**git 常用命令**

**添加文件**

● git add .

功能：添加当前目录以及子目录下被修改、新建的文件

● git add -u

功能：-u=--update，update tracked files，添加当前目录以及子目录下被修改的文件

● git add -A

功能：-A=--all，添加所有目录下已被修改和已被删除文件

**删除文件**

**删除仓库文件**

1、先查看哪些文件可以删除

|  |
| --- |
| git rm -n <file>        # 删除一个文件  git rm -r -n <dir>/\*    # 删除 dir 下的所有文件 |

参数含义：

* -r    递归移除目录
* -n    加上这个参数，执行命令时，是不会删除任何文件，而是展示此命令要删除的文件列表预览，所以一般用这个参数先看看要删除哪些文件，防止误删，确认之后，就去掉此参数，真正的删除文件。

2、执行删除

|  |
| --- |
| git rm <file>        # 删除一个文件  git rm -r <dir>/\*    # 删除 dir 下的所有文件 |

3、提交

|  |
| --- |
| git commit -m "删除文件" |

**删除已添加到索引的文件**

|  |
| --- |
| # 查看  git rm -n --cached <file>        # 删除一个文件  git rm -r -n --cached <dir>/\*    # 删除 dir 下的所有文件    # 执行删除  git rm --cached <file>        # 删除一个文件  git rm -r --cached <dir>/\*    # 删除 dir 下的所有文件 |

或

|  |
| --- |
| git reset HEAD              # 删除已添加到索引的全部文件  git reset .                 # 删除已添加到索引的全部文件（如果从来没有 commit 可以用这个）  git reset HEAD -- <file>    # 删除已添加到索引的某个文件 |

**清空工作区的内容**

|  |
| --- |
| git checkout .    # 本地所有修改的，没有的提交的文件，都返回到原来的状态，但是对于新建的文件和文件夹无法清空，必须组合下面命令  git clean -df     # 删除所有未被追踪（untracked）的文件和文件夹  git clean 参数     -d    删除所有未被追踪（untracked）的文件和文件夹     -f    强制删除     -n    显示将要删除的文件和目录 |

**git diff**

● git diff [branch\_name] [– <file\_path>]

功能：查看工作树与索引区的差异；指定 branch\_name 指定特定分支来进行对比，可以是本地分支也可以是远程分支；指定 -- <file\_path> 可以仅查看某个文件的差异

● git diff –cached [– <file\_path>]

功能：查看索引区与 HEAD 的差异；指定 -- <file\_path> 可以仅查看某个文件的差异

● git diff <commit> [– <file\_path>]

功能：查看工作树与 commit 的差异，commit 可以是 HEAD；指定 -- <file\_path> 可以仅查看某个文件的差异

**git stash**

● git stash save <save message>

功能： 执行存储时，添加备注，方便查找，只有 git stash 也是可以的，但查找时不方便识别

● git stash list

功能：查看 stash 了哪些存储

● git stash show

功能：显示做了哪些改动（简洁模式，只有文件名，后边跟上加号和减号），默认显示第一个存储；如果要显示其它存贮，后面加 stash@{$num}，比如第二个 git stash show stash@{1}

● git stash show -p

功能：显示做了哪些改动（详细模式，文件更改的具体内容），默认显示第一个存储；如果想显示其它存存储，命令：git stash show stash@{$num} -p，比如第二个：git stash show stash@{1} -p

● git stash apply

功能：应用某个存储，但不会把存储从存储列表中删除，默认使用第一个存储，即 stash@{0}；如果要使用其它的，命令：git stash apply stash@{$num}， 比如第二个：git stash apply stash@{1}

● git stash pop

功能：应用某个存储，并把存储从存储列表中删除，默认为第一个 stash，即 stash@{0}；如果要应用并删除其它 stash，命令：git stash pop stash@{$num} ，比如应用并删除第二个：git stash pop stash@{1}

● git stash drop stash@{$num}

功能：丢弃 stash@{$num} 存储，从存储列表中删除这个存储

● git stash clear

功能：删除所有缓存的 stash

**修改提交**

**修改最近的一次提交**

|  |
| --- |
| #-> 编辑某些漏掉的文件，并 git add <file>，此步骤可有可无  git commit –-amend [--no-edit] [--author="author\_name <author\_email>" | --reset-author]    #注意，email 前后加尖括号 |

功能：指定 amend 选项执行提交的话，可以修改同一个分支最近的提交内容和注解。

* --no-edit    不需要修改注解，不弹注解编辑框
* --author="authro\_name <author\_email>" | --reset-author    修改提交作者

主要使用的场合：

* 添加最近提交时漏掉的档案
* 修改最近提交的注解

**修改过去多次的提交**

|  |
| --- |
| git rebase -i <commit> |

详细操作步骤参看：《ProGit\_v2.1.55(Git官方书籍) > P242(重写历史)》或 [用 rebase -i 修改提交](https://backlog.com/git-tutorial/cn/stepup/stepup7_6.html)

**重置到过去某次提交**

● git reset –soft <commit>

功能：重置 HEAD 和 branch 的同时，保留索引区和工作树里比 commit 更加新的内容，相当于回到 git add 执行后的状态。

● git reset [–mixed] <commit>

功能：重置 HEAD 和 branch 的同时，清空索引区里比 commit 更加新的内容，保留工作树里比 commit 更加新的内容。

● git reset –hard <commit>

功能：重置 HEAD 和 branch 的同时，清空索引区和工作树里比 commit 更加新的内容。

注意：

本地回退到以前的版本后，然后 git push 会报错，因为本地版本比远程版本旧，此时可用 git push -f 强制推送到远程仓库

**回滚过去的提交**

|  |
| --- |
| git revert <commit>    # 回滚某次提交  git revert <commit1>..<commit2>    # 回滚区间 (commit1, commit2] 的提交，左开右闭 |

注意：

* 注意 revert 命令会对每个撤销的 commit 进行一次提交，指定 --no-commit 后可以最后一起手动提交
* 回滚之后用 git push 会报错，此时可用 git push -f 强制推送到远程仓库

详细操作步骤参看：[git revert 的操作 -> 简书](https://www.jianshu.com/p/5e7ee87241e4)

**git log**

● git log [–graph] [–oneline] [--author="<author\_name>"]

功能：查看提交记录。指定 --graph 选项能以文本形式显示更新记录的流程图；指定 --oneline 选项，能在一行中显示提交的信息；指定 --author=“<author\_name>” 可以只看某个作者的提交，作者名不一定是全匹配，只要包含那个子串就会匹配，使用 git log --author="<author\_name1>|<author\_name2>" 可以搜索包含作者 1 或作者 2 的提交信息。

● git log <file\_name>

功能：只看 file\_name 的历史提交记录

● git log -p <file\_name>

功能：只看 file\_name 的历史提交记录，并且显示每次文件变化的地方

● git log --grep <key\_words>

功能：搜索提交历史记录，不支持多个关键词的与或非语法

● git log –oneline | grep <Linux grep 命令语法>

功能：通过管道模式，使用 Linux grep 命令搜索提交历史记录，支持的语法更多

● git reflog show --date=iso [<branch name>]

功能：依托本地记录，查看当前用户在所有分支或指定某个分支的所有操作记录（包括何时创建分支、commit、reset、已删除的 commit 等操作），git log 不能查看已删除的 commit 记录

**git show**

● git show <commit> <file>

功能：查看某个版本的某个文件变化的内容

● git show <commit>:<file>

功能：查看某个版本的某个文件的内容

**git checkout**

● git checkout <branch>

功能：切换分支

● git checkout – <files>

功能：还原正在手头上修改，还没被添加到索引里的文件

**分支管理**

**查看分支**

● git branch

功能：查看本地分支

● git branch -r

功能：查看远程分支

● git branch -a

功能：查看远程和本地分支

**创建分支**

● git branch <new\_branch\_name>

功能：创建分支（没有任何提示）

● git checkout -b <new\_branch\_name> <commit>

功能：根据指定版本号创建分支

● git checkout -b <new\_branch\_name> remotes/<branch\_name>

功能：新建并切换到 new\_branch\_name 分支，从远程分支下载内容到此分支

● git push --set-upstream origin <branch\_name>

功能：在本地创建新分支后用此命令推送到远程

**删除分支**

● git branch -d <branch\_name>

功能：删除指定的本地分支

● git push --delete origin <branch\_name>

功能：删除指定的远程分支

● git fetch -p/--prune

功能：清理本地无效分支（远程已删除本地没删除的分支）

**重命名分支**

|  |
| --- |
| # 1. 重命名本地分支  git branch -m <old\_branch\_name> <new\_branch\_name>    # 2. 删除远程分支  git push --delete origin <old\_branch\_name>    # 3. 上传新命名的本地分支  git push origin <new\_branch\_name>    # 4. 把修改后的本地分支与远程分支关联  git branch --set-upstream-to origin/<new\_branch\_name> |

**切换分支**

● git checkout <branch>

功能：切换分支

**合并分支**

● git merge [-Xignore-all-space | -Xignore-space-change] <branch\_name>

参数说明：

* -Xignore-all-space       在比较行时完全忽略空白修改
* -Xignore-space-change    在比较时将一个空白符与多个连续的空白字符视作等价的

如合并的时候因为两个分支文件的换行符引起冲突，可以加以上参数，详细教程参看：《ProGit\_v2.1.55(Git官方书籍) > P269(高级合并)》或 [7.8 Git 工具 - 高级合并(网页版)](https://git-scm.com/book/zh/v2/Git-%E5%B7%A5%E5%85%B7-%E9%AB%98%E7%BA%A7%E5%90%88%E5%B9%B6)

若合并出现冲突参看：[用 merge 合并 并解决合并的冲突 -> 猴子都能懂的GIT入门](https://backlog.com/git-tutorial/cn/stepup/stepup2_7.html)

● git rebase <branch\_name>

详细操作步骤参看：[用 rebase 合并 -> 猴子都能懂的GIT入门](https://backlog.com/git-tutorial/cn/stepup/stepup2_8.html)

● git cherry-pick  [options] ( <commit> | <commit> ... | <commit1>..<commit2> | ..<branch> )

功能：将某一次/多次/(commit1, commit2]/属于 branch 的祖先但不属于当前 branch 的祖先的所有提交合入到本地当前分支上

常用 options：

* --quit             退出当前的 chery-pick 序列
* --continue          继续当前的 chery-pick 序列
* --abort            取消当前的 chery-pick 序列，恢复当前分支
* -n, --no-commit    不自动提交
* -e, --edit          编辑提交信息

注意：如果一次 cherry-pick 了多次提交，则多次提交会合并成一个。因此，为了不丢失每次的提交信息，最好每次 cherry-pick 一个。

步骤：

|  |
| --- |
| # 1. 选择需要合并的提交  git cherry-pick <commit>  # 2. 如果有冲突则  git add <conflict file>  git cherry-pick --continue |

常见问题：

1) The previous cherry-pick is now empty, possibly due to conflict resolution

原因：在 cherry-pick 时出现冲突，解决冲突后本地分支中内容和 cherry-pick 之前相比没有改变，因此当在以后的步骤中继续 git cherry-pick 或执行其他命令时，由于此时还处于上次 cherry-pick，都会提示该信息，表示可能是由于解决冲突造成上一次 cherry-pick 内容是空的。

解决方案：1)执行 git cherry-pick --abort 取消上次操作；2)执行 git commit --allow-empty，表示允许空提交。

2) fatal: You are in the middle of a cherry-pick – cannot amend

原因：在 cherry-pick 时出现冲突，没有解决冲突就执行 git commit --amend 命令，从而会提示该信息。

解决方案：首先在 git commit --amend 之前解决冲突，并完成这次 cherry-pick：git add .; git cherry-pick --continue

更多详细教程，参看：[Git cherry-pick 这个命令你会经常用到(知乎)](https://zhuanlan.zhihu.com/p/90816644)

**查看分支图**

|  |
| --- |
| git log --graph --decorate --oneline --simplify-by-decoration --all |

参数说明：

* --graph                     以图形显示，要不然没有最左边的分支线
* --decorate                  装饰，分支名字高亮等
* --oneline                   每次的提交内容在一行显示，比较紧凑
* --simplify-by-decoration    只显示分支合并，拉出等重大操作的大致前后关系图，否则显示每次提交
* --all                       让 --simplify-by-decoration 再稍微详细一点

**git config**

**设置提交的用户名和邮箱**

|  |
| --- |
| git config [--global] user.name "<user\_name>"  git config [--global] user.email "<email>" |

参数说明：

* --global    加上为全局设置，不加只对当前工程有效（适用于多人在同一台服务器上使用 git）

**设置命令别名**

|  |
| --- |
| git config --global alias.<alias\_name> <command\_name> |

**设置换行符**

Win 和 Linux 换行符

* Linux / Mac OS 的换行符为：0x0A（LF；\n）
* DOS / Windows 的换行符为：0x0D0A（CRLF；\r\n）

autocrlf 的配置项

* true     提交时转换为 LF，拉取时转换为 CRLF
* input    提交时转换为 LF，拉取时不转换
* false    提交拉取均不转换

e.g.

|  |
| --- |
| # 提交时转换为 LF，拉取时转换为 CRLF  git config --global core.autocrlf true  # 提交时转换为 LF，拉取时不转换  git config --global core.autocrlf input  # 提交拉取均不转换  git config --global core.autocrlf false |

safecrlf 的配置项

* true    拒绝提交包含混合换行符的文件
* false   允许提交包含混合换行符的文件
* warn    提交包含混合换行符的文件时给出警告

e.g.

|  |
| --- |
| # 拒绝提交包含混合换行符的文件  git config --global core.safecrlf true  # 允许提交包含混合换行符的文件  git config --global core.safecrlf false  # 提交包含混合换行符的文件时给出警告  git config --global core.safecrlf warn |

**更改编辑器为 vim**

|  |
| --- |
| git config --global core.editor vim |

**不忽略/忽略文件权限变化**

|  |
| --- |
| git config --global core.filemode true    #不忽略文件权限变化  git config --global core.filemode false   #忽略文件权限变化 |

使用场景：文件并没有修改，git status 却显示修改，可能是文件权限不一样导致的，配置为 false 即可

**git remote**

● git remote set-url origin <new\_url>

功能：重定向远程仓库地址

**重要概念**

**HEAD**

HEAD 指向的是现在使用中的分支的最后一次更新。通常默认指向 master 分支的最后一次更新。通过移动 HEAD，就可以变更使用的分支。

**stash**

还未提交的修改内容以及新添加的文件，留在索引区域或工作树的情况下切换到其他的分支时，修改内容会从原来的分支移动到目标分支。但是如果在 checkout 的目标分支中相同的文件也有修改，checkout 会失败的。这时要么先提交修改内容，要么用 stash 暂时保存修改内容后再 checkout。stash 是临时保存文件修改内容的区域。stash 可以暂时保存工作树和索引里还没提交的修改内容，您可以事后再取出暂存的修改，应用到原先的分支或其他的分支上。

**配置项**

**忽略文件**

在 .git 文件夹同层建立一个 .gitignore 文件，添加需要忽略的文件

**设置代理**

设置 http 代理：

|  |
| --- |
| git config --global http.proxy http://<user>:<password>@<proxy\_server>:<port>  git config --global https.proxy http://<user>:<password>@<proxy\_server>:<port>    #如果没有用户名和密码则：  git config --global http.proxy http://<proxy\_server>:<port>  git config --global https.proxy http://<proxy\_server>:<port> |

取消 http 代理：

|  |
| --- |
| git config --global --unset http.proxy  git config --global --unset https.proxy |

设置 sock5 代理：

|  |
| --- |
| git config --global http.proxy socks5://<user>:<password>@<proxy\_server>:<port>  git config --global https.proxy socks5://<user>:<password>@<proxy\_server>:<port>    #如果没有用户名和密码则：  git config --global https.proxy socks5://<proxy\_server>:<port>  git config --global https.proxy socks5://<proxy\_server>:<port> |

一般情况下都是这样的：

|  |
| --- |
| git config --global http.proxy socks5://127.0.0.1:1080  git config --global https.proxy socks5://127.0.0.1:1080 |

**git clone http/https 记住账号和密码**

|  |
| --- |
| vim .git/config  #然后在文件末尾添加以下两行内容：  [credential]  helper = store [--file <path>/.git-credentials]    #如果多人用一个 root 账号同时使用 git，各自需要配置不同的路径，否则会串 |

**常见错误**

**git 修改密码后，git push 报"fatal: Authentication failed for xxxx"错误**

先后输入如下两条命令，然后再次 git push 会提示重新输入用户名和密码

|  |
| --- |
| git config --system --unset credential.helper  git config --global credential.helper store |

**error: The following untracked working tree files would be overwritten by checkout**

|  |
| --- |
| git clean -d -fx |

**git clone 提示：fatal: unable to access xxxxx : Could not resolve host : gitlab.xxx.com**

原因：无法解析主机名字，在 /etc/host 中添加主机名和 ip 映射即可

步骤：

1、先 ping 一下 gitlab.xxx.com 看是否可以 ping 通

2、如果报错，则找一台可以 ping 通的机器，看一下 ping 通后的 ip 是多少

3、接下来以 CentOS 为例：

|  |
| --- |
| vim /etc/hosts  #TODO  按照格式，在文件末尾添加 ip 和主机名映射，e.g.「xx.xxx.xx.xx gitlab.xxx.com」  /etc/init.d/networking restart    #重启网络 |

**git error: unable to rewind rpc post data - try increasing http.postBuffer**

|  |
| --- |
| 当使用 git 一次提交的数据过大时，会出现以上错误，增大 postBuffer 即可  git config --global http.postBuffer 100000000（设置为 100M）  git config --list（查看一下） |

**error: RPC failed; result=18, HTTP code = 200**

|  |
| --- |
| # 解决办法1：  git repack -a -f -d --window=250 --depth=250    # 解决办法2：git gc --aggressive  git repack -a -f -d --window=250 --depth=250  # 解决办法3：  # TODO -> 进入 git 仓库目录  git config http.postBuffer 24288000 |

**error: server certificate verification failed. CAfile: /etc/ssl/certs/ca-certificates.crt CRLfile: none**

|  |
| --- |
| Linux下：  export GIT\_SSL\_NO\_VERIFY=1    Windows下：  set GIT\_SSL\_NO\_VERIFY 1    git：  git config --global http.sslVerify false |

**使用场景分析**

**Github 进行 fork 后如何与原仓库同步**

1、这里以 fork xiaolai 的仓库为例。下面所有操作，如无特别说明，都是在你的本地仓库的目录下操作。比如我的本地仓库为 /from-liujuanjuan-the-craft-of-selfteaching

2、进入到本地仓库的目录：

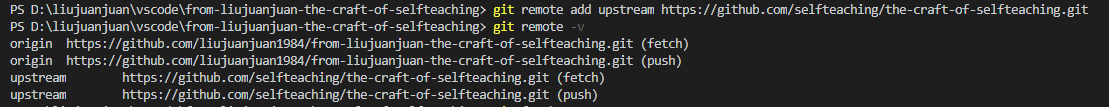
image

3、执行命令 git remote -v 查看你的远程仓库的路径：

image

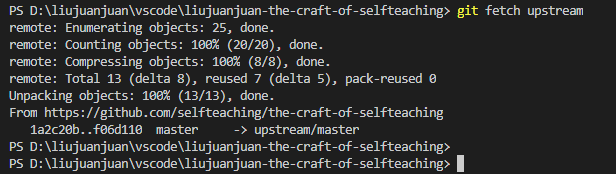
如果只有上面 2 行，说明你未设置 upstream(中文叫：上游代码库)。一般情况下，设置好一次 upstream 后就无需重复设置。

4、执行命令 git remote add upstream <https://github.com/selfteaching/the-craft-of-selfteaching.git> 把 xiaolai 的仓库设置为你的 upstream。这个命令执行后，没有任何返回信息，所以再次执行命令 git remote -v 检查是否成功。

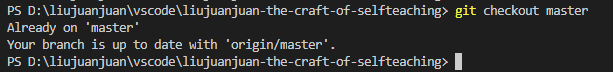


5、执行命令 git status 检查本地是否有未提交的修改。如果有，则把你本地的有效修改，先从本地仓库推送到你的 github 仓库。最后再执行一次 git status 检查本地已无未提交的修改。

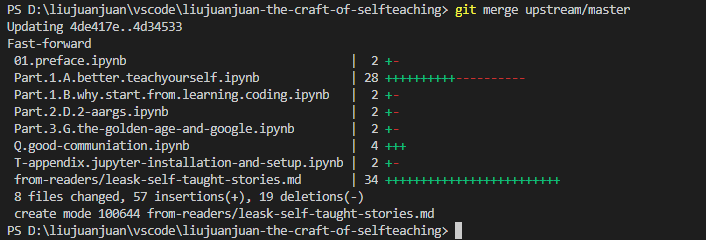
6、执行命令 git fetch upstream 抓取 xiaolai 原仓库的更新：



7、执行命令 git checkout master 切换到 master 分支：



8、执行命令 git merge upstream/master 合并远程的 master 分支：



9、执行命令 git push 把本地仓库向 github 仓库（你fork到自己名下的仓库）推送修改。

现在你已经解决了 fork 的仓库和原仓库版本不一致的问题。可以放心向 xiaolai 发起 merge request 了。如果以上操作你花了不少时间，而 xiaolai 的仓库又恰好更新了，那就重头再来一遍......

**git 对 Word 进行版本控制**

[点我查看教程网址](https://git-scm.com/book/zh/v2/%E8%87%AA%E5%AE%9A%E4%B9%89-Git-Git-%E5%B1%9E%E6%80%A7)

**在工程中引用其它子模块**

[点我查看教程网址](https://git-scm.com/book/zh/v2/Git-%E5%B7%A5%E5%85%B7-%E5%AD%90%E6%A8%A1%E5%9D%97)