目录

[Git 入门教程 1](#_Toc23116)

[一. Git是目前世界上最先进的分布式版本控制系统（没有之一） 1](#_Toc31191)

[1． 集中式版本控制系统 1](#_Toc417)

[2． 分布式版本控制系统 2](#_Toc14322)

[二. git的下载和配置 2](#_Toc26919)

[1． windows系统的git安装 2](#_Toc24962)

[2． Git的配置： 3](#_Toc31539)

[三. 创建git本地仓库和远程仓库 3](#_Toc29128)

[1． 常用的linux命令 3](#_Toc7997)

[2． 创建本地仓库并初始化 4](#_Toc3762)

[3． 自行注册自己的github账号 5](#_Toc5766)

[4． 创建自己的秘钥，有的话（在自己的用户主目录下看是否有.ssh文件夹）直接跳过，进入第五步 5](#_Toc860)

[5． 创建远端github仓库 7](#_Toc23321)

[6． 将自己的本地仓库的远端github仓库关联起来，并将本地仓库的内容推送到远端github仓库里面 8](#_Toc14235)

[四. 图解Git的核心概念和跑自己工作项目 9](#_Toc31639)

[1． Git运行的核心图解和一些概念解释 9](#_Toc21769)

[2. 刚接手公司项目时，git 开发流程。注意： 为了方便操作，红字部分为具体执行的代码 9](#_Toc693)

[五. 常用的git指令 12](#_Toc3357)

[1． 提示性指令 12](#_Toc12115)

[2． 撤销操作 12](#_Toc20821)

[3． 删除文件 13](#_Toc10895)

[4． 分支操作 13](#_Toc5515)

[5． git stash 缓存区的使用 14](#_Toc4942)

[6． git merge 和git rebase： 两者都是合并冲突 16](#_Toc13780)

[7 冲突的具现形式 17](#_Toc13056)

[六. 常见bug,欢迎大家随时补充 18](#_Toc9392)

[1 . 误触或者手动进入vim 编辑器如何退出 18](#_Toc3511)

**Git 入门教程**

1. Git是目前世界上最先进的分布式版本控制系统（没有之一）

首先，明确两个概念： 分布式版本控制系统和集中式版本控制系统

* 1. 集中式版本控制系统

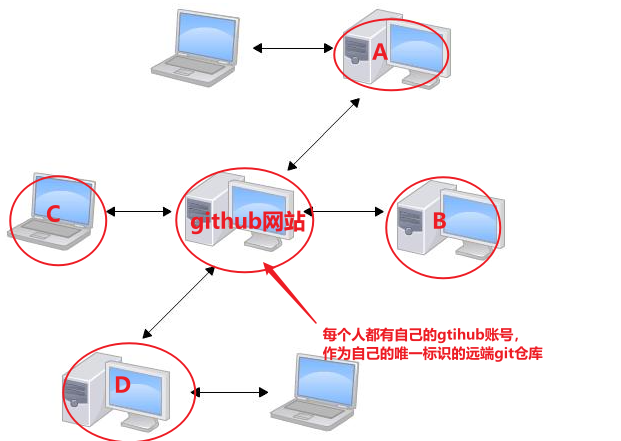
只有一中央服务器，每次干活的开始和结束都要访问中央服务器，而且需要联网，存在两个缺陷，第一个安全性问题，如果中央服务器出问题，存放在中央服务器的资料可能会别销毁或者篡改，第二个是网速慢的问题，因为每次提交都要联网，所以说当网速慢的时候，提交100M的文件可能需要5分钟，这样的话效率会降低



* 1. 分布式版本控制系统

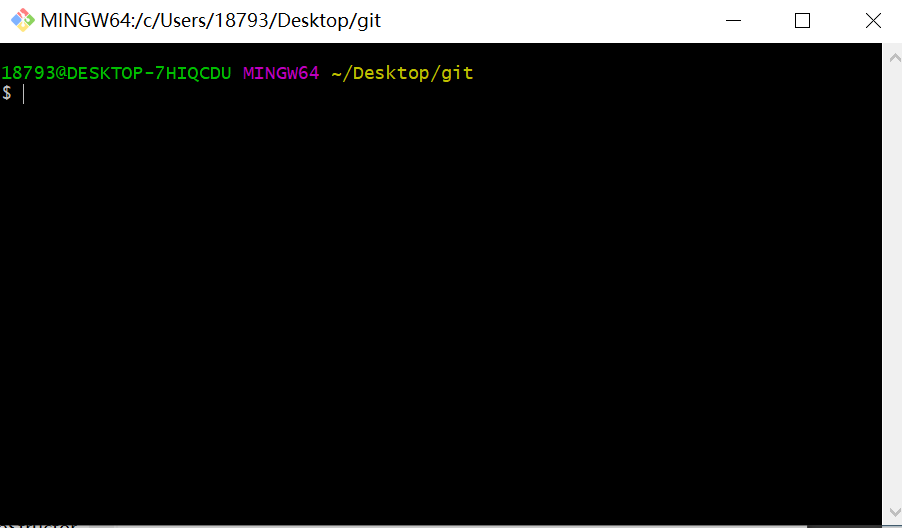
分布式版本控制系统没有中央服务器，通过与github网站的远程连接，每个人的电脑都可以创建一个自己本地的服务器，相比于集中式版本控制系统，分布式有两个更优化的点，第一是不需要联网，在本地仓库开发完毕之后可以直接推到和本地关联的远端github仓库里面，这样就完成了一次存档。第二点是安全性问题，当你本地仓库的代码出现问题，别慌，只需要执行 git fetch语句从远端github仓库重新拷贝一份就行。

Git分布式版本控制系统为什么是最有竞争力的呢，是因为他存档的是并不是文件本身，而是你对文件的修改，并生成一条条的历史记录，我们可以通过不断地在这些历史记录之间跳转，就能回到我们之前或者之后的任意一个时间点的开发状态



1. git的下载和配置
   1. windows系统的git安装

在Windows上使用Git，可以从Git官网直接[下载安装程序](https://git-scm.com/downloads" \t "https://www.liaoxuefeng.com/wiki/896043488029600/_blank)，然后按默认选项安装即可，安装完成后，在开始菜单里找到“Git”->“Git Bash”，蹦出一个类似命令行窗口的东西。就说明Git安装成功！



* 1. Git的配置：

因为git是分布式版本控制系统，每台安装了git的电脑都是一台服务器，所以我们必须要为我们的服务器设置唯一标志，也就是我们的名字和Email，配置如下：

$ git config --global user.name "Your Name"

$ git config --global user.email ["email@example.com"](mailto:\"email@example.com\")

我们可以用以下语句来查看自己的配置

$ git config -l

注意： git config 的 --global 参数： 用了这个参数，表示我们这台电脑上的所有git仓库都会用这个配置，当然也可以对某个仓库指定不同的用户名和Email地址。

1. 创建git本地仓库和远程仓库
   1. 常用的linux命令

首先，你要熟悉几个Linux指令：

mkdir: 创建目录

touch: 创建文件

cd: 实现目录之间的跳转，

ls: 查看当前目录下的文件和文件夹

pwd: 显示当前的文件目录，

cat <filename>： 获取当前文件的镜像

下面来看一个完整的例子：

$ mkdir learngit

$ cd learngit

$ pwd

/Users/michael/learngit

* 1. 创建本地仓库并初始化

接下来，我们进入正题，我们来创建本地仓库

$ mkdir learngit

$ cd learngit

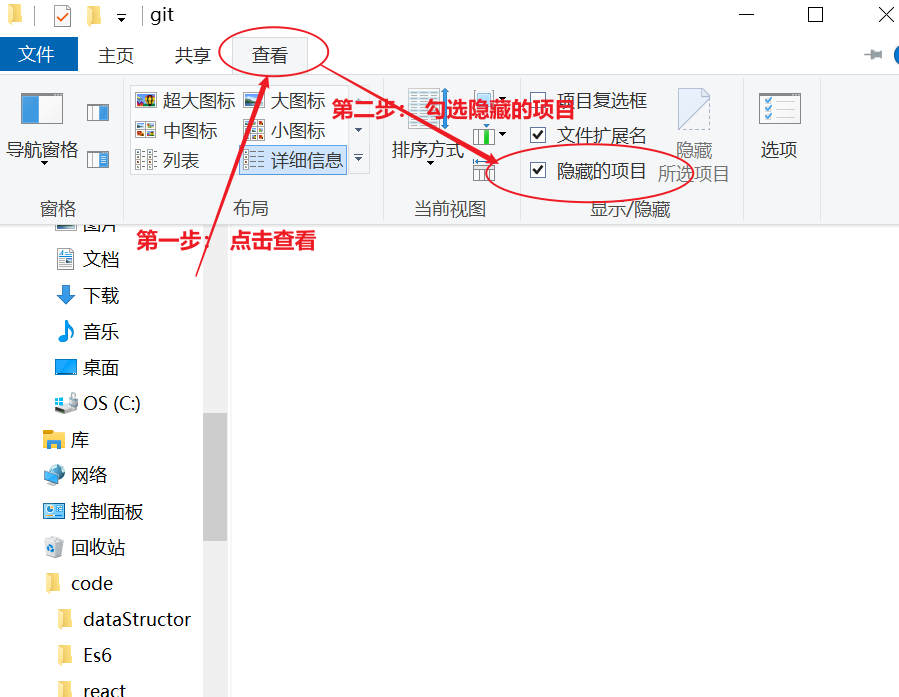
我们可以通过在git Bash编辑器里面输入上述指令 在你右键Git Bah的地方创建learngit文件夹，然后进入该文件夹，或者说我们可以手动的创建文件夹learngit，并进入，

然后，在git Bash 里面输入以下指令来将我们的这个文件夹设置为我们的本地仓库，以便于和远程git仓库来做一个关联

$ git init

Initialized empty Git repository **in** /Users/18793/learngit/.git/

执行完上述语句， 我们的文件夹下面会出现一个 .git 文件夹，没有出现的话，不要着急，他只是隐藏起来了，来到我们的learngit文件夹下，执行下面的操作，你就可以看到他了，或者说在在git Bash里面我们也可以执行 ls -ah 来查看它



* 1. 自行注册自己的github账号
  2. 创建自己的秘钥，有的话（在自己的用户主目录下看是否有.ssh文件夹）直接跳过，进入第五步

首先，由于我们的本地仓库和GitHub仓库之间的传输是通过SSH加密的，所以，我们需要先来配置自己的秘钥

第1步：创建SSH Key。在用户主目录（一般是C:\Users\18793(这是我的用户名，你到时候要按照自己的去找)）下，看看有没有.ssh目录，如果有，再看看这个目录下有没有id\_rsa和id\_rsa.pub这两个文件，如果已经有了，可直接跳到下一步。如果没有，打开Git Bash，创建SSH Key：

$ ssh-keygen -t rsa -C "youremail@example.com"

你需要把邮件地址换成你自己的邮件地址，然后一路回车，使用默认值即可，由于这个Key也不是用于军事目的，所以也无需设置密码。

如果一切顺利的话，可以在用户主目录里找到.ssh目录，里面有id\_rsa和id\_rsa.pub两个文件，这两个就是SSH Key的秘钥对，id\_rsa是私钥，不能泄露出去，id\_rsa.pub是公钥，可以放心地告诉任何人。

接下来： 我们可以在.ssh文件目录下git Bash 执行以下语句，来获取我们公钥的镜像文件

$ cat id\_rsa.pub



第2步：登陆GitHub，打开“Account settings”，“SSH and GPG Keys”页面：

然后，点“Add SSH Key”，填上任意Title，在Key文本框里粘贴id\_rsa.pub文件的内容：



点“Add Key”，你就应该看到已经添加的Key：



这样，我们的ssh key（秘钥）就已经配置完毕了，

* 1. 创建远端github仓库

现在，我们开始创建自己的远程github库

首先，登陆GitHub，然后，在右上角找到“Create a new repository”按钮，创建一个新的仓库：



在Repository name填入learngit，其他保持默认设置，点击“Create repository”按钮，就成功地创建了一个新的Git仓库：



目前，我们的github上的这个learngit仓库还是空的，GitHub告诉我们，可以从这个仓库克隆出新的仓库，也可以把一个已有的本地仓库与之关联，然后，把本地仓库的内容推送到GitHub仓库

* 1. 将自己的本地仓库的远端github仓库关联起来，并将本地仓库的内容推送到远端github仓库里面

现在，我们来让我们的远程github仓库和本地的git仓库关联起来

git remote add origin git@github.com:baifengbxl/learngit.git

注意： 这里的baifengbxl一定要配置成你自己的账号名，否则，你在本地关联的就是我的远程库，关联没有问题，但是你以后推送是推不上去的，因为你的SSH Key公钥不在我的账户列表中。

添加后，远程库的名字就是origin，这是Git默认的叫法，也可以改成别的，但是origin这个名字一看就知道是远程库。

接下来，我们把我们本地仓库的内容全部推送到远端的github仓库里面

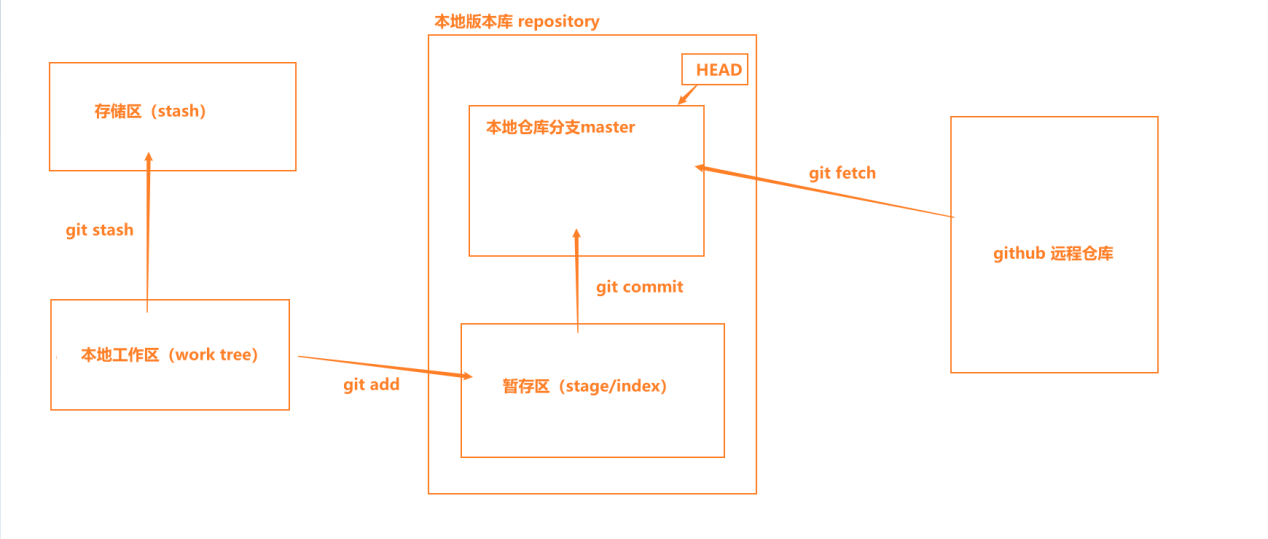
$ git push -u origin master

由于远程库是空的，我们第一次推送master分支时，加上了-u参数，Git不但会把本地的master分支内容推送的远程新的master分支，还会把本地的master分支和远程的master分支关联起来，在以后的推送不需要去加 -u

现在我们的初始化配置已经完成，下面我们实现本地仓库的远端github仓库的交互，包括下拉fetch和推送push代码，对本地仓库的一些修改等等

1. 图解Git的核心概念和跑自己工作项目
   1. Git运行的核心图解和一些概念解释

首先，我们来了解git运行的核心和一些概念



工作区： 自己编辑自己代码的地方，这些代码存放在本地仓库里面

缓存区(stage/index)：通过git add 将将文件添加到了缓存区里，代表我们将当前的文件加入到了git版本控制里面，这样我们的代码就可以提交到本地分支master了

本地分支master：和远程github仓库分支对应的本地分支

HEAD： 指向当前master的最新一次提交，他的本质是一个指针，我们的可以通过修改HEAD来操作我们在本地分支master上面的commit记录，比如，回到之前的某个提交版本，或者前往最新的一次提交版本，eg: git reset --hard HEAD^ 代表回到上一次提交

存储区（stash）: 对工作区的文件做一层备份，以应对突发情况，比如你开发到一半，老板让你改bug，你可以现将你正在开发的代码git stash放到存储区，改完bug之后再回到当前分支通过git stash pop 将你之前存储的开发代码拿出继续开发

远程github仓库：我们在github上面创建的仓库

1. 刚接手公司项目时，git 开发流程。注意： 为了方便操作，红字部分为具体执行的代码

1 创建自己的本地工作目录，可手动，可git Bash ，下面我们将全程用 git Bash 操作

mkdir work

cd work

mkdir webfront

cd webfront

pwd

/c/Users/18793/Desktop/workgit/webfront // 自己当前的文件目录

2 配置自己的git 环境，作为你在这台电脑上访问github上面公司项目仓库的的唯一标识

git config --global user.name "xuyan"

git config --global user.email "xuyan@winsense.ai"

git config -l //查看自己已完成的配置

3 查看自己是否拥有秘钥，有的话直接进入第4步

ssh-keygen -t rsa -C “这里换上你的邮箱’’, 一路回车，在出现选择时输入Y，再一路回车直到生成密钥。会在/U用户sers/\*\*\*/（用户主目录）路径下生成一个.ssh文件夹，密钥就存储在其中。

ssh-keygen -t rsa -C ‘violet@bf.com’,

在自己的用户主目录下面查看自己是否有.ssh文件夹，有的话在此处git Bash，执行下面的命令获取自己的公钥

Cat ~/.ssh/id\_rsa.pub

拿到自己的公钥，登录自己的的github，进入setting => SSH and GPGS keys 选项填写自己的公钥， 这样，我们的秘钥就配置完了

4 拉取项目代码

Git clone ‘项目代码的地址，直接去项目仓库里面复制就行，下面是一个例子’ :

git clone [git@github.com:baifengbxl/gitskillls.git](mailto:git@github.com:baifengbxl/gitskillls.git)

5 进入项目文件目录，切换并创建自己本地的开发分支

cd jc-car-pcweb/

创建本地的开发分支，进入该分支， 并且和远端的github仓库的dev分支对齐

git checkout -b dev origin/dev

查看当前的开发分支

gitk (git 的 gui 工具，可以看一下现在分支的情况)

6 在本地编辑器里面打开项目，打开编辑器终端（我用的是vscode），安装依赖

npm install

安装完成之后会出现一个node module 目录，代表我们的依赖安装成功

7 按照项目封装的jenkins自动化部署代码跑项目，执行

./project.sh dev // 注意： 我这里只是一个例子，具体按照查看项目的readme.md文件自适配

或者直接在编辑器终端里面执行

npm run dev // 注意：这可能会忽略掉一些项目的配置文件

执行上面完的命令，我们的项目就跑起来了

下面我们接着说一些我们在项目中用到的指令

1 本地代码开发或者修改完毕，提交到本地仓库

git add -- .

git commit -m ‘注释’

2 将最新的github远端仓库里的项目代码同步到本地

git fetch

3 将远端的代码的本地的代码合并

git merge 或者 git rebase origin/dev （更倾向于使用git rebase.,后面会对二者做一个区分）

4 若是在git rebase的时候出现冲突，按照提示解决冲突，然后执行

git add -- .

git rebase --continue

如果有冲突的话，解决冲突之后继续执行步骤4

5 本地的代码冲突完全解决之后，push到远端github仓库里面，注意： 这一步有风险可能因为你的一些bug出现一些问题，恢复的话比较麻烦，可以在push之前先问问师兄或者师姐

git push origin dev

以上便是我们在接手公司项目的时候配置环境变量的基本流程

1. 常用的git指令
   1. 提示性指令

gitk : git的gui可视化工具，查看本地master分支的提交情况

git satus： 查看自己当前的git 状态， 当自己不知道怎么操作时，可以Git status ，他会给我们一些提示信息

git log ： 查看日志（详细）， 可以获取每次提交的的commit id,便于我们对自己的commit历史记录做操作（通过修改HEAD的指向）

git log --graph --pretty=oneline --abbrev-commit // 查看当前的分支合并图

git refslog ： 查看提交日志（简略）

git diff <filename> : 对比自己对某文件的修改

git diff HEAD -- <filename>: 可以查看工作区和本地仓库（版本库）的最新区别

* 1. 撤销操作

1 版本回退，（撤销自己的提交，也就是会到上一次的提交记录）本质是对HEAD指针的操作

git reset --hard HEAD^

git reset --hard HEAD~100, 回到之前100的提交记录

2 丢弃对工作区某文件的修改: 本质是使用最近一次的add 或者 commit 来覆盖本地工作区的代码

git checkout -- <filename> // 注意这里的-有两个，一个的话是切换分支

3 丢弃对stage暂存区的的修改，就是已经 git add 的内容:

git reset HEAD <filename>

这个时候你的stage的内容覆盖了你的工作区，此时，你还要撤销对工作区的修改，来保证你的工作区是干净的

git checkout -- <filename>

* 1. 删除文件

1 删除工作区的文件

rm <filename>

2 删除本地仓库的文件

git rm <filename>

然后提交

git commit -m ‘注释’

3 如果误删本地工作区的文件 ，可以直接从本地仓库里面直接拷一份出来就行了

git checkout -- <filename>

* 1. 分支操作

1 创建分支dev

git branch dev // dev为分支名

2 从当前分支切换到dev分支

git checkout dev

3 一步到位： 创建并切换到dev分支

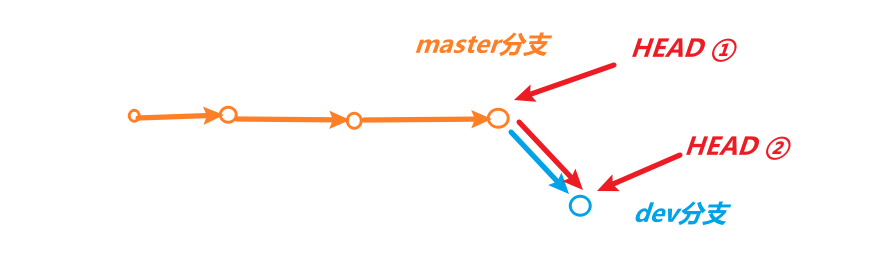
git checkout -b dev // -b表示branch

4 查看分支

git branch // git branch命令会列出所有分支，当前分支前面会标一个\*号。

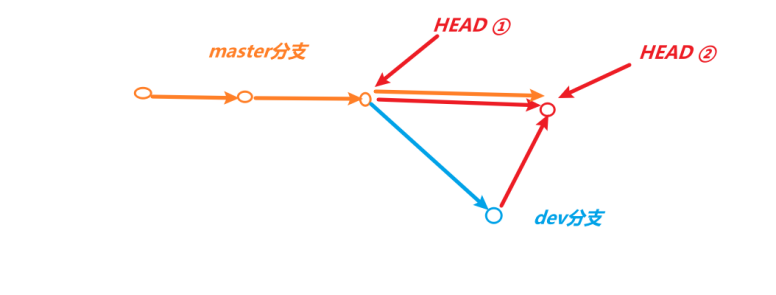
5 分支合并

git merge dev //表示当前分支合并到dev分支，默认会开启 Fast Forward 模式



--no-ff会关闭Fast Foward 模式

git merge --no-ff dev //合并master和dev的最新一次提交，生成新的节点



6 删除分支

git branch -d dev

git branch -D dev //强制删除分支dev

* 1. git stash 缓存区的使用

1 将修改存到stash里面

git stash 等同于 git stash save 这条命令行产生了两个结果： ① 修改存储到stash里面 ② 使本地master分支指向HEAD

注意几个点：

① 新增的文件，直接执行stash是不会被存储的 ，需要用git add -- . 将修改的文件添加到git版本控制里面

② git add 只是把文件加到git 版本控制里，并不等于就被stash起来了，git add和git stash 没有必然的关系，但是执行git stash 能正确存储的前提是文件必须在git 版本控制中才行。

2 将stash里面的内容取出来

git stash pop

或者执行下面的两条命令，效果是一样的

git stash apply //将缓存区的内容取出来，但不会删除缓存区的备份

git stash drop //删除缓存区的备份

3 其他的stash指令

git stash list  ：查看stash了哪些存储

git stash show ：显示做了哪些改动，默认show第一个存储,如果要显示其他存贮，后面加stash@{$num}，比如第二个 git stash show stash@{1}

git stash show -p : 显示第一个存储的改动，如果想显示其他存存储，命令：git stash show  stash@{$num}  -p ，比如第二个：git stash show  stash@{1}  -p

git stash apply :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}，如果要使用其他个，git stash apply stash@{$num} ， 比如第二个：git stash apply stash@{1}

git stash pop ：命令恢复之前缓存的工作目录，将缓存堆栈中的对应stash删除，并将对应修改应用到当前的工作目录下,默认为第一个stash,即stash@{0}，如果要应用并删除其他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@{$num} ：丢弃stash@{$num}存储，从列表中删除这个存储

git stash clear ：删除所有缓存的stash

4 一些小技巧

 常规 git stash 的一个限制是它会一下暂存所有的文件。有时，只备份某些文件更为方便，让另外一些与代码库保持一致。一个非常有用的技巧，用来备份部分文件：

① add 那些你不想备份的文件（例如： git add file1.js, file2.js）

② 调用 git stash –keep-index。只会备份那些没有被add的文件。

③ 调用 git reset 取消已经add的文件的备份，继续自己的工作。

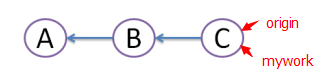
* 1. git merge 和git rebase： 两者都是合并冲突

1 说一下冲突的问题

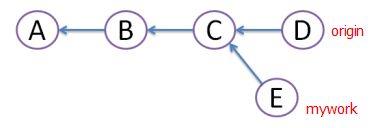
先了解两条常用的指令：

git pull = git fetch + git merge  
git pull --rebase = git fetch + git rebase

假设有3次提交A,B,C。

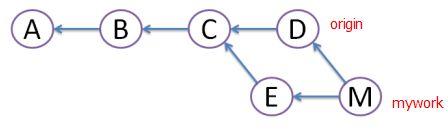


在远程分支origin的基础上创建一个名为"mywork"的分支并提交了，同时有其他人在"origin"上做了一些修改并提交了。

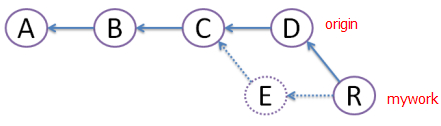


其实这个时候E不应该提交，因为提交后会发生冲突。如何解决这些冲突呢？有以下两种方法：

2 git merge  
用git fetch命令把"origin"分支上的修改fetch下来与本地提交合并（merge）成版本M，但这样会形成图中的菱形，



2 git rebase  
创建一个新的提交R，R的文件内容和上面M的一样，但我们将E提交废除，当它不存在（图中用虚线表示）git rebase的好处是避免了菱形的产生，保持提交曲线为直线，让大家易于理解。



注意两个点：

1. 在rebase的过程中，有时也会有冲突，这时git会停止rebase并让用户去解决冲突，解决完冲突后，用git add命令去更新这些内容，然后不用执行git commit,直接执行git rebase --continue,这样git会继续apply余下的补丁。
2. 在任何时候，都可以用git rebase --abort参数来终止rebase的行动，并且mywork分支会回到rebase开始前的状态。

3 git merge 和git rebase 的工作流的区别

1. git rebase 的工作流

git rebase

while(存在冲突) {

git status

找到当前冲突文件，编辑解决冲突

git add -- .

git rebase --continue

if( git rebase --abort ){

break;

}

}

1. git merge 的工作流

git pull (git fetch + git merge)

如果存在冲突，解决冲突

git pull

7 冲突的具现形式

一般情况下rebase都是会有冲突的，详细查看冲突可以用命令git status然后就会显示哪个文件有冲突，然后打开有冲突的哪个文件，会发现有一些“<<<<<<<”， “=======”， “>>>>>>>” 这样的符号。下面给他们做一些解释

① 基本解释

“<<<<<<<” 表示冲突代码开始

“=======” 表示test与master冲突代码分隔符

“>>>>>>>" 表示冲突代码的结束

②用法

<<<<<<<

所以这一块区域test的代码

=======

这一块区域master的代码

>>>>>>>

当我们手动解决解决完冲突之后接着执行(注意：改完代码之后要把 >>>>> ===== <<<<<< 这些符号删掉，他只是一个标示性作用)

git add -- .

git rebase --continue

就可以接着处理下一条冲突了，处理完之后重复上面的步骤

1. 常见bug,欢迎大家随时补充

1 . 误触或者手动进入vim 编辑器如何退出

（1）进入的方式，误触或者手动进入，手动进入有下面两种方式

第一种， 要在git Bash里面编辑某文件file可以通过 vi <filename> 进入当前文件

第二种， 在git add 的时候可以通过 git add -p 进入补丁模式，对自己的修改做区块(hunk)划分 ,选择提交部分修改

（2）退出的方法

首先说一下vim的三种编辑模式

① 命令模式(command mode): 按Esc 就可以进入命令行模式也是系统默认模式

② 插入模式(insert mode): 输入模式可以按 o i a 中的任意一个都可以进入 退出(按Esc)可以进入末行和命令行模式

③ 末行模式(last line mode): 它的主要功能是退出编辑器 也可以保存退出文档

退出的话，需要进入末行模式，然后输入 ctrl + ; + 下面的三条语句任意一条，按需求自适应

q! 【强制退出不保存】 q【退出不保存】 wq【退出并保存后面也可以加个！】