

Git面试真题（10题）

1. 说说你对版本管理的理解？常用的版本管理工具有哪些？



1.1. 是什么

版本控制（Version control），是维护工程蓝图的标准作法，能追踪工程蓝图从诞生一直到定案的过程。此外，版本控制也是一种软件工程技巧，借此能在软件开发的过程中，确保由不同人所编辑的同一程序文件都得到同步。

透过文档控制，能记录任何工程项目内各个模块的改动历程，并为每次改动编上序号。

一种简单的版本控制形式如下：赋给图的初版一个版本等级“A”。当做了第一次改变后，版本等级改为“B”，以此类推。

版本控制能提供项目的设计者，将设计恢复到之前任一状态的选择权。

简言之，你的修改只要提到到版本控制系统，基本都可以找回，版本控制系统就像一台时光机器，可以让你回到任何一个时间点。

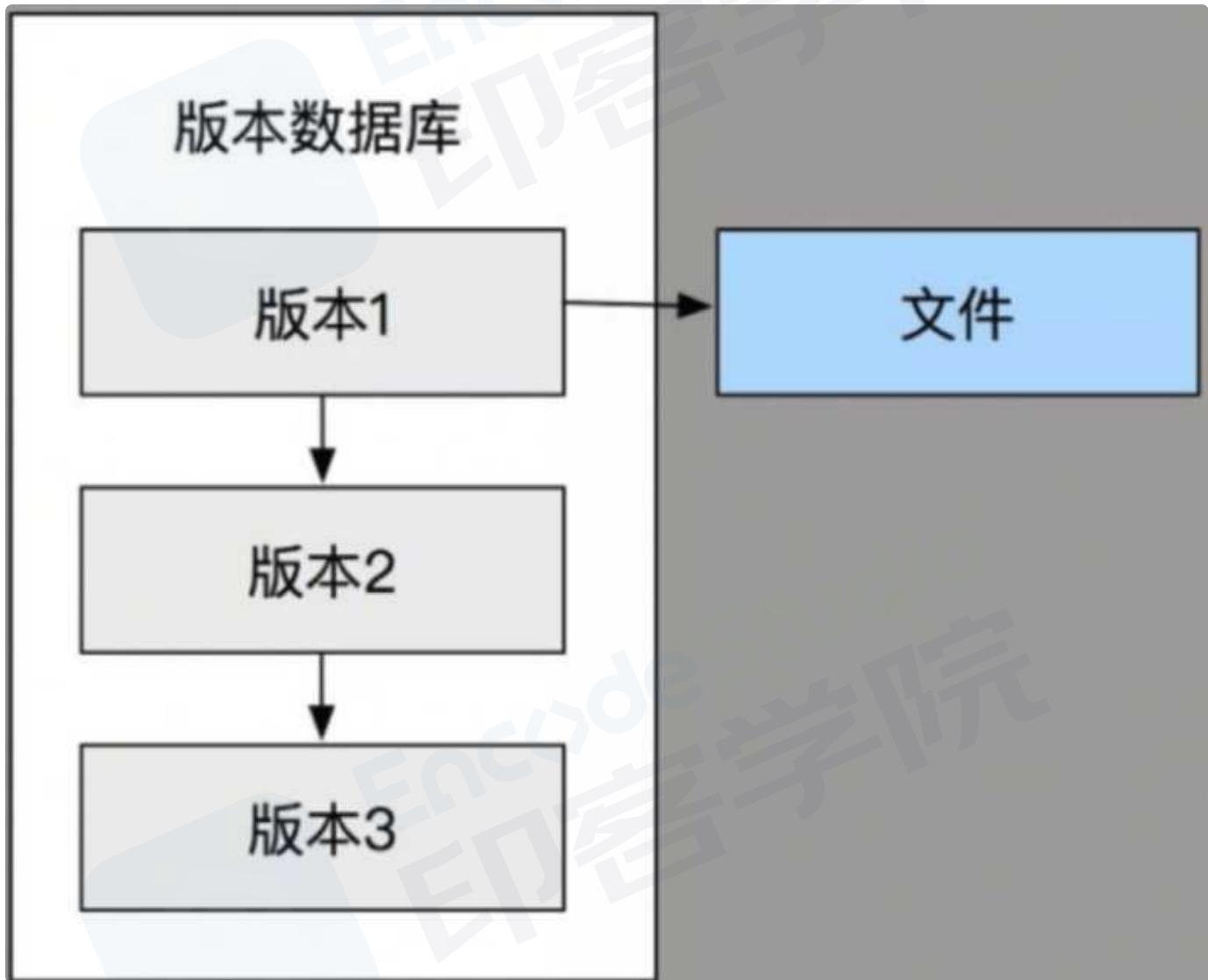
1.2. 有哪些

版本控制系统在当今的软件开发中，被认为是理所当然的配备工具之一，根据类别可以分成：

- 本地版本控制系统
- 集中式版本控制系统
- 分布式版本控制系统

1.2.1. 本地版本控制系统

结构如下图所示：



优点：

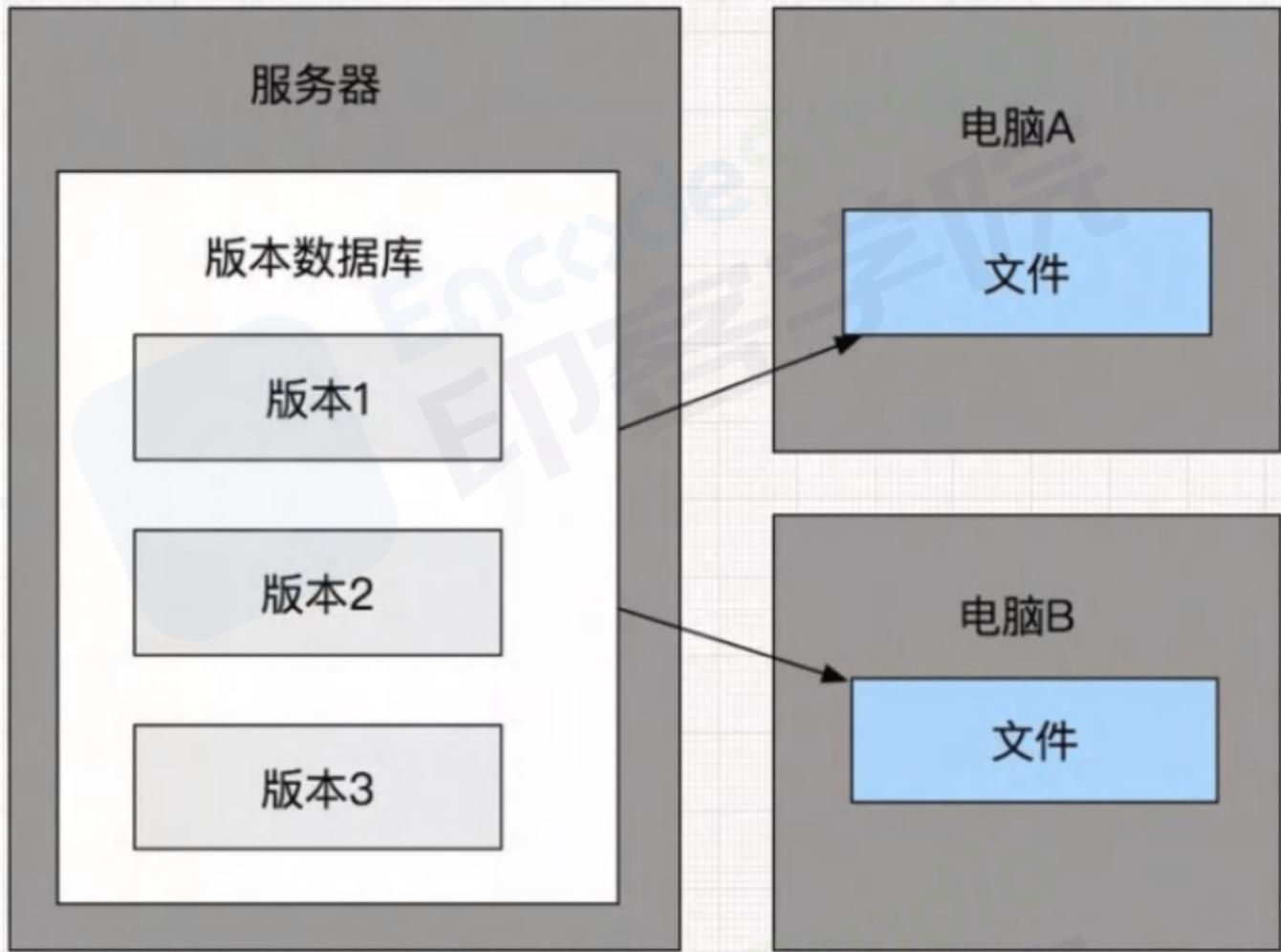
- 简单，很多系统中都有内置
- 适合管理文本，如系统配置

缺点：

- 其不支持远程操作，因此并不适合多人版本开发

1.2.2. 集中式版本控制系统

结构如下图所示：



优点：

- 适合多人团队协作开发
- 代码集中化管理

缺点：

- 单点故障
- 必须联网，无法单机工作

代表工具有 **SVN** 、 **CVS** :

1.2.3. SVN

TortoiseSVN 是一款非常易于使用的跨平台的 版本控制/版本控制/源代码控制软件

1.2.4. CVS

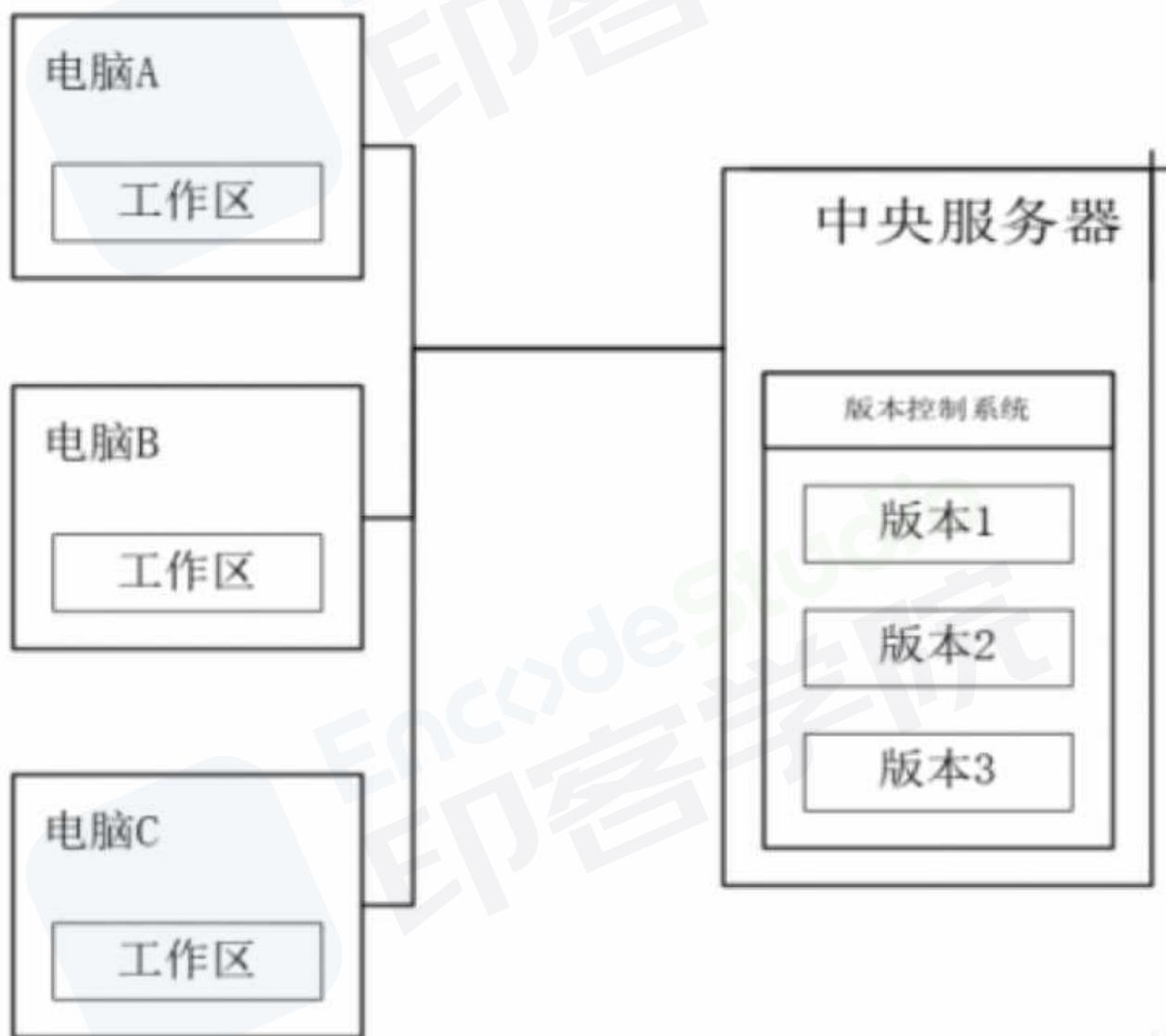
CVS 是版本控制系统，是源配置管理（SCM）的重要组成部分。使用它，您可以记录源文件和文档的历史记录。

老牌的版本控制系统，它是基于客户端/服务器的行为使得其可容纳多用户，构成网络也很方便。

这一特性使得 CVS 成为位于不同地点的人同时处理数据文件（特别是程序的源代码）时的首

1.2.4.1. 分布式版本控制系统

结构如下图：



优点：

- 适合多人团队协作开发
- 代码集中化管理

- 可以离线工作
- 每个计算机都是一个完整仓库

分布式版本管理系统每个计算机都有一个完整的仓库，可本地提交，可以做到离线工作，则不用像集中管理那样因为断网情况而无法工作

代表工具为 `Git` 、 `HG` :

1.2.5. Git

`Git` 是目前世界上最先进的分布式版本控制系统，旨在快速高效地处理从小型到大型项目的所有事务
特性：易于学习，占用内存小，具有闪电般快速的性能

使用 `Git` 和 `Gitlab` 搭建版本控制环境是现在互联网公司最流行的版本控制方式

1.2.6. HG

`Mercurial` 是一个免费的分布式源代码管理工具。它可以有效地处理任何规模的项目，并提供简单直观的界面

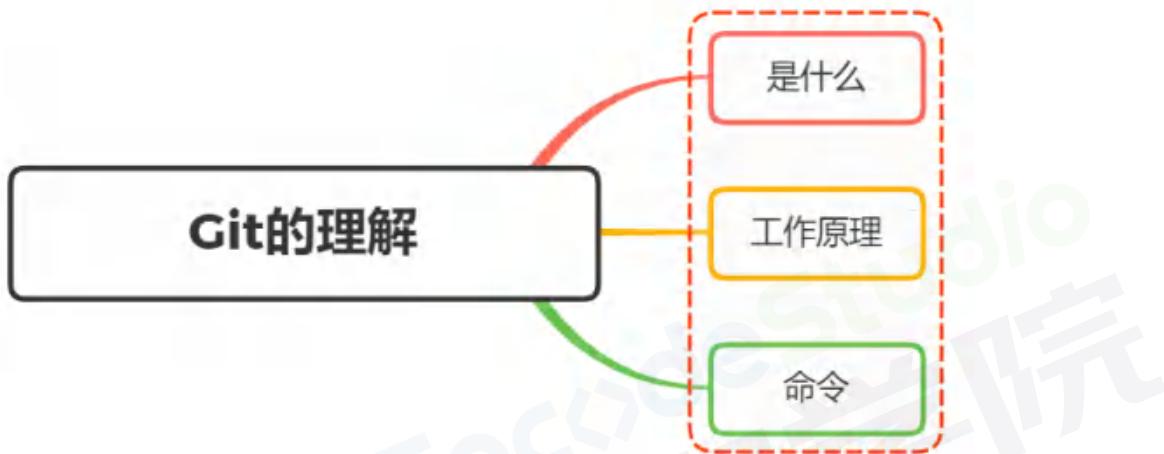
`Mercurial` 是一种轻量级分布式版本控制系统，采用 `Python` 语言实现，易于学习和使用，扩展性强

1.3. 总结

版本控制系统的优点如下：

- 记录文件所有历史变化，这是版本控制系统的基本能力
- 随时恢复到任意时间点，历史记录功能使我们不怕改错代码了
- 支持多功能并行开发，通常版本控制系统都支持分支，保证了并行开发的可行
- 多人协作并行开发，对于多人协作项目，支持多人协作开发的版本管理将事半功倍

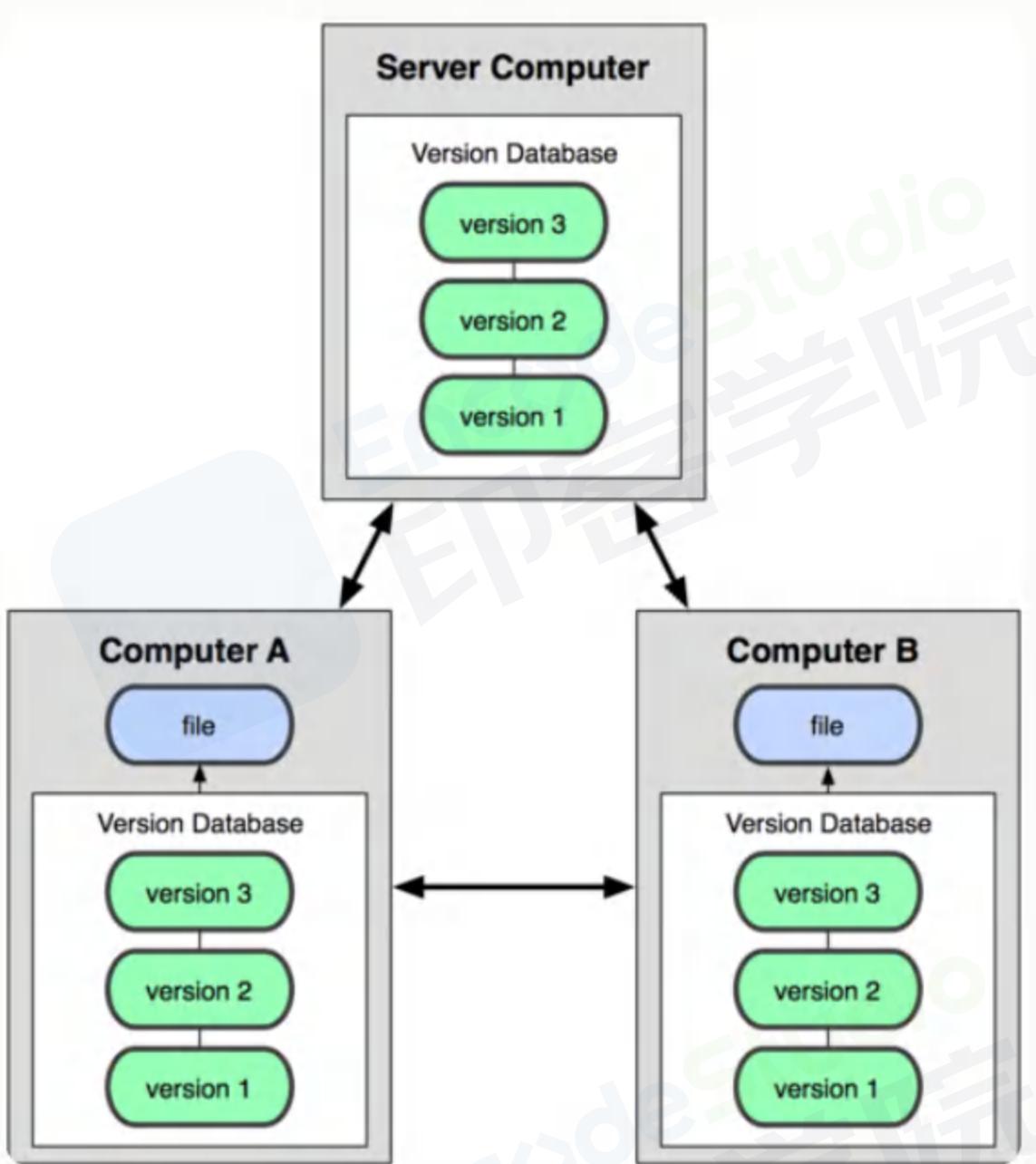
2. 说你对Git的理解？



2.1. 是什么

git，是一个分布式版本控制软件，最初目的是为更好地管理 Linux 内核开发而设计

分布式版本控制系统的客户端并不只提取最新版本的文件快照，而是把代码仓库完整地镜像下来。这么一来，任何一处协同工作用的服务器发生故障，事后都可以用任何一个镜像出来的本地仓库恢复



项目开始，只有一个原始版仓库，别的机器可以 `clone` 这个原始版本库，那么所有 `clone` 的机器，它们的版本库其实都是一样的，并没有主次之分

所以在实现团队协作的时候，只要有一台电脑充当服务器的角色，其他每个人都从这个“服务器”仓库 `clone` 一份到自己的电脑上，并且各自把各自的提交推送到服务器仓库里，也从服务器仓库中拉取别人的提交

`github` 实际就可以充当这个服务器角色，其是一个开源协作社区，提供 `Git` 仓库托管服务，既可以让你参与你的开源项目，也可以参与别人的开源项目

2.2. 工作原理

当我们通过 `git init` 创建或者 `git clone` 一个项目的时候，项目目录会隐藏一个 `.git` 子目录，其作用是用来跟踪管理版本库的

`Git` 中所有数据在存储前都计算校验和，然后以校验和来引用，所以在我们修改或者删除文件的时候，`git` 能够知道

`Git` 用以计算校验和的机制叫做 SHA-1 散列（hash，哈希），这是一个由 40 个十六进制字符（0-9 和 a-f）组成字符串，基于 `Git` 中文件的内容或目录结构计算出来，如下：

LaTeX | 复制代码

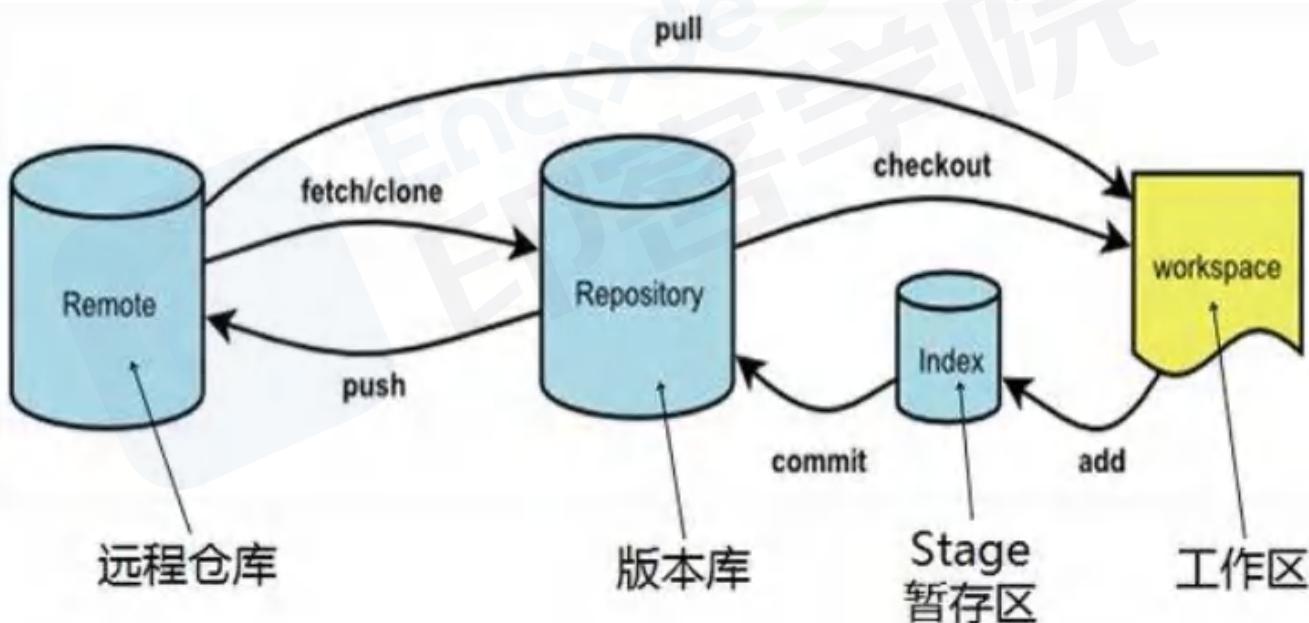
```
1 24b9da6552252987aa493b52f8696cd6d3b00373
```

当我们修改文件的时候，`git` 就会修改文件的状态，可以通过 `git status` 进行查询，状态情况如下：

- 已修改 (modified)：表示修改了文件，但还没保存到数据库中。
- 已暂存 (staged)：表示对一个已修改文件的当前版本做了标记，使之包含在下次提交的快照中。
- 已提交 (committed)：表示数据已经安全的保存在本地数据库中。

文件状态对应的，不同状态的文件在 `Git` 中处于不同的工作区域，主要分成了四部分：

- 工作区：相当于本地写代码的区域，如 `git clone` 一个项目到本地，相当于本地克隆了远程仓库项目的一个副本
- 暂存区：暂存区是一个文件，保存了下次将提交的文件列表信息，一般在 `Git` 仓库目录中
- 本地仓库：提交更新，找到暂存区域的文件，将快照永久性存储到 `Git` 本地仓库
- 远程仓库：远程的仓库，如 `github`



2.3. 命令

从上图可以看到，`git` 日常简单的使用就只有上图6个命令：

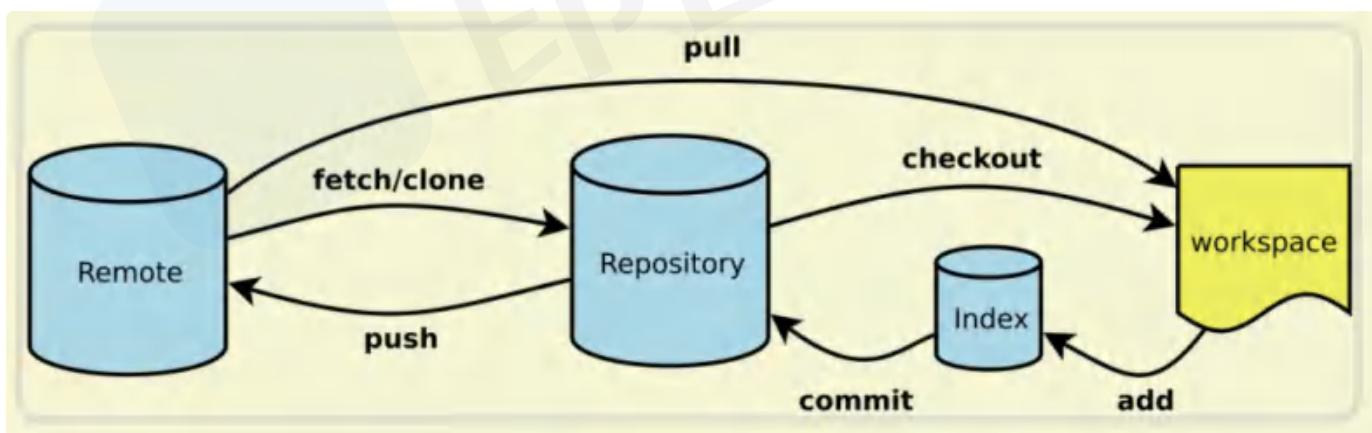
- add
- commit
- push
- pull
- clone
- checkout

3. 说说Git常用的命令有哪些？



3.1. 前言

`git` 的操作可以通过命令的形式来执行，日常使用就如下图6个命令即可



实际上，如果想要熟练使用，超过60多个命令需要了解，下面则介绍下常见的的 `git` 命令

3.2. 有哪些

3.3. 配置

`Git` 自带一个 `git config` 的工具来帮助设置控制 `Git` 外观和行为的配置变量，在我们安装完 `git` 之后，第一件事就是设置你的用户名和邮件地址

后续每一个提交都会使用这些信息，它们会写入到你的每一次提交中，不可更改

设置提交代码时的用户信息命令如下：

- `git config [--global] user.name "[name]"`
- `git config [--global] user.email "[email address]"`

3.3.1. 启动

一个 `git` 项目的初始有两个途径，分别是：

- `git init [project-name]`: 创建或在当前目录初始化一个git代码库
- `git clone url`: 下载一个项目和它的整个代码历史

3.3.2. 日常基本操作

在日常工作中，代码常用的基本操作如下：

- `git init` 初始化仓库，默認為 master 分支
- `git add .` 提交全部文件修改到缓存区
- `git add <具体某个文件路径+全名>` 提交某些文件到缓存区
- `git diff` 查看当前代码 add后，会 add 哪些内容
- `git diff --staged` 查看现在 commit 提交后，会提交哪些内容
- `git status` 查看当前分支状态
- `git pull <远程仓库名> <远程分支名>` 拉取远程仓库的分支与本地当前分支合并
- `git pull <远程仓库名> <远程分支名>:<本地分支名>` 拉取远程仓库的分支与本地某个分支合并
- `git commit -m "<注释>"` 提交代码到本地仓库，并写提交注释
- `git commit -v` 提交时显示所有diff信息

- git commit --amend [file1] [file2] 重做上一次commit，并包括指定文件的新变化

关于提交信息的格式，可以遵循以下的规则：

- feat: 新特性，添加功能
- fix: 修改 bug
- refactor: 代码重构
- docs: 文档修改
- style: 代码格式修改，注意不是 css 修改
- test: 测试用例修改
- chore: 其他修改，比如构建流程，依赖管理

3.3.3. 分支操作

- git branch 查看本地所有分支
- git branch -r 查看远程所有分支
- git branch -a 查看本地和远程所有分支
- git merge <分支名> 合并分支
- git merge --abort 合并分支出现冲突时，取消合并，一切回到合并前的状态
- git branch <新分支名> 基于当前分支，新建一个分支
- git checkout --orphan <新分支名> 新建一个空分支（会保留之前分支的所有文件）
- git branch -D <分支名> 删除本地某个分支
- git push <远端库名> :<分支名> 删除远程某个分支
- git branch <新分支名称> <提交ID> 从提交历史恢复某个删掉的某个分支
- git branch -m <原分支名> <新分支名> 分支更名
- git checkout <分支名> 切换到本地某个分支
- git checkout <远端库名>/<分支名> 切换到线上某个分支
- git checkout -b <新分支名> 把基于当前分支新建分支，并切换为这个分支

3.3.4. 远程同步

远程操作常见的命令：

- git fetch [remote] 下载远程仓库的所有变动
- git remote -v 显示所有远程仓库
- git pull [remote] [branch] 拉取远程仓库的分支与本地当前分支合并

- git fetch 获取线上最新版信息记录，不合并
- git push [remote] [branch] 上传本地指定分支到远程仓库
- git push [remote] --force 强行推送当前分支到远程仓库，即使有冲突
- git push [remote] --all 推送所有分支到远程仓库

3.3.5. 撤销

- git checkout [file] 恢复暂存区的指定文件到工作区
- git checkout [commit] [file] 恢复某个commit的指定文件到暂存区和工作区
- git checkout . 恢复暂存区的所有文件到工作区
- git reset [commit] 重置当前分支的指针为指定commit，同时重置暂存区，但工作区不变
- git reset --hard 重置暂存区与工作区，与上一次commit保持一致
- git reset [file] 重置暂存区的指定文件，与上一次commit保持一致，但工作区不变
- git revert [commit] 后者的所有变化都将被前者抵消，并且应用到当前分支

`reset`：真实硬性回滚，目标版本后面的提交记录全部丢失了

`revert`：同样回滚，这个回滚操作相当于一个提价，目标版本后面的提交记录也全部都有

3.3.6. 存储操作

你正在进行项目中某一部分的工作，里面的东西处于一个比较杂乱的状态，而你想转到其他分支上进行一些工作，但又不想提交这些杂乱的代码，这时候可以将代码进行存储

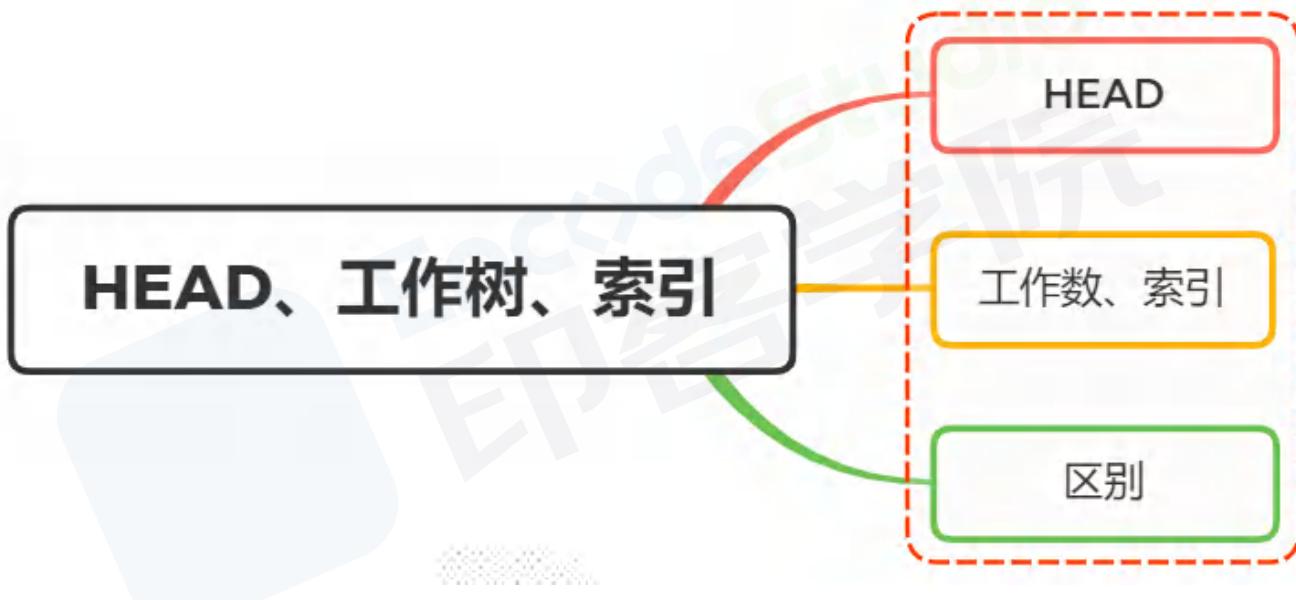
- git stash 暂时将未提交的变化移除
- git stash pop 取出储藏中最后存入的工作状态进行恢复，会删除储藏
- git stash list 查看所有储藏中的工作
- git stash apply <储藏的名称> 取出储藏中对应的工作状态进行恢复，不会删除储藏
- git stash clear 清空所有储藏中的工作
- git stash drop <储藏的名称> 删除对应的某个储藏

3.4. 总结

`git` 常用命令速查表如下所示：

Git 常用命令速查表	
创建版本库	<code>\$ git clone <url></code> #克隆远程版本库 <code>\$ git init</code> #初始化本地版本库
修改和提交	<code>\$ git status</code> #查看状态 <code>\$ git diff</code> #查看变更内容 <code>\$ git add .</code> #跟踪所有改动过的文件 <code>\$ git add <file></code> #添加指定的文件 <code>\$ git mv <old> <new></code> #文件改名 <code>\$ git rm <file></code> #删除文件 <code>\$ git rm --cached <file></code> #停止跟踪文件但不删除 <code>\$ git commit -m "commit message"</code> #提交所有更新过的文件 <code>\$ git commit --amend</code> #修改最后一次提交
查看提交历史	<code>\$ git log</code> #查看提交历史 <code>\$ git log -p <file></code> #查看指定文件的提交历史 <code>\$ git blame <file></code> #以列表方式查看指定文件的提交历史
撤销	<code>\$ git reset --hard HEAD</code> #撤消工作目录中所有未提交文件的修改内容 <code>\$ git checkout HEAD <file></code> #撤消指定的未提交文件的修改内容 <code>\$ git revert <commit></code> #撤消指定的提交
分支与标签	<code>\$ git branch</code> #显示所有本地分支 <code>\$ git checkout <branch/tag></code> #切换到指定分支或标签 <code>\$ git branch <new-branch></code> #创建新分支 <code>\$ git branch -d <branch></code> #删除本地分支 <code>\$ git tag</code> #列出所有本地标签 <code>\$ git tag <tagname></code> #基于最新提交创建标签 <code>\$ git tag -d <tagname></code> #删除标签
合并与衍合	<code>\$ git merge <branch></code> #合并指定分支到当前分支 <code>\$ git rebase <branch></code> #衍合指定分支到当前分支
远程操作	<code>\$ git remote -v</code> #查看远程版本库信息 <code>\$ git remote show <remote></code> #查看指定远程版本库信息 <code>\$ git remote add <remote> <url></code> #添加远程版本库 <code>\$ git fetch <remote></code> #从远程库获取代码 <code>\$ git pull <remote> <branch></code> #下载代码及快进合并 <code>\$ git push <remote> <branch></code> #上传代码及快速合并 <code>\$ git push <remote> :<branch/tag-name></code> #删除远程分支或标签 <code>\$ git push --tags</code> #上传所有标签

4. 说说Git 中 HEAD、工作树和索引之间的区别?

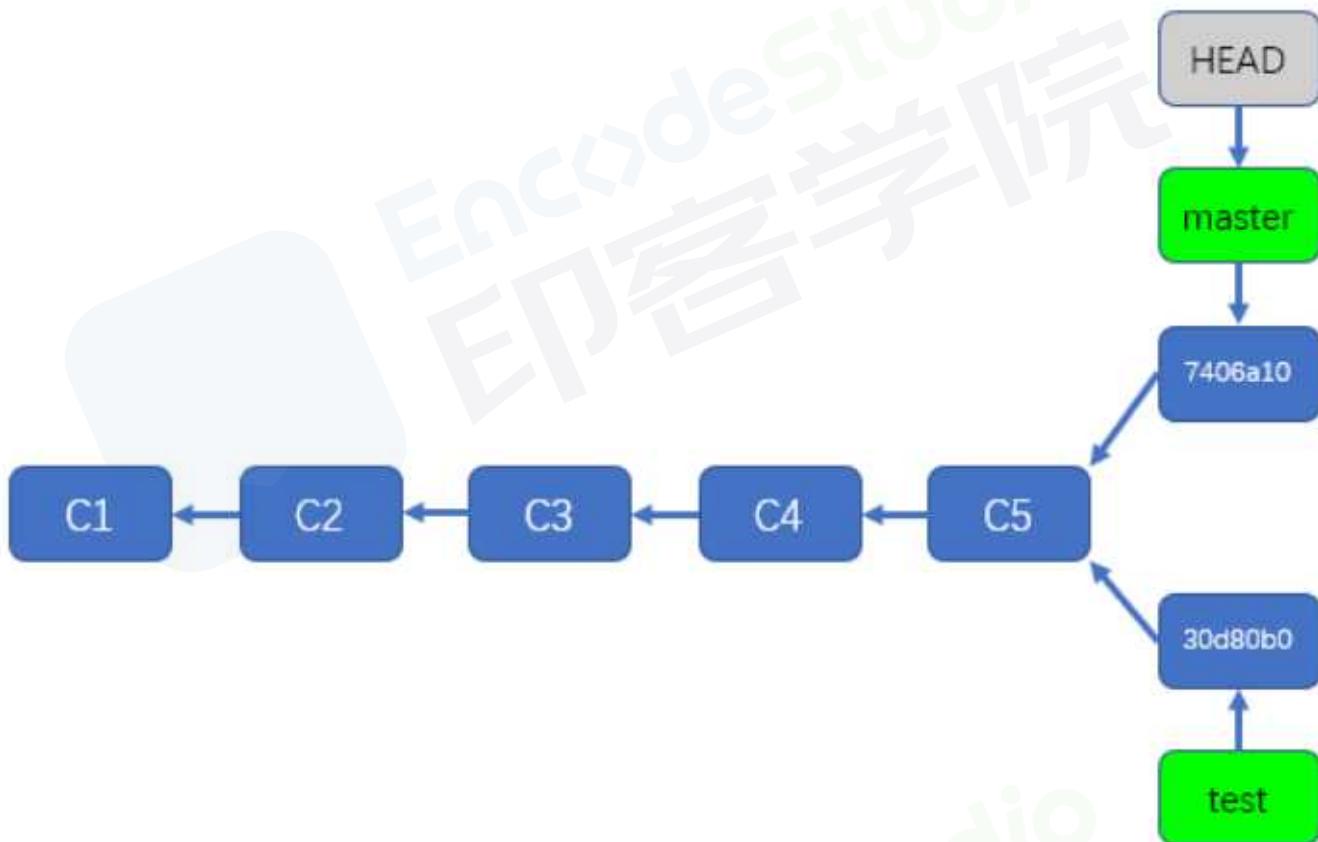


4.1. HEAD

在 `git` 中，可以存在很多分支，其本质上是一个指向 `commit` 对象的可变指针，而 `Head` 是一个特别的指针，是一个指向你正在工作中的本地分支的指针

简单来讲，就是你现在在哪儿，`HEAD` 就指向哪儿

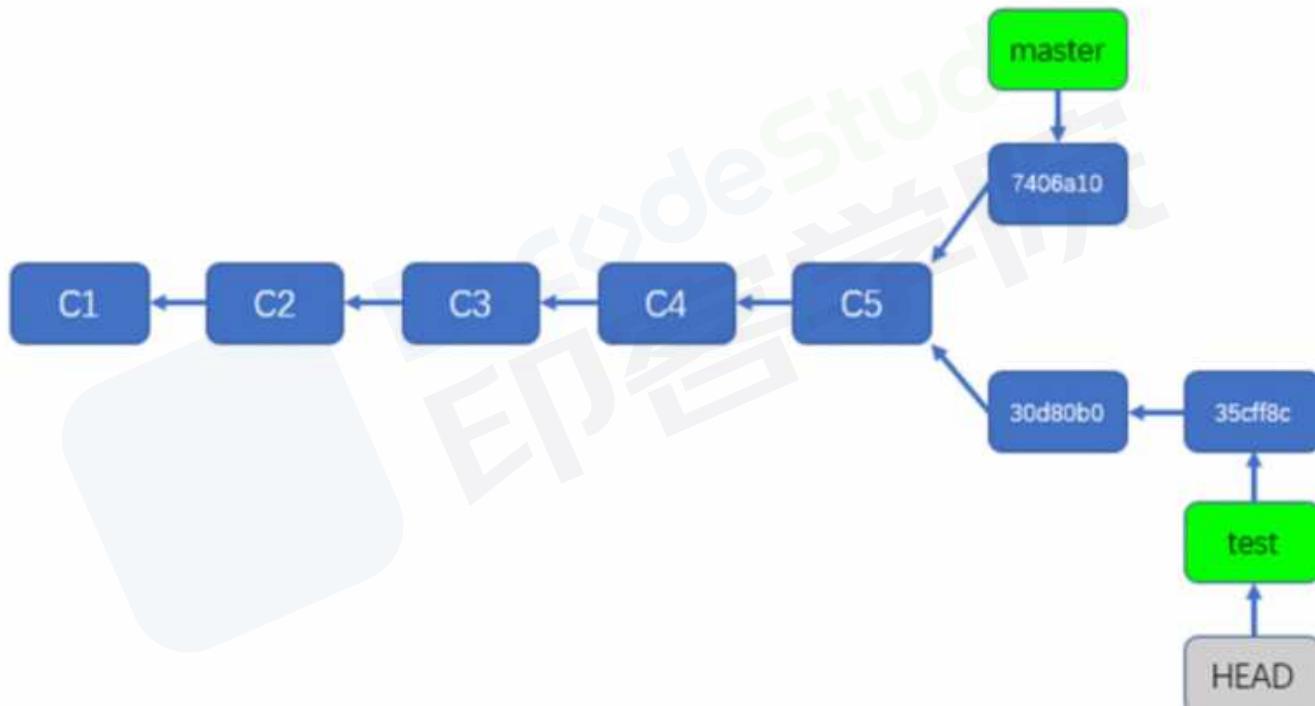
例如当前我们处于 `master` 分支，所以 `HEAD` 这个指针指向了 `master` 分支指针



然后通过调用 `git checkout test` 切换到 `test` 分支，那么 `HEAD` 则指向 `test` 分支，如下图：



但我们在 `test` 分支再一次 `commit` 信息的时候，`HEAD` 指针仍然指向了 `test` 分支指针，而 `test` 分支指针已经指向了最新创建的提交，如下图：



这个 `HEAD` 存储的位置就在 `.git/HEAD` 目录中，查看信息可以看到 `HEAD` 指向了另一个文件

```

Plain Text | 复制代码

1 $ cat .git/HEAD
2 ref: refs/heads/master
3
4 $ cat .git/refs/heads/master
5 7406a10efcc169bbab17827aeda189aa20376f7f

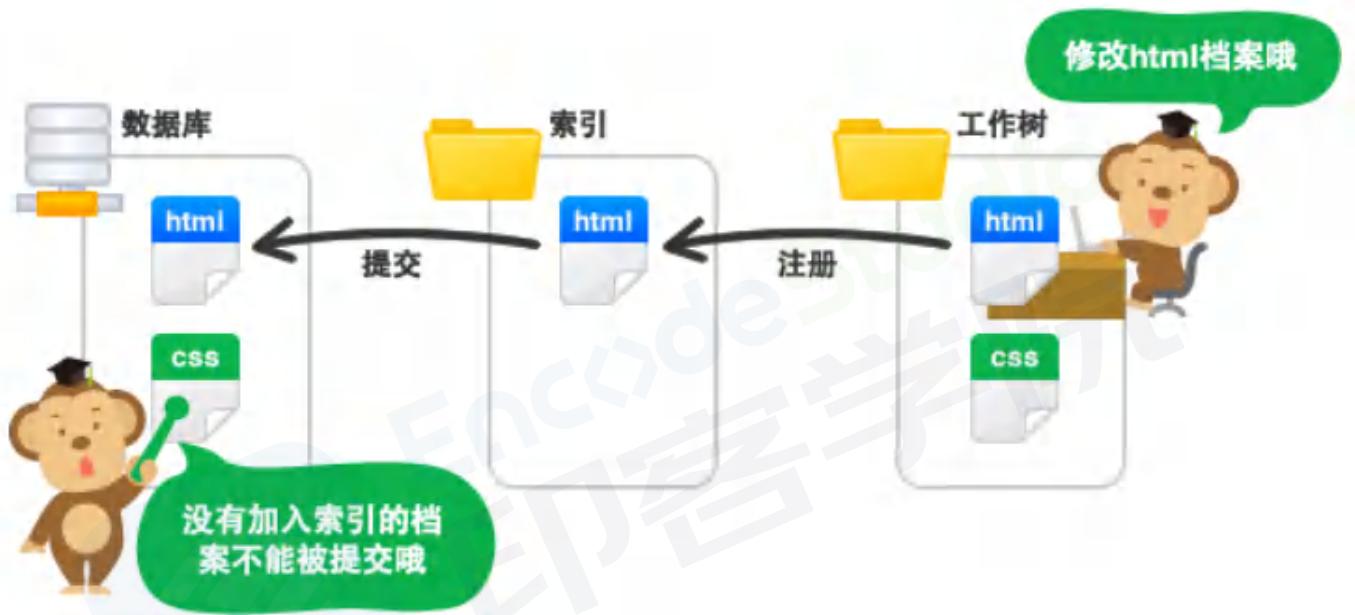
```

这个文件的内容是一串哈希码，而这个哈希码正是 `master` 分支上最新的提交所对应的哈希码
所以，当我们切换分支的时候，`HEAD` 指针通常指向我们所在的分支，当我们在某个分支上创建新的提交时，分支指针总是会指向当前分支的最新提交
所以，`HEAD`指针 ——> 分支指针 ——> 最新提交

4.2. 工作树和索引

在 `Git` 管理下，大家实际操作的目录被称为工作树，也就是工作区域

在数据库和工作树之间有索引，索引是为了向数据库提交作准备的区域，也被称为暂存区域



`Git` 在执行提交的时候，不是直接将工作树的状态保存到数据库，而是将设置在中间索引区域的状态保存到数据库

因此，要提交文件，首先需要把文件加入到索引区域中。

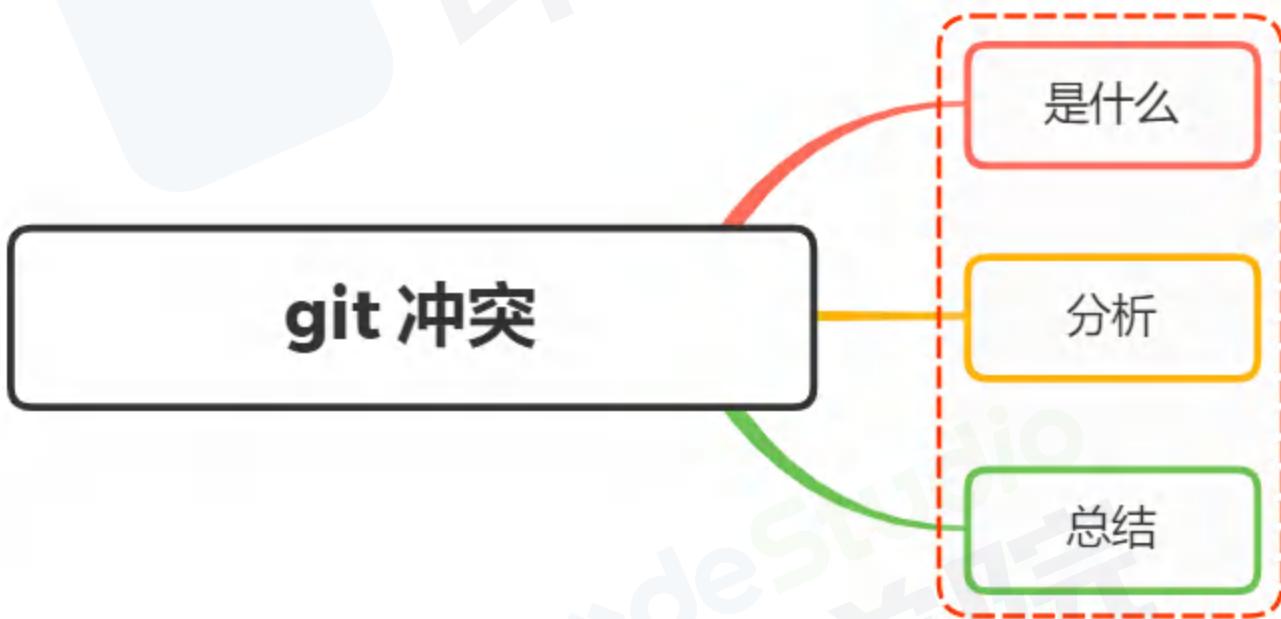
所以，凭借中间的索引，可以避免工作树中不必要的文件提交，还可以将文件修改内容的一部分加入索引区域并提交

4.3. 区别

从所在的位置来看：

- HEAD 指针通常指向我们所在的分支，当我们在某个分支上创建新的提交时，分支指针总是会指向当前分支的最新提交
- 工作树是查看和编辑的（源）文件的实际内容
- 索引是放置你想要提交给 git 仓库文件的地方，如工作树的代码通过 git add 则添加到 git 索引中，通过 git commit 则将索引区域的文件提交到 git 仓库中

5. 说说 git 发生冲突的场景？如何解决？



5.1. 是什么

一般情况下，出现分支的场景有如下：

- 多个分支代码合并到一个分支时
- 多个分支向同一个远端分支推送

具体情况就是，多个分支修改了同一个文件（任何地方）或者多个分支修改了同一个文件的名称

如果两个分支中分别修改了不同文件中的部分，是不会产生冲突，直接合并即可

应用在命令中，就是 `push`、`pull`、`stash`、`rebase` 等命令下都有可能产生冲突情况，从本质上讲，都是 `merge` 和 `patch`（应用补丁）时产生冲突

5.2. 分析

在本地主分支 master 创建一个 a.txt 文件，文件起始位置写上 master commit，如下：



```
a.txt M 1 master commit
```

然后提交到仓库：

- git add a.txt
- git commit -m 'master first commit'

创建一个新的分支 feature1 分支，并进行切换，如下：



```
Plain Text | 复制代码
```

```
1 git checkout -b feature1
```

然后修改 a.txt 文件首行文字为 feature1 commit，然后添加到暂存区，并开始进行提交到仓库：

- git add a.txt
- git commit -m 'feature1 first change'

然后通过 git checkout master 切换到主分支，通过 git merge 进行合并，发现不会冲突

此时 a.txt 文件的内容变成 feature1 commit，没有出现冲突情况，这是因为 git 在内部发生了快速合并

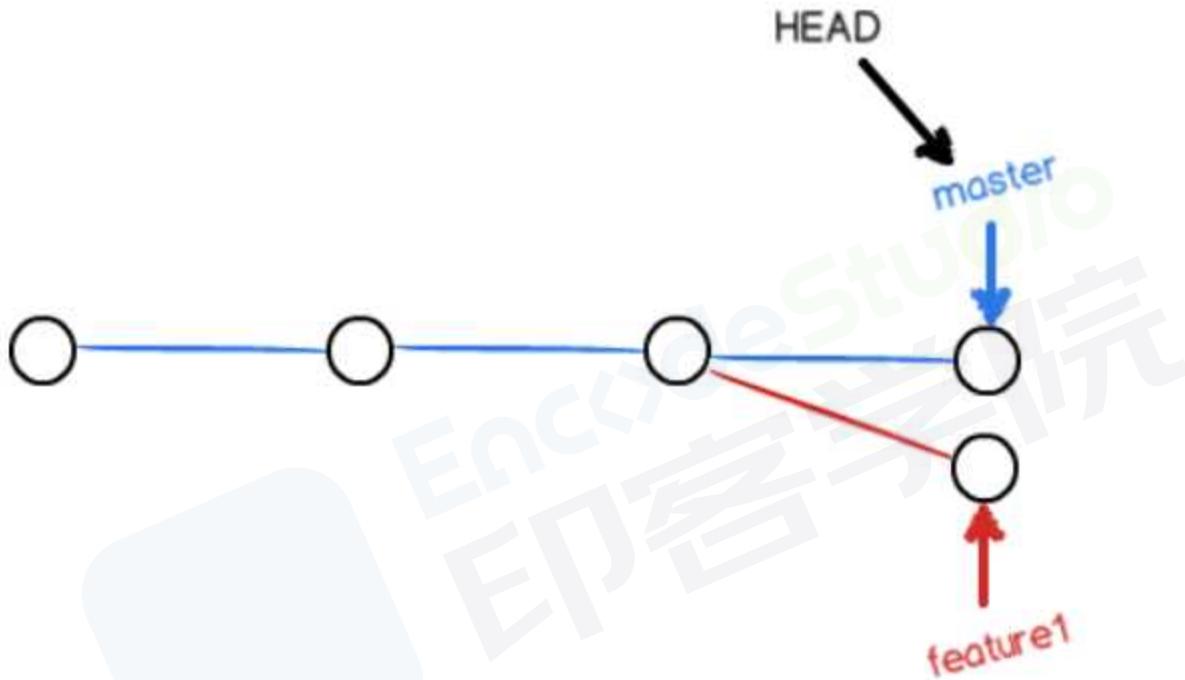
如果当前分支的每一个提交(commit)都已经存在另一个分支里了，git 就会执行一个“快速向前”(fast forward)操作

git 不创建任何新的提交(commit)，只是将当前分支指向合并进来的分支

如果此时切换到 feature1 分支，将文件的内容修改成 feature second commit，然后提交到本地仓库

然后切换到主分支，如果此时在 a.txt 文件再次修改，修改成 master second commit，然后再次提交到本地仓库

此时， master 分支和 feature1 分支各自都分别有新的提交，变成了下图所示：



这种情况下，无法执行快速合并，只能试图把各自的修改合并起来，但这种合并就可能会有冲突

现在通过 `git merge feature1` 进行分支合并，如下所示：

```
PS E:\Users\user\Desktop\git_conflict> git merge feature1
Auto-merging a.txt
CONFLICT (content): Merge conflict in a.txt
Automatic merge failed; fix conflicts and then commit the result.
```

从冲突信息可以看到，`a.txt` 发生冲突，必须手动解决冲突之后再提交

而 `git status` 同样可以告知我们冲突的文件：

```
PS E:\Users\user\Desktop\git_conflict> git status
On branch master
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:   a.txt
```

打开 `a.txt` 文件，可以看到如下内容：

```
You, seconds ago | 1 author (You) | 采用当前更改 | 采用传入的更改 | 保留双方更改 | 比较变更
<<<<< HEAD (当前更改)
master second commit
=====
feature1 second commit
>>>>> feature1 (传入的更改)
```

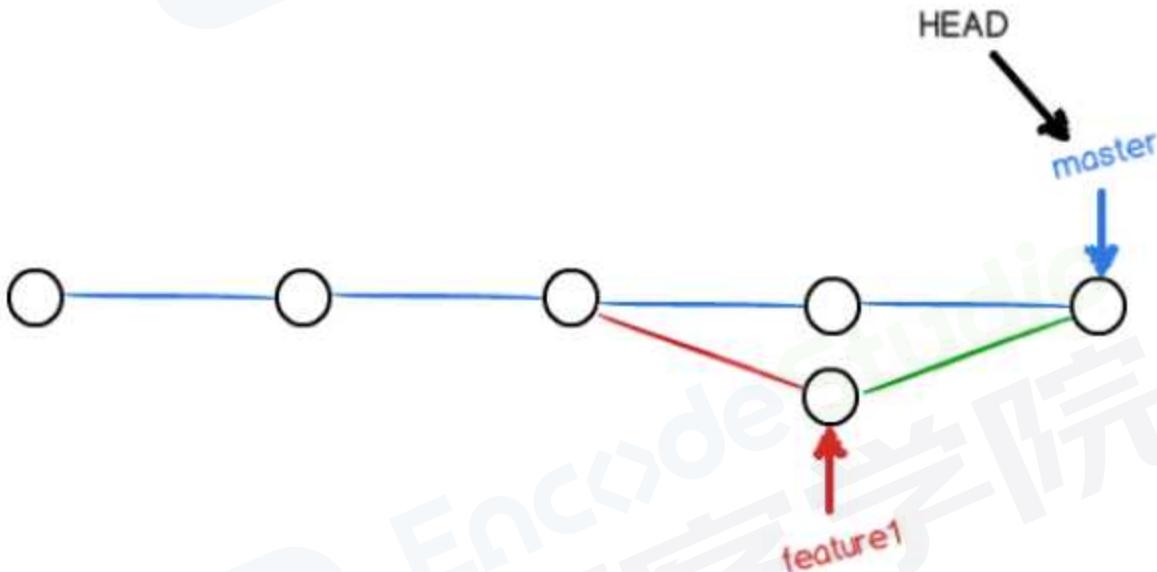
git 用 <<<<< , ====== , >>>>> 标记出不同分支的内容:

- <<<<< 和 ====== 之间的区域就是当前更改的内容
- ====== 和 >>>>> 之间的区域就是传入进来更改的内容

现在要做的事情就是将冲突的内容进行更改，对每个文件使用 git add 命令来将其标记为冲突已解决。一旦暂存这些原本有冲突的文件，Git 就会将它们标记为冲突已解决然后再提交：

- git add a.txt
- git commit -m "conflict fixed"

此时 master 分支和 feature1 分支变成了下图所示：



使用 git log 命令可以看到合并的信息：

```
PS E:\Users\user\Desktop\git_conflict> git log --graph --pretty=oneline --abbrev-commit
* 01eface (HEAD -> master) conflict fixed
|\ 
* 9f31dcc (feature1) feature1 second commit
* e8f60e8 master second commit
* dfc041b master first commit
|
* d401a65 feature1 second commit
* 6346f1e (feature1) master second commit
* 726fd15 (dev) dev commit
* 4d2295c first commit
```

5.3. 总结

当 Git 无法自动合并分支时，就必须首先解决冲突，解决冲突后，再提交，合并完成

解决冲突是把 Git 合并失败的文件手动编辑为我们希望的内容，再提交

6. 说说Git中 fork, clone, branch这三个概念，有什么区别？



6.1. 是什么

6.1.1. fork

fork，英语翻译过来就是叉子，动词形式则是分叉，如下图，从左到右，一条直线变成多条直线



转到 git 仓库中， fork 则可以代表分叉、克隆出一个（仓库的）新拷贝



包含了原来的仓库（即upstream repository，上游仓库）所有内容，如分支、Tag、提交
如果想将你的修改合并到原项目中时，可以通过的 Pull Request 把你的提交贡献回 原仓库

6.1.2. clone

`clone`，译为克隆，它的作用是将文件从远程代码仓下载到本地，从而形成一个本地代码仓
执行 `clone` 命令后，会在当前目录下创建一个名为 `xxx` 的目录，并在这个目录下初始化一个 `.git` 文件夹，然后从中读取最新版本的文件的拷贝

默认配置下远程 `Git` 仓库中的每一个文件的每一个版本都将被拉取下来

6.1.3. branch

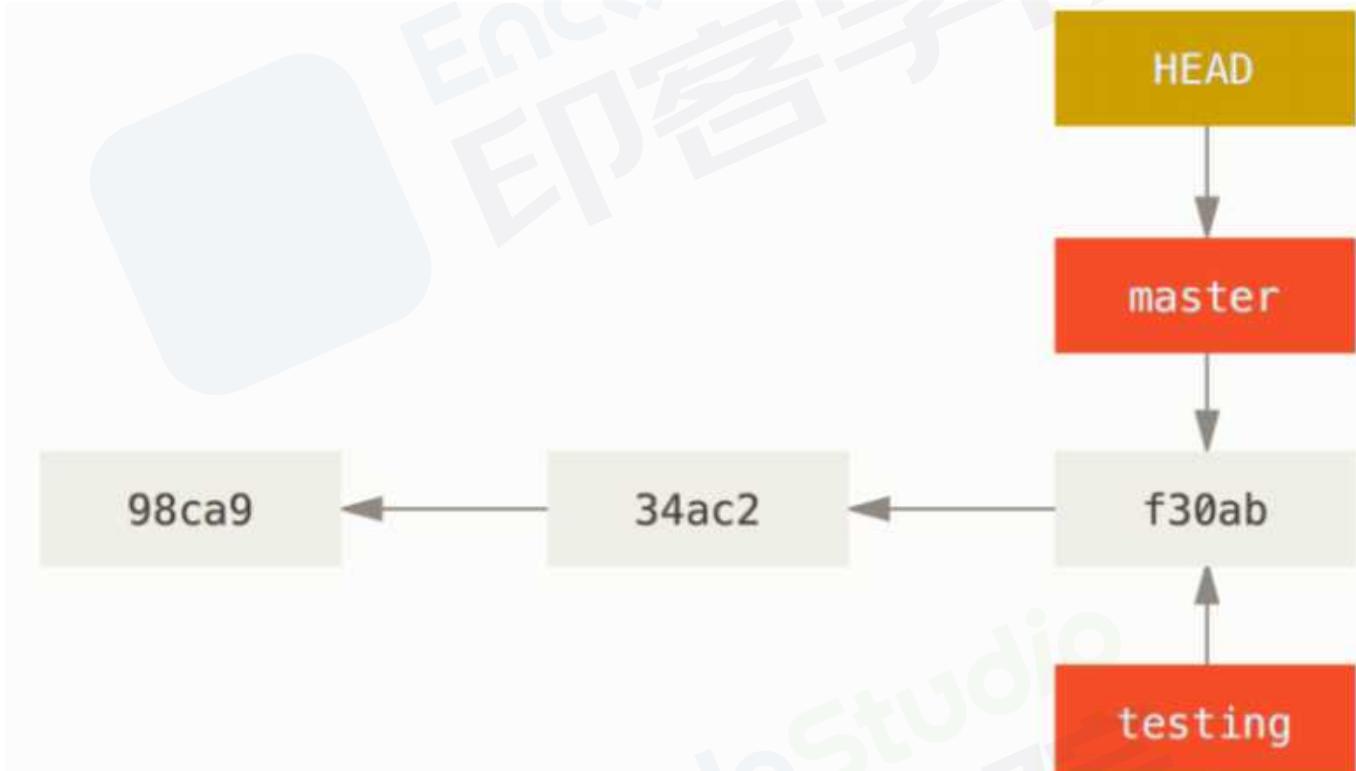
`branch`，译为分支，其作用简单而言就是开启另一个分支，使用分支意味着你可以把你的工作从开发主线上分离开来，以免影响开发主线

`Git` 处理分支的方式十分轻量，创建新分支这一操作几乎能在瞬间完成，并且在不同分支之间的切换操作也是一样便捷

在我们开发中，默认只有一条 `master` 分支，如下图所示：



通过 `git branch` 可以创建一个分支，但并不会自动切换到新分支中去



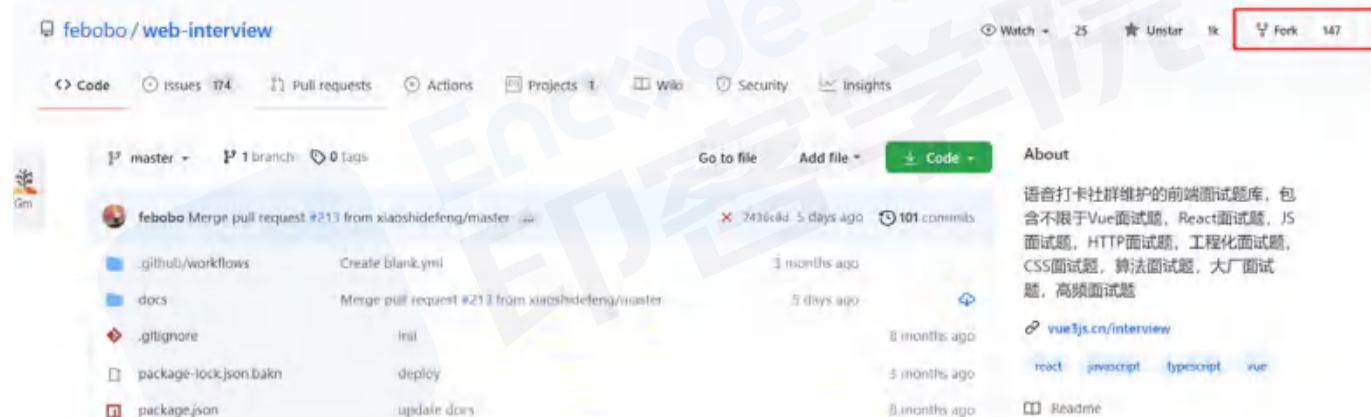
通过 `git checkout` 可以切换到另一个 `testing` 分支



6.2. 如何使用

6.2.1. fork

当你在 `github` 发现感兴趣开源项目的时候，可以通过点击 `github` 仓库中右上角 `fork` 标识的按钮，如下图：

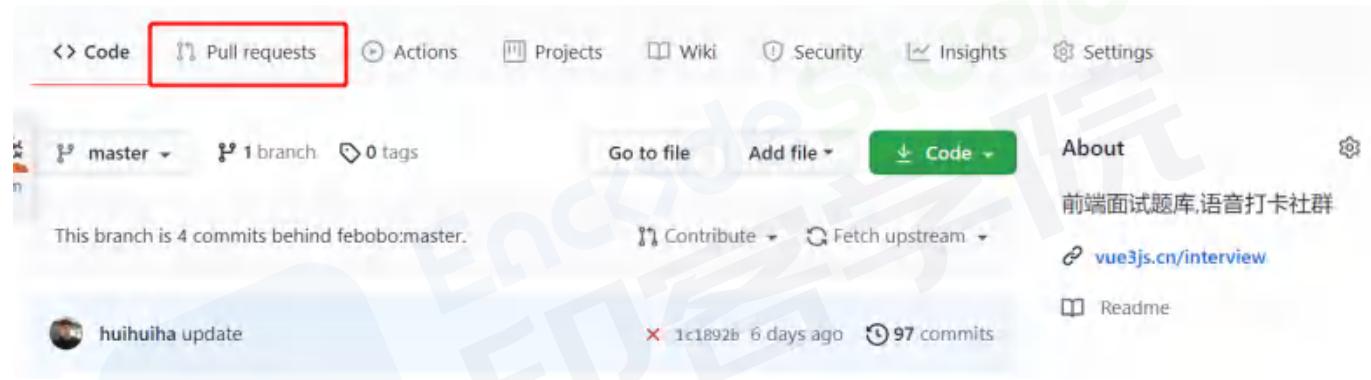


点击这个操作后会将这个仓库的文件、提交历史、issues和其余东西的仓库复制到自己的 `github` 仓库中，而你本地仓库是不会存在任何更改

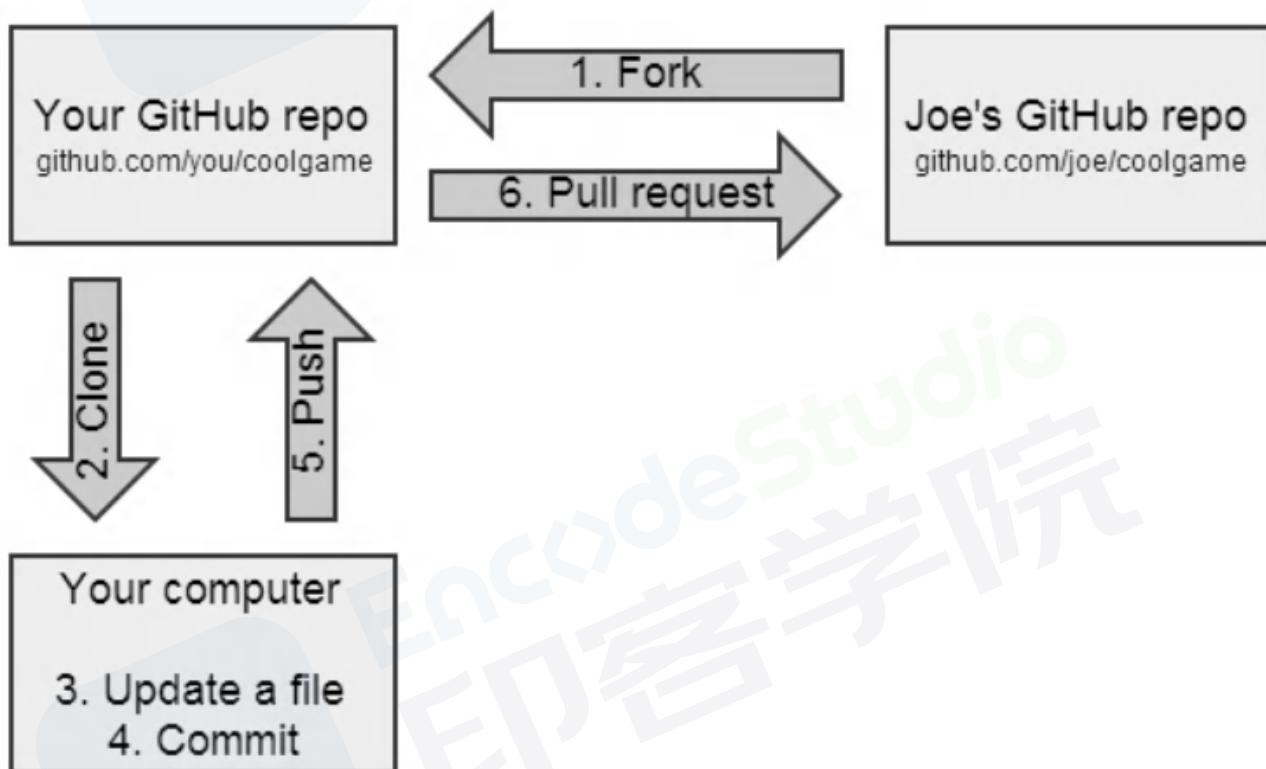
然后你就可以通过 `git clone` 对你这个复制的远程仓库进行克隆

后续更改任何东西都可以在本地完成，如 `git add`、`git commit` 一系列的操作，然后通过 `push` 命令推到自己的远程仓库

如果希望对方接受你的修改，可以通过发送 `pull requests` 给对方，如果对方接受。则会将你的修改内容更新到仓库中



整体流程如下图



6.2.2. clone

在 `github` 中，开源项目右侧存在 `code` 按钮，点击后则会显示开源项目 `url` 信息，如下图所示：

This screenshot shows a GitHub repository page for a user named 'huihuiha'. The repository is named 'web-interview'. At the top, it says 'This branch is 4 commits behind febobo:master.' Below this, there's a list of files and folders: '.github/workflows', 'docs', '.gitignore', and 'package-lock.json'. On the right side, there are several actions: 'Clone' (with HTTPS, SSH, and GitHub CLI options), 'Open with GitHub Desktop', and 'Download ZIP'. A timestamp at the bottom right indicates the page was last updated '2 months ago'.

通过 `git clone xxx` 则能完成远程项目的下载

6.2.3. branch

可通过 `git branch` 进行查看当前的分支状态，

如果给了 `--list`，或者没有非选项参数，现有的分支将被列出；当前的分支将以绿色突出显示，并标有星号

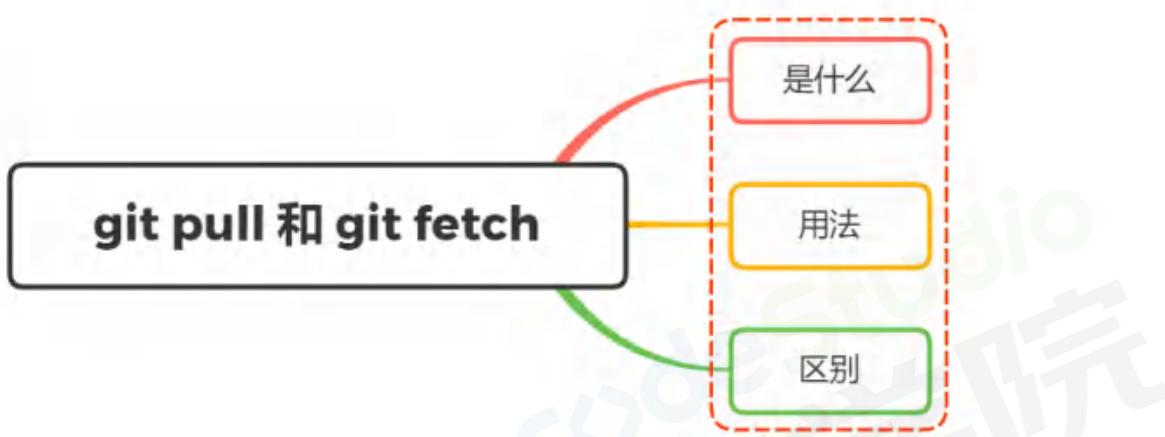
以及通过 `git branch` 创建一个新的分支出来

6.3. 区别

其三者区别如下：

- fork 只能对代码仓进行操作，且 fork 不属于 git 的命令，通常用于代码仓托管平台的一种“操作”
- clone 是 git 的一种命令，它的作用是将文件从远程代码仓下载到本地，从而形成一个本地代码仓
- branch 特征与 fork 很类似，fork 得到的是一个新的、自己的代码仓，而 branch 得到的是一个代码仓的一个新分支

7. 说说对git pull 和 git fetch 的理解？有什么区别？

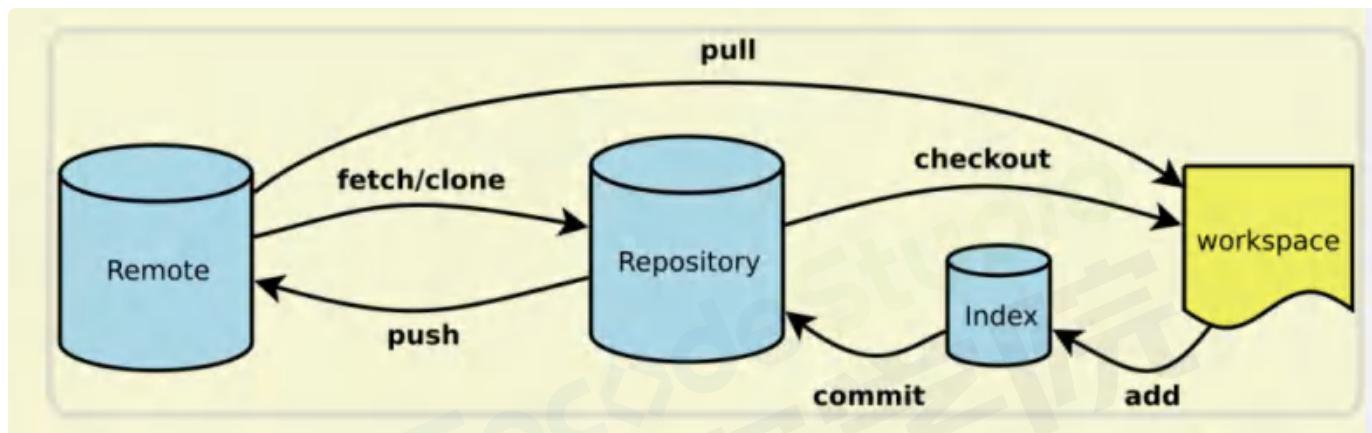


7.1. 是什么

先回顾两个命令的定义

- git fetch 命令用于从另一个存储库下载对象和引用
- git pull 命令用于从另一个存储库或本地分支获取并集成(整合)

再来看一次 `git` 的工作流程图, 如下所示:



可以看到, `git fetch` 是将远程主机的最新内容拉到本地, 用户在检查了以后决定是否合并到工作本机分支中

而 `git pull` 则是将远程主机的最新内容拉下来后直接合并, 即: `git pull = git fetch + git merge`, 这样可能会产生冲突, 需要手动解决

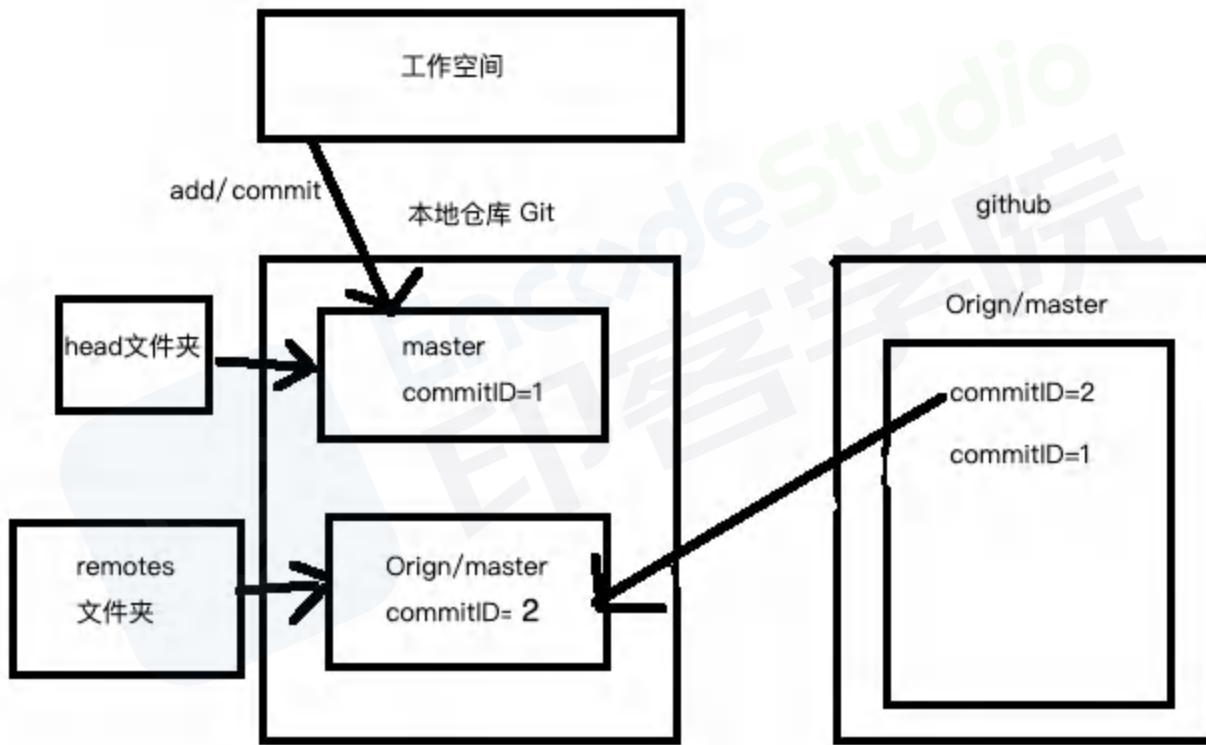
在我们本地的 `git` 文件中对应也存储了 `git` 本地仓库分支的 `commit ID` 和跟踪的远程分支的 `commit ID`, 对应文件如下:

- `.git/refs/head/[本地分支]`
- `.git/refs/remotes/[正在跟踪的分支]`

使用 `git fetch` 更新代码, 本地的库中 `master` 的 `commitID` 不变

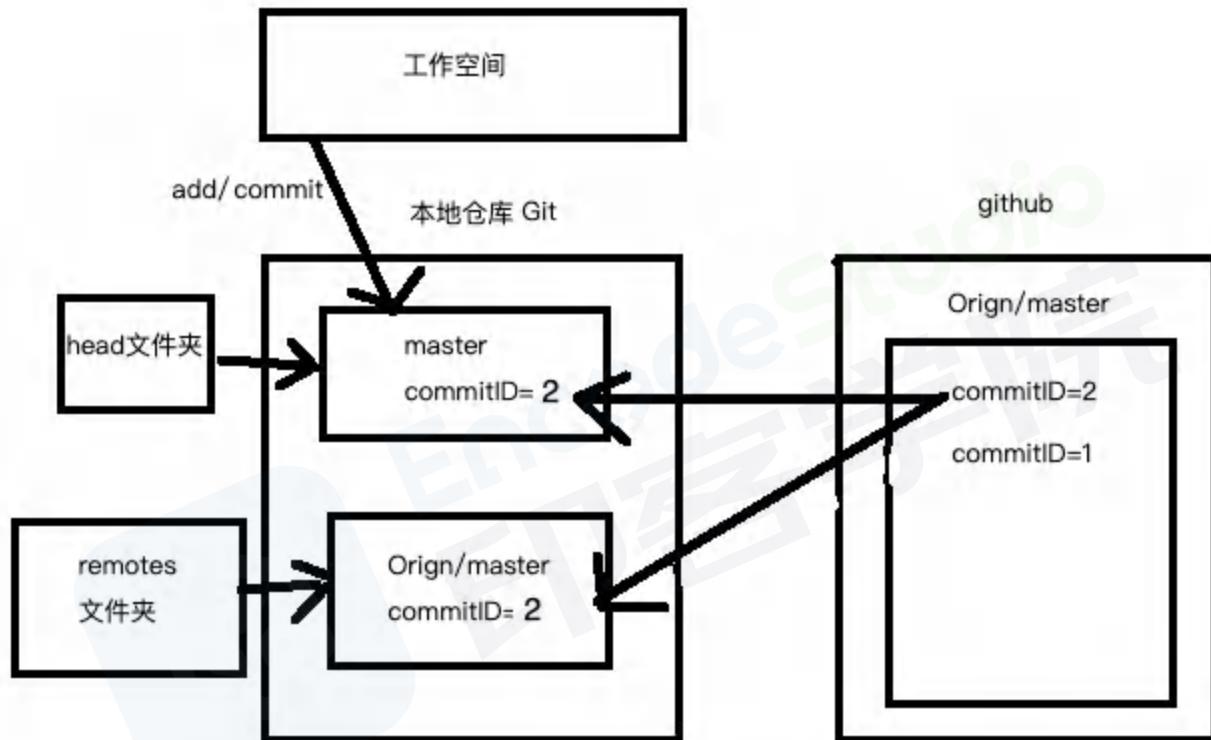
但是与 `git` 上面关联的那个 `origin/master` 的 `commit ID` 发生改变

这时候本地相当于存储了两个代码的版本号，我们还要通过 `merge` 去合并这两个不同的代码版本



也就是 `fetch` 的时候本地的 `master` 没有变化，但是与远程仓关联的那个版本号被更新了，接下来就是在本地 `merge` 合并这两个版本号的代码

相比之下，使用 `git pull` 就更加简单粗暴，会将本地的代码更新至远程仓库里面最新的代码版本，如下图：



7.2. 用法

一般远端仓库里有新的内容更新，当我们需要把新内容下载的时候，就使用到 `git pull` 或者 `git fetch` 命令

7.2.1. fetch

用法如下：

```
1 git fetch <远程主机名> <远程分支名>:<本地分支名>
```

例如从远程的 `origin` 仓库的 `master` 分支下载代码到本地并新建一个 `temp` 分支

```
1 git fetch origin master:temp
```

如果上述没有冒号，则表示将远程 `origin` 仓库的 `master` 分支拉取下来到本地当前分支

这里 `git fetch` 不会进行合并，执行后需要手动执行 `git merge` 合并，如下：

```
Plain Text | 复制代码  
1 git merge temp
```

7.2.2. pull

两者的用法十分相似，`pull` 用法如下：

```
Plain Text | 复制代码  
1 git pull <远程主机名> <远程分支名>:<本地分支名>
```

例如将远程主机 `origin` 的 `master` 分支拉取过来，与本地的 `branchtest` 分支合并，命令如下：

```
Plain Text | 复制代码  
1 git pull origin master:branchtest
```

同样如果上述没有冒号，则表示将远程 `origin` 仓库的 `master` 分支拉取下来与本地当前分支合并

7.3. 区别

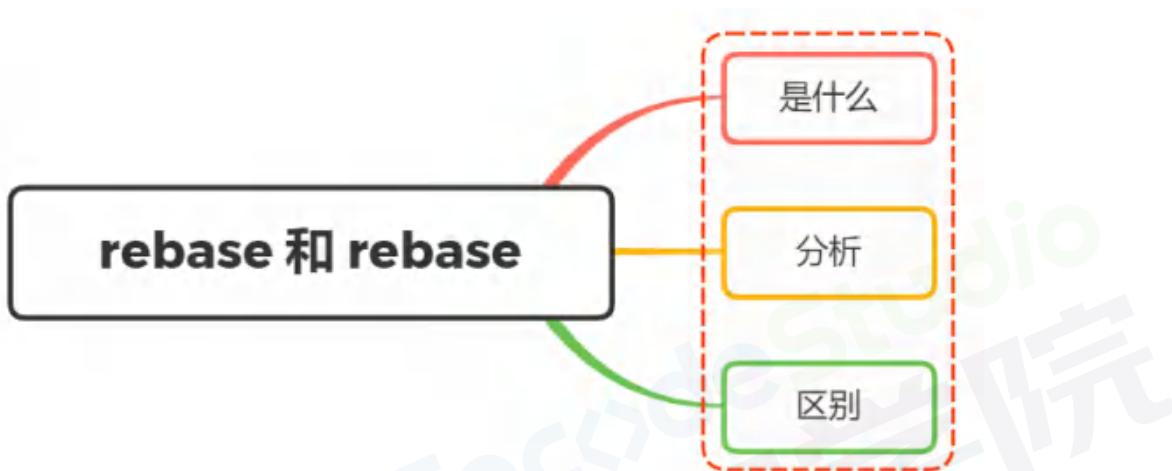
相同点：

- 在作用上他们的功能是大致相同的，都是起到了更新代码的作用

不同点：

- `git pull` 是相当于从远程仓库获取最新版本，然后再与本地分支 `merge`，即 `git pull = git fetch + git merge`
- 相比起来，`git fetch` 更安全也更符合实际要求，在 `merge` 前，我们可以查看更新情况，根据实际情况再决定是否合并

8. 说说你对 `git rebase` 和 `git merge` 的理解？区别？



8.1. 是什么

在使用 `git` 进行版本管理的项目中，当完成一个特性的开发并将其合并到 `master` 分支时，会有两种方式：

- `git merge`
- `git rebase`

`git rebase` 与 `git merge` 都有相同的作用，都是将一个分支的提交合并到另一分支上，但是在原理上却不相同

用法上两者也十分的简单：

8.1.1. git merge

将当前分支合并到指定分支，命令用法如下：

```
Plain Text | 复制代码  
1 git merge xxx
```

8.1.2. git rebase

将当前分支移植到指定分支或指定 `commit` 之上，用法如下：

```
Plain Text | 复制代码  
1 git rebase -i <commit>
```

常见的参数有 `--continue`，用于解决冲突之后，继续执行 `rebase`

```
Plain Text | 复制代码  
1 git rebase --continue
```

8.2. 二、分析

8.2.1. git merge

通过 `git merge` 将当前分支与 `xxx` 分支合并，产生的新的 `commit` 对象有两个父节点

如果“指定分支”本身是当前分支的一个直接子节点，则会产生快照合并

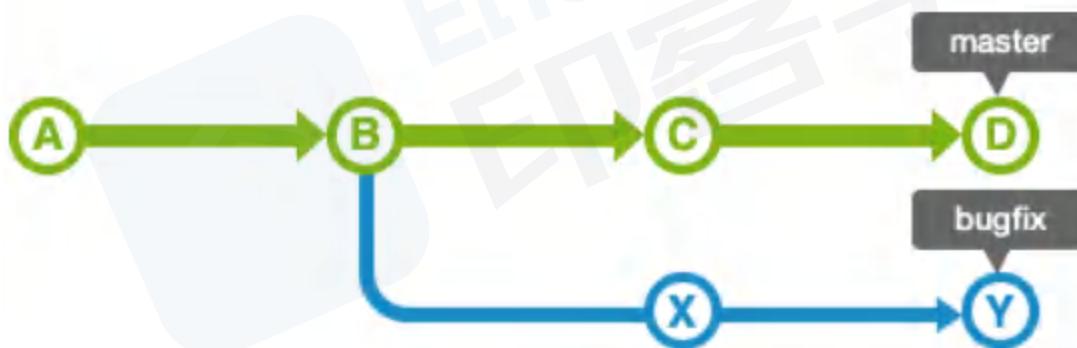
举个例子，`bugfix` 分支是从 `master` 分支分叉出来的，如下所示：



合并 `bugfix` 分支到 `master` 分支时，如果 `master` 分支的状态没有被更改过，即 `bugfix` 分支的历史记录包含 `master` 分支所有的历史记录

所以通过把 `master` 分支的位置移动到 `bugfix` 的最新分支上，就完成合并

如果 `master` 分支的历史记录在创建 `bugfix` 分支后又有新的提交，如下情况：



这时候使用 `git merge` 的时候，会生成一个新的提交，并且 `master` 分支的 `HEAD` 会移动到新的分支上，如下：



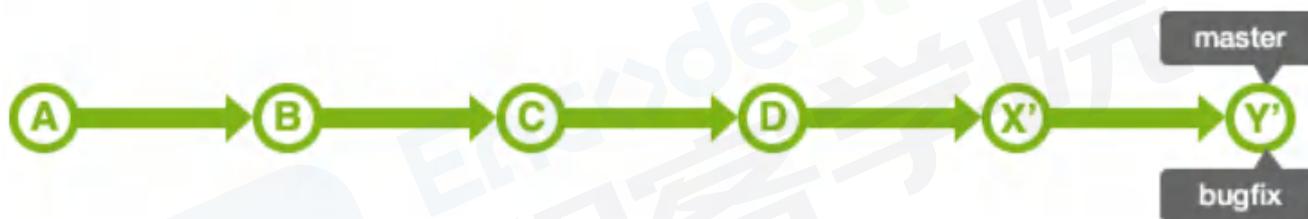
从上面可以看到，会把两个分支的最新快照以及二者最近的共同祖先进行三方合并，合并的结果是生成一个新的快照

8.2.2. git rebase

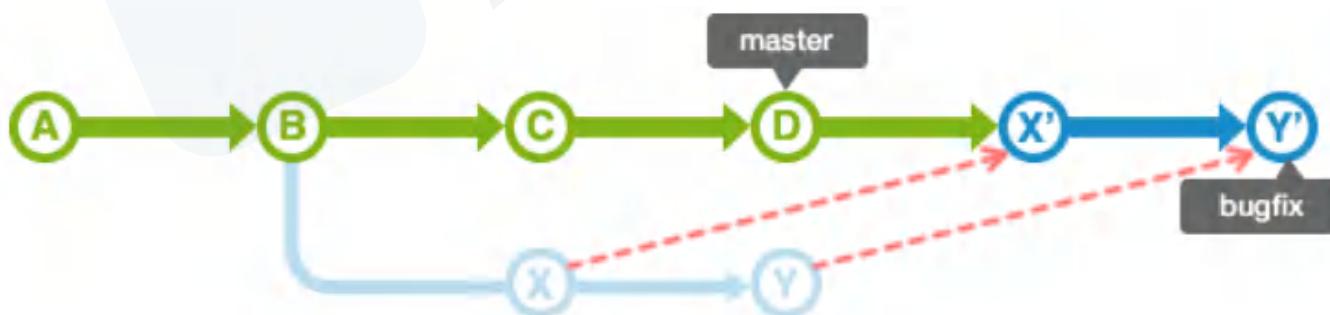
同样，`master` 分支的历史记录在创建 `bugfix` 分支后又有新的提交，如下情况：



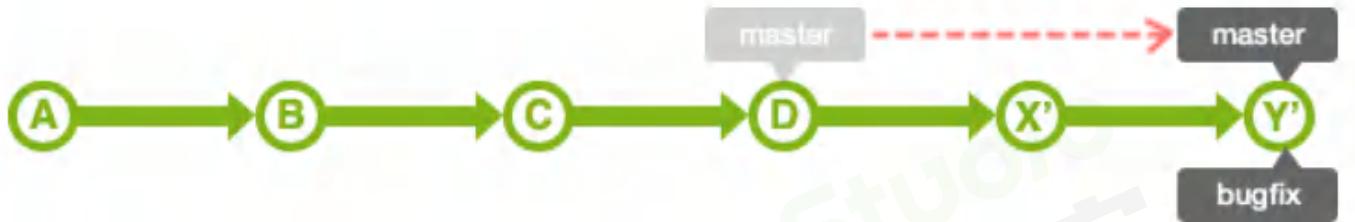
通过 `git rebase`，会变成如下情况：



在移交过程中，如果发生冲突，需要修改各自的冲突，如下：



`rebase` 之后，`master` 的 `HEAD` 位置不变。因此，要合并 `master` 分支和 `bugfix` 分支



从上面可以看到，`rebase` 会找到不同的分支的最近共同祖先，如上图的 B

然后对比当前分支相对于该祖先的历次提交，提取相应的修改并存为临时文件（老的提交 X 和 Y 也没有被销毁，只是简单地不能再被访问或者使用）

然后将当前分支指向目标最新位置 D，然后将之前另存为临时文件的修改依序应用

8.3. 区别

从上面可以看到，`merge` 和 `rebase` 都是合并历史记录，但是各自特性不同：

8.3.1. merge

通过 `merge` 合并分支会新增一个 `merge commit`，然后将两个分支的历史联系起来

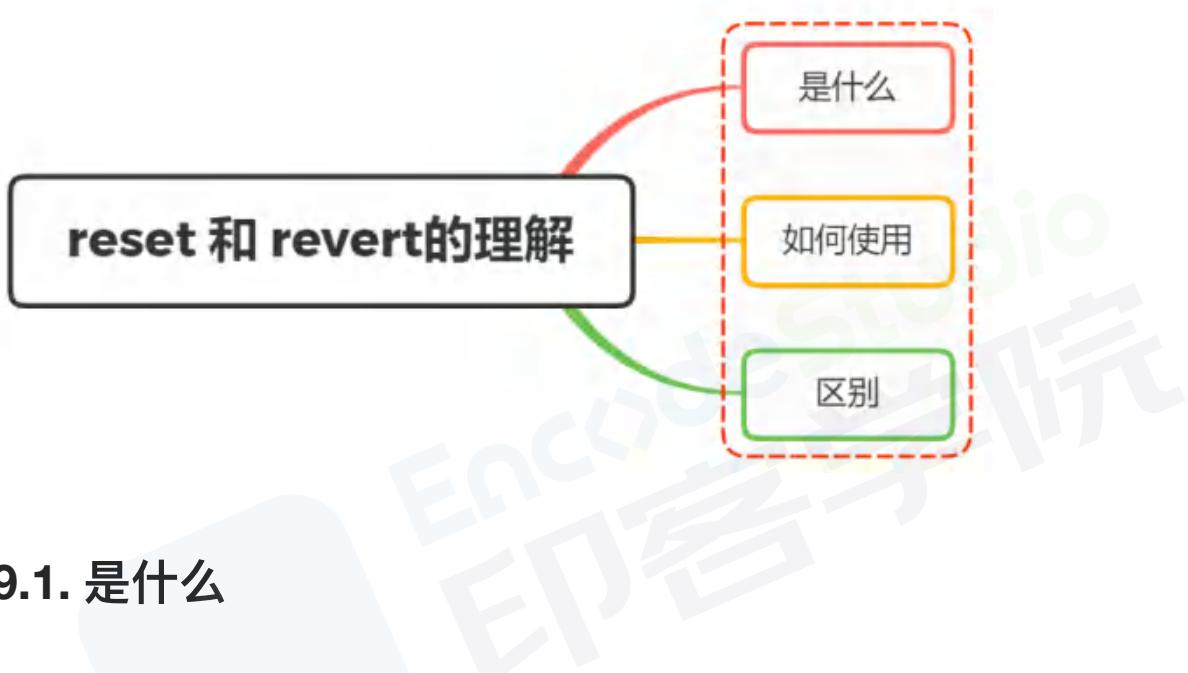
其实是一种非破坏性的操作，对现有分支不会以任何方式被更改，但是会导致历史记录相对复杂

8.3.2. rebase

`rebase` 会将整个分支移动到另一个分支上，有效地整合了所有分支上的提交

主要的好处是历史记录更加清晰，是在原有提交的基础上将差异内容反映进去，消除了 `git merge` 所需的不必要的合并提交

9. 说说你对git reset 和 git revert 的理解？区别？

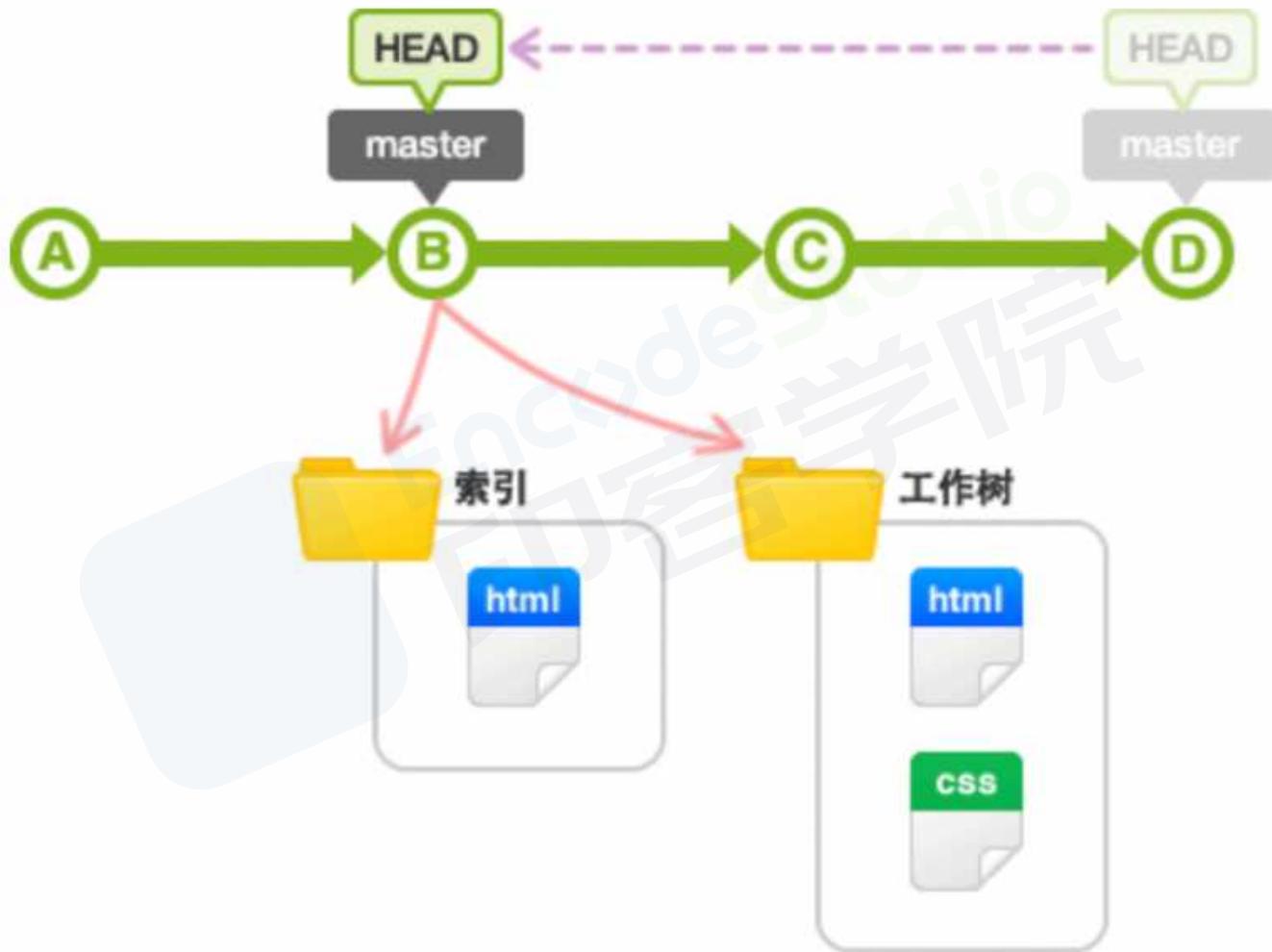


9.1. 是什么

9.1.1. git reset

`reset` 用于回退版本，可以遗弃不再使用的提交

执行遗弃时，需要根据影响的范围而指定不同的参数，可以指定是否复原索引或工作树内容



9.1.2. git revert

在当前提交后面，新增一次提交，抵消掉上一次提交导致的所有变化，不会改变过去的历史，主要是用于安全地取消过去发布的提交



9.2. 如何用

9.2.1. git reset

当没有指定 `ID` 的时候，默认使用 `HEAD`，如果指定 `ID`，那么就是基于指向 `ID` 去变动暂存区或工作区的内容

```
1 // 没有指定ID，暂存区的内容会被当前ID版本号的内容覆盖，工作区不变
2 git reset
3
4 // 指定ID，暂存区的内容会被指定ID版本号的内容覆盖，工作区不变
5 git reset <ID>
```

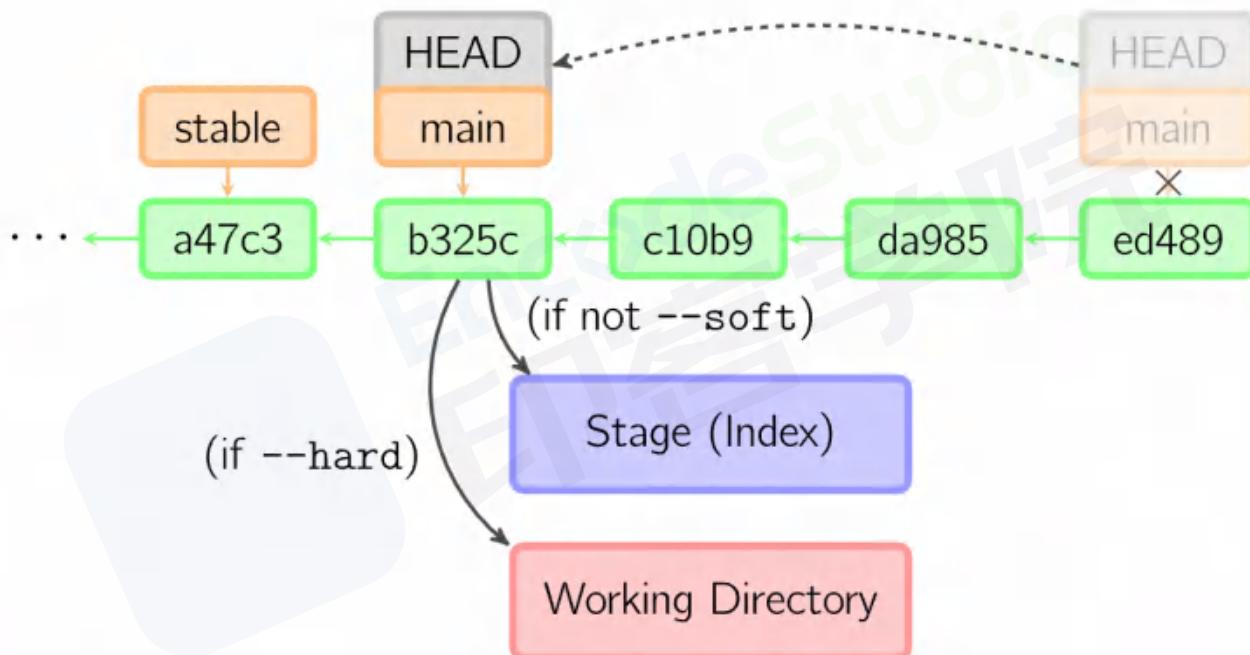
日志 `ID` 可以通过查询，可以 `git log` 进行查询，如下：

```
1 commit a7700083ace1204ccdff9f71631fb34c9913f7c5 (HEAD -> master)
2 Author: linguanghui <linguanghui@baidu.com>
3 Date:   Tue Aug 17 22:34:40 2021 +0800
4
5     second commit
6
7 commit e31118663ce66717edd8a179688a7f3dde5a9393
8 Author: linguanghui <linguanghui@baidu.com>
9 Date:   Tue Aug 17 22:20:01 2021 +0800
10
11    first commit
```

常见命令如下

- `--mixed` (默认)：默认的时候，只有暂存区变化
- `--hard`参数：如果使用 `--hard` 参数，那么工作区也会变化
- `--soft`：如果使用 `--soft` 参数，那么暂存区和工作区都不会变化

```
git reset HEAD~3
```



9.2.2. git revert

跟 `git reset` 用法基本一致，`git revert` 撤销某次操作，此次操作之前和之后的 `commit` 和 `history` 都会保留，并且把这次撤销，作为一次最新的提交，如下：

```
1 git revert <commit_id>
```

如果撤销前一个版本，可以通过如下命令：

```
1 git revert HEAD
```

撤销前前一次，如下：

```
1 git revert HEAD^
```

9.3. 区别

撤销 (revert) 被设计为撤销公开的提交 (比如已经push) 的安全方式, `git reset` 被设计为重设本地更改

因为两个命令的目的不同, 它们的实现也不一样: 重设完全地移除了一堆更改, 而撤销保留了原来的更改, 用一个新的提交来实现撤销

两者主要区别如下:

- `git revert` 是用一次新的commit来回滚之前的commit, `git reset` 是直接删除指定的commit
- `git reset` 是把HEAD向后移动了一下, 而`git revert`是HEAD继续前进, 只是新的commit的内容和要 revert的内容正好相反, 能够抵消要被revert的内容
- 在回滚这一操作上看, 效果差不多。但是在日后继续 merge 以前的老版本时有区别

`git revert`是用一次逆向的commit“中和”之前的提交, 因此日后合并老的branch时, 之前提交合并的代码仍然存在, 导致不能够重新合并

但是`git reset`是之间把某些commit在某个branch上删除, 因而和老的branch再次merge时, 这些被回滚的commit应该还会被引入

- 如果回退分支的代码以后还需要的情况则使用 `git revert`, 如果分支是提错了没用的并且不想让别人发现这些错误代码, 则使用 `git reset`

10. 说说你对git stash 的理解? 应用场景?



10.1. 是什么

stash, 译为存放, 在 git 中, 可以理解为保存当前工作进度, 会把暂存区和工作区的改动进行保存, 这些修改会保存在一个栈上

后续你可以在任何时候任何分支重新将某次的修改推出来, 重新应用这些更改的代码

默认情况下, `git stash` 会缓存下列状态的文件:

- 添加到暂存区的修改 (staged changes)
- Git 跟踪的但并未添加到暂存区的修改 (unstaged changes)

但以下状态的文件不会缓存:

- 在工作目录中新的文件 (untracked files)
- 被忽略的文件 (ignored files)

如果想要上述的文件都被缓存, 可以使用 `-u` 或者 `--include-untracked` 可以工作目录新的文件, 使用 `-a` 或者 `--all` 命令可以当前目录下的所有修改

10.2. 如何使用

关于 `git stash` 常见的命令如下:

- `git stash`
- `git stash save`
- `git stash list`
- `git stash pop`
- `git stash apply`
- `git stash show`
- `git stash drop`
- `git stash clear`

10.2.1. git stash

保存当前工作进度, 会把暂存区和工作区的改动保存起来

10.2.2. git stash save

`git stash save` 可以用于存储修改. 并且将 `git` 的工作状态切回到 `HEAD` 也就是上一次合法提交上

如果给定具体的文件路径, `git stash` 只会处理路径下的文件.其他的文件不会被存储, 其存在一些参数:

- `--keep-index` 或者 `-k` 只会存储为加入 git 管理的文件
- `--include-untracked` 为追踪的文件也会被缓存, 当前的工作空间会被恢复为完全清空的状态
- `-a` 或者 `--all` 命令可以当前目录下的所有修改, 包括被 git 忽略的文件

10.2.3. git stash list

显示保存进度的列表。也就意味着, `git stash` 命令可以多次执行, 当多次使用 `git stash` 命令后, 栈里会充满未提交的代码, 如下:

```
linguanghui@macdeMacBook-Pro git-test % git stash list
stash@{0}: WIP on test: 8431f5e first commit
stash@{1}: WIP on test: 8431f5e first commit
```

其中, `stash@{0}`、`stash@{1}` 就是当前 `stash` 的名称

10.2.4. git stash pop

`git stash pop` 从栈中读取最近一次保存的内容, 也就是栈顶的 `stash` 会恢复到工作区

也可以通过 `git stash pop + stash` 名字执行恢复哪个 `stash` 恢复到当前目录

如果从 `stash` 中恢复的内容和当前目录中的内容发生了冲突, 则需要手动修复冲突或者创建新的分支来解决冲突

10.2.5. git stash apply

将堆栈中的内容应用到当前目录, 不同于 `git stash pop`, 该命令不会将内容从堆栈中删除

也就说该命令能够将堆栈的内容多次应用到工作目录中, 适应于多个分支的情况

同样, 可以通过 `git stash apply + stash` 名字执行恢复哪个 `stash` 恢复到当前目录

10.2.6. git stash show

查看堆栈中最新保存的 `stash` 和当前目录的差异

通过使用 `git stash show -p` 查看详细的不同

通过使用 `git stash show stash@{1}` 查看指定的 `stash` 和当前目录差异

```
linguanghui@macdeMacBook-Pro git-test % git stash show  
a.txt | 3 +--  
1 file changed, 2 insertions(+), 1 deletion(-)  
linguanghui@macdeMacBook-Pro git-test % git stash show -p  
diff --git a/a.txt b/a.txt  
index 72943a1..e6d83f1 100644  
--- a/a.txt  
+++ b/a.txt  
@@ -1 +1,2 @@  
-aaa  
+aaa  
+1  
linguanghui@macdeMacBook-Pro git-test % git stash show stash@{1}  
c.txt | 1 +  
1 file changed, 1 insertion(+)
```

10.2.7. git stash drop

`git stash drop` + `stash` 名称表示从堆栈中移除某个指定的stash

10.2.8. git stash clear

删除所有存储的进度

10.3. 应用场景

当你在项目的一部分上已经工作一段时间后，所有东西都进入了混乱的状态，而这时你想要切换到另一个分支或者拉下远端的代码去做一点别的事情

但是你创建一次未完成的代码的 `commit` 提交，这时候就可以使用 `git stash`

例如以下场景：

当你的开发进行到一半，但是代码还不想进行提交，然后需要同步去关联远端代码时。如果你本地的代码和远端代码没有冲突时，可以直接通过 `git pull` 解决

但是如果可能发生冲突怎么办。直接 `git pull` 会拒绝覆盖当前的修改，这时候就可以依次使用下述的命令：

- `git stash`
- `git pull`
- `git stash pop`

或者当你开发到一半，现在要修改别的分支问题的时候，你也可以使用 `git stash` 缓存当前区域的代码

- git stash: 保存开发到一半的代码
- git commit -m '修改问题'
- git stash pop: 将代码追加到最新的提交之后