Git Usage

Create by alex on 2018.05.07

---

1. Install

1. Install git

- $ sudo apt update
- $ sudo apt install git

2. Install tools

- $ sudo apt install tig
- $ sudo apt install git-cola
- $ sudo apt install meld
- $ sudo apt install colordiff
- $ sudo apt install diffmerge

---

2. Config

1. Config Scope

- /etc/gitconfig # 对所有用户有效 'System'
- ~/.gitconfig # 对当前用户有效 'Global'
- ./Project_Working_Dir/.git/config # 仅对当前项目有效 'Local' NOTES: 这三个级别配置的内容都是相同的，不同的是优先级：同样的配置信息的优先级是：local > global > system; 也就是说，针对同样的某个配置，如果同时存在在local和global中时，local中的配置将会覆盖global中的配置项的值，当然也会同样覆盖system中的配置项，这样做的好处是：既方便共用，又可以个性化自定义。 遵循的理念是：对于所有用户都通用的配置项放在system中；对于每个独立的用户相对的共用项放在global;对于某个仓库特殊的配置放在local.

2. Config Common

- $ git config -h # 查看帮助
- $ git config -l # 列出所有配置信息
- $ git config -e # 编辑配置
- $ git config –get # 获取某个配置项
- $ git config –add # 增加一项配置 = $ git config --local --add alias.ls "status" =
- $ git config –unset # 取消指定设置 = $ git config --local --unset alias.ls

2. Config Local

- $ git config --local -l # 列出git仓库级别配置信息
- $ git config --local -e # 编辑git仓库级别配置信息,也可以直接编辑.git/config这个文件
- $ git config --local user.name "Alex"
- $ git config --local user.email "alex.yuan@emerson.com"

3. Config Global

- $ git config --global --list
- $ git config --global user.name "alex"
- $ git config --global user.email "yxinsiva@163.com"
- $ git config --global core.editor emacs
- $ git config --global core.editor vim
- $ git config --global core.excludesfile ~/.gitignore # https://github.com/github/gitignore
- $ git config --global color.ui auto
- $ git config --global alias.glog "log --color --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit"
- $ git config --global alias.glog "log --color --graph --date=iso --pretty=format:'%Cred%h%Creset %C(bold blue)<%cn>%Creset %Cgreen(%cd)%Creset -%C(yellow)%d%Creset %s' --abbrev-commit"
- $ git config --global alias.glog "log --color --graph --date=iso --pretty=format:'%Cred%h%Creset %Cgreen(%cd)%Creset %C(bold blue)<%cn>%Creset -%C(yellow)%d%Creset %s' --abbrev-commit"
- $ git config --global alias.glog "log --color --graph --date=iso --pretty=format:'%Cred%h%Creset %Cgreen(%cd)%Creset %C(bold blue)<%cn>%Creset -%s %C(yellow)%d%Creset' --abbrev-commit"
- $ git config -e [--global]
- $ git config [--global] user.name "[name]"
- $ git config [--global] user.email "[email address]"

4. Config System

- $ git config --sytem --list

5. Config Diff

- difftool = meld

- $ git config --global difftool.prompt true
- $ git config --global alias.d difftool
- $ git config --global diff.tool meld
- $ git config --global merge.tool meld
- $ git d (hashid)

- difftool = vimdiff

- $ git config --global difftool.prompt true
- $ git config --global alias.d difftool
- $ git config --global diff.tool vimdiff
- $ git config --global merge.tool vimdiff

- $ git d (hashid)

6. Set SSH Github

- $ ssh-keygen # generate ssh key
- $ vim /home/alexyuan/.ssh/id_rsa.pub # copy ssh key and put it in github ssh key for access
- $ ssh -T git@github.com # Check the access rights and authority
- $ ssh -vT git@github.com
- $ eval "$(ssh-agent -s)" # For example feedback: Agent pid 11918
- $ ssh-add

3. Creat Repository

- Clone

- $ git clone https://github.com/yxinalex/Repository.git #HTTP
- $ git clone git@github.com:yxinalex/Repository.git #SSH

- Location

- $ git init
- $ git init [project_name]
- $ rm -rf .git # Delete local repository!!!!!

- Name

- Workspace：工作区
- Index / Stage：暂存区
- Repository：仓库区 或 本地仓库
- Remote：远程仓库
- git status 时, modified 是显示红色还是绿色，是根据于 commit 处状态来决定的
    - 显示红色是 「Woring Dir 区的 commit」与 「Staged Snapshot 的 commit / Commit History 的 commit」有区别
    - 显示绿色是 「Woring Dir 区的 commit / Staged Snapshot 的 commit」 与「Commit History 的 commit」有区别
    - 红色 modified 为「Woring Dir 区的 commit」还未被 add。
    - 绿色 modified 为「Staged Snapshot 的 commit」还未被 commit。

4. Add/Remove/Rename

- 添加指定文件到暂存区

- $ git add .
- $ git add *.c *.h *.py

- $ git add [file1] [file2] ...
- $ git add [dir]
- $ git add [dir1] [dir2] [dir3] ....

- 删除工作区文件，并且将这次删除放入暂存区

- $ git rm . -r
- $ git rm ./* -r
- $ git rm -r *.obj
- $ git rm -r *.obj *.bin *.doc *docx *.xls *.xlsx *.pdf *.graphml *.uml *.suo *.sdf *.opensdf *.idb *.pdb
- $ git rm [file1] [file2] ...
- $ git rm [dir]

- 停止追踪指定文件，但该文件会保留在工作区

- $ git rm --cached [file]
- $ git rm --cached .
- $ git rm --cached -r [dir]
- $ git rm --cached -r *.obj

- 删除掉**untracked**状态的目录

- $ git clean -fd # 使用git rm -rf dir 命令删除非空目录之后，本地还是会有空的目录存在，这时候空目录已经是untracked状态了, 再删除掉untracked状态的目录

- 改名文件，并且将这个改名放入暂存区

- $ git mv [file-original] [file-renamed]

5. Stash

- $ git stash # 想切换分支,但是不想提交正在进行中的工作，可以向堆栈上推送一个储藏保存当前变更
- $ git stash list # 查看现有的储藏
- $ git stash drop stash@{0} # 移除指定的储藏
- $ git stash apply # 重新应用最近的储藏，对工作区的变更被重新应用，但是被暂存的文件没有重新被暂存
- $ git stash apply --index # 重新应用最近的储藏，对工作区的变更和暂存区的变更，都重新应用
- $ git stash pop # 重新应用最近的储藏，并将其从栈顶移除
- $ git stash branch [branch_name] # 从储藏中创建分支

6. Commit

- $ git commit -m "comments" # 提交暂存区到仓库区
- $ git commit [file1] [file2] -m "comments" # 提交暂存区的指定文件到仓库区
- $ git commit -a # 提交工作区自上次commit之后的变化，直接到仓库区
- $ git commit -a -m "comments"
- $ git commit -v # 提交时显示所有diff信息
- $ git commit --amend # 使用一次新的commit，替代上一次提交

- $ git commit --amend -m "comments" # 如果代码没有任何新变化，则用来改写上一次commit的提交信息
- $ git cherry-pick [commit] # 选择一个commit，合并进当前分支?

7. Branch

List

- $ git branch # 列出所有本地分支
- $ git branch -r # 列出所有远程分支
- $ git branch -a # 列出所有本地分支和远程分支

Create

- $ git branch [branch-name] # 新建一个分支，但依然停留在当前分支
- $ git checkout -b [new_branch] # 新建一个分支，并切换到该分支
- $ git checkout -b [new_branch] [tag] # 新建一个分支，指向某个tag
- $ git checkout -b [new_branch] [commit] # 新建一个分支，指向某个commit
- $ git branch [new_branch] [commit] # 新建一个分支，指向指定commit
- $ git checkout -b local-b origin/remote-b # 建立新分支，建立关系，切换到新分支：基于远程分支"origin/remote-b"，创建一个叫"local-b"的分支，并切换到分支local-b
- $ git branch --track local-b origin/remote-b # 建立新分支，建立关系，不切换到新分支：新建一个分支，与指定的远程分支建立追踪关系，不切换分支，仍停留在当前分支
- $ git branch --set-upstream local-b origin/remote-b # 不建立新分支，只建立追踪关系，不切换分支：在现有分支与指定的远程分支之间建立追踪关系

Rename Branch

- $ git branch -m [branch-old-name] [branch-new-name]

Switch

- $ git checkout [branch-name] # 切换到指定分支，并更新暂存区和工作区，HEAD发生变化，指向新的Branch

Delete

- $ git branch -d [branch-name] # 删除分支
- $ git branch -D [branch-name] # Force 删除分支
- $ git push origin --delete [branch-name] # 删除远程分支
- $ git branch -dr # 删除远程分支

8. Tag

- $ git tag # 列出所有tag
- $ git tag [tag] # 新建一个tag在当前commit
- $ git tag [tag] [commit] # 新建一个tag在指定commit
- $ git tag [tag] -m [message] [commit] #新建一个tag在指定commit, 并带有注释message

- $ git tag -d [tag] # Delete one tag
- $ git push origin [tag] # 推送Tag到远程
- $ git push origin --tags # 通过--tags 参数来推送所有本地的Tag
- $ git pull origin [tag] # 取回Tag从远程
- $ git pull origin --tags # 通过--tags 参数来取回所有远程的Tag

删除远程Tag.

- $ git tag -d <Tag 名字> # 当本地Tag已经Push到远程代码仓库后，再要删除这个Tag,就必须删除本地Tag.
- $ git push origin :refs/tags/<Tag 名字> # 删除本地Tag后，再重新Push到远程的代码仓库。

NOTES: The real difference between a tag and a branch is that the branch moves with the Commit, but the tag does not. As described in the section "misunderstanding branches", when Git advances a Commit, the branch it is in moves forward with it. But once the tag is posted, no matter how committed it is, the tag stays where it was posted. So you can think of branches as moving labels. There is a famous line in the movie cape no.7: "stay, or I will go with you", which is used in the concept of "stay is the label, and go with you is the branch".

9. Check

- $ git status # 显示有变更的文件
- $ git reflog # 显示当前分支的最近几次提交
- $ git blame [file] # 显示指定文件是什么人在什么时间修改过
- $ git log # 显示当前分支的版本历史
- $ git log -n # 显示当前分支的最近n次的提交版本历史
- $ git log --all
- $ git log --graph
- $ git log --color --graph
- $ git log --stat # 显示commit历史，以及每次commit发生变更的文件
- $ git log --stat -3 # 显示commit历史，以及每次commit发生变更的文件，但只显示最近3个commit
- $ git log --follow [file] # 显示某个文件的版本历史，包括文件改名
- $ git whatchanged [file] # 显示某个文件的版本历史，包括文件改名
- $ git log -p [file] # 显示指定文件相关的每一次diff
- $ git log --color --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit
- $ git log --color --graph --date=iso --pretty=format:'%Cred%h%Creset %C(bold blue)<%cn>%Creset %Cgreen(%cd)%Creset -%C(yellow)%d%Creset %s' --abbrev-commit
- $ git log --color --graph --date=iso --pretty=format:'%Cred%h%Creset %Cgreen(%cd)%Creset %C(bold blue)<%cn>%Creset -%C(yellow)%d%Creset %s' --abbrev-commit
- $ git log --color --graph --date=iso --pretty=format:'%Cred%h%Creset %Cgreen(%cd)%Creset %C(bold blue)<%cn>%Creset -%s %C(yellow)%d%Creset' --abbrev-commit
- $ git log --pretty="%h - %s" --author=alex --after="2018-05-10" --before="2018-05-20" --grep="Comments Message" --no-merges
- $ git log --pretty="%h - %s" --committer=alex --after="2018-05-10" --before="2018-05-20" --grep="Comments Message" --no-merges
- $ git log commit # 查询commit之前的记录，包含commit
- $ git log commit1 commit2 # 查询commit1与commit2之间的记录，包括commit1和commit2
- $ git log commit1..commit2 # 同上，但是不包括commit1
- $ git log tag1 # 查询标签tag1之前的commit

- $ git log tag1 tag2 # 查询标签tag1和tag2之间的commit
- $ git log tag1..tag2 # 同上，但是不包括tag1和tag2
- $ git glog -3 # 显示commit历史，但只显示最近3个commit
- $ git glog --stat -3 # 显示commit历史，以及每次commit发生变更的文件，但只显示最近3个commit
- $ git glog --author=alex --after="2018-05-10" --before="2018-05-20" --grep="Comments Message" --no-merges
- $ git glog --committer=alex --after="2018-05-10" --before="2018-05-20" --grep="Comments Message" --no-merges
- $ git show --name-only [commit] # 显示某次提交发生变化的文件
- $ git show [commit] # 显示某次提交的元数据和内容变化
- $ git show [tag]
- $ git show [tag] [commit]
- $ git show [branch]
- $ git show [branch] [commit]
- $ git checkout # 汇总显示工作区、暂存区与HEAD的差异
- $ git checkout HEAD # 汇总显示工作区、暂存区与HEAD的差异

10. Diff

diff file level

- $ git diff [file] # 显示工作区和暂存区的差异
- $ git diff --cached [file] # 显示暂存区和HEAD(或上一个commmit)的差异
- $ git diff --staged [file] # 同上
- $ git diff HEAD [file] # 显示工作区和当前分支的HEAD的差异
- $ git diff [branch] [file] # 工作区的文件与branch 分支的文件进行比较
- $ git diff [branch1] [branch2] [file] # 比较两个分支的指定文件的差异
- $ git diff [commit] [file] # 工作区与某一次提交进行比较
- $ git diff [commit1] [commit2] [file] # 比较两个commit的指定文件的差异
- $ git diff -- cached [commit] [file] # 比较暂存区与指定commit-id的差异
- $ git diff -- cached [branch] [file] # 比较暂存区与指定branch-id的差异

diff commit level

- $ git diff # 显示工作区与暂存区之间的差异
- $ git diff --cached # 显示暂存区与当前分支最新commit之间的差异
- $ git diff HEAD # 显示工作区与当前分支最新commit之间的差异
- $ git diff HEAD --cached # 显示暂存区与当前分支最新commit之间的差异
- $ git diff HEAD --stat # 显示工作区与当前分支最新commit之间的文件差异列表
- $ git diff HEAD HEAD^
- $ git diff [branch1] [branch2] # 显示两个branch之间的差异
- $ git diff [commit1] [commit2] # 任意两次commit之间的差别
- $ git diff [branch1] [branch2] --stat # 显示两个branch之间的差异的文件列表，文件新增和删除的总行数，用‘+’和‘-’的数量表示新增和删除的比例
- $ git diff [commit1] [commit2] --stat # 显示两个commit之间的差异的文件列表，文件新增和删除的总行数，用‘+’和‘-’的数量表示新增和删除的比例

- $ git diff [commit1] [commit2] --stat-width=60 --stat-name-width=10 --stat-count=3 # 显示两个commit之间的差异的文件列表，一行显示的总宽度，文件名显示宽度，文件列表显示个数
- $ git diff [commit1] [commit2] --stat=60,10,3 # 同上，简写
- $ git diff [commit1] [commit2] --shortstat # 显示两个commit之间的差别信息：文件变化的总数量，新增行数的总数量，和删除行数的总数量
- $ git diff [commit1] [commit2] --numstat # 显示两个commit之间的差别的文件列表，分别显示文件新增的行数和删除的行数
- $ git diff [commit1] [commit2] --dirstat #同下
- $ git diff [commit1] [commit2] --dirstat=changes #显示两个commit之间的差别以文件夹为单位
- $ git diff [commit1] [commit2] --dirstat=lines #显示两个commit之间的差别以文件夹为单位
- $ git diff [commit1] [commit2] --dirstat=files #显示两个commit之间的差别以文件夹为单位
- $ git diff [commit1] [commit2] --ignore-space-at-eol #比较两个commit之间的差别,Ignore changes in whitespace at EOL.
- $ git diff [commit1] [commit2] -b, --ignore-space-change #比较两个commit之间的差别,Ignore changes in amount of whitespace. This ignores whitespace at line end, and considers all other sequences of one or more whitespace characters to be equivalent.
- $ git diff [commit1] [commit2] -w, --ignore-all-space #比较两个commit之间的差别,Ignore whitespace when comparing lines. This ignores differences even if one line has whitespace where the other line has none.
- $ git diff [commit1] [commit2] --ignore-blank-lines #比较两个commit之间的差别,Ignore changes whose lines are all blank.

difftool = vimdiff, meld, diffmerge

- $ git difftool HEAD
- $ git difftool HEAD --stat
- $ git d HEAD
- $ git d HEAD --stat

11. Redo

checkout file level

- $ git checkout . # 用暂存区的所有文件直接覆盖本地文件，HEAD不变

NOTES: If "$ rm ./* -r", then "$ git checkout ."

- $ git checkout [file] # 用暂存区的指定文件重置工作区，对象是暂存区的file，HEAD不变
- $ git checkout [commit] [file] # 用commit的指定文件重置暂存区和工作区，对象是commit history中的commit，HEAD不变，注意会将暂存区和工作区中的filename文件直接覆盖
- $ git checkout [branch] [file] # 用branch的指定文件重置暂存区和工作区，对象是commit history中的branch，HEAD不变，注意会将暂存区和工作区中的filename文件直接覆盖
- $ git checkout HEAD [file] # 用HEAD 的指定文件重置暂存区和工作区，对象是commit history中的HEAD，HEAD不变，注意会将暂存区和工作区中的filename文件直接覆盖

- $ git checkout HEAD^^ [file_path] #用HEAD之前的第二个commit的指定文件重置暂存区和工作区，对象是commit history中的commit，HEAD不变
- $ git checkout HEAD . # 用HEAD 重置暂存区和工作区，对象是commit history中的HEAD，HEAD不变

NOTES: If "$ git rm ./* -r", then it is required "$ git checkout HEAD ."

- $ git checkout HEAD [file_path] # 用HEAD 中指定路劲的文件夹 重置暂存区和工作区，对象是commit history中的HEAD，HEAD不变
- $ git checkout [commit] [file_path] # 用commit 中指定路劲的文件夹 重置暂存区和工作区，对象是commit history中的HEAD，HEAD不变
- $ git checkout [branch] [file_path] # 用branch 中指定路劲的文件夹 重置暂存区和工作区，对象是commit history中的HEAD，HEAD不变

##NOTES: "git checkout [...]", precondition: new file/folde in stage or delete file/folde in working, the following:

- 1. These file/folde is not reset by commit to stage/working.
- 2. These file/folde is not reset by stage to working.
- 3. Whatever, the "checkout" is just reset file/fold of (working)(stage/working) which thse file/fold is exist in (stage)(commit).
- 4. In this case, plesae follow the reminder from git-bash, to "reset, add/rm, checkout"

checkout commit level

- $ git checkout # 汇总显示工作区、暂存区与HEAD的差异
- $ git checkout HEAD # 汇总显示工作区、暂存区与HEAD的差异
- $ git checkout HEAD^ # 用Commit History中的HEAD^ 重置 Staged 和 Working，HEAD发生变化指向HEAD^，Be in 'detached HEAD' state
- $ git checkout HEAD^^ # 用Commit History中的HEAD^^重置 Staged 和 Working，HEAD发生变化指向HEAD^，Be in 'detached HEAD' state

NOTES: If be in 'detached HEAD' state, use command to restore : $ git checkout [master or branch]

NOTES: If be in 'detached HEAD' state, use command to new branch : $ git branch [branch_name]

- $ git checkout [branch] # 用指定的Branch重置Staged和Wroking，HEAD发生变化指向新的Branch，(即就是，切换到指定分支，并更新暂存区和工作区)

- $ git checkout [commit] # 用指定的commit重置Staged和Working，HEAD发生变化指向commit，并且是匿名分支，Be in 'detached HEAD' state

NOTES: If be in 'detached HEAD' state, use command to restore: $ git checkout [master or branch]

NOTES: If be in 'detached HEAD' state, use command to new branch : $ git branch [branch_name]

reset file level

- $ git reset [file] # 重置暂存区的指定文件，与上一次commit保持一致，工作区不变，HEAD不变 # NOTES: "git reset [file]" doesn't have "--soft, --mixed, --hard" parameter
- $ git reset [Commit] [file] # 用commit中指定的文件重置暂存区的指定文件，工作区不变，HEAD不变
- $ git reset HEAD [file] # 用HEAD中指定的文件重置暂存区的指定文件，工作区不变，HEAD不变

reset commit level

- $ git reset HEAD # HEAD不变，用HEAD重置暂存区，但工作区保持不变
- $ git reset HEAD --hard # HEAD不变，用HEAD重置暂存区和工作区
- $ git reset HEAD^ # 重置当前分支的HEAD为HEAD之前的第一个版本(回退版本，一个^表示一个版本)，同时重置暂存区，但工作区保持不变
- $ git reset HEAD^^ # 重置当前分支的HEAD为HEAD之前的第二个版本(回退版本，一个^表示一个版本)，同时重置暂存区，但工作区保持不变
- $ git reset HEAD~3 # 重置当前分支的HEAD为HEAD之前的第三个版本(回退版本，一个^表示一个版本)，同时重置暂存区，但工作区保持不变
- $ git reset [commit] --soft # 重置当前分支的HEAD为指定commit，但是保持暂存区和工作区不变
- $ git reset [commit] --mixed # 重置当前分支的HEAD为指定commit，同时重置暂存区，但工作区保持不变
- $ git reset [commit] # 同上 --mixed
- $ git reset [commit] --hard # 重置当前分支的HEAD为指定commit，同时重置暂存区和工作区，与指定commit一致
- $ git reset [commit] --keep # ?

NOTES: "git reset [file]" & "git checkout [file]" Difference:

```
                                    # checkout 只能同时重置工作区和暂存区
                                    # reset file 或者 reset --mixed 能只重置暂存区而不重置工作区
```

- Sample:
- $ git reset HEAD ./Project/main.c # 用reset file 命令重置暂存区，主要是重置暂存区。如果只想重置暂存区，但又不想重置工作区，需要用reset file

- $ git checkout HEAD . # 用checkout 命令重置暂存区和工作区，主要是重置工作区，因为checkout不能只重置工作区，它是工作区和暂存区一起重置

revert

- $ git revert [commit] # 新建一个commit，用来撤销指定commit， 后者的所有变化都将被前者抵消，并且应用到当前分支

```
|-----------------------------------------------------------------------------------|
|working |index |HEAD |target-commit |args                  |working |index |HEAD   |
|-----------------------------------------------------------------------------------|
|A       |B     |C    |D             |--soft                |A       |B     |D      |
|        |      |     |              |--mixed               |A       |D     |D      |
|        |      |     |              |--hard                |D       |D     |D      |
|        |      |     |              |--merge (disallowed)  |        |      |       |
|-----------------------------------------------------------------------------------|

|-----------------------------------------------------------------------------------|
|working |index |HEAD |target-commit |args                  |working |index |HEAD   |
+-----------------------------------------------------------------------------------+
|A       |B     |C    |C             |--soft                |A       |B     |C      |
|        |      |     |              |--mixed               |A       |C     |C      |
|        |      |     |              |--hard                |C       |C     |C      |
|        |      |     |              |--merge (disallowed)  |        |      |       |
|-----------------------------------------------------------------------------------|
```

12. Merge

- $ git merge [branch] # 合并指定分支到当前分支
- $ git merge [origin/master] # 将远端master分支的代码merge进本地当前分支,?
- $ git mergetool # 使用merge工具解决merge冲突

13. Remote

Init Remote Repository

- $ git clone https://github.com/yxinalex/Repository.git #HTTP
- $ git clone git@github.com:yxinalex/Repository.git #SSH

Look/Add/Remove/Rename Remote

- $ git remote -v # 显示所有远程仓库
- $ git branch -vv # 显示本地分支跟踪的远程分支
- $ git remote show [remote] # 显示某个远程仓库的信息
- $ git remote add [shortname] [url] # 增加一个新的远程仓库，并命名
- $ git remote remove [shortname_origin] # Remove one remote repository
- $ git remote rename origin-old-name origin-new-name # Rename the origin name
- Sample-1:

- $ git remote add origin git@github.org:alex/repository.git # First, create remote connection (When you have local repository)(Please enusere the remote server has already this repository)
- $ git push -u origin master # Second, push master to remote origin
- Sample-2:
- $ git remote set-url origin git@github.org:alex/repository.git # First, create remote connection
- $ git push -u origin master # Second, push master to remote origin

Create Tracked Remote-Location

- $ git checkout -b origin/remote-b # 建立新分支，建立关系，切换到新分支：基于远程分支"origin/remote-b"，创建一个叫"origin/remote-b"的分支，并切换到分支orign/remote-b
- $ git checkout -b local-b origin/remote-b # 建立新分支，建立关系，切换到新分支：基于远程分支"origin/remote-b"，创建一个叫"local-b"的分支，并切换到分支local-b
- $ git branch --track local-b origin/remote-b # 建立新分支，建立关系，不切换到新分支：新建一个分支，与指定的远程分支建立追踪关系，不切换分支，仍停留在当前分支
- $ git branch --set-upstream local-b origin/remote-b # 不建立新分支，只建立追踪关系，不切换分支：在现有分支与指定的远程分支之间建立追踪关系

Fetch Remote

- $ git fetch [remote] # 下载远程仓库的所有变动
- $ git fetch [remote] [branch] # 取回特定分支的更新
- $ git fetch origin master
- $ git fetch origin local-b:remote-b # 取回远程remote-b到本地local-b，不切换分支仍停留在当前分支，不merge remote-b到当前分支
- $ git log -p FETCH_HEAD # 取回更新后，会返回一个FETCH_HEAD ，指的是某个branch在服务器上的最新状态

可以在本地通过它查看刚取回的更新信息, 通过这些信息来判断是否产生冲突，以确定是否将更新merge到当前分支

- $ git log -p master..origin/master
- $ git merge origin/master
- $ git merge FETCH_HEAD # 将拉取下来的最新内容合并到当前所在的分支中
- $ git checkout -b new_branch_for_fetch # 将拉取下来的最新内容 place 本地指定新的分支
- $ git fetch --all
- Smaple-1:
- $ git fetch --all
- $ git fetch origin
- $ git fetch origin HumDehumFeature
- $ git checkout HumDehumFeature

- $ git checkout -b HumDehumFeature origin/HumDehumFeature

Pull Remote

- $ git pull --all
- $ git pull [remote] [tag] # 取回远程仓库的变化，并与本地分支合并
- $ git pull [remote] [branch] # 取回远程仓库的变化，并与本地分支合并
- $ git pull origin remote-master-name # 取回远程仓库上指定分支，并与本地当前分支合并
- $ git pull origin remote-branch-name # 取回远程仓库上指定分支，并与本地当前分支合并
- $ git pull origin remote-b:local-b # 使用远程的对应分支来更新对应的本地分支，不切换分支，但会Merge remote-b到当前分支

Push Remote

- $ git push [remote] [tag] # 提交指定tag
- $ git push [remote] [branch] # 上传本地指定分支到远程仓库
- $ git push origin remote-master-name # 推送本地当前分支的变化，到远程仓库上指定分支
- $ git push origin remote-branch-name # 推送本地当前分支的变化，到远程仓库上指定分支。如果远程仓库没有此分支名，将创建新的分支 remote-branch-name
- $ git push origin local-b:remote-b # 使用本地的对应分支来更新对应的远程分支
- $ git push -u origin master # 把origin设置成upstream, 以后就可以直接使用不带别的参数的git pull从之前push到的分支来pull

如果当前分支与多个主机存在追踪关系，则可以使用-u选项指定一个默认主机，这样后面就可以不加任何参数使用 git push

- $ git push [remote] --force # 强行推送当前分支到远程仓库，即使有冲突
- $ git push [remote] --all # 不管是否存在对应的远程分支，将本地的所有分支都推送到远程主机，这时需要使用–all选项
- $ git push [remote] --tags # 提交所有tag

Delete Remote

- $ git branch -r # Look all of remote branches
- $ git push origin --delete remote-b # Delete the remote branch
- $ git push origin :remote-b # Same above (Push local empty branch to remote branch as delete remote branch)
- $ git push origin --delete tag remote-tag # Delete the remote tag
- $ git tag -d tag-name # The other way to delete the remote tag
- $ git push origin :refs/tags/tag-name # Above contine...
- $ git branch -r -d origin/remote-b # Delete remote-tracking branch origin/remote-b, just delete local tracking
- $ git remote add origin # 彻底删除/替换git远程仓库, 将某个工程向现在的仓库强制推送即可
- $ git push --force --set-upstream origin master # Above contine...