# Git

## 创建仓库

### create a new repository on the command line

echo "# TestGit" >> README.md

git init

git add README.md

git commit -m "first commit"

git remote add origin https://github.com/yk14123/TestGit.git

git push -u origin master

### push an existing repository from the command line

git remote add origin https://github.com/yk14123/TestGit.git

git push -u origin master

git remote add origin <https://xxx>

使用origin来代表https://xxx的地址

git push –u origin master首次使用

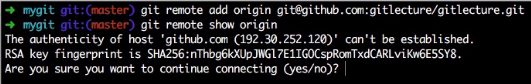
-u指的是，建立origin远程和本地master联系

下次直接使用git push 即可！

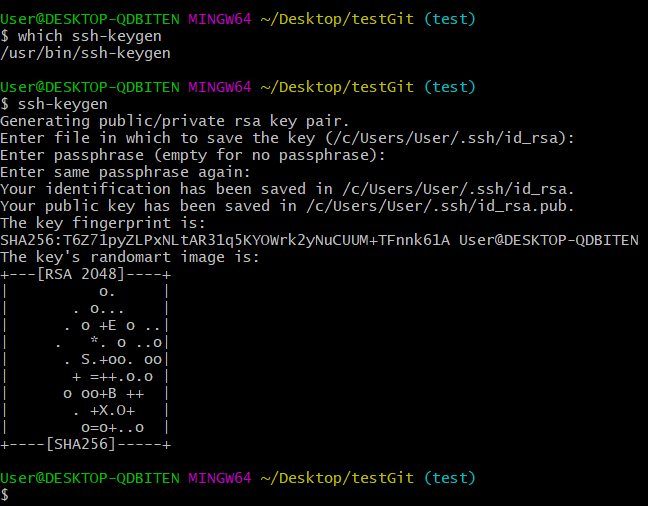
### SSH方式

核心：生成公钥和私钥，然后把公钥放到github服务端。私钥保存在本地。每次push的时候，验证通过才能push

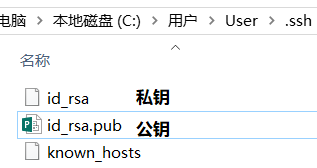
SSH的key只能用一次。如果已经使用，需要删除重新生成配置



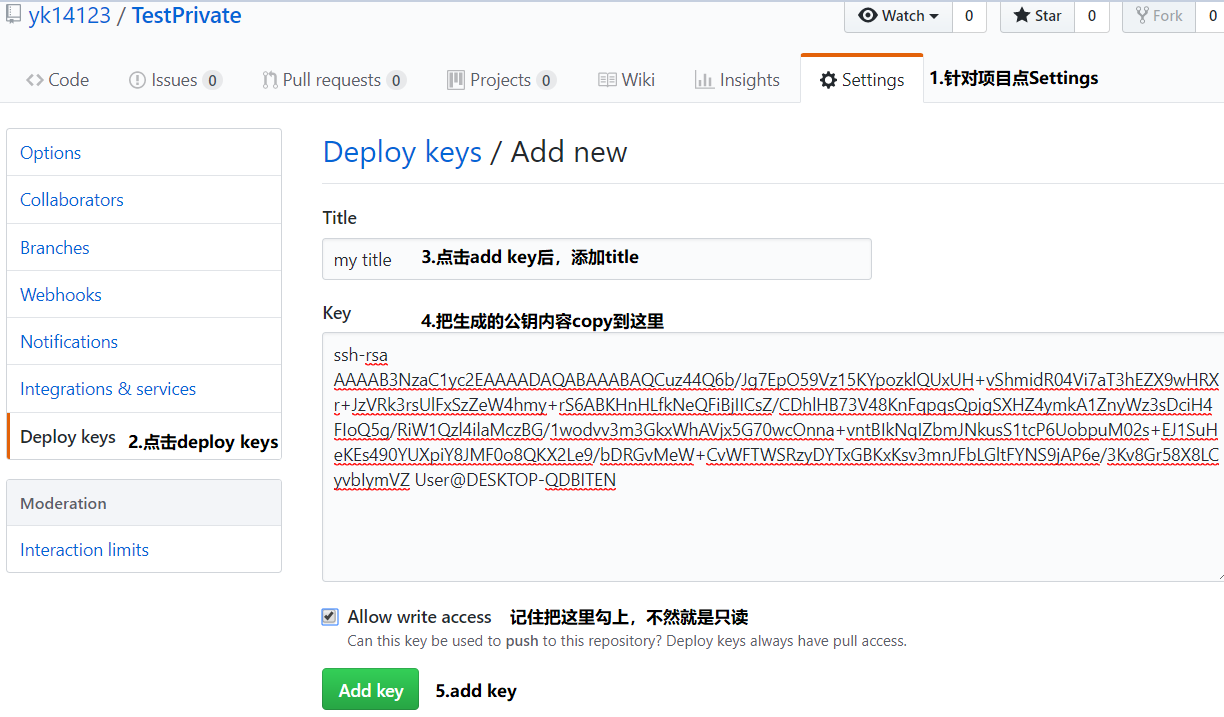
1.添加ssh传输



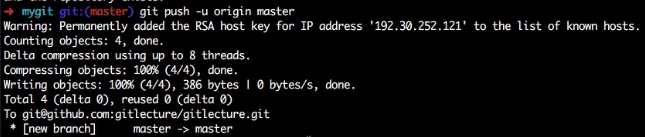
2.生成ssh公钥和私钥



3.把ssh生成得公钥copy到github的deploy key。如下图



4.最后使用git push –u origin master即可完成push



### 配置用户名和密码

针对用户全局



git config --global user.name “xxxx”

git config --global user.email xxxx@xxx.com

具体配置xxx的时候，加或者不加双引号”都可以。或者’单引号也可以

针对特定仓库

git config --local

同时配置 local优先级最高，global次之，system最低

## 文件状态介绍

### 工作区状态

不管文件目前在什么状态，只要有新的修改，都会来到工作区状态。

checkout后会恢复到之前的状态。

如果之前是版本库状态，就恢复到版本库状态，如果是暂存区状态，就恢复到暂存区状态

#### git add

通过git add . 添加到暂存区

### 暂存区状态

#### 全新文件删除

新文件，新图片，指的是没有被添加到版本库的文件！已经被add进入了暂存区。

如果需要删除，有2种操作方式

##### 方式一：这时直接rm 新文件。

会出现有趣的事情，暂存区里面文件还在，但是文件在目录里面被删除了

这时候，delete文件，这个动作，将作为新的修改，添加到工作区！这时需要确认真的删除，还是恢复

真的删除操作：git add .

真的恢复： git checkout .

总结：先rm，其实文件已经移除了，再add一下，把删除动作同步到暂存区！！

先rm再同步add

##### 方式二：先reset

这样的话，会把新文件，从暂存区，回退到工作区。

因为是新文件，还没有添加到版本库里面，所以checkout .无效！！只能通过rm 新文件来删除

总结：先reset，只是把文件移到工作区，文件还在，再rm一下，执行删除！

先reset移到工作区，再rm删除。必须要用Linux命令删除



##### 方式三：git rm --cached效果同方式二



git rm --cached newFile.txt可以让新文件，从暂存区，还原到工作区

效果同git reset

#### git commit

通过git commit –m “介绍提交的内容” 添加到本地版本库

#### Git commit -am “xxxxx”

该命令等同于把 git add .和git commit -m 合二为一！！

前提是该文件已经在版本库存在！

### 本地版本库状态

如果文件已经被提交到了本地版本库后，

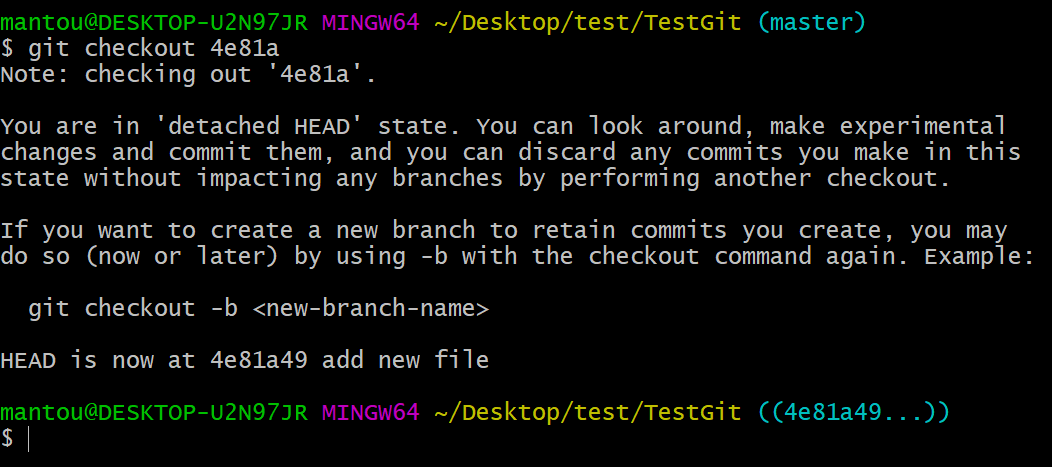
#### Git checkout commitId

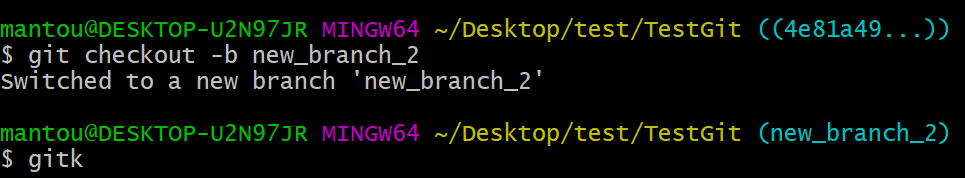
Git reset –-hard commit\_id和 git checkout commit\_id的区别

git checkout commitId的区别：

1. HEAD不移动，HEAD还是在原始分支的最后一次提交位置
2. 适用于在整个链路中的之前某个位置，分叉另一个分支，而保证整体提交链路不变

原来创建分支，都是在当前HEAD下，创建一个和原来一模一样的分支，现在想在之前的某个提交点上，创建新的分支，怎么办，使用git checkout commit\_id





使用git checkout –b <新的分支名> 来给当前游离的分支赋予名字！

可以看见，在4e81a这个提交点，创建了一个新的分支

在这个分支修改后，最后可以合并给HEAD！



如果直接做了修改，而忘记创建分支。也会报提示

git branch <new-branch-name> commit\_id 来创建分支！！

Git reset -–hard 是把整个链路版本回退！！

#### 3种回退本地版本方式

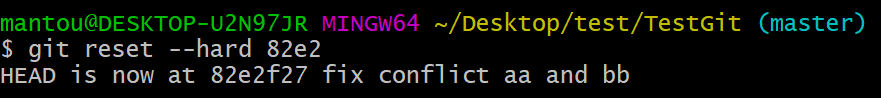
##### Git reset --hard HEAD^

Git reset --hard HEAD^^^^

##### Git reset --hard HEAD~1

Git reset --hard HEAD~4

##### Git reset --hard commit\_id



回退到任意一个版本，不管是新还是旧，通过commit\_id比较方便。

可以通过git reflog，查看历史的commit\_id

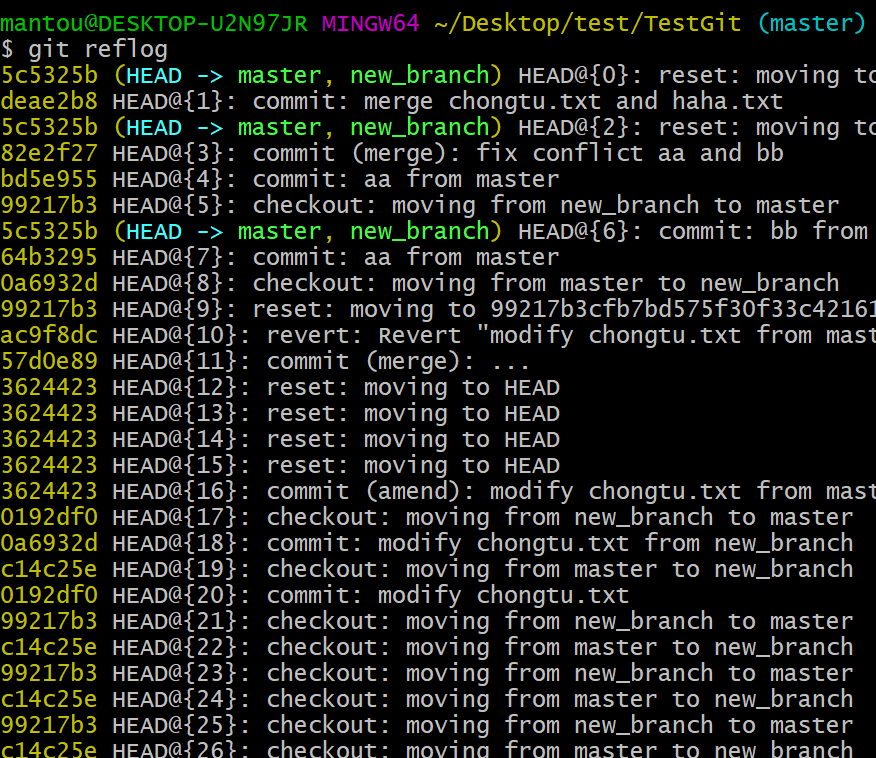
Commit\_id只需要输入4-5个就可以了

#### Git reflog

Git reflog是记录操作日志

所有操作过的日志，id全部记录下来了！

我们针对于HEAD的所有操作，都会被记录下来



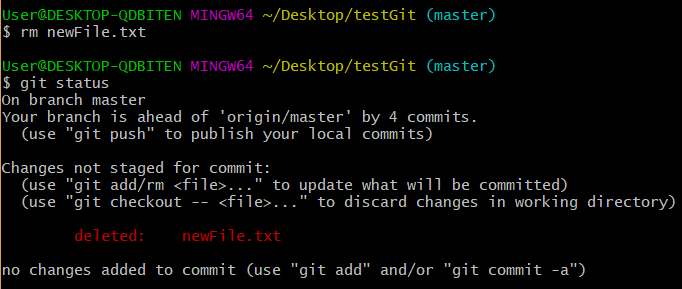
#### 修改文件

那么修改文件的话，文件会进入工作区！

再add 进入 暂存区

最后commit 完成提交后，修改成功！！

#### 删除文件



通过Linux命令删除版本库文件 rm xxxx

这样的话，删除文件动作，直接到了工作区！

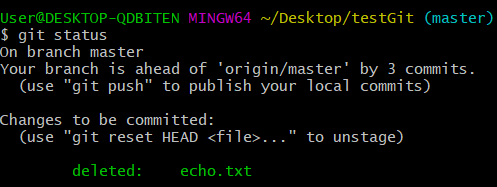
撤销删除快，1步，checkout

确认删除慢，2步，add，commit



通过git命令删除版本库文件 git rm xxxx

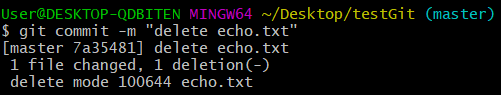
这样会把删除文件的动作，从版本库移动到暂存区



撤销删除慢，2步，reset，checkout

确认删除快，1步，commit

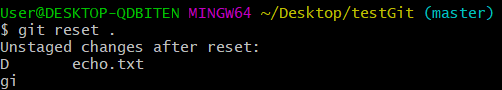
##### 删除版本库文件



如果需要真的删除，需要重新git commit

因为版本库的文件，git视为极其重要的文件，所以必须仅仅通过commit来确认删除！！

##### 撤销删除版本库文件



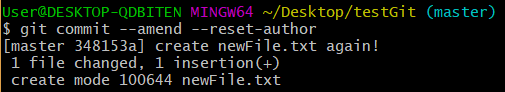
先把删除文件动作，从暂存区，移动到工作区



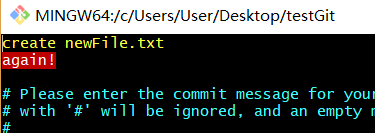
再通过工作区checkout，撤销该命令

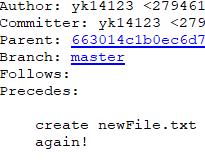
#### 修改commit注释

##### git commit --amend --reset-author

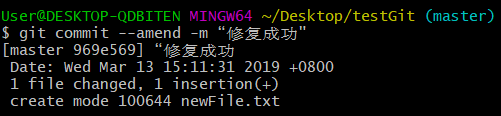


这个命令是精准修复注释！



通过gitk 发现注释已经改变

##### git commit –amend –m “再次修复”



这个命令可以直接简单修复注释！

#### git push

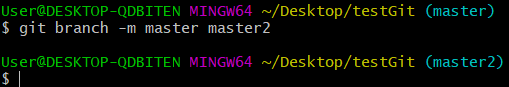
通过git push 添加到远程版本库

### 远程版本库状态

## 本地分支

HEAD指向的是分支。而不是分支指向的提交

### 分支改名



git branch –m <原分支名> <改后的分知名>

### 创建分支



创建分支，是在当前记录点下创建分支

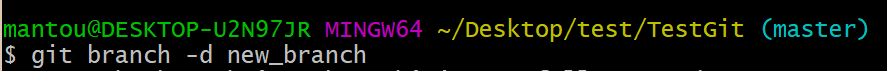
git checkout –b new\_branch 既创建分支，又切换到new\_branch分支

### 切换分支



git checkout – 切换到上一个分支 和 cd – 是一个意思。回到上一个操作

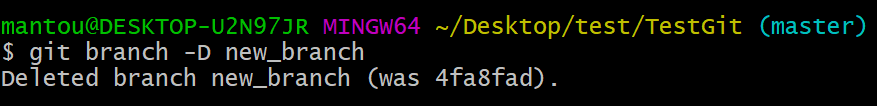
### 删除分支



删除分支，只能在其他分支的基础上进行操作

不能在自己的分支删除自己

#### Git branch -D new\_branch



当前有2个分支，master和new\_branch分支

New\_branch分支在master分支之后，添加了一个新文件haha2.txt

而master没有这个文件

这时，如果在master分支上，删除new\_branch分支，会出现丢失haha2.txt文件的情况

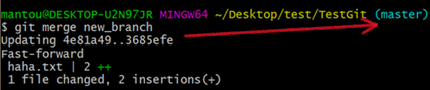
这时，使用git branch -d new\_branch 无效

必须要使用git branch -D new\_branch才可以

-D使用大D删除分支，则会把当前分支，还有该分支独有的文件一并删除！！

### 合并分支

git merge



把new\_branch的新内容，合并到master上面

把master分支的内容，变成new\_branch的内容和自己原来的内容的合并！

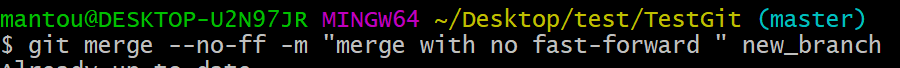
让多个提交线，合并到一个commit线当中

多个branch，只是指针，指向着同一个commit提交线中

Git默认使用fast-forward模式

直接让旧的提交指针，指向新的提交

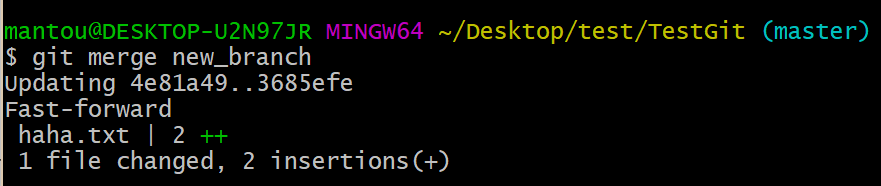
在这种模式下，删除分支会丢失分支信息



Git merge –no-ff -m “xxxxx” <分支名>

这样合并，禁用fast-forward模式，就产生新的提交点

#### 没有冲突直接合并



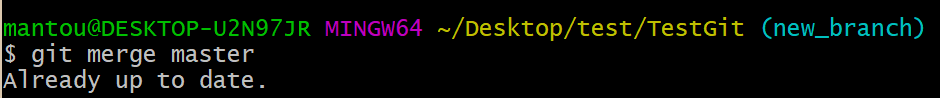
切换到旧的，被合并分支

然后git merge 新的，合并分支

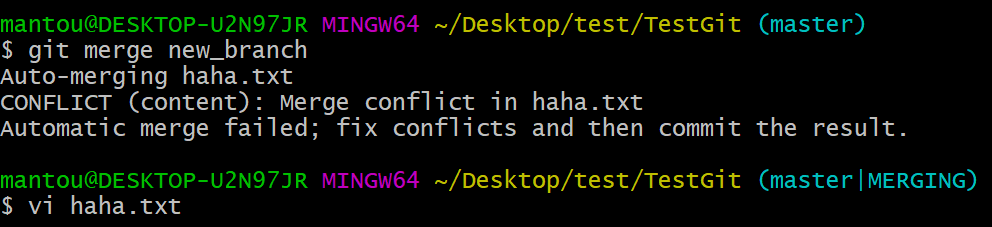
把new\_branch上的新增修改内容，合并到master上面

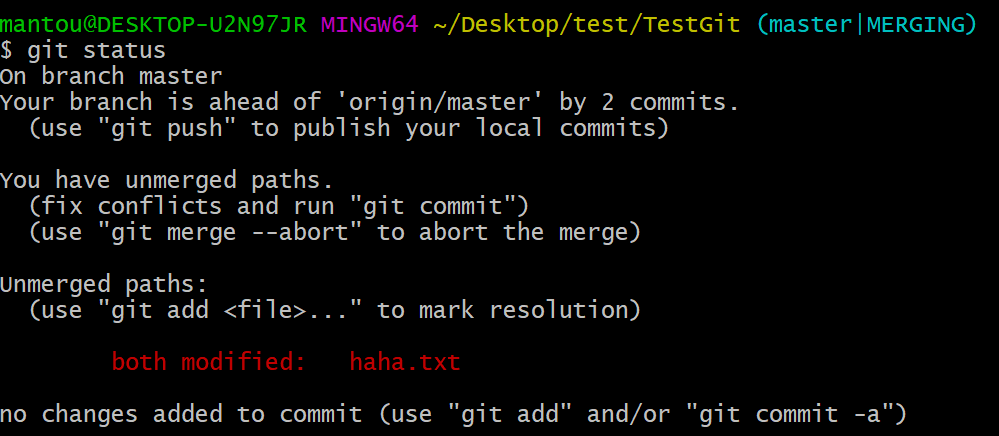
因为new\_branch节点比较新，把新的合并到旧的上面

把旧的合并到新的，是无效的

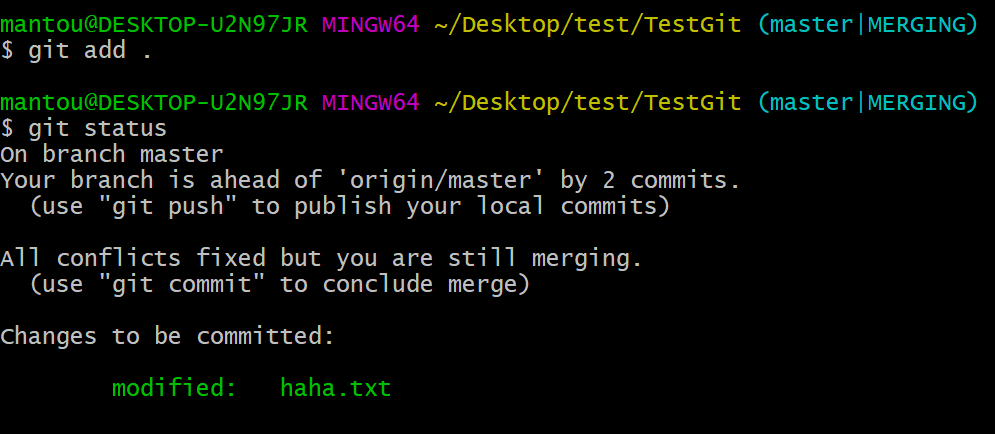


#### 有冲突



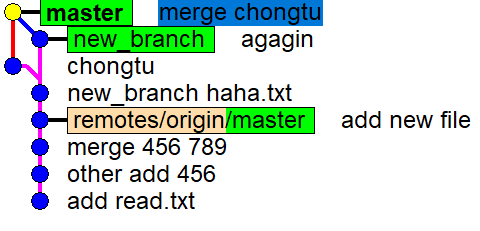


通过git add 来标记已经解决冲突

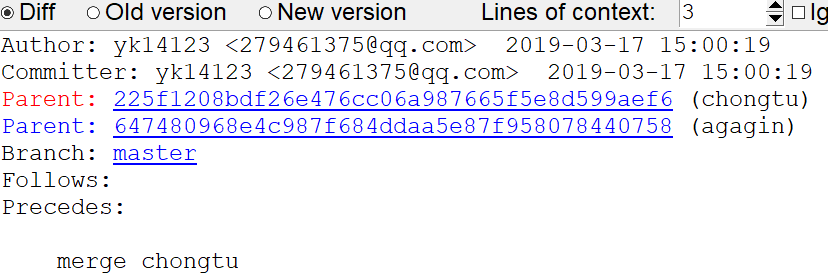


All conflicts 已经被fixed。需要commit 来确认提交

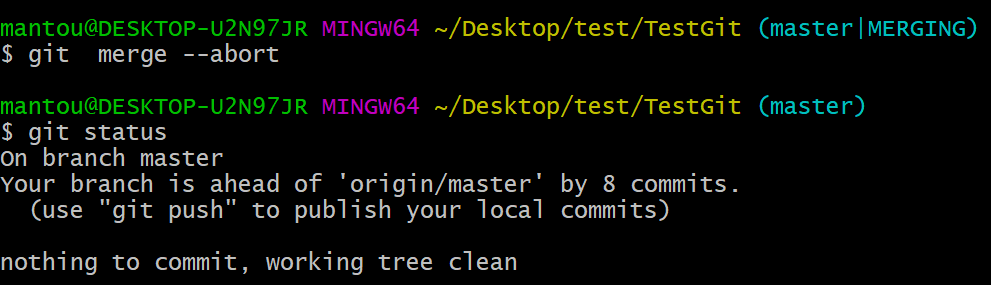
处理完冲突以后，重新add commit 即可解决冲突



可以看见，当前master节点，指向了2个父节点



##### 取消合并分支



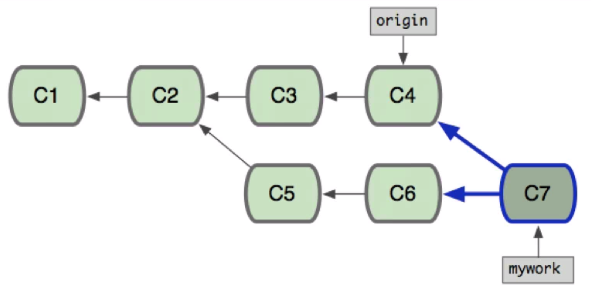
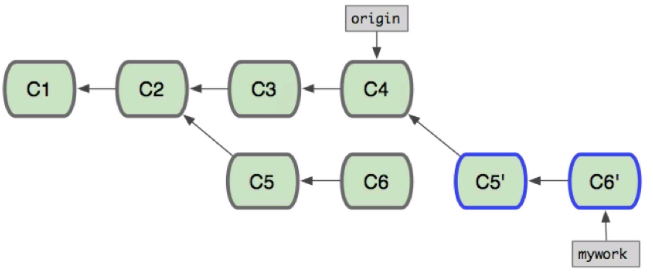
Git merge --abort

### git rebase xxx

rebase 变基，改变根基

一般来说，rebase都是针对自己本地分支，没有推送到远程版本库

不要对master分支执行rebase

左一为merge的效果 右一为rebase的效果

左一的git命令为

git checkout mywork

git merge origin

右一的git命令为

git checkout mywork

git rebase origin

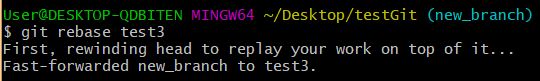
把mywork rebase到 origin的后面！！

把mywork分支的内容，从与origin的祖先分支提交点，开始计算copy一份，然后拼接到origin的提交点的后面！

现在mywork和origin完全是一条整体分支！

Rebase：Copy+拼接的过程中，只要完全没有冲突，就fast-merge合并！有冲突，就停下来处理冲突！！

#### 没有冲突直接合并

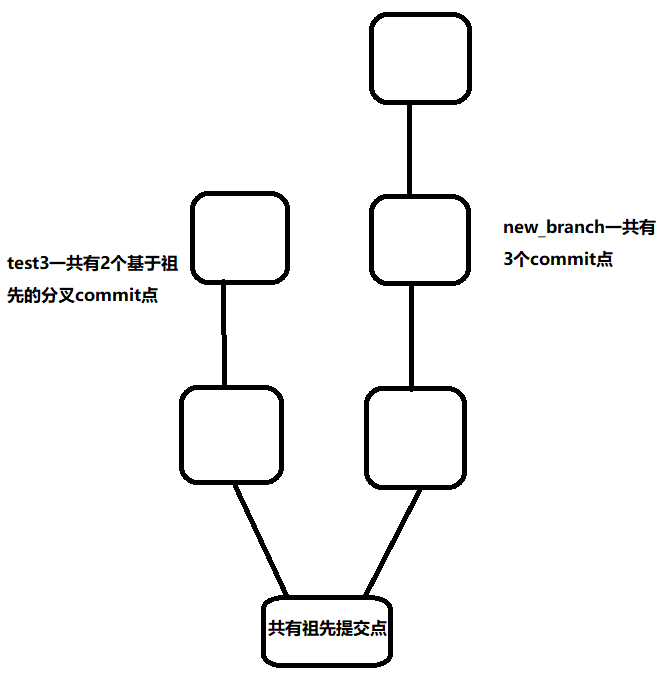


把new\_branch上面的提交，copy一份，然后拼接到test3分支的后面！！

没有冲突的话，会执行快速合并！

git rebase <最终合并到哪一个分支的后面>

#### 有冲突

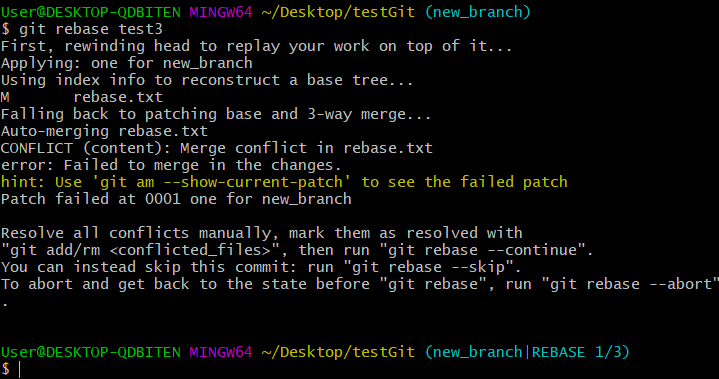


所以当我们执行

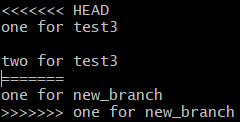
git checkout new\_branch

git rebase test3的时候

因为需要拷贝当前new\_branch分支下的3个commit，copy并且拼接到test3分支提交的后面，所以我们可以看见new\_branch | REBASE 1/3 这里是3个！



3个commitId的copy，提交点。



因为是new\_branch的3个节点，拼接到test3后面，所以有趣的事情：HEAD是基于test3的！

如果某一个copy过来的提交点和被拼接的分支没有冲突的话，会执行快速合并！

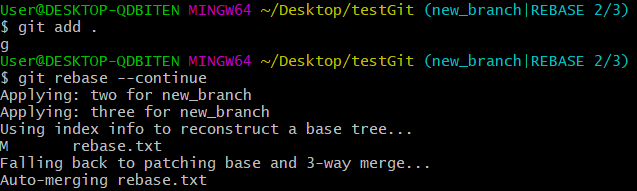


比如new\_branch的第3个commit节点，只是新增了一句话，而这一句话和原来test3完全没有任何冲突。所以直接auto fast-merging快速合并！

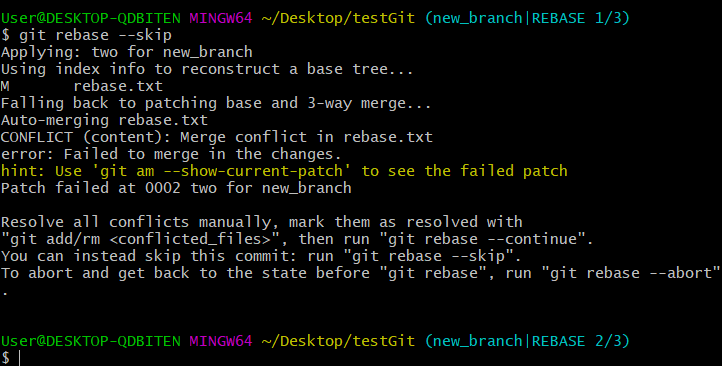
##### git rebase --continue

解决冲突后，先进行git add .添加

再执行git rebase --continue



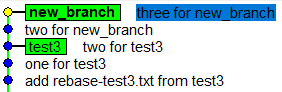
##### git rebase --skip



比如，这里有3个copy过来的提交

这里的操作，表示第一个copy过来commit节点的任何改变，都不应用到最终的结果中！直接把第一个commit提交点完全抛弃！

本次合并的结果如下图：

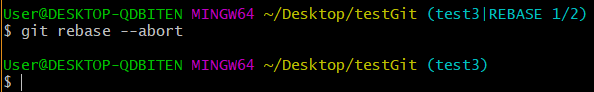
 

可以看见，one for new\_branch这个提交点已经丢弃了！！合并的最终结果里面没有它

###### 心得体会：

1. 如果只是纯粹的代码冲突。那么比如rebase，copy了10个节点。前面9个节点都可以跳过合并（执行git rebase --skip），只需要最后一次git rebase --continue即可。因为最后一个copy过来的commit提交，完全能够包含前面9次的代码修改！！
2. 但是，如果前面10个节点，有新增加的文件！则执行git rebase --skip后，新增文件会被丢弃！！所以最好还是每个commit都仔细查看比较合适！！
3. 合并冲突的过程中，如果2个节点，完全没有冲突，会自动执行合并！

##### git rebaes --abort



合并的任意时刻，都可以随时取消rebase合并！

### git rebase -i xxx

效果和git rebase xxx一样

也是基于相同的祖先提交节点，copy+拼接到指定的分支后面

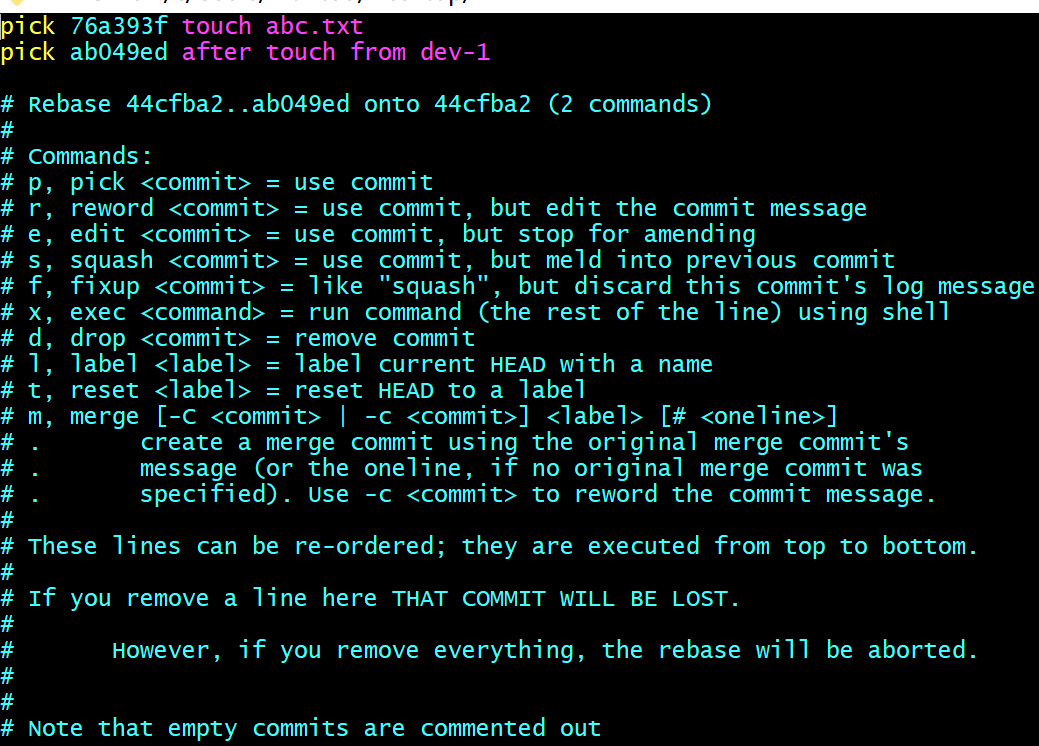
不过该命令，极其灵活，可以完全自定义：

1. 完全自定义copy后的提交节点顺序
2. 完全自定义copy后的提交节点增减
3. 完全自定义copy后的提交message

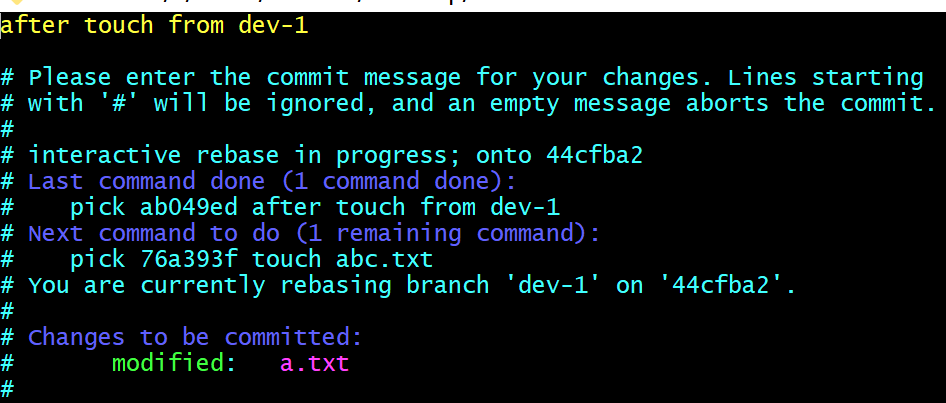
执行git rebase -i <需要被拼接的分支名>后，弹出如下，信息。

可以完全自定义一切

可以增减和调换顺序！



如下是完全自定义commit message



### git cherry-pick xxx

前提是保留2个完全独立的分支！需要把一个分支的提交修改，copy一份到另一个分支。2个分支完全独立

有2个分支

test3和new\_branch。我在new\_branch分支上面，提交了2次，C1和C2



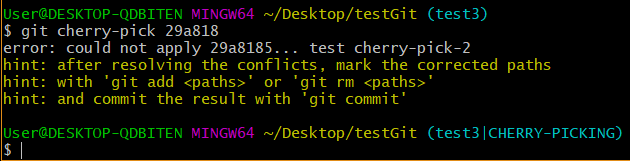
但是其实这2次提交，我也要应用到test3分支上面。怎么操作呢？

new\_branch分支上的2次提交也保留，copy一份给test3

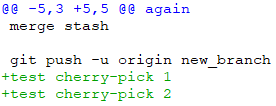
1. git checkout test3(切换到test3分支)
2. git cherry-pick commit\_Id（把new\_branch上的commit\_id提交，复制到test3）

如果依次拷贝，如先把 test cherry-pick-1 拷贝，再把test cherry-pick-2拷贝，2次依次执行git cherry-pick commitId，

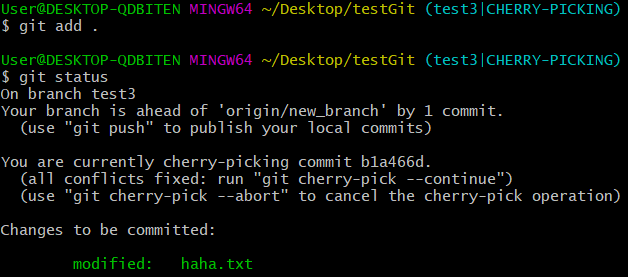
直接把test cherry-pick-2 的commitId拷贝给test3，如下图，会出现冲突！

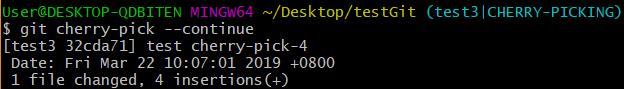






#### git cherry-pick --continue

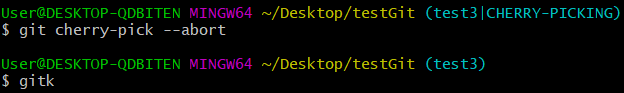




1. --continue是针对暂存区的操作！
2. 当修复冲突后，先git add .添加到暂存区
3. 有2种修复方式，可以调用git cherry-pick --continue或者直接git commit
4. git cherry-pick --continue后，会出现让你输入commit message的对话框。输入确认后，就完成提交

本质和git commit区别不大

#### git cherry-pick --abort



退出冲突

相当于git cherry-pick <commitId>该命令不执行，而退出来

## 远程分支

需要先git pull

再git push

1. master 很稳定的，生产系统发版本的分支
2. test分支，测试人员在该分支上面拉取代码测试bug
3. develop，日常开发，不停的变换，每天更新
4. hotfix 当出现紧急bug的时候，从master分支拉取代码，修复再合并

### 完整命令

#### git push origin srcBranch: destBranch

git push <远程地址名> <本地分支>:<远程分支>

origin：github地址

srcBranch：本地分支名

destBranch：远程分知名

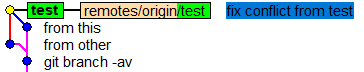
#### git pull origin srcBranch:destBranch

origin：github地址

srcBranch：远程分支名

destBranch：本地分知名

### git pull

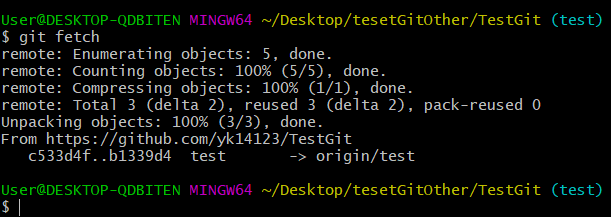


pull = fetch + merge

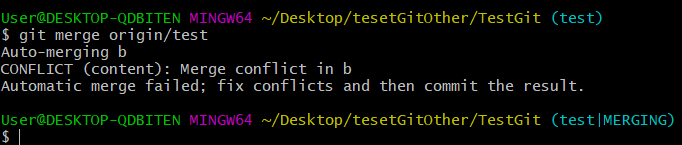
可以看见。有2个同事都针对某个文件，提交了2个分支

这时合并merge会看见2个分支提交情况！

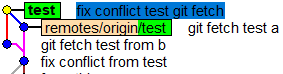
### git fetch



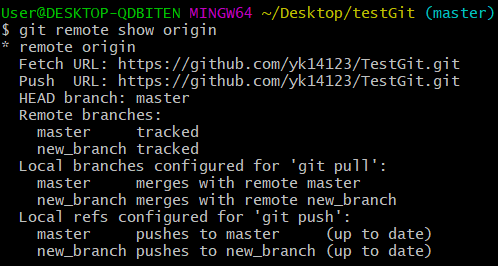
git fetch是一定会成功的。因为只是把网络上面的提交下拉到本地。至于和本地有没有冲突，fetch不关心



合并提交后，gitk查看结果和git pull一样

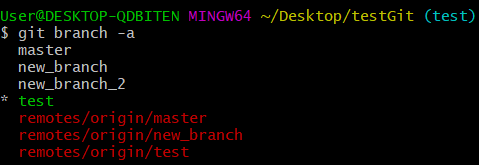


### git remote show origin



查看远程关联分支详细信息

### git branch -a



查看所有分支，包括远程分支

### 推送到远程分支

refspec 引用规则

适用于远程没有分支！，本地init后，第一次推送到远程



如果新建的分支，在远程没有。那么使用如下命令，建立引用规则：

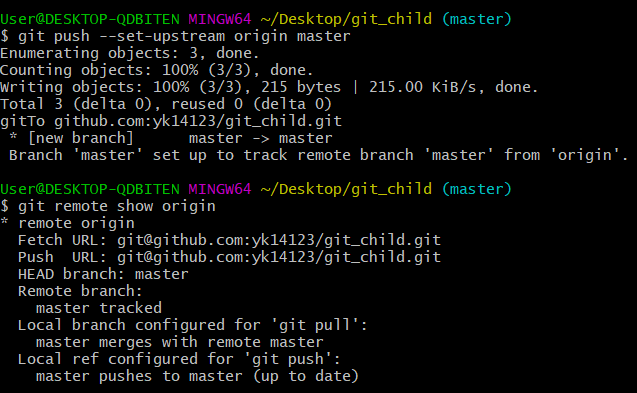
git push --set-upstream origin test

1. 远程的分支名称和本地的分支名称一样
2. 本地的test分支push到远程的test分支

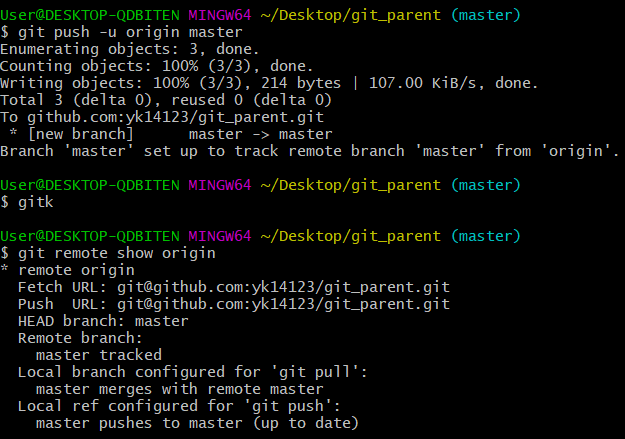
git push --set-upstream origin <本地分支名>:<单独的远程分支名>

git push origin <本地分支名>:<单独的远程分支名>

如果喜欢推送到远程的时候，远程分支名为另外的单独的名字，可以使用上面的命令



git push –u origin test效果同上，建议用set-upstream命令



### 从远程分支拉取最新数据

远程已经存在分支了，本地有个已经存在的分支希望与其建立pull建立

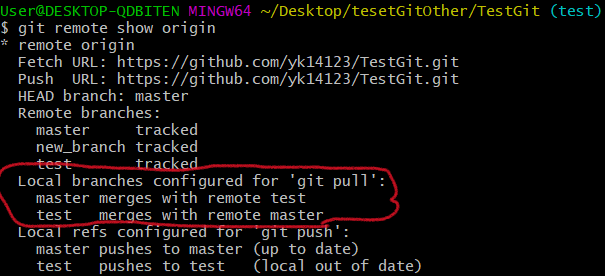


git branch --set-upstream-to=origin/<branch> test

git branch --set-upstream-to=origin/<远程分支名> 本地分支名

设置本地的branch，关联远程的branch

随意切换，设置本地分支，到底是从远程哪个分支pull数据！



可以看见这里 master分支和test分支互换了一个pull位置

### git checkout –b dev origin/dev

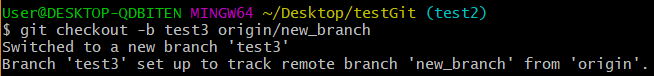
远程已经存在分支了，本地有新的分支希望与其建立pull建立

该命令等同于

先git branch dev

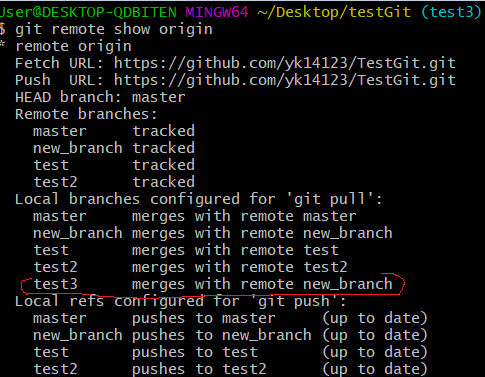
再 git branch --set-upstream-to=origin/dev dev

的合并版本



新建一个a分支。该分支和远程的某个分支仅仅建立pull关系！

建立pull关联，远程不创建a分支

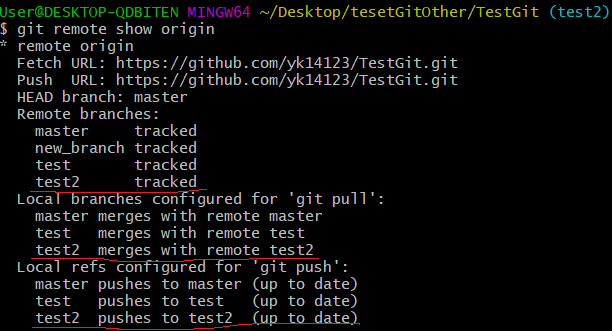


### git checkout --track origin/test

远程已经存在分支了，本地有个新的分支希望与其建立pull和push关系

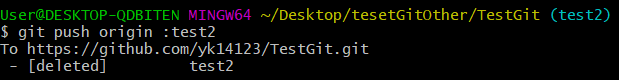
在本地新建一个test分支。和远程test建立完全的pull和push关系





### 删除远程分支

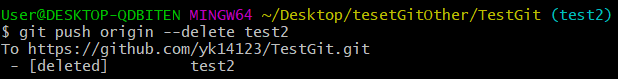
方式一：



git push origin <空>:<远程分支名>

把一个本地的空分支，推送给远程。从而达到删除远程分支的目的

方式二：

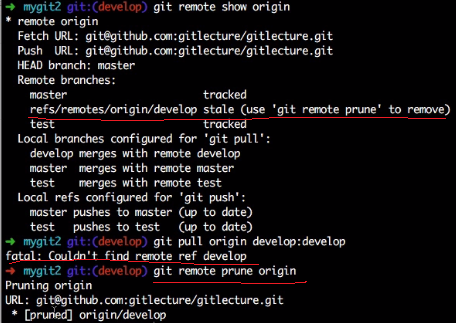


git push origin --delete <远程分知名>

#### 删除分支特殊状况

A同事删除了远程分支。B同事再pull得时候，会出现问题

解决如下：



git remote prune origin

同步一下。把本地的分支也删除即可

### 重命名远程分支

只有先删除远程分支

再在本地创建新的分支

再重新push，以达到改名远程分支名的效果

## SubModule

单向修改

child项目，只能由child人员维护修改

主项目，只能pull，read，不能修改child项目

主项目也可以修改，不过有很多问题

### 背景

为什么需要subModule

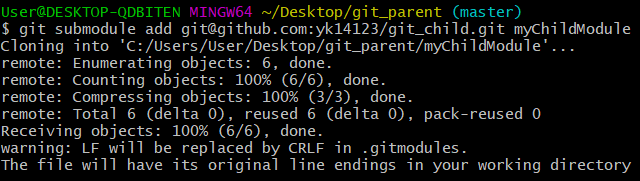
如果一个项目A，需要依赖另一个项目B

1. 可以把B打成jar包引入
2. 可以把B代码放在服务器上，通过gradle引入
3. 可以手工拷贝B代码引入

而项目B要频繁更新，所以以上方式都不合适

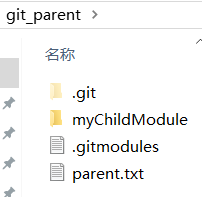
### 添加sub依赖

把项目child，引入到parent项目

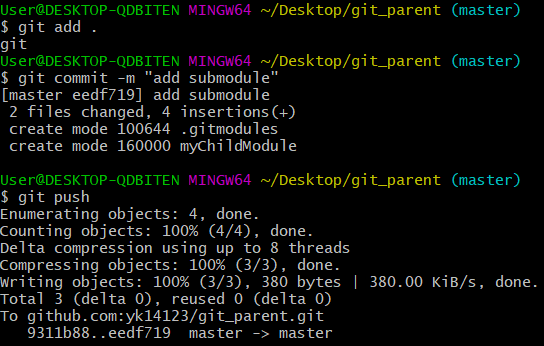


git submodule add <子项目地址> <引入到parent项目后存放的目录>

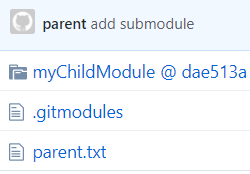
引入到parent项目后存放的目录，该目录不能提前存在！！

有1个myChildModule文件夹和.gitmodules隐藏文件

正常 git add . git commit –m “xxx”后

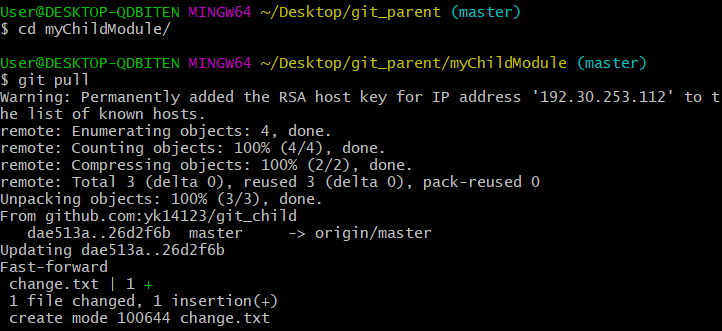


git push完成可以看见如下情况

点击myChildModule后，要跳转child项目！！！

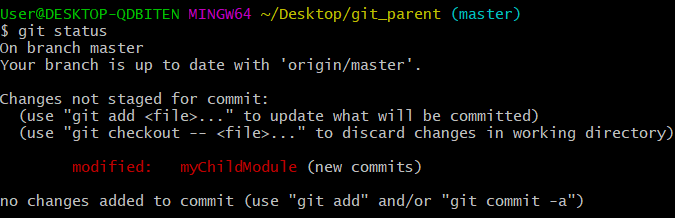
### 下载最新依赖

#### 方式一：进入子module目录逐个更新



只需要cd 进入module的文件夹

然后在这个文件夹里面，执行git pull即可！！



回到主项目parent目录，可以看到git status报红色：子module被修改

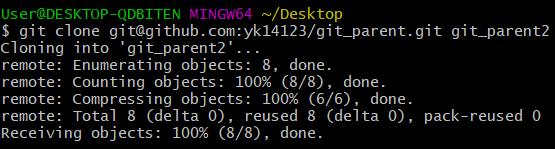
正常git add. git commit –m “xxx” git push即可

#### 方式二：一次性更新全部子目录module

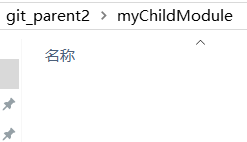


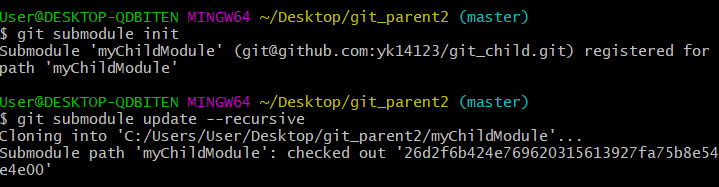
git submodule foreach git pull 一次性全部拉取子module！

#### 方式三：git clone下载主项目



使用git clone下载带子module的主项目的时候，子module无内容！！

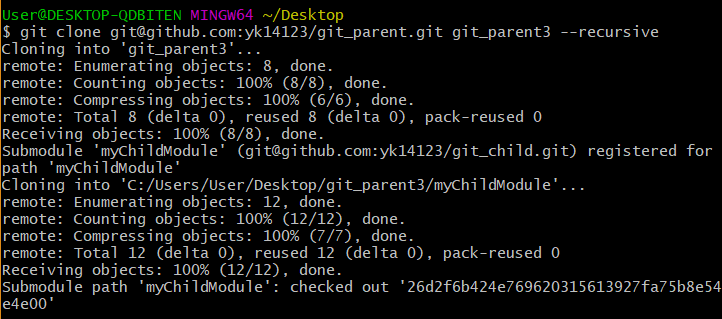




需要依次执行

1. git submodule init
2. git submodule update --recursive

done!



或者直接执行，如下命令，可以节省以上3步

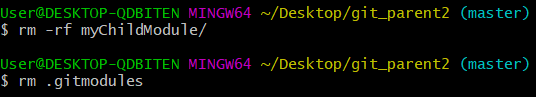
git clone <远程主项目地址> <拷贝到本地哪个文件夹的名字> --recursive

拷贝到本地哪个文件夹，该文件夹不能提前存在！

### 删除子module

1. 将submodule从缓存区删除
2. 将submodule的实体从工作区删除
3. 最后将.gitmodules目录删除





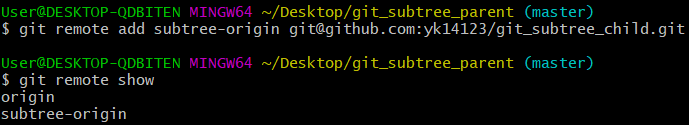
## SubTree

双向修改

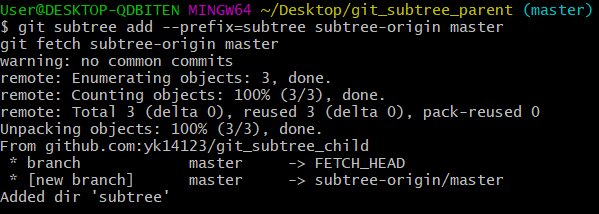
child项目，可以由主项目人员或者child人员随意维护修改

主项目也可以修改，没有问题

### 添加subTree依赖



先添加网络别名，subtree-origin代替child项目的http url地址



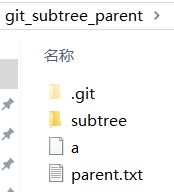
git subtree add –prefix=<主项目新建文件夹名> <子项目远程地址> <子项目远程分支> --squash

主项目新建文件夹名：把子项目放在主项目下的哪个位置，哪个文件夹的名称！

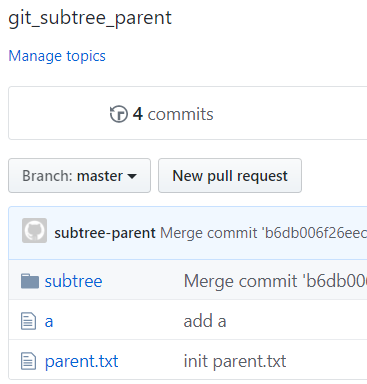
子项目远程地址：可以用别名代替

子项目远程分支：master

--squash：没有子项目的提交历史，log被压缩合并了。

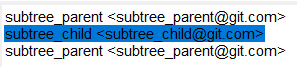
只有subtree文件夹

git push后，可以看见如下结果：

点击subtree后，不跳转child项目！！！

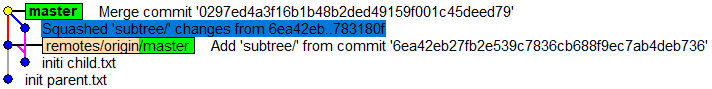
#### squash

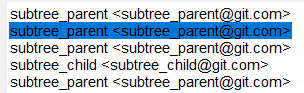




如果不加--squash的话，那么合并出的结果

通过gitk查看，发现可以看见完整的child的提交记录！





如果加了--squash，则只有parent自己的提交记录！把child的记录合并到了parent里面

可以看见。点亮的这一条记录，明明是child更新的。但是记录却是parent提交！

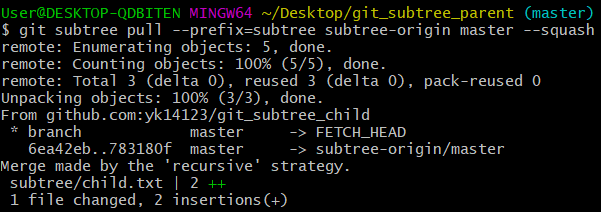
不管child有几个提交。都会被合并成1个，并且这1个再和parent合并！

如果不关心child提交历史，加上squash，可以避免污染主分支

如果最开始add的时候，没有加--squash，那么以后都不要加，避免合并冲突！！！

如果最开始add的时候，加了--squash，那么以后都要加，避免合并冲突！！！

### 从child项目更新依赖



git subtree pull –prefix=<主项目新建文件夹名> <子项目远程地址> <子项目远程分支> --squash

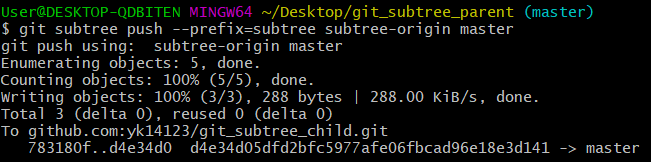
主项目新建文件夹名：把子项目放在主项目下的哪个位置，哪个文件夹的名称！文件夹可以提前存在

子项目远程地址：可以用别名代替

子项目远程分支：master

--squash：没有子项目的提交历史，log被压缩合并了。git log只能看见主项目的提交历史。详见：squash

### 从parent项目推送依赖



git subtree push --prefix=<来自本地child的文件夹名称> <远程child地址> <远程child分支>

1. 首先修改child文件后，先git add . git commit –m “xx” git push到自己的parent远程
2. 再push到child进行修改

### 从parent项目中提取生成child项目

git subtree split

## 常用命令

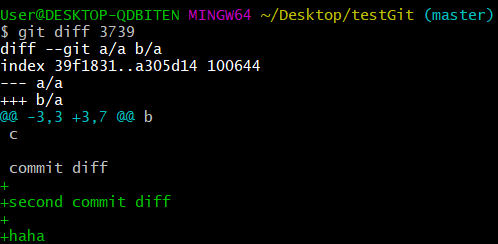
初始化空的git仓库



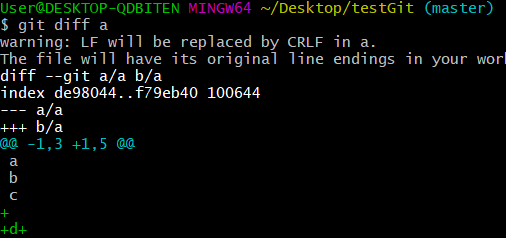
### git diff

和某一个版本的文件做比较，输入commitId

版本库里面的内容，做了如下修改后，能够变成工作区内容！！



#### 工作区与暂存区比较

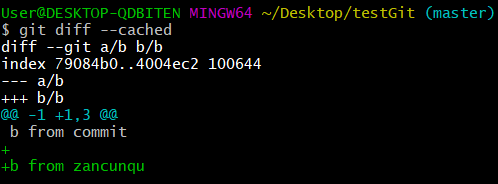


git diff <文件名>

直接就是比较工作区和暂存区的diff

暂存区里面的内容，做了如下修改后，能够变成工作区内容！！

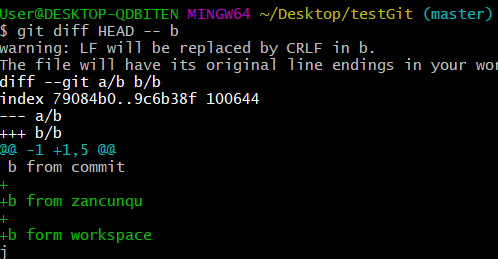
#### 暂存区与版本库比较



git diff --cached 比较的是暂存区和最新的版本库的比较

版本库里面的内容，做了如下修改后，能够变成暂存区内容！！

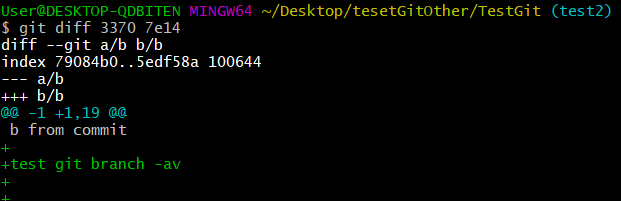
#### 工作区与版本库比较



git diff HEAD -- <文件名>

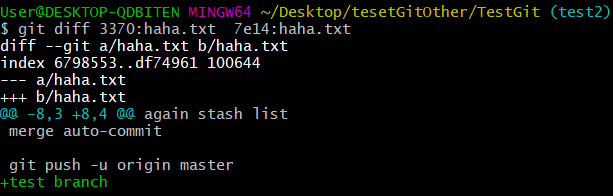
版本库里面的内容，做了如下修改后，能够变成工作区内容！！

#### 版本库与版本库比较



git diff commitId commitId

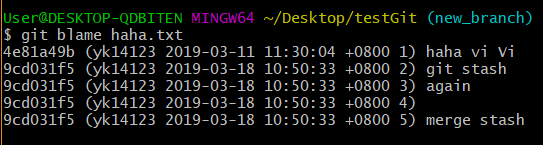
比较2个版本库的所有文件差异



git diff commitId:filename commitId:filename

比较2个版本库的某个文件

### git blame



查看某个文件在某个分支的所有操作记录

### 标签管理

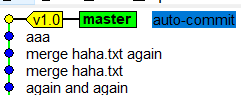
标签是一个静态的commitId的标识。可以看成commitId的别名。用以发布新版本

#### 创建标签

##### 简易标签

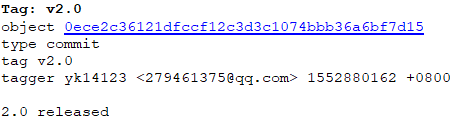


打上标签后，gitk查看，变成黄色图案

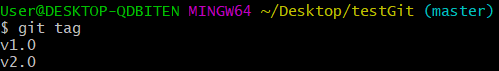


* 命令git tag <tagname>用于新建一个标签，默认为HEAD，也可以指定一个commit id；

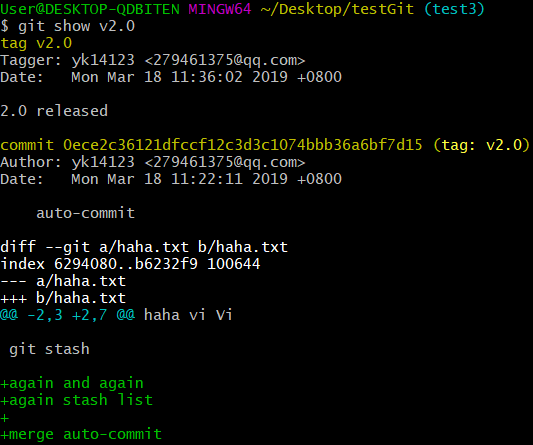
##### 带注释信息的tag

* 
* 
* 命令git tag -a <tagname> -m "blablabla..."可以指定标签信息；

##### 查看标签



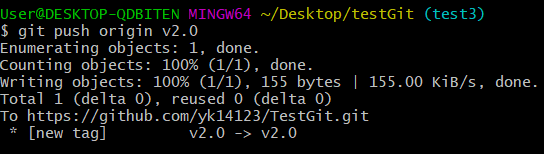
* 命令git tag可以查看所有标签。



**git show <tag名>**

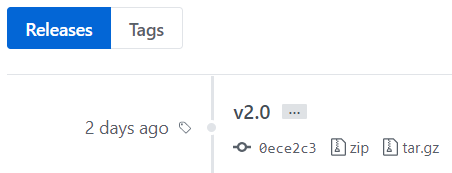
#### 操作标签

##### 推送标签



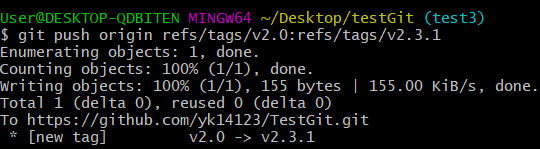
git push origin <标签名称>

对于git来说。标签就代表着release！！



* 命令git push origin <tagname>可以推送一个本地标签；
* 
* 命令git push origin --tags可以推送全部未推送过的本地标签；

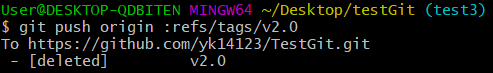
###### 完整命令：



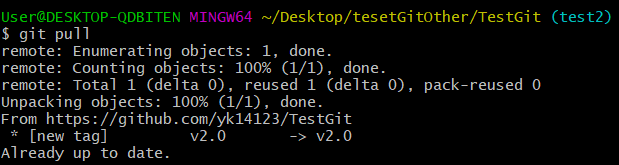
git push origin refs/tags/本地标签名:refs/tags/远程标签名

##### 删除标签



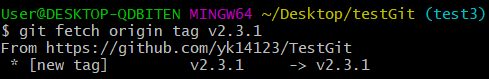
* 命令git tag -d <tagname>可以删除一个本地标签；
* 
* 命令git push origin :refs/tags/<tagname>可以删除一个远程标签。
* 
* git push origin --delete tag <tag名>也可以删除一个远程标签

##### 下载标签



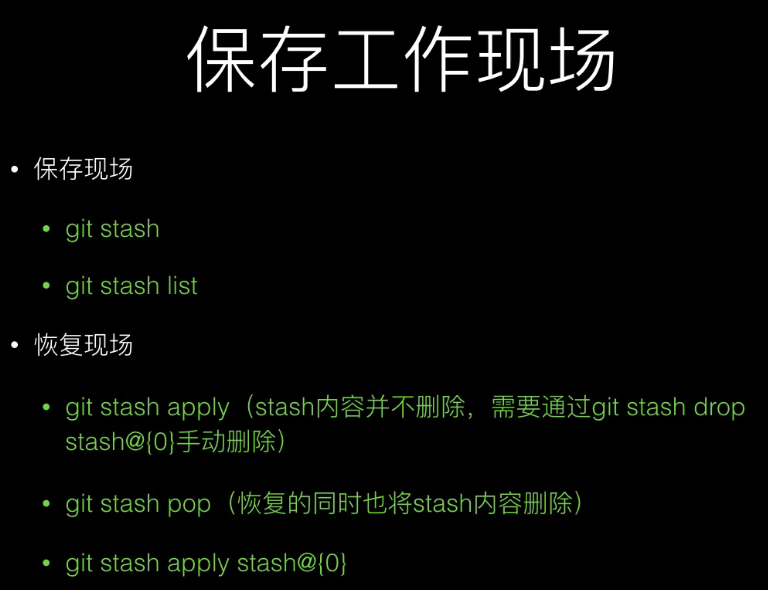
直接git pull，会下载所有的标签

###### 下载指定标签



git fetch origin tag <指定标签名>

### git stash



Git还提供了一个stash功能，可以把当前工作区现场“储藏”起来，等以后恢复现场后继续工作：

$ git stash

Saved working directory **and** index state WIP on dev: f52c633 add merge

工作区是干净的，刚才的工作现场存到哪去了？用git stash list命令看看：

$ git stash list

stash@{0}: WIP on dev: f52c633 add merge

工作现场还在，Git把stash内容存在某个地方了，但是需要恢复一下，有两个办法：

一是用git stash apply恢复，但是恢复后，stash内容并不删除，你需要用git stash drop来删除；

另一种方式是用git stash pop，恢复的同时把stash内容也删了：

$ git stash pop

On branch dev

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: hello.py

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: readme.txt

Dropped refs/stash@{0} (5d677e2ee266f39ea296182fb2354265b91b3b2a)

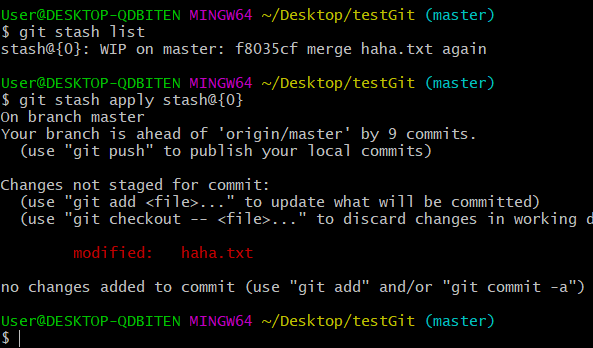
再用git stash list查看，就看不到任何stash内容了：

$ git stash list

你可以多次stash，恢复的时候，先用git stash list查看，然后恢复指定的stash，用命令：

$ git stash apply stash@{0}

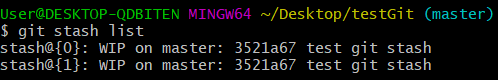
#### 指定恢复stash

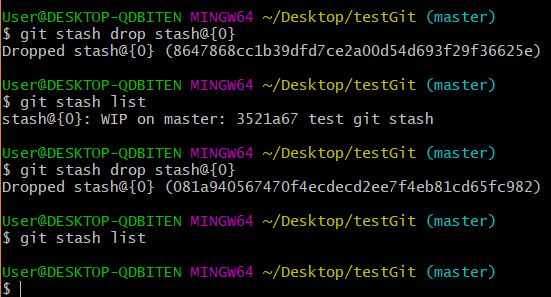


git stash apply stash@{0}指定恢复0位置的stash

数字越大，表示越旧

#### 删除stash





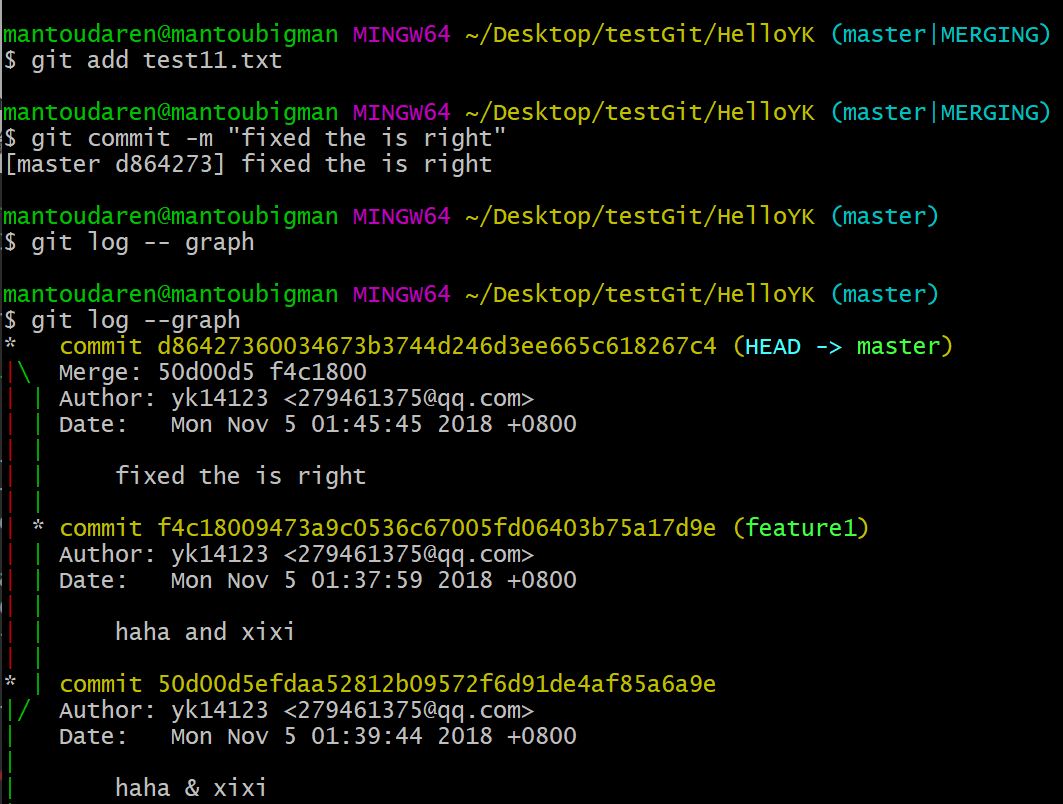
stash删除是需要一条一条的删除

git stash drop stash@{0}

### git log

查看log日志

#### Git log -graph

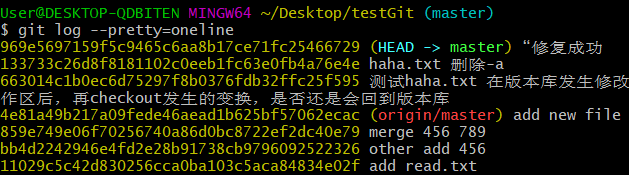


#### git –n

git -3

只显示3条log

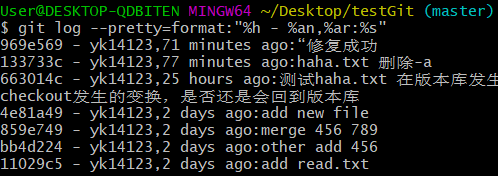
#### git log –pretty=oneline



单行显示commitID和commit注释

#### git log –pretty=format:”%h - %an,%ar:%s”

定制显示log的格式



### git checkout .

恢复工作区的操作

如果有未提交的操作，任何没有add的操作都将恢复！

这个操作。仅仅是针对工作区的文件内容，会发生变化，会恢复到初始！



### git reset HEAD

恢复暂存区的操作

把已经add的文件，回退到工作区！

这个操作。仅仅是针对暂存区的文件内容，不会发生变化，仅仅是暂存区状态，转为工作区状态！

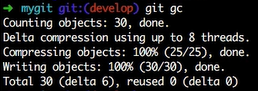


### Git reset --hard

详见本地版本库版本回退

## 特殊命令

### git gc



会把分散到不同地方的文件，集中压缩打包到一个地方。并且建立引用关系

### alis



可以使用br来代替branch

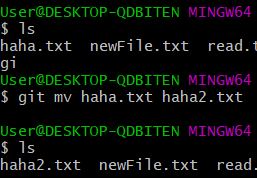
起另外一个别名，来代替原有字符串

### echo



把newFile.txt里面的所有内容，全部替换为 welcome

### git mv



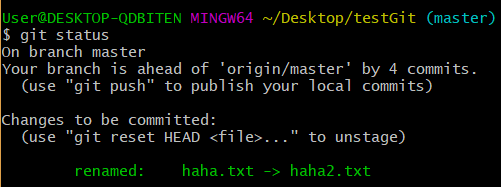
更换文件名称

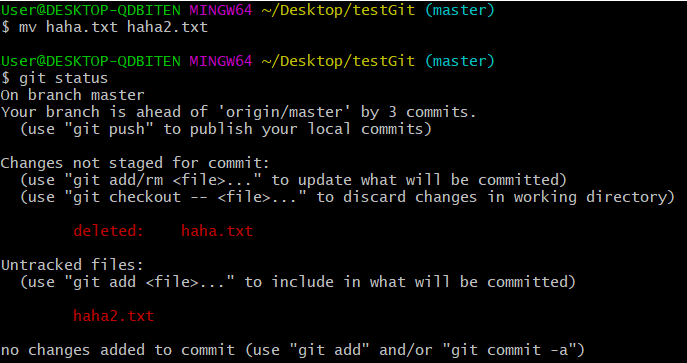
Linux的命令：mv xxx 是移动文件到哪里去

git mv aaa bbb 是把文件名aaa，重命名为bbb

并且加了git 的命令都是从版本库，移动到暂存区！

对于git来说重命名。等同于先删除1个文件，再创建1个内容一样的新文件名，并且还操作了add，非常方便！！





对于Linux命令，mv 重命名，等同于先删除1个文件，再创建1个内容一样的新文件名

## git 忽略

添加 .gitignore ，在文本里面输入需要忽略的文件名如: haha.txt

可以发现，haha.txt已经不能被git识别了。

最后需要把.gitignore add commit到版本库即可

### 支持正则表达

.gitignore文件内容如下：

\*.b // 以.b结尾的都忽略

!a.b //以.b结尾的都忽略，除了a.b文件不忽略！切记，git执行顺序是从上到下，所以上下两行代码顺序不能调换

#表示这一行注释掉，git不扫描

这个文件夹下面什么内容都没有，为空的话，git自动忽略。不需要配置！

/mydir 忽略和.gitignore平级目录的mydir文件夹，但是不忽略二级目录文件夹/todo/mydir

mydir/ 忽略所有mydir文件夹，不管是否在根目录，都忽略mydir下的所有文件

mydir/\*.txt 忽略mydir/aa.txt 但不包含mydir/server/bb.txt不忽略二级目录

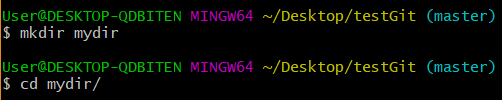
mydir/\*\*/\*.txt 就是忽略所有mydir，包含子目录下的所有以txt结尾的文件

/\*/text.txt 忽略二级目录下的text.txt文件

/\*\*/text.txt 忽略所有目录下的text.txt文件

### 目录操作

#### 创建目录



mkdir xxxx

#### 删除目录



rm -rf xxx/

因为目录里面有可能有子文件，子文件夹。所以-rf是作用就是循环遍历删除

rmdir xxx 删除一个目录，但是目录里面不能有文件