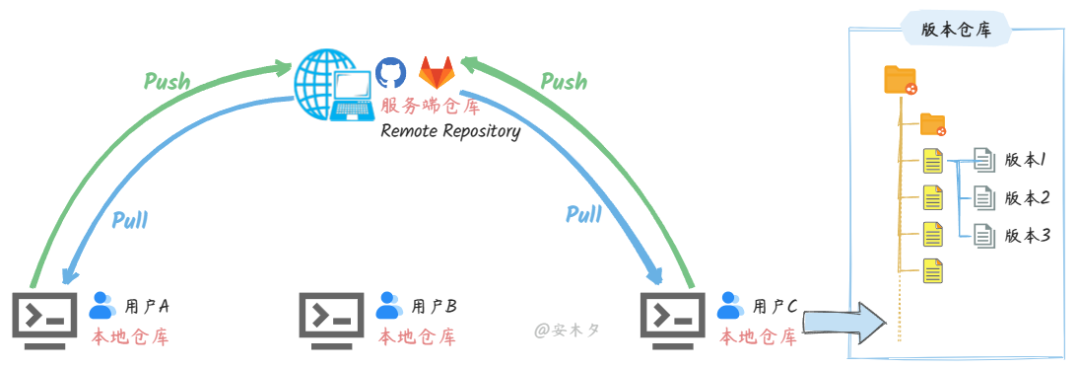
## Git简介

### 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Linux之父被迫开发的，为了解决Linux混乱的代码管理而开发的。Linux和Git之父 李纳斯·托沃兹（Linus Benedic Torvalds），来自1969年的芬兰。

### Git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

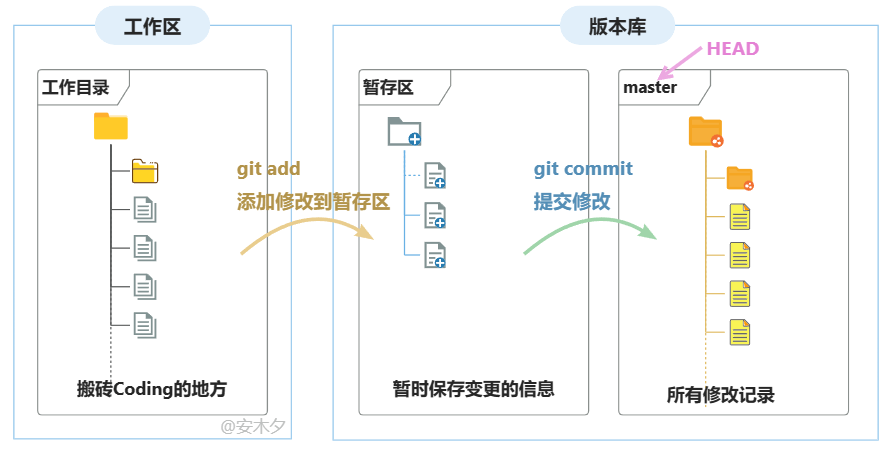
概念名称描述工作区（Workspace）就是在电脑里能看到的代码库目录，是我们搬砖的地方，新增、修改的文件会提交到暂存区暂存区（stage 或 index）用于临时存放文件的修改，实际上上它只是一个文件（.git/index），保存待提交的文件列表信息。版本库/仓库（Repository）Git的管理仓库，管理版本的数据库，记录文件/目录状态的地方，所有内容的修改记录（版本）都在这里。服务端/远程仓库（origin 或 remote）服务端的版本库，专用的Git服务器，为多人共享提供服务，承担中心服务器的角色。本地版本库通过push指令把代码推送到服务端版本库。本地仓库用户机器上直接使用的的的版本库分支（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、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。

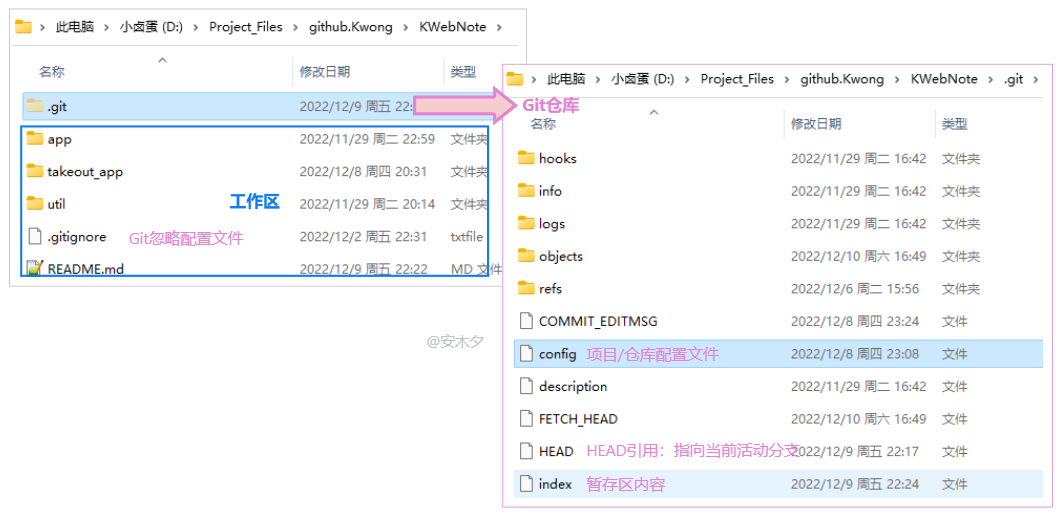
暂存区（stage或index） 用于临时存放文件的修改，实际上上它只是一个文件（.git/index），保存待提交的文件列表信息。

用git add 命令将工作区的修改保存到暂存区。

版本库/仓库（Repository /rɪˈpɑːzətɔːri/ 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。

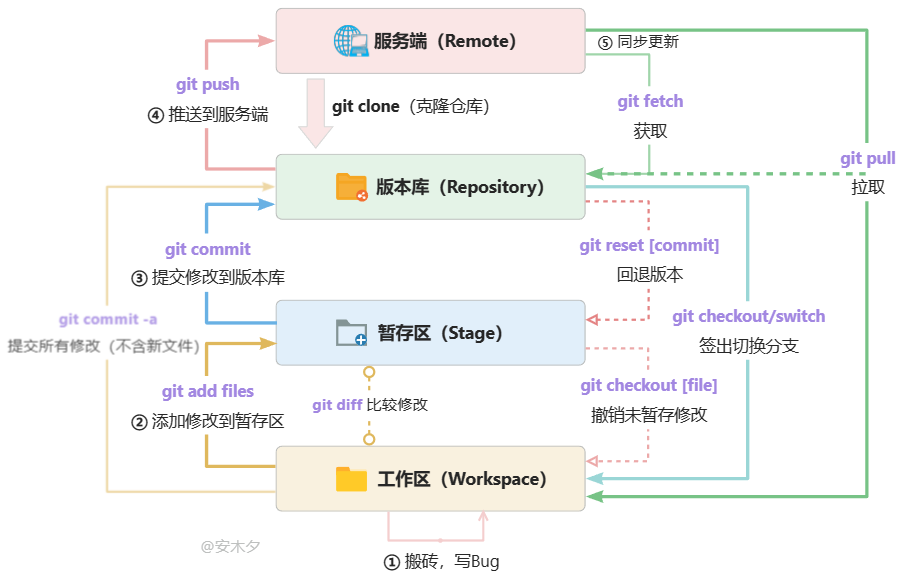
.git/config为该仓库的配置文件，可通过指令修改或直接修改。

index文件就是存放的暂存区内容。

#### 2.3、Git基本流程

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。

1. 准备仓库：创建或从服务端克隆一个仓库。
2. 搬砖：在工作目录中添加、修改代码。
3. 暂存（git add）：将需要进行版本管理的文件放入暂存区域。
4. 提交（git commit）：将暂存区域的文件提交到Git仓库。
5. 推送（git push）：将本地仓库推送到远程仓库，同步版本库。
6. 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

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：

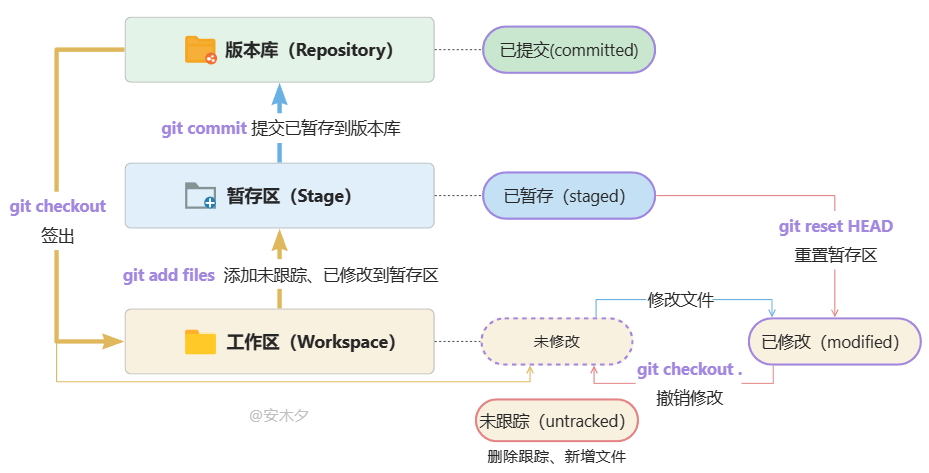
未跟踪（untracked）：新添加的文件，或被移除跟踪的文件，未建立跟踪，通过git add添加暂存并建立跟踪。

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。



### 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。

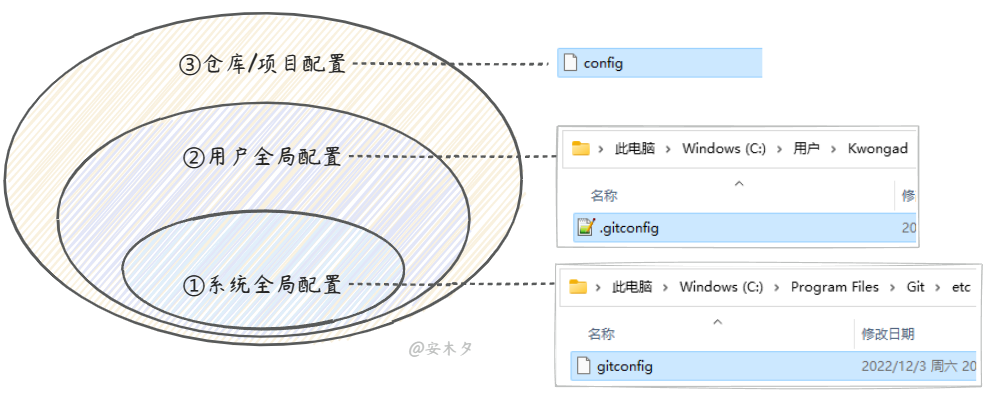
指令git --version查看安装版本号

$ git --versiongit 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① < ② < ③

1. 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。
2. 用户全局配置(--system)：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。
3. 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

### Git的GUI工具们

如果不想用命令行工具，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。

对于Git，建议用命令行，这样你才能真的理解Git的思想？✔️其实都只是工具而已，适合自己的就是最好的，没必要纠结，多写点Bug更重要！

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。

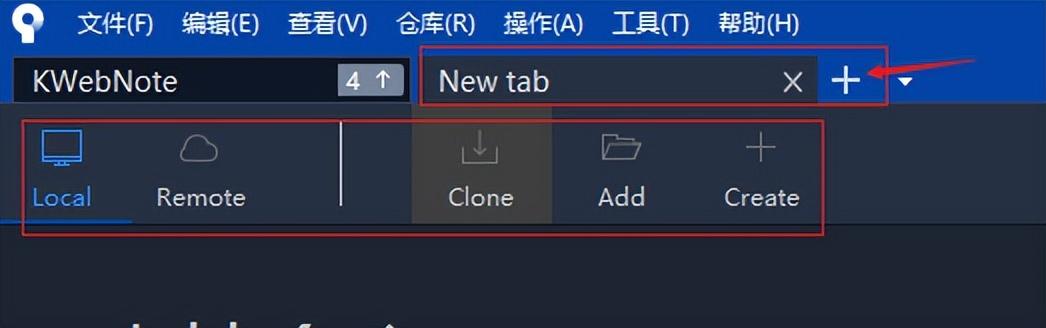
GitHub Desktop：Github官方出品的Git管理工具。

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。

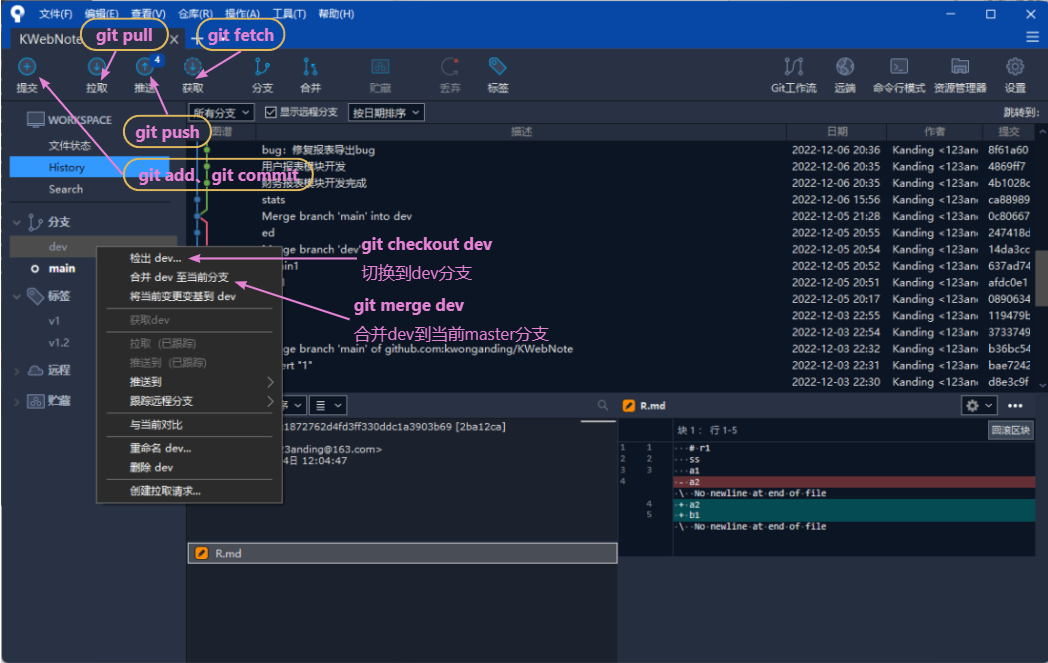
#### 4.1、SourceTree

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。

#### 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。

### 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。

克隆远程仓库，如在github上创建“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 0Receiving 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。

#### 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git add [file1] [file2]添加文件到暂存区，包括修改的文件、新增的文件git add [dir]同上，添加目录到暂存区，包括子目录git add .同上，添加所有修改、新增文件（未跟踪）到暂存区git rm [file]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：

执行git add .暂存：

#### 5.3、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。

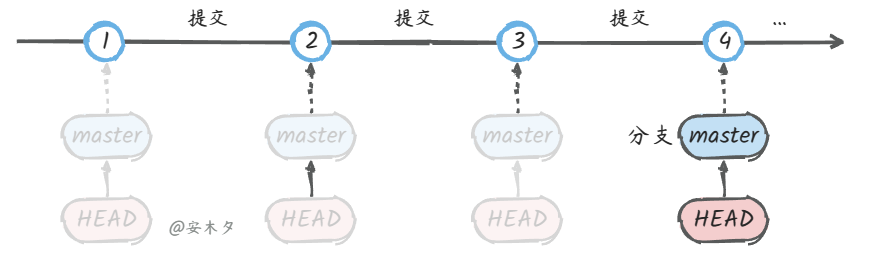
描述：针对本次提交的描述说明，建议准确填写，就跟代码中的注释一样，很重要。

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。

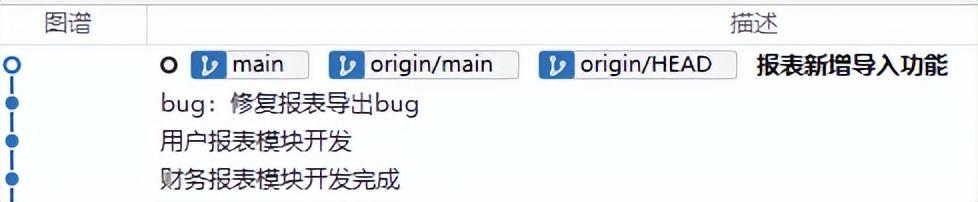
SHA1 是一种哈希算法，可以用来生成数据摘要

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：

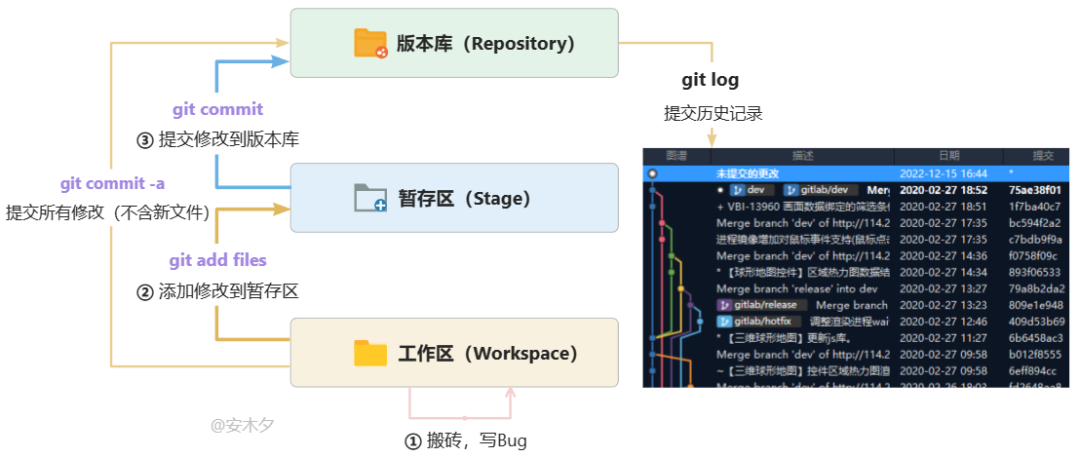


一般情况，每完成一个小功能、一个Bu就可以提交一次，这样会形成比较清晰的历史记录。

指令：

指令描述git commit -m '说明'提交变更，参数-m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-a，表示直接从工作区提交到版本库，略过了git add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commit，替代上一次提交，会修改commit的hash值（id）git log -n20查看日志(最近20条)，不带参数-n则显示所

日志git log -n20 --oneline参数“--oneline”可以让日志输出更简洁（一行）git log -n20 --graph参数“--graph”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-s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 -n2commit 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 --oneline5444126 (HEAD -> main, origin/main, origin/HEAD) Update README.md228362e Merge branch 'main' of github.com:kwonganding/KWebNote

#### 5.4、Git的“指针”引用们

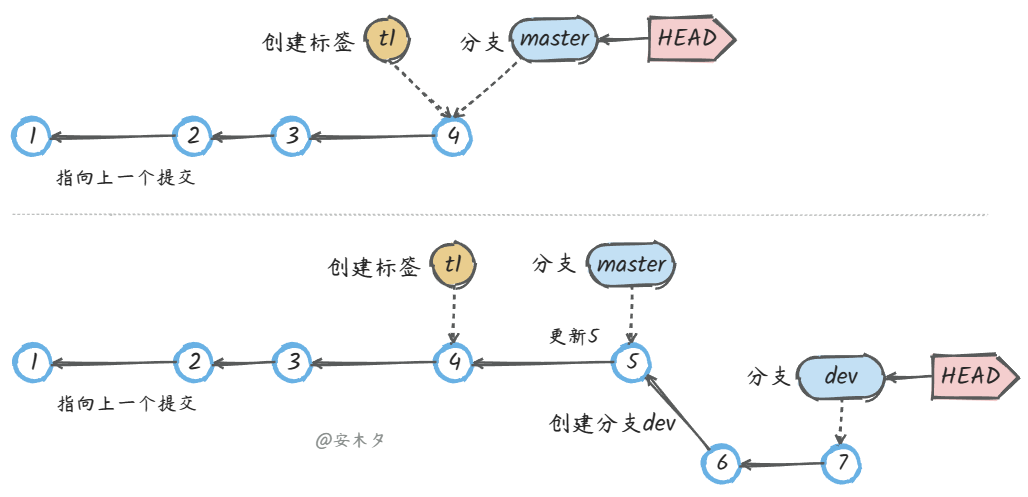
Git中最重要的就是提交记录了，其他如标签、分支、HEAD 都对提交记录的“指针”引用，指向这些提交记录，理解这一点很重要。

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

a2e2c9caea35e176cf61e96ad9d5a929cfb82461

# main分支指向最新的提交

$ cat .git/refs/heads/main8f4244550c2b6c23a543b741c362b13768442090

# HEAD指向当前活动分支

$ cat .git/HEADref: refs/heads/main

# 切换到dev分支，HEAD指向了dev

$ git switch dev

Switched to branch 'dev'

$ cat .git/HEADref: 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 -n105acc914 (HEAD -> main) HEAD@{0}: reset: moving to HEAD~738748b (dev) HEAD@{1}: reset: moving to HEAD~9312c3e HEAD@{2}: reset: moving to HEAD~

db03fcb 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 d3e15f91b81fb3 HEAD@{7}: reset: moving to HEAD~141ea423 HEAD@{8}: reset: moving to HEAD~

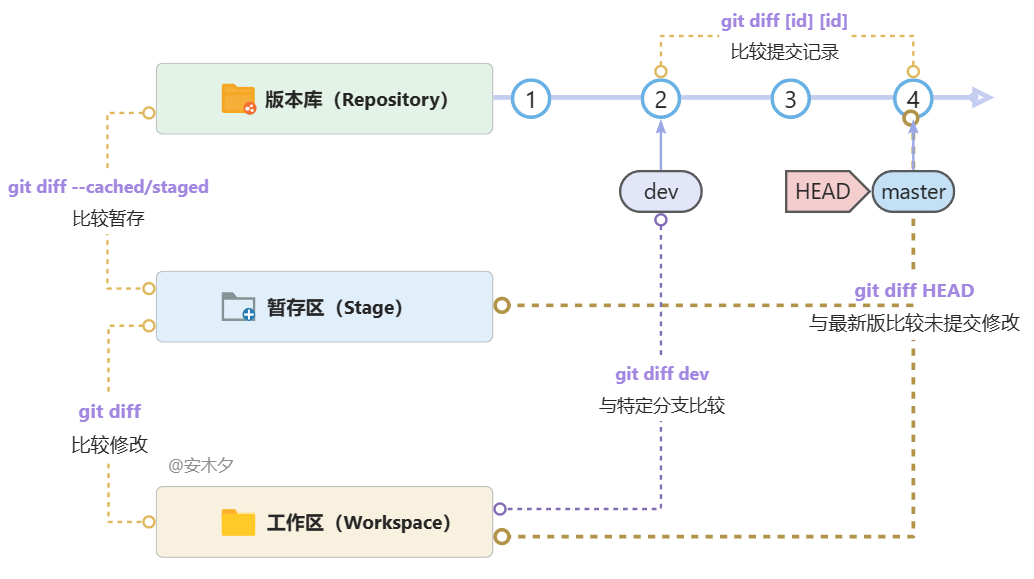
d3e15f9 HEAD@{9}: reset: moving to HEAD~

#### 5.6、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

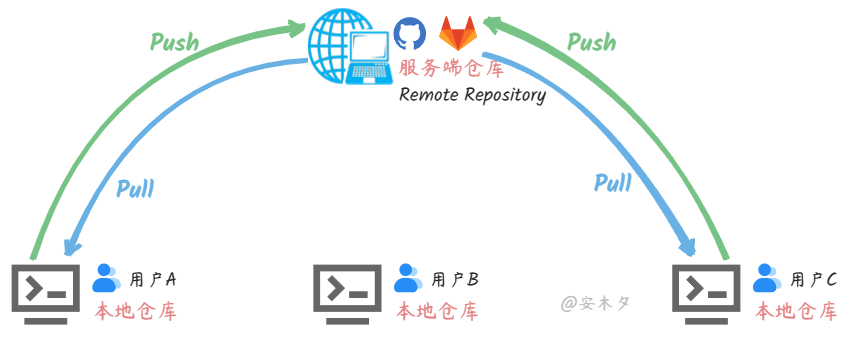
$ git diff 8f4244 1da22

# 显示今天你写了多少行代码：工作区+暂存区

$ git diff --shortstat "@{0 day ago}"

### 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：

HTTS：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：HTTS

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。

$ 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。

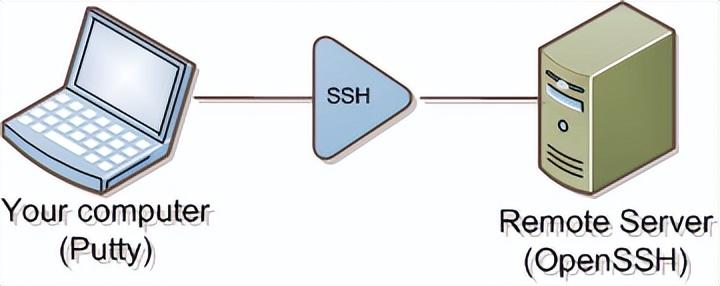
# 参数“--global”全局有效，也可以针对仓库设置“--local”# store 表示永久存储，也可以设置临时存储git config --global credential.helper store

# 存储内容如下，打开文件“仓库\.git\.git-credentials”

https://kwonganding:[加密内容付费可见]@github.com

远程用户登录：SSH

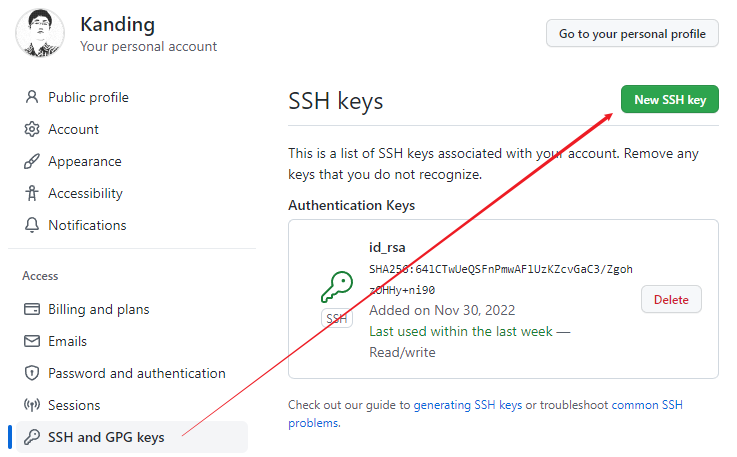
SSH（Secure Shell，安全外壳）是一种网络安全协议，通过加密和认证机制实现安全的访问和文件传输等业务，多用来进行远程登录、数据传输。SSH通过公钥、私钥非对称加密数据，所以SSH需要生成一个公私钥对，公钥放服务器上，私有自己留着进行认证。



① 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。



② 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。



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-v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–u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。

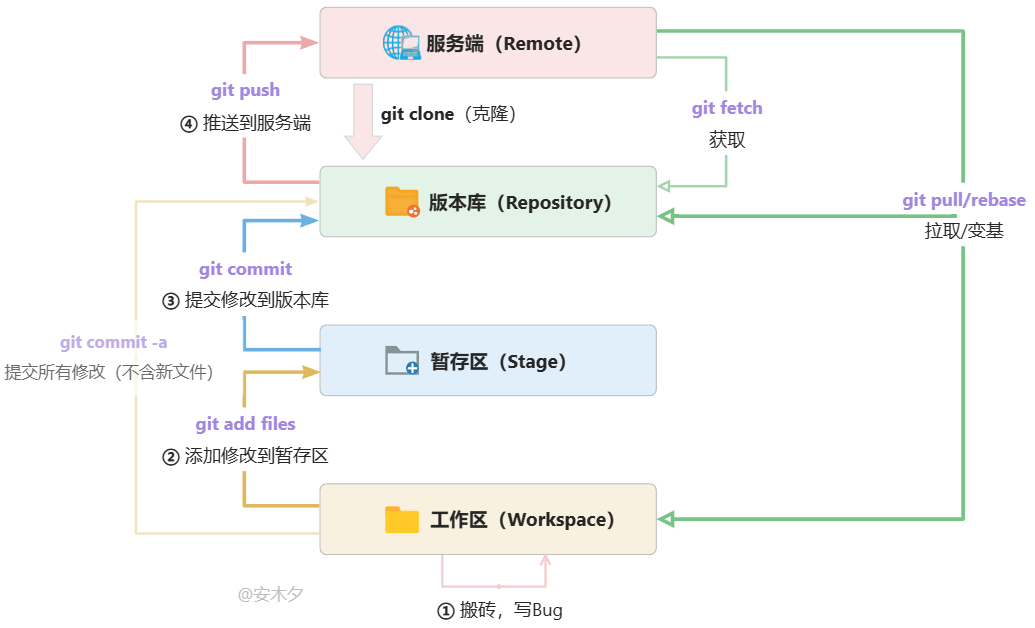
推送（push）：推送本地仓库到远程仓库。

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。



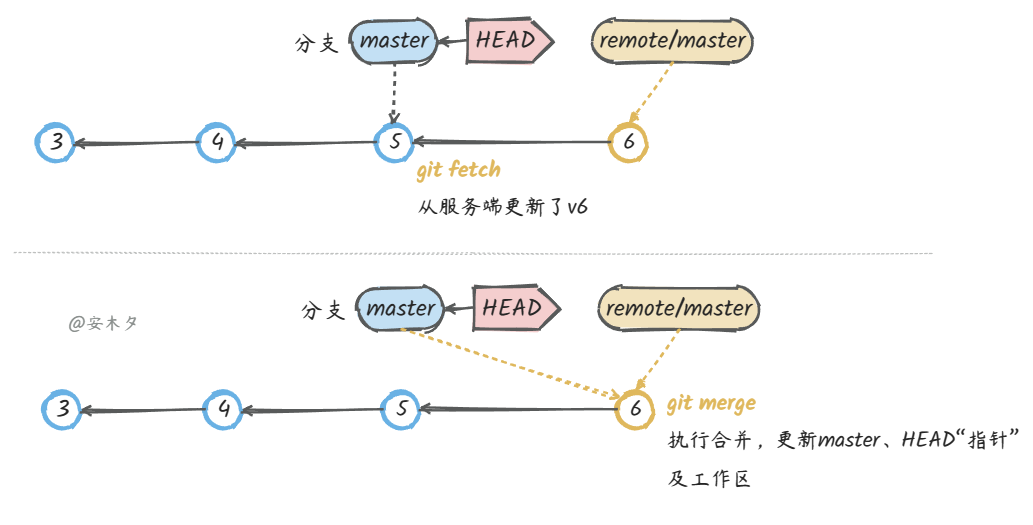
#### 6.4、fetch与pull有什么不同？

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 fetchremote: 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 0Unpacking 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 mergeUpdating 2ba12ca..c64f5b5Fast-forward

README.md | 2 +-

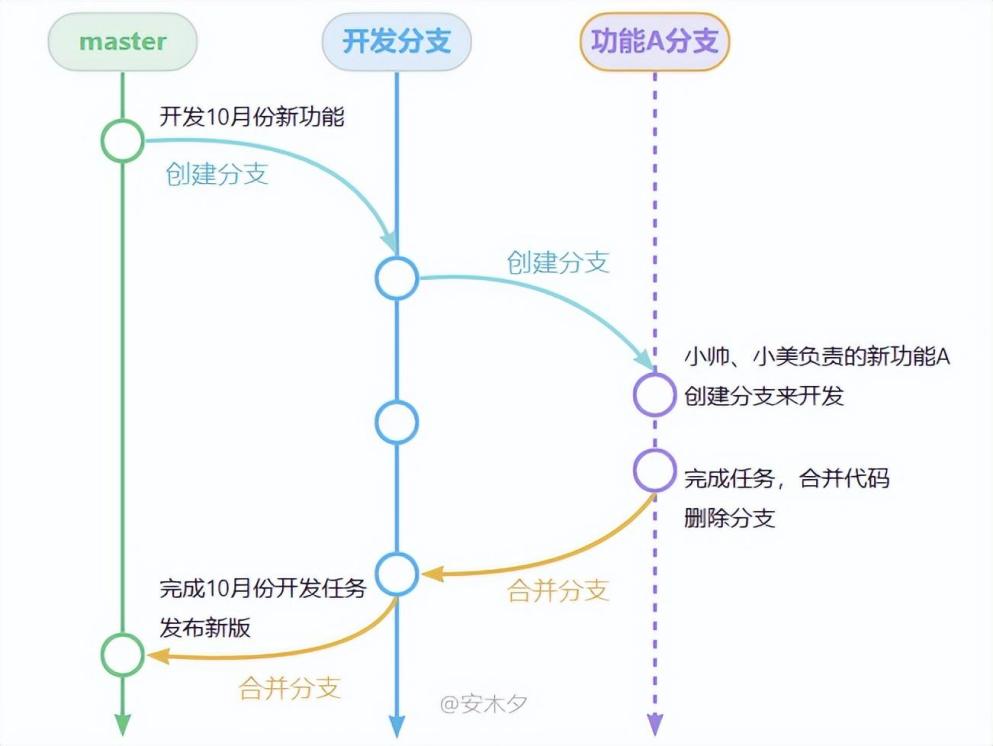
1 file changed, 1 insertion(+), 1 deletion(-)

### 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-v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dev分支，-D（大写）强制删除git checkout -b dev从当前分支创建并切换到dev分支git checkout -b feature1 dev从本地dev分支代码创建一个 feature1分支，并切换到新分支git branch [branch] [commit]新建一个分支，指向指定commit idgit 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remote的hotfix分支创建本地hotfix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/hotfixgit 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HEAD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dev分支到当前分支，参数--no-ff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master分支✅switch：新的分支切换指令切换功能和checkout一样，switch只单纯的用于切换git switch master切换到已有的master分支git switch -c dev创建并切换到新的dev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。

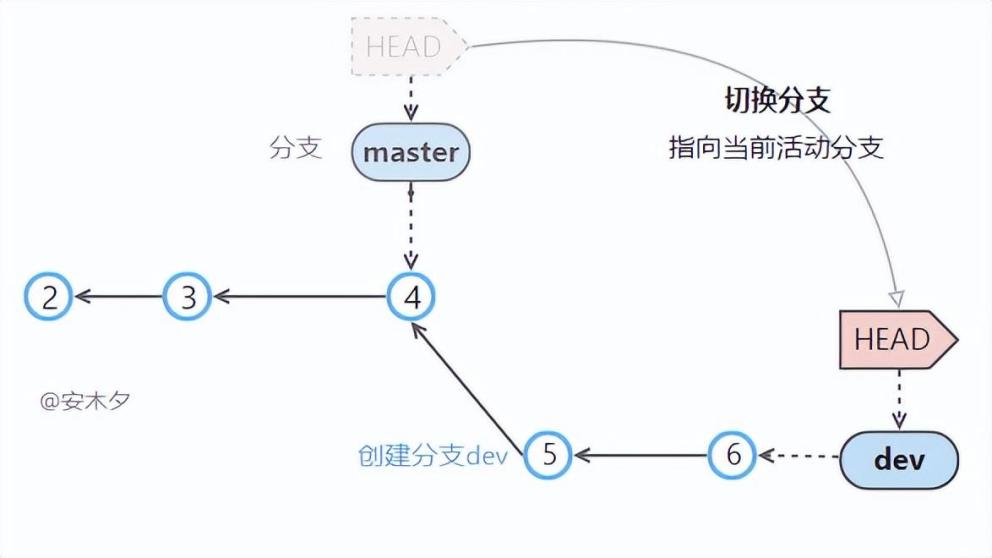
git switch：专门用来实现分支切换。

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/HEADref: refs/heads/dev

使用 git checkout dev切换分支时，干了两件事：

1. HEAD指向dev：修改HEAD的“指针”引用，指向dev分支。
2. 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”。

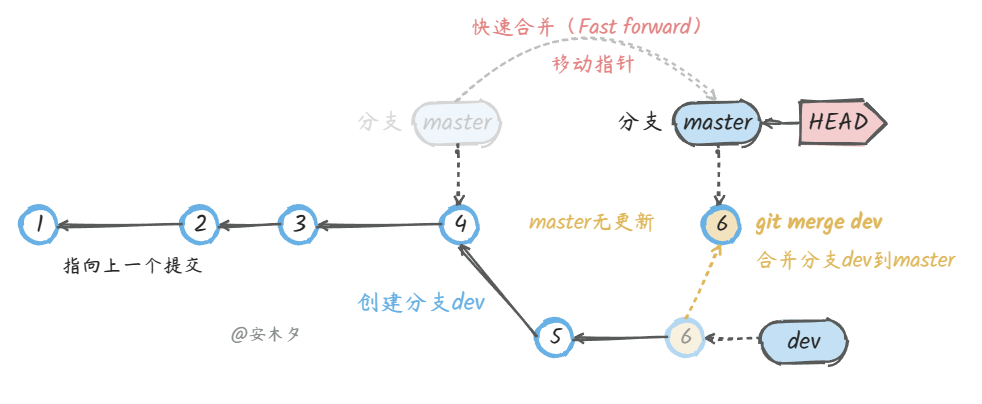
#### 7.4、合并merge&冲突

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。

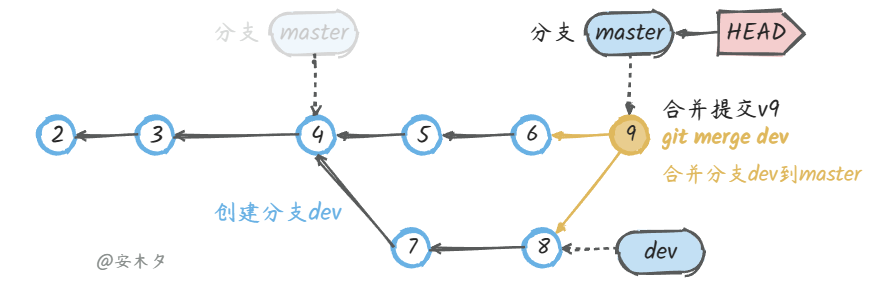
强制不用快速合并：git merge --no-ff -m "merge with no-ff" dev，参数--no-ff不启用快速合并，会产生一个新的合并提交记录。

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（和并）提交v9。

合并提交v9就有两个祖先v6、v8。

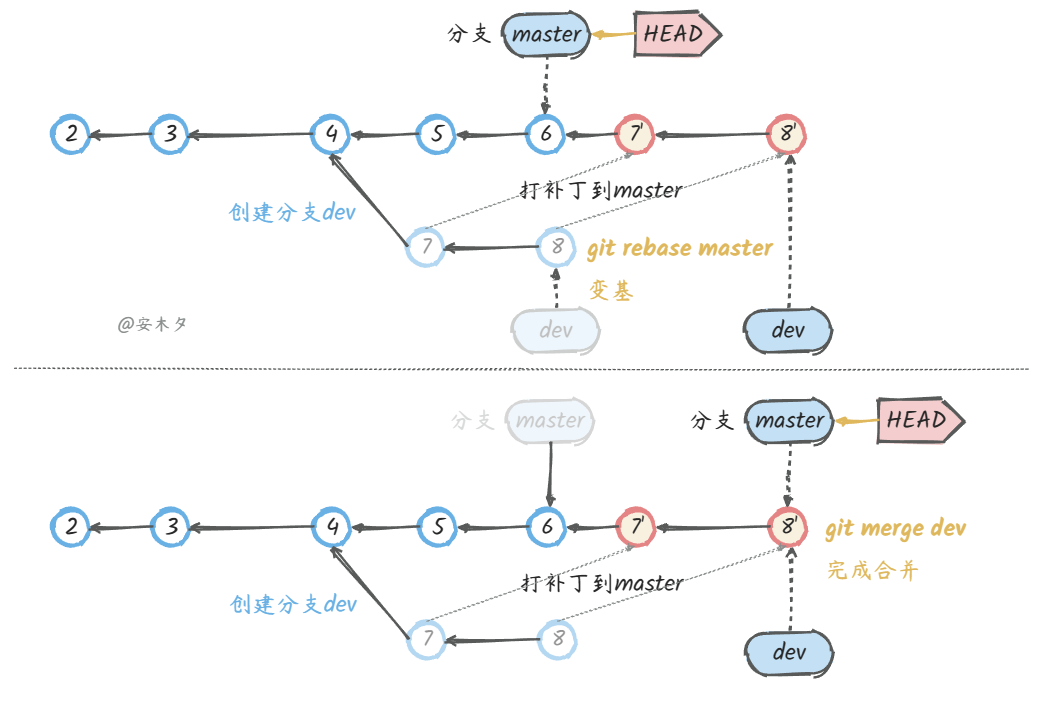
处理冲突<<<<<<< HEAD

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。



#### 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

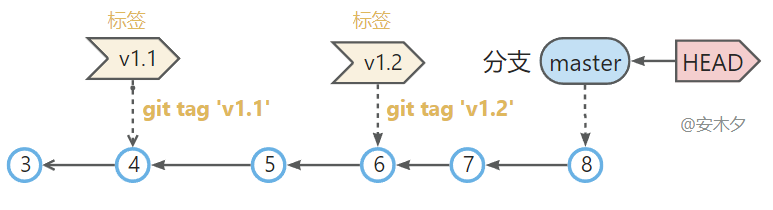
$ git merge dev

### 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

a2e2c9caea35e176cf61e96ad9d5a929cfb82461

# 查看标签列表

$ git tag

v1

### 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！

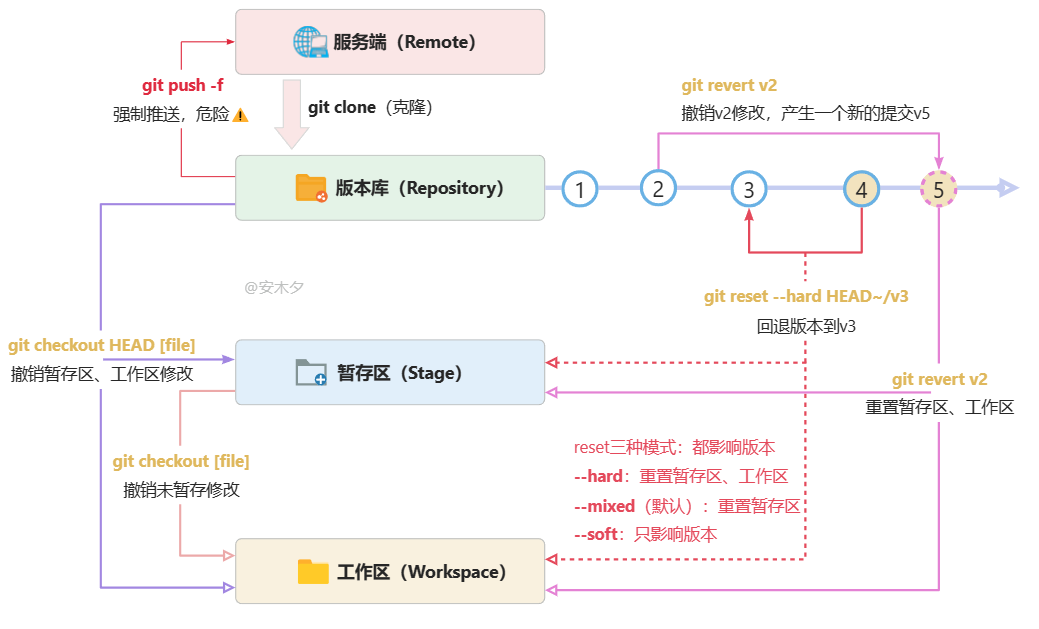
还没提交的怎么撤销？checkout、reset

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。

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，file指定文件git checkout HEAD .撤销工作区、暂存区的修改，用HEAD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，file指定文件

git reset撤销暂存区状态，同git reset HEAD，不影响工作区git reset HEAD [file]同上，指定文件file，HEAD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git checkout签出git reset --soft [commit]移动分支master、HEAD到指定的版本，不影响暂存区、工作区，需手动git checkout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push则重新push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

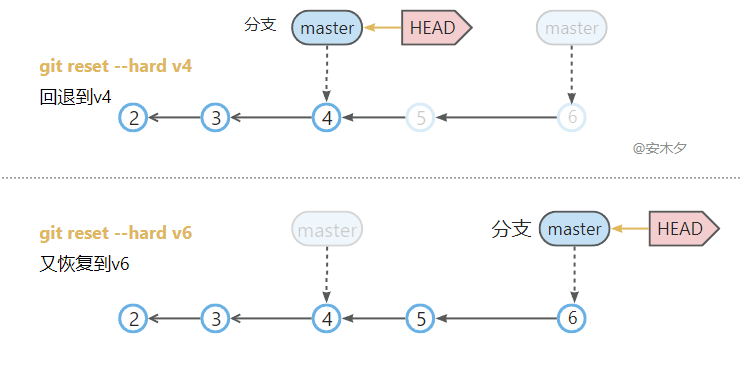
#### 9.2、回退版本reset

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。

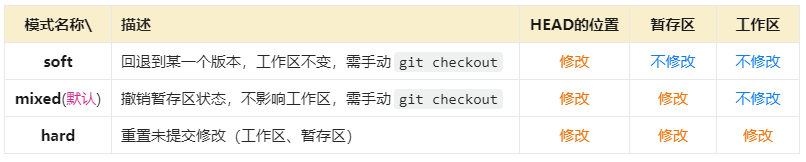
也可以重新恢复到v6版本：git reset --hard v6，就是移动master、HEAD的“指针”地址。



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git checkout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git checkout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

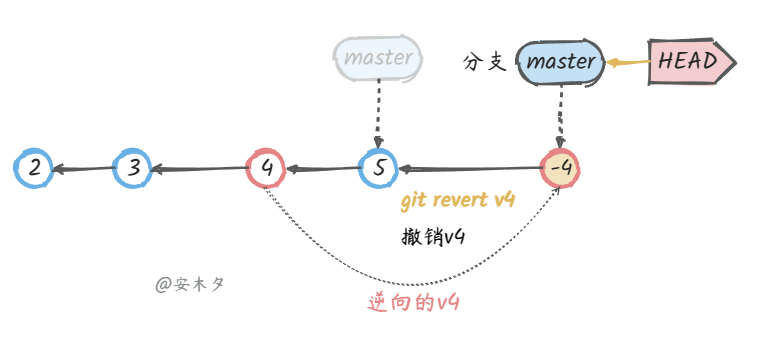
$ git reset --hard 5f8b961 # 查看所有历史提交记录

$ 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bug4869ff7 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checkoutresetrevert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]

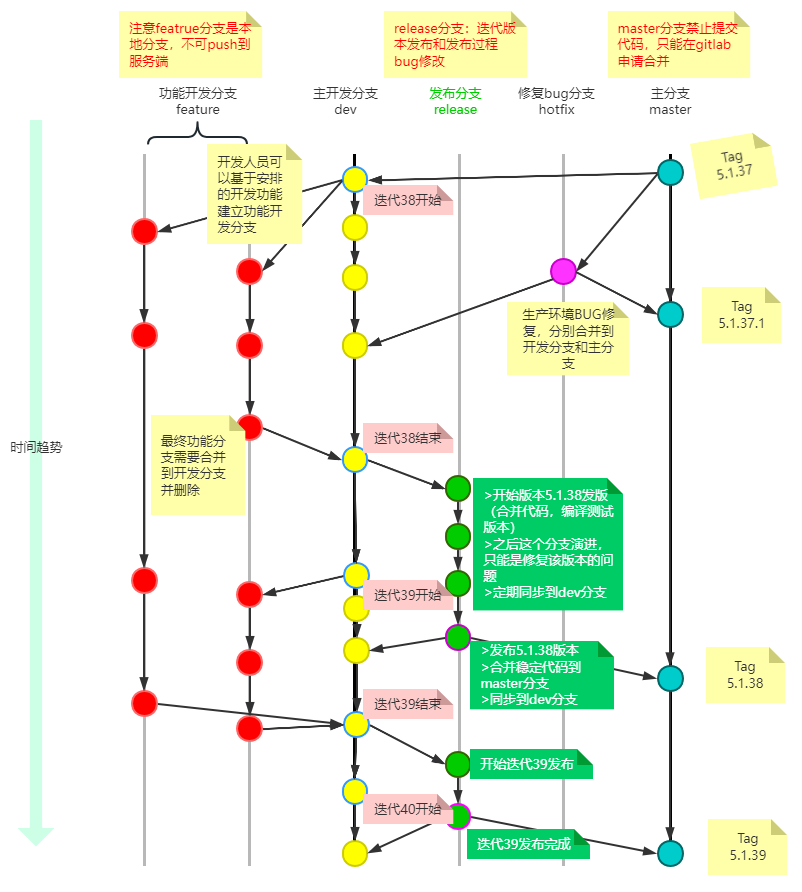
安全性只针对未提交修改，安全如回退了已push提交，不安全

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）。

### 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：原图地址-processon



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。

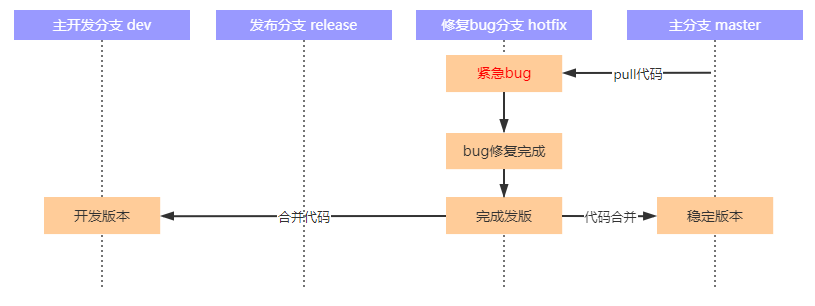
热修复分支：hotfix，针对线上紧急问题、bug修复的代码分支，修复完后合并到主分支、开发分支。

① 切换到hotfix分支，从master更新代码；

② 修复bug；

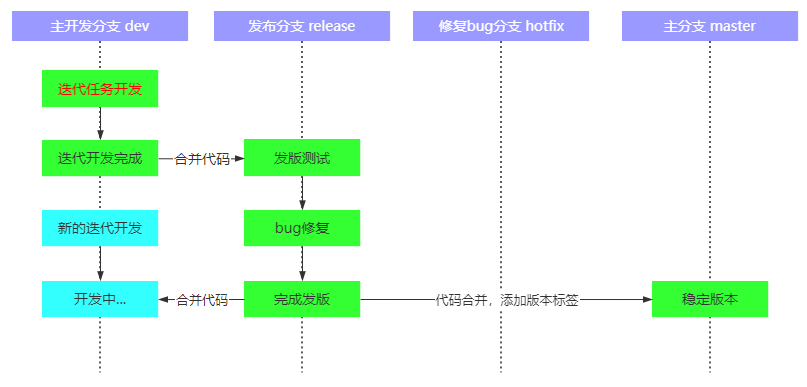
③ 合并代码到dev分支，在本地Git中操作即可；

④ 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.

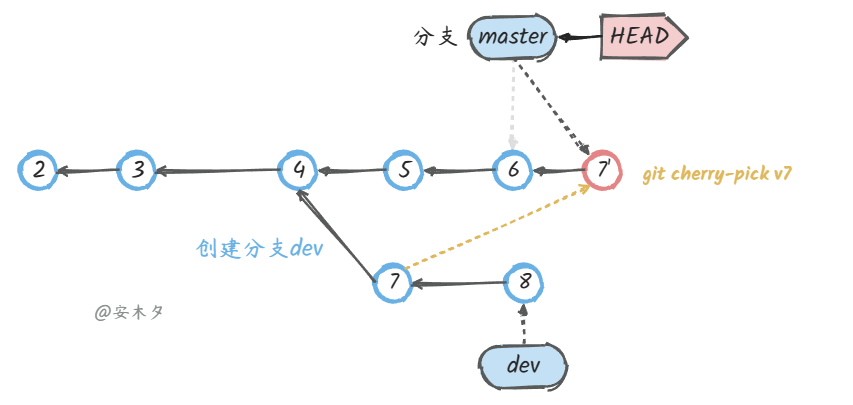
如果切换分支时，未提交修改的内容没有冲突，是可以成功切换的，未提交修改会被带过去。

指令描述git stash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git stash list查看所有被隐藏的内容列表git stash pop恢复被隐藏的内容，同时删除隐藏记录git stash save "message"同git stash，可以备注说明messagegit stash apply恢复被隐藏的文件，但是隐藏记录不删除git stash drop删除隐藏记录

当然这里先提交到本地也是可以的，只是提交不是一个完整的功能代码，而是残缺的一部分，影响也不大。

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]