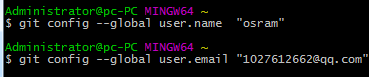
## 基础命令总结

源自：<http://mp.weixin.qq.com/s/43DRPU-O0X3MnGLvj21MbQ>

* 1. Git是分布式版本控制系统，所以需要填写用户名和邮箱作为一个标识。

注意：git config --global 参数，有了这个参数，表示你这台机器上所有的Git仓库都会使用这个配置，当然你也可以对某个仓库指定的不同的用户名和邮箱。



* 1. 创建仓库

pwd //显示当前目录

cd D: //切换到d盘，跟Windows命令行一样

mkdir testgit //在d盘下建文件夹testgit

git init //把当前目录变成仓库，该文件夹里过多一个.git文件夹

* 1. 添加文件到仓库

git add readme.txt //将文件添加到暂存区，无提示代表成功

git commit –m “此处添加注释” //将文件提交到仓库，一定要加注释，否则提交不了

git status //查看是否有文件未提交

* 1. 版本回退

git log //从近到远显示日志

git log –-pretty=oneline //显示一行日志

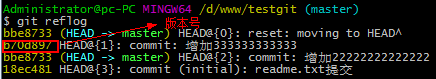
git log -p 显示每次提交的diff

git reset –-hard HEAD^ //回退到上个版本

git reset -–hard HEAD~100 //回退到前100个版本

* 1. 版本恢复

git reflog //查看版本号（不知道版本号时）



git reset –hard b70d897 //恢复到某个版本号

* 1. 查看文件内容

cat readme.txt //查看文本内容

* 1. 修改文件内容

git diff readme.txt //文件修改后，未提交时，可以查看跟修改前的区别

需重复git add readme.txt和git commit –m ‘注释’ 两步操作

-- 两边都要有空格

若想撤销修改，使用：

git checkout -- readme.txt //丢弃工作区的修改

<1>修改后还没有放入暂存区，撤销后就回到跟版本库一模一样的状态

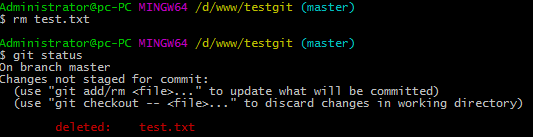
<2>上传到暂存区后再修改，撤销操作之后就回到添加暂存区后的状态

<3>即都只能往前撤一步。

* 1. 删除文件

跟添加文件类似。同样删除文件之后也需要commit

rm test.txt //删除文件



<1>git commit –m ‘注释’ //文件从仓库彻底删除

<2>git checkout -- test.txt //恢复文件

* 1. 不删除本地文件，只删除缓冲区的文件

some-directory 是相对于本地根目录下的文件夹/文件路径

|  |
| --- |
| git rm -r --cached some-directory  git commit -m 'Remove the now ignored directory "some-directory"'  git push origin master |

* 1. 忽略文件

忽略：git update-index --assume-unchanged

恢复：**git update-index --no-assume-unchanged**

## 工作区和暂存区

工作区：就是你在电脑上看到的目录，比如目录下testgit里的文件(.git隐藏目录版本库除外)。或者以后需要再新建的目录文件等等都属于工作区范畴。

版本库(Repository)：工作区有一个隐藏目录.git,这个不属于工作区，这是版本库。其中版本库里面存了很多东西，其中最重要的就是stage(暂存区)，还有Git为我们自动创建了第一个分支master,以及指向master的一个指针HEAD。

　　我们前面说过使用Git提交文件到版本库有两步：

　　第一步：是使用 git add 把文件添加进去，实际上就是把文件添加到暂存区。

第二步：使用git commit提交更改，实际上就是把暂存区的所有内容提交到当前分支上。

## 远程仓库

本地Git仓库和github仓库之间的传输是通过SSH加密的，所以需要一点设置：

**第一步**：创建SSH Key。在用户主目录下，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，如果有的话，直接跳过此如下命令，如果没有的话，打开命令行，输入如下命令：

$ ssh-keygen -t rsa -C “haiyan.xu.vip@gmail.com（你的邮箱）”

按3个回车，密码为空。在C:\Users\Administrator\.ssh目录下生成两个文件：



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。

**第二步：**

登录github,打开” settings”中的SSH Keys页面，然后点击“Add SSH Key”,填上任意title，在Key文本框里黏贴id\_rsa.pub文件的内容。

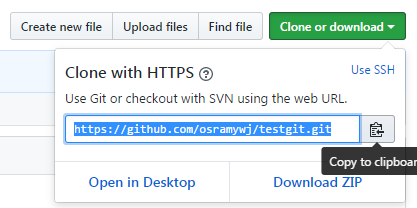
**第三步**：

在github上新建一个仓库，将他与本地仓库建立连接。

在本地的testgit仓库下运行命令：

git remote add origin <https://github.com/osramywj/testgit.git>

git remote remove origin //取消关联

上面的关联地址来自于

推送内容到github：

git push –u origin master

有时会出现下面错误：

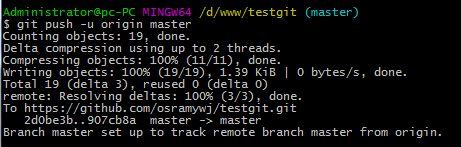


原因：github中的README.md文件不在本地代码目录中，可以通过如下命令进行代码合并【注：pull=fetch+merge】

git pull --rebase origin master

再通过命令git push -u origin master完成代码上传到github的操作。

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，在以后的推送或者拉取时就可以简化命令。



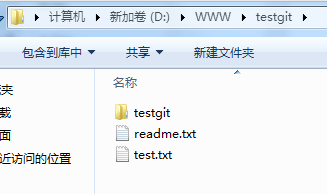
自此，从本地做提交，就可以用命令：

git push origin master

## 从远程库克隆

git clone <https://github.com/osramywj/testgit.git>

就会在本地仓库下生成一个testgit目录了



## 创建与合并分支

创建并切换分支：git checkout –b 本地分支名 origin/远程分支名（可选）

相当于两条命令：git branch 分支名 //创建分支

git checkout 分支名 //切换分支

查看所有分支： git branch //在当前分支前加\*号

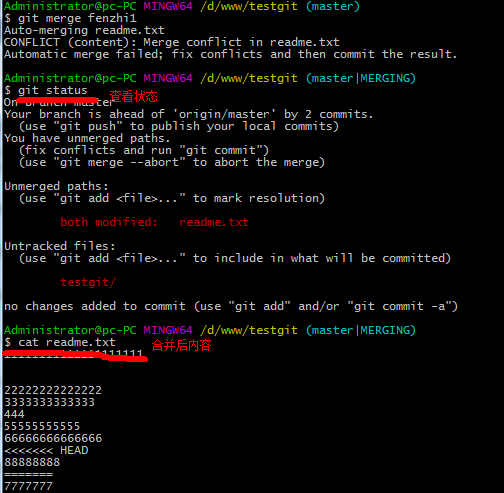
在不同的分支上的操作，相互独立！若要将B分支合并到A上，可以在A分支上使用命令：

git merge B //合并指定分支到当前分支

git merge –-no-ff B 保留合并的分支信息

git branch –d 分支名 //删除分支

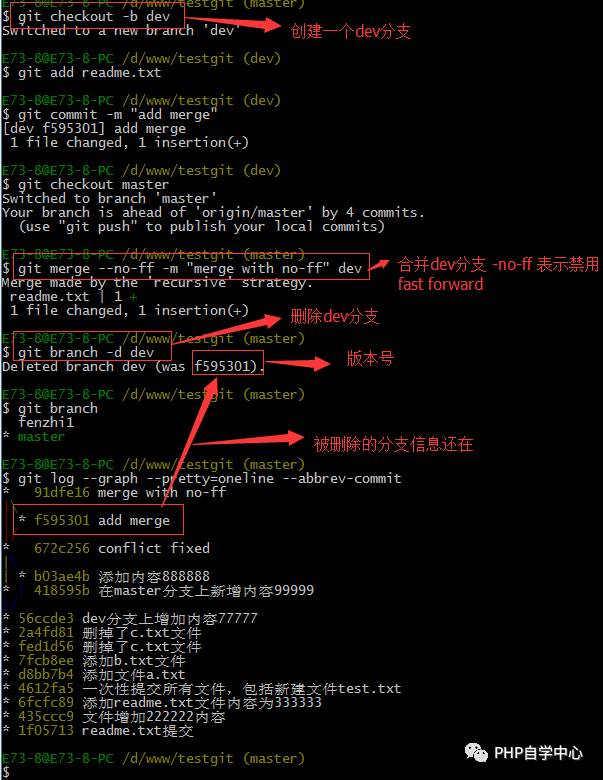
冲突解决：



之后将冲突的代码做修改，再add commit提交即可

合并分支一般使用”Fast forward”模式，删除分支后会丢掉分支信息。采用下面命令可保留分支信息

git merge –-no-ff –m ‘注释’ 分支名 //禁用fast forward

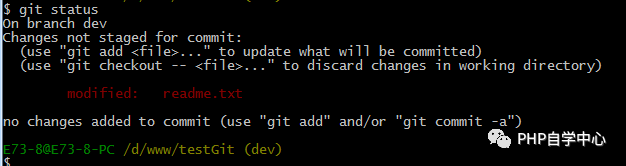


分支策略：首先master主分支应该是非常稳定的，也就是用来发布新版本，一般情况下不允许在上面干活，干活一般情况下在新建的dev分支上干活，干完后，比如要发布，或者说dev分支代码稳定后可以合并到主分支master上来。

## bug分支

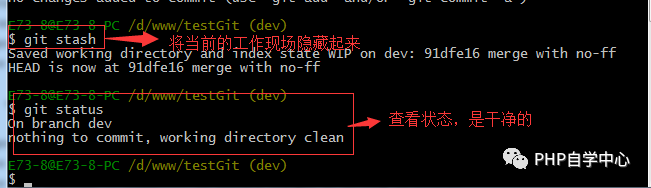
在Git中，分支是很强大的，每个bug都可以通过一个临时分支来修复，修复完成后，合并分支，然后将临时的分支删除掉。

比如我在开发中接到一个404 bug时候，我们可以创建一个404分支来修复它，但是，当前的dev分支上的工作还没有提交。比如如下：



若需要先保存dev分支的工作，可使用：

git stash //将当前工作现场贮存起来

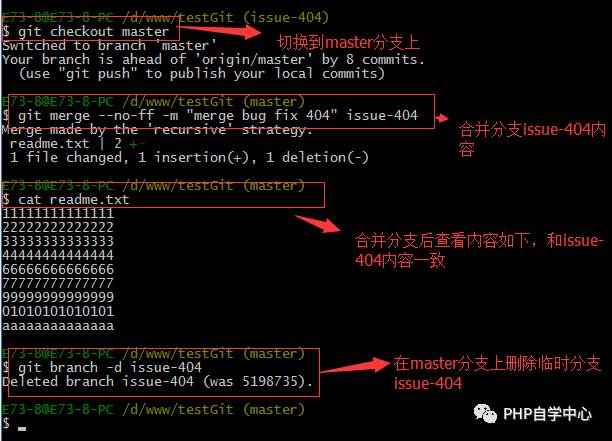


所以现在我可以通过创建issue-404分支来修复bug了。

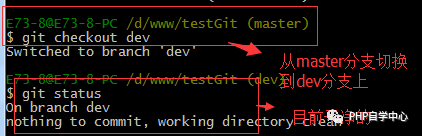
首先我们要确定在那个分支上修复bug，比如我现在是在主分支master上来修复的，现在我要在master分支上创建一个临时分支（此临时分支的内容跟master一样），演示如下：



修复完成后，切换到master分支上，并完成合并，最后删除issue-404分支。演示如下：



回到dev上继续未完成的工作：



git stash list //查看贮存的列表

恢复方法：a> git stash apply 恢复后，stash内容并不删除，你需要使用命令git stash drop来删除

b> git stash pop 恢复同事将stash内容也删除

## 对两个分支合并冲突的理解

以上例子可以看出，当在dev分支更改文件提交后，又在master分支更改相同位置后提交，此时再合并的话就会冲突，因为两个分支上的版本就不一样了。

但是当在master分支下简历dev分支，然后在dev分支上修改提交，再回到master上合并就不会出现问题，因为只有dev的内容变了，master没有变。

即两边同时修改，提交后，合并会冲突，但只有一方修改提交再合并就不会冲突

其中<<<<<<< HEAD 到 ======= 中间的内容是local（本地）提交的。

======= 到 >>>>>>> commit-id 是远程仓库中的内容。

## 多人协作

当你从远程库克隆时候，实际上Git自动把本地的master分支和远程的master分支对应起来了，并且远程库的默认名称是origin。

git remote //查看远程库的信息

git remote –v //查看远程库的详细信息

1. 推送分支

切到本地仓库master分支上，获取所有提交，再找到远程仓库origin中找到master分支，将远程仓库没有的提交都添加进去

git push origin master

git push [remoteName] [localBranchName：remoteBranch]

\*如果想把本地的某个分支test提交到远程仓库，并作为远程仓库的master分支，或者作为另外一个名叫test的分支，如下：

$git push origin test:master // 提交本地test分支作为远程的master分支

$git push origin test:test // 提交本地test分支作为远程的test分支

也可以：

git push origin HEAD^:master //将当前分支向上移动一位，之后推到远程仓库的master分支

若只有：remoteBranch，则会删除远程仓库的分支以及本地的远程分支

git push origin :remoteBranch //删除远程分支

1. 抓取分支

git pull <remote> <remoteBranch:localBranch>

//remote是远程库，remoteBranch是远程库的分支, localBranch是本地分支

后两个参数跟git fetch是一样的，将远程分支更新到指定的本地分支；

该命令首先git fetch remoteBranch 到localBranch，然后git merge localBranch ，合并到本地仓库的当前HEAD处。

若先指定本地dev分支跟远程dev分支的链接，git pull后就不用加任何东西；

git branch –-set-upstream origin/<branch> dev

多人协作工作模式一般是这样的：

首先，可以试图用git push origin branch-name推送自己的修改.

如果推送失败，则因为远程分支比你的本地更新早，需要先用git pull试图合并。

如果合并有冲突，则需要解决冲突，并在本地提交。再用git push origin branch-name推送。

**Git基本常用命令如下：**

　　mkdir：         XX (创建一个空目录 XX指目录名)

　　pwd：          显示当前目录的路径。

　　git init          把当前的目录变成可以管理的git仓库，生成隐藏.git文件。

　　git add XX       把xx文件添加到暂存区去。

　　git commit –m “XX”  提交文件 –m 后面的是注释。

　　git status        查看仓库状态

　　git diff  XX      查看XX文件修改了那些内容

　　git log          查看历史记录

　　git reset  --hard HEAD^ 或者 git reset  --hard HEAD~ 回退到上一个版本

　　(如果想回退到100个版本，使用git reset –hard HEAD~100 )

　　cat XX         查看XX文件内容

　　git reflog       查看历史记录的版本号id

　　git checkout -- XX  把XX文件在工作区的修改全部撤销。

　　git rm XX          删除XX文件

　　git remote add origin https://github.com/tugenhua0707/testgit 关联一个远程库

　　git push –u(第一次要用-u 以后不需要) origin master 把当前master分支推送到远程库

　　git clone https://github.com/tugenhua0707/testgit  从远程库中克隆

　　git checkout –b dev  创建dev分支 并切换到dev分支上

git branch  查看当前所有的分支

git branch -m old\_branch new\_branch 改本地分支名

　　git checkout master 切换回master分支

　　git merge dev   在当前的分支上合并dev分支(合并是按commit时间戳顺序的)

　　git branch –d dev 删除dev分支

　　git branch name  创建分支

　　git stash 把当前的工作隐藏起来 等以后恢复现场后继续工作

　　git stash list 查看所有被隐藏的文件列表

　　git stash apply 恢复被隐藏的文件，但是内容不删除

　　git stash drop 删除文件

　　git stash pop 恢复文件的同时 也删除文件

　　git remote 查看远程库的信息

　　git remote –v 查看远程库的详细信息

git push origin master  Git会把master分支推送到远程库对应的远程分支上

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 一、新建代码库  # 在当前目录新建一个Git代码库  $ git init  # 新建一个目录，将其初始化为Git代码库  $ git init [project-name]  # 下载一个项目和它的整个代码历史  $ git clone [url]  二、配置  Git的设置文件为.gitconfig，它可以在用户主目录下（全局配置），也可以在项目目录下（项目配置）。  # 显示当前的Git配置  $ git config --list  # 编辑Git配置文件  $ git config -e [--global]  # 设置提交代码时的用户信息  $ git config [--global] user.name "[name]"  $ git config [--global] user.email "[email address]"  三、增加/删除文件  # 添加指定文件到暂存区  $ git add [file1] [file2] ...  # 添加指定目录到暂存区，包括子目录  $ git add [dir]  # 添加当前目录的所有文件到暂存区  $ git add .  # 删除工作区文件，并且将这次删除放入暂存区  $ git rm [file1] [file2] ...  # 停止追踪指定文件，但该文件会保留在工作区  $ git rm --cached [file]  # 改名文件，并且将这个改名放入暂存区  $ git mv [file-original] [file-renamed]  四、代码提交  # 提交暂存区到仓库区  $ git commit -m [message]  # 提交暂存区的指定文件到仓库区  $ git commit [file1] [file2] ... -m [message]  # 提交工作区自上次commit之后的变化，直接到仓库区  $ git commit -a  # 提交时显示所有diff信息  $ git commit -v  # 使用一次新的commit，替代上一次提交  # 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 commit --amend -m [message]  # 重做上一次commit，并包括指定文件的新变化  $ git commit --amend [file1] [file2] ...  五、分支  # 列出所有本地分支  $ git branch  # 列出所有远程分支  $ git branch -r  # 列出所有本地分支和远程分支  $ git branch -a  # 新建一个分支，但依然停留在当前分支  $ git branch [branch-name]  # 新建一个分支，并切换到该分支  $ git checkout -b [branch]  # 新建一个分支，指向指定commit  $ git branch [branch] [commit]  # 新建一个分支，与指定的远程分支建立追踪关系  $ git branch --track [branch] [remote-branch]  # 切换到指定分支，并更新工作区  $ git checkout [branch-name]  # 建立追踪关系，在现有分支与指定的远程分支之间  $ git branch --set-upstream [branch] [remote-branch]  # 合并指定分支到当前分支  $ git merge [branch]  # 选择一个commit，合并进当前分支  $ git cherry-pick [commit]  # 删除分支  $ git branch -d [branch-name]  # 删除远程分支  $ git push origin --delete [branch-name]  git push origin :branch\_name //也可删除远程分支  $ git branch -dr [remote/branch]  六、标签  # 列出所有tag  $ git tag  # 新建一个tag在当前commit  $ git tag [tag]  # 新建一个tag在指定commit  $ git tag [tag] [commit]  # 查看tag信息  $ git show [tag]  # 提交指定tag  $ git push [remote] [tag]  # 提交所有tag  $ git push [remote] --tags  # 新建一个分支，指向某个tag  $ git checkout -b [branch] [tag]  七、查看信息  # 显示有变更的文件  $ git status    # 显示当前分支的版本历史  $ git log  # 显示commit历史，以及每次commit发生变更的文件  $ git log --stat  # 显示某个文件的版本历史，包括文件改名  $ git log --follow [file]  $ git whatchanged [file]  # 显示指定文件相关的每一次diff  $ git log -p [file]  # 显示指定文件是什么人在什么时间修改过  $ git blame [file]   | **名称** | **含义** | | --- | --- | | 工作区 | 写代码的地方 | | 暂存区 | git add 到的地方 | | 版本库 | git commit 到的地方 |   # 显示暂存区和工作区的差异(默认不加参数)  $ git diff  # 显示暂存区和上一个commit（版本库）的差异  $ git diff --cached [file]  # 显示工作区与当前分支最新commit之间的差异  $ git diff HEAD  $ git diff commitId 比较版本库的某次commit与工作区的差异  # 显示两次提交之间的差异  $ git diff [first-branch]...[second-branch]  # 显示某次提交的元数据和内容变化  $ git show [commit]  # 显示某次提交发生变化的文件  $ git show --name-only [commit]  # 显示某次提交时，某个文件的内容  $ git show [commit]:[filename]  # 显示当前分支的最近几次提交  $ git reflog  八、远程同步  # 下载远程仓库的所有变动  $ git fetch [remote] [remoteBranch]:[localBranch]  remoteBranch:指远程分支  localBranch:本地分支  该命令可以讲远程仓库remote中的remoteBranch分支内容更新到本地的localBranch分支中（参数顺序跟git push 正好相反）  若省略[]内的参数，将会把所有远程仓库的分支，更新到本地的远程分支中  若参数为[：localBranch],则会在本地新建一个分支  git fetch origin :localBranch //本地新建一个localBranch分支  git branch -a 显示所有本地仓库  其中红色的: ，是远程分支，但是它是属于本地的分支，只是对应的是远程的分支。  使用git fetch 时，其实就是更新了本地的远程分支（上述红色分支），跟远程仓库的分支同步。所以git fetch并不会改变你本地仓库状态  # 显示所有远程仓库  $ git remote -v  # 显示某个远程仓库的信息  $ git remote show [remote]  # 增加一个新的远程仓库，并命名  $ git remote add [shortname] [url]  # 取回远程仓库的变化，并与本地分支合并  $ git pull [remote] [branch]  # 上传本地指定分支到远程仓库  $ git push [remote] [branch]  # 强行推送当前分支到远程仓库，即使有冲突  $ git push [remote] --force  # 推送所有分支到远程仓库  $ git push [remote] --all    九、撤销  # 恢复暂存区的指定文件到工作区  $ git checkout [file]  # 恢复某个commit的指定文件到工作区  $ git checkout [commit] [file]  # 恢复上一个commit的所有文件到工作区  $ git checkout .  # 重置暂存区的指定文件，与上一次commit保持一致，但工作区不变  $ git reset [file]  # 重置暂存区与工作区，与上一次commit保持一致  $ git reset --hard  # 重置当前分支的指针为指定commit，同时重置暂存区，但工作区不变  $ git reset [commit]  # 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 --hard [commit]  # 重置当前HEAD为指定commit，但保持暂存区和工作区不变  $ git reset --keep [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  $ git revert [commit] |

## 高级用法

* 1. git add -p [file\_name]

如果对文件有多次改动，并且想分多次提交时使用



* 输入 y 来暂存该块
* 输入 n 不暂存
* 输入 e 手工编辑该块
* 输入 d 退出或者转到下一个文件
* 输入 s 来分割该块

先输入s，将会分块让你判断该块是否需要提交，y表示add，n表示放弃add。最后commit的只是你选择add的部分。

* 1. 撤销及回退
     1. 场景

场景1：若文件还在工作区，用git checkout -- file 来丢弃工作区的修改

场景2：若文件已经添加进了暂存区，用git reset HEAD <file> 就回到了场景1,即从暂存区删除了，但还在工作区

场景3：若文件已经提交进了版本库，需要用下面的版本回退。

* + 1. git revert

git revert HEAD //撤销最近一次提交

git revert HEAD~1 //撤销最近两次提交（注意:数字从0开始）

git revert commit\_id //撤销commit\_id这次提交（只是把这次提交撤销，不影响其前后的提交）

若无冲突，则会重新生成一条commit，代码是回退的，所有人pull之后，他们的代码也会回退。

若有冲突，需要修改冲突，保留parent…，之后add,

然后git revert --continue 继续回退

git revert --abort 随时终止回退

* + 1. git reset

//适用于还没有将本地仓库推送到远程仓库的时候

//HEAD指当前版本

git reset --hard HEAD^ 指针回退到上一个版本

git reset --hard HEAD^^ 指针回退到上上一个版本

git reset --hard HEAD~10 指针回退到上10个版本

git reset --hard commitId 指针回退到commitId的版本

参数：--soft:仅撤销commit，暂存区和工作区均不变

--mixed: 撤销commit和add ，工作区内容不变

--hard : 暂存区、工作区、版本库都变化

* + 1. git revert 和git reset的区别

git revert: 在正常的commit历史中再commit一次,只不过是反向提交, 创建了一个撤消了上次提交(HEAD)的新提交

git revert 版本号

git revert HEAD 撤销最近的一个提交

git revert HEAD^ 撤销上上次提交

git reset: 只是将指向git版本的指针回退了



重点：如果你的代码已经发布到远程仓库，千万不要用git reset，这样会产生很多冲突，用git revert则不会。

参考链接：<http://gitbook.liuhui998.com/4_9.html>

* 1. 修改commit message信息

1. 首先需要找到你需要修改的那个commit前一个commit，运行git rebase:

|  |
| --- |
| git rebase -i 32e087f |

1. 弹出编辑框，显示该commit后的所有commit，将你需要修改的commit前的‘pick’，修改为‘reword’，之后:wq保存退出，
2. 接着会再次弹出编辑框，这次编辑框中只有之前改为reword的那个commit，此时修改commit message内容，保存后退出
3. 在第二步可以同时改多个，第三步会依次修改依次保存。
   1. git rebase



处理1：用"pull"命令，在mywork分支上把origin分支pull一遍，与你的修改合并，结果会看起来就像一个新的"合并的提交"(merge commit):



git merge处理的结果： C1—C2—C3—C5—C4—C6—C7(手工处理冲突产生一次提交C7)，且git merge合并的commit是按时间顺序合并的。

若采用git rebase：



在mywork分支上：

$ git rebase origin

以上命令会把你的"mywork"分支里的每个提交(commit)取消掉，并且把它们临时 保存为补丁(patch)(这些补丁放到".git/rebase"目录中),然后把"mywork"分支更新 到最新的"origin"分支，最后把保存的这些补丁应用到"mywork"分支上。

当'mywork'分支更新之后，它会指向这些新创建的提交(commit),而那些老的提交会被丢弃。 如果运行垃圾收集命令(pruning garbage collection), 这些被丢弃的提交就会删除. （请查看 git gc)



用合并(merge)和用rebase所产生的历史的区别：



在rebase过程中，也会出现冲突conflict，这时候需要先解决冲突，之后执行git add XXX命令去更新此冲突文件，然后，无需git commit，只要执行

git rebase --continue即可；在任何时候，你可以用--abort参数来终止rebase的行动

git rebase master

git add test.txt 解决冲突后要add冲突文件

git rebase --continue 解决冲突后继续rebase

git rebase --abort 任何时候执行都会回到rebase之前的状态

git rebase barnch1 branch2

branch1:base分支

branch2:需要rebase的分支

tips:若指定branch2，则会先执行checkout branch2，然后再执行rebase。

将branch2的根节点接到branch1后面，并且checkout到branch2分支

git rebase [upstream [branch]]

若指定branch，则会先chekcout 到branch分支，后执行rebase；

若不指定upstream，则会rebase 默认绑定的远程对应分支

* 1. 交互式rebase

git rebase -i [start] [end]

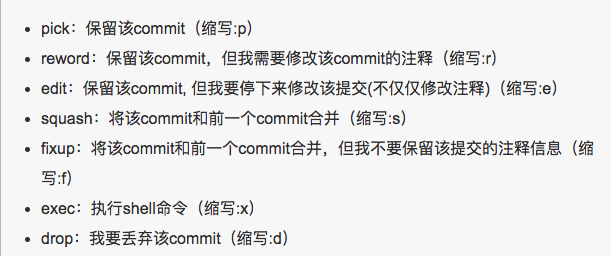
git rebase -i HEAD~4

-i:-interactive

会出现一个UI界面，里面你可以调整该分支中commit的顺序，也可以删除commit等

start/end为一个区间（前开后闭），若不指定end，则默认至HEAD

每个commit前都会有以下命令可选



* 1. cherry-pick

将一个分支的某些提交，复制到另一个分支

命令：

git cherry-pick commitId 复制单个commit到当前分支，放置到当前分支的顶部

git cherry-pick C1 C2 复制多个commit，不连续,有顺序

git cherry-pick commitId1..commitId2 复制连续的多个commit

注意：commit之间为两个’.’ 并不包含commitId1（前不包含，后包含）

若复制过来的分支跟当前分支有冲突，先解决冲突，然后add && commit，然后再继续执行：

git cherry-pick –continue

* 1. HEAD指针

默认情况下，HEAD指针指向分支名，分支名指向commitId

即：HEAD -> master -> C1

分离的HEAD：

git checkout C1 将HEAD移动到commitId处的提交

此时HEAD与分支名分离，即：HEAD -> C1

git checkout master^ 表示将HEAD移动到master分支的父节点(先切分支后移动HEAD)

git checkout HEAD^ 将HEAD向上移动1位

git checkout HEAD~3 将HEAD向上移动3位

^和~的区别：

^指向上移动一位，^^向上移动两位，^2表示向上移动到另外一个merge分支



^后不加数字，默认移动到正上方那个commit，加数字2则移动到领带一个 merge分支。

链式移动：

git checkout HEAD~^2~2

//想向上移动一位，后切到merge的非主分支，后再向上移两位

强制修改分支位置：

比如将master分支指向另一个分支的某个提交，

注意：该命令只是将分支的位置更改了，并没有切换分支（并没有checkout）

//将master分支指向HEAD指针向上移三位的位置或commitId的位置

git branch -f master HEAD~3

git branch -f master commitId

* 1. git commit --amend

修改最近一次提交，将替换掉最后一次提交，采用本次提交

* 1. git tag v1.0 commitId

为commit设置tag

//查找最近的tag

git describe <ref>

ref: 任何能被 Git 识别成提交记录的引用,可以是分支名，也可以是commitId,若不知名，默认为HEAD

返回结果：<tag>\_numCommit\_g<hash>

tag名，ref距离tag有多少commit，tag所在的commitId

* 1. git pull

拉取远程仓库的步骤实际是这样的：

git fetch //拉取远程仓库的更新，并将其同步给本地的远程分支例如origin/master

然后使用方法1：

git rebase origin/master 可以将master分支 rebase到刚更新下来的远程分支中，这样commit就是一条直线；

也可以使用方法2：

git merge origin/master 将master分支与刚更新的远程分支merge，这样commit就会有分支；

最后git push ，这是远程仓库更新了，本地的远程分支也会更新。

以上操作：

git fetch + git rebase origin/master 就等于git pull --rebase

git fetch + git merge origin/master 就等于git pull

* 1. 远程跟踪

修改对应跟踪关系：

方法1:

//在本地建一个totallyNotMaster分支，跟远程仓库的master分支对应。

git checkout -b totallyNotMaster origin/master

方法2：

通用句式：

git branch (--set-upstream-to=<upstream> | -u <upstream>) [<branchname>]

branchname若不指明，则默认指向当前分支

git branch --set-upstream-to=origin/master foo

简写：git branch -u origin/master foo

//若当前分支就是foo，foo参数可以省略

* 1. 删除本地的远程分支

当删除了远程仓库的分支后，执行git branch -a 发现本地还存在很多远程仓库已经删除的远程分支，这时可以执行：

git remote show origin //可以查看remote地址，远程分支，还有本地分支与之相对应关系等信息。



这些状态为stale的，就是要被移除的，

执行：git remote prune origin 即可移除所有stale的分支。

或者直接

git pull --prune

git fetch -prune

来删除本地已无用的远程分支

## git 忽略

.gitignore文件只能忽略掉那些没有被git track的文件；

若已经提交到版主仓库里的文件，需要先git rm --cached删除追踪状态 ,之后再添加进.gitignore文件中。

git update-index --assume-unchanged 只能忽略已在版本控制中的文件；但是若该文件被人修改过并更新到远程仓库，你还是能更新下来该文件，并且你对该文件的忽略也会失效，你必须重新git update-index --assume-unchanged。

正确使用git update-index的场景是：当你正在编辑一个巨大的文件，还未做完，不想每次检查变更（因为很慢），使用git update-index，将其忽略，当工作完成后，再重新恢复。