1. 设置用户名 邮箱

git config --global user.name "jingleifan"

git config --global user.email "jingleifan@126.com"

1. 显示当前目录

Pwd

1. 将目录改为仓库目录

Git init

1. 显示隐藏目录并显示大小

Ls –ah

1. 添加文件
   1. git add 文件名.txt
2. 提交文件
   1. Git commit –m “注释”
3. 查看不同

Git diff 文件名.txt

1. 查看历史版本

Git log

Git log --pretty=oneline

1. 回退版本

退回到git最新版

Git reset –-hard head^ 或者 git reset –hard head~1

将缓存区文件撤回工作区

git reset --HEAD 文件名.txt

撤销所有暂存区域文件。

git reset

1. 查看操作日志

git reflog

1. 撤回修改
   1. 还没有被放到暂存区，现在，撤销修改就回到和版本库一模一样的状态
   2. 已经添加到暂存区后，又作了修改，现在，撤销修改就回到添加到暂存区后的状态
   3. 以上传，但工作区删除，从版本库恢复到工作空间

git checkout -- 文件名

1. 工作空间删除

Rm 文件名

删除后回复从版本库git checkout -- 文件名

删除后版本库删除 git rm 文件名 ；git commit

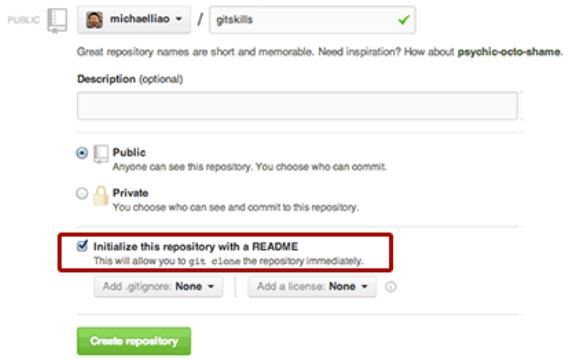
1. 链接远程仓库
   1. 在github网站注册账户
   2. 创建ssh Key
      1. 查看用户目录下是否有个.ssh ，
      2. 没有创建ssh-keygen -t rsa -C “jingleifan@126.com”
      3. 一路回车，不需要设置密
   3. 使用cd ~/.ssh ;文件下有两个文件将公钥id\_rsa.pub 内容关联git的key
   4. 本地仓库链接远程仓库：

git remote add origin git@github.com:jingleifan/git\_reposity.git

* 1. 将本地库推送到远端

git push -u origin master

1. 先建立远程仓库，在克隆到本地



* 1. 从远端克隆到本地

git clone git@github.com:jingleifan/git\_reposity.git

1. 分支
   1. 创建并切换分支：git checkout –b dev
   2. 查看当前分支 : git branch
   3. 合并某分支到当前分支：git merge <name>
   4. 合并不使用fast forward模式：git merge --no-ff -m "merge with no-ff" dev
   5. Bug分支：
      * 1. 用git stash 缓冲区暂存
        2. 恢复：

Git stash apply；之后需要git stash drop 删除

Git stash pop； 自动恢复并删除

* + - 1. 用git stash list 查看所有
      2. 合并 git pull
      3. 当前分支没有跟踪信息，需要连接远程，

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

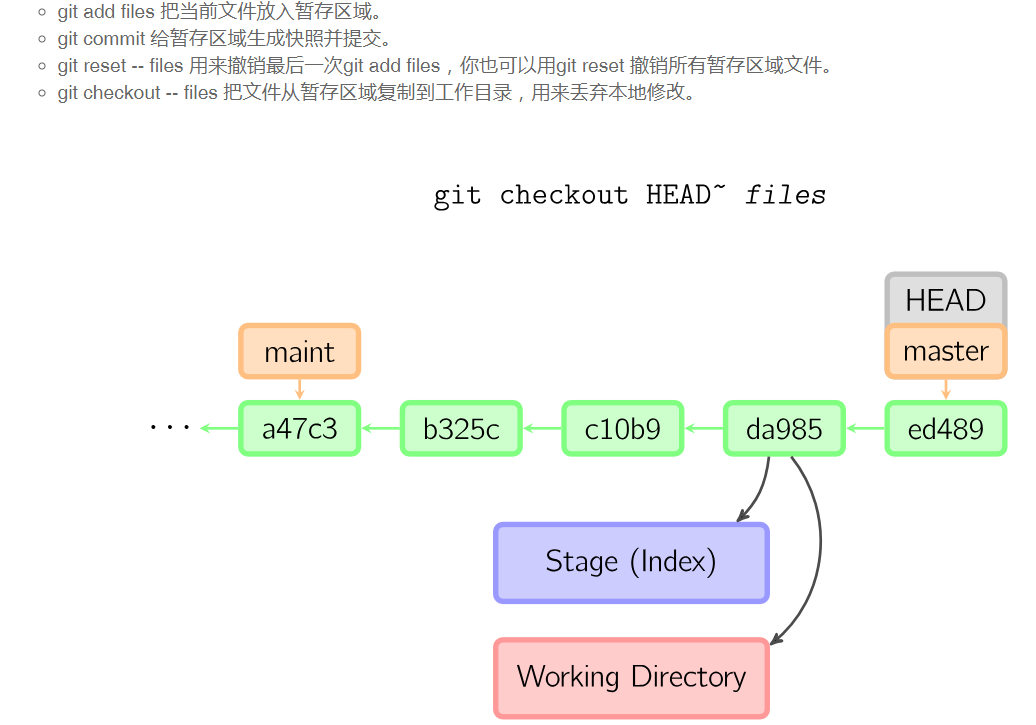
1. 标签
   1. 新建标签 git tag <tagname>
   2. 新建标签 指定标签信息 git tag –a <tagname> -m “信息”
   3. 查看所有标签 git tag
   4. 删除标签 git tag -d v0.01
   5. 推送到远端 git push origin v1.o

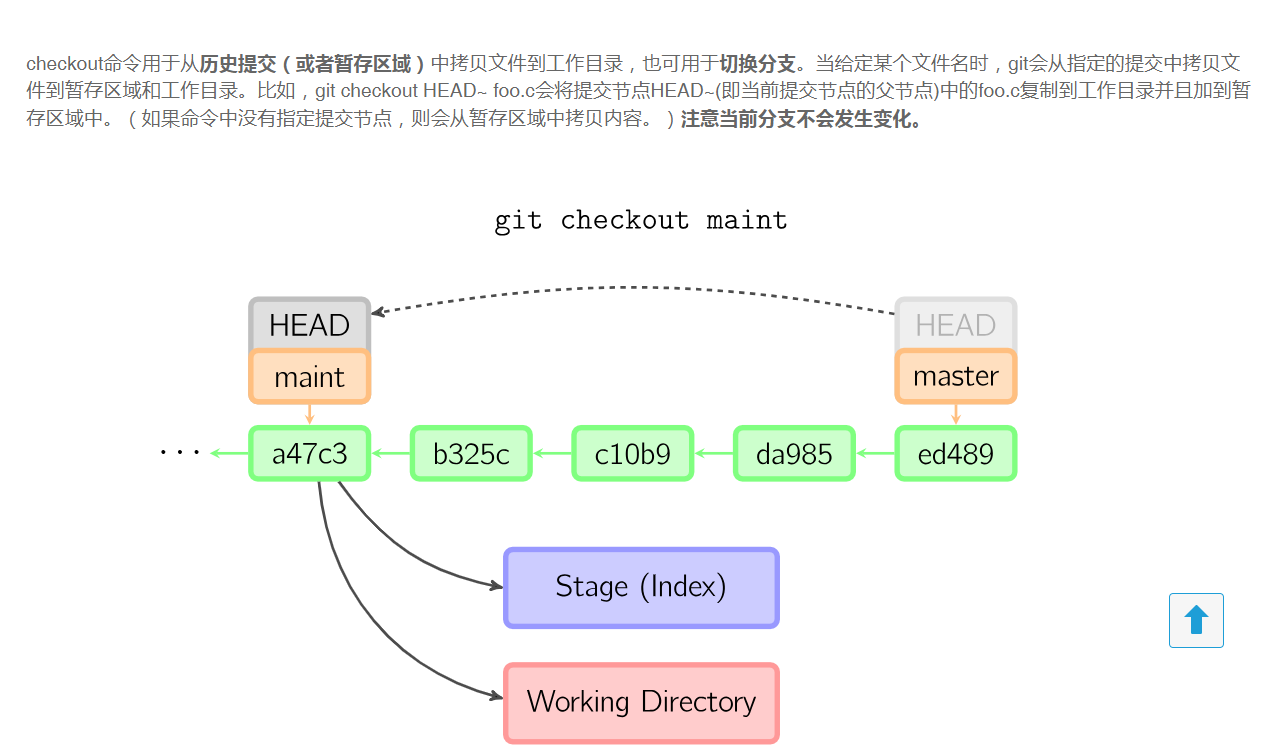
推送所有到远端： git push origin –tags

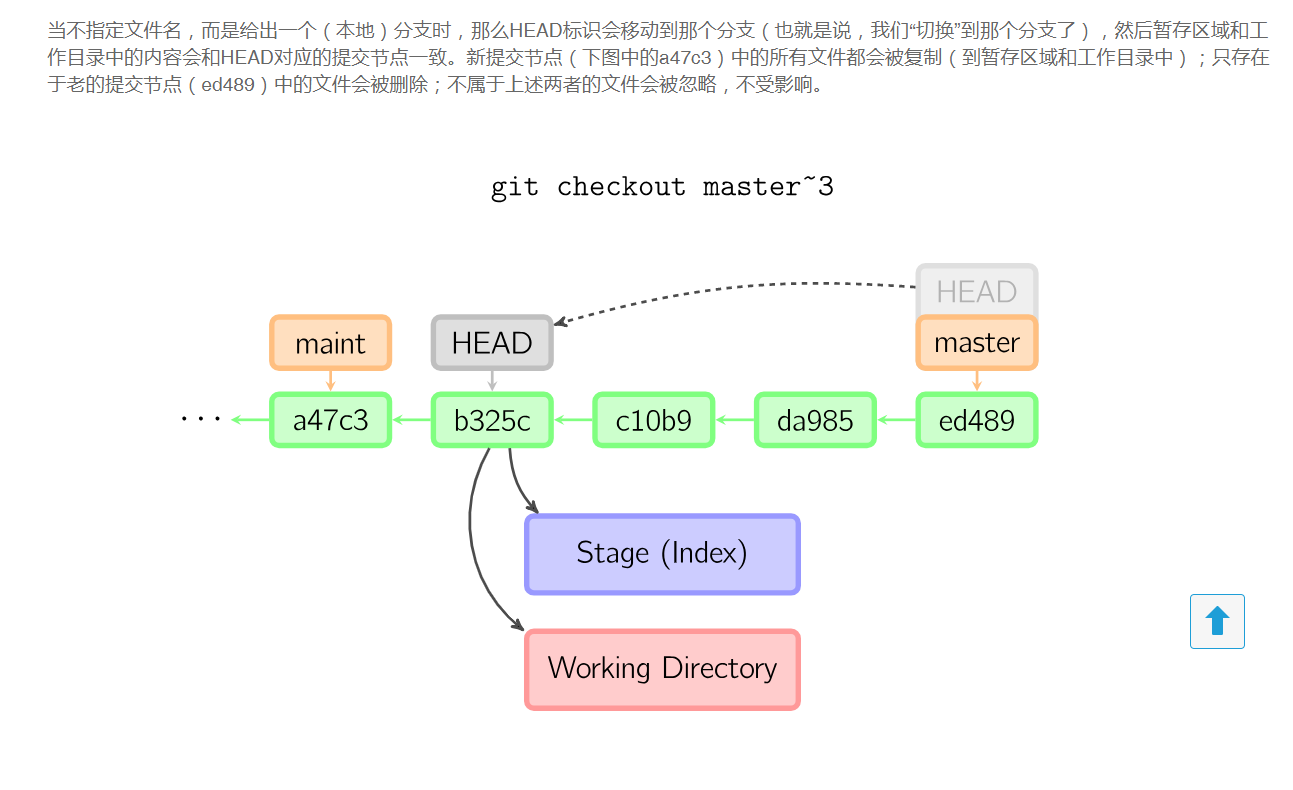
* 1. 删除远程标签，先从本地删除，再删除远端

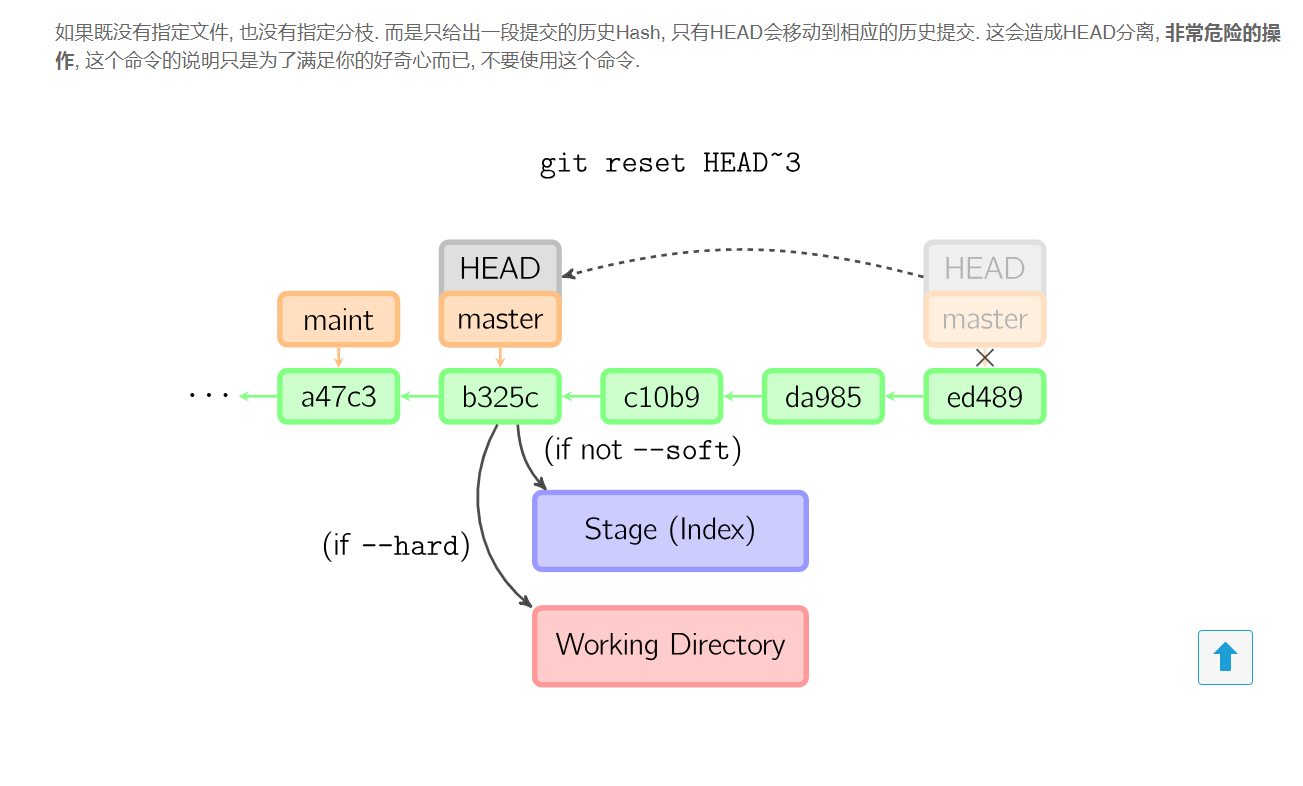
格式：git tag -d v0.01

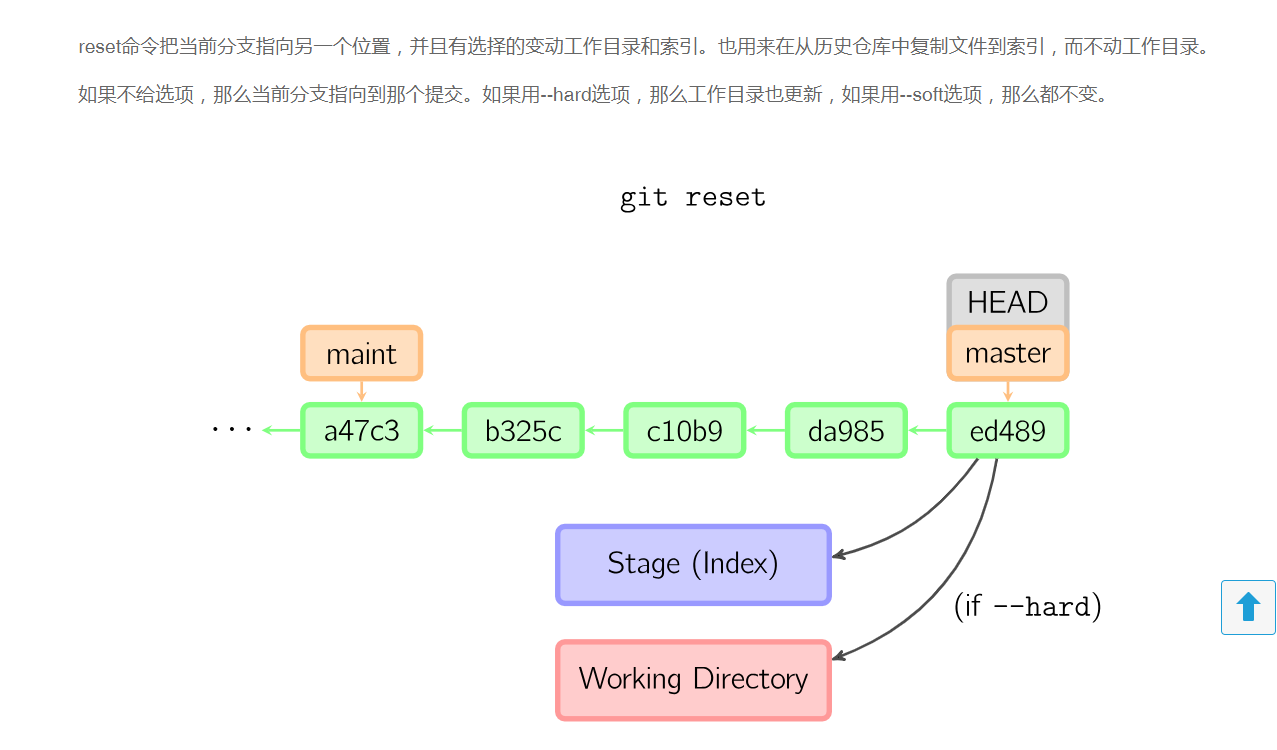
Git push origin:refs/tags/v0.01











## 帮助文档

|  |
| --- |
| 1. git init  初始化一个 Git **仓库**（repository），即把当前所在目录变成 Git 可以管理的仓库。  2. git add 文件  把文件添加到 **暂存区**（stage），可被 track 追踪纪录下来。**可多次使用来添加多个文件**。  3. git add \*  添加所有修改到暂存区，效果同 **git add all**，待验证。  4. git add -A  暂存所有的文件，包括新增加的、修改的和删除的文件。  5. git add .  暂存新增加的和修改的文件，不包括已删除的文件。即当前目录下所有文件。  6. git add -u  暂存修改的和删除的文件，不包括新增加的文件。  7. git add -i  交互式添加文件到暂存区。  8. git add -p  暂存文件的一部分。  9. git commit -m "本次提交说明"  一次性把暂存区所有文件修改提交到仓库的当前分支。注意：提交信息可为中文也可为英文，若为英文则通常用**一般现在时**。如果不加参数 -m 则会跳转到编辑器强制填写提交说明信息。  10. git commit -am "本次提交说明"  使用该命令，Git 就会自动把所有已经跟踪过的文件暂存起来一并提交，从而跳过 git add 步骤，参数 -am 也可写成 **-a -m**。**“在 oh-my-zsh 下，直接用 gcam “message” 就搞定了”**，—魔都三帅语。  11. git commit *--amend*  **重新提交**，最终只会有一个提交，第二次提交将**代替**第一次提交的结果。尤其适用于提交完了才发现漏掉了几个文件没有添加，或者提交信息写错了的情况。  12. git commit *--amend --reset-author*  在上一次 commit 之后想重新更新一下时间。amend 实际上修改了上一个 commit。所以**如果已经 push 了上一个 commit，请尽量不要 amend**。如果一定要 amend 已经 push 了的 commit，请确保这个 commit 所在的 branch 只有你一个人使用（否则会给其他人带来灾难），然后在 amend 之后使用 git push –force。只要多加小心，该命令貌似没什么卵用。  13. git commit -p  commit 文件的一部分，**适合工作量比较大的情况**。之后，Git 会对每块修改弹出一个提示，询问你是否 stage，按 y/n 来选择是否 commit 这块修改，? 可以查看其他操作的说明。 DIFF AND SHOW 1. git status  显示当前仓库的最新状态。提交之后，工作区就是“干净的”，即没有新的修改；有未提交文件时，最上面显示的是在 staging area，即将被 commit 的文件；中间显示没有 stage 的修改了的文件，最下面是新的还没有被 Git track 的文件。**“在 oh-my-zsh 下，输入 gst 就出来了，谁用谁知道，装逼利器，效率杠杠的”**，—魔都三帅语。  2. git status -s 或 git status --short  状态简览。输入此命令后，有如下几种情况（总共5种情况）：**新添加的未跟踪文件前面有 ?? 标记，新添加到暂你可能注意到了 M 有两个可以出现的位置，出现在右边的 M 表示该文件被修改了但是还没放入暂存区，出现在靠左边的 M 表示该文件被修改了并放入了暂存区。**  3. git diff  查看**工作区中的修改**。  4. git diff *--staged 或 git diff --cached*  查看**暂存区中的修改**。  5. git diff <commit id1> <commit id2>  比较两次 commit 之间的差异。  6. git diff <branch1> <branch2>  在两个 branch 之间比较。  7. git diff 文件  查看指定文件具体修改了哪些内容。  8. git diff HEAD *-- 文件*  查看版本库最新版本和工作区之间的区别，貌似没什么卵用。  9. git difftool *--tool-help*  查看系统支持哪些 Git Diff 插件，貌似没什么卵用。  10. git show  查看最后一个 commit 的修改。  11. git show HEAD~3  查看倒数第四个 commit 的修改，**HEAD~3 就是向前数三个的 commit**，即倒数第四个 commit。  12. git show deadbeef  查看 hash 为 deadbeef 的 commit 的修改。  13. git blame 文件  查看谁什么时间改了哪些文件。 LOG AND RESET 1. git log  显示从最近到最远的提交日志，包括每个提交的 SHA-1 校验和、作者的名字和电子邮件地址、提交时间以及提交说明等基本信息。  2. git log -p -2  除显示基本信息之外，还显示每次提交的内容差异，**-2** 意思是仅显示最近两次提交。**特别适用于进行代码审查，或者快速浏览某个搭档提交的 commit 所带来的变化**。  3. git log --start  显示每次提交的简略的统计信息，貌似不太好用。  4. git log --graph  查看分支合并图。  5. git log *--pretty=oneline*  简化日志信息，将每个提交放在一行显示，**查看的提交数很大时非常有用**，也可带有 **–graph** 参数，效果同**git config format.pretty oneline**。  6. git log *--graph --pretty=oneline --abbrev-commit*  查看分支的合并情况，包括分支合并图、一行显示、提交校验码缩略显示。  7. git log --oneline --decorate  查看各个分支当前所指的**提交对象（commit object）**。Git 仓库中有五个对象：三个 blob 对象（保存着文件快照）、一个树对象（记录着目录结构和 blob 对象索引）以及一个提交对象（包含着指向前述树对象的指针和所有提交信息）。  8. git log --oneline --decorate --graph --all  查看分叉历史，包括：提交历史、各个分支的指向以及项目的分支分叉情况。  9. git reset *--hard HEAD^*  回退到上一个版本。同理，回退到上上个版本为：**HEAD^ ^**， 回退到上100个版本为：**HEAD-100**，貌似波浪号 ~ 也可以，变成倒数第101个。  10. git reflog  纪录每一次命令，**可用于查找某一提交版本的 commit id**。  11. git reset *--hard <commit id>*  **回退到某一提交过的版本**，如果已经 push，则回退的意义不大了。恢复一个彻底删掉的 commit，见链接：[https://github.com/xhacker/GitProTips/blob/master/zh\_CN.md#别人-push-了修改我无法-push-了怎么办](https://github.com/xhacker/GitProTips/blob/master/zh_CN.md#%E5%88%AB%E4%BA%BA-push-%E4%BA%86%E4%BF%AE%E6%94%B9%E6%88%91%E6%97%A0%E6%B3%95-push-%E4%BA%86%E6%80%8E%E4%B9%88%E5%8A%9E)。 WORKING AREA AND REPO AREA 1. 工作区（Working Directory）  项目所在的文件目录。  2. 版本库（Repository）  工作区有一个隐藏目录文件 **.git**（可通过命令 **ls -ah** 查看隐藏文件），这就是 Git 的版本库。版本库里主要有称为**stage**的暂存区、Git 自动创建的**master** 分支，以及指向 master 的一个指针**HEAD**，表示版本库的最新版本。 RESET AND REVERT 1. git checkout *-- 文件*  丢弃**工作区**的修改，包括修改后还没有放到暂存区和添加到暂存区后又作了修改两种情况。总之，让该文件回到最近一次**git commit** 或**git add** 之后的状态。**注意：没有 – ，就变成了切换分支的命令了**。郭神的书 P195 没有 – 符号，可能是笔误。  2. git reset HEAD 文件  把**暂存区**的修改撤销（unstage），回退到工作区。**注意**：在 Git 中任何已提交的东西几乎总是可以恢复的。甚至那些被删除的分支中的提交或使用 –amend 选项覆盖的提交也可以恢复。然而，任何未提交的东西丢失后很可能再也找不到了。  3. git reset *--hard*  重置所有文件到未修改的状态。  4. git reset <commit SHA>  重置到某个 commit。  5. git reset HEAD~1  将当前 branch 重置为倒数第二个 commit（即丢弃最后一个 commit）。git reset 有三个参数可以选择，–soft、–mixed 和 –hard。  6. git reset *--soft*  修改最后一个 commit。貌似没什么卵用。  7. git revert <commit id>  还原某个 commit。还原（revert）的实质是产生一个新的 commit，内容和要还原的 commit 完全相反。比如，A commit 在 main.c 中增加了三行，revert A 产生的 commit 就会删除这三行。**如果我们非常确定之前的某个 commit 产生了 bug，最好的办法就是 revert 它**。git revert 后 git 会提示写一些 commit message，此处最好简单描述为什么要还原；而重置（reset）会修改历史，常用于还没有 push 的本地 commits。  8. git revert HEAD  还原到上次 commit。 REMOVE 1. git rm 文件  把文件从版本库中删除，不会再追踪到。  2. git rm -f 文件  **强制删除**版本库中**有修改**的文件。  3. git rm *--cached 文件*  把文件从版本库中删除，但让文件**保留在工作区且不被 Git 继续追踪（track）**，通常适用于在 rm 之后把文件添加到 .gitignore 中的情况。  4. git rm log/\\*.log  删除 log/ 目录下扩展名为 .log 的所有文件。  5. git rm \\*~  删除以 ~ 结尾的所有文件。 REMOTE REPOREMOTE AND PUSH 1. git remote  查看已经配置的远程仓库服务器，效果同 **git remote show**。  2. git remote -v  显示需要读写远程仓库使用的 Git 保存的简写与其对应的 URL。  3. git ls-remote <remote-name>  显示获得远程引用的完整列表。  4. git remote show <remote-name>  **参数 remote-name 通常都是缩写名 origin**，可以得到远程分支更为详细的信息以及 pull 和 push 相关提示信息。  5. git remote add <shortname> <url>  添加并关联一个远程库。其中，shortname 一般是 origin，也可以是其他字符串，用来代替整个 url。  6. git push  推送本地修改到 origin。  7. git push -u origin master  关联后，使用该命令**第一次**推送 master 分支的所有内容，后续再推送的时候就可以省略后面三个参数了，其中参数 u 代表上游（**upstream**）的意思。  8. git push origin 远程分支（通常是 master）  推送最新修改。**注意：多人协作时，除了 merge 可能会发生冲突之外，推送时也有可能发生冲突**。在他人推送之后是不能立即推送自己的修改的，想想也是，因为可能会覆盖他人的工作，所以必须先拉取（pull）别人的修改合并（merge）之后才能推送。如果不是第一次推送，后面的参数可省略。  9. git push <remote-name> <commit SHA>:<remote-branch\_name>  push 一部分 commit。例如：git push origin 9790eff:master 即为 push 9790eff 之前的所有 commit 到 master。  10. git remote rename old\_name new\_name  重命名一个远程仓库的简写名。  11. git remote rm <remote-name>  移除一个远程仓库。  12. git remote add origin http:*//github.com/username/<repo name>.git*  Create a remote repo named origin pointing at your Github repo (after you’ve already created the repo on Github) (used if you git init since the repo you created locally isn’t linked to a remote repo yet).  13. git remote add origin git@github.com:username/<repo name>.git  Create a remote repo named origin pointing at your Github repo (using SSH url instead of HTTP url). CLONE 1. git clone git@github.com:username/<repo name>.git  从远程库（**origin**）克隆一份到本地，仓库名同远程仓库名。  2. git clone https:*//github.com/username/repo name.git*  作用同上。但**不建议使用 https 协议**，原因有二：一是速度慢；二是每次推送必须输入口令，麻烦。但在某些只开放 http 端口的公司内部就无法使用原生的 ssh 协议而只能用 https，仓库名同远程仓库名。  3. git clone <repo url> <folder name>  克隆一个仓库到指定文件夹。  4. git clone <repo url> .  克隆一个仓库到当前文件夹（应该是空的）。 BRANCHBRANCH AND MERGE 1. git branch  列出本地当前所有分支，方便查看。当前分支前面会标有一个 **\*** 号。  2. git branch -r  查看远程分支列表。  3. git branch -a  显示所有分支，包括本地和远程。  4. git branch -v  查看每一次分支的最后一次提交。  5. git branch -vv  查看设置的所有跟踪分支。将所有的本地分支列出来并且包含更多的信息，如每一个分支正在跟踪哪个远程分支与本地分支是否是领先、落后或是都有。  6. git branch *--merged*  查看所有已经被 merge 的 branch。  7. git branch --no-merged  查看所有还没被 merge 的 branch。  8. git branch *--merged | xargs git branch -d*  删除所有已经被 merge 的 branch。  9. git checkout -b 分支  **创建**并**切换**到新的分支，相当于下面两条命令：**git branch 分支** +**git checkout 分支**。  10. git checkout -  切换到上一个 branch。  11. git cherry-pick <commit id>  假如我们在某个 branch 做了一大堆 commit，而当前 branch 想应用其中的一个，可以使用该命令。  12. git merge 分支  合并**指定分支到当前**所在的分支。  13. git merge --no-ff -m "提交说明信息" 分支  参数 **–no-ff** 表示禁用 **Fast forward 快进模式**，用普通模式合并，这样合并后的历史有分支，能看出来曾经做过合并，而 fast forwad 合并就看不出来曾经做过合并。  14. git branch -d 分支  普通删除分支（相对强制删除而言）。一般情况下，先合并完分支，然后再删除，否则会删除失败，除非使用 -D 参数强制删除。**注意：因为创建、合并和删除分支非常快，所以 Git 鼓励使用分支完成某个任务，合并后再删除分支，这个直接在 master 分支上工作效果是一样的，但过程更安全。**  15. git branch -D 分支  强行删除分支，尤其适用分支内容有了新的修改但还没有被合并的情况。  16. git push origin --delete 远程分支 或 git push origin:远程分支  删除一个远程分支。基本上这个命令做的只是从服务器上移除这个指针。 Git 服务器通常会保留数据一段时间直到垃圾回收运行，所以如果不小心删除掉了，通常是很容易恢复的。  17. git push origin -delete 分支  在本地和远程同步删除分支。  18. git rebase 目标分支（通常是 master）  在本地 master 上进行**变基**操作。**注意：merge 与 rebase 都是整合来自不同分支的修改。**   * **merge** 会把两个分支的最新快照以及二者最近的共同祖先进行**三方合并**，合并的结果是生成一个新的快照（并提交）。 * **rebase** 会把提交到某一分支**（当前分支）**上的所有修改都**转移**至另一分支**（目标分支）**上，**就好像“重新播放”一样**。 * **变基**是将一系列提交按照原有次序依次应用到另一分支上，而**合并**是把最终结果合在一起。简言之：这两种整合方法的**最终结果没有任何区别**，但是**变基使得提交历史更加整洁**。 * 采用变基操作后，项目的**最终维护者**就不再需要进行整合工作，只需要快进合并便可。 * **git rebase –ongo 目标分支 第一分支 第二分支**：选中在第二分支里但不在第一分支里的修改，将它们在目标分支（通常是 master）上重演。 * **变基有风险**，需要遵守的准则是：**不要对在你的仓库外有副本的分支执行变基**。否则，会导致混乱。总的原则是，只对尚未推送或分享给别人的本地修改执行变基操作清理历史，**从不对已推送至别处的提交执行变基操作**，这样才能享受到两种方式带来的便利。 * 还可以有这样的命令：git rebase -i master，git rebase -i 22e21f2，git rebase -i HEAD~3。  STASH 1. git stash  把当前分支的工作现场**储存**起来，等以后恢复现场后继续工作。一般适用于还没有 commit 的分支代码。  2. git stash list  查看储存的工作现场纪录列表。  3. git stash apply + git stash drop  用 git stash apply 命令恢复最近 stash 过的工作现场，但是恢复后，stash 内容并不删除，用 git stash drop 命令来删除。apply 和 drop 后面都可以加上某一指定的 stash\_id。  4. git stash pop  相当于上面两条命令，恢复回到工作现场的同时把 stash 内容也删除了。  5. git stash clear  清空所有暂存区的 stash 纪录。drop 是只删除一条，当然后面可以跟 stash\_id 参数来删除指定的某条纪录，不跟参数就是删除最近的。  6. git stash apply stash@{0}  上面命令中**大括号中的数字不是固定的**，因为可以多次 stash，恢复的时候，先用 git stash list 命令查看，然后恢复指定的 stash。  7. git biselect  发现了一个 bug，用该命令知道是哪个 commit 导致的，貌似不太好用。 PULL AND PUSH 1. git push origin 分支  把该分支上的所有本地提交推送到远程库对应的远程分支上。  2. git checkout 分支 origin/分支  如果远程有某一个分支而本地没有，怎用该命令把远程的这个分支迁到本地。  2. git checkout -b 分支 origin/分支  把远程分支迁到本地**顺便切换**到该分支。  3. git pull  抓取远程库最新提交，拉取并合并。  4. git fetch  没有 merge 的 pull。  5. git branch --set-upstream 分支 origin/分支  建立本地分支和远程分支的关联。  6. git submodule update *--recursive*  第三方依赖与远程同步，还可以在最后添加 -f 参数。 TAGINIT TAG 1. git tag  查看所有标签。**注意：标签不是按照时间列出，而是按照字母排序**，但这并不重要。  2. git show <tag-name>  查看标签信息。  3. git tag -l 'tag-name'  使用特定的模式查找标签。  4. git checkout <tag-name>  切换 tag。  5. git tag <tag name> <commit id>  在需要打标签的分支上创建一个**轻量标签（lightweight）**，默认为 HEAD，也可以指定一个 commit id。  6. git tag -a <tag-name> -m "标签说明文字" <commit id>  创建**附注标签（annotated）**，用 **-a** 指定标签名，**-m** 指定说明文字，也可以指定一个 commit id。  7. git tag -a <tag-name> 提交的校验和或部分校验和  后期打标签，即对过去的提交打标签。**校验和（checksum）：长度为 40位的16进制数的 SHA-1 值字符串**。然而，只要没有冲突，通常可以用一个比较短的前缀来表示一个 commit。  8. git tag -s <tag-name> -m "标签说明文字" <commit id>  通过 **-s** 用私钥签名一个标签。签名采用 GPG 签名，因此，必须首先按照 pgp（GnuPG），如果没有找到 gpg，或者没有 gpg 秘钥对，就会报错。如果报错，请参考 GnuPG 帮助文档配置 Key。 MANIPULATE TAG 1. git tag -d <tag-name>  删除一个本地标签。因为创建的标签都只存储在本地，不会自动推送到远程。所以，打错的标签可以在本地安全删除。  2. git push origin <tag-name>  推送本地某个标签到远程，默认情况下，git push 命令并不会推送标签到远程，必须显示推送。  3. git push origin --tags  参数 **–tags** 表示一次性推送全部未推送到远程的本地标签，当其他人从仓库中克隆或拉取，他们也能得到那些标签。  4. git push origin :refs/tags/<tag-name>  删除一个远程标签，先从本地删除，再用该命令从远程删除。  5. git checkout -b <branch-name> <tag-name>  在特定的标签上创建一个新分支，貌似没什么卵用。 CUSTOM GITIGNORE 1. git add -f 文件  使用 **-f** 参数，强制添加被 **.gitignore** 忽略的文件到 Git。  2. git check-ignore -v 文件  可能是 .gitignore 写得有问题，使用该命令找出到底哪个命令写错了。  3. [https:*//github.com/github/gitignore](https://github.com/github/gitignore)*  GitHub 上的一个十分详细的针对数十种项目及语言的 .gitignore 文件列表。  **注意：**   * 忽略某些文件时，需要编写 .gitignore 文件； * .gitignore 文件本身要放到版本库里，并且可以对 .gitignore 做版本管理。  ALIAS 1. git config --global alias.st status  使用 **git st** 代替 **git status** 命令。  2. git config --global alias.co checkout  使用 **git co** 代替 **git checkout** 命令。  3. git config --global alias.cm commit  使用 **git cm** 代替 **git commit** 命令。  4. git config --global alias.br branch  使用 **git br** 代替 **git branch** 命令。  5. git config --global alias.unstage 'reset HEAD --'  使用 **git unstage 文件** 命令代替 **git reset HEAD – 文件** 命令。  6. git config --global alias.last 'log -1'  配置一个 **git last** 命令，让其显示最近一次的提交信息。  7. git config --global alias.lg "log --color --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit"  丧心病狂地配置 **git lg** 命令，让显示 log 更加优雅，逼格更高。  8. git config --global alias.visual '!gitk'  Git 只是简单地将别名替换为对应的命令。，如果想要执行外部命令而不是一个 Git 子命令，可以在命令前面加入 **!** 符号。 如果自己要写一些与 Git 仓库协作的工具的话，那会很有用。貌似没什么卵用。 HELP  1. 1. git help <key> 2. 2. git <key> *--help* 3. 3. man git-<key> 4. 4. git *--help*  OTHERS 1. git config -l 或 git config --list  列举所有 Git 能找到的配置，如果有重复的变量名，Git 会使用它找到的每一个变量的最后一个配置。  2. git config <key>  检查 Git 的某一项配置。  3. git config *--glabal core.editor <vim/emacs/...>*  配置默认文本编辑器。  4. git config --global color.ui true  让 Git 显示颜色，使命令输出看起来更醒目。  5. git config core.ignorecase false  **Git 是大小写不敏感的**，如果要大小写敏感需要执行此命令。  6. git config --global core.quotepath false  设置显示中文文件名。  7. it config --global credential.helper cache  如果正在使用 HTTPS URL 来推送，Git 服务器会询问用户名与密码。 默认情况下它会在终端中提示服务器是否允许你进行推送。如果不想在每一次推送时都输入用户名与密码，可以设置一个 “credential cache”。 最简单的方式就是将其保存在内存中几分钟，使用该命令即可，貌似没什么卵用。   1. 8. git config --global user.name "your name" 2. git config --global user.email "your email"   设置 commit 中的姓名和 email，去掉 –global 参数则为针对每个 repo 单独设定姓名和邮箱。  9. git commit *--author "your name <your email>"*  以其他身份 commit。  10. git mv old\_filename new\_filename  重命名文件。相当于下面三条命令：   * mv old\_filename new\_filename * git rm old\_filename * git add new\_filename   11. git log 常用选项   * -p — 按补丁格式显示每个更新之间的差异。 * –stat — 显示每次更新的文件修改统计信息。 * –shortstat — 只显示 –stat 中最后的行数修改添加移除统计。 * –name-only — 仅在提交信息后显示已修改的文件清单。 * –name-status – 显示新增、修改、删除的文件清单。 * –abbrev-commit — 仅显示 SHA-1 的前几个字符，而非所有的 40 个字符。 * –relative-date — 使用较短的相对时间显示（比如，“2 weeks ago”）。 * –graph — 显示 ASCII 图形表示的分支合并历史。 * –pretty — 使用其他格式显示历史提交信息。可用的选项包括 oneline，short，full，fuller 和 format（后跟指定格式）。   12. git log --pretty=format："%h - %an, %ar : %s"  自定义 log 显示样式，也可带有 **–graph** 参数。常用的格式占位符写法及其代表的意义：   * %H — 提交对象（commit）的完整哈希字串 * %h — 提交对象的简短哈希字串 * %T — 树对象（tree）的完整哈希字串 * %t — 树对象的简短哈希字串 * %P — 父对象（parent）的完整哈希字串 * %p — 父对象的简短哈希字串 * %an — 作者（author）的名字 * %ae — 作者的电子邮件地址 * %ad — 作者修订日期（可以用 –date= 选项定制格式） * %ar — 作者修订日期，按多久以前的方式显示 * %cn — 提交者(committer)的名字 * %ce — 提交者的电子邮件地址 * %cd — 提交日期 * %cr — 提交日期，按多久以前的方式显示 * %s — 提交说明   13. git log --since=2.weeks  显示按照时间限制的 log 信息，这个时间格式可以是：**“2008-01-15”** 或 **“2 years 1 day 3 minutes ago”** 等。可用的参数还有：–until，–author，–grep(提交说明中的关键字)等。注意：如果要得到同时满足这两个选项搜索条件的提交，就必须用 –all-match 选项。否则，满足任意一个条件的提交都会被匹配出来。  14. git log -Sfunction\_name  显示添加或移除某一个特定函数的引用（字符串）的提交。  15. 限制 git log 输出的选项   * -(n) — 仅显示最近的 n 条提交 * –since, –after — 仅显示指定时间之后的提交。 * –until, –before — 仅显示指定时间之前的提交。 * –author — 仅显示指定作者相关的提交。 * –committer — 仅显示指定提交者相关的提交。 * –grep — 仅显示含指定关键字的提交 * -S — 仅显示添加或移除了某个关键字的提交   For example，**git log –pretty=”%h - %s” –author=gitster –since=”2008-10-01” \ –before=”2008-11-01” –no-merges – t/**，即为：查看 Git 仓库中，2008 年 10 月期间，作者提交的但未合并的测试文件。 |