--------------------------------------------------

git简易使用手册

--------------------------------------------------

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

http://www.git-scm.com/book/zh/起步-Git-基础

# git 与 其他版本控制系统的主要差别

git 是直接记录快照 ，而非差异比较-

Git 和其他版本控制系统的主要差别在于:

Git 只关心文件数据的整体是否发生变化，

而大多数其他系统则只关心文件内容的具体差异。

# git 中文件的三种状态

committed 已提交 : 该文件已经被安全地保存在本地数据库中了

modified 已修改(未暂存) : 修改了某个文件，但还没有提交保存

staged 已暂存 : 把已修改的文件放在下次提交时要保存的清单中

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

git初始化设置

//

// 设置用户名 与 邮箱 , 这些设置将被记录在 ~/.gitconfig 文件中

//

$ git config --global user.name "Edison Zhang"

$ git config --global user.email "edisonewtong@163.com"

**// 查看某一设置**

$ git config user.name

// 查看全部设置

$ git config --list

// 删除某一设置

$ git config --unset --global user.name

//

// 设置命令的别名

//

// 全局命令 别名 被记录在 /etc/gitconfig or /usr/local/etc/gitconfig 配置文件 中

$ git config --system alias.st status // git st <--> git status

// 本用户的全局配置的 命令 别名 被记录在 ~/.gitconfig

$ git config --global alias.st status // git st <--> git status

// git 命令输出中开启颜色显示

$ git config --global color.ui true

// 版本库级别 ( 在工作区内的 .git/config 文件 ) cd gitdir; git config -e

// 全局 ( 在 ~/.gitconfig ) // git config -e -global

// 系统级 ( 在 /etc/gitconfig or /usr/local/etc/gitconfig // git config -e --system

git 配置文件的优先级顺序 版本库 > 全局 ( ~/.gitconfig ) > 系统级 ( /etc/gitconfig )

// 查看git 某个Command 的说明

**git help <command>**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

git init

// version 1

$ cd aaa

$ mkdir gittestr

$ cd gittest

$ git init

// version 2

$ cd aaa

$ git init gittest // create a directory , and init it as a git repo

$ cd gittest

// 显示版本库的 .git 目录所在的位置

$ git rev-parse --git-dir

// 显示工作区的根目录

$ git rev-parse --show-toplevel

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

http://www.git-scm.com/book/zh/Git-基础-取得项目的-Git-仓库

有2种方法取得Git项目的仓库

1. 在当前工作目录中创建新仓库 git init

2. 从已有仓库克隆 git clone <url> [new\_folder\_name] // 注意是 clone 而不是 checkout

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

2.2 Git 基础 - 记录每次更新到仓库

git status

// 查看本地仓库中的文件状态

$ git status

// 显示精简状态的输出

$ git status -s

**// 只显示在版本管理控制内的文件的 status , 不显示 未加入版本管理的 新增文件**

$ git status -u**no**

**// 只显示 path 指定的路径的文件 或者目录的状态**

$ git status -- <path>

**// 只显示 被忽略掉的文件，这些文件可能是在 .gitignore中被指定的，也可能通过 git rm --cached <filename> 来指定**

$ git status **--ignored**

git add

// 将指定的新文件加入到版本控制中

// 或 将已经在版本控制[ 已被修改过 ]的文件存入 已暂存 区域

$ git add <file\_name> // file\_name 可以是通配符

// 将当前文件夹下(递归所有子文件夹)下的[未]加入已暂存区域的文件 加入 已暂存区域

$ git add .

.gitignore 文件

可以在 git 仓库根目录下新建一个名为 .gitignore的文件

在这个文件中，可以添加要忽略的文件

\*.[oa] // 忽略所有 .o 或 .a 结尾的文件

//

// example

// .gitignore

# 此为注释 – 将被 Git 忽略

# 忽略所有 .a 结尾的文件

\*.a

# 但 lib.a 除外

!lib.a

# 仅仅忽略项目根目录下的 TODO 文件，不包括 subdir/TODO

/TODO

# 忽略 build/ 目录下的所有文件

build/

# 会忽略 doc/notes.txt 但不包括 doc/server/arch.txt

doc/\*.txt

# ignore all .txt files in the doc/ directory

doc/\*\*/\*.txt

git diff

// 本地文件 <--> 暂存区域

// 查看当前文件 与 暂存区域(可能未提交)进行比较

$ git diff <file\_name>

// 已经暂存 <--> 最近一次已经提交

// 查看已经暂存文件 与 上次提交时的快照的差异

$ git diff --cached <file\_name> // git 1.6.1 及更高还可以用 --staged 替换 --cached

// 本地文件 <--> 最近一次已经提交

// 查看当前文件 与 上次已提交时的快照的差异

$ git diff HEAD <file\_name>

// 版本a <--> 版本b

$ git diff 5023c241696dfc185bab29bbebd36ad0b70d1ad8 bf2dff73ac5662e445e2408b6b1df81626de1593

// 版本a file <--> 版本b file

$ git diff 5023c241696dfc185bab29bbebd36ad0b70d1ad8 bf2dff73ac5662e445e2408b6b1df81626de1593 <file\_name>

// 版本a file <--> 版本b 2个版本的文件修改记录

$ git diff 5023c241696dfc185bab29bbebd36ad0b70d1ad8 bf2dff73ac5662e445e2408b6b1df81626de1593 **--name-stauts**

// 版本a 相比较他的前一个版本的 提交修改，包括文件的具体修改详情

$ git show e7297e969

// 版本a 相比较他的前一个版本的 提交日志，只显示文件的修改状态

$ git **show** e7297e969 **--name-stauts**

git commit

// 提交暂存区域的内容

$ git commit -m "commit messages : .... "

// 将已经修改过的文件加入到 暂存区域 后，然后再提交

$ git commit -a -m "commit messages : ......"

// 如果前一次的提交有漏掉的文件没有添加，或者仅仅想修改上一次提交时的提交记录 , 可以用这个命令，**此时会弹出一个提交窗口，然后你可以修改上一次的提交日志**

**// 最终你只会有一个提交 - 第二次提交将代替第一次提交的结果**

$ git commit **--amend**

// 以下3行命令只是对 'initial commit' 的补充，并不生成新的提交记录

e.g.

$ git commit -m 'initial commit'

$ git add forgotten\_file

$ git commit --amend

git rm

// 将文件从git仓库中删除(也同时从本地磁盘中删除)

$ git rm <file\_name>

// 将文件从git仓库中删除(也同时从本地磁盘中删除),

// 如果此文件已经被修改过,并已经放入暂存区域的文件

$ git rm -f <file\_name>

// 将文件从git仓库中删除跟踪 ( 但是, 不从本地磁盘删除 )

$ git rm --cached <file\_name>

git mv

// 文件移动位置 或 文件重命名

$ git mv file\_from file\_to

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

2.4 Git 基础 - 撤消操作

git reset

// 将已经暂存，但还未提交的文件，从暂存区移除( 本地文件[不会] 被覆盖回 已经暂存时的文件 )

$ git reset HEAD <file\_name>

git checkout

// 版本回滚 | 取消对文件的修改 (文件会被回滚到 上次暂存时的内容)

$ git checkout -- <file\_name>

**// 回滚单个文件到某一个指定的版本**

$ git checkout commit-id -- file\_name

**// 版本回滚到 指定的 版本**

$ git checkout 5023c241696dfc185bab29bbebd36ad0b70d1ad8 **// 游离状态**

$ git checkout -b rollback **5023c241696dfc185bab29bbebd36ad0b70d1ad8** **// 非游离状态，新建了一个分支名为rollback**

$ git reset --hard 5023c241696dfc185bab29bbebd36ad0b70d1ad8 **// 强制回退**

// 版本更新到 最新的 版本 (如果本地的当前分支是 master)

$ git checkout master // 注意本地分支 默认名字就是 master

// **如果想回滚到之前一个版本，但是还是作为一些新的提交** (如A版本只修复了a.cpp ，修复了一个Bug, n个版本过去后，现在是F版本，发现由于A版本的a.cpp的改动又导致了新的问题出来，那么想回滚这个A版本的a.cpp , 并且作为一次新的提交，G版本)

$ **git revert** <A-HashCode>

// 回滚n个版本，但保留回滚操作的日志

e.g.

$ git log

34e204f EdisonZhang Wed May 29 10:45:21 2019 +0800 5. add multiply invoke code in main.cpp

2d082ca EdisonZhang Wed May 29 10:40:10 2019 +0800 4. add multiply func in func.cpp

57007a0 EdisonZhang Wed May 29 10:23:59 2019 +0800 3. add tmp varible in main.cpp

4fcbf88 EdisonZhang Wed May 29 10:17:40 2019 +0800 2. add tmp varible in func.cpp

f12b4a6 EdisonZhang Wed May 29 10:08:51 2019 +0800 1st. Build Successful

如果想n次回滚，生成n条新的回滚操作的日志, 假定，想回滚后，回到 4fcbf88 日志的状态，

即要回滚 34e204f 2d082ca 57007a0 , 这3条操作

// 注意 hashCodeA..hashCodeB 是一个区间表示法，即**前开后闭** ( hashCodeA , hashCodeB ]

// 如果想 前闭后闭，那么就用 hashCodeA**^**..hashCodeB ==> [hashCodeA , hashCodeB]

// 因此，想回滚后，使回滚后的结果与 4fcbf88 一致，也可以是

// $ git revert 4fcbf88..34e204f ，因为区间是 (4fcbf88 , 34e204f] 与 [ 57007a0, 34e204f]是相同的区间

$ git revert 57007a0**^**..34e204f // 然后进入 Vim的编写提交日志的界面，然后不断地 :wq 就可以了

$ git log

**fb734e8 EdisonZhang Wed May 29 14:23:29 2019 +0800 Revert "3. add tmp varible in main.cpp**

**76107a7 EdisonZhang Wed May 29 14:23:28 2019 +0800 Revert "4. add multiply func in func.cpp"**

**62b7717 EdisonZhang Wed May 29 14:23:26 2019 +0800 Revert "5. add multiply invoke code in main.cpp"**

34e204f EdisonZhang Wed May 29 10:45:21 2019 +0800 5. add multiply invoke code in main.cpp

2d082ca EdisonZhang Wed May 29 10:40:10 2019 +0800 4. add multiply func in func.cpp

57007a0 EdisonZhang Wed May 29 10:23:59 2019 +0800 3. add tmp varible in main.cpp

4fcbf88 EdisonZhang Wed May 29 10:17:40 2019 +0800 2. add tmp varible in func.cpp

f12b4a6 EdisonZhang Wed May 29 10:08:51 2019 +0800 1st. Build Successful

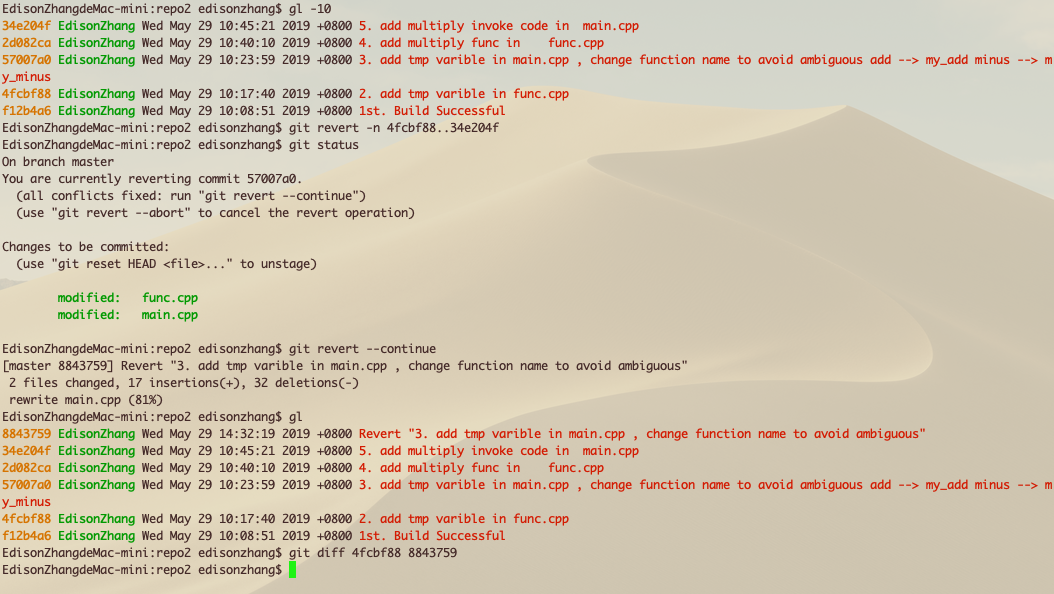
$ git diff 4fcbf88 **fb734e8**

**结果显示，没有差异，说明回滚达到了预期**

**// 如果想多条合并成1条，可以添加参数 -n, 详细操作见下图**

**// gl <==> git log**

$ git revert -n 57007a0**^**..34e204f



git log

// --stat 参数可以看到每次提交的文件变更统计

$ git log --stat

// --pretty=oneline 参数让log以精简格式输出

$ git log --pretty=oneline

// 设置 git log 的 color

// %C(yellow) 表示使用黄色

// %Creset 表示还原回默认的颜色

// %h 表示使用短的Hash值

**// %cn 表示提交者的名字 | %an表示作者的名字 ，作者是第1次提交的人，提交者是之后对原作者的内容进行修改的人**

// %cd 表示提交时间

// %s 表示提交时的说明日志

$ **git log --pretty='%C(yellow)%h%Creset %C(red)%cn%Creset %cd %C(green)%s%Creset'**

// 在历史刻录中，查找 添加或者删除了某个关键字的日志 **-S<key\_word>**

// 比如： 我在某一次添加了一个函数叫 int minus(int a, int b) { return a-b; }

// 就能找到这一条日志

$ **git log -Sminus**

// 列出从 (A , B] 的所有日志

$ **git log <commit-A>..<commit-B>**

// e.g git log fixbug103 --not master <==> git log fixbug103..master

$ git log <branch-name-behind> --not <branch-name-after>

// 以分支流程的图形化的方式显示日志

**$ git log --graph**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

3.2 Git 分支 - 分支的新建与合并

git branch

// 新建一个基于当前版本的分支br1，但是不切换到分支 br1

$ git branch br1 // based on current version

// 新建一个基于以前某一个版本(5023c)的分支br2 , 但不切换到分支 br2

$ git branch br2 **5023c241696dfc185bab29bbebd36ad0b70d1ad8**

$ git checkout br1 // 切换到br1

$ git checkout br2 // 切换到br2

// 新建分支 并 切换到这个 新建的分支

$ git checkout -b <new\_branch\_name>

==

$ git branch <new\_branch\_name> // 新建一个分支,但当前分支还在 master

$ git checkout <new\_branch\_name> // 将分支切换到 new\_branch\_name

**// 基于某个旧的 <base\_hash\_id> 新建一个分支，并切换到这个新建的分支上**

**$ git checkout -b <new\_branch\_name> <base\_hash\_id>**

git checkout

// 切换分支到 指定的分支 existed\_branch\_name

$ git checkout <existed\_branch\_name>

//

// 当在进行分支合并时，发现有冲突conflicted files 的文件时，可以使用以下命令，选用本地或者合并的那个分支的文件

//

$ git branch

**\* fix**

**master**

$ git rebase master // 与之类似的是 $ git checkout master && git merge fix

<<<<<<< HEAD

a = 1122; // --ours **master** 部分

=======

a = 1133; // --theirs **fix** 部分

>>>>>>> n1, 11-->1133

// **--theirs** 表示**使用** 当前分支**fix**的修改，**弃用**目标分支**master**的修改

$ git checkout **--theirs** -- somepath/<conflicted\_file>

// **--ours** 表示**使用**目标分支**master**的修改，弃用**当前**分支**fix** 的修改

$ git checkout **--ours** -- somepath/<conflicted\_file>

git merge

$ git checkout master

// 将 existed\_branch\_name 分支合并到当前的分支 ( master )

$ git merge <existed\_branch\_name>

// git branch -d testing， 注意如果这个分支是还没有进行合并的分支，可能会报错

// **error: The branch 'testing' is not fully merged.** // **If you are sure you want to delete it, run 'git branch -D testing'.**

// 删除某个指定的分支

$ git branch -d <existed\_branch\_name>

// 查看所有分支(包含本地和远程的)

git branch -a

// 查看本地分支

git branch

// 查看远程分支

git branch --remote

// **本地分支 与远程分支 的 关联**

$ git branch --set-upstream-to origin/devtest devtest

// or

$ git branch -u origin/devtest devtest

**// 删除远程分支**

git push origin --delete <RemoteBranchName>

git push origin :<RemoteBranchName>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

2.5 Git 基础 - 远程仓库的使用

git remote

// 查看远程分支 , show remote repo info

$ git remote

$ git remote -v

git remote add

// 添加一个新的远程仓库

$ git remote add <shortname> <url>

git fetch

// 拉取远程仓库中所有你本地仓库中还没有的数据 ，并不自动合并到当前工作分支

$ git fetch <remote-name>

**// fetch后， 新建一个分支 与 远程已经下载下来的分支进行关联**

**$ git checkout -b <local-branch-name> <fetched-remote-repo-name>/<remote-repo's branch-name>**

**假定远程一个名为 newFeature2020 的分支(本地的分支名可以和远程的分支名不一致，但是为了能够1-1对应，建立起明确的对应关系，因此最好本地的分支名和远程的起相同的名称)**

e.g.

$ git checkout -b newFeature2020 origin/newFeature2020

// 从远程仓库，拉取一个分支到本地，但是 \*\*\* 不合并 \*\*\*

$ git fetch <remote-name> +<remote-branch-name>:<local-branch-name>

e.g.

**$ git fetch origin +remoteBranch:localBranch** // 如果本地仓库中不存在 localBranch 这个分支，上面的命令会在本地创建这个分支 localBranch

git pull

// 拉取远程分支，并合并到要本地仓库的当前分支

$ git pull <remote-name> <branch-name>

git push

// 推送(上传)数据到远程仓库

$ git push <remote-name> **<branch-name>**

等价于

$ git push <remote-name> refs/heads/**<local\_branch-name>:**refs/heads/**<remote\_branch-name>**

$ git push <remote-name> **<local\_branch-name>:<remote\_branch-name>**

// e.g. 将会推送本地的dev分支 --> 远程的 remote\_dev分支

$ git push origin dev:remote\_dev

git remote show

// 查看远程仓库的详细信息

$ **git remote show** <remote-name>

git remote rename

// 重命名远程仓库

$ git remote rename <old-name> <new-name>

git remote rm

// 删除远程仓库

$ git remote rm <remote-name>

**// Export un-versioned repo**

// 导出不带版本信息的文件夹

git **archive** -o latest.zip HEAD // 基于最新提交建立归档文件latest.zip

git **archive** -o partial.tar HEAD src doc // 只把目录src和doc建立到归档partial.tar中

git **archive** --format=zip -o xxx.zip HEAD // 设置档案文件格式为 zip

git **archive** --format=tar -o xxx.tar HEAD // 设置档案文件格式为 tar

//

// forcelly roll Back && forcelly roll to new

//

git log

\* 98abc5a (HEAD, master) more stuff added to foo

\* b7057a9 initial commi

// 非强制回滚

git checkout b7057a9

// 强制回滚到某个旧的版本 删除之前的提交内容

git reset --hard b7057a9

// 查看 checkout 与 reset的记录

git reflog

b7057a9 HEAD@{0}: reset: moving to b7057a9

98abc5a HEAD@{1}: commit: more stuff added to foo

b7057a9 HEAD@{2}: commit (initial): initial commit

// 强制回到最新的版本

git reset --hard 98abc5a

git clean命令用来从你的工作目录中删除所有没有tracked过的文件

git clean经常和git reset --hard 一起结合使用.

记住reset只影响tracked 的文件, 所以需要clean来删除untracked 的文件.

结合使用这两个命令能让你的工作目录完全回到一个指定的<commit>的状态

// 是一次clean的演习, 告诉你哪些文件会被删除. 记住他不会真正的删除文件, 只是一个提醒

git clean -n

// 删除当前目录下所有untracked 的文件. 他不会删除.gitignore文件里面指定的文件夹和文件, 不管这些文件有没有被track过

git clean -f

// 删除指定路径下的untracked 的文件

git clean -f <path>

// 删除当前目录下没有untracked 的文件和文件夹

**git clean -df**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

tag

// 显示本地tag list

git tag

// 在当前的commit-id 处 ， 创建一个本地的轻量标签 (lightweight) ,

git tag <tag\_name>

// 创建一个附注(附加了注释的 annotated )标签

git tag -a <TagName> -m ' The tag commit message '

// **后期 补打**标签之前某一次提交的Tag

$ git tag -a <TagName> <commit-id>

// 删除本地标签

git tag **-d** <tag\_name>

// 删除远程 标签

git push origin <branch\_name> **:<tagName>**

// 推送 tag 到远程服务器仓库

git push origin <branch\_name> **--tags**

// 查看远程所有 tag

git ls-remote --tags origin

// 显示某一 tag 的详细情况

git show <tag\_name>

// 切换到某一标签所在的历史代码

git checkout <tag\_name>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

stash

// 列出当前所有的暂存数据

git stash **list**

// 将当前的工程区，已经修改过但没有add过，或者已经add，但没有commit 的修改记录 暂存起来

**git stash**

// 暂存你的修改，并用 message 进行标识

git stash **push -m "your stash message"**

// 列出 某一个 暂存项的，文件列表

git stash **show <stash-id>**

// 列出 某一个 暂存项的，文件列表, 以及具体的改动的内容

git stash show **-p** <stash-id>

// 丢弃掉 指定的 <stash-id>

git stash drop <stash-id>

// 弹出 (堆栈顶部) 最近一次的 stash-id , 还原回之前正在修改的状态

git stash **pop**

**游离头指针（detacthed HEAD)**

**在“游离头指针”状态下，如果你做了某些更改然后提交它们，标签不会发生变化，但你的新提交将不属于任何分支，并且将无法访问，除非确切的提交哈希。**

**// 切换到一个 非游离状态的tag分支 ，名为 version2 , tag名为 v2.0.0**

$ **git checkout -b version2 v2.0.0** Switched to a new branch 'version2'

**Git 别名**

$ git config --global alias.co checkout $ git config --global alias.br branch $ git config --global **alias.ci** **commit** //  git **commit** 时，只需要输入 git **ci** $ git config --global alias.st status

**Git HEAD 指针**

那么，Git 又是怎么知道当前在哪一个分支上呢？ 也很简单，它有一个名为 **HEAD**的特殊指针。。 **在 Git 中，HEAD它是一个指针，HEAD指针：指向当前所在的本地分支**

你可以简单地使用 git log 命令查看**各个分支当前所指的对象**。 提供这一功能的参数是 **--decorate**

$ git log --oneline --decorate f30ab (HEAD, master, testing) add feature #32 - ability to add new 34ac2 fixed bug #1328 - stack overflow under certain conditions 98ca9 initial commit of my project