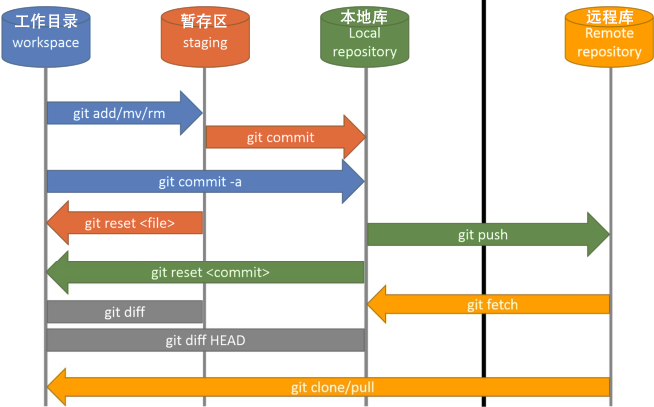
Git 词典

**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

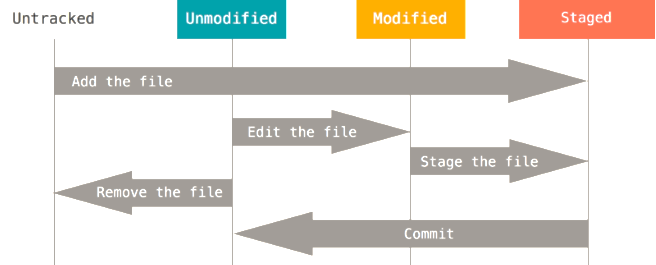
●还原本地版本为远程最新

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

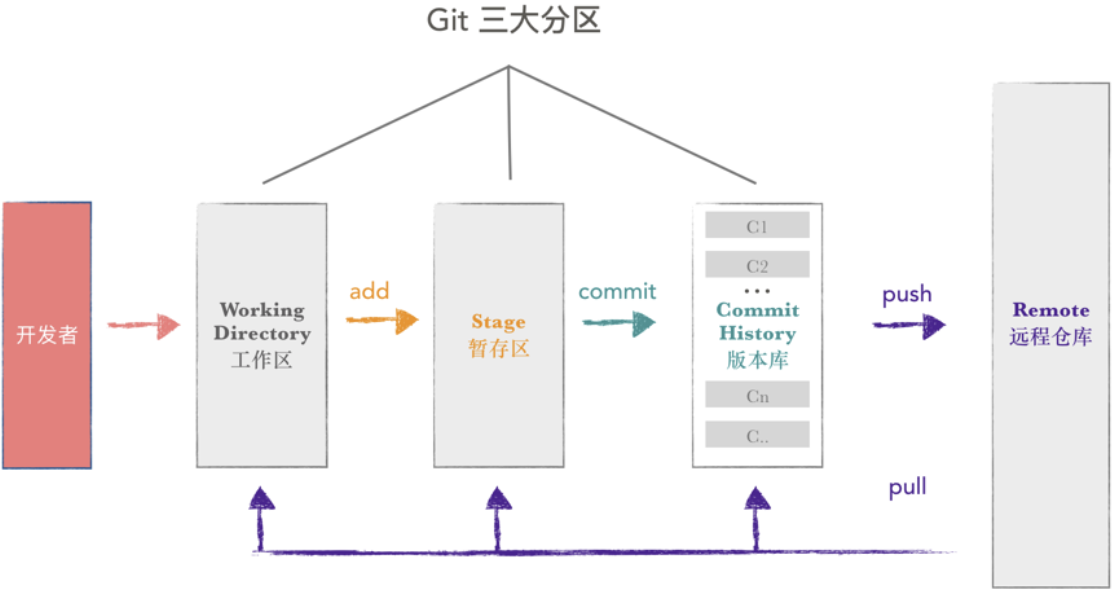
◆1下图展示了**文件**在本地管理中各操作的去向和状态：

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

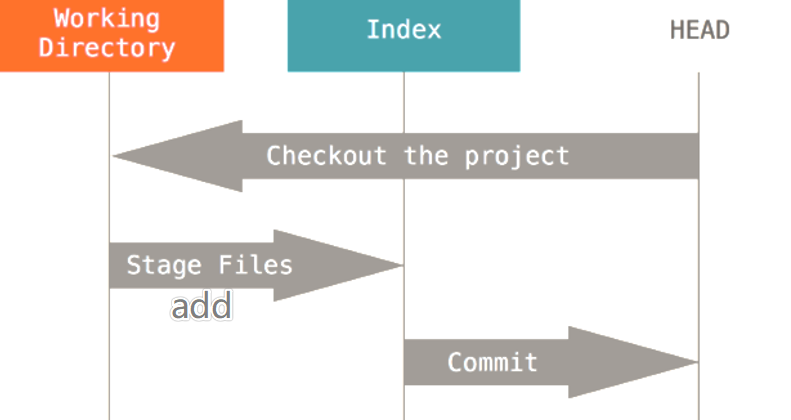
通过操作来管理三棵树，这里“树”代表文件的集合。

|  |  |
| --- | --- |
| **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）

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：

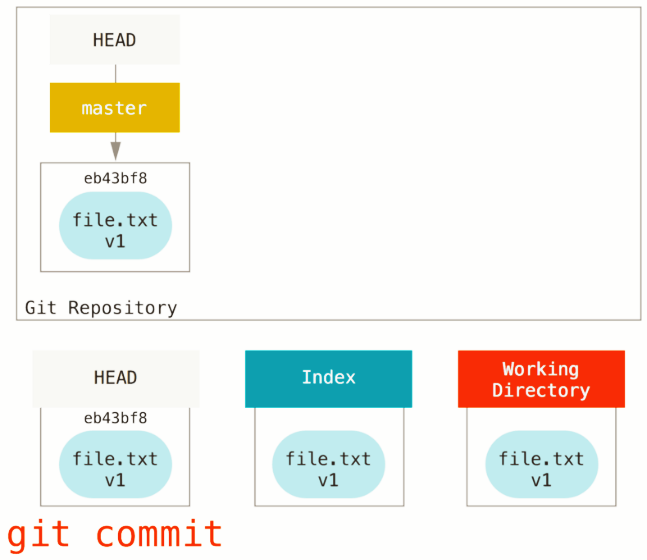




此时只有工作目录有内容，我们要提交这个文件，用git add来获取工作目录中的内容复制到索引中。



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。

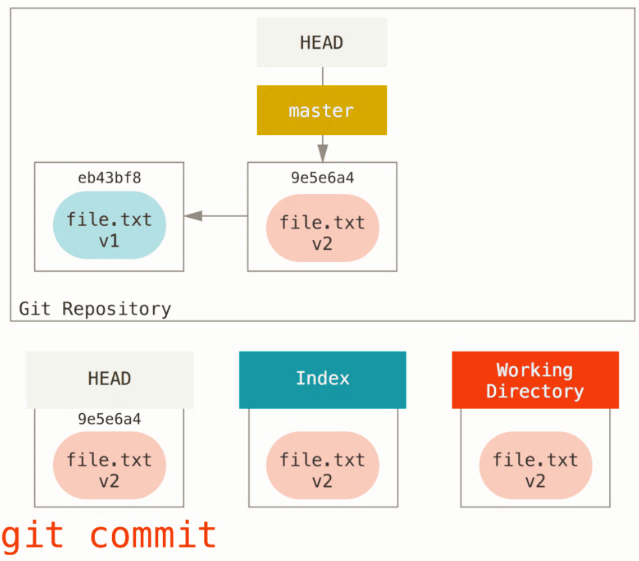
现在对文件进行修改然后提交它，修改后的文件为v2版本，并标记为红色，



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，

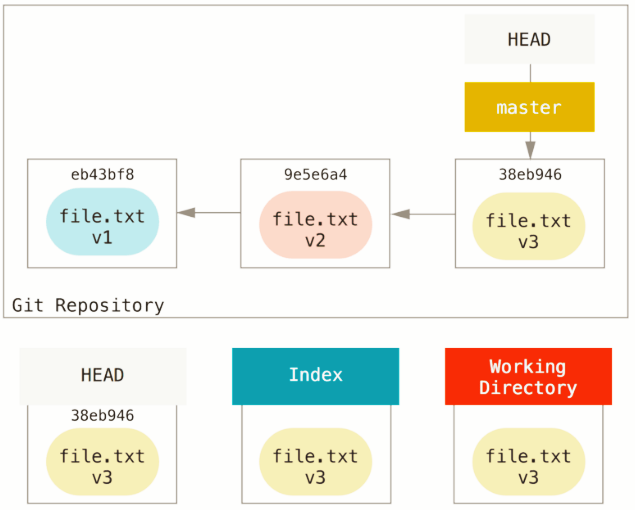


切换分支或克隆的过程也类似，当检出一个分支时会修改**HEAD**指向新的分支引用，将**索引**填充为该次提交的快照，然后将**索引**的内容复制到**工作目录**。

◆3 END

# ◇4 重置的作用 reset

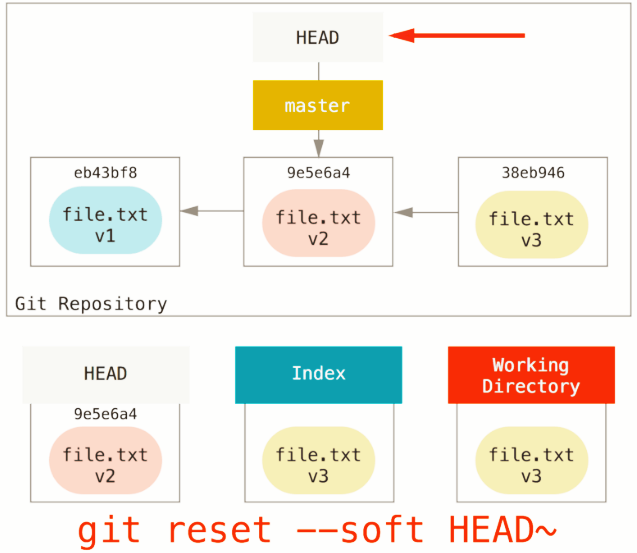
假设我们再次修改file.txt文件并第三次提交它，现在历史看起来是这样的：



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）

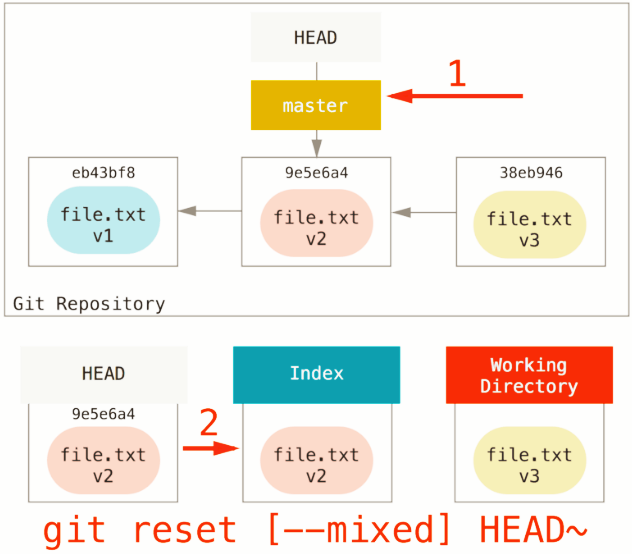
第一步：移动HEAD

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）。

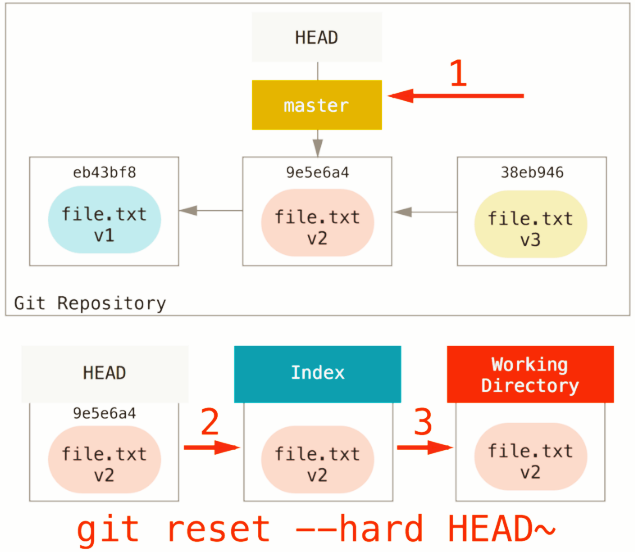


第二步：更新索引（--mixed）

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：

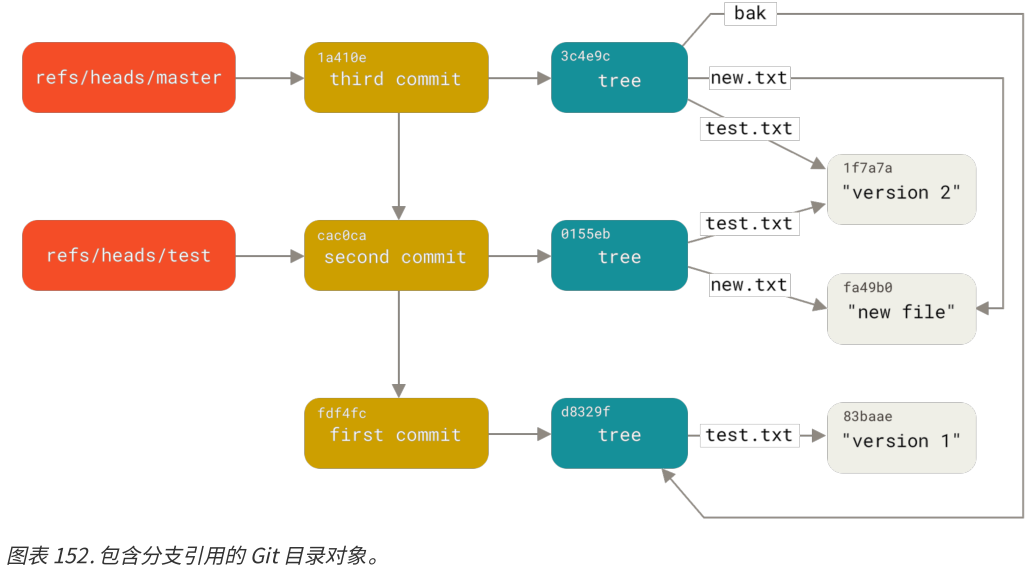
git update-ref refs/heads/test cac0ca

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

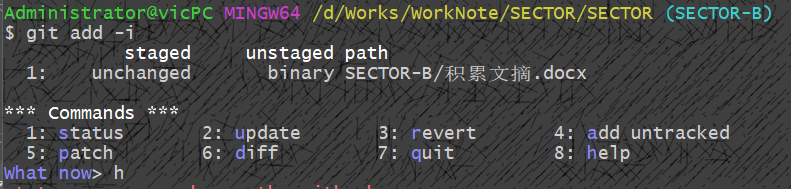
git add file1 file2 #跟踪多个文件

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 <url> <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

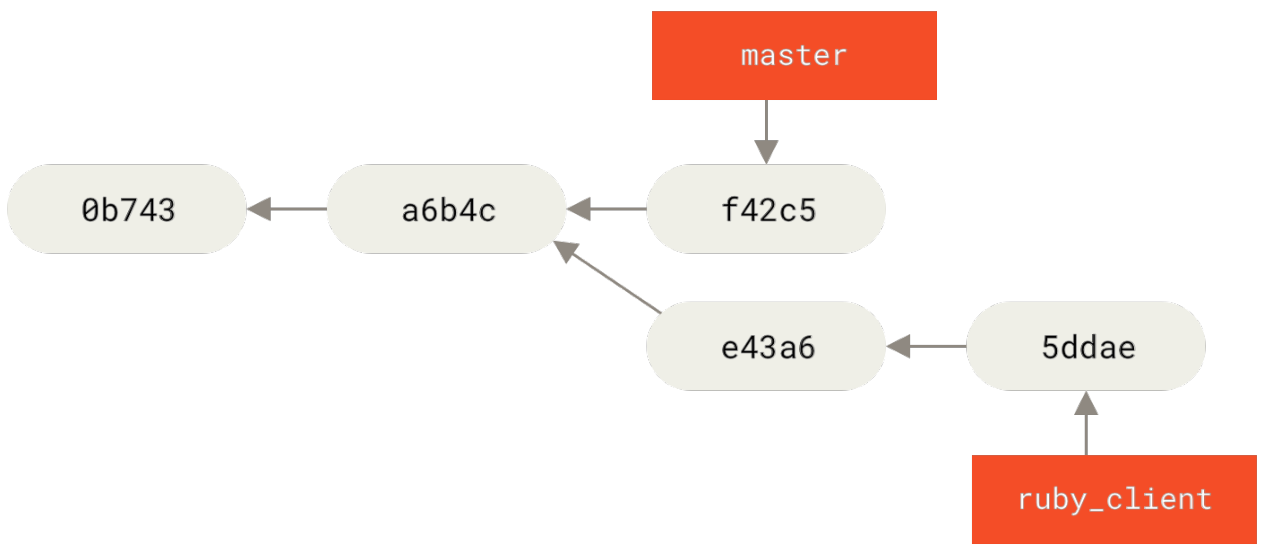
-d 含整个文件夹

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

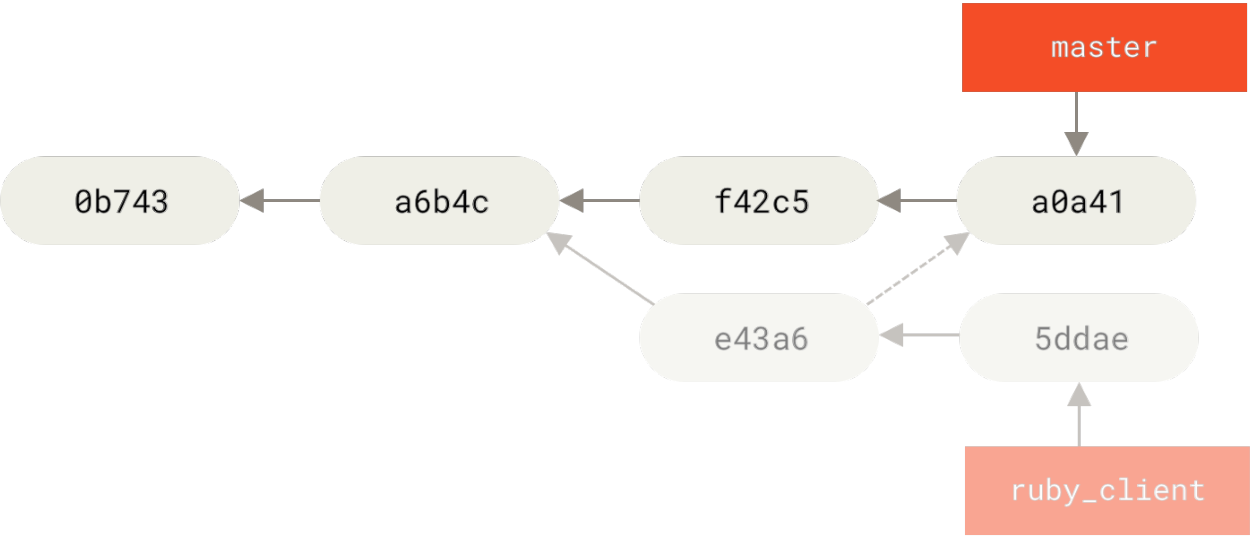
## cherry-pick

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

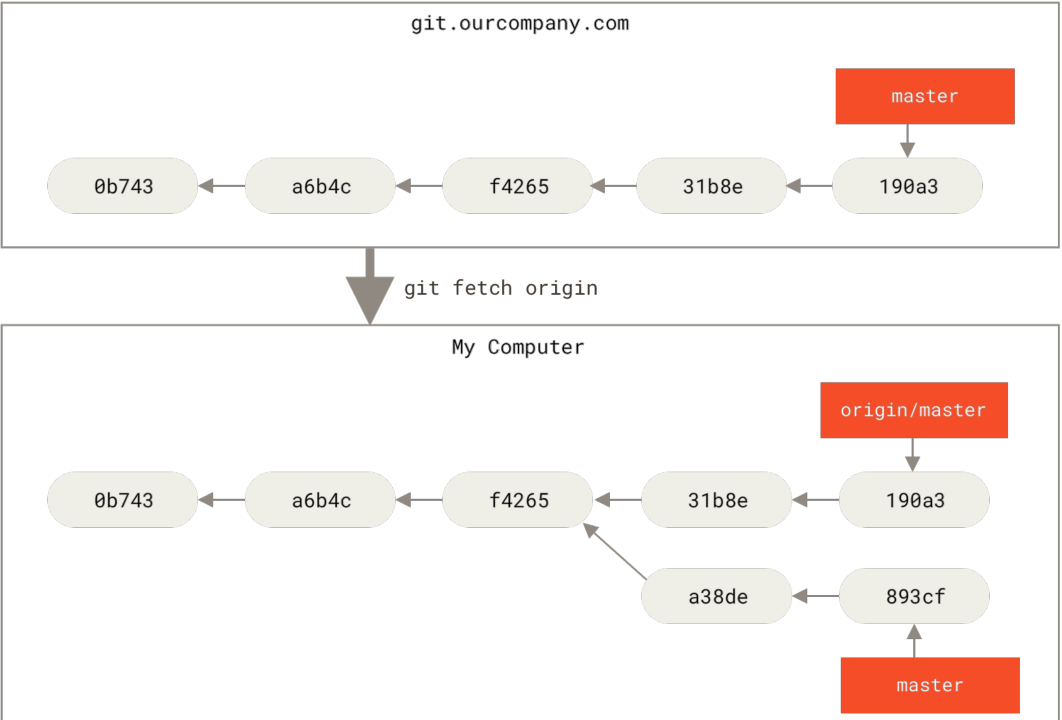
git diff head <file> #工作区vs版本库

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

## fast-forwards

# I

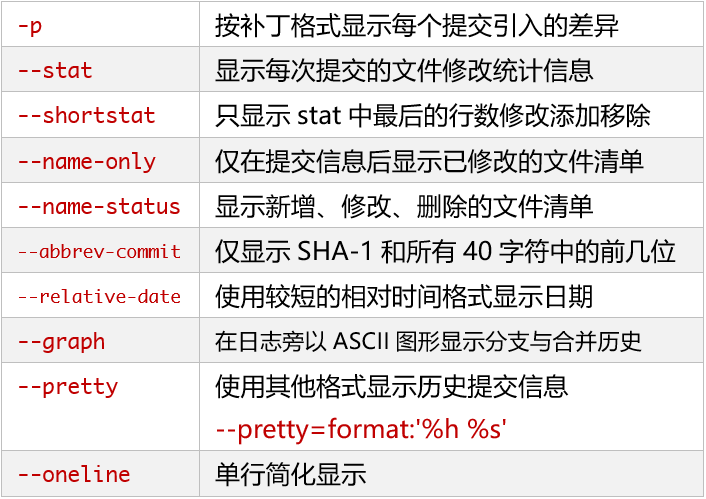
## init

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>

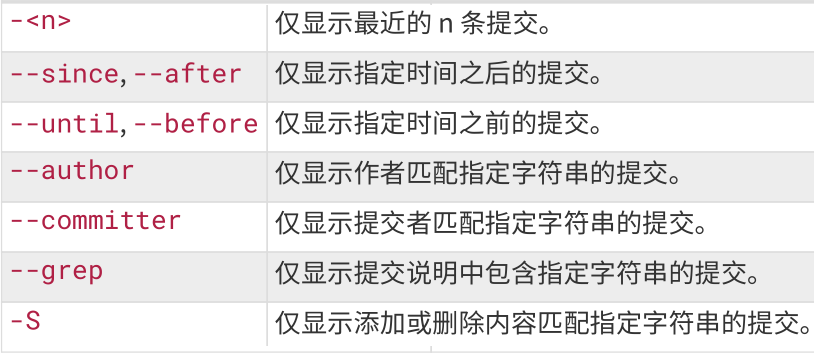
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit

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）

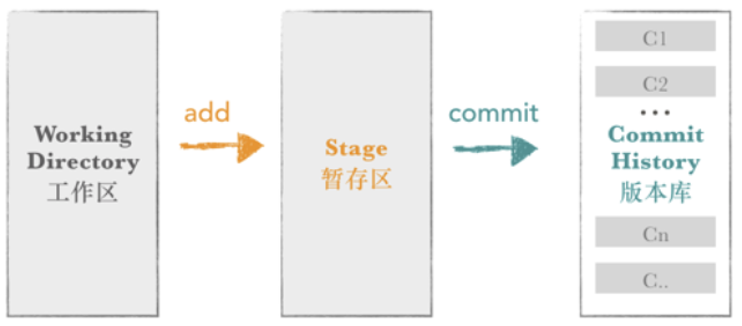
回滚文件，移动HEAD指向索引的位置，不改变索引和工作目录（◆2、4重置的作用）

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）

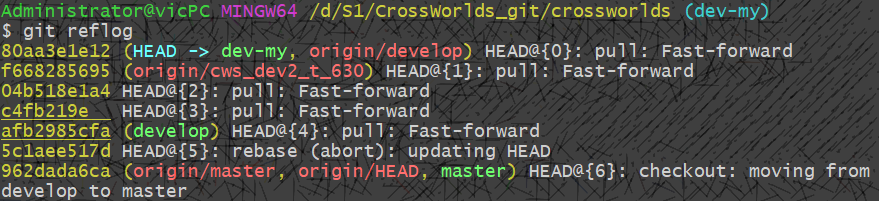
git reset HEAD <files> #可以使用通配符“\*”

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

还原撤销指定的提交。

git revert <commitSHA-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

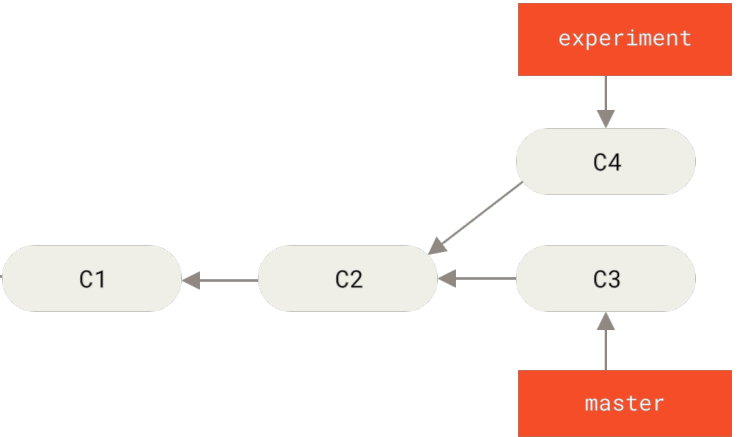
## rm

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：



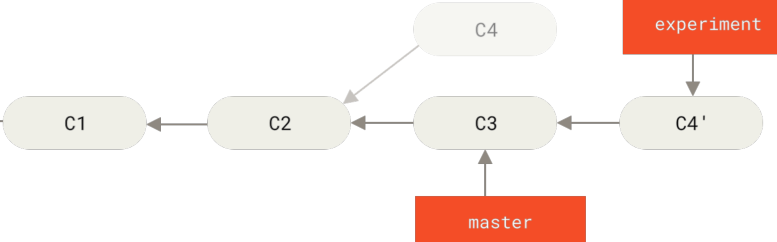
(图表 36. 通过合并操作来整合分叉的历史)

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。

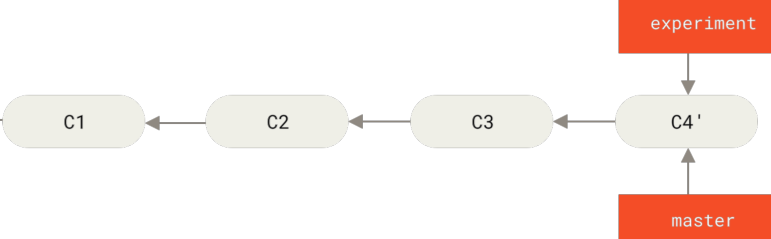


(图表37.将C4中的修改变基到C3上)

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。

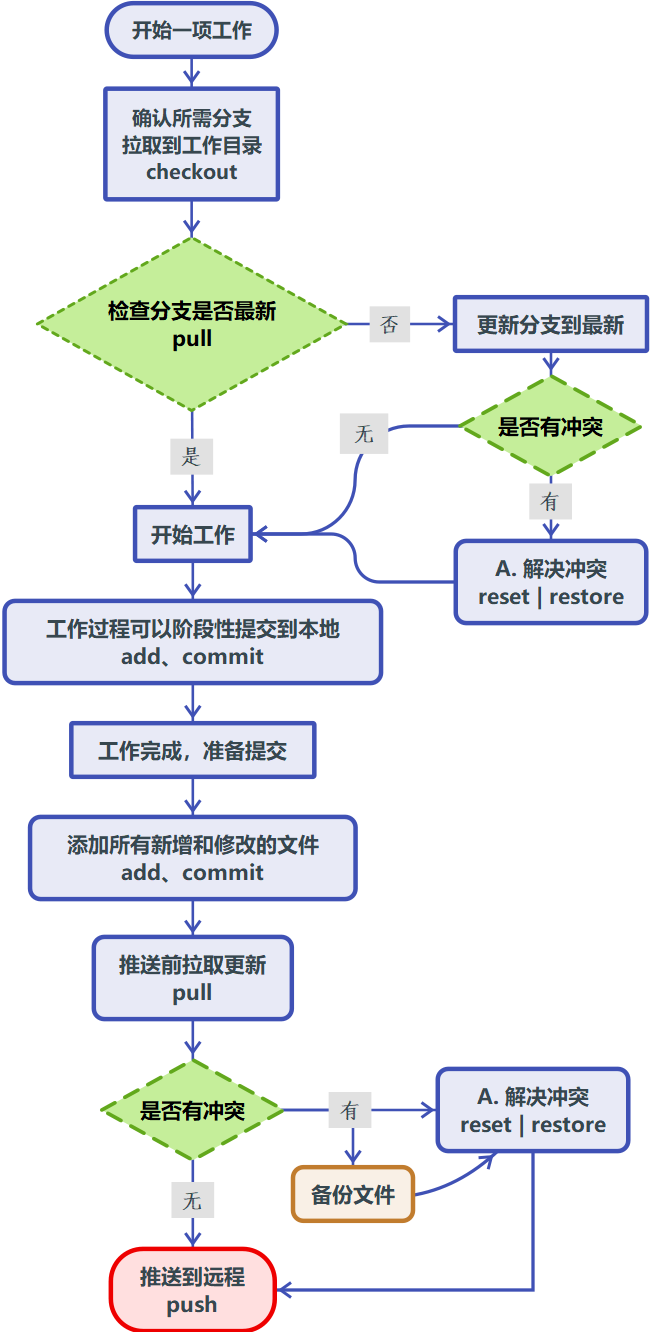
git switch <branch> //切换分支

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

# U

# Git之美术操作简书

**工作流程图：**



## 解决冲突

pull拉取更新不成功的原因无非就是跟踪的文件出现冲突，可以通过一些方法让工作目录处于同步状态，可以用git add相关文件来告诉Git文件冲突已经解决。

一、重置reset

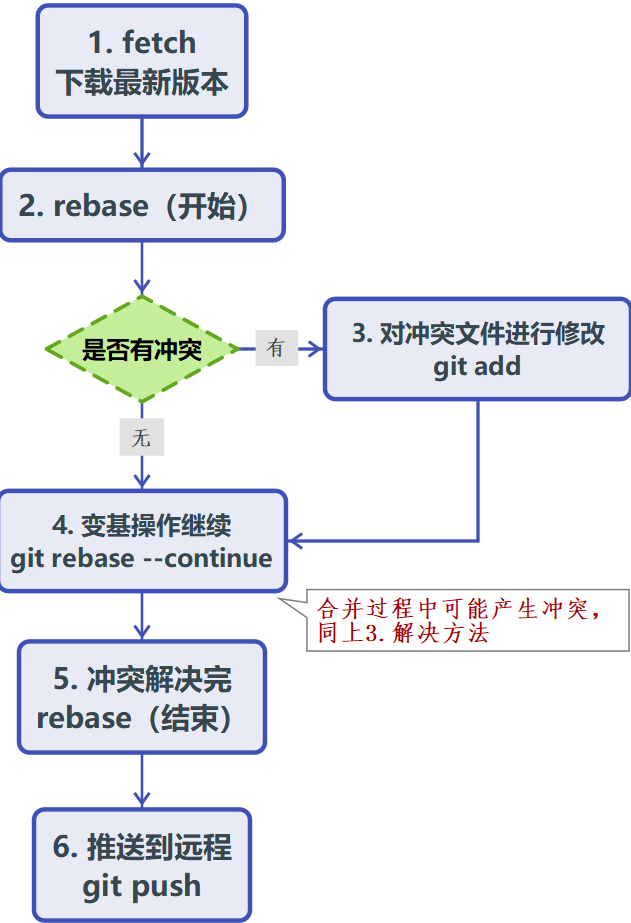
在修改的文件量可控的情况下，可以手动将本地修改备份到外部目录中，然后利用reset重置工作目录。

二、cherry-pick与检出checkout

三、变基rebase

利用变基操作将远程分支作为本地工作分支的基础，实现拉取最新远程的文件的同步，完成变基后就可以直接push推送了。

**rebase流程图：**



git rebase origin/<远程分支名> //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 add <冲突的文件> //第3步操作

git rebase --continue //第4步继续变基操作

## 重命名本地分支(branch -m)

git branch -m <old-name> <new-name>

## ▲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的版本#

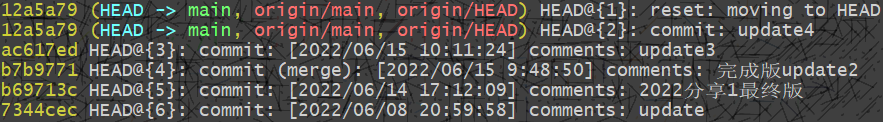
## 停止跟踪文件(rm --cached)

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

## ▲查看操作历史(reflog)

git reflog



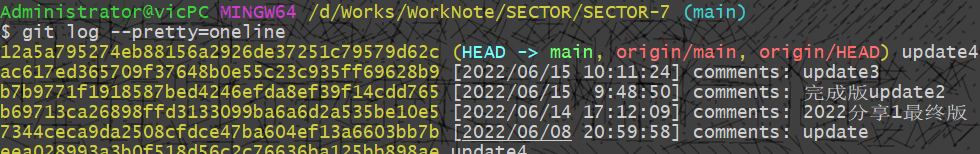
可以用git show HEAD@{5}通过指针号查看提交记录

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）

## ▲查看commit历史(log --pretty)

git log --pretty=oneline [<branch>|留空-当前分支]

返回：SHA-1 commits



## ▲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> #也可指定显示条数-2

git show --raw #省略后面id查看最后一次提交

查看最近3条更新日志，简单显示涉及文件：

git log -3 –stat#列出文件，显示修改添加删除文件数

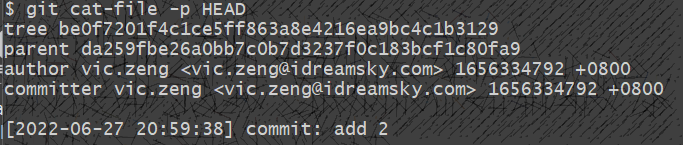
git log --name-only -1 #查看最近1条提交的文件

## 查看HEAD等快照目录列表(cat-file)

git cat-file（底层命令）

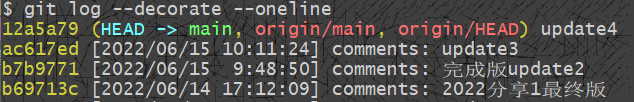
|  |  |
| --- | --- |
| -t | 显示对象类型 |
| -s | 显示对象大小 |
| -e | 如对象存在且有效则不输出，否则返回0 |
| -p | 根据对象类型，显示对象内容，包括文件内容 |

git cat-file -p HEAD



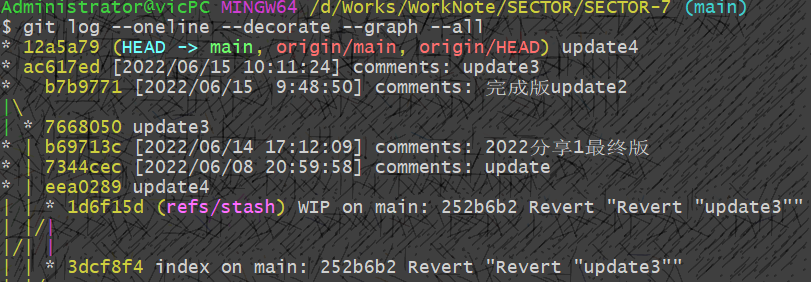
## ▲查看各分支指向的对象(--decorate)

git log --oneline --decorate



输出提交历史、各分支指向及项目的分支分叉情况：

git log --oneline --decorate --graph --all



## 查看单词层面的对比(log --word-diff)

git log -U<num> --word-diff #-U后设置显示行数，默认3行

我们常用 ​​-p​​​ 选项展开显示每次提交的内容差异，用 ​​-2​​ 则仅显示最近的两次更新

git log -p --word-diff -2

## ▲查看远程仓库的名称与连接地址(remote)

可以添加多个远程连接地址，指定分支与远端的跟踪。

git remote -v

可以添加多个仓库连接，





添加完仓库连接后，可以将本地分支与远程仓库建立链接，之后就可以拉取推送

git branch --set-upstream-to=origin/master master

## 更新远程仓库分支列表(remote)

查看远程分支列表发现远端不符时，可以执行如下命令：

git remote update <origin> --prune

## ▲修改最后一次提交--amend

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

有两种情况处理方式如下，

**一、只修改提交信息：**

直接执行上面的命令，按i进入修改状态进行提交信息修改，按Esc退出修改状态，按shift+:输入命令wq（保存并退出）。

**二、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。

## ▲重置指定commit历史到工作目录(reset)

git reset --hard [<SHA-1>|留空提取最新本地提交]

## ▲恢复本地库记录到缓存区(reset)

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

## ▲干净还原远程版本分支3种方法

1：

git fetch –all #拉取远程到本地库

git reset --hard origin/<branch>

2：

git reset --hard #重置到最顶历史

git pull origin <branch> #拉取远程分支

3：或只需执行一条，直接重置到远程分支：

git reset --hard origin/<branch>

## ▲还原文件为本地或远程(checkout)

在使用cherry-pick解决冲突时会用到

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 <SHA-1> <files> #还原某提交的文件

可以先查看提中包含的文件，

git show --raw <SHA-1> #查看指定提交中的文件

git log --name-only #列出所有提交中的文件

再提取想要的文件：

git checkout -- <file> #还原到缓存区的版本

git checkout <SHA-1> <files> #还原某提交的文件

## ▲查看缓存区中的文件内容(checkout --)

git checkout -- #列出缓存区中的文件

## ▲拉取某分支的某文件(checkout)

可以是本地分支，也可以是远程分支origin/<branch>

git checkout <branch> <file> <…>

## ▲拉取远程分支在本地创建新跟踪分支(ck)

git checkout -b <new-branch> origin/<branch>

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

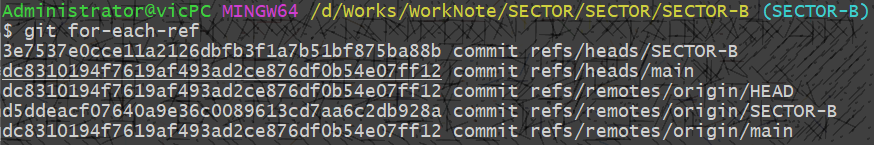
## 查看两分支间未合并的提交(log)

查看<分支B>中有哪些提交未合并到<分支A>

git log <分支A>..<分支B> --oneline

## 列出所有分支引用的SHA1(for-each-ref)

git for-each-ref



## ▲清理工作目录(clean)

清除未跟踪(Untracked)的文件，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

储藏到缓存区stash再清理：

git stash -u #暂存包含未跟踪的文件

## 在某个提交基础上创建新分支<branch>

git update-ref refs/heads/<branch> <SHA-1>

git log –pretty=oneline <branch> #查看新分支

## 删除本地、远程分支

git branch -d <branch> #删除本地分支,强制用-D

git push origin –delete <branch> #删除远程

## 退出变基过程操作 (master|REBASE 1/9)

git rebase --abort

## Git用户名和邮箱(config)

查看

git config user.name

git config user.email

设置

git config --global user.name 'username'

## ！出现Unlink of file ’..’

文件被其他程序暂用Git无法修改，可以关掉暂用文件的相关程序，或重启电脑。

## ！有时查看不到远程部分分支

解决办法：

git fetch

git branch -a #这时应该就能看到所有分支

## 查看数据占用多少空间(gc)

查看对象数量及占用空间

git gc

快速查看占用空间大小

git count-objects -v

## 设置快捷命令

设置常用命令的简化自定义替代字符

git config --global alias.<custom> '<status>'

下面列出最常用的快捷设置(注意英文单引号别打错)

git **his** #查看commit历史

git config --global alias.his 'log --pretty=oneline'

git **dec** #查看各分支指向的对象--decorate

git config --global alias.dec 'log --oneline --decorate'

git **st** #查看当前状态status

git config --global alias.st 'status'

git **cus** #自定义格式查看历史（）

git config --global alias.cus 'log --pretty=format:"%h %s | %ar | %cn"'

git **lsf** #列出各提交的文件内容，单行显示

git config --global alias.lsf 'log --name-only --oneline'

## 查看所有现有配置设置：

git config –list

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

# 附录

## .gitignore排除跟踪语法

**排除整个文件夹**

/folder

**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

## vim文本编辑操作指南

有时在写合并信息时会用到。

进入编辑模式：i

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!

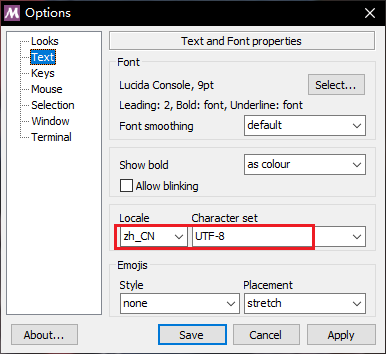
显示行号: set number

## 解决Git Bash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 bash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