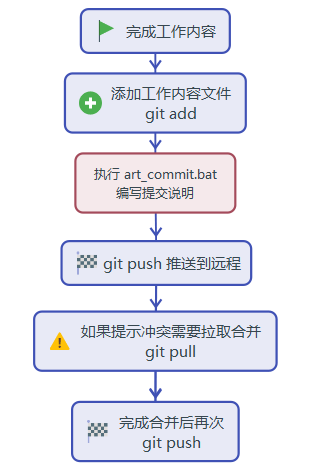
**Git 词典**

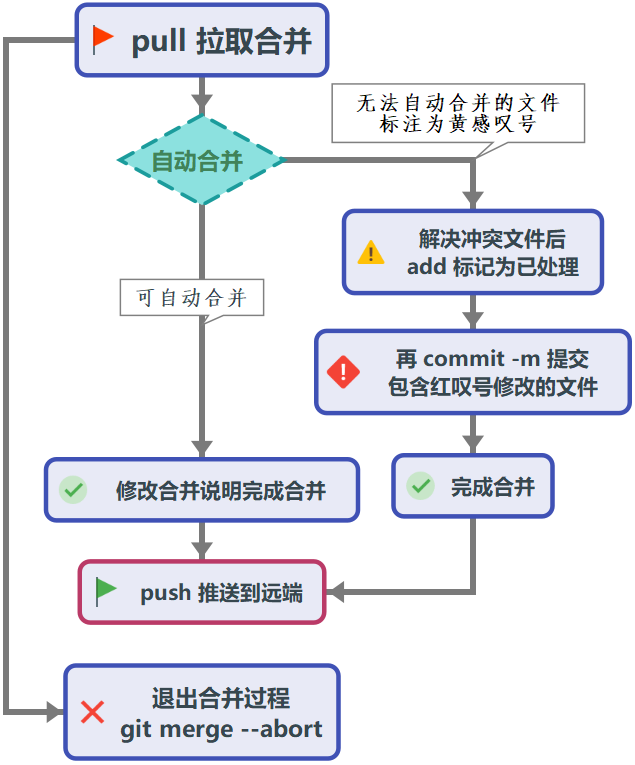
**流程图**

**★二之国项目提交流程**

务必使用 art\_commit.bat 可以自动记录辅助信息及版本号。

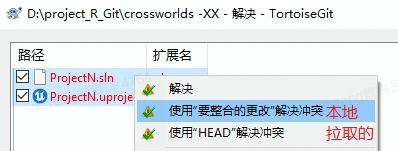


**★拉取合并推送上传流程**



**通过TortoiseGit解决冲突**

选择要保留的版本



**Git之美术操作简书**

**命令格式说明：**

红色字部分表示命令重点主体，<>角括号中显示了用户要填入的内容类型，#井号后面跟着的是此段命令的注解说明，▲标题三角形标注的是比较常用的方法，★流程图。

**快捷指令参考**

需执行批处理文件进行一次性设置，下载后执行：GitSet\_utf-8.bat

GitSet\_utf-8.bat

|  |
| --- |
| git st #status快捷查看当前版本状态 git dec #查看各提交，本地与远程所指位置（常用）后面追加-3只显示最近3个提交 git lsf #列出各提交包含的文件内容（紧凑）后面追加-3只显示最近3个提交，以此类推 git lsff #列出各提交的文件内容及统计（详细） git lsff HEAD^ #列出最后第二次提交以下所有提交文件内容，HEAD^^每加一个“^”向后移一个版本 git cus #自定义显示：提交ID | 提交说明 | 提交时间 | 提交作者 |

|  |
| --- |
| git lsff d1463cb0bc -1 #查看指定提交ID的文件内容，后面追加的数值可以大于1 |

**克隆指定分支创建本地分支(clone)**

第一个<分支>填写远程分支名，不要加origin/，后面填入本地创建的<项目文件夹名>

|  |
| --- |
| git clone -b <分支> https://github.com/victools.git <文件夹名> |

**本地分支与远程分支建立跟踪**

|  |
| --- |
| git branch --set-upstream-to=origin/<branch> <branch> |

**▲添加准备提交的文件(add)**

|  |
| --- |
| git add <file>… #可以使用通配符\* git add . #添加所有文件包括未跟踪的 git add -u #只添加跟踪的文件 |

**▲添加跟踪的并提交说明(commit)**

直接添加并提交，省去add步骤

|  |
| --- |
| git commit -m 'message' #提交说明 git commit -am 'message' #添加提交所有文件(包含未跟踪的)并写说明(省略add步骤) git commit -u -am 'message' #使用-u只添加跟踪的 |

**▲修改最后一次提交(--amend)**

**如果有文件需要修改或添加：**

修改添加文件并git add添加到缓存区后，然后再执行命令git commit --amend修改提交信息。

|  |
| --- |
| git commit --amend -m 'new-message' |

**▲删除缓存区add的反向操作(restore\reset)**

|  |
| --- |
| git reset HEAD <files> #可以使用通配符“\*” git restore --staged <file> #删除缓存区 git reset HEAD \*Pro\* #unstage文件名包含“Pro”的文件 git reset --keep #清空缓存区 |

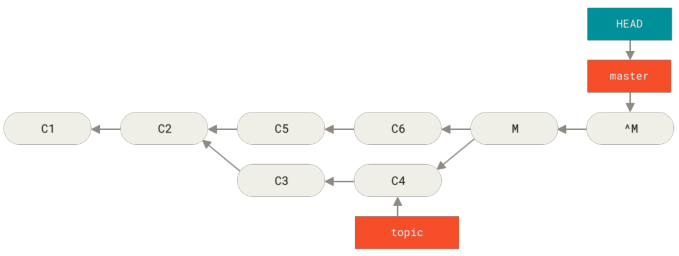
**▲撤销工作目录修改(restore)**

|  |
| --- |
| git restore <file> #回到最后一次add的版本，如果缓存区中没有则回到最后一次commit的 |

**还原提交(revert)**

|  |
| --- |
| git revert -m 1 HEAD #-m 1 标记指出 “mainline” 需要被保留下来的父结点 |

当你引入一个合并到 HEAD（git merge topic），新提交有两个父结点：第一个是 HEAD（C6），第二个是将要合并入分支的最新提交（C4）。 在本例中，我们想要撤消所有由父结点 #2（C4）合并引入的修改，同时保留从父结点 #1（C6）开始的所有内容。



**▲撤销合并(reset)**

|  |
| --- |
| git reset --hard HEAD~ #移动分支到合并之前的历史(撤销工作目录) git reset --soft <SHA-1> #撤销到缓存区(add的文件所在位置) |

**强制还原本地变更(fetch)**

|  |
| --- |
| git fetch --all #拉取全部 git reset --hard origin/<分支名> #撤销工作目录未提交的修改，从远程分支还原 git pull #再拉取合并到工作目录 |

|  |
| --- |
| git reset --hard HEAD #撤销工作目录所有未提交的修改 git reset --hard [<SHA-1>|留空提取最新本地提交] #还原commit历史到工作目录 |

**保留本地版本(merge)**

|  |
| --- |
| git merge --abort #退出合并 git reset --merge # |

**▲恢复本地库记录到缓存区(reset)**

当commit提交了错误的文件时，做撤回的操作，撤回后再对错误的文件（git restore --staged <file>）从缓存区移除。

|  |
| --- |
| git reset --soft HEAD^ #恢复最近一次commit内容到缓存区（便于重新修改提交） |

**停止跟踪文件(rm --cached)**

|  |
| --- |
| git rm --cached <file> #停止跟踪但不删除文件，之后直接commit提交文件既不跟踪状态，这之前不能add |

**▲暂存工作目录中的修改恢复工作目录(stash)**

在 pull 拉取合并操作前使用，解决工作目录中文件冲突，拉取合并不会影响未跟踪的文件，所以可以不用暂存未跟踪的文件。

|  |
| --- |
| git stash #暂存跟踪并修改的文件 git stash -u #暂存包含未跟踪的文件 |

查看暂存的记录文件内容

|  |
| --- |
| git stash list #列出所有暂存记录 git stash show stash@{0} |

**切换、复制分支(switch)**

可以先用git branch查看本地所有分支（切换分支前最好先做stash暂存本地更改，避免不同分支的文件冲突）

|  |
| --- |
| git switch <分支名> #切换分支 git switch -c <new-branch> #复制当前分支并切换分支 |

**▲还原文件、切换\创建分支(checkout)**

|  |
| --- |
| git checkout <提交ID号> <files> #还原某提交的文件 git checkout <branch> #切换分支，-f强行切换 git checkout <branch> <file> #拉取指定分支中的文件，也可以是远程分支origin/<branch> git checkout -- #列出所有变更的内容 git checkout -- <file> #丢弃工作目录中的变更，替换缓存区的内容 |

在项目刚克隆下来时通常本地只有一个master分支，需要在本地创建与远程关联的所需分支，可以如下操作：

|  |
| --- |
| git checkout -b <new-branch> origin/<branch> #拉取远程分支在本地创建新跟踪分支 git checkout -t origin/<branch> #或者无需填名字建立一个和远程分支名字一样的分支 |

**查看提交的文件(show --raw、log --stat)**

查看指定提交的文件列表：

|  |
| --- |
| git show --raw <SHA1> #也可指定显示条目数-2 git show --raw #省略后面id查看最后一次提交 git log -3 --stat #列出文件，显示修改添加删除文件数 git log --name-only -1 #查看最近1条提交的文件 |

**查看指定文件的提交历史**

|  |
| --- |
| git log -p <file> git log -p <file> **>** out.txt #一个“>”将内容输出到out.txt文件里便于查看，两个“>>”累加方式写入。 |

**查看操作历史(reflog)**

可以用git show HEAD@{5}通过指针号查看提交记录

可以使用git reset --hard <SHA-1>还原历史到指定提交对象（注意当前工作目录的备份）

|  |
| --- |
| git reflog |

**★查看各分支的区别(show-branch)**

|  |
| --- |
| git show-branch #列出所有本地分支的最后提交信息，后面加 -a 查看远程所有分支 |



**▲清理工作目录(clean)**

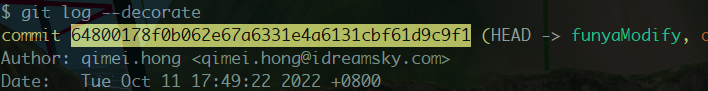
清除未跟踪(Untracked)的文件，（此操作直接删除不可还原）：

|  |
| --- |
| git clean -d #移除未跟踪及空子目录-f强制-n演练 |

**☆将提交应用到其他分支（cherry-pick）**

该操作用于将当前分支的提交内容应用到其他分支中，几个操作步骤如下

|  |
| --- |
| git log --decorate #查看提交的SHA-1值并复制 git checkout <其他分支> #切换到要迁移到的目标分支 git pull #拉取该分支更新 git cherry-pick <粘贴SHA-1值> #将指定的<SHA-1>提交应用到当前分支 git push #校验完成后再推送到远程 |

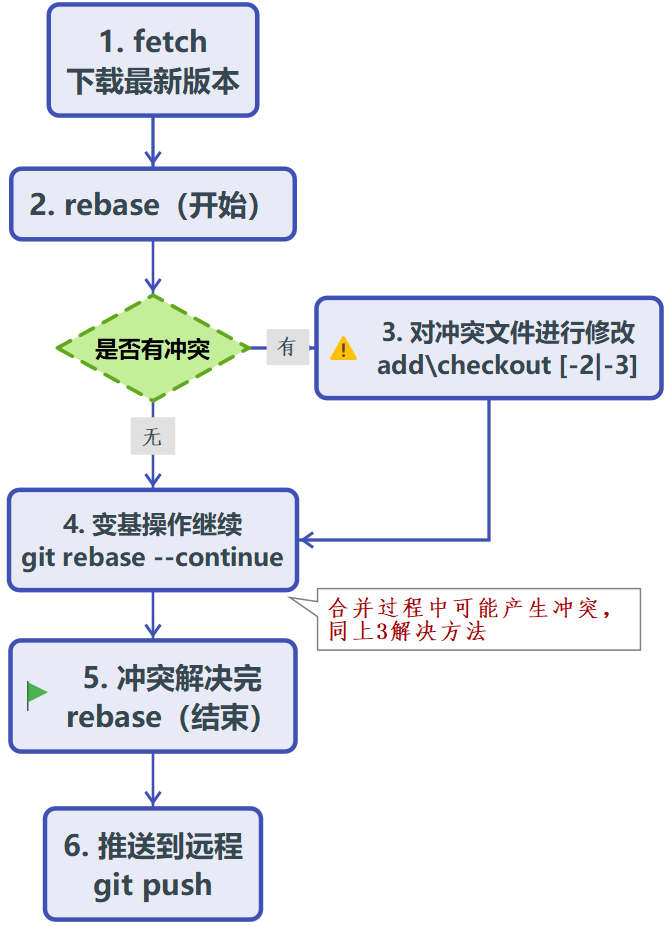


**创建远程分支**

|  |
| --- |
| git checkout -b my-test //在当前分支下创建my-test的本地分支分支 git push origin my-test //将my-test分支推送到远程 git branch --set-upstream-to=origin/my-test //将本地分支my-test关联到远程分支my-test上 |

**★变基流程(rebase)**

|  |
| --- |
| git fetch #第1步拉取远程最新版本到本地 git rebase <branch> #第2步操作冲突的文件选择使用哪边的来源<本地>或<远程> git checkout -2 <files> #-2使用当前版本，-3使用来源版本 git add <解决完冲突的文件> #第3步操作 git rebase --continue #第4步继续完成变基操作 |



**创建新储存库(remote)**

|  |
| --- |
| **#create a new repository on the command line** echo "# SECTOR-7" >> README.md git init git add README.md git commit -m "first commit" git branch -M main git remote add origin https://github.com/vic315/SECTOR-7.git git push -u origin main  **#或者 推送现有储存库 or push an existing repository from the command line** git remote add origin https://github.com/vic315/SECTOR-7.git git branch -M main git push -u origin main |

**查看所有配置包括自定义命令(config --list)**

|  |
| --- |
| git config --list git config --list --show-origin #查看config目录 |

**其他操作**

**删除文件**

|  |
| --- |
| rm <file> |

**内容输出到文件 >out.txt**

在任何命令后面添加>文件.txt, 将输出内容存入指定文件中

|  |
| --- |
| git dec **>out.txt** # > 全新替换写入到文件 git st **>>out.txt** # >> 以追加方式写入 |

|  |
| --- |
| **THE END** |

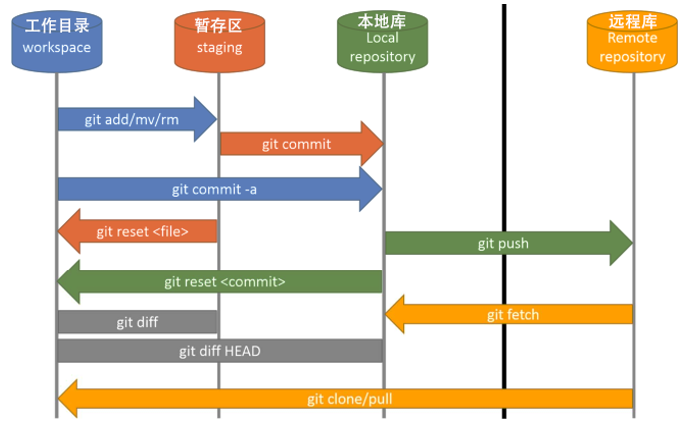
***以下内容未完成编辑 ↓***

**前言：**

该词典的前面部分介绍Git系统的各部分知识，通过这部分知识了解其工作原理，对后面灵活使用git命令很有帮助，之后**词典**部分是以git命令的首字母排序，进行命令功能说明运用方法及示例的展示，方便查询使用。

**Git地图**

下图中显示了Git各个模块的位置，以及各命令控制跟踪文件的去向。



|  |
| --- |
| **帮助中的语法：**  [-i] 中括号里面是命令可用的参数，根据参数说明使用  参数中“|”表示或，使用时两个参数只能选一个，如：  [-x | -X]  <…> 角括号中的表示用户输入的内容  多数命令可用命令参数：  -h #查看命令使用帮助  -i --interractive #交互式终端  -f --force #强制操作  -n --dry-run #操作预览  HEAD #当前用户所在版本  因此查看当前提交版本内容可以用：  git show --raw HEAD  HEAD^ #^（脱字符）表示父级，HEAD的父级提交，也就是上一次  HEAD^^ #以此类推，两个^表示上上次提交 |

●查看当前版本状态（不知道干啥时就用它）

|  |
| --- |
| git status  git status -s #简洁模式 |

●查看远程仓库地址

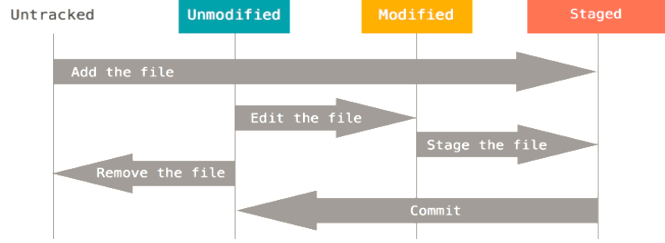
|  |
| --- |
| git remote -v |

●还原本地版本为远程最新

|  |
| --- |
| git reset --hard #恢复到上一本地commit版本  git pull origin develop #拉取指定版本库分支覆盖本地 |

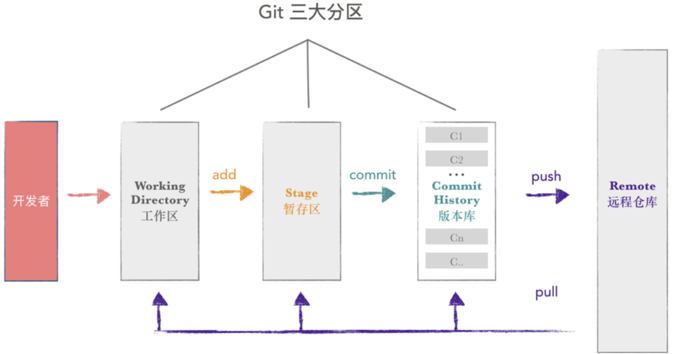
◆1下图展示了**文件**在本地管理中各操作的去向和状态：

未跟踪的 无修改状态 **修改状态** **缓存区**



上图修改状态下的Stage the file既是add。

◆2 Git三大分区



**◇3 Git三棵树的思维框架**

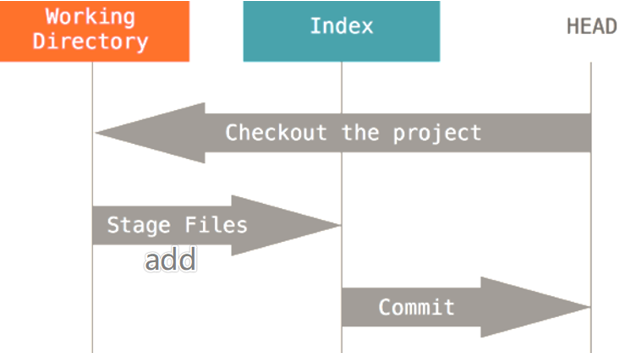
通过操作来管理三棵树，这里“树”代表文件的集合。

|  |  |
| --- | --- |
| **树** | **用途** |
| **HEAD** | 上次提交的快照，下次提交的父节点 |
| **Index** | 预期的下一次提交的快照 |
| **Working Directory** | 工作目录 |

**工作流程：**

经典的git工作流程是通过操纵这三个区域来以更加连续的状态记录项目快照的。

**工作目录 索引 指针**



**工作目录**

也叫**工作区**，可以把工作目录当做**沙盒**，在修改提交到缓存区并记录到历史之前可以随意更改。

**索引**

是你**预期的下一次提交**，Git的缓存区，通过git  
  
commit将它们转换为树来作新的提交。（通过幕后命令ls-files查看索引）

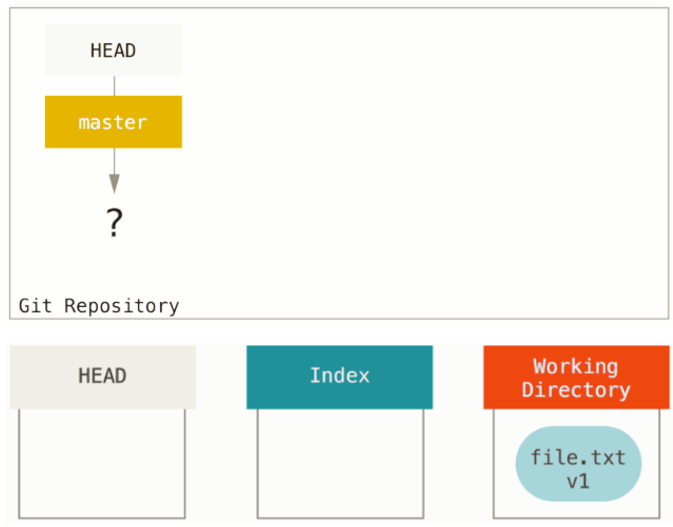
|  |
| --- |
| git ls-files -s |

**HEAD**

是当前分支引用的指针，它总是指向该分支的最后一次提交。可以理解它是**该分支上的最后一次提交**的快照。（通过底层命令cat-file和ls-tree显示HEAD快照的目录列表）

|  |
| --- |
| SQL git cat-file -p HEAD tree daf6913ca6b55a86a13d63567781abb50b035c1b parent 20b8b2c6503208d52ec0e15e7c8c9e2082296f71 author vic.zeng <vic.zeng@idreamsky.com> 1655633403 +0800 committer vic.zeng <vic.zeng@idreamsky.com> 1655633403 +0800 git ls-tree -r HEAD 100644 blob 2b90b7cdf14d86c313e13ceb9f5719b472d96dc3 "git\name.smmx" 100644 blob de395e44b5fe8a8c5437978173dc9fadfd7ce7df "\name.docx" |

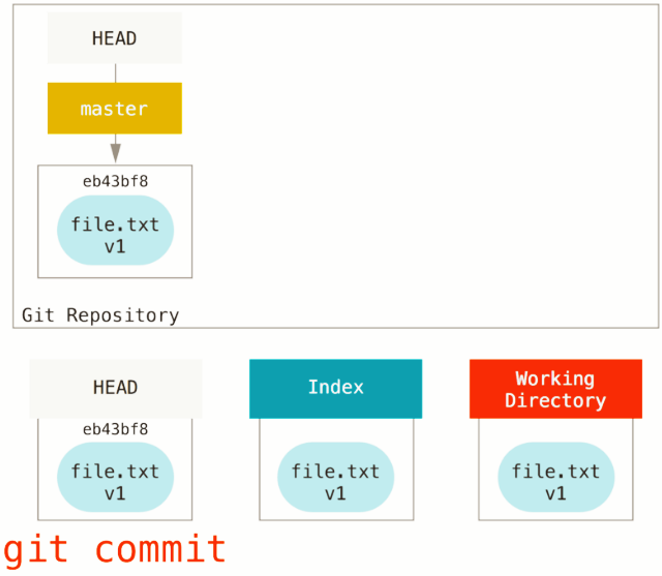
下面我们以图形方式来观察一下三棵树的工作流程：



此时只有工作目录有内容，我们要提交这个文件，用git add来获取工作目录中的内容复制到索引中。



接着运行git commit取得索引中的内容并保存为一个永久的快照，创建一个指向该快照的提交对象，最后更新当前分支指向本次提交。



此时运行git status会发现没有任何改动，因为三棵树完全相同。

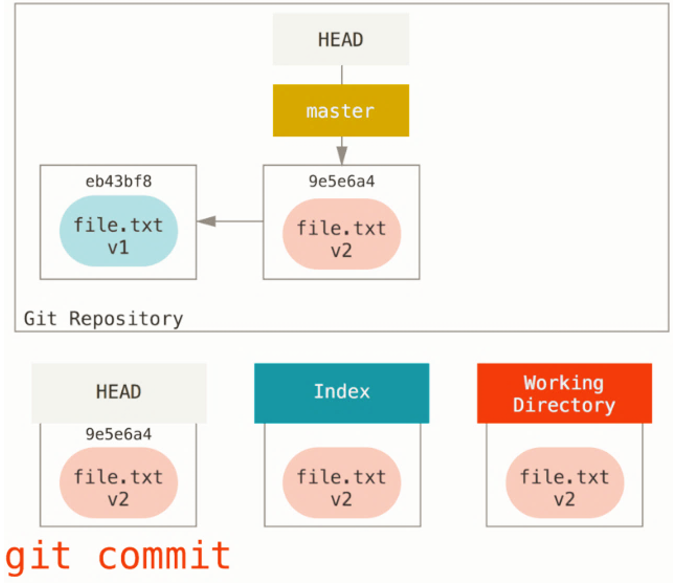
现在对文件进行修改然后提交它，修改后的文件为v2版本，并标记为红色，



接着我们运行git add来将它缓存到索引中，



现在预期的下一次提交与上次提交不同，同样运行git commit来完成提交，

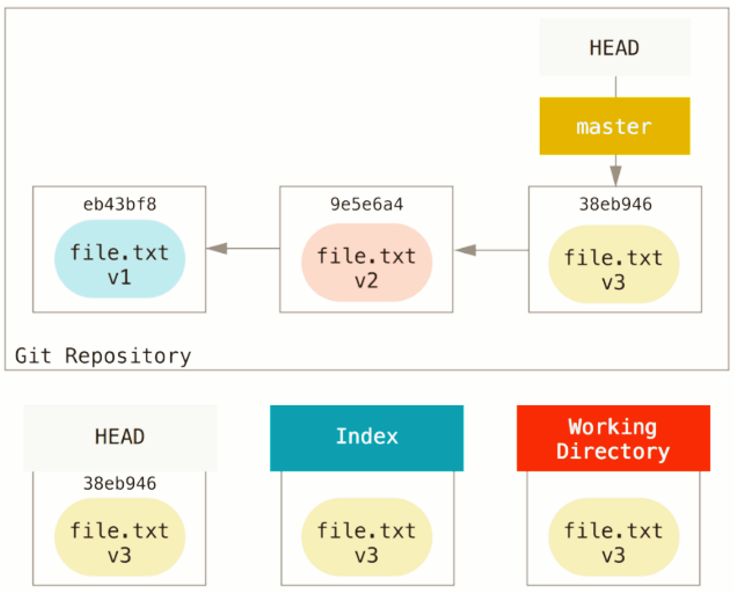


切换分支或克隆的过程也类似，当检出一个分支时会修改**HEAD**指向新的分支引用，将**索引**填充为该次提交的快照，然后将**索引**的内容复制到**工作目录**。

◆3 END

**◇4 重置的作用 reset**

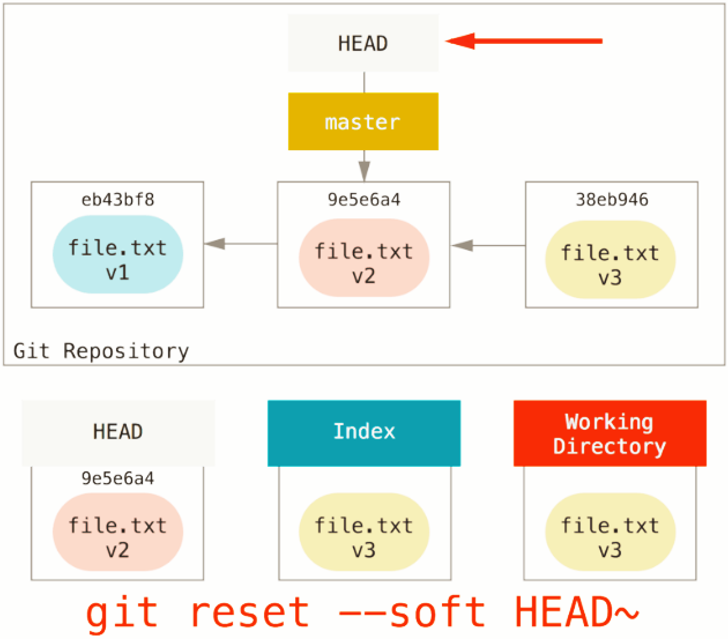
假设我们再次修改file.txt文件并第三次提交它，现在历史看起来是这样的：



reset用简单的方式直接操纵三棵树，以下三个基本操作（移动HEAD、更新索引--mixed、更新工作目录--hard）

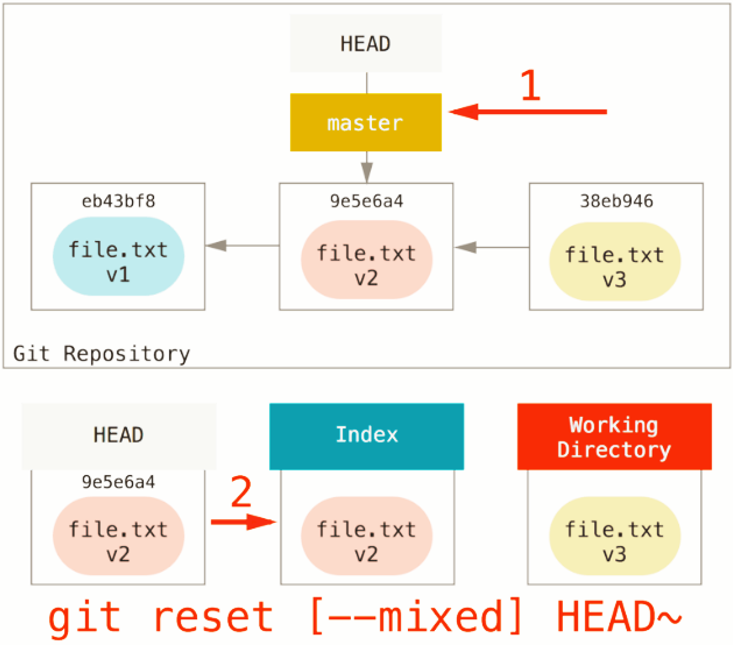
第一步：移动HEAD

reset做的第一件事是移动HEAD的指向，这与改变HEAD不同（checkout做的），运行git  
  
reset 9e5e6a4将会使master指向9e5e6a4索引，它**本质上是撤销了上一次commit命令**，当你reset回HEAD~（HEAD的父结点）时，就是把该分支移动会上一位置，而不会改变索引和工作目录，现在可以更新索引并再次git commit来完成git commit --amend要做的事情了（修改最后一次提交）。

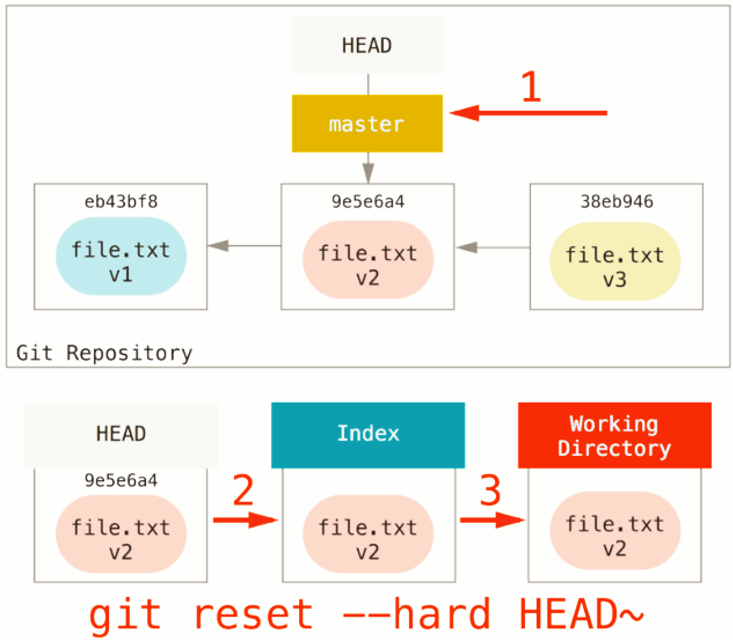


第二步：更新索引（--mixed）

现在运行status的话会看到新的HEAD以绿色标出它和索引的区别。接下来reset会用HEAD指向的当前快照的内容来更新索引。



第三步：更新工作目录（--hard）



将索引更新到工作目录，必须注意，--hard标记是reset命令唯一危险的用法，强制覆盖工作区后三棵树将保持一致，如果git数据库中有v3版本，可以通过reflog来找回。

**Git分支**

Git分支的本质：一个指向某一系列提交之首的指针或引用。

**Git引用**

引用（references，简写为refs）

可以在.git/refs目录下找到这类含有SHA-1值（识别码、哈希值）的文件，如果想查看一个提交（例如1a410e，SHA-1值的前几位）开始往前的历史，可以运行git log 1a410e这样的命令来显示历史：

|  |
| --- |
| git log 1a410e |

若要创建一个新的引用来帮助记忆最新提交所在位置，从技术上可以做如下操作：

|  |
| --- |
| echo 1a410efbd13591db07496601ebc7a059dd55cfe9 > .git/refs/heads/master |

现在，就可以在Git命令中使用这个创建的新引用来代替SHA-1值了：

|  |
| --- |
| git log –pretty=oneline master 1a410efbd13591db07496601ebc7a059dd55cfe9 third commit cac0cab538b970a37ea1e769cbbde608743bc96d second commit fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit |

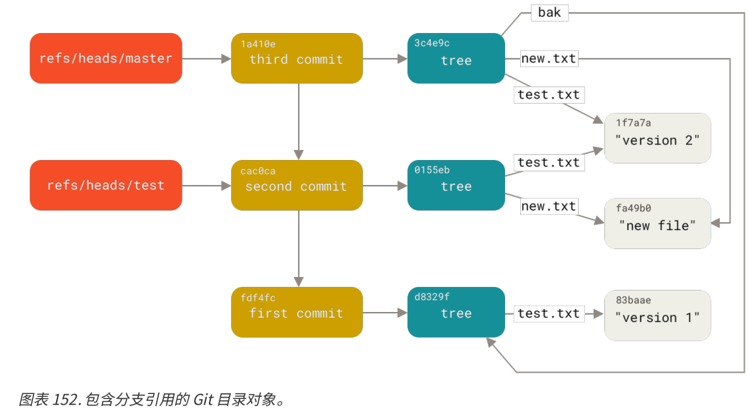
不提倡直接编辑引用文件。如果想更新某个引用，使用更安全的命令update-ref来完成：

|  |
| --- |
| git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9 |

若想在第二个提交上创建一个分支如下操作：

|  |
| --- |
| git update-ref refs/heads/test cac0ca git log –pretty=oneline test cac0cab538b970a37ea1e769cbbde608743bc96d second commit fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit |

至此，我们Git数据库从概念上看如下：



**HEAD引用**

HEAD文件通常是一个符号引用（symbolic reference），指向目前所在分支。

查看HEAD文件内容如下：

|  |
| --- |
| $ cat .git/HEAD  ref: refs/heads/**develop** |

如果执行git checkout test，Git会像这样更新HEAD

|  |
| --- |
| $ cat .git/HEAD  ref: refs/heads/**test** |

当执行git commit时，该命令会创建一个提交对象，并用HEAD文件中那个引用所指向的SHA-1值设置其父提交字段。可以借助git symbolic-ref命令查看HEAD引用对应的值：

|  |
| --- |
| $ git symbolic-ref HEAD  refs/heads/**master** |

该命令同样可以设置HEAD引用的值：

|  |
| --- |
| $ git symbolic-ref HEAD refs/heads/**test**  $ cat .git/HEAD  ref: refs/heads/**test** |

**远程引用**

远程引用（remote reference），如果添加了一个远程版本库并执行过推送，Git会记录最近一次推送时每一个分支对应的值，并保存在refs/remotes目录下。

**查看远程版本库分支对应的SHA-1值：**

|  |
| --- |
| $ cat .git/resf/remotes/origin/develop |

**祖先引用 (脱字符^)（波浪号~）**

尾部加上^，会引用上一个提交，可以加多个脱字符，显示上上一个提交，^2后面加数字代表第几个父级：

|  |
| --- |
| git show HEAD^^ #第二个父提交  git show HEAD^3 #第三个父提交 |

**i**在Windows的cmd上转义脱字符不能直接输入：

|  |
| --- |
| git show HEAD^^ #单个双写  git show “HEAD^” #或加双引号 |

另一种知名祖先提交的方法是~（波浪号），同样是指向父提交，因此HEAD~与HEAD^是等价的，区别在于后面加数字的时候，HEAD~2代表“第一父提交的第一父提交”，也就是“祖父提交”。也可以组合使用两个语法，通过HEAD~3^2来取得之前引用的第二父提交。

**TIPS（技巧）**

提**交技巧：**

|  |
| --- |
| 像错误修复或功能、资源添加之类不同含义的更改，要尽量分开提交，这样可以方便事后从提交历史记录里找出特定的修改内容。 |

提**交信息规范：**

|  |
| --- |
| 提交信息编写很重要，查看他人的修改或自己检查历史记录时可以一目了然。  建议提交讯息格式如下：  <修改的类型>:<修改内容的摘要>  示例：  资源新增：新增家园1号建筑  模型修改：2号建筑外部配件减面优化 |

利**用通配符快速处理文件：**

|  |
| --- |
| 同一批工作内容的文件可以包含相同的主名字命名，在添加此类文件时可通过通配符一次找到所有相关文件。例如：一个建筑的资源包含模型、材质球、贴图等资源，他们的命名参考是：模型命名“Mesh\_MOD\_001”材质球命名“M\_MOD\_001”贴图命名：“T\_MOD\_001”，他们种类不同，但都包含还有MOD\_001名字，那么就可以使用通配符“\*”一次获取他们：“\*MOD\_001”，这里“\*”代替文件名中不相同的部分。 |

**词 典**

**A**

**add （添加到缓存）**

添加文件到缓存区（Stage），若添加未跟踪的文件也可以说开始跟踪一个文件（add命令后可以加一个文件、多个文件、文件夹，文件可以用“\*”通配符替代）

（◆1）

|  |
| --- |
| **用法：**git add  [<-参数>] [--]  <路径名>…  -n 操作预演，列出将要添加的内容  -v 详细模式  -x 包含未跟踪的和gitignor排除的文件  -X 只对gitignor收录的文件  -d 含整个文件夹  -u 只更新跟踪的文件 |

|  |
| --- |
| git add \*.c #跟踪所有后缀c的文件  git add file1 file2 #跟踪多个文件  git add . #添加包含所有变更和新增  git add . -n #[-n]列出将要添加的内容  git add -u #只添加跟踪的文件 |

[-i]交互式操作

**B**

**branch（分支）**

分支操作

创建新分支，没有切换到新分支：

|  |
| --- |
| git branch <newbranch> |

创建新分支，并切换到新分支

|  |
| --- |
| git checkout -b <newbranch> |

查看所有未合并工作的分支

|  |
| --- |
| git branch –no-merged #查看已合并用--merged |

查看本地分支与远程仓库的连接情况

|  |
| --- |
| git branch -vv |

查看本地与远程所有分支

|  |
| --- |
| git branch -a |

**blame（查看文件修改记录）**

查看指定文件的修改记录

|  |
| --- |
| git blame <file> |

**C**

**clone**

克隆远程仓库到本地

|  |
| --- |
| git clone <远程地址> <新文件夹(选填)> |

commit

提交到本地库（◆1、2、3）

|  |
| --- |
| git commit -am 'message' #对变更的文件直接提交信息，省略add步骤 |

**checkout**

用来切换分支，或检出内容到工作目录（◆3）

不同于reset --hard，checkout对工作目录是安全的，它会通过检查来确保不会弄丢更改的文件。

下图中显示，reset会移动HEAD分支的指向，而checkout移动HEAD自身。

|  |
| --- |
| git checkout <branch> #切换分支，-f强行切换 git checkout -- <file> #丢弃工作目录中的更改，拉取缓存区的内容 git checkout <SHA-1> #可将工作目录检出到某个commit提交，HEAD指向该提交。 git checkout **--orphan** <new-branch> #基于当前分支创建一个干净的没有提交历史的新分支 git checkout –3 <file> #还原到远程版本--theirs git checkout –2 <file> #还原到本地库版本--ours git checkout -- <file> #放弃工作目录修改，还原最近一次添加的（缓存区的） |

**clean**

对没有被跟踪tracked或被.gitignor收录的文件进行删除清理（未被add或commit的本地文件）

|  |
| --- |
| **用法：**git clean [-d] [-x | -X] [--] <多路径>…  -n 列出要清除的文件（在操作前最好加-n参数，确认文件无误时再去掉-n执行真正删除操作）  -f 强制操作  -x 除了未跟踪的也包含gitignor排除的文件  -X 只对gitignor收录的文件  -d 含整个文件夹 |

|  |
| --- |
| git clean -n #列出将要删除的未跟踪文件（untracked）  git clean -Xdn #列出gitignor排除的文件 |

**cherry-pick**

将某分支的一个提交中引入的变更作为一个新的提交引入当前分支，从一个分支单独一个或者两个提交而不是合并整个分支。

将提交 e43a6 拉取到 master 分支，先切换到master

|  |
| --- |
| git cherry-pick e43a6 |

下面示例表示采用提交e43a6来自编号1的父分支：

|  |
| --- |
| git cherry-pick -m 1 e43a6 |

代码冲突时cherry-pick会停下，让用户决定如何操作：

--continue

用户解通过修改文件加入缓存区(git add .)决冲突操作后执行git cherry-pick --continue

--abort

放弃合并，回到操作前的样子

--quit

退出cherry-pick，但是不回到操作前的样子。

**cat**

查看文件内容

|  |
| --- |
| cat <file> |

**D**

**diff**

对比各区中的文件修改差异（◆2）

|  |
| --- |
| git diff <file> #工作区vs暂存区  git diff head <file> #工作区vs版本库  git diff --cached <file>#暂存区vs版本库 |

**F**

**fetch**

从远程仓库抓取代码到本地，下载对象和引用，不会修改工作目录中的内容，只会获取数据然后让用户自己合并。也可用在建立远程仓库跟踪git fetch <origin>

图表 32. git fetch 更新你的远程跟踪分支

抓取远程仓库：

|  |
| --- |
| git fetch --all #抓取所有远程仓库  git branch -vv #查看本地分支与远程仓库的连接情况 |

**I**

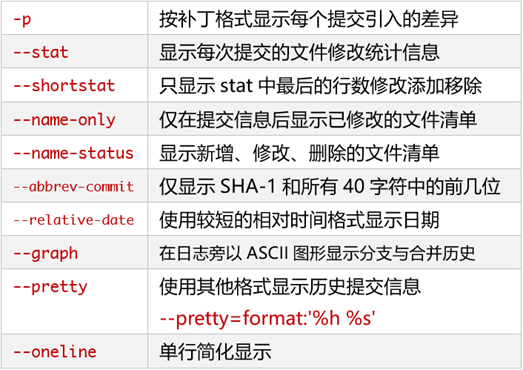
**init**

初始化本地配置文件，可用于本地版本管理。

**L**

**log**

查看提交历史。



为了使信息更加详细，我们可以使用如下命令，以标准

查看分支中的提交历史，如不填分支名则查看当前分支

|  |
| --- |
| git log -–pretty=oneline <master>  1a410efbd13591db07496601ebc7a059dd55cfe9  third commit |

日志格式输出引用日志

|  |
| --- |
| git log -g |

限制性定制输出项

以时间作限制，列出最近2周的所有提交

|  |
| --- |
| git log --since=2.weeks  git log --since=6.days |

例：三周前四月十八日之后未合并的

|  |
| --- |
| git log --oneline --before={3.weeks.ago}  --after={2010-04-18} --no-merges |

使用format定制显示格式：

|  |
| --- |
| git log --pretty=format:"%h %s | %ar | %cn" |



**ls-files**

**M**

**merge**

合并分支。

合并时会尝试智能合并，遇到文本时会在文本中标注当前版本（HEAD）与合并目标分支<branch>的内容，供用户自行编辑保存，并commit提交完成合并过程。

文本冲突时的样子用‘=====’分隔两个分支内容：



|  |  |
| --- | --- |
| -n |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
| --autostash | 自动暂存工作目录修改，并在完成合并后自动pop取出 |

**mv**

是一个便利的命令，用于移到一个文件并且执行‘git add’及在老文件上执行‘git rm’。

运行git mv就相当于运行了下面三条命令：

|  |
| --- |
| mv README.md README  git rm README.md  git add README |

**P**

**pull**

**push**

推送本地提交到远程仓库。

|  |
| --- |
| git push <remote> <branch>  git push origin branch #强制操作添加-f |

第一次推送需要与远程建立链接加-u，后面的推送就可以直接用git push了。

|  |
| --- |
| git push -u origin branch |

删除远程仓库的分支：

|  |
| --- |
| git push origin --delete <branch> |

**R**

**remote**

用于远程仓库的操作。

增加一个本地版本库到现有的 Git 项目，执行如下命令：

|  |
| --- |
| git remote add <origin> </srv/git/project.git>  #删除添加的远程连接地址，origin即是远程仓库名称  git remote remove <origin> |

查看仓库地址信息：

|  |
| --- |
| git remote -v |

显示远程仓库详细信息：

|  |
| --- |
| git remote show <origin> |

**reset（更新索引）**

回滚文件，移动HEAD指向索引的位置，不改变索引和工作目录（◆2、4重置的作用）

|  |
| --- |
| git reset --soft #暂存区->工作区（移动HEAD）  git reset --soft HARD^ #恢复最近一次commit内容到缓存区，版本以上的内容都会放入缓存区，已add状态。  git reset --mixed #版本库->暂存区（更新索引）  git reset --hard #版本库->暂存区->工作区（更新工作区） |

删除缓存区的内容（add的反向操作）

|  |
| --- |
| git reset HEAD <files> #可以使用通配符“\*”  git reset HEAD \*Pro\* #unstage包含“Pro”的文件 |

**reflog**

可以在任何时候执行该命令来了解你曾经做过什么。

|  |
| --- |
| git reflog |

为了使信息更加详细，我们可以使用git log -g

**remove**

（◆1）

**revert（还原）**

还原撤销指定的提交。

|  |
| --- |
| git revert <SHA-1> #撤销某个历史提交，该操作也将作为一个新的提交历史堆栈 |

**restore**

撤消工作区的修改返回到最近一次commit(当前版本库)的版本或者add(缓存区)的版，优先返回缓存区的版本。

撤销工作目录的修改：

|  |
| --- |
| git restore <file>… #回到最后一次add或commit的版本  git restore --staged <file>… #删除暂存区的版本 |

**rm**

用来从工作区或缓存区移除文件的命令。

**rebase**

该操作称为变基，将提交到某一分支上的所有修改都移至另一分支上。

|  |
| --- |
| git rebase --abort #退出变基操作过程 |

(图表 35. 分叉的提交历史)

experiment merge到master的样子：

(图表 36. 通过合并操作来整合分叉的历史)

例子中，检出 experiment 分支，然后将它变基到 master 分支上：

|  |
| --- |
| git checkout experiment //先切换到要变基的分支  git rebase master //再变基到指定基础分支 |

它的原理是首先找到这两个分支（即当前分支experiment、目标基底分支 master） 的最近共同祖先 C2，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，然后将当前分支指向目标基底 C3, 最后以此将之前另存为临时文件的修改依序应用目标基底 C3, 最后以此将之前另存为临时文件的修改依序应用。

(图表37.将C4中的修改变基到C3上)

现在回到 master 分支，进行一次快进合并。

|  |
| --- |
| git checkout master  git merge experiment |

(图表38. master 分支的快进合并)

此时，C4指向的快照就和图36中的C5一模一样了。rebase合并分支使提交历史看起来似一条直线。

深入了解查看progit.pdf中的更有趣的变基例子

**rerere**

显示解决方案的当前状态——开始解决前与解决后的样子。

**S**

**status**

**stash**

暂存当前工作区的文件

|  |
| --- |
| git stash //只暂存跟踪的变更文件  git stash -u //--include-untracked，暂存包含未跟踪的，除.gitignore排除的文件  git stash --all //暂存所有文件包含未跟踪和.gitignore排除的（工作项目勿用） |

**show**

查看某个分支在一定时间前的位置，如下显示昨天master分支的顶端指向哪个提交：

|  |
| --- |
| git show master@{yesterday}  git show @{yesterday} //不指定分支显示当前分支 |

**switch**

切换分支，也可以通过它创建并切换分支，与checkout切换分支的功能相同，但它更为底层，所以如果是本地分支间的切换可以用switch。

|  |
| --- |
| git switch <branch> //切换分支  git switch -c <new-branch> //复制当前分支并切换分支 |

**U**

**附录**

**.gitignore排除跟踪语法**

**排除整个文件夹**

/folder

根据子文件夹的层级数加上通配符如下

\*/\*/\*/.idea

斜杠后紧跟两个连续的星号"\*\*"，表示多级目录

Build/\*\*/\*.ico

**忽略大小写的方法，中括号中写入大小写字母**

**下例中不论是library还是Library都会排除**

/[Ll]ibrary/

/[Tt]emp/

**反排除!**

!/folder/

**使用通配符\*排除所有exe文件**

\*.exe

在使用git开发时，我们往往不需要上传更新某些文件或文件夹，每次要手动取消上传或更新就显得比较麻烦，所以我们可以通过配置.gitignore文件的方法，忽略某些文件(夹)提交、更新。

教程

在项目根目录内新建文本文件，并重命名为.gitignore，该文件语法如下

# 以'#'开始的行，被视为注释.

# 忽略掉所有文件名是 a.txt的文件.

a.txt

# 忽略所有生成的 java文件,

\*.java

# a.java是手工维护的，所以例外.

!a.java

# 忽略所有.o和 .a文件.

\*.[oa]

配置语法：

以斜杠“/”开头表示目录；

以星号“\*”通配多个字符；

以问号“?”通配单个字符

以方括号“[]”包含单个字符的匹配列表；

以叹号“!”表示不忽略(跟踪)匹配到的文件或目录；

常用的规则：

1)/target/ 过滤整个target文件夹

2)\*.html 过滤所有.html文件

3)/doc/doc.doc 过滤某个具体文件

**注意:**

我们不能在windows中直接建立.gitnore文件，因为是点开头，没有文件名，没办法直接在windows目录下直接创建，必须通过右键Git ，按照linux的方式来新建.gitignore文件或者直接在IDE中创建

有时即使创建了.gitignore文件，却依然不能忽略，请检查语法是否有误，若果确定无误，可以在项目根路径下使用Git 以下命令：

|  |
| --- |
| git rm -r --cached . git add . git commit -m 'update .gitignore' |

**vim文本编辑操作指南**

编辑合并信息时会用到。

**未进入编辑模式操作**

移动光标：**H**:← **J**:↓  **K**:↑ **L**:→

**gg**：光标跳到第一行

**shift+g**：光标跳到最后一行

**dg**：全部删除

**shift+V**：当前光标开始配合上下键选择多行

**0**：Home

**$**：End

**x**：删除字

**dd**：删除行

**d0**：删除光标所在行**左边**的内容

**d$**：删除光标所在行**右边**的内容

**u**：Undo撤回操作

**ctrl+r**：Redo

**r**：替换当前字符

**d**3**w**：删除光标右边3个单词

**ciw**：删除当前单词，进行编辑

**cc**：剪切整行

**p**：粘贴剪切的内容

**w**：保存

**进入编辑模式：i**

**alt+u**：Undo

**alt+p**：黏贴内容

**退出编辑模式：Esc**

命令输入模式：**shift+:** （下方出现冒号输入如下命令）

退出: **q**

保存并退出: **wq**

强制退出: **q!**

**显示行号: set number**

**:**7 快速跳转到某行

**解决Git 中文显示乱码问题**

在控制台窗口标题栏右键点击，选择Options

Text选项，设置zh\_CN和UTF-8

然后在git 终端执行命令：

|  |
| --- |
| git config --global core.quotepath false  git config --global gui.encoding utf-8  git config --global i18n.commitencoding utf-8  git config --global i18n.logoutputencoding utf-8  export LESSCHARSET=utf-8 |