图解git原理与日常实用指南

缘起

读了"扔物线"老师的小册《Git 原理详解及实用指南》感觉收获良多，于是想写点东西做一个总结，即加深自己的印象也希望能给社区小伙伴一点帮助，写的不对的地方还请多多指导。身为一个初入前端半年的菜鸟，由伊始的只知道git是用来托管代码的工具到逐步了解中央版本控制系统与分布式版本控制系统(git)的原理与区别；从之前只会基本的add、commit、pull、push操作到使用stash、merge、reset方便得不亦乐乎，都得益于对git原理的深入理解，逼话少说，咋们直接进入正题。前方长篇预警...

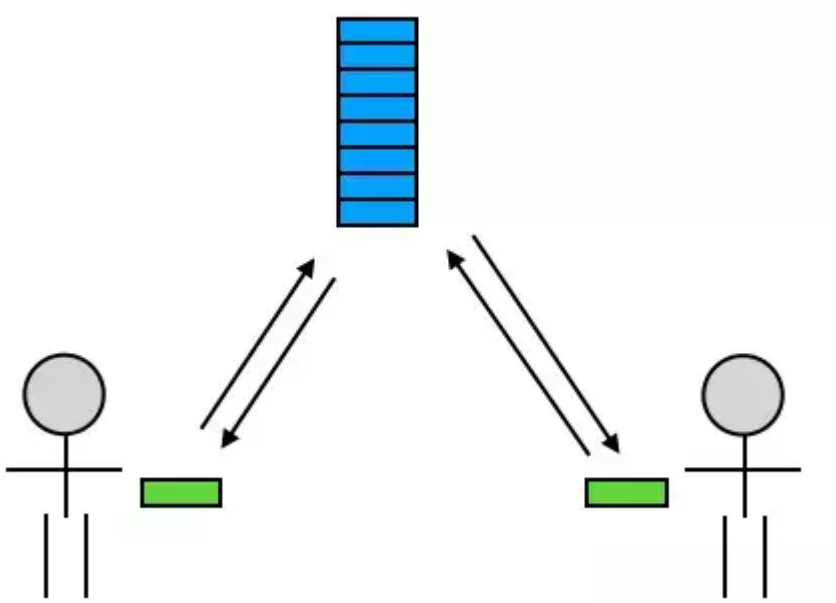
从了解版本控制系统开始

所谓版本控制，就是在文件修改的历程中保留修改历史，可以方便的撤销（如同文本编辑的撤销操作一般，只是版本控制会复杂的多）之前对文件的修改。一个版本控制系统的三个核心内容：版本控制（最基本的功能），主动提交（commit历史）和远程仓库（协同开发）。

中央式版本控制系统（VCS）

工作模型

> 1. 主工程师搭好项目框架
> 2. 在公司服务器创建一个远程仓库，并提交代码
> 3. 其他人拉取代码，并行开发
> 4. 每个人独立负责一个功能，开发完成提交代码
> 5. 其他人随时拉取代码，保持同步

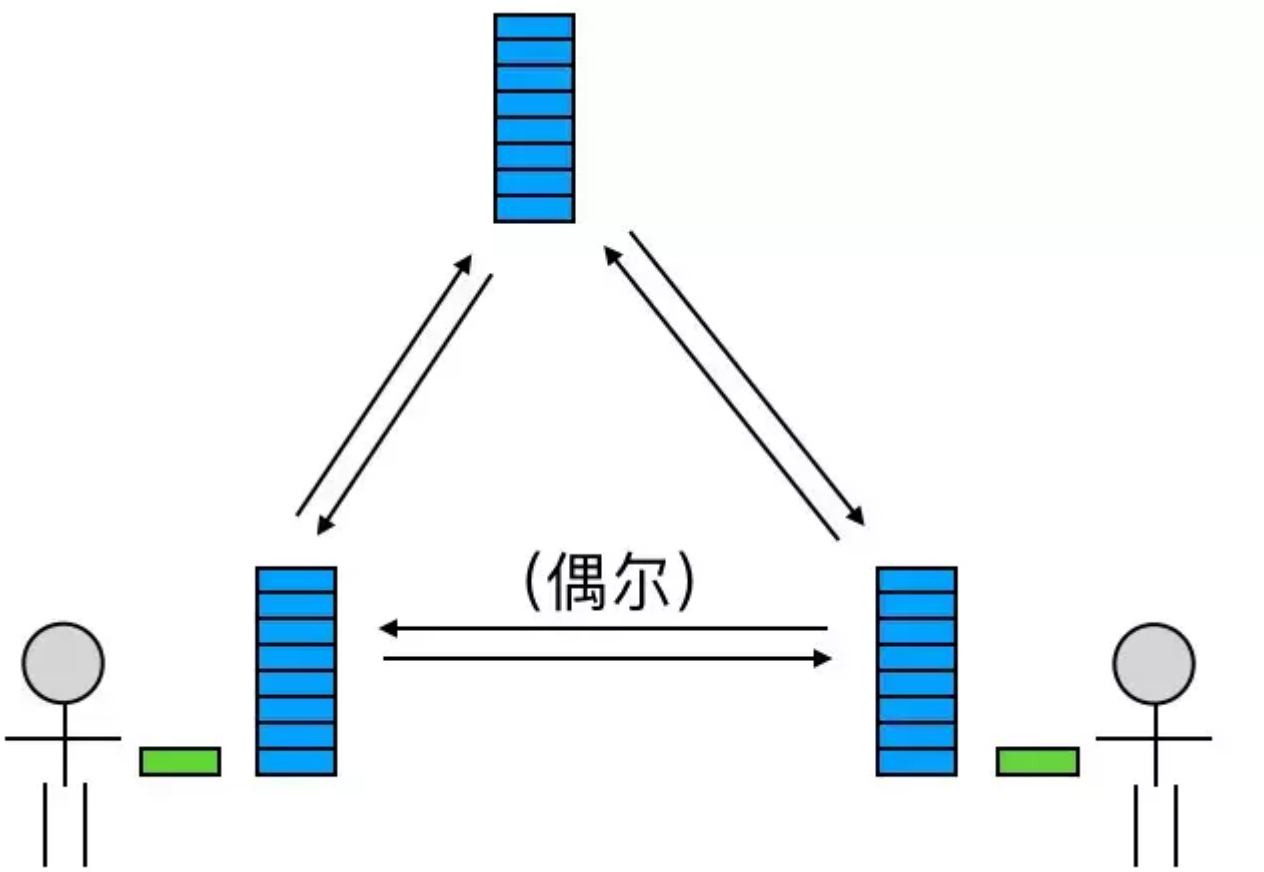


分布式版本控制系统（DVCS）

分布式与中央式的区别主要在于，分布式除了远程仓库之外团队中每一个成员的机器上都有一份本地仓库，每个人在自己的机器上就可以进行提交代码，查看版本，切换分支等操作而不需要完全依赖网络环境。

工作模型

> 1. 主工程师搭好项目框架，并提交代码到本地仓库
> 2. 在公司服务器创建一个远程仓库，并将1的提交推送到远程仓库
> 3. 其他人把远程仓库所有内容克隆到本地，拥有了各自的本地仓库，开始并行开发
> 4. 每个人独立负责一个功能，可以把每一个小改动提交到本地（由于本地提交无需立即上传到远程仓库，所以每一步提交不必是一个完整功能，而可以是功能中的一个步骤或块）
> 5. 功能开发完毕，将和这个功能相关的所有提交从本地推送到远程仓库
> 6. 每次当有人把新的提交推送到远程仓库的时候，其他人就可以选择把这些提交同步到自己的机器上，并把它们和自己的本地代码合并



分布式版本管理系统的优缺点：

优点

- 大多数操作本地进行，数度更快，不受网络与物理位置限制，不联网也可以提交代码、查看历史、切换分支等等
- 分布提交代码，提交更细利于review

缺点

- 初次clone时间较长
- 本地占用存储高于中央式系统

继续深入git原理

假设你已经安装好了git并将代码clone到了本地，新手移步git安装与代码拷贝指南。

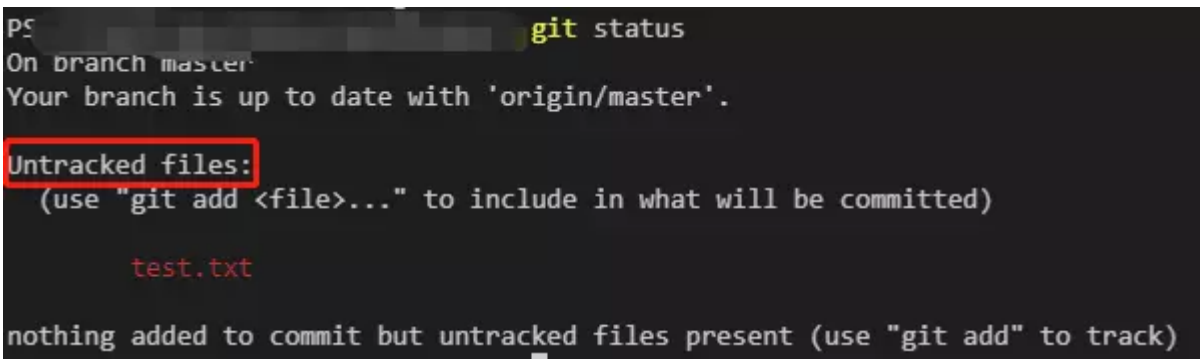
git最基本的工作模型

首先理解三个基本概念：

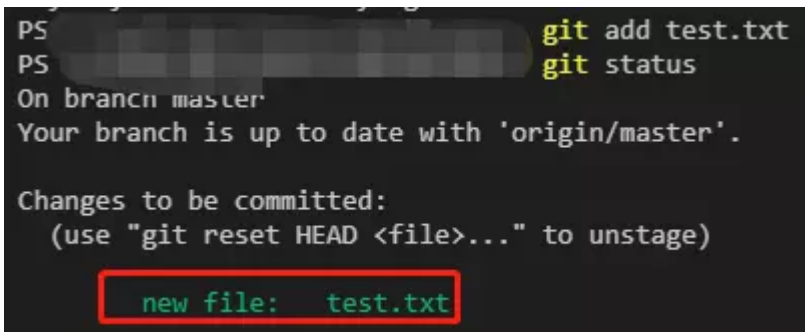
- 工作区：就是你在电脑里能看到的目录
- 版本库：工作区有一个隐藏目录.git，这个不算工作区，而是Git的本地版本库,你的所有版本信息都会存在这里
- 暂存区：英文叫stage, 或index。一般存放在 ".git目录下" 下的index文件（.git/index）中，所以我们把暂存区有时也叫作索引（index）

工作模型

1.首先新建一个test.txt文件并对其进行修改，通过status可以查看工作目录当前状态，此时test.txt对git来说是不存在的（Untracked）



2.然后通过add命令将修改放入暂存区（git开始追踪它）



可以看到，test.txt 的文字变成了绿色，它的前面多了「new file:」的标记，而它的描述也从 "Untracked files" 变成了 "Changes to be commited"。这些都说明一点：test.txt 这个文件的状态从 "untracked"（未跟踪）变成了 "staged"（已暂存），意思是这个文件中被改动的部分（也就是这整个文件）被记录进了 staging area（暂存区）

> stage 这个词在 Git 里，是「集中收集改动以待提交」的意思；而 staging area ，就是一个「汇集待提交的文件改动的地方」。简称「暂存」和「暂存区」。至于 staged 表示「已暂存」，就不用再解释了吧?

3.现在文件已经放入暂存区，可以用commit命令提交：

```
PS D:\...\keeper> git commit -m '新增test.txt文件'
[master fbce093] 新增test.txt文件
 1 file changed, 1 insertion(+)
 create mode 100644 keeper/test.txt
```

在这里你也可以直接commit提交会进入commit信息编辑页面，而加上-m参数可以快捷输入简短的提交备注信息，这样你就完成了一次提交（可以通过`git log`查看提交历史）

接着对该文件再次进行修改，输入`git status`可以看到，该文件 又变红了，不过这次它左边的文字不是 "New file:" 而是 "modified:"，而且上方显示它的状态也不是 "Untracked" 而是 "not staged for commit"，意思很明确：Git 已经认识这个文件了，它不是个新文件，但它有了一些改动。所以虽然状态的显示有点不同，但处理方式还是一样的：

```
PS D:\... git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   test.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

接下来再次将该文件add、commit，查看log可以看到已经存在两条提交记录

```
PS ...> git log
commit 61bcf7c989f0419bf8cfdebd7819e3a76d45d85c (HEAD -> master)
Author: kirtios <lijh@getui.com>
Date:   Sun Feb 24 09:30:50 2019 +0800

    修改test.txt文件

commit fbce09324bde20867e7a5f5627e17fbcb5e55b17
Author: kirtios <lijh@getui.com>
Date:   Sun Feb 24 09:18:18 2019 +0800

    新增test.txt文件
```

4.最后通过push把本地的所有commit上传到远程仓库：

```
PS D:\...keeper> git push
Counting objects: 8, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (8/8), 712 bytes | 89.00 KiB/s, done.
Total 8 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 1 local object.
To https://github.com/...git
   82a6955..61bcf7c  master -> master
```

团队工作基本模型

工作模型
1.在上面基本操作的基础上，同事 commit 代码到他的本地，并 push 到远程仓库
2.你把远程仓库新的提交通过 pull指令拉取到你的本地

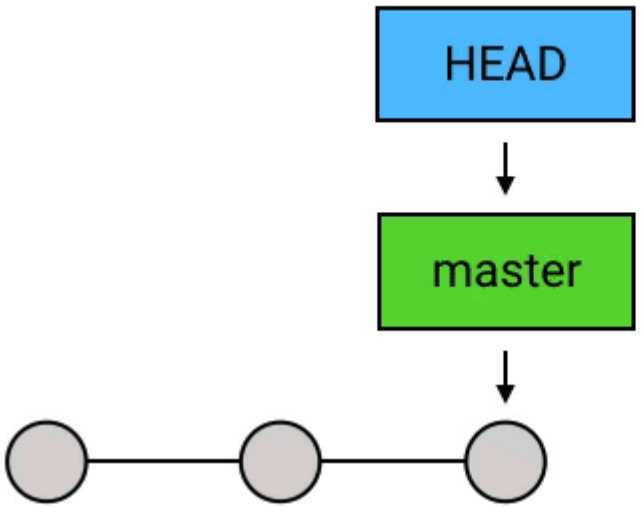
通过这个流程，你和同事就可以简单地合作了：你写了代码，commit，push 到远程仓库，然后他 pull 到他的本地；他再写代码，commit, push 到远程仓库，然后你再 pull 到你的本地。你来我往，配合得不亦乐乎。（但是有时候push会失败）

> 为什么会失败?
> 因为 Git 的push 其实是用本地仓库的commit记录去覆盖远程仓库的commit记录（注：这是简化概念后的说法，push 的实质和这个说法略有不同），而如果在远程仓库含有本地没有的commit的时候，push （如果成功）将会导致远端的commit被擦掉。这种结果当然是不可行的，因此 Git 会在 push 的时候进行检查，如果出现这样的情况，push 就会失败

这时只需要先通过`git pull`（实为fetch和merge的组合操作）将本地仓库的提交和远程仓库的提交进行合并，然后再push就可以了

Feature Branching：最流行的工作流

核心：
（1）任何新的功能（feature）或 bug 修复全都新建一个 branch 来写；
（2）branch 写完后，合并到 master，然后删掉这个 branch（可使用`git origin -d 分支名`删除远程仓库的分支）。



优势：
（1）代码分享：写完之后可以在开发分支review之后再merge到master分支
（2）一人多任务：当正在开发接到更重要的新任务时，你只要稍微把目前未提交的代码简单收尾一下，然后做一个带有「未完成」标记的提交（例如，在提交信息里标上「TODO」），然后回到 master 去创建一个新的 branch 进行开发就好了。

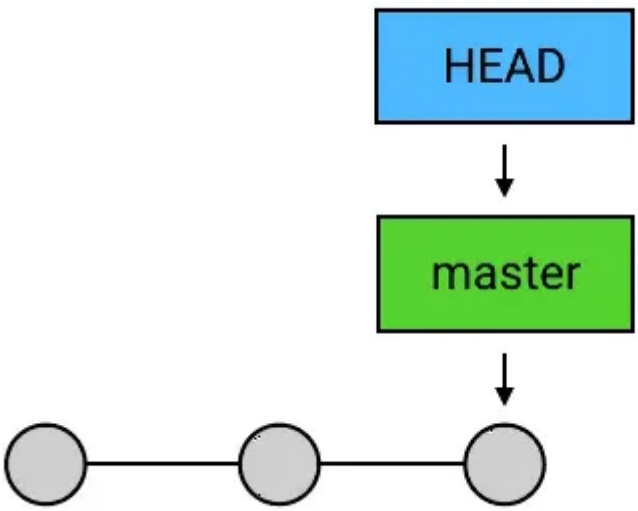
HEAD、branch、引用的本质以及push的本质

HEAD：当前commit的引用

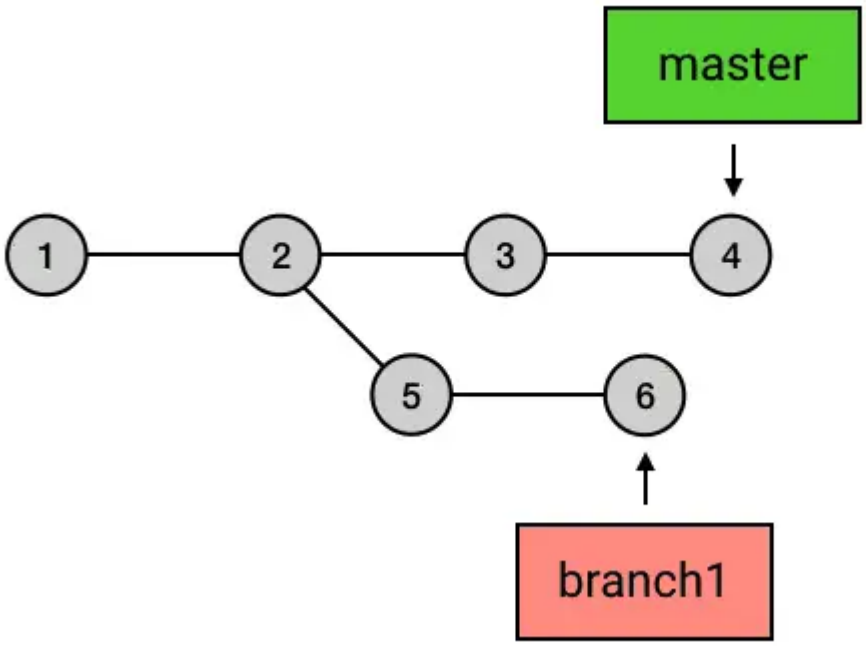
当前 commit 在哪里，HEAD 就在哪里，这是一个永远自动指向当前 commit 的引用，所以你永远可以用 HEAD 来操作当前 commit，

branch：

HEAD 是 Git 中一个独特的引用，它是唯一的。而除了 HEAD 之外，Git 还有一种引用，叫做 branch（分支）。
HEAD 除了可以指向 commit，还可以指向一个branch，当指向一个branch时，HEAD会通过branch间接指向当前commit，HEAD移动会带着branch一起移动：

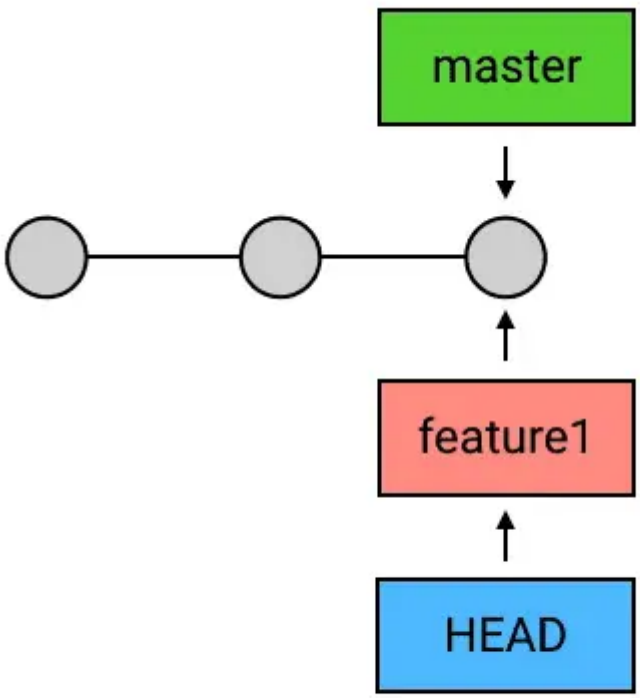


branch 包含了从初始 commit 到它的所有路径，而不是一条路径。并且，这些路径之间也是彼此平等的。

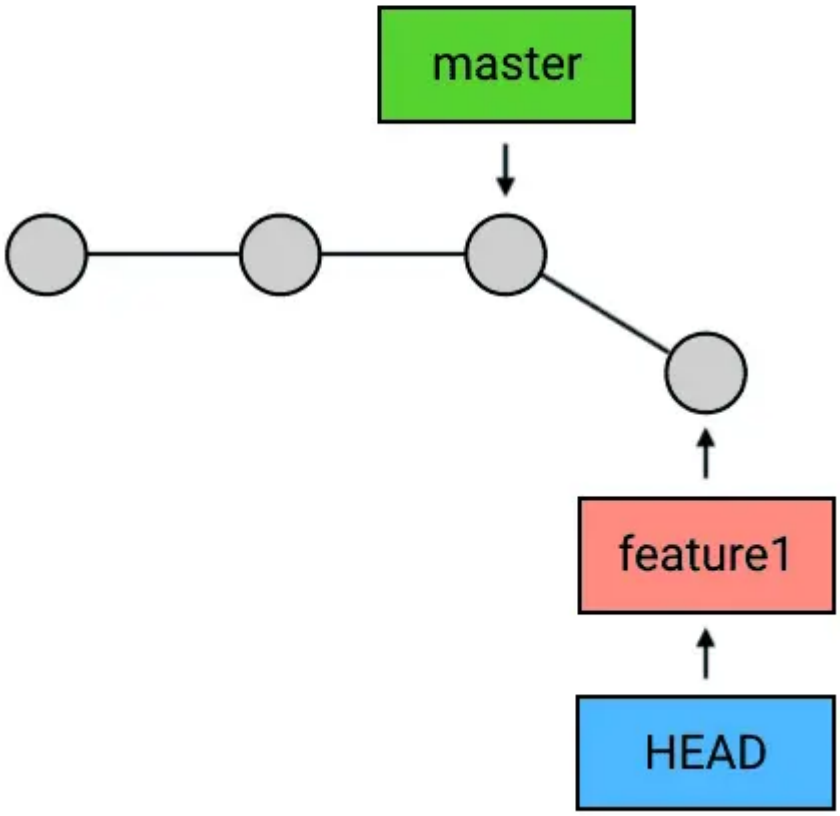


像上图这样，master 在合并了 branch1 之后，从初始 commit 到 master 有了两条路径。这时，master 的串就包含了 1 2 3 4 7 和 1 2 5 6 7 这两条路径。而且，这两条路径是平等的，1 2 3 4 7 这条路径并不会因为它是「原生路径」而拥有任何的特别之处

创建branch：`git branch 名称`
切换branch：`git checkout 名称`（将HEAD指向该branch）
创建+切换：`git checkout -b 名称`
在切换到新的 branch 后，再次 commit 时 HEAD 就会带着新的 branch 移动了：



而这个时候，如果你再切换到 master 去 commit，就会真正地出现分叉了：



删除branch：`git branch -d 名称`
注意：
（1）HEAD 指向的 branch 不能删除。如果要删除 HEAD 指向的 branch，需要先用 checkout 把 HEAD 指向其他地方。
（2）由于 Git 中的 branch 只是一个引用，所以删除 branch 的操作也只会删掉这个引用，并不会删除任何的 commit。（不过如果一个 commit 不在任何一个 branch 的「路径」上，或者换句话说，如果没有任何一个 branch 可以回溯到这条 commit（也许可以称为野生 commit？），那么在一定时间后，它会被 Git 的回收机制删除掉）
（3）出于安全考虑，没有被合并到 master 过的 branch 在删除时会失败（怕误删未完成branch）把-d换成-D可以强制删除

引用的本质

所谓引用，其实就是一个个的字符串。这个字符串可以是一个 commit 的 SHA-1 码（例：c08de9a4d8771144cd23986f9f76c4ed729e69b0），也可以是一个 branch（例：ref: refs/heads/feature3）。

Git 中的 HEAD 和每一个 branch 以及其他的引用，都是以文本文件的形式存储在本地仓库 .git 目录中，而 Git 在工作的时候，就是通过这些文本文件的内容来判断这些所谓的「引用」是指向谁的。

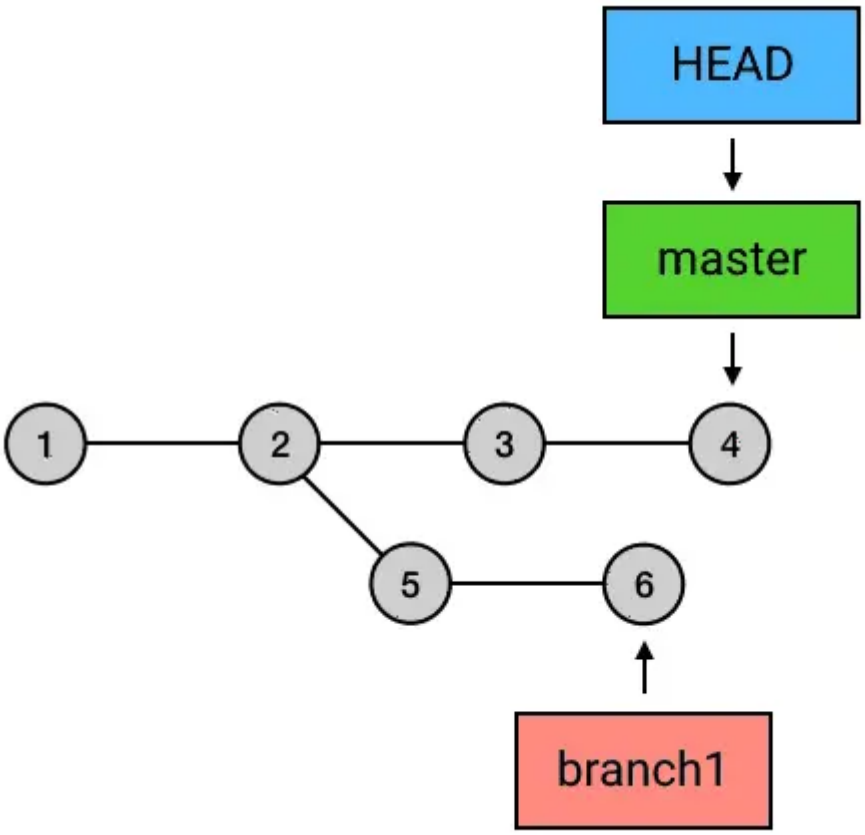
push的本质：把 branch 上传到远程仓库

（1）把当前branch位置上传到远程仓库，并把它路径上的commits一并上传
（2）git中（2.0及以后版本），`git push`不加参数只能上传到从远程仓库clone或者pull下来的分支，如需push在本地创建的分支则需使用`git push origin 分支名`的命令
（3）远端仓库的HEAD并不随push与本地一致，远端仓库HEAD永远指向默认分支（master），并随之移动（可以使用`git br -r`查看远程分支的HEAD指向）。

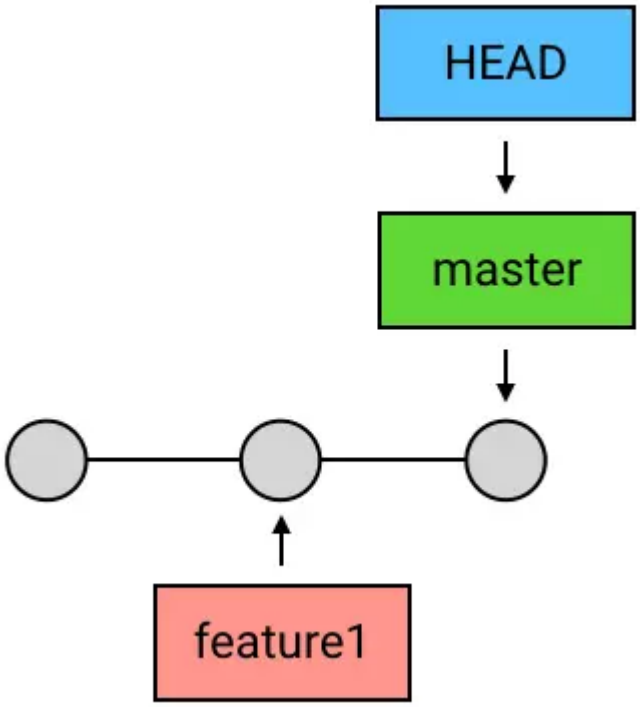
开启git操作之旅

merge：合并

含义：从目标 commit 和当前 commit （即 HEAD 所指向的 commit）分叉的位置起，把目标 commit 的路径上的所有 commit 的内容一并应用到当前 commit，然后自动生成一个新的 commit。



当执行`git merge branch1`操作，Git 会把 5 和 6 这两个 commit 的内容一并应用到 4 上，然后生成一个新的提交 7 。
merge的特殊情况：
（1）merge冲突：你的两个分支改了相同的内容，Git 不知道应该以哪个为准。如果在 merge 的时候发生了这种情况，Git 就会把问题交给你来决定。具体地，它会告诉你 merge 失败，以及失败的原因；这时候你只需要手动解决掉冲突并重新add、commit（改动不同文件或同一文件的不同行都不会产生冲突）；或者使用`git merge --abort`放弃解决冲突，取消merge
（2）HEAD 领先于目标 commit：merge是一个空操作：



此时merge不会有任何反应。
（3）HEAD 落后于 目标 commit且不存在分支（fast-forward）：

git会直接把HEAD与其指向的branch（如果有的话）一起移动到目标commit。

rebase：给commit序列重新设置基础点

有些人不喜欢 merge，因为在 merge 之后，commit 历史就会出现分叉，这种分叉再汇合的结构会让有些人觉得混乱而难以管理。如果你不希望 commit 历史出现分叉，可以用 rebase 来代替 merge。

可以看出，通过 rebase，5 和 6 两条 commits 把基础点从 2 换成了 4 。通过这样的方式，就让本来分叉了的提交历史重新回到了一条线。这种「重新设置基础点」的操作，就是 rebase 的含义。另外，在 rebase 之后，记得切回 master 再 merge 一下，把 master 移到最新的 commit。

> 为什么要从 branch1 来 rebase，然后再切回 master 再 merge 一下这么麻烦，而不是直接在 master 上执行 rebase?
>
> 从图中可以看出，rebase 后的每个 commit 虽然内容和 rebase 之前相同，但它们已经是不同的 commit 了（每个commit有唯一标志）。如果直接从 master 执行 rebase 的话，就会是下面这样：
>
> 这就导致 master 上之前的两个最新 commit （3和4）被剔除了。如果这两个 commit 之前已经在远程仓库存在，这就会导致没法 push：
>
> 所以，为了避免和远程仓库发生冲突，一般不要从 master 向其他 branch 执行 rebase 操作。而如果是 master 以外的 branch 之间的 rebase（比如 branch1 和 branch2 之间），就不必这么多费一步，直接 rebase 就好。

需要说明的是,rebase 是站在需要被 rebase 的 commit 上进行操作，这点和 merge 是不同的。

stash：临时存放工作目录的改动

stash 指令可以帮你把工作目录的内容全部放在你本地的一个独立的地方，它不会被提交，也不会被删除，你把东西放起来之后就可以去做你的临时工作了，做完以后再来取走，就可以继续之前手头的事了。
操作步骤：
（1）`git stash`可以加上save参数后面带备注信息（`git stash save '备注信息'`）
（2）此时工作目录已经清空，可以切换到其他分支干其他事情了
（3）`git stash pop`弹出第一个stash（该stash从历史stash中移除）；或者使用`git stash apply`达到相同的效果（该stash仍存在stash list中），同时可以使用`git stash list`查看stash历史记录并在apply后面加上指定的stash返回到该stash。
注意：没有被track的文件会被git忽略而不被stash，如果想一起stash，加上-u参数。

reflog：引用记录的log

可以查看git的引用记录，不指定参数，默认显示HEAD的引用记录；如果不小心把分支删掉了，可以使用该命令查看引用记录，然后使用checkout切到该记录处重建分支即可。

> 注意：不再被引用直接或间接指向的 commits 会在一定时间后被 Git 回收，所以使用 reflog 来找回被删除的 branch 的操作一定要及时，不然有可能会由于 commit 被回收而再也找不回来。

看看我都改了什么

log：查看已提交内容

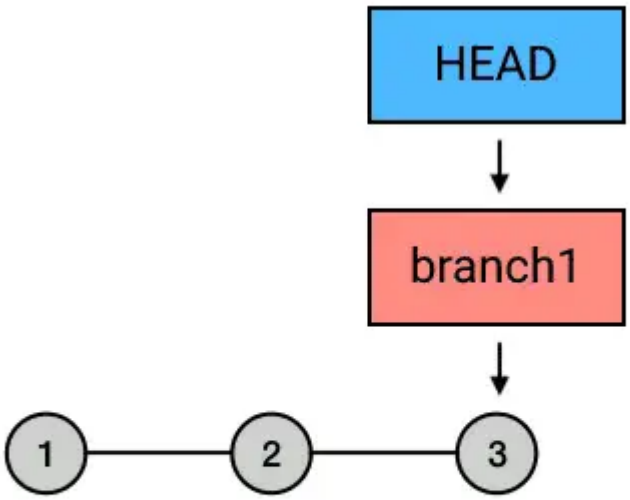
`git log -p`可以查看每个commit的改动细节（到改动文件的每一行）
`git log --stat`查看简要统计（哪几个文件改动了）
`git show 指定commit 指定文件名`查看指定commit的指定文件改动细节

diff：查看未提交内容

`git diff --staged`可以显示暂存区和上一条提交之间的不同。换句话说，这条指令可以让你看到「如果你立即输入 git commit，你将会提交什么」
`git diff`可以显示工作目录和暂存区之间的不同。换句话说，这条指令可以让你看到「如果你现在把所有文件都 add，你会向暂存区中增加哪些内容」
`git diff HEAD`可以显示工作目录和上一条提交之间的不同，它是上面这二者的内容相加。换句话说，这条指令可以让你看到「如果你现在把所有文件都 add 然后 git commit，你将会提交什么」（不过需要注意，没有被 Git 记录在案的文件（即从来没有被 add 过的文件，untracked files 并不会显示出来。因为对 Git 来说它并不存在）实质上，如果你把 HEAD 换成别的commit，也可以显示当前工作目录和这条 commit 的区别。

刚刚提交的代码发现写错了怎么办?

再提一个修复了错误的commit？可以是可以，不过还有一个更加优雅和简单的解决方法：commit --amend。
具体做法：
（1）修改好问题
（2）将修改add到暂存区
（3）使用`git commit --amend`提交修改，结果如下图：



减少了一次无谓的commit。

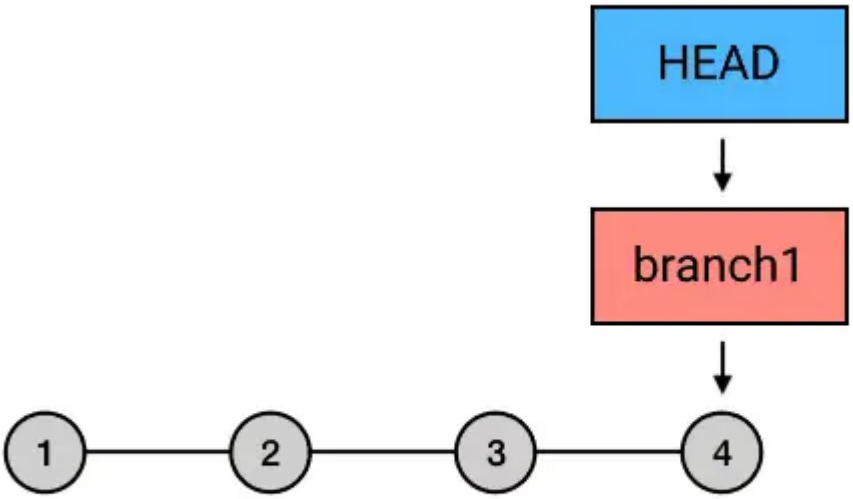
错误不是最新的提交而是倒数第二个?

使用rebase -i（交互式rebase）：
所谓「交互式 rebase」，就是在 rebase 的操作执行之前，你可以指定要 rebase 的 commit 链中的每一个 commit 是否需要进一步修改，那么你就可以利用这个特点，进行一次「原地 rebase」。

操作过程：
（1）`git rebase -i HEAD^^`

> 说明：在 Git 中，有两个「偏移符号」： ^ 和 ~。
> ^ 的用法：在 commit 的后面加一个或多个 ^ 号，可以把 commit 往回偏移，偏移的数量是 ^ 的数量。例如：master^ 表示 master 指向的 commit 之前的那个 commit； HEAD^^ 表示 HEAD 所指向的 commit 往前数两个 commit。
> ~ 的用法：在 commit 的后面加上 ~ 号和一个数，可以把 commit 往回偏移，偏移的数量是 ~ 号后面的数。例如：HEAD~5 表示 HEAD 指向的 commit往前数 5 个 commit。

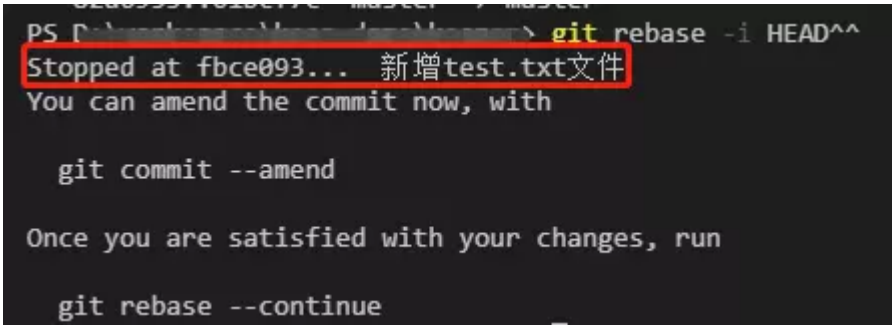
上面这行代码表示，把当前 commit （ HEAD 所指向的 commit） rebase 到 HEAD 之前 2 个的 commit 上：



(2)进入编辑页面，选择commit对应的操作，commit为正序排列，旧的在上，新的在下，前面黄色的为如何操作该commit，默认pick（直接应用该commit不做任何改变），修改第一个commit为edit（应用这个 commit，然后停下来等待继续修正）然后:wq退出编辑页面，此时rebase停在第二个commit的位置，此时可以对内容进行修改：

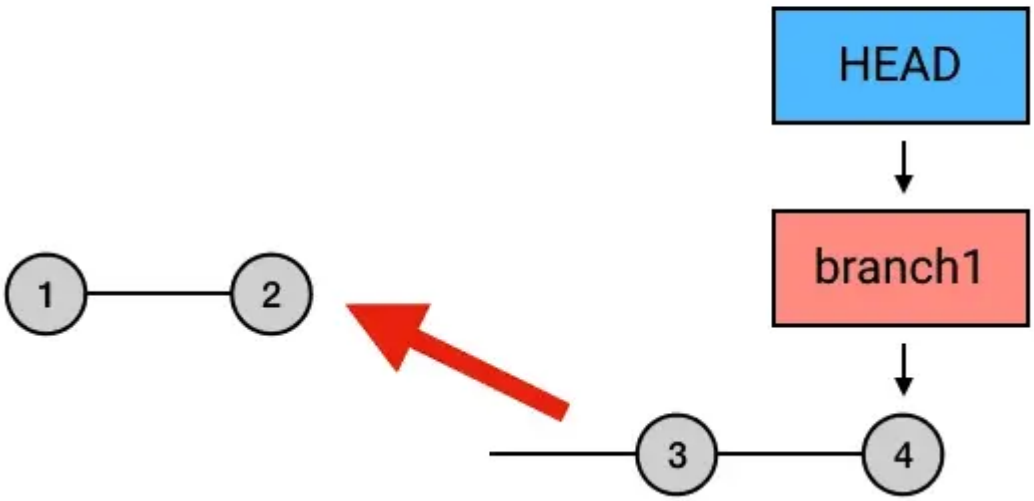
```
pick fbce093 新增test.txt文件
pick 61bcf7c 修改test.txt文件

# Rebase 82a6955..61bcf7c onto 82a6955 (2 commands)
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
~
~
~
~
```



（3）修改完后使用add，commit --amend将修改提交

（4）`git rebase --continue`继续 rebase 过程，把后面的 commit 直接应用上去，这次交互式 rebase 的过程就完美结束了，你的那个倒数第二个写错的 commit 就也被修正了：

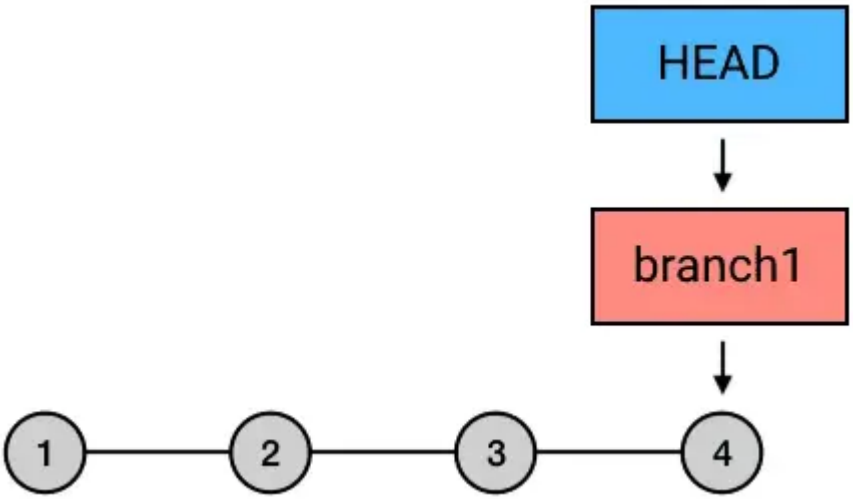


想直接丢弃某次提交?

reset --hard 丢弃最新的提交

`git reset --hard HEAD^`

> HEAD^ 表示 HEAD 往回数一个位置的 commit，上节刚说过，记得吧?



用交互式 rebase 撤销历史提交

操作步骤与修改历史提交类似，第二步把需要撤销的commit修改为drop，其他步骤不再赘述。

用 rebase --onto 撤销提交

`git rebase --onto HEAD^^ HEAD^ branch1`
上面这行代码的意思是：以倒数第二个 commit 为起点（起点不包含在 rebase 序列里），branch1 为终点，rebase 到倒数第三个 commit 上。

错误代码已经push?

有的时候，代码 push 到了远程仓库，才发现有个 commit 写错了。这种问题的处理分两种情况：

出错内容在自己的分支

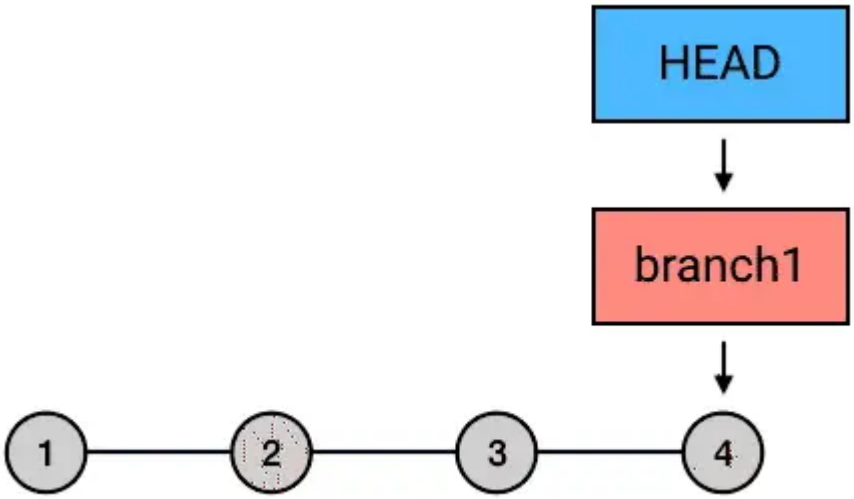
假如是某个你自己独立开发的 branch 出错了，不会影响到其他人，那没关系用前面几节讲的方法把写错的 commit 修改或者删除掉，然后再 push 上去就好了。但是此时会push报错，因为远程仓库包含本地没有的 commits（在本地已经被替换或被删除了），此时直接使用`git push origin 分支名 -f`强制push。

问题内容已合并到master

（1）增加新提交覆盖之前内容
（2）使用`git revert 指定commit`它的用法很简单，你希望撤销哪个 commit，就把它填在后面。如：`git revert HEAD^`
上面这行代码就会增加一条新的 commit，它的内容和倒数第二个 commit 是相反的，从而和倒数第二个 commit 相互抵消，达到撤销的效果。在 revert 完成之后，把新的 commit 再 push 上去，这个 commit 的内容就被撤销了。
它和前面所介绍的撤销方式相比，最主要的区别是，这次改动只是被「反转」了，并没有在历史中消失掉，你的历史中会存在两条 commit：一个原始 commit，一个对它的反转 commit。

reset：不止可以撤销提交

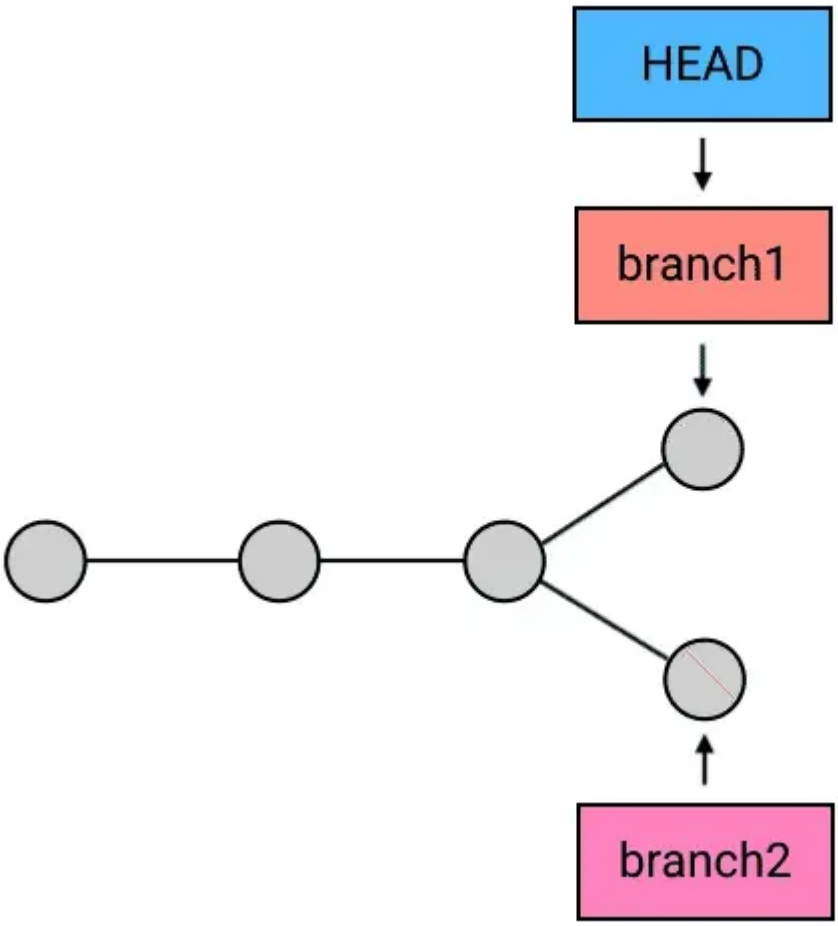
`git reset --hard 指定commit`你的工作目录里的内容会被完全重置为和指定commit位置相同的内容。换句话说，就是你的未提交的修改会被全部擦掉。
`git reset --soft 指定commit`会在重置 HEAD 和 branch 时，保留工作目录和暂存区中的内容，并把重置 HEAD 所带来的新的差异放进暂存区。
什么是「重置 HEAD 所带来的新的差异」？就是这里：



`git reset --mixed（或者不加参数） 指定commit`保留工作目录，并且清空暂存区。也就是说，工作目录的修改、暂存区的内容以及由 reset 所导致的新的文件差异，都会被放进工作区。简而言之，就是「把所有差异都混合（mixed）放在工作区中」。

checkout：签出指定commit

checkout的本质是签出指定的commit，不止可以切换branch还可以指定commit作为参数，把HEAD移动到指定的commit上；与reset的区别在于只移动HEAD不改变绑定的branch；`git checkout --detach`可以把 HEAD 和 branch 脱离，直接指向当前 commit。



最后

希望我的总结能给大家带来些许帮助，也希望和大家一起学以致用，一起成长。最后，万分感谢扔老师的小册，强势安利《git原理详解与实用指南》，认准扔物线。