- 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

master :默认开发分支　　Head :默认开发分支
origin :默认远程版本库　　Head^ :Head 的父提交

创建版本库

```
$ git clone <url>          #克隆远程版本库
$ git init                 #初始化本地版本库
```

修改和提交

```
$ git status               #查看状态
$ git diff                 #查看变更内容
$ git add .                #跟踪所有改动过的文件
$ git add <file>           #跟踪指定的文件
$ git mv <old> <new>       #文件改名
$ git rm <file>            #删除文件
$ git rm --cached <file>   #停止跟踪文件但不删除
$ git commit -m "commit message"
                           #提交所有更新过的文件
$ git commit --amend       #修改最后一次提交
```

查看提交历史

```
$ git log                  #查看提交历史
$ git log -p <file>        #查看指定文件的提交历史
$ git blame <file>         #以列表方式查看指定文件
                           的提交历史
```

撤消

```
$ git reset --hard HEAD    #撤消工作目录中所有未提交
                           文件的修改内容
$ git checkout HEAD <file> #撤消指定的未提交文件的修
                           改内容
$ git revert <commit>      #撤消指定的提交
```

分支与标签

```
$ git branch               #显示所有本地分支
$ git checkout <branch/tag> #切换到指定分支或标签
$ git branch <new-branch>  #创建新分支
$ git branch -d <branch>   #删除本地分支
$ git tag                  #列出所有本地标签
$ git tag <tagname>        #基于最新提交创建标签
$ git tag -d <tagname>     #删除标签
```

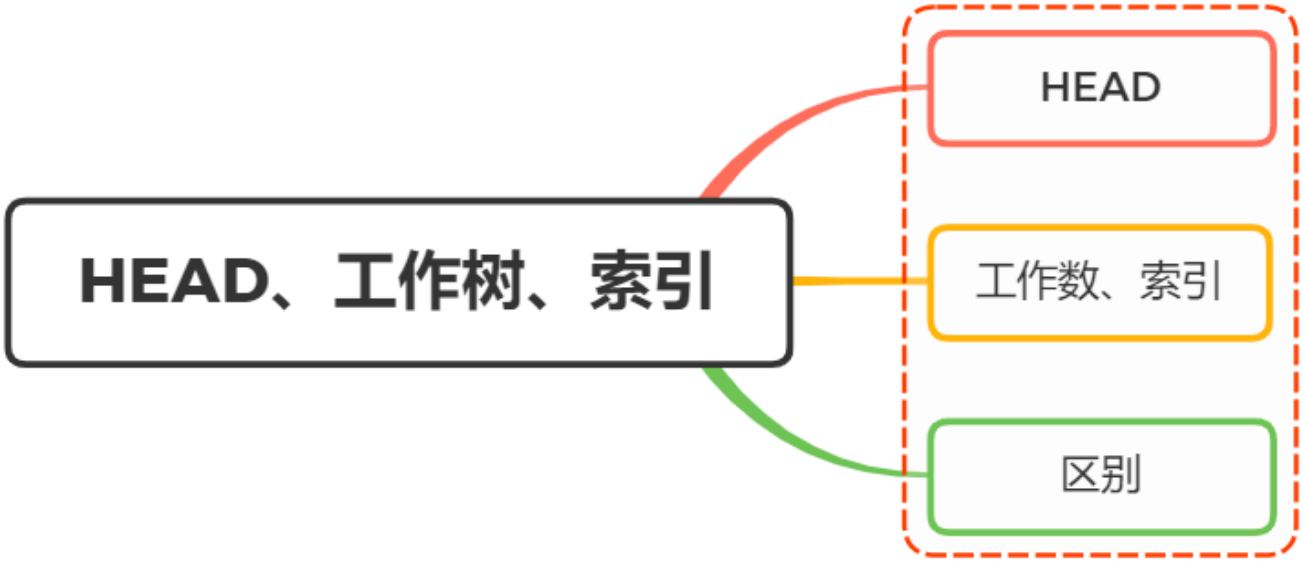
合并与衍合

```
$ git merge <branch>       #合并指定分支到当前分支
$ git rebase <branch>      #衍合指定分支到当前分支
```

远程操作

```
$ git remote -v            #查看远程版本库信息
$ git remote show <remote> #查看指定远程版本库信息
$ git remote add <remote> <url>
                           #添加远程版本库
$ git fetch <remote>       #从远程库获取代码
$ git pull <remote> <branch> #下载代码及快速合并
$ git push <remote> <branch> #上传代码及快速合并
$ git push <remote> :<branch/tag-name>
                           #删除远程分支或标签
$ git push --tags          #上传所有标签
```

4. 说说Git 中 HEAD、工作树和索引之间的区别?

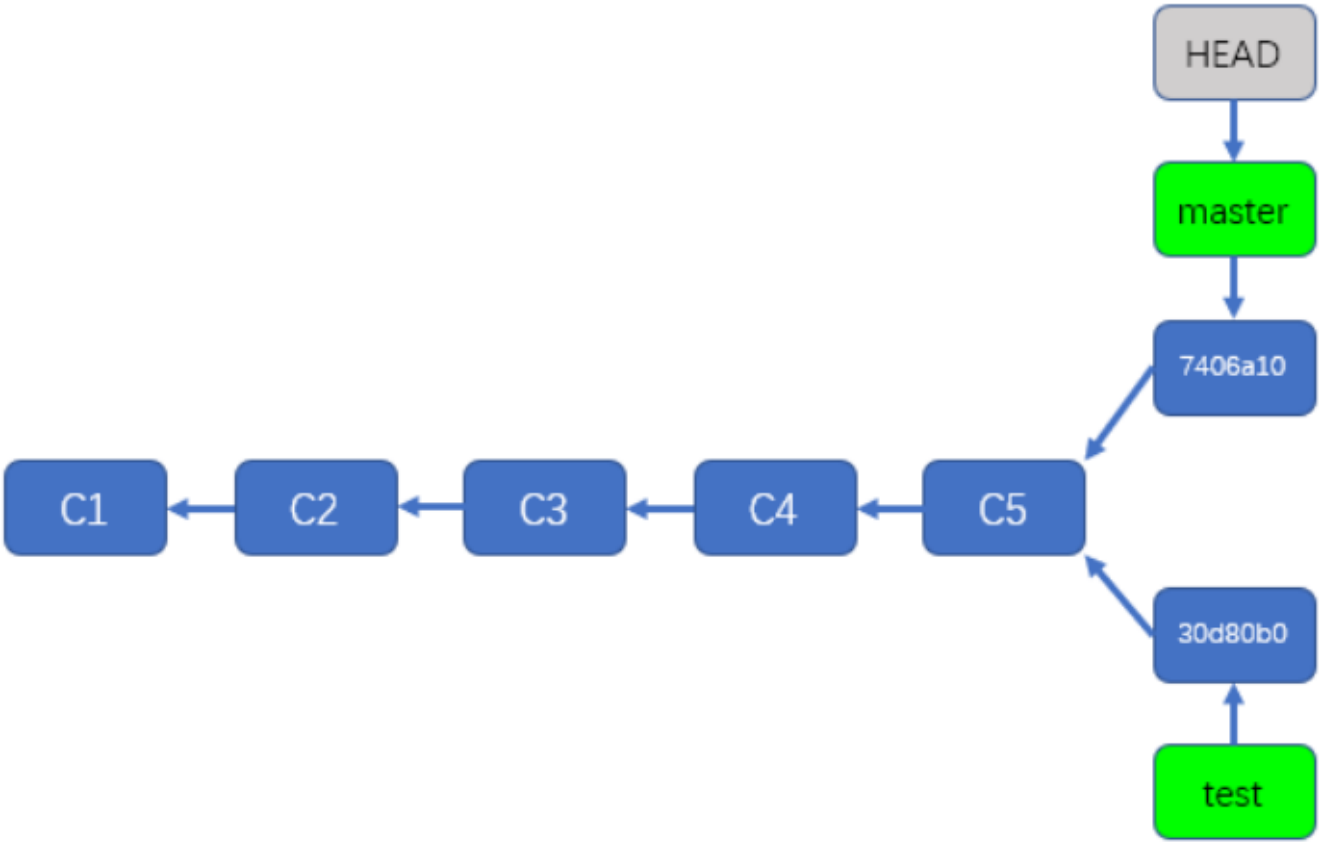


4.1. HEAD

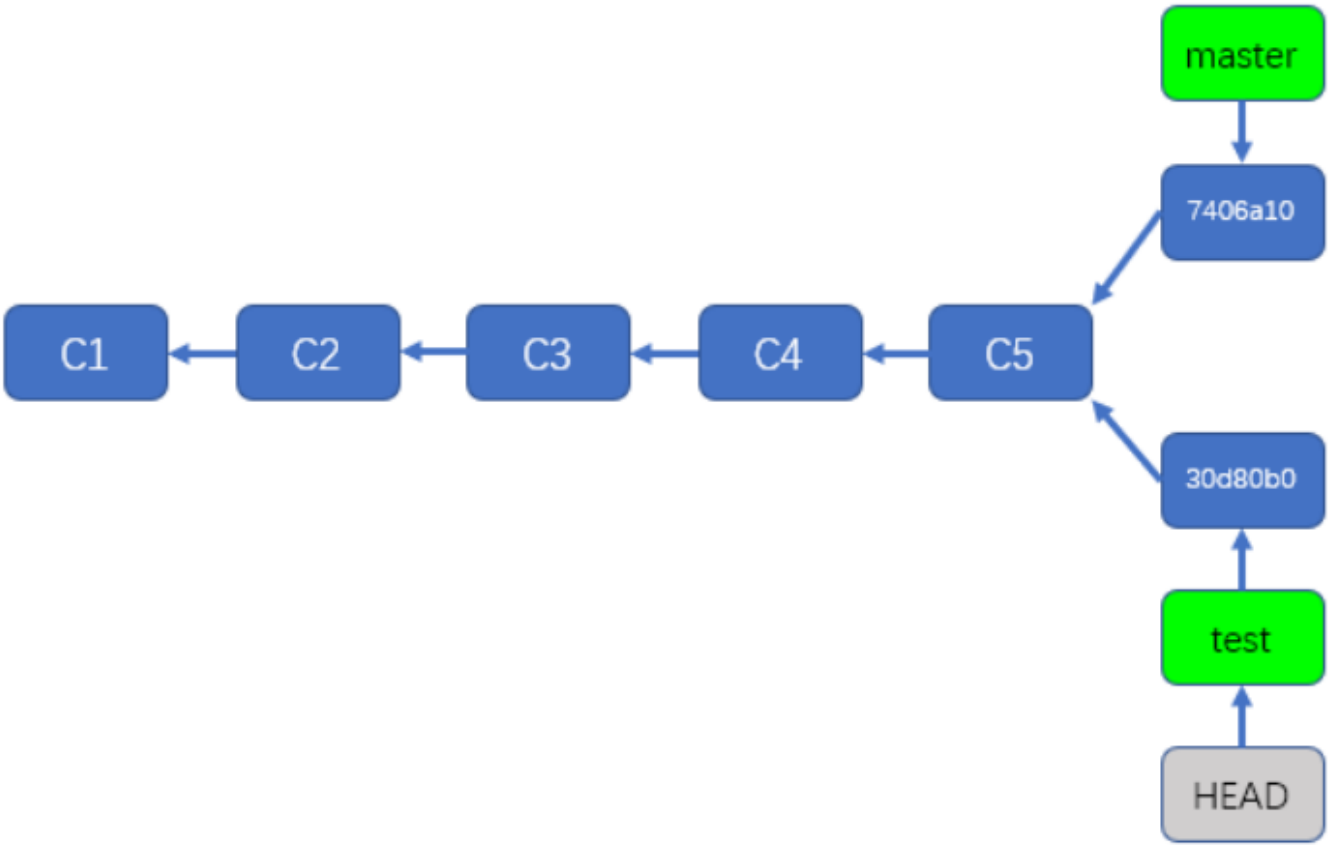
在 `git` 中，可以存在很多分支，其本质上是一个指向 `commit` 对象的可变指针，而 `Head` 是一个特别的指针，是一个指向你正在工作中的本地分支的指针

简单来讲，就是你现在在哪儿，HEAD 就指向哪儿

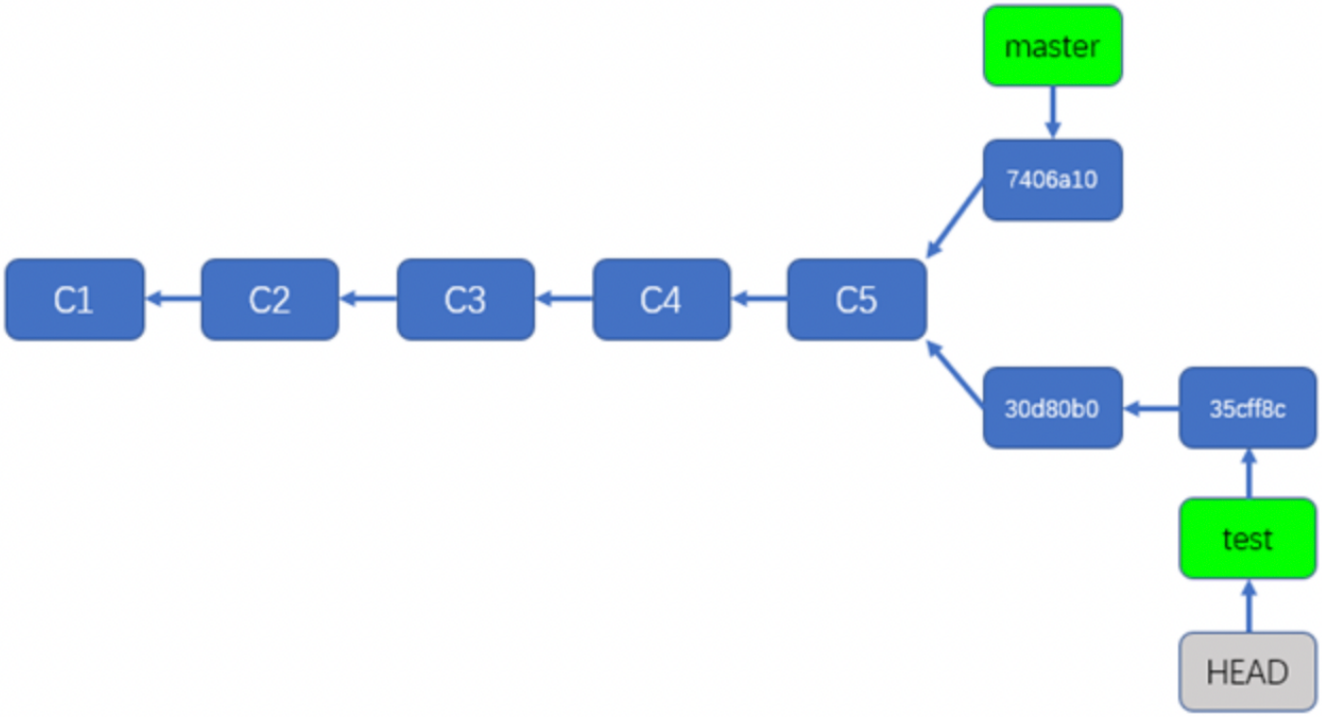
例如当前我们处于 `master` 分支，所以 `HEAD` 这个指针指向了 `master` 分支指针



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

```
$ cat .git/HEAD
ref: refs/heads/master

$ cat .git/refs/heads/master
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

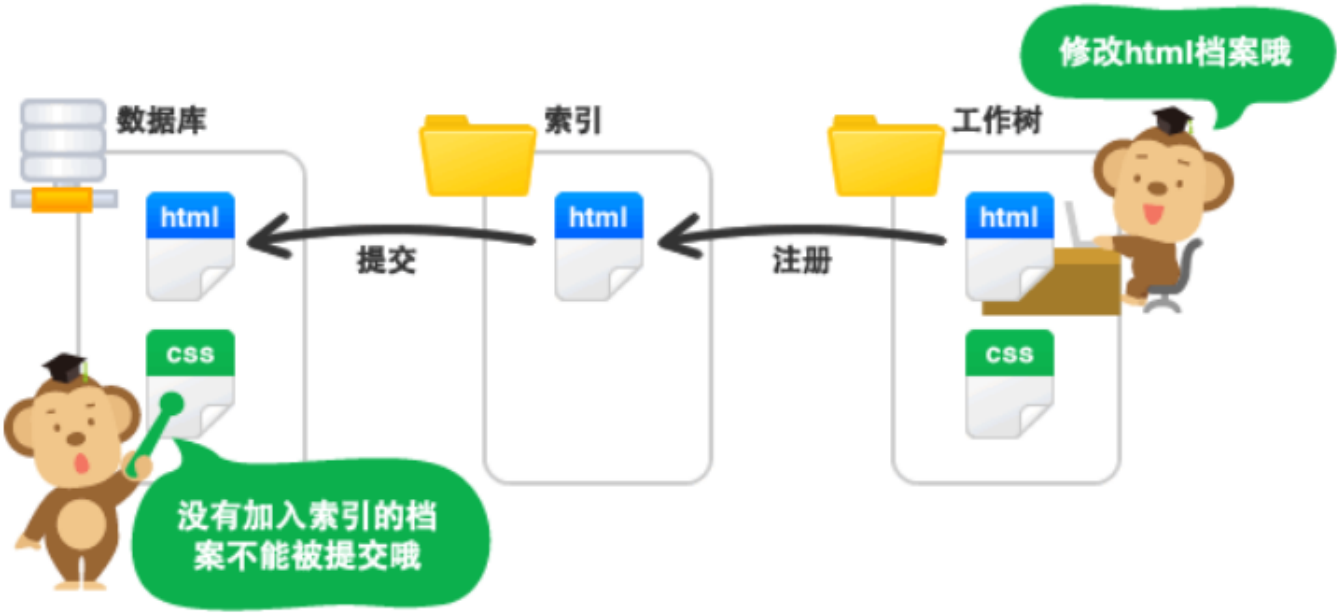
所以，当我们切换分支的时候， `HEAD` 指针通常指向我们所在的分支，当我们在某个分支上创建新的提交时，分支指针总是会指向当前分支的最新提交

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。

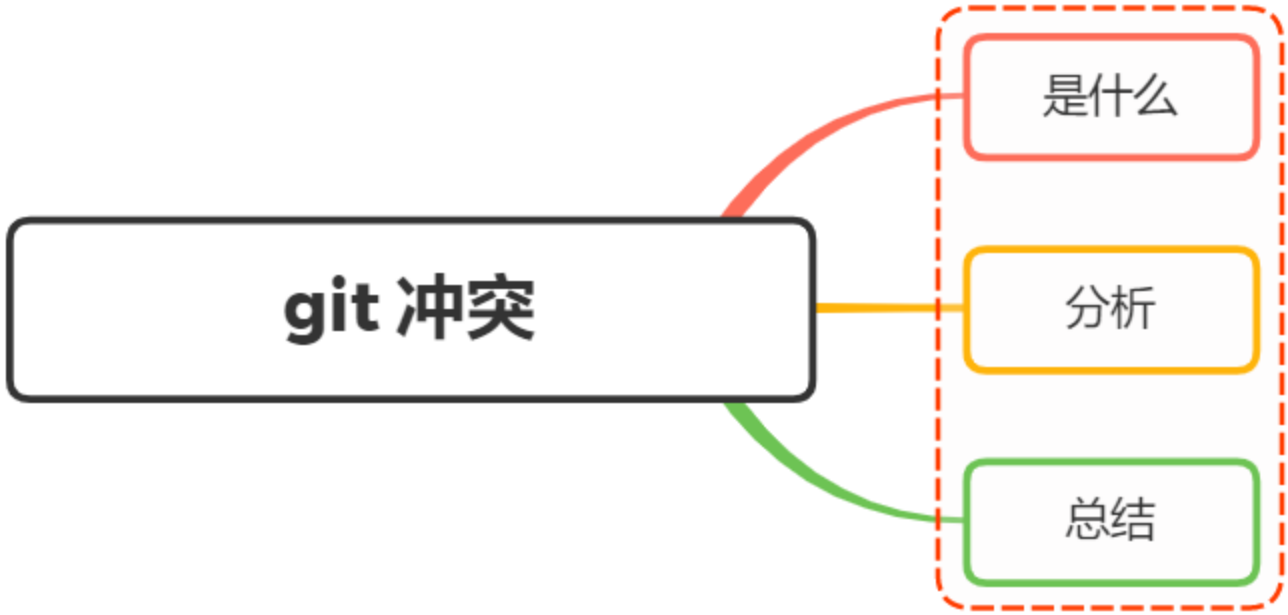
所以，凭借中间的索引，可以避免工作树中不必要的文件提交，还可以将文件修改内容的一部分加入索引区域并提交

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，从本质上来讲，都是 `merge` 和 `patch` （应用补丁）时产生冲突

5.2. 分析

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

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 `featurel1` 分支，并进行切换，如下：

```
Plain Text | 复制代码
1  git checkout -b featurel1
```

然后修改 `a.txt` 文件首行文字为 `featurel 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 'featurel 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

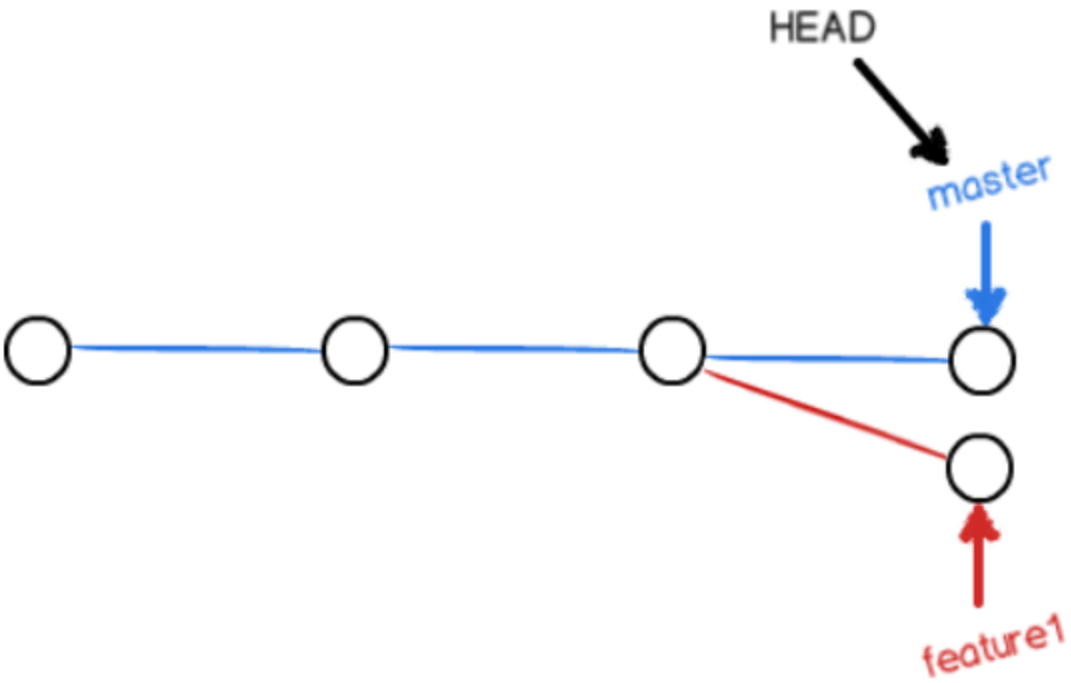
此时 `a.txt` 文件的内容变成 `featurel commit` ，没有出现冲突情况，这是因为 `git` 在内部发生了快速合并

> 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 `featurel` 分支，将文件的内容修改成 `featrue 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 `mastet 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 featurel` 进行分支合并，如下所示：

```
PS E:\Users\user\Desktop\git_conflict> git merge featurel1
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)
```

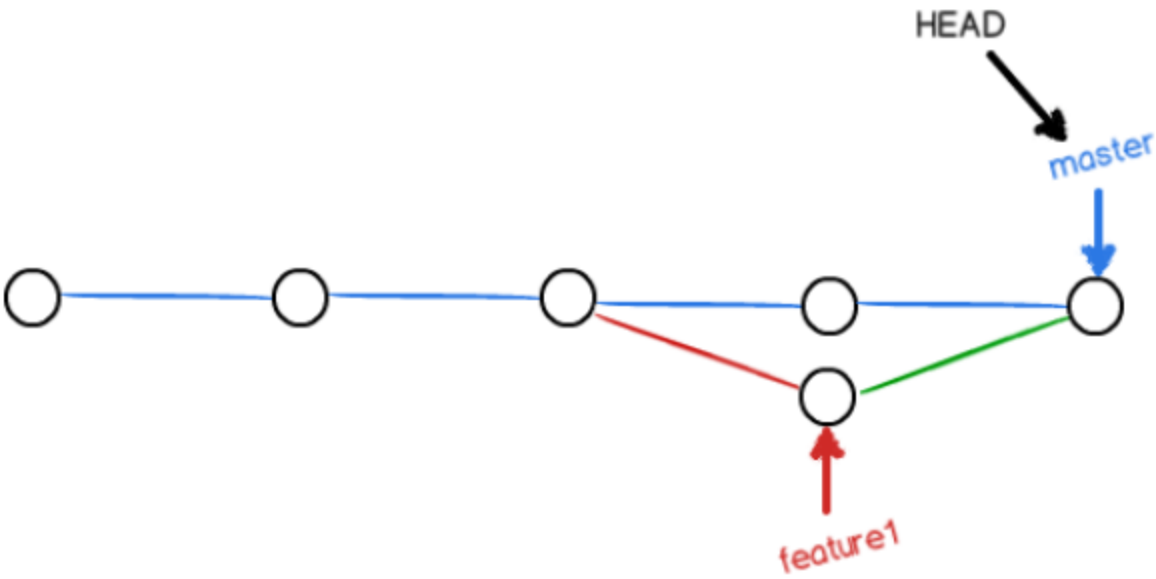
`git` 用 `<<<<<<<` ， `=======` ， `>>>>>>>` 标记出不同分支的内容：

- <<<<<<< 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：

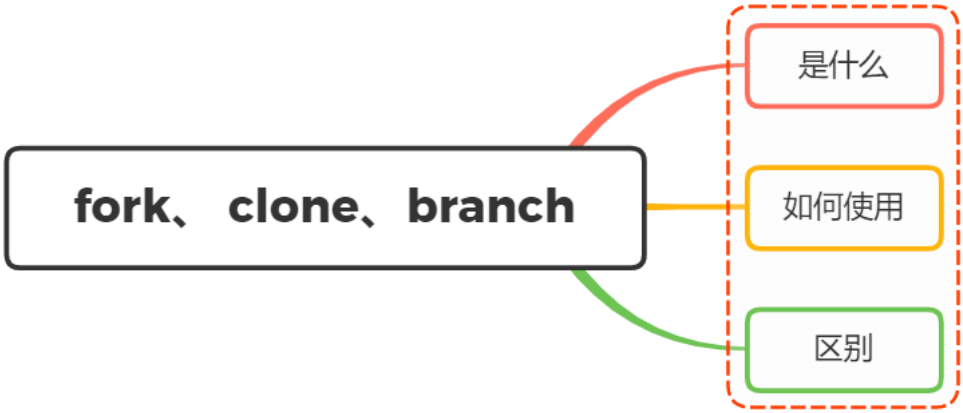
```
PS E:\Users\user\Desktop\git_conflict> git log --graph --pretty=oneline --abbrev-commit
*   01eface (HEAD -> master) conflict fixed
|\
| * 9f31dcc (feature1) featurel secont commit
* | e8f60e8 master second commit
* | dfc041b master first commit
|/
* d401a65 featurel second commit
* 6346f1e (featurel)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

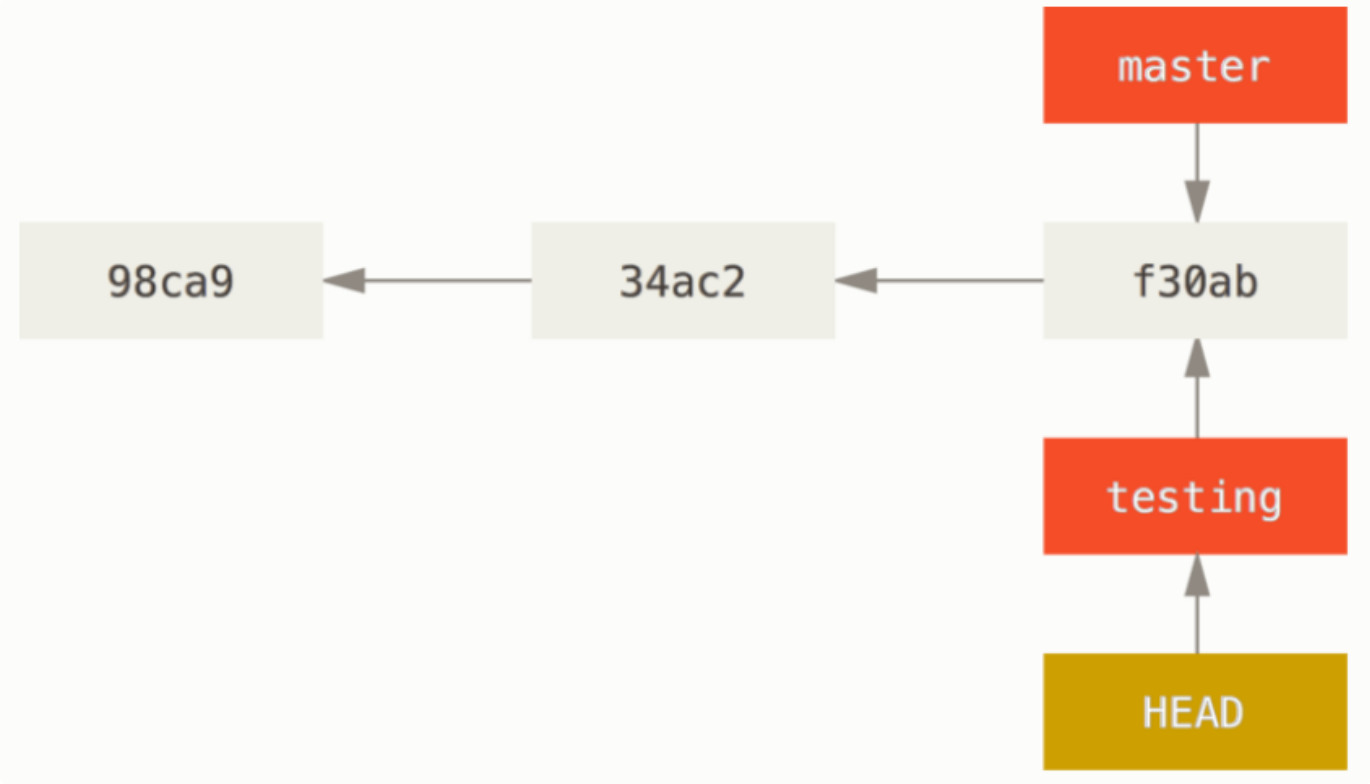
`branch`，译为分支，其作用简单而言就是开启另一个分支， 使用分支意味着你可以把你的工作从开发主线上分离开来，以免影响开发主线

`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

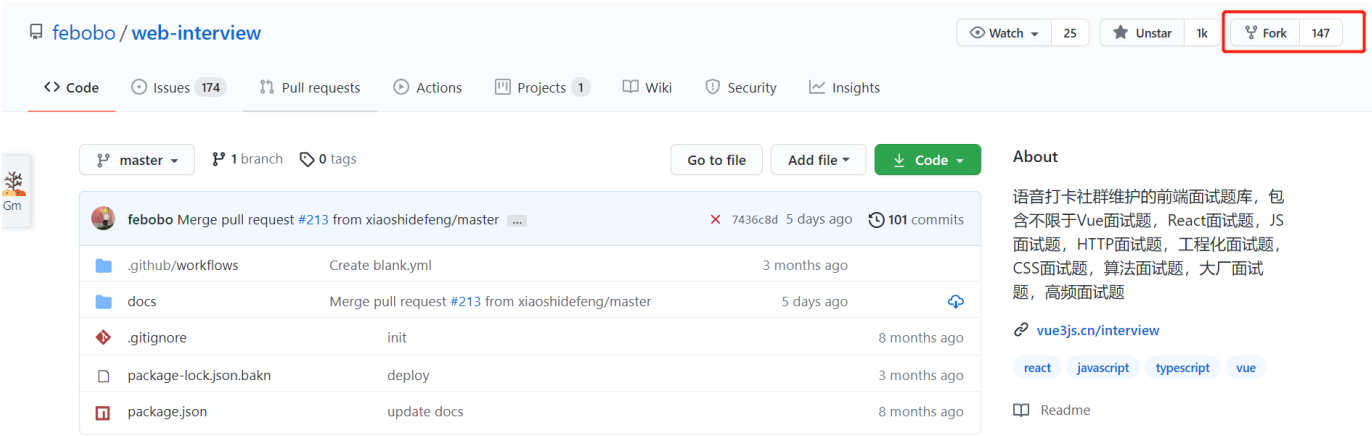
通过 `git checkout` 可以切换到另一个 `testing` 分支



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：

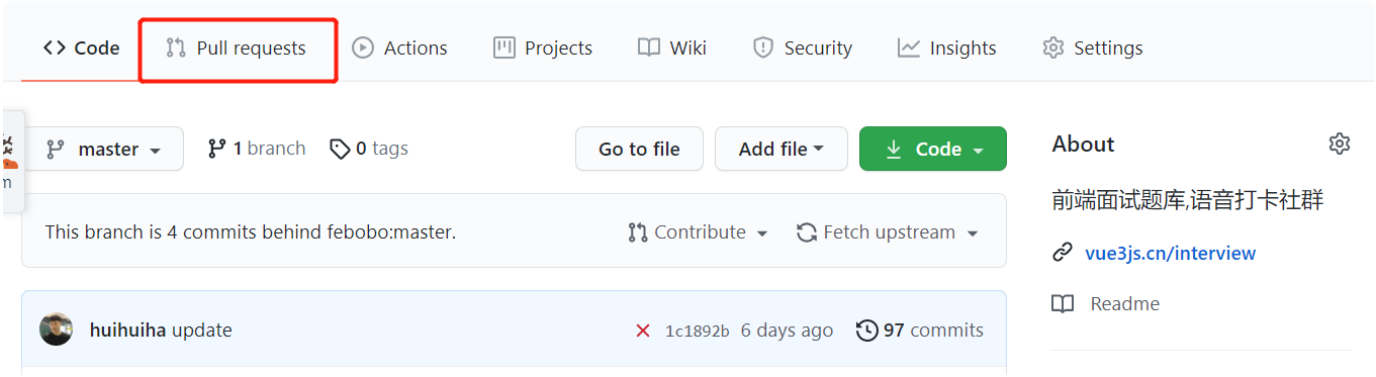


点击这个操作后会将这个仓库的文件、提交历史、issues和其余东西的仓库复制到自己的 `github` 仓库中，而你本地仓库是不会存在任何更改

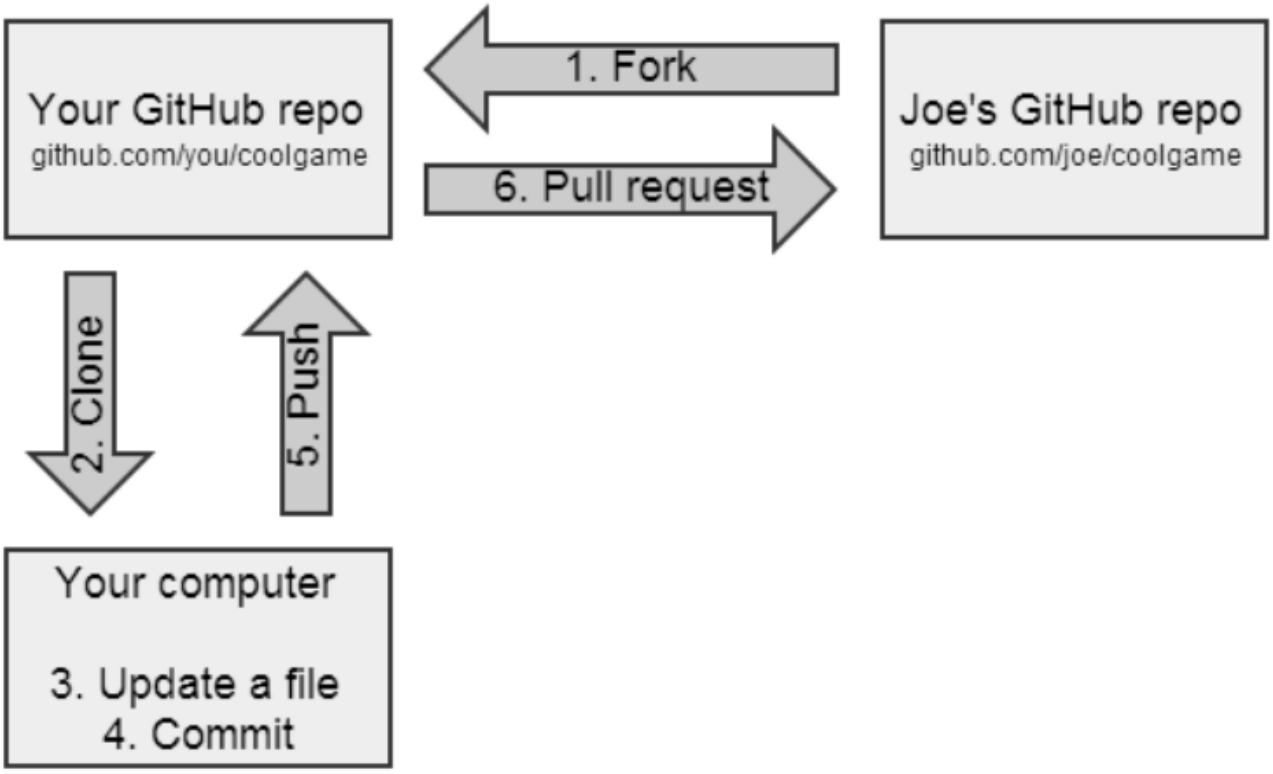
然后你就可以通过 `git clone` 对你这个复制的远程仓库进行克隆

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

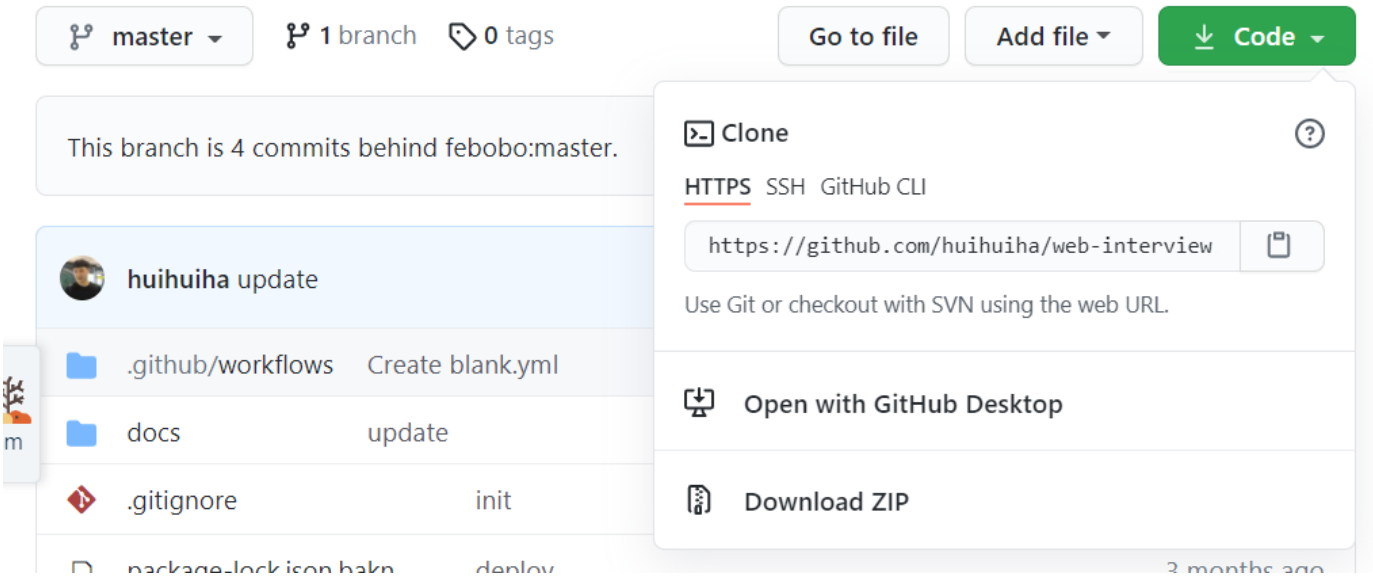


整体流程如下图



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：



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

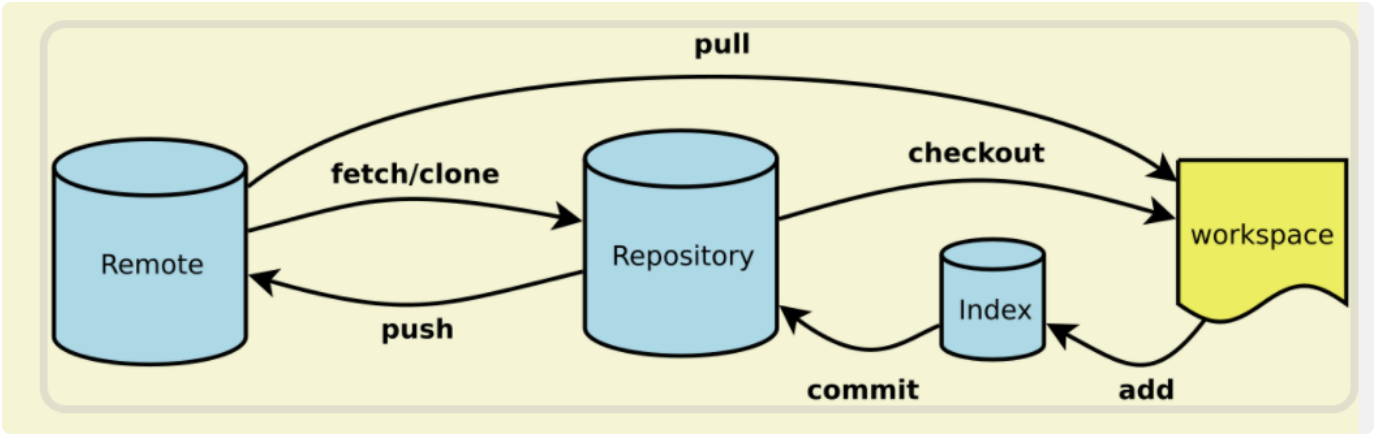
7. 说说对git pull 和 git fetch 的理解？有什么区别？

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

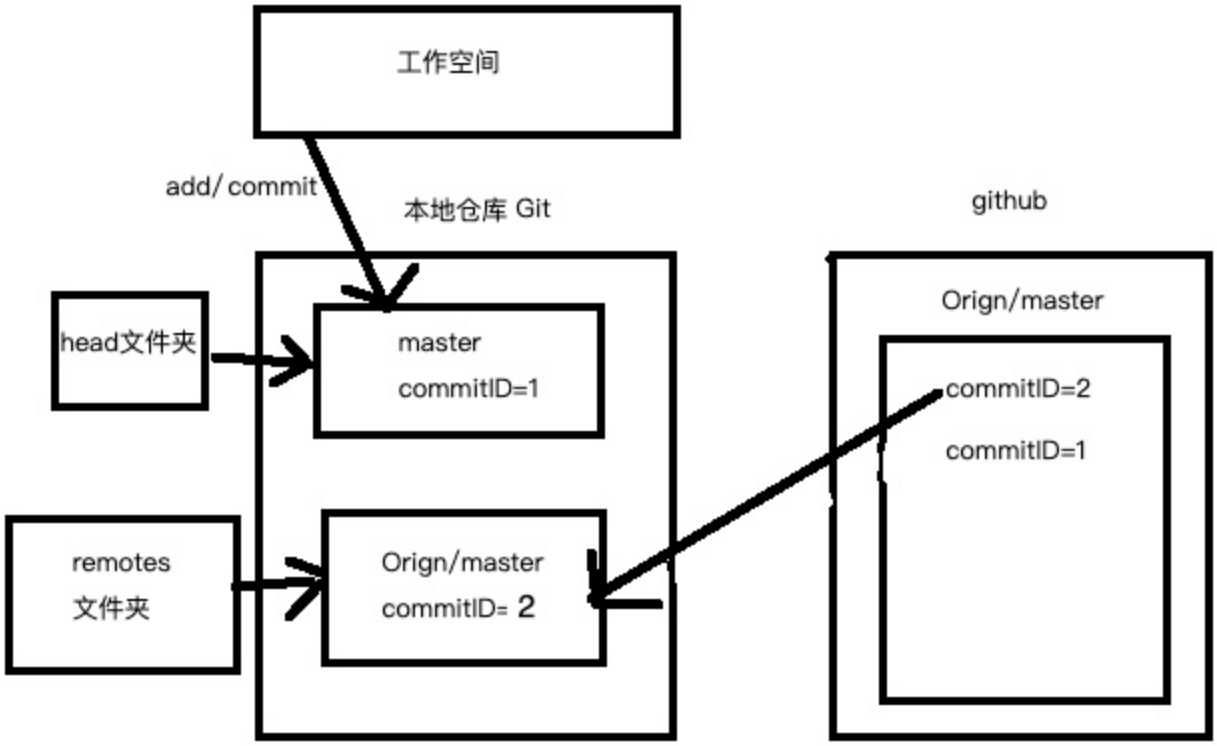
在我们本地的 `git` 文件中对应也存储了 `git` 本地仓库分支的 `commit ID` 和 跟踪的远程分支的 `commit ID` ，对应文件如下：

- .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

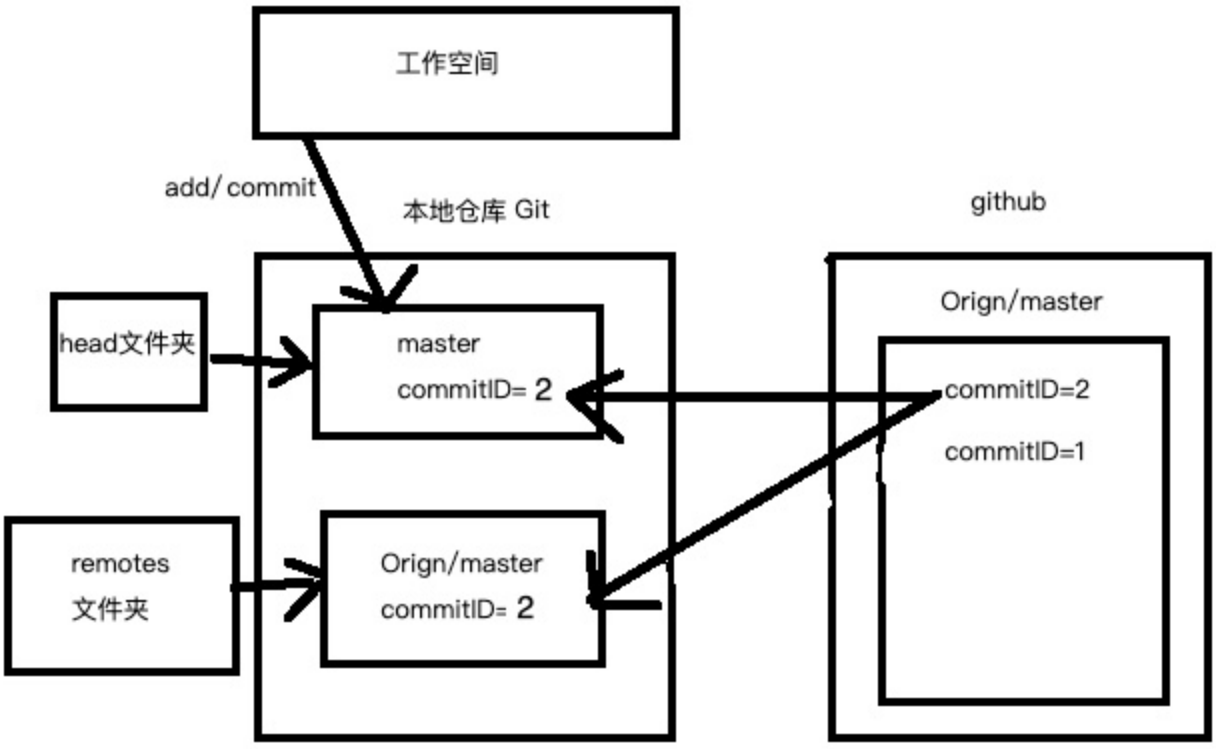
但是与 `git` 上面关联的那个 `orign/master` 的 `commit ID` 发生改变

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：

```
Plain Text | 复制代码
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

```
Plain Text | 复制代码
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：

```
git merge temp
```

7.2.2. pull

两者的用法十分相似，`pull` 用法如下：

```
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：

```
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

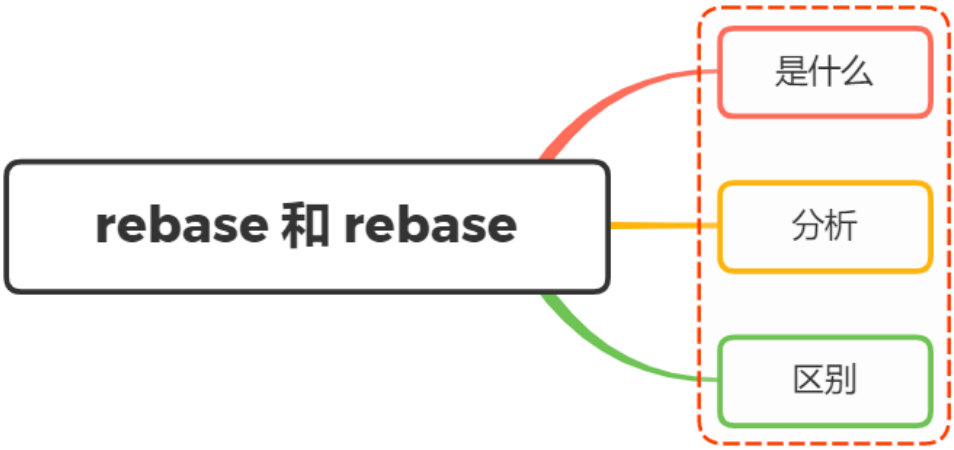
相同点：

- 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

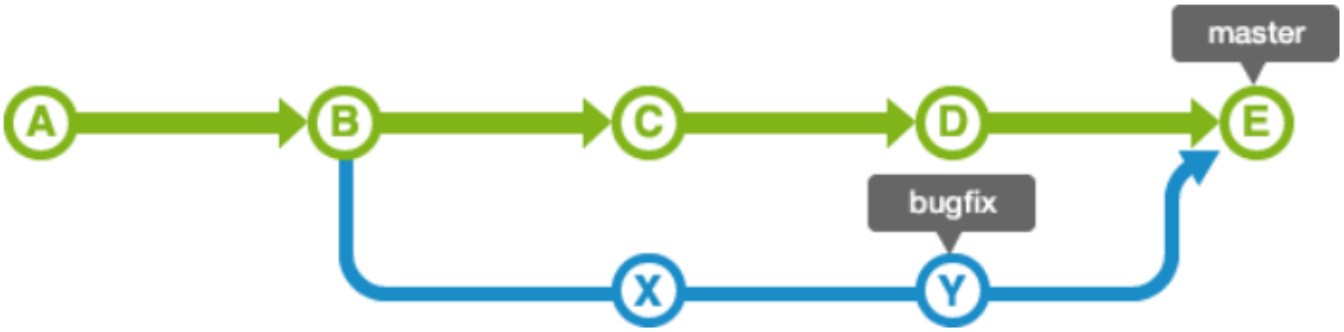
举个例子， `bugfix` 分支是从 `master` 分支分叉出来的，如下所示：

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：

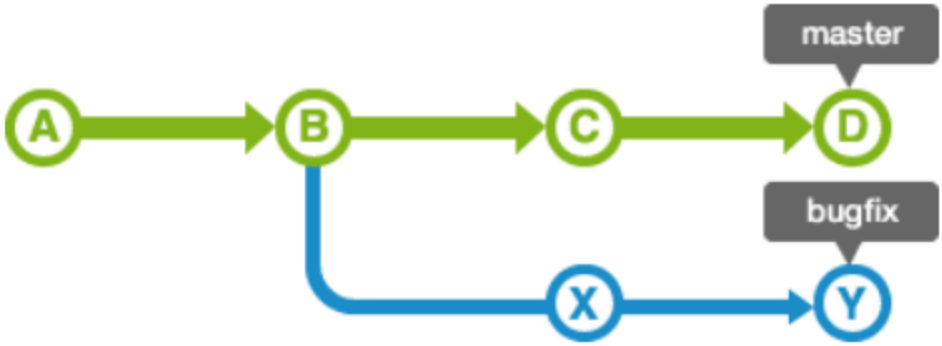
这时候使用 `git merge` 的时候，会生成一个新的提交，并且 `master` 分支的 `HEAD` 会移动到新的分支上，如下：



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 `rebasea` 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

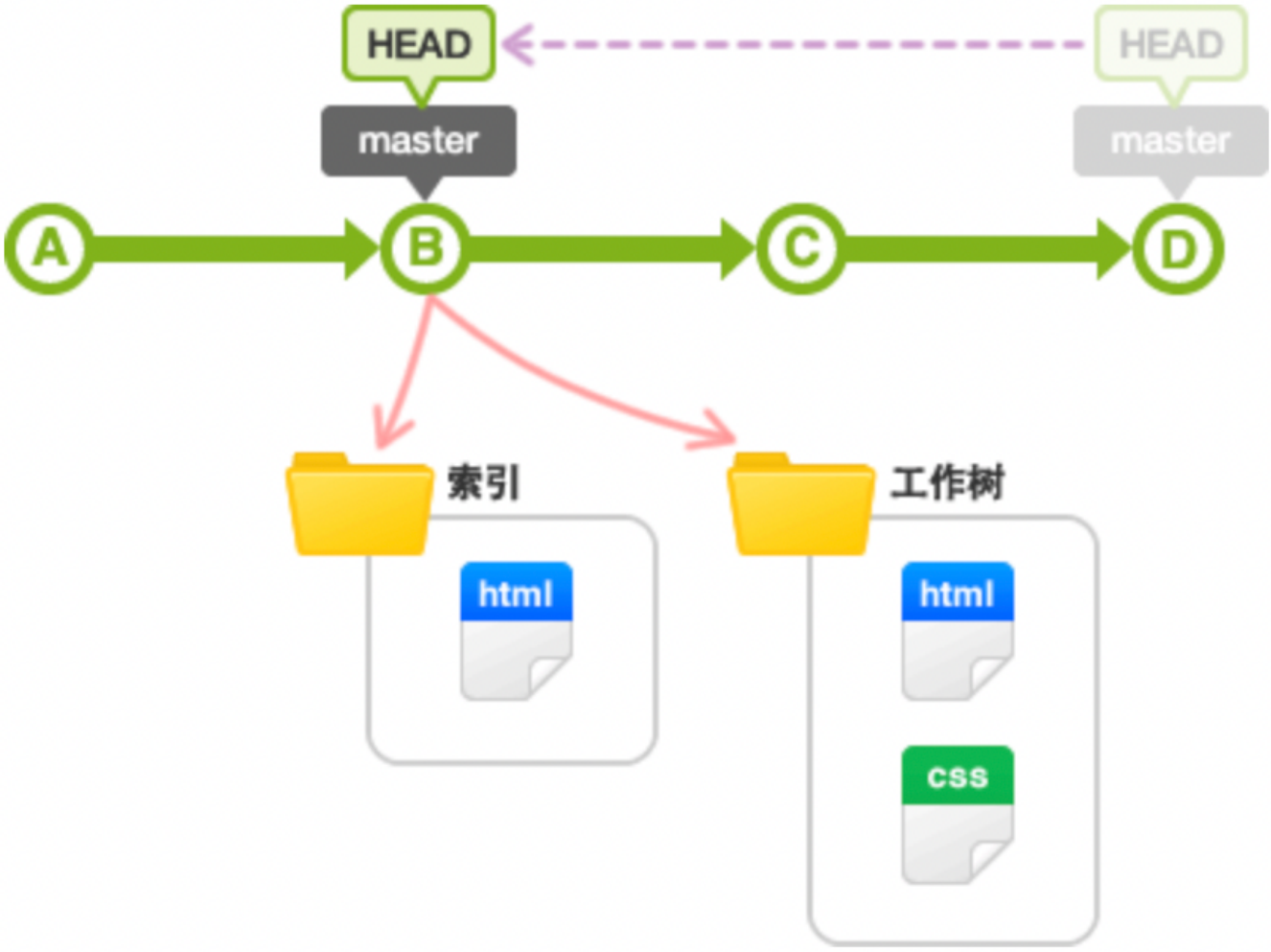
9. 说说你对git reset 和 git revert 的理解？区别？

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

Plain Text | 复制代码

```
// 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
git reset

// 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
git reset <ID>
```

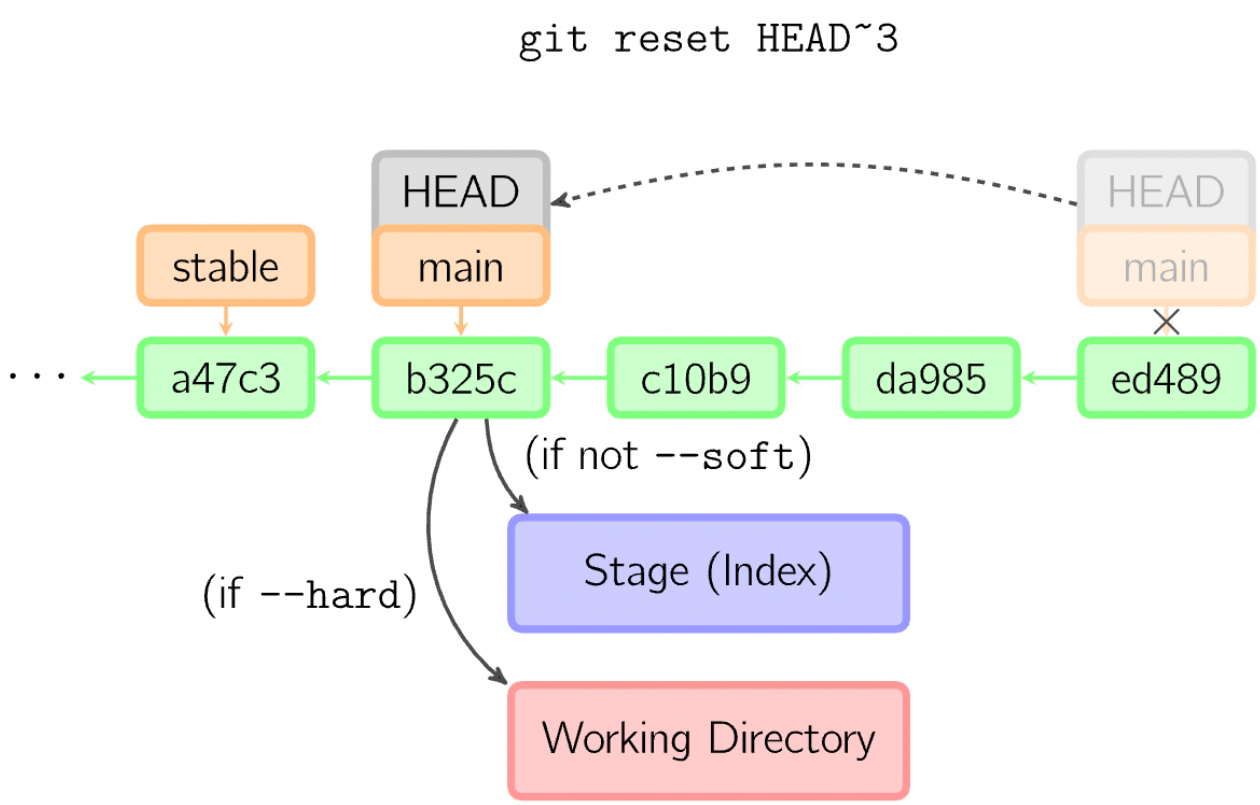
日志 `ID` 可以通过查询，可以 `git log` 进行查询，如下：

Plain Text | 复制代码

```
commit a7700083ace1204ccdff9f71631fb34c9913f7c5 (HEAD -> master)
Author: linguanghui <linguanghui@baidu.com>
Date:   Tue Aug 17 22:34:40 2021 +0800

    second commit

commit e31118663ce66717edd8a179688a7f3dde5a9393
Author: linguanghui <linguanghui@baidu.com>
Date:   Tue Aug 17 22:20:01 2021 +0800

    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



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^
```