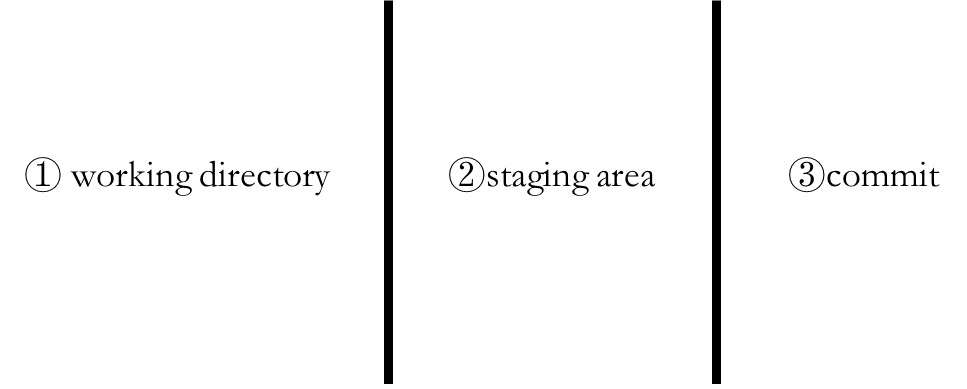
#### 基础



1. 文件（包括目录文件）是变化元；
2. 3个区域逻辑上是分离的（**理解git的核心，git其实很简单，不要被各种提示所迷惑**），通过竖线两边之间的对比得到“变化”：modified、deleted或者new；
3. 抓住“3个区域逻辑上是分离的“一个区域的某个文件发生变化，另一个区域不受影响，但是git可以对比竖线两边，发现这种变化并给出提示（**抓住核心，多用git status多思考提示，来回几次就明白了**）；
4. 将工作目录恢复为某次的commit（如restore，checkout，switch）行为是：工作目录的同名文件会被覆盖，不同名文件会保留（时刻清晰）；
5. 严格遵守原则“区域②是一次完成的snapshot”（**对该区域有什么要时刻清晰**）；
6. **多练手，多看别人的C项目文件组织**；

###### Config

配置等级有system（系统级）、global（用户级）以及local（当前repository级）

git config --global user.name "Haiman Liao" #有空格需要双引号（git config user.name查看用户名）

git config --global use.email haiman.liao@wbstar.com #没有空格不需要双引号

git config --global core.editor "code --wait" #配置VScode为默认编辑器（有空格加双引号）

git config --global core.autocrlf true #Windows的写true，Linux和Mac写input

git config --global -e #查看所有的global配置信息（在家目录下的.gitconfig中）

###### Basic workflow

git init #在当前工作目录下建立一个repository即.git，删除这个则一切历史全无

git add 文件名 #将文件的change添加到staged area，多个文件用空格隔开，可以用.或者\*之类的

git commit #添加说明文件并将staged area的所有东西snapshot到repository

git commit -m "xx" #添加说明信息直接mmit

git rm 文件名 #将文件从staging area和工作目录一起删除（rm只删除区域①的）

git rm --cached A #将文件A只从staging area中删除；

git mv A B #将文件A改名为B（mv只改变区域①的）

git log #查看commit历史

git log --oneline #以简短形式查看commit历史

git log --oneline --all --graph

#以图形化方式显示commit的时间线

git log --oneline --patch A

#显示文件A在commit时间线所发生的改变

git ls-files #查看staged area（区域②）的文件

git status #**比对竖线两边，检测变化，给出相应提示**（别被提示迷惑）

git status -s #以简短的形式查看文件状态

git show HEAD~1 #查看上一次commit所发生的改变，0表示本次commit

git ls-tree HEAD #列出本次commit的所有文件（最前面是文件的mode，然后是blob/tree，分别代表目录文件和regular file）

git show 1c2356 #结合上一条命令使用，其中1c2356是上一条命令列出的文件标识符（文件ID）前面的几个字符，该命令可以列出对应文件的内容，如果是目录文件则列出其包含的文件；

###### Tag

git tag v1.0 1c2356 #给commit ID为1c2356（只需要实际ID前面6个字符）打上标签，相当于给该commit起一个名字（如果不指明commit ID，那么默认是HEAD~0），之后用git log --oneline查看便可看到对应的commit前面有tag（在打tag之前，可以先使用git log --oneline找到commit的ID）；之后用到该commit的地方可以用它的tag代替，不需要再输入其ID，比如：

1. git show v0.0
2. git ls-tre v0.0

git tag -a [commit ID] -m "message"

#给commit打上有注释的tag

git tag #列出所有的tag

git tag -n #列出所有的tag以及它们的注释（没有用-a的tag的注释是commit的注释，但只显示第一行）

git tag -d v1.0 #删除标签

###### checkout

git checkout [commit]

#commit可以是HEAD，HEAD~x，ID或者tag

###### Restore and reverse

git restore --staged A #区域②的文件A变回上一次commit状态（其他区域不变）：不想commit该变化

git retore A #区域①文件A变回上一次add状态：不想add该变化）

git restore --source=HEAD~1 A #区域①文件A还原为区域③commit过的文件A（其他区域不变）

###### Ignore file

建立.gitignore文件，里面添加文件（包括目录文件）

bin/ #忽略文件夹

file1.txt #忽略具体文件

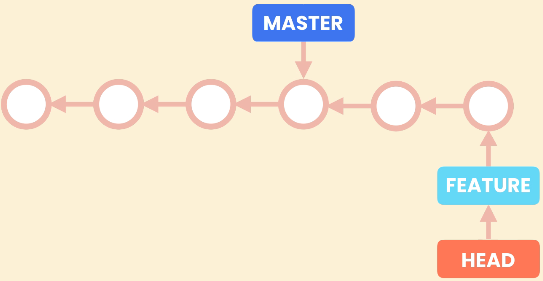
main.\* #忽略某类模式文件

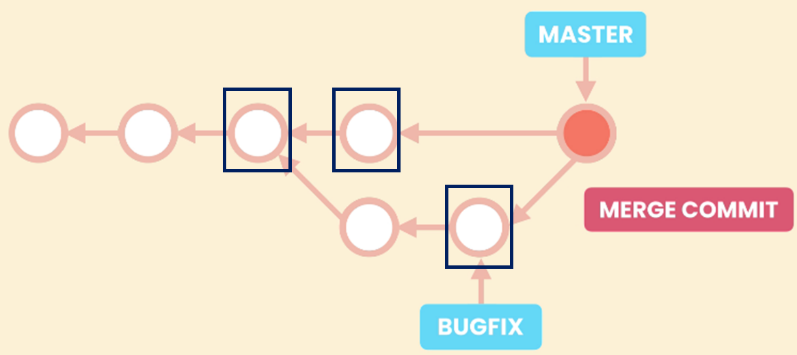
（以上忽略元是文件）

**说明：**

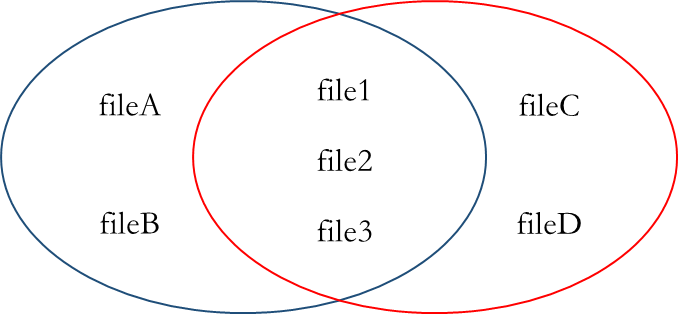
1. 忽略的是工作目录中的文件，add的时候不会add里面指定的文件；
2. 如果staging area存在文件A，.gitignore也登记了文件A（在写进.gitignore之前已经add A了），那么git add的时候是不会忽略A的，此时需要将A只从staging area删除（git rm显然不行），用到命令：
   1. git rm --cached A，如果A是非空目录文件则git rm --cached -r A（r是recursively）；
3. .gitignore本身不会被忽略，另外再强调一遍：忽略是指忽略工作目录中的文件，git add命令会用到.gitignore文件信息；

#### 进阶





1. 前面的基础是3个区域的对比，进阶的基础是commit线（时间线，分支线）即上图：白圈代表commit，最左边是第一次的commit，commit之间的箭头表示“可以根据本次commit，reference到上一次的commit”；
2. 在上图的基础上，如果master分支有新的commit，那么commit线便会出现分叉；
3. merging原理：
   1. 如果是第一张图，显然直接将master指针移动到feature即可（fast-forward merge），因为master没有新的commit；
   2. 对应第二张则是three-way merge即创建一个新的commit，git通过比对3个黑框的commit让用户决定如何生成新的commit；
4. commit之间是独立的，两个commit的文件可以是完全不同的，“commit是基于上一次commit”的认知是错的（那个箭头只代表“可以根据本次commit索引到上一次commit”），每一次commit的内容只和staging area有关（不认清这一点，你会对merging感到困难，特别是three-way merging）
   1. 对比fast-forward mergd和none fast-forward merge便可理解：虽然同为线性commit，但是后者将好几个commit进行比对合并（各个commit的内容从原理上是独立的），两种merge的结果会不同，前者merge的结果就是最后一次commit，后者是最后一次commit和其他commit的结合（比如某个文件的内容merge的时候取其他commit，当然如果所有文件的内容都取最后一次commit，那么两种commit结果上是一样，但原理上仍旧不同，“看视频，多理解”）
5. merging的核心就是“基于文件集（2次或者3次commit）”的比对
6. 将出现分叉的commit进行merge的核心是比对各个commit里面的文件，类似于集合的相加，把文件看成集合的元素，合并后的commit拥有各个commit的所有文件：{A, B, C} + {W, E, A} = {A, B, C, W, E}，然后对于同名文件A（文件一般是目录文件和regular file）再进行内容上的比对，发现conflict，让用户取舍；
   1. **继续探讨6：**git的merge是如何进行比对的，规则是怎么样，什么情况才是conflict？
      1. 同一行的代码在不同分支不一样；
      2. 一个文件在一个分支中存在，但在另一个分支被删除；
      3. 同一个文件在不同分支内容不一样（其实包含ii）；
7. 每个branch的3个区域是独立，新创建分支的共享之前的commit历史（看上面的图示），第一次切换到新分支，它的working directory和staging area内容继承上一个分支环境的；之后就是独立的了，对于working directory，切换之前需要将被tracked的文件变动（通过区域1和区域2）commit或者stash



红色分支和蓝色分支

###### Branching and merging

**意义（软件开发方式）**：保持master的稳定，以随时可以发布新版本，然后如果要拓展新功能，则新开一个分支（branching），在该分支下添加新功能，测试新功能，确保新功能可以稳定使用之后，将其合并（merging）到master（master要确保是稳定的）

1. 英语词典开发基本款（可以用）用一条master分支开发；
2. 出基本款之后，添加新功能（如文章收藏功能）用branching开发，新功能测试稳定后合并到master，变成新的基本款；

git branch xxx #创建分支xxx（此时并没有转到所创建的分支），git status可以查看当前所在的分支（别commit到错误的分支）；刚建立新的分支马上切换到该分支，该分支的工作目录，staging area以及之前所有的commit都是共享的（看第一张图）；

git switch xxx #切换到分支xxx，该命令会把工作目录的内容，还原为目标分支最后一次commit，工作目录原有的同名文件会被覆盖（因此switch之前需要把所有的tracked文件commit——通过比对区域①和区域②的文件，再用一下git status -s便可以理解什么叫tracked文件）：

1. staging area的内容不变，因为该区域是要转换到的目标分支的staging area，它不变是合逻辑的，它的内容被保持记录在.git中，它不变是合逻辑的，它是目标分支的staging area（千万别混乱：每个分支都有各自的“3个区域”）；
2. 把所有的tracked文件commit或者stash，git拒绝switch；
3. 那些untracked文件要么是新文件（不重名，不担心被覆盖），要么是tracked过但后来又将其从staged area中移除变成untracked（这相当于告诉git，我不再关系该文件的版本控制了，所以此时可能会重名覆盖，但是没有关系）；
4. **切换分支的表现：“还原原来的工作环境”+看提示（D file1或者M file1等，这些提示表明相关改动会带到目标分支）+（多用git ls-files以及ls，多思考，会熟悉并适应的，不用过多纠结）**；
   1. 重名覆盖（覆盖改动的重名文件，覆盖untracked的重名文件）；
   2. 删除不恢复（在工作目录把某一个文件删了，那么在switch回来的时候是不会恢复的）
   3. “最后一次commit + 转换提示（D或M或A）可以得出工作目录和暂存区的情况”，**看提示**，多用ls，git ls-files，多思考；

git switch -C xxx #将以上两步合并

git branch #列出所有的branch以及目前所在的branch；

git branch -m xxx yyy #重命名分支xxx为yyy

git branch -d xxx #删除分支

git branch -D xxx #强制删除分支（如果无视没有fully merged的警告）

git branch --merged #列出已经merge到当前分支（转到master分支下）的分支

git branch --no-merged #列出还没有和当前分支（转到master分支下）merge的分支

git stash push [-a] -m "new stash"

#将工作目录的所作的修改进行stash（不包括??即untracked文件，如果要包括则加上参数-a）

git stash list #列出所有的stash

git stash show 1 #将序号1的stash发生的改变列出（通过git stash list可以查看到stash序号）；

git stash apply 1 #将stash 1所保存的工作目录的修改还原到工作目录当中（多敲命令多思考）；

git stash drop 1 #删除stash 1；

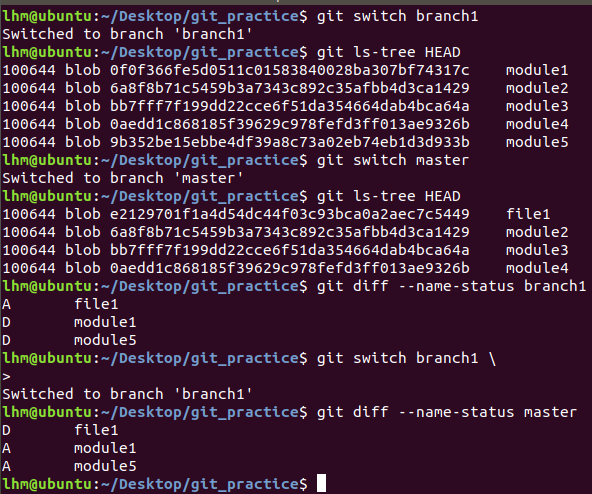
git stash drop clear #删除所有的stash

git log [master..]xxx #列出出现在分支xxx但没有在分支master的commit；

git diff [master..]xxx #比对master分支最新commit和xxx分支最新commit（格式也是以文件为单位比对）；

git diff --name-status [master..]xxx

#列出“如果分支要和master分支合并，受影响的文件状态”，类似于git status -s，git status -s是比对3个区域给出A（add），M（modified），N（new）等提示，本命令是比对两个分支最后一次commit里面的文件信息，给出A，M等提示（用过一次就明白了）；



git merge xxx #Fast-forward合并（以上都是指与master合并，所以需要先switch到master分支）；

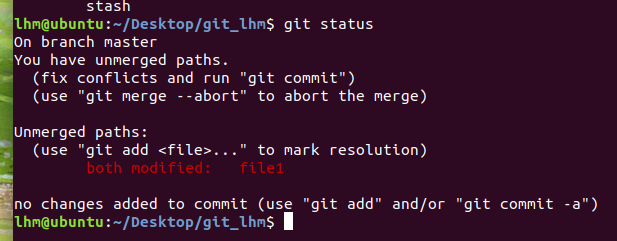
git merge --no-ff xxx #不用master合并

（以上2种方式是基于线性commit，合并的结果是xxx分支最后一次commit，它们区别是commit线不同）

（接上：它们不检查冲突，即使是第二种，它们合并的结果最终都是xxx分支的最后一次commit）

非线性的merge也是用命令git merge xxx

猜想：同一个文件内容不同会有merge conflict；



处在merge状态下（必须要用abort退出，否则在该状态下执行无关命令可能会出错，可以执行add和commit，因为本质该状态还是在master分支下）

文件名相同，内容不同（合并冲突）；合并冲突git会有提示（同名文件在不同分支内容不同，文件在一个分支被删除，在另一个分支没有），正常git的合并逻辑是“分支文件一起聚合”，然后可能会出现以上两种冲突，merge时若出现冲突，git会停留在merge状态，让用户自己解决（对于冲突1，把两个分支的内容都保留，让用户取舍；对于冲突2，保留被删除的文件在目录下，让用户取舍；如果OK用户便可以add并commit完成merge，并退出merge状态，用命令git merge --abort回到clean状态）；

git reset --hard HEAD~1 #将HEAD指针指向新的commit，同时新commit也替换掉staged area和working directory（不是仅同名覆盖，而是所有tracked的都被新commit snapshot代替）；（git switch也可以理解为commit“替换掉”working directory的tracked files）

git reset --mixed HEAD~1 #将HEAD指向新commit，新commit只替换staged area；

git reset --soft HEAD~1 #只将HEAD指向新commit；

git revert -m 1 HEAD #提交新的commit，新commit和当前commit（HEAD）的上一个parent commit一样（parent commit指的是按master分支算的parent，因为当前在master分支下）

reset是本地仓库时可以做的，因为转移HEAD指向可能会产生不可到达（除非用commit ID）的commit，git一般会定时检查并删除这些不可到达的commit，相当于修改了历史，所以在与他人合作时不能使用这种方式来undo一个merge，应该用revert，revert是在当前commit之上提交一个和要undo到的步骤的一样的commit；

git restore --source=mybranch -- xxx

#（当前在master分支下）把分支mybranch下文件xxx的最新版本restore到working directory下；

###### Github

前面讲的是本地repository的操作，而GitHub的核心就是管理远程的repository（之前的commit，.git，.gitignore等概念都是一样的）

git clone是把远程仓库复制到当前目录下（新建一个远程仓库的文件夹，当前目录不需要是git仓库）；而git pull是将远程仓库的commit与本地仓库的 commit进行merge（git pull的当前目录是一个git仓库）；两个命令都会得到origin/main，因此它们也都可以push到远程（origin/main）（可以理解为关于远程仓库的网络通达信息被存储在.git文件夹中，用git remote -v命令可以看到这些信息）；

git clone [https / ssh] #将远程仓库复制到本地

git clone [https / ssh] [local name]

#将远程仓库复制到本地并将其重命名（重命名一般没必要）

git fetch #远程仓库origin/main有新的commit，本地仓库的origin/main是不同步的，要想同步可以使用该命令（同步之后仍然不能在本地的origin/main分支提交commit，只能在本地的master提交commit，或者将master和origin/main分支合并）

git pull #相当于git fetch + git merge

###### Rewriting history