Git

# Git相关名词的基本概念



本地仓库

暂存区

工作区

## 工作区 workspace:

## 暂存区 Index:

## 本地仓库 Repository:

## 远端 Remote:

## HEAD:

### git HEAD存放位置

HEAD指的就是 .git/HEAD文件, 它存储着但概念分枝的名字, 可以打开看看:

**ref: refs/head/master**

由此,我们可以知道当前处于master分枝. 继续往下走: 打开refs/heads/master 文件, 是一串密码:

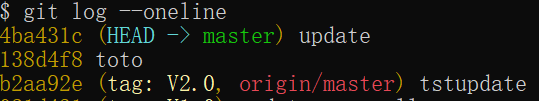
1. 7e136f508b982790db5686482075c60ee3ee4fed

这是master分枝上最新提交的commit Id.

### git HEAD的作用

1. 作用一: 显示当前所处于哪个分枝

我们在查看git提交日志的时候,会经常碰到HEAD, 如:



HEAD -> master : 告诉我们当前处于master分枝

1. 作用二: 指代当前分枝的最新提交的commit Id

有时候,我们commit提交代码后, 发现这一次commit的内容有错误的, 面对这种情况有两种解决方法:

###### 修改错误的内容,再次commit一次;

###### 使用git reset命令撤销只一次错误的commit;

此时, head出场了, 它经常与reset连用, 如下所示:

Text

Description automatically generated

[Git HEAD 详细介绍 - MyBatis中文官网](http://www.mybatis.cn/archives/2096.html)

# 准备

## Git 安装:

* 1. 进入git官网: [Git (git-scm.com)](https://git-scm.com/)
  2. 点击下载并安装:



## git help: 查看帮助信息;

## 初始化配置:

### git config –global user.name “osun” : 配置用户名

### git config –global user.email [osun@sensirion.com](mailto:osun@sensirion.com) : 配置邮箱

### git config –list : 列出当前配置

### git config –local –list : 列出本地仓库的配置

### git config –global –list : 列出全局配置

### git config –system –list : 列出系统配置

### git config –global merge.tool <vimdiff> : 配置解决冲突时使用哪种分析工具

### git config –global color.ui auto : 配置git命令输出为彩色的

### git config –global core.editor vi :配置git使用的文本编辑器

### 注:

也可以去配置文件 .gitconfig里直接修改,默认在: C:\Users\用户\.gitconfig

# 基本概念

# 查询命令

## git ls-files : 查看本地repository里的文件

即已经本地commit的文件

## git show [options] <object> : 显示信息

什么都不加: 显示的是上一次commit时更改的内容;

tagName : 标签名, 显示该标签修改的内容

commitId: 显示该次commit时更改的内容

## git remote show <远端名称> :打印对应远端的基本信息

# 工作区操作命令

## git checkout <被修改了的文件> : 取消修改

如果文件被修改了,**还没提交到暂存区**,用此命令可以取出本地仓库中的文件进行覆盖.

## git checkout -- <文件名> : 撤销上一次对文件的操作

前提条件是文件还没有提交到暂存区.

这个命令和git checkout <文件名> 比较类似, 目前没发现有不同.

## git checkout HEAD <file> : 撤销未提交文件的修改

## git diff <文件全称>:

查询工作区文件和git仓库中的文件的区别，也就是修改后还没暂存 ,所以这一步需要在add之前;

## git add <文件名/路径>/. : 提交新文件/修改的文件到暂存区

指定添加的文件/路径, 或 '.' 表示所有

## git mv <原文件名> <新文件名> : 重命名文件并提交到暂存区

## git restore --staged <文件/路径> : 从暂存区退到工作区

备注:

## git restore <文件名> : 放弃文件的修改

## git rm:

在 Git 中，术语 rm 代表删除。它被用来删除单个文件或一个文件集合。git rm 的主要功能是将跟踪的文件从 Git 索引中移除。此外，它还可以用来从工作目录和暂存索引中删除文件。

被删除的文件必须是该分支要删除的理想文件。在索引中不能有对其内容的更新。否则，删除过程可能会很复杂，有时甚至不会发生。但它可以通过 -f 选项强制进行。

### git rm <文件名/路径> : 删除工作区文件，并且也从暂存区删除对应文件的记录

注: 删除的是工作区文件,需要重新commit及push操作.该文件还是被追踪的.

### git rm -f <文件/路径>: 从工作区和暂存区删除

### git rm –cached <文件名/路径> : 从暂存区删除

上面的删除方式,即使文件被删除了,但是还是会被git追踪,也就是说所有的记录都再,随时可以找回来.

如果不想让git追踪这个文件,也就是不想分享这个文件.

从暂存区中删除文件，但是工作区依然还有该文件, 工作区文件图标上会显示,

## cat 文件名: 查看文件内容

显示文件的整个内容

# 暂存区操作命令

## git commit : commit操作

直接git commit的话, 会怎样?

答案: Git 会尝试为你打开一个编辑器以填写提交信息。 如果 Git 在你对它的配置中找不到相关信息，默认会打开 vim。如下图：

Text

Description automatically generated

然后git自动打开vim, 你需要把备注信息编辑在此处. 操作方式参考vim.

Text

Description automatically generated

## git commit -a : 修改后的文件直接commit

等同于: git add . + git commit

另一种写法: git commit –all

注意: 新建的文件除外.

## git commit –amend : 将最近的修改追加到上一次的提交上。

比如: 刚刚commit后,发现漏改了个东西, 然后补上,又不想把这个作为一个独立的commit内容,这时可以用git commit --amend

## git commit -C : 重新使用之前提交过的信息去提交新的更改

## git commit -c : 与 -C 相比多了一项修改编辑的步骤

## git commit -m "<comments>": 加备注commit

带着comments commit

另一种写法: git commit --message “comments”

描述 ：使用给定 comments 作为提交消息，即 Git 就不会打开编辑器了，再让你填备注信息了

**注意** ：-m 和 -c -C -F 是相互排斥的！。

## git commit -am : 添加并提交

或: git commit -a -m “info”

如: git commit -am “info”

功能同git add . + git commit -m “info” , git commit -am可以省略git add . 这一步.

**了解 -a的意义就明白了:**

-a, --all  
Tell the command to automatically stage files that have been modified and >deleted, but new files you have not told Git about are not affected.

自动把当前所有修改和删除文件放到栈上，但你没有添加过的不会受影响。

## git commit -v : 提交时显示所有改动信息

## git commit -a -v :

## git commit -p : 通过创建新提交来替换当前分支的提交信息。

另一种写法: git commit --patch

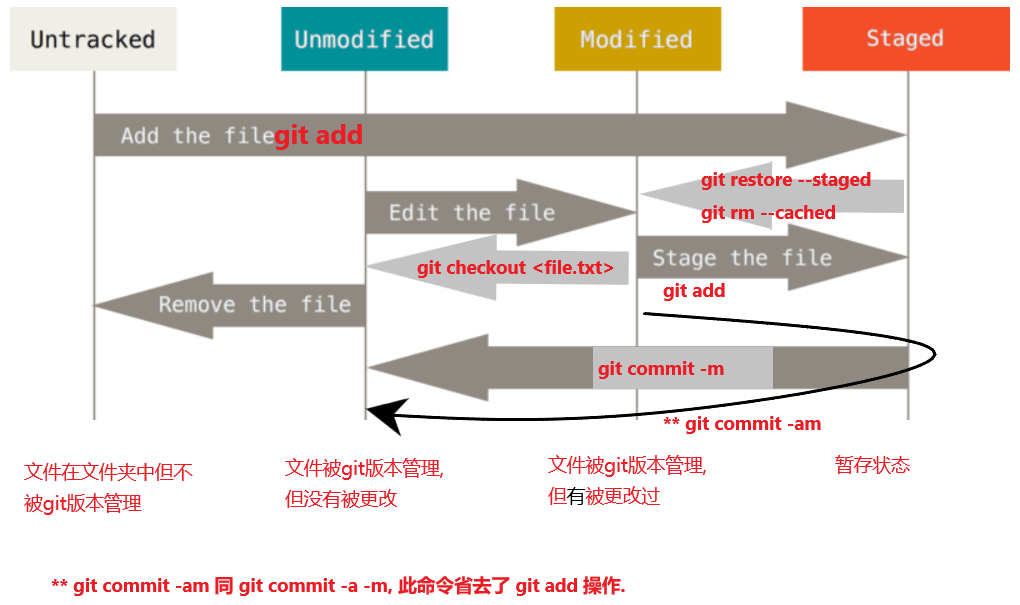
## git diff --cached / git diff --staged : 比较仓库中与暂存区的差异

git diff –cached <文件名> : 查看指定文件的差异

## git reset HEAD <文件名> : 取消暂存区已经暂存的文件

# 本地仓库操作命令

## 本地文件的四种状态



Untracked: 未跟踪, 此文件在文件夹中, 但并没有加入到git库, 不参与版本控制. 通过git add 状态变为Staged.

Unmodify: 文件已经入库, 未修改, 即版本库中的文件[快照](https://so.csdn.net/so/search?q=%E5%BF%AB%E7%85%A7&spm=1001.2101.3001.7020)内容与文件夹中完全一致. 这种类型的文件有两种去处, 如果它被修改, 而变为Modified. 如果使用git rm移出版本库, 则成为Untracked文件

Modified: 文件已修改, 仅仅是修改, 并没有进行其他的操作. 这个文件也有两个去处, 通过git add可进入暂存staged状态, 使用git checkout 则丢弃修改过, 返回到unmodify状态, 这个git checkout即从库中取出文件, 覆盖当前修改

Staged: 暂存状态. 执行git commit则将修改同步到库中, 这时库中的文件和本地文件又变为一致, 文件为Unmodify状态. 执行git reset HEAD filename取消暂存, 文件状态为Modified

## git init : 初始化本地仓库

在建好的文件夹下面执行该命令, 然后文件夹里会多个 .git文件夹,说明初始化成功;

## git remote add [远程别名] [远程仓库ssh地址]: 关联本地和远端仓库

前提条件:

1. 本地有一个仓库,并已经初始化
2. 远端也有一个已经创建好的仓库

如:

Text

Description automatically generated

可见, git remote add xxx: 只是将本地仓库和远程仓库进行了关联,分枝并未建立联系,所以还需通过下面两种方式将本地分枝和远程分枝建立关系:

1. git pull <远程主机名> <远程分支名>:<本地分支名>:
   1. 本地仓库是空的Text

      Description automatically generated
   2. 本体仓库不为空:会有如下错误

Text

Description automatically generated

**解决办法:**

* + - 1. 将远程master分枝pull到本地的非master分枝,如bhtst分枝;
      2. 将bhtst和本地的master分枝**强制**合并.

如:

Text

Description automatically generated

1. git push <远程主机名> <本地分支名>:<远程分支名>:

Text

Description automatically generated

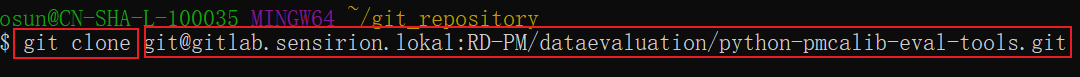
## git clone <ssh/https> : 将远程仓库克隆到本地

* 1. 进入远程仓库,可以选择”Clone with SSH”(需要先SSH key 配置)或者”clone with Https”

Graphical user interface, text, application

Description automatically generated

* 1. 打开gitbash,将远程仓库克隆到本地:



## git branch –set-upstream-to=origin/remote\_branch local\_branch

将本地分枝和远端分枝关联起来:

origin : 远端仓库地址

remote\_branch : 远端分枝

local\_branch : 本地分枝

**比如:** git branch –set-upstream-to=origin/SEN4XSW3.0 SENSW3.0

其它代码方式:

1. git branch -u origin/SEN4XSW3.0 SEN4XSW3.0

2. git branch -u origin/SEN4XSW3.0

## git status: 查看状态

Text

Description automatically generated

绿色是已经添加了的文件,红色是还没有添加的;

* 1. git restore --staged <file>: 将file从要提交的清单中移除;
  2. git restore <file>: 放弃对file的修改

## git log : 查看提交历史

### git log作用:

可以查看历史提交记录, 会输出信息: commit hash, author, date, commit message.

如:

Text

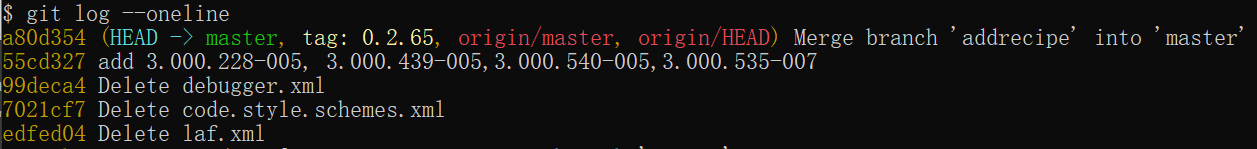
Description automatically generated

### git log参数介绍:

git log参数众多, 选择合理的参数,可以方便查看国王的仓库提交的历史记录. 主要参数有:

#### **--oneline :** 简化默认的输出

仅仅输出 commit hash 前7个字符串和 commit message, 如:



#### **--stat:** 增加增删的统计数据, 方便我们查看仓库的改动情况

如:

A picture containing graphical user interface

Description automatically generated

#### --author: 过滤作者

在团队合作开发过程中，如果只想查看某个小伙伴提交的代码，怎么办呢？此时可以加上 --author 参数用来过滤 commit，限定输出给定的用户，如下所示：

Text

Description automatically generated

#### --before, --after : 限定指定日期范围的日志

Text

Description automatically generated

#### --p : 输出每次提交的内容差异

p 参数很常用，它能控制输出每个 commit 具体修改的内容，输出的形式以 diff 的形式给出。如下所示：

Text

Description automatically generated

#### **-n :** 限定log输出的条数

如果想看最近的 n 条提交日志，怎么办？很简单。直接在 log 命令之后，加 -n 参数即可.

## git show : 查看提交日志及格式化

### git show介绍

git show 可以用于显示提交日志的相关信息。它与 git log 的区别在于，git show 侧重于显示版本差异，输出每个commit具体修改的内容，而 git log 则侧重于显示版本的提交历史，文末有更多关于两者的介绍。

### git show xxx : 查看日志

**默认显示:** 直接git show默认显示的是HEAD, 如:

Text

Description automatically generated

**commit id :** 显示指定commit Id的提交信息,如:

Text

Description automatically generated

**commitid filename :** 指定具体某次commitid及具体文件

Text

Description automatically generated

**tagName : 实际这个等同于传了个commit id.**

Text

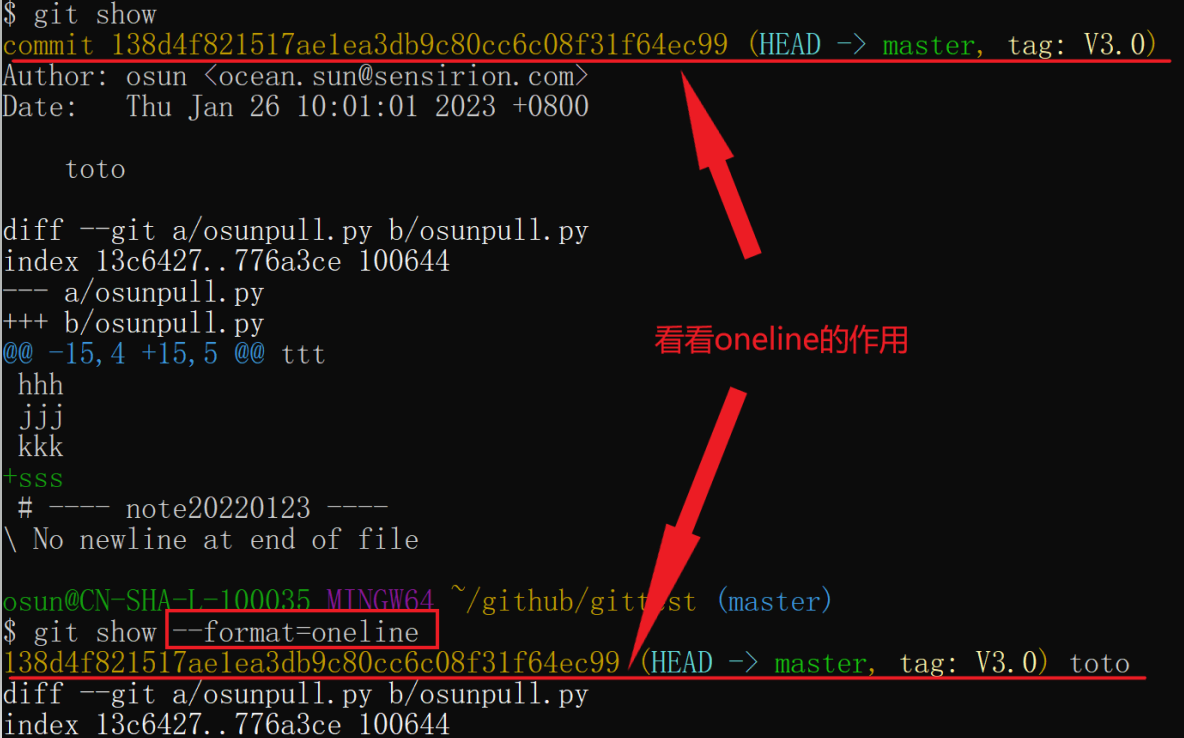
Description automatically generated

### git show <格式化> : 对显示的内容进行格式化

git show --pretty[=<format>] | --formate=<format>

--pretty=format:short与 – format=short两者是等价的.

**oneline:** 以一行的格式输出 完整的commit id和提交的备注

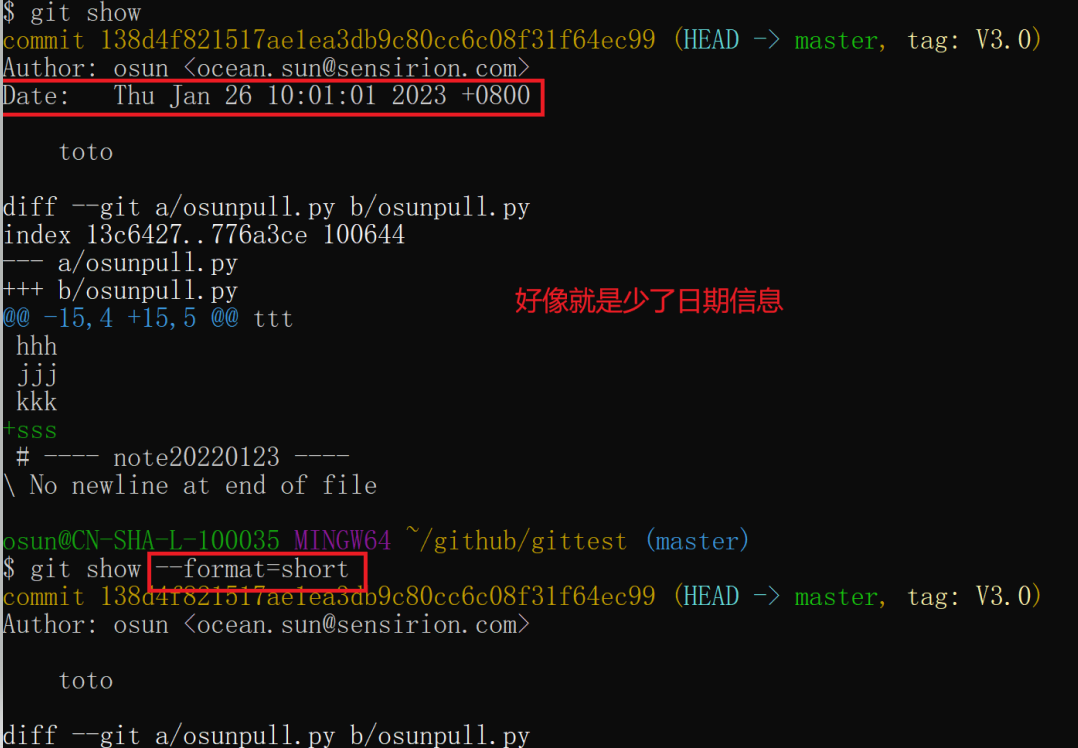


**short:** 以短格式输出提交的日志: commit id, 作者, 备注

**medium :** 默认, 输出: commit id, 作者, 日期, 备注

**full :** 输出: commit id, 作者, commit(看上去也是作者), 备注

**fuller :**  输出: commit id, 作者, 作者日期, commit, commit日期, 备注



**email:** 以Email格式输出, 这个看起来比较习惯

**raw :** 以原始格式输出提交日志

### git show VS git log :

git show命令同git log -p输出类似，只不过它只显示一个commit的内容，如果不指定commit hash，它默认输出HEAD指向commit的内容。

**备注**：git log -p：控制输出每个commit具体修改的内容，输出的形式以diff的形式给出。

## git blame <file> : 以列表方式查看当前文件每一行的最后一次更改记录

## git diff HEAD HEAD^ : 上一次commit与上一次commit的父亲的差异

## 取消本地仓库和远端仓库的关联

只要删除本地仓库里的 .git文件夹即可. 比如: **rm -rf .git**

## rm -rf .git: 删除本地仓库

该种方式是强制删除,有时候文件夹不为空它不让删除.

## 本地分枝的主要操作

git branch 命令允许你创建、列出、重命名和删除分支。对分支的许多操作都是由git checkout和git merge命令应用的。所以，git 分支与 git checkout 和 git merge 命令是紧密结合的。

### git branch : 查看本地分枝

### git branch -r : 查看远程分枝

### git branch -a : 查看所有分枝

### git branch <name>: 创建本地分枝

如: git branch test1, 创建test1分枝

### git checkout : 切换分枝 / 放弃修改

git checkout，应该是Git中最简单，但也是最难以理解的命令吧。checkout 本意是检出的意思，跟“切换”往往不太相关，但它往往用于切换分支，让人感觉有点莫名其妙。其实，git checkout 的作用有两个：第一个作用是切换分支，第二个作用是放弃修改。

#### **git checkout <分枝名>:** 切换分枝

1. 分支名是本地已经存在的分枝,则切换到该分枝
2. 分支名本地不存在,但是它是远端的分支名,则git会将远端的该分枝checkout到本地

#### **git checkout -b <name>:** 创建并切换分枝

如: git checkout -b test2. 创建test2分枝并立即切换

#### **git checkout . :** 放弃工作区中全部修改

Text

Description automatically generated

#### **git checkout -- filename :** 放弃指定文件的修改

### git branch -m <当前分支名> <新的分支名> 重命名分枝

注意: 新旧分枝名之间没有冒号‘:‘

### git branch -d <name> 删除已经合并了的分枝

如果该分支有未合并的修改，Git会阻止你删除它

### git branch -D <name> 强制删除未合并的分枝

当然也可以删除已经合并了分枝.

### git branch -vv : 查看本地发分枝和远程分枝映射关系

### git branch -v :列出每个分枝的最后一次提交信息

### git branch –merged : 查看哪些分枝合并到了当前分枝

### git branch –no-merged : 查看所有未合并的工作分枝

### git merge <branchName>: 将名为name的分枝与当前分枝合并

如:

A picture containing text

Description automatically generated

### git merge –abort :取消merge

如果merge时出现冲突, 则可以用这个取消merge

### git merge <commit> : 将指定的提交合并到当前活动分枝

## 本地操作远端

此过程是 Repository--> Remote 的过程, 所有首先需要将文件提交到本地仓库.

Diagram

Description automatically generated

### git remote -v : 查看远端仓库地址

如果本地仓库和远端已经关联则显示远端地址, 否则报错. 需要进行关联

### git push xxx : 将本地仓库推送到远端

**origin <本地分支名>:<远端分支名> :** 这是完整的写法

**origin <本地分枝名> :** 将当前分枝推送到同名的远程分枝

如果省略远程分支名，则表示将本地分支推送与之存在"追踪关系"的远程分支（通常两者同名），如果该远程分支不存在，则会被新建。

**origin :** 如果当前分支与远程分支之间存在追踪关系，则本地分支和远程分支都可以省略。

**git push :** 如果当前分支只有一个追踪分支或已经设置默认追踪关系，那么主机名都可以省略。

### git push -u origin master当前分枝有多个追踪分枝,指定默认分枝

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。

或者,本地分枝名称改了, 将新名称的分枝与远端分枝进行关联.

### git push origin :heads/<远端分支名>: 删除远程分枝

如下: 同git push origin :< 远端分支名>

Text

Description automatically generated

### git push origin :< 远端分支名> 删除远程分枝

如下: 同git push origin :heads/<远端分支名>

Text

Description automatically generated

如果省略本地分支名，则表示删除指定的远程分支，因为这等同于推送一个空的本地分支到远程分支。

### git push origin : 删除远程分枝

实际使用默认的分枝追踪, 推送了一个空的分枝到远端

### git push origin --delete <远程分支名> : 删除远程分枝

### pull

**git pull作用：**

用于从远程获取代码并合并本地的版本。git pull 的命令格式是：

git pull <远程仓库名> <远程分支名>:<本地分支名>

例如:

git pull origin master:master

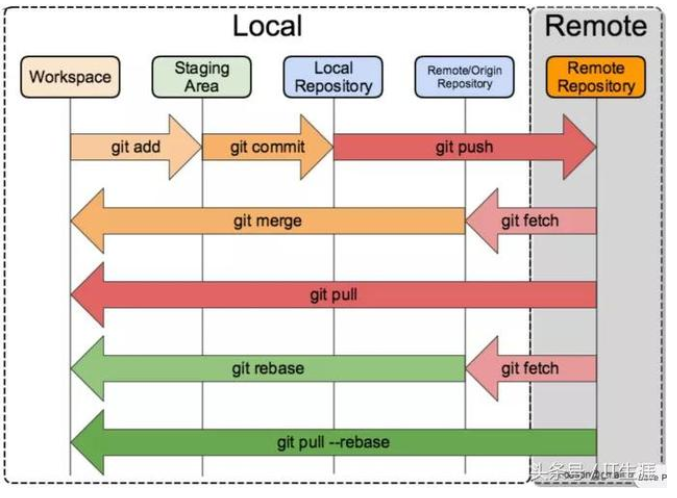
还可以:

git pull origin master 远端分枝和本地同名

git pull origin 远端分枝和本地当前分枝同名

git pull 远端分枝和本地当前分枝同名, 且是默认远端仓库

git pull图示理解:



由上图可知:

1. git pull = git fetch + git merge 相当于从远程仓库获取最新版本，然后再与本地分支merge合并;

2. 同时它还可以进行 rebase 操作，即使用：git pull –rebase

### git pull xxx : 将远端仓库更新到本地

**<远程主机名> <远程分支名>:<本地分支名> :**

这是完整用法, 如: git pull origin master\_remote:master\_local

**git pull <远程主机名>:**

前提条件是: 远端分枝和本地分枝已经存在映射关系.

如: git pull origin

**git pull :** 如果本地仓库只和一个远端仓库映射, 或者已经设定默认追踪关系,后面的都可以省略.

### git remote add <远端仓库名称> <远端仓库地址> 关联本地和远端仓库

#### 远端仓库名称 可以自定义,类似变量名,

如: git remote abc [git@github.com:sunocean01/test.git](mailto:git@github.com:sunocean01/test.git)

#### 可以1对多: 本地仓库可以关联多个远端仓库,

如:

osun@CN-SHA-L-100035 MINGW64 ~/github/gittest (master)

$ git remote -v

origin git@github.com:sunocean01/test.git (fetch)

origin git@github.com:sunocean01/test.git (push)

sv https://gitlab.sensirion.lokal/rd-esp/rd-esp-all/process-release/pmmodule/swverification.git (fetch)

sv https://gitlab.sensirion.lokal/rd-esp/rd-esp-all/process-release/pmmodule/swverification.git (push)

### git remote rename <当前远端名称> <新的远端名称> 改远端名称

如: git remote rename origion origin

### git remote set-url <远端名称> <新的远端url> 修改远端url

如当前远端名称和url的对应关系是origion: url1, 现在想把url1用url2替换, 然后就可以用: git remote set-url origin url2

### git remote remove :删除一个远端名称

### git remote show :打印远端名称清单

### git remote prune <远端名称>:同步远端分枝

Text

Description automatically generated

远端的sha和tt分枝已经被删除了, 但是本地信息没有更新, 可以用**git remote show origin**进行查看,如:

Text

Description automatically generated

此时就可以按照提示用**git remote prune**进行同步.

[(154条消息) git remote prune origin\_BryantLmm的博客-CSDN博客\_git prune remote](https://blog.csdn.net/BryantLmm/article/details/85130091?ops_request_misc=%257B%2522request%255Fid%2522%253A%2522167559506916800188597369%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=167559506916800188597369&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduend~default-1-85130091-null-null.142%5ev73%5ewechat,201%5ev4%5eadd_ask,239%5ev1%5econtrol&utm_term=git%20remote%20prune&spm=1018.2226.3001.4187)

### git remote update origin –prune :更新本地的远程分枝列表

## git reset: 版本回退

### git reset的作用

有时候，我们commit提交代码后，发现这一次commit的内容是有错误的，面对这种情况有两种解决方法：

**解决方法1：**修改错误内容，再次commit一次

**解决方法2：**使用git reset 命令撤销这一次错误的commit

第一种方法比较直接，但会多次一次commit记录，建议使用第二种方法，因为错误的commit没必要保留下来。

### git reset的基本用法

语法: git reset [--soft | --mixed | --hard] [HEAD/commitId]

说明:

Timeline

Description automatically generated

Diagram

Description automatically generated

**--mixed :** 保留工作目录, 并清空暂存区中未提交的修改

为默认, 它的执行结果是: 保留工作目录, 并且清空暂存区未提交的修改,也就是说,工作目录的修改,暂存区未提交的内容以及由reset所导致的新的文件差异,都会被方进工作目录. 简而言之,就是把所有差异都混合(mixed)放在工作目录中.

**-- soft : 将上一次提交的文件退到暂存区**

reset –soft会在重置版本库的时候,保留工作目录和暂存区中的内容, 并把重置版本库带来的差异放进暂存区. 也就是说,工作目录的内容不受影响, 而暂存区中已经commit的内容编程add的状态.

**-- hard: 最危险的操作, 容易造成丢失**

reset –hard会在重置版本的同时, 重置暂存区和工作区里的内容. 也就是说,就是在暂存区中你没有commit的修改也会被全部抹掉,在工作区中你没有add的修改也被全部擦掉.

**HEAD:**

Head : 表示当前版本，就是最新提交的版本，即使用git log打印出来的位于第一位的版本

head^: 上一个版本,如: git reset –hard head^

Head^^: 上上一个版本, 如: git reset –hard head^^

Head~100: 向上100个版本, 如: git reset –hard head~100

<commit id>: 指定版本号,比如commit id 是cadab353589f3eef075817b890dafe8b722d802b, 可以只提供前几位比如: **git reset --hard cadab353589f**, git 会自动查找.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **git reset产生影响** | | | | **表现** |
| 选项 | HEAD | 索引（暂存区） | 工作目录 | 原有文件内容的变更 比如,现在有版本: A,B,C,D, 现在要回退到版本B: 回到的状态可以分两步来理解: 1. B版本已经被成功commit到本地仓库; 2. 对于C,D所更改的内容处理方式不用,如下: |
| --soft | 是 | 否 | 否 | C,D 更改的内容被放到了暂存区 |
| --mixed | 是 | 是 | 否 | C,D 更改的内容被放到了工作区 |
| --hard | 是 | 是 | 是 | C,D 更改的内容全部被删除 |

**注：**版本回退以后，使用git log打印的历史记录都是回退版本之前的数据，之后的都没有了，不过放心，git总有后悔药可以吃哒~

1. 如果命令行窗口没有关闭:

直接去前面找commit id即可；

1. 如果命令行窗口关闭了，或者第二天后悔了:

可以进入到该目录下，使用git reflog命令来查看以前的每一次命令，可以获得每次提交的commit id,就可以版本回退了。

$ git reflog 可以查看命令历史，包含提交的commit id

回退原理:

简单讲，就是说只要进行了代码提交，git内部都会按照时间节点进行记录，每条记录都有commit id作为唯一标识(就像是链表每个节点都有唯一的地址一样)，HEAD总是指向当前版本(就像指针一样)。所谓的版本回退，仅仅是讲Head从当前版本指向了指定的版本，然后将工作区的文件也修改了。

## git revert <commitId> : 撤销指定的提交

Diagram

Description automatically generated

## git stash:

有时你想切换分支，但你正在处理当前项目的一个未完成部分。你不想对半成品进行提交。Git 缓存允许你这样做。git stash 命令使你能够在不提交当前分支的情况下切换分支。

一般来说，stash的意思是 “把东西安全地存放在一个隐蔽的地方”。在Git中，储藏室的意义也是如此；Git在不提交的情况下暂时安全地保存了你的数据。

Diagram, timeline

Description automatically generated

### git stash: 暂存工作

git stash

### git stash save “not finished…“ : 带备注信息的储藏

### git stash list: 查看储藏室清单

### git stash apply: 重新启用最后一次的储藏

### git stash apply <stashId> : 启用指定的储藏版本

### git stash show : 查看文件的变化

指的是: 被藏匿的文件当时是做了哪些修改.

### git stash show -p: 查看文件具体更改的内容

这个也是在git stash show的基础上查看更具体的内容.

### git stash pop : 重新启用之前的储藏

git stash pop 命令与 git stash apply 很相似。这两个命令的主要区别是 stash pop 命令在应用后从堆栈中删除藏书。

### git stash drop : 删除最近一个储藏库

### git stash drop <stashId> : 删除指定储藏库

### git stash clear: 清空储藏室

### git stash branch <branchName> :

如果你把一些工作藏在某一分支上，并继续在该分支上工作。那么，在合并过程中可能会产生冲突。所以，把工作藏在一个单独的分支上是好的。

git stash branch 命令允许用户将工作藏在一个单独的分支上，以避免冲突。这个分支的语法如下。

## 标签

简单的理解，tag 就是 对某次 commit 的一个标识，相当于起了一个别名。

例如，在项目发布某个版本的时候，针对最后一次commit 起一个 v1.0.100 这样的标签来标识里程碑的意义。

有两种类型的标签 ： 轻量标签（lightweight）、附注标签(annotated)

**【轻量标签 】：** 只是某个commit 的引用，可以理解为是一个commit的别名；

**【附注标签】 ：**是存储在git仓库中的一个完整对象，包含打标签者的名字、电子邮件地址、日期时间 以及其他的标签信息。

它是可以被校验的，可以使用 GNU Privacy Guard (GPG) 签名并验证。

### git tag : 列出所有的tag

### git tag -l <xx\*> : 使用通配符列出所有匹配的标签

比如: git tag -l V1\*

### git tag -l -n : 查看所有标签及备注信息

### git ls-remote –tags origin : 查看远程分枝标签

### 标签的种类及添加方法:

#### 轻量标签的创建

1. git tag <标签名>:
2. git tag <标签名> <commit Id>: 给指定的Commit Id添加标签

#### 附注标签的创建

1. git tag -a <标签名> -m <附注信息>: 直接给当前的commit Id添加附注标签
2. git tag -a <标签名> <commit Id> -m <附注信息>: 给指定的commit Id添加附注标签

说明:

-a : annotated, 表示附注标签

-m : 指附注信息

### git tag -d <标签名称>: 删除本地标签

如: git tag -d V1.0

### git push origin <标签名> : 将本地标签推送到远端

如: git push origin V1.0

### git push origin --tags : 将所有本地标签推送到远端

### git push origin --delete <标签名> : 删除远程仓库标签

### git push origin :refs/tags/<标签名> : 删除远程标签

如: git tag push origin :refs/tags/V1.0 : 删除远程V1.0标签

# 远端仓库操作

## 创建远程仓库

### Sensirion gitlab:

Graphical user interface, application

Description automatically generated

### Github:

Graphical user interface, text, application

Description automatically generated

# .gitignore介绍

在工程中，并不是所有文件都需要保存到版本库中的，例如“target”目录及目录下的文件就可以忽略。在Git工作区的根目录下创建一个特殊的.gitignore文件，然后把要忽略的文件名填进去，Git就会自动忽略这些文件或目录。

## .gitignore文件的创建

在git bash中输入命令: touch .gitignore即可创建 .gitignore文件, 然后就可以把需要忽略的文件放在里面, 然后把 .gitignore文件提交就可以.

## .gitignore文件的编辑

### 用记事本直接打开编辑

### 用vim进行编辑

#### vim 有两种工作模式：

1.命令模式：接受、执行 vim操作命令的模式，打开文件后的默认模式；

2.编辑模式：对打开的文件内容进行 增、删、改 操作的模式；

3.在编辑模式下按下ESC键，回退到命令模式；在命令模式下按i，进入编辑模式

#### 创建、打开文件(在git bash中)：

1.输入 touch 文件名 ，可创建文件。

如: touch .gitignore 就创建了 .gitignore文件

2.使用 vim 加文件路径（或文件名）的模式打开文件，如果文件存在则打开现有文件，如果文件不存在则新建文件。

如: vim .gitignore 就打开了 .gitignore文件,默认进入命令模式

3.键盘输入字母i进入插入编辑模式。

敲i后是看不到i的,直接回车就行,然后就可以编辑了.

#### 保存文件：

ZZ或 :wq : 保存并退出vim

ESC退到命令模式, 输入ZZ或 :wq保存并退出vim.

:w : 保存但不退出vim

ESC退到命令模式, 输入 :w保存但不退出vim.

#### 放弃所有文件修改：

**:q!** : 放弃所有文件修改：

按下ESC键进入命令模式，键入"**:q!**"回车后放弃修改并退出vim。

**:e!** :放弃所有文件修改，但不退出vi:

即回退到文件打开后最后一次保存操作的状态，继续进行文件操作：按下ESC键进入命令模式，键入"**:e!**"，回车后回到命令模式。

#### 查看文件内容：

在git窗口，输入命令：cat 文件名

#### 创建文件夹/文件

在git窗口，输入命令：touch 文件夹名/文件名

## 语法和示例:

|  |  |  |
| --- | --- | --- |
| 符号 | 描述 | 示例 |
|  | 忽略指定的文件,把整个文件名包括后置都放进来. | tmpfile.txt |
| # | 以#开头表示注释 |  |
| / | 表示根目录 | /bin忽略根目录下的bin文件或文件夹  bin/ 忽略bin文件夹下的所有内容 |
| \* | 匹配零个或多个字符 | \*.pdf匹配 .pdf结尾的文件 |
| \*\* | 匹配多级目录,可在开始,中间,结束 | dc/\*\*/\*checkpoint.ipynb dc匹配文件夹下任意级别目录下的 \*checkpoint.ipynb文件 |
| ! | 否定 | !\*.a匹配不是以 .a结尾的文件 |
| ? | 匹配单个字符 |  |
| [] | 匹配单个字符列表 | [Dd]escription.txt大写或小写开头的 |

## 忽略文件原则

* 忽略**操作系统自动生成的文件**，比如缩略图等；
* 忽略**编译生成的中间文件、可执行文件**等，也就是如果一个文件是通过另一个文件自动生成的，那自动生成的文件就没必要放进版本库，比如Java编译产生的.class文件；
* 忽略**你自己的带有敏感信息的配置文件**，比如存放口令的配置文件。

## 定义全局 .gitignore

除了可以在项目中定义.gitignore文件外，还可以设置全局的.gitignore文件来管理所有Git项目的行为。

这种方式在不同的项目开发者之间是不共享的，是属于项目之上Git应用级别的行为。

可以在任意目录下创建相应的.gitignore文件，然后再使用以下命令配置Git

git config --global core.excludesfile ~/.gitignore

## 常见问题

1. 无法将文件添加到git,原因是这个问被 .gitignore忽略了.

# SSH Key的配置

## SSH介绍:

1. SSH 是一种安全协议; git里支持四种安全协议: SSH, https,git,ftp
2. 作用: 连接客户端和服务端;

## SSH Key 的创建:

1. cd ~: 回家, 这样创建的密钥就会在 ./ssh 下面,windows里地址如下:

Graphical user interface, text, application, email

Description automatically generated

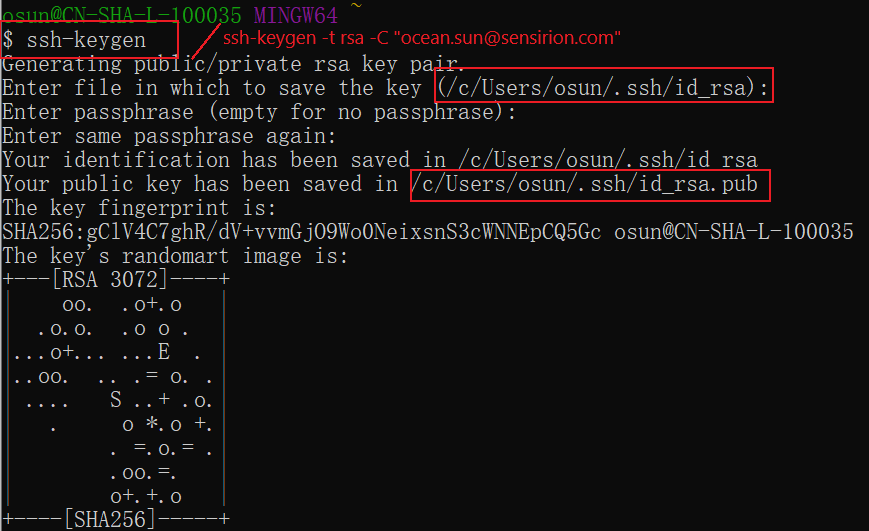
1. 生成公玥和私玥:

有如下几种方式(貌似在任何位置都可以,但要用git bash 打开命令行):

* 1. ssh-keygen
  2. ssh-keygen -t rsa -C "youremail@example.com"

Note: 默认是在 \*\* .ssh/id\_rsa & id\_rsa.pub (私玥放自己电脑,公玥放服务器上)

**这三个直接回车就行**



1. SSH agent是否正常运行检查: 如果返回PID 表示正常运行

Text

Description automatically generated

1. 复制公玥, 有两种方法:
   1. 在命令行中打开公玥:
      1. cd .ssh : 先进入 .ssh 文件,及公玥私玥所在的位置;
      2. cat id\_rsa.pub : 打开公玥,选中复制即可.

示例:

Text

Description automatically generated

* 1. 找到公玥所在的位置,用记事本打开,然后复制:

Graphical user interface, text

Description automatically generated

1. 黏贴到gitlab里的SSH Keys (github和gitlab 稍微有点区别):

**Sensirion gitlab:**

Graphical user interface, text, application, email

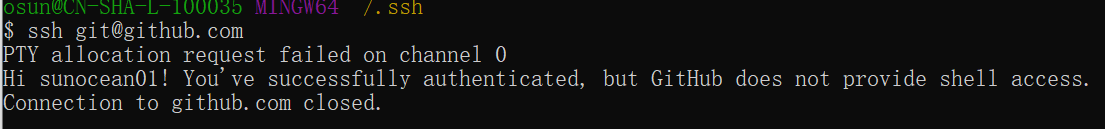
Description automatically generated

**GitHub public:**

Graphical user interface, text, application

Description automatically generated

1. 连接github/gitlab:
   1. ssh [git@github.com](mailto:git@github.com)



# Project VS repository

## 理解:

**Repositories:** 直接翻译过来即是资料库的意思，它可以保存代码工程、项目代码、图片、文本等，平时做的笔记也是可以放入的

**Projects（project boards）**:指的是管理项目板的意思，它包括议题、拉取请求和注释，在选择的列中分类为卡片。 您可以拖放或使用键盘快捷键对列中的卡片重新排序，在不同列之间移动卡片，以及更改列的顺序。

## 官方说明：

**Repositories：**一个远程存放源代码的仓库（可以理解成一个文件夹），通常用来存放项目中的所有文件，并记录着每个文件的修订记录。你也可以与他人共享仓库的所有权。

**Projects：**项目板可帮助您组织工作并确定工作的优先级。您可以为特定的功能工作，全面的路线图甚至发布清单创建项目板。使用项目板，您可以灵活地创建适合您需求的定制工作流。

## 关于使用：

通常repository我们用来保存代码，而project用来进行项目管理，可以添加任务，制作工作流程等操作。

# 参考链接:

1. [Git - Reference (git-scm.com)](https://git-scm.com/docs)
2. [Git 入门介绍 - MyBatis中文官网](http://www.mybatis.cn/git/index.html)
3. [Home · geeeeeeeeek/git-recipes Wiki (github.com)](https://github.com/geeeeeeeeek/git-recipes/wiki)
4. [Git – Ignore命令|极客教程 (geek-docs.com)](https://geek-docs.com/git/git-cmds/git-ignore.html)