	在现有分支与指定的远程分支之间建立跟踪关联：
git branch --set-upstream hotfix remote/hotfix	
git checkout [branch-name]	切换到指定分支，并更新工作区
git checkout .	撤销工作区的（未暂存）修改，把暂存区恢复到工作区。
git checkout HEAD .	撤销工作区、暂存区的修改，用 <code>HEAD</code> 指向的当前分支最新版本替换
git merge [branch]	合并指定分支到当前分支
git merge --no-ff dev	合并 <code>dev</code> 分支到当前分支，参数 <code>--no-ff</code> 禁用快速合并模式
git push origin --delete [branch-name]	删除远程分支
git rebase master	将当前分支变基合并到 <code>master</code> 分支
<input checked="" type="checkbox"/> switch：新的分支切换指令	切换功能和 <code>checkout</code> 一样， <code>switch</code> 只单纯的用于切换
git switch master	切换到已有的 <code>master</code> 分支
git switch -c dev	创建并切换到新的 <code>dev</code> 分支

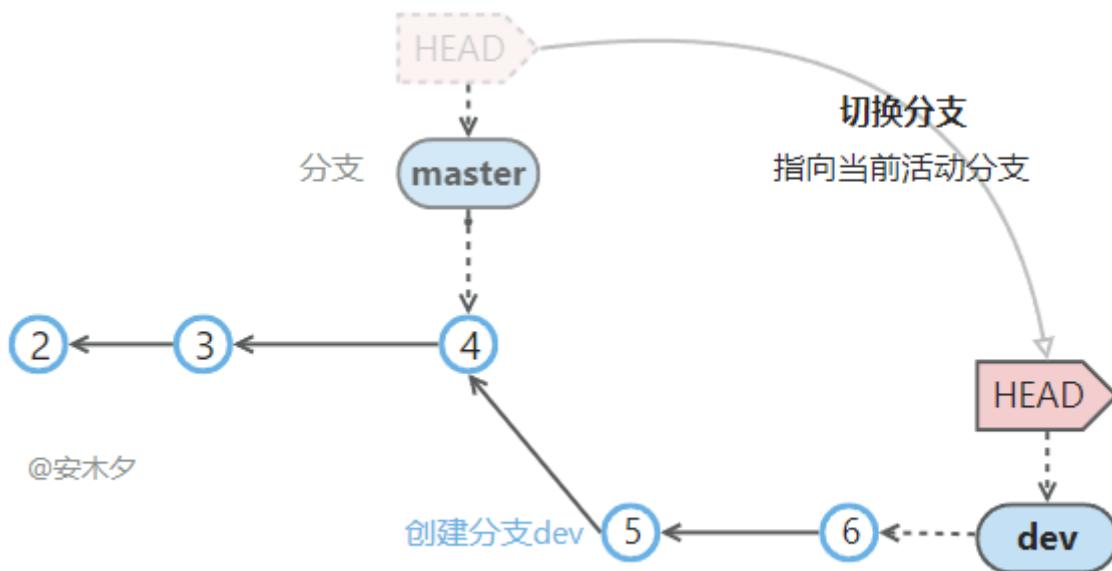
关于 `checkout` 指令：`checkout` 是 Git 的底层指令，比较常用，也比较危险，他会重写工作区。支持的功能比较多，能撤销修改，能切换分支，这也导致了这个指令比较复杂。在 Git 2.23 版本以后，增加了 `git switch`、`git reset` 指令。

- `git switch`：专门用来实现分支切换。
- `git reset`：专门用来实现本地修改的撤销，更多可参考后续“reset”内容。

```
$ git branch
  dev
* main
# 列出了当前的所有分支，星号“*”开头的“main”为当前活动分支。
```

7.3、分支的切换 checkout

代码仓库可以有多个分支，**master** 为默认的主分支，但只有一个分支在工作状态。所以要操作不同分支，需要切换到该分支，**HEAD** 就是指向当前正在活动的分支。



```
# 切换到dev分支，HEAD指向了dev
# 此处 switch 作用同 checkout, switch只用于切换，不像checkout功能很多
$ git switch dev
Switched to branch 'dev'
$ cat .git/HEAD
ref: refs/heads/dev
```

使用 **git checkout dev** 切换分支时，干了两件事：

- **HEAD 指向 dev**：修改 **HEAD** 的“指针”引用，指向 **dev** 分支。
- **还原工作空间**：把 **dev** 分支内容还原到工作空间。

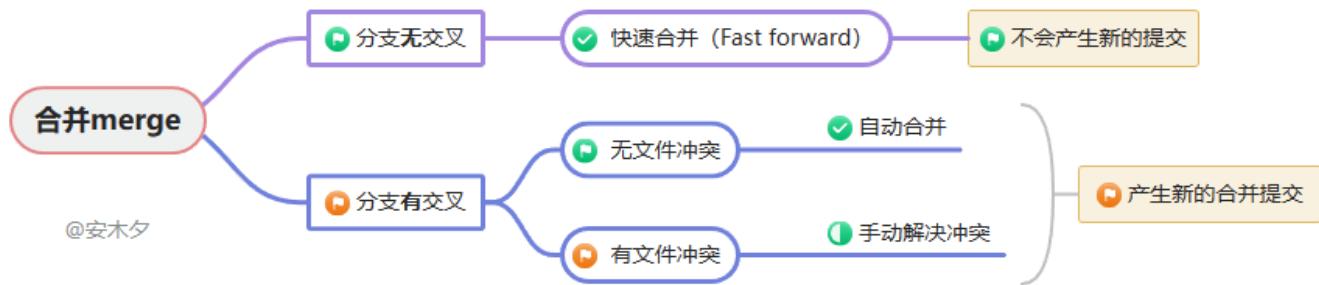
此时的活动分支就是 **dev** 了，后续的提交就会更新到 **dev** 分支了。

切换时还没提交的代码怎么办？

- 如果修改（包括未暂存、已暂存）和待切换的分支没有冲突，则切换成果，且未提交修改会一起带过去，所以要注意！
- 如果有冲突，则会报错，提示先提交或隐藏，关于隐藏可查看后续章节内容“stash”。

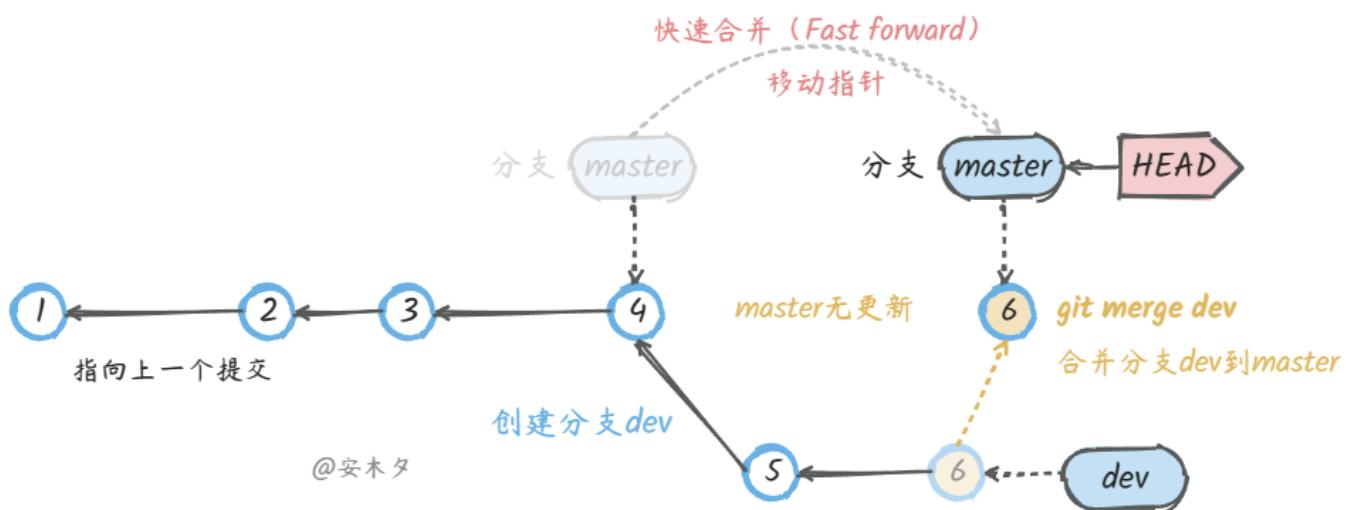
7.4、合并 merge & 冲突

把两个分支的修改内容合并到一起，常用的合并指令 `git merge [branch]`，将分支 `[branch]` 合并到当前分支。根据要合并的内容的不同，具体合并过程就会有多种情况。



快速合并 (Fast forward)

如下图，`master` 分支么有任何提交，“`git merge dev`”合并分支 `dev` 到 `master`，此时合并速度就非常快，直接移动 `master` 的“指针”引用到 `dev` 即可。这就是快速合并 (Fast forward)，不会产生新的提交。



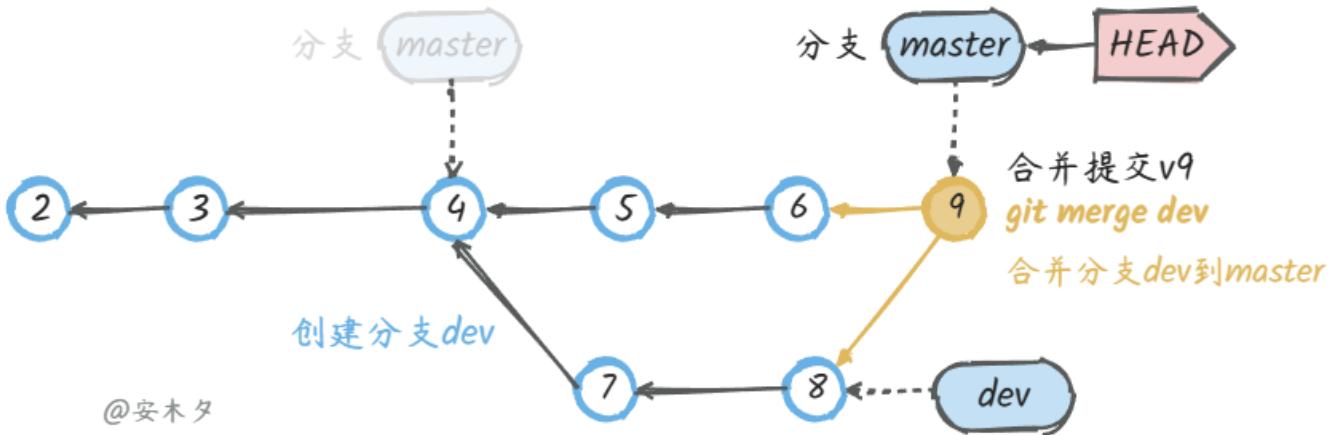
- 合并 `dev` 到 `master`，注意要先切换到 `master` 分支，然后执行 `git merge dev`，把 `dev` 合并到当前分支。

强制不用快速合并：`git merge --no-ff -m "merge with no-ff" dev`，参数 `--no-ff` 不启用快速合并，会产生一个新的合并提交记录。

普通合并

如果 `master` 有变更，存在分支交叉，则会把两边的变更合并成一个提交。

- 如果两边变更的文件不同，没有什么冲突，就自动合并了。
- 如果有修改同一个文件，则会存在冲突，到底该采用哪边的，程序无法判断，就换产生冲突。冲突内容需要人工修改后再重新提交，才能完成最终的合并。



上图中，创建 `dev` 分支后，两个分支都有修改提交，因此两个分支就不在一条顺序线上了，此时合并 `dev` 到 `master` 就得把他们的修改进行合并操作了。

- `v5`、`v7` 共同祖先是 `v4`，从这里开始分叉。
- Git 会用两个分支的末端 `v6` 和 `v8` 以及它们的共同祖先 `v4` 进行三方合并计算。合并之后会生成一个新（合并）提交 `v9`。
- 合并提交 `v9` 就有两个祖先 `v6`、`v8`。

处理冲突 <<<<< HEAD

在有冲突的文件中，`<<<<< HEAD` 开头的内容就表示是有冲突的部分，需要人工处理，可以借助一些第三方的对比工具。人工处理完毕后，完成合并提交，才最终完成此次合并。`=====` 分割线上方是当前分支的内容，下方是被合并分支的变更内容。

```

1  <html>
2   <head>
3     <title>hello</title>
4   </head>
5   <body>
6     <<<<< HEAD
7     Hello.
8     =====
9     <strong>Hello</strong>
10    >>>>> 17c860612953c0f9d88f313c8dfbf7d858e02e91
11   </body>
12 </html>

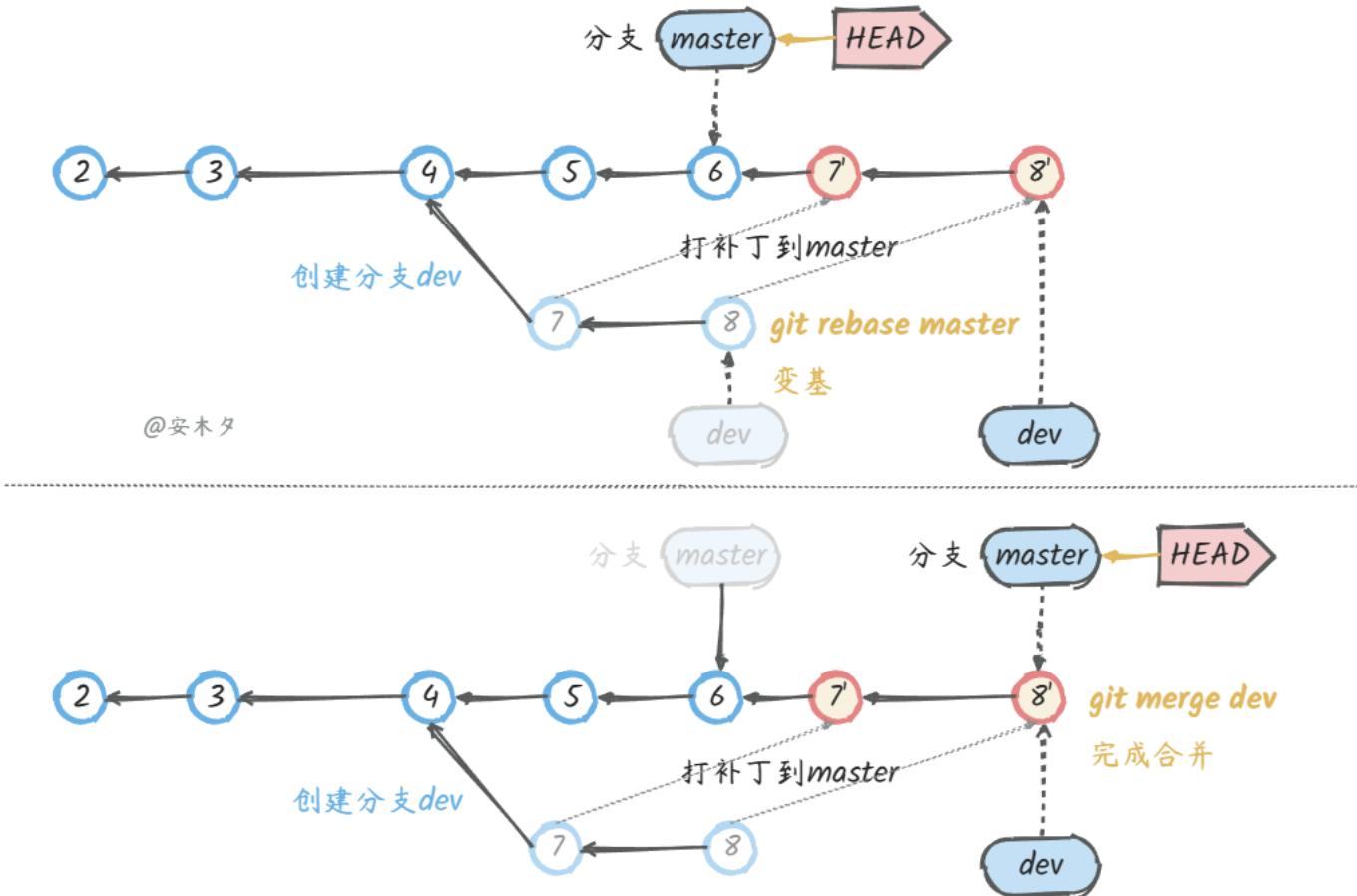
```

发生冲突的部分

<<<<<
 这就是发生冲突的部分！
 >>>>>

7.5、变基 rebase

把两个分支的修改内容合并到一起的办法有两种：`merge` 和 `rebase`，作用都是一样的，区别是 `rebase` 的提交历史更简洁，干掉了分叉，`merge` 的提交历史更完整。



- 在 `dev` 上执行 “`git rebase master`” 变基，将 `dev` 分支上分叉的 `v7`、`v8` 生成补丁，然后在 `master` 分支上应用补丁，产生新的 `v7'`、`v8'` 新的提交。
- 然后回到 `master` 分支，完成合并 `git merge dev`，此时的合并就是快速合并了。
- 最终的提交记录就没有分叉了。

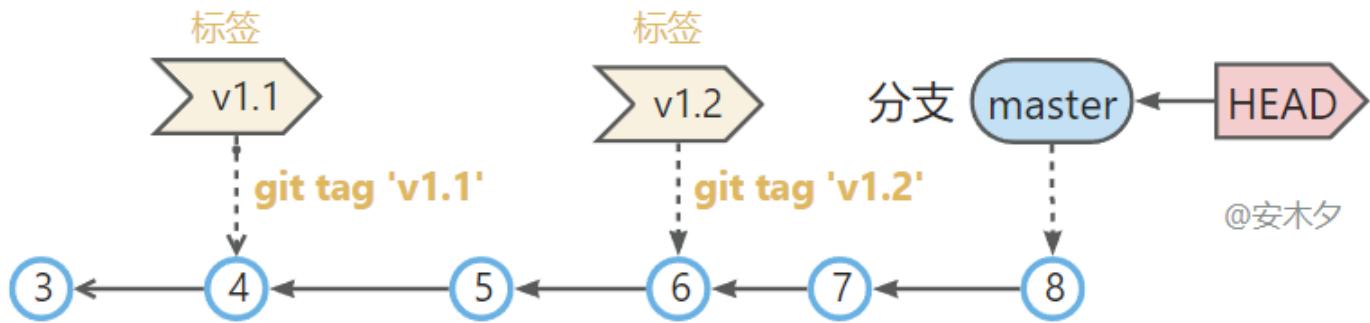
```
$ git rebase master
$ git checkout master
$ git merge dev
```

08、标签管理

标签 (Tags) 指的是某个分支某个特定时间点的状态，是对某一个提交记录的固定“指针”引用。一经创建，不可移动，存储在工作区根目录下 `.git\refs\tags`。可以理解为某一次提交（编号）的别名，常用来标记版本。所以发布时，一般都会打一个版本标签，作为该版本的快照，指向对应提交 `commit`。

当项目达到一个关键节点，希望永远记住那个特别的提交快照，你可以使用 `git tag` 给它打上标签。比如我们今天终于完成了 V1.1 版本的开发、测试，并成功上线了，那就可给今天最后这个提交打一个标签“V1.1”，便于版本管理。

默认标签是打在最新提交的 `commit` 上的，如果希望在指定的提交上打标签则带上提交编号（`commit id`）：`git tag v0.9 f52c633`



指令	描述
git tag	查看标签列表
git tag -l 'a*'	查看名称是“a”开头的标签列表，带查询参数
git show [tagname]	查看标签信息
git tag [tagname]	创建一个标签，默认标签是打在最新提交的 commit 上的
git tag [tagname] [commit id]	新建一个 tag 在指定 commit 上
git tag -a v5.1 -m'v5.1 版本'	创建标签 v5.1.1039, -a 指定标签名, -m 指定说明文字
git tag -d [tagname]	删除本地标签
git checkout v5.1.1039	切换标签，同切换分支
git push [remote] v5.1	推送标签，标签不会默认随代码推送推送到服务端
git push [remote] --tags	提交所有 tag

如果要推送某个标签到远程，使用命令 `git push origin [tagname]`，或者，一次性推送全部到远程：`git push origin --tags`

注意：标签总是和某个 `commit` 挂钩。如果这个 `commit` 既出现在 `master` 分支，又出现在 `dev` 分支，那么在这两个分支上都可以看到这个标签。

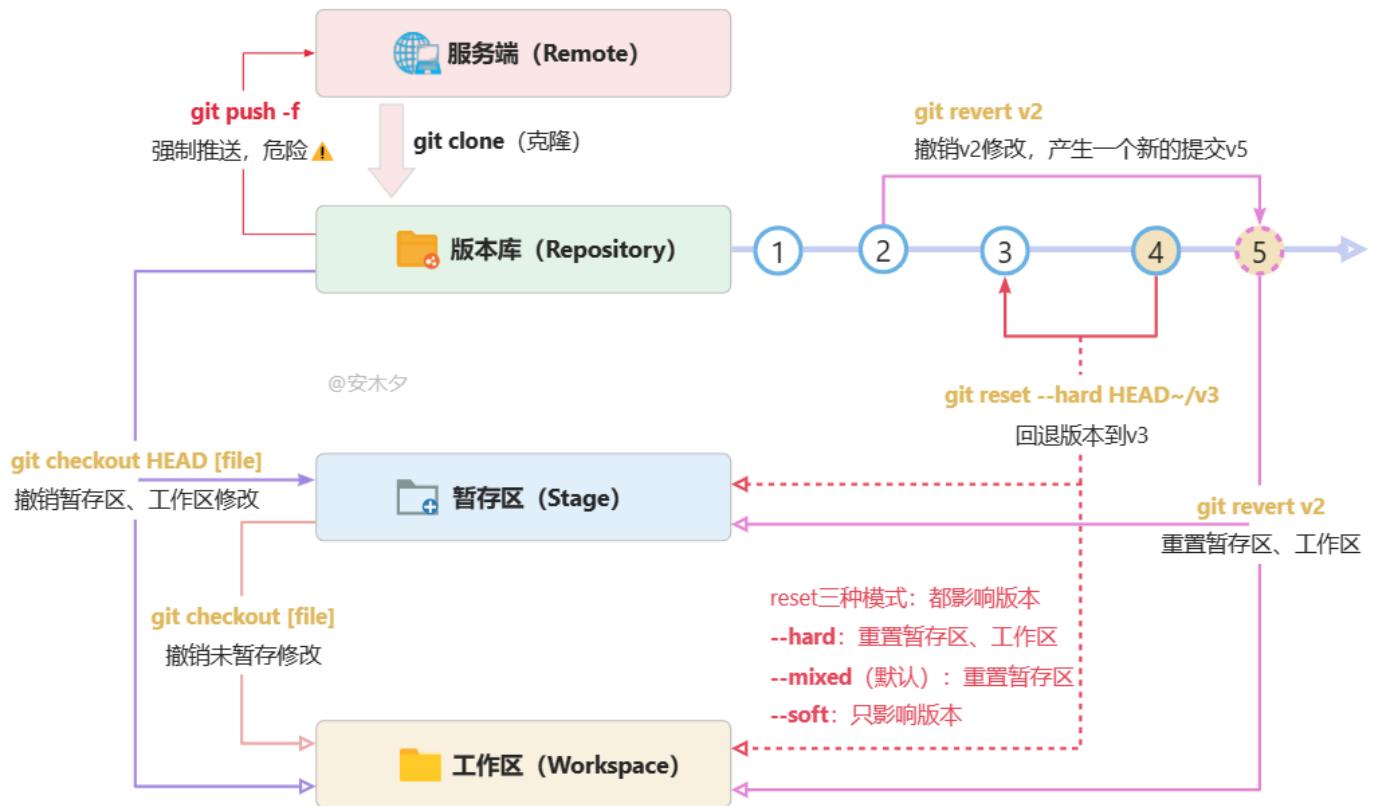
```
# tag
$ git tag -a 'v1' -m'v1版本'
$ cat .git/refs/tags/v1
a2e2c9cae35e176cf61e96ad9d5a929cfb82461

# 查看标签列表
$ git tag
v1
```

09、后悔药 - 怎么撤销变更？

发现写错了要回退怎么办？看看下面几种后悔指令吧！

- 还没提交的怎么撤销？—— `checkout`、`reset`
 - 还未提交的修改（工作区、暂存区）不想要了，用签出指令（`checkout`）进行撤销清除。
 - 或者用 `checkout` 的新版回滚指令 `reset`。
- 已提交但么有 `push` 的提交如何撤销？—— `reset`、`revert`
- 已 `push` 的提交如何撤销？—— 同上，先本地撤销，然后强制推送 `git push origin -f`，⚠ 注意慎用！记得先 `pull` 获取更新。



9.1、后悔指令 ⚔

指令	描述
git checkout .	撤销工作区的（未暂存）修改，把暂存区恢复到工作区。不影响暂存区，如果没暂存，则撤销所有工作区修改
git checkout [file]	同上， <code>file</code> 指定文件
git checkout HEAD .	撤销工作区、暂存区的修改，用 <code>HEAD</code> 指向的当前分支最新版本替换工作区、暂存区
git checkout HEAD [file]	同上， <code>file</code> 指定文件
git reset	撤销暂存区状态，同 <code>git reset HEAD</code> ，不影响工作区
git reset HEAD [file]	同上，指定文件 <code>file</code> , <code>HEAD</code> 可省略
git reset [commit]	回退到指定版本，清空暂存区，不影响工作区。工作区需要手动 <code>git checkout</code> 签出
git reset --soft [commit]	移动分支 <code>master</code> 、 <code>HEAD</code> 到指定的版本，不影响暂存区、工作区，需手动 <code>git checkout</code> 签出更新
git reset --hard HEAD	撤销工作区、暂存区的修改，用当前最新版
git reset --hard HEAD~	回退到上一个版本，并重置工作区、暂存区内容。
git reset --hard [commit]	回退到指定版本，并重置工作区、暂存区内容。
git revert[commit]	撤销一个提交，会用一个新的提交（原提交的逆向操作）来完成撤销操作，如果已 <code>push</code> 则重新 <code>push</code> 即可

- `git checkout .`、`git checkout [file]` 会清除工作区中未添加到暂存区的修改，用暂存区内容替换工作区。
- `git checkout HEAD .`、`git checkout HEAD [file]` 会清除工作区、暂存区的修改，用 `HEAD` 指向的当前分支最新版本替换暂存区、工作区。

```
# 只撤销工作区的修改（未暂存）
$ git checkout .
Updated 1 path from the index
```

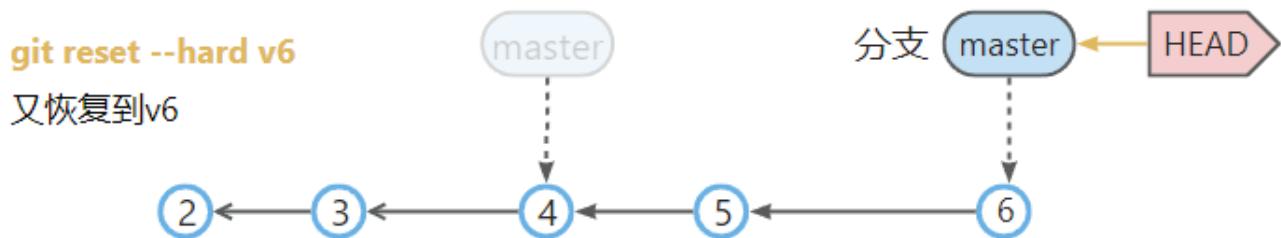
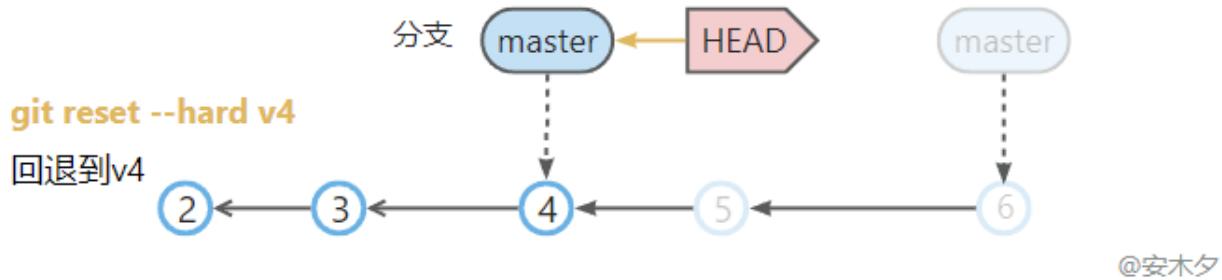
```
# 撤销工作区、暂存区的修改
$ git checkout HEAD .
Updated 1 path from f951a96
```

9.2、回退版本 `reset`

`reset` 是专门用来撤销修改、回退版本的指令，支持的场景比较多，多种撤销姿势，所以参数组合也比较多。简单理解就是移动 `master` 分支、`HEAD` 的“指针”地址，理解这一点就基本掌握 `reset` 了。

如下图：

- 回退版本 `git reset --hard v4` 或 `git reset --hard HEAD~2`, master、HEAD 会指向 v4 提交，v5、v6 就被废弃了。
- 也可以重新恢复到 v6 版本：`git reset --hard v6`，就是移动 master、HEAD 的“指针”地址。



`reset` 有三种模式，对应三种参数：`mixed`（默认模式）、`soft`、`hard`。三种参数的主要区别就是对工作区、暂存区的操作不同。

- `mixed` 为默认模式，参数可以省略。
- 只有 `hard` 模式会重置工作区、暂存区，一般用这个模式会多一点。

模式名称 \ 描述	HEAD 的位置	暂存区	工作区
soft 回退到某一个版本，工作区不变，需手动 <code>git checkout</code>	修改	不修改	不修改
mixed(默认) 撤销暂存区状态，不影响工作区，需手动 <code>git checkout</code>	修改	修改	不修改
hard 重置未提交修改（工作区、暂存区）	修改	修改	修改

穿梭前，用 `git log` 可以查看提交历史，以便确定要回退到哪个版本。要重返未来，用 `git reflog` 查看命令历史，以便确定要回到未来的哪个版本。

```
git reset [--soft | --mixed | --hard] [HEAD]
```

```
# 撤销暂存区
```

```
$ git reset
```

```
Unstaged changes after reset:
```

```
M       R.md
```

```
# 撤销工作区、暂存区修改
```

```
$ git reset --hard HEAD
```

```
# 回退版本库到上一个版本，并重置工作区、暂存
```

```
$ git reset --hard HEAD~
```

```
# 回到原来的版本（恢复上一步的撤销操作），并重置工作区、暂存
```

```
$ git reset --hard 5f8b961
```

```
# 查看所有历史提交记录
```

```
$ git reflog
```

```
ccb9937 (HEAD -> main, origin/main, origin/HEAD) HEAD@{0}: commit: 报表新增导入功能
```

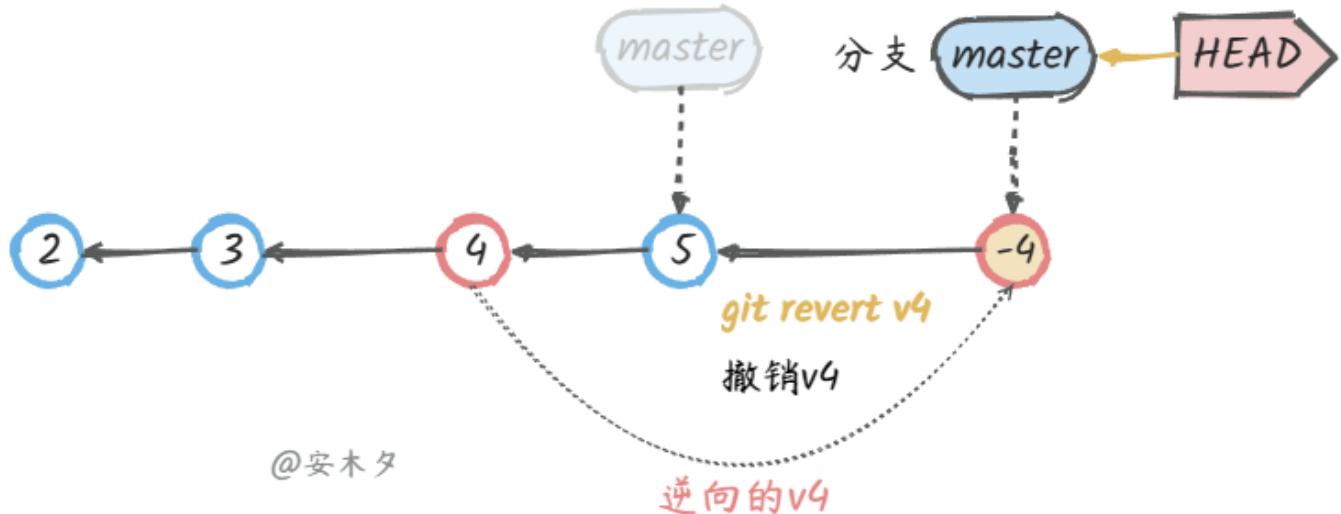
```
8f61a60 HEAD@{1}: commit: bug :修复报表导出bug
```

```
4869ff7 HEAD@{2}: commit: 用户报表模块开发
```

```
4b1028c HEAD@{3}: commit: 财务报表模块开发完成
```

9.3、撤销提交 revert

安全的撤销某一个提交记录，基本原理就是生产一个新的提交，用原提交的逆向操作来完成撤销操作。注意，这不同于 `reset`，`reset` 是回退版本，`revert` 只是用于撤销某一次历史提交，操作是比较安全的。



如上图：

- 想撤销 v4 的修改，执行 `git revert v4`，会产生一个新的提交 v-4，是 v4 的逆向操作。
- 同时更新 maser、HEAD “指针”位置，以及工作区内容。
- 如果已 push 则重新 push 即可。

```
# revert撤销指定的提交，"-m"附加说明
$ git revert 41ea42 -m'撤销对***的修改'
[main 967560f] Revert "123"
    1 file changed, 1 deletion(-)
```

9.4、checkout/reset/revert 总结

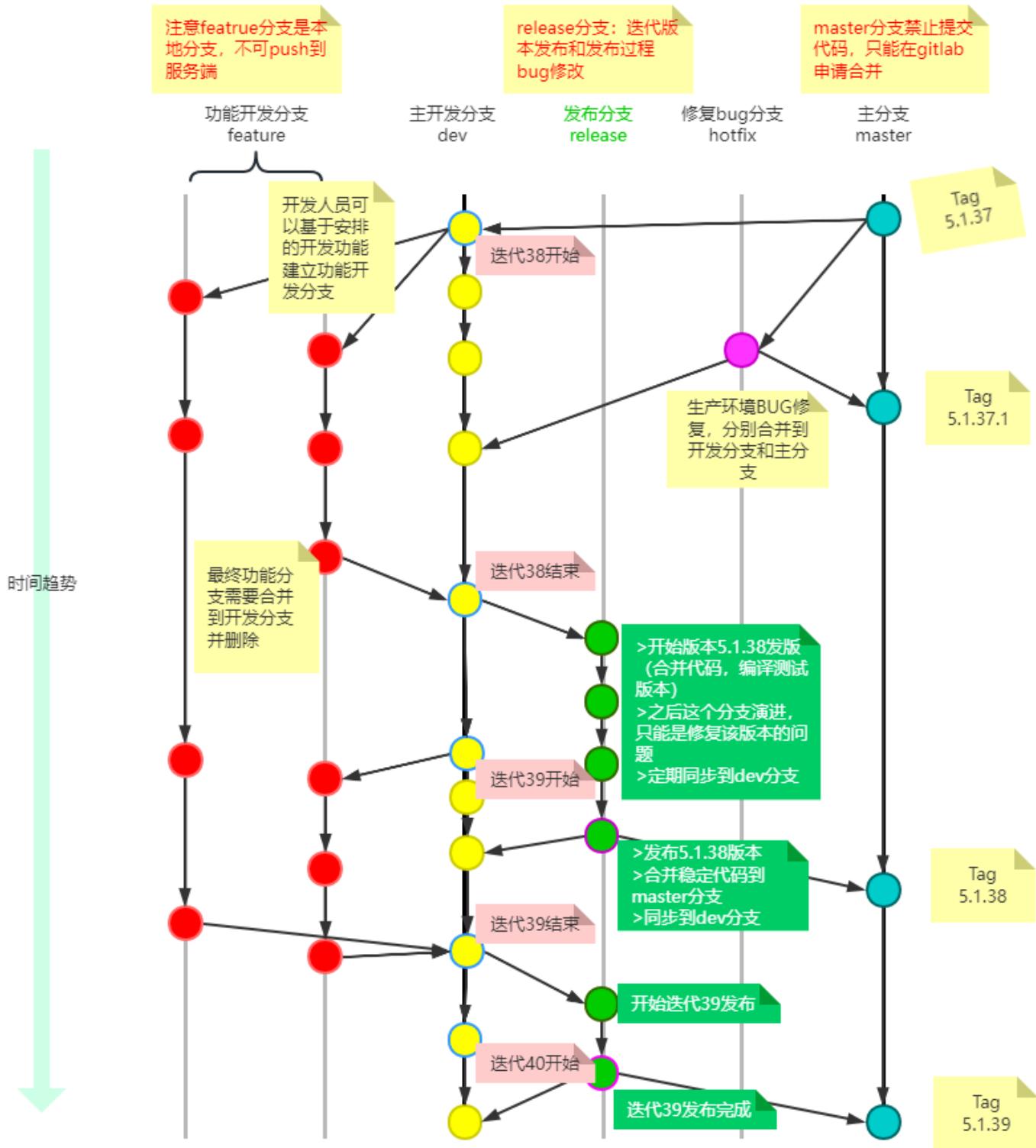
标题 \ 指令	checkout	reset	revert
主要作用（撤销）	撤销工作区、暂存区未提交修改	回退版本，重置工作区、暂存区	撤销某一次提交
撤销工作区	git checkout [file]	git reset HEAD [file]	
撤销工作区、暂存区	git checkout HEAD [file]	git reset --hard HEAD [file]	
回退版本		git reset --hard [commit]	
安全性	只针对未提交修改，安全	如回退了已 push 提交，不安全	安全

可看出 `reset` 完全可以替代 `checkout` 来执行撤销、回退操作，`reset` 本来也是专门用来干这个事情的，可以抛弃 `checkout` 了（撤销的时候）。

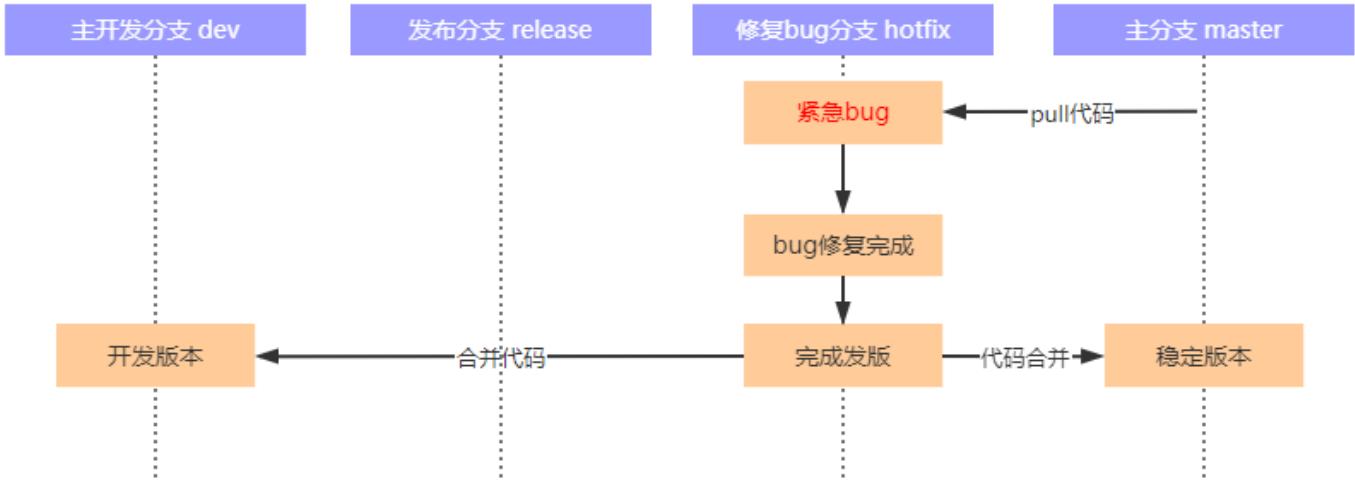
10、工作中的 Git 实践

10.1、Git flow

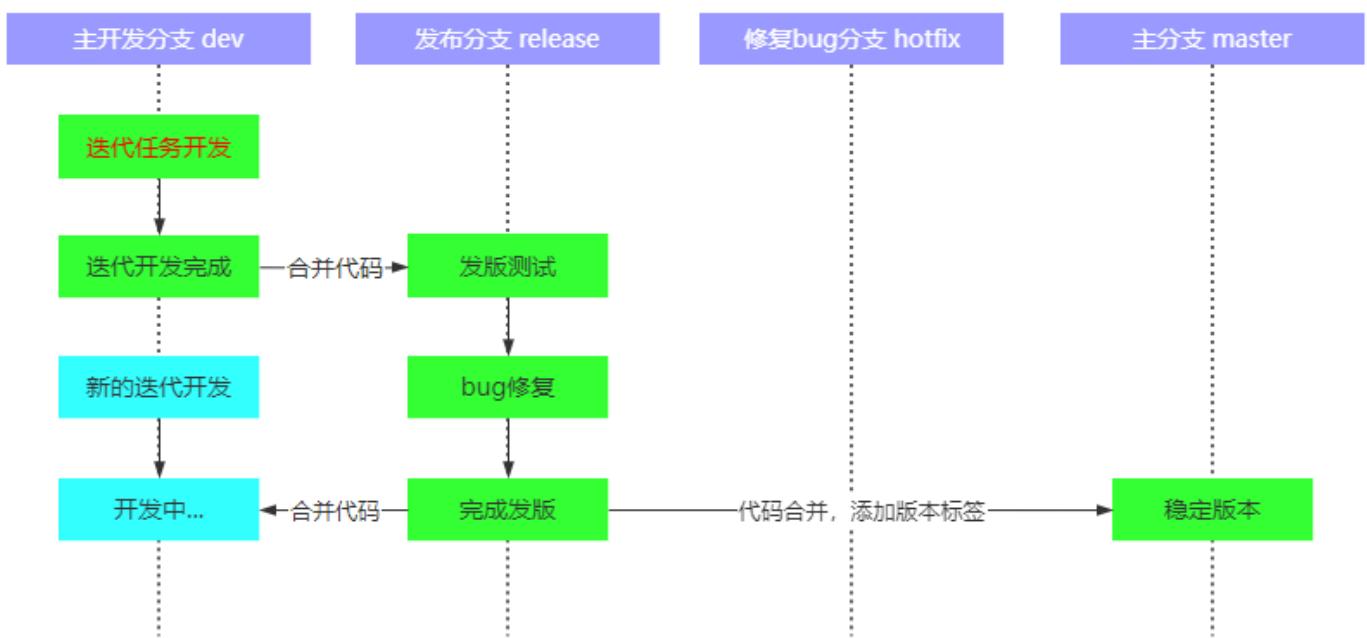
Git flow (Git 工作流程) 是指软件项目中的一种 Git 分支管理模型，经过了大量的实践和优化，被认为是现代敏捷软件开发和 DevOps (开发、技术运营和质量保障三者的交集) 的最佳实践。Git flow 主要流程及关键分支：



- **主分支 : master** · 稳定版本代码分支，对外可以随时编译发布的分支，不允许直接 Push 代码，只能请求合并 (pull request)，且只接受 hotfix、release 分支的代码合并。
- **热修复分支 : hotfix** · 针对线上紧急问题、bug 修复的代码分支，修复完后合并到主分支、开发分支。
 - 切换到 hotfix 分支，从 master 更新代码；
 - 修复 bug；
 - 合并代码到 dev 分支，在本地 Git 中操作即可；
 - 合并代码到 master 分支。



- **发版分支 : release** · 版本发布分支，用于迭代版本发布。迭代开发完成后，合并 `dev` 代码到 `release`，在 `release` 分支上编译发布版本，以及修改 bug (定时同步 bug 修改到 `dev` 分支) 。测试完成后此版本可以作为发版使用，然后把稳定的代码 `push` 到 `master` 分支，并打上版本标签。
- **开发分支 : dev** · 开发版本分支，针对迭代任务开发的分支，日常开发原则上都在此分支上面，迭代完成后合并到 `release` 分支，开发、发版两不误。



- **其他开发分支 : dev-xxx** · 开发人员可以针对模块自己创建本地分支，开发完成后合并到 `dev` 开发分支，然后删除本地分支。

10.2、金屋藏娇 stash

当你正在 `dev` 分支开发一个功能时，代码写了一半，突然有一个线上的 bug 急需要马上修改。`dev` 分支 Bug 没写完，不方便提交，就不能切换到主分支去修复线上 bug。Git 提供一个 `stash` 功能，可以把当前工作区、暂存区未提交的内容“隐藏”起来，就像什么都没发生一样。

```

# 有未提交修改，切换分支时报错
$ git checkout dev
error: Your local changes to the following files would be overwritten by checkout:
 README.md
Please commit your changes or stash them before you switch branches.
Aborting

# 隐藏
$ git stash
Saved working directory and index state WIP on main: 2bc012c s

# 查看被隐藏的内容
$ git stash list
stash@{0}: WIP on main: 2bc012c s

# 比较一下，什么都没有，一切都没有发生过！
$ git diff

# 去其他分支修改bug，修复完成回到当前分支，恢复工作区
$ git stash pop

```

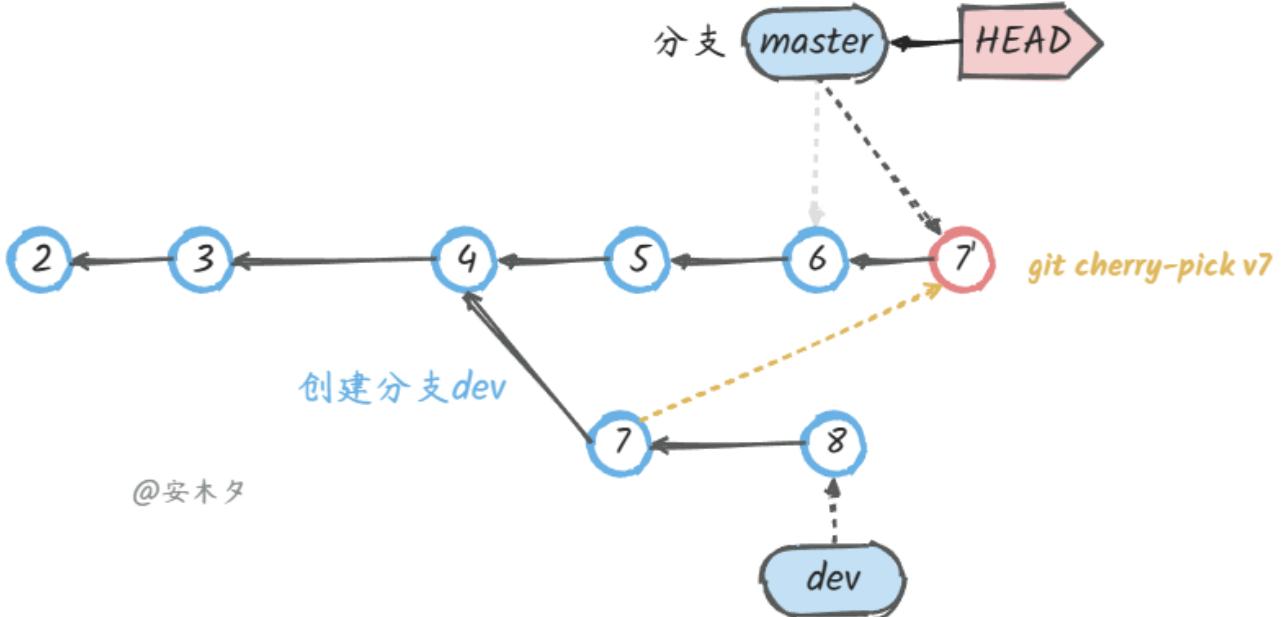
在上面示例中，有未提交修改，切换分支时报错。错误提示信息很明确了，`commit` 提交或 `stash` 隐藏：`Please commit your changes or stash them before you switch branches.`

如果切换分支时，未提交修改的内容没有冲突，是可以成功切换的，未提交修改会被带过去。

指令	描述
<code>git stash</code>	把未提交内容隐藏起来，包括未暂存、已暂存。等以后恢复现场后继续工作
<code>git stash list</code>	查看所有被隐藏的内容列表
<code>git stash pop</code>	恢复被隐藏的内容，同时删除隐藏记录
<code>git stash save "message"</code>	同 <code>git stash</code> ，可以备注说明 <code>message</code>
<code>git stash apply</code>	恢复被隐藏的文件，但是隐藏记录不删除
<code>git stash drop</code>	删除隐藏记录

拣选提交 `cherry-pick`

当有一个紧急 bug，在 `dev` 上修复完，我们需要把 `dev` 上的这个 bug 修复所做的修改“复制”到 `master` 分支，但不想把整个 `dev` 合并过去。为了方便操作，Git 专门提供了一个 `cherry-pick` 命令，让我们能复制一个特定的提交到当前分支，而不管这个提交在哪个分支。



如上图，操作过程相当于将该提交导出为补丁文件，然后在当前 HEAD 上重放，形成无论内容还是提交说明都一致的提交。

- 希望把 dev 分支上的 v7 提交的内容合并到 master，但不需要其他的内容。
- 在 master 分支上执行指令 git cherry-pick v7，会产生一个新的 v7' 提交，内容和 v7 相同。
- 同时更新 master、HEAD，以及工作区。

```
# 选择一个commit，合并进当前分支
$ git cherry-pick [commit]
```

转载

<https://www.cnblogs.com/anding/p/16987769.html#scroller-33>

参考

https://blog.csdn.net/byd_chao/article/details/82821897