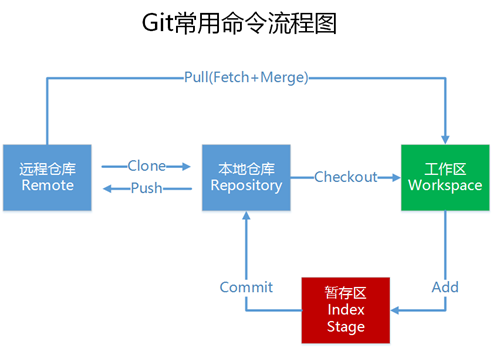
# Git的简介

## 1.1 Git的介绍

Git是一个开源的分布式版本控制系统，可以有效、高速地处理从很小到非常大的项目版本管理。[1]  Git 是 Linus Torvalds 为了帮助管理 Linux 内核开发而开发的一个开放源码的版本控制软件



## Git的特点

### git功能特性

* 从服务器上克隆完整的Git仓库（包括代码和版本信息）到单机上
* 在自己的机器上根据不同的开发目的，创建分支，修改代码，合并分支
* 在单机上自己创建的分支上提交代码
* 把服务器上最新版的代码fetch下来，然后跟自己的主分支合并
* 根据需要把指定提交代码生成补丁（patch），把补丁发送给协同开发者
* 应用补丁，合并代码（包含提交的信息）
* 向公共服务器提交结果，然后通知所有开发人，所有开发人从服务器上pull代码到本地仓库，log会看到所有提交信息

### 1.2.2 git的优缺点

优点：

适合[分布式开发](https://baike.baidu.com/item/%E5%88%86%E5%B8%83%E5%BC%8F%E5%BC%80%E5%8F%91" \t "_blank)，强调个体。

公共服务器压力和数据量都不会太大。

速度快、灵活。

任意两个开发者之间可以很容易的解决冲突。

离线工作。

缺点：

资料少（起码中文资料很少）。

学习周期相对而言比较长。

不符合常规思维。

代码保密性差，一旦开发者把整个库克隆下来就可以完全公开所有代码和版本信息。

# 2 .Git和svn的对比

SVN的优点：

1、采用集中式，易于管理，保证安全性；

2、管理方便，逻辑明确，理念符合常规思维；

3、代码的一致性高；

4、适合人数不多的项目开发；

5、允许一个文件有任意多的可命名属性，会关注所有的文件类型；

6、支持二进制文件，更容易处理大文件；

7、支持空目录。

SVN的缺点：

1、服务器压力太大，数据库容量暴增；

2、必须连接在服务器上，否则基本不能工作、提交、对比、还原等；

3、不适合开源开发。

Git的优点：

1、适合分布式开发，强调个体；

2、公共的服务器压力和数量都不会太大；

3、速度快， 成熟的架构，开发灵活；

4、任意两个开发者之间可以很容易的解决冲突；

5、离线工作，管理代码成本低，不需要依赖服务器；

6、部署方便。基本上下个命令就可以用；

7、良好的分支机制，可以让主干代码保持干净。

Git的缺点：

1、资料少，学习成本比较大，学习周期比较长，要求人员素质比较高；

2、不符合常规思维；

3、代码保密性差，一旦开发者把整个库克隆下来就可以完全公开所有代码和版本信息。

# 3 .Git的协同开发

## 3.1 git分支管理

每个开发者都拥有自己的版本控制库，在自己的版本库上可以任意的执行提交代码、创建分支等行为。例如，开发者认为自己提交的代码有问题？没关系，因为版本库是自己的，回滚历史、反复提交、归并分支并不会影响到其他开发者。

实例：

1. 开发某项目。
2. 为实现某个新的需求，创建一个分支iss53。
3. 在这个分支上开展工作。

　　　假设临时有个很严重的其他问题需要紧急修补，按照下面的方式处理：

1. 返回master分支。
2. 为这次紧急修补建立一个新分支hotfix，并在其中修复问题。
3. 通过测试后，回到服务器所在的master分支，将修补分支合并进来，然后再推送到服务器上。
4. 删除hotfix
5. 切换到之前实现新需求的分支iss53，继续工作。

　　　工作完成以后，将分支iss53合并入master

　　如果iss53与hotfix工作没有冲突，则合并成功，如果有冲突则需要再进行处理判断后，再合并。

## **3.2** git在实际项目中的应用

这是Git版本控制的分支管理总体图：



 其中最重要的是**主分支**：

* master：用来版本发布，主要反映当前线上运行的代码。
* develop：作为开发分支，用来存放最新开发的代码，有时候会叫它“集成分支”。

develop分支用来存放开发完成的代码，当develop分支上的代码测试没有问题后，merge到master分支上，并发布。



**辅助分支**：

* **feature branches**

feature分支也可以叫做主题分支。例如在新版中要开发的新功能，可以用feature来存放。名字可以另外取，例如要增加insert和search功能，可以增加insertFeature和searcheFeature的分支，开发好后，再分别merge到develop分支中。



　　其中--no-ff 的参数，ff表示fastforward，--no--ff，就表示会对历史的分支做一个新的提交对象。加参数和不加参数的区别如下：（可以使用 sourceTree 或者命令git log --graph查看）  


* **release branches**

这个分支可以是来自develop；必须合并到develop和master。release分支可以理解为更多的是作作为版本控制、缺陷镜像修复和最后发布前的复查工作。

为版**hotfix branches**

这个是一个需要紧急修复bug的分支。当线上产品有bug，我们可以开一个hotfix分支，修复完bug后，再merge，删除。

它来自master，融合向develop和master。

关系如下：



# 4．建立版本库

4.1建立一个文件夹作为git的版本库

4.2创建一个身份：

git config –global user.name ‘guochunbiao’

git config –global user.email ‘guochunbiao00416@163.com’

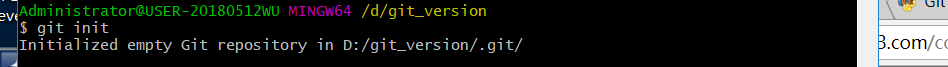
查看身份象征:

git config user.name

git config user.email

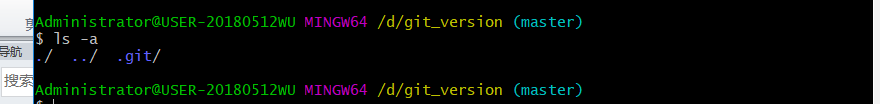
创建版本库:

git init

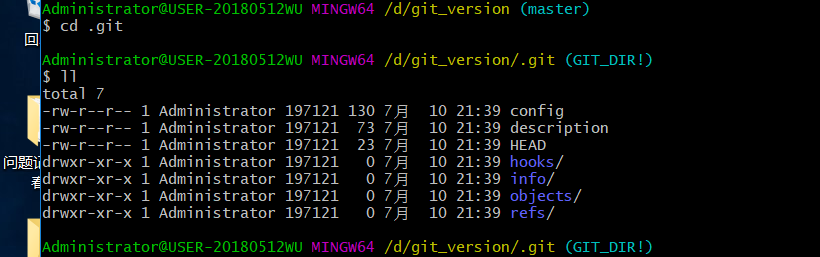


注意.git 是个隐藏文件夹

ls –a 显示所有文件，包括隐藏文件

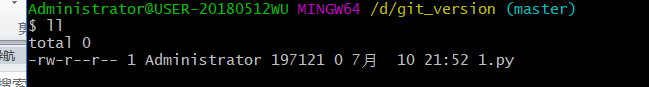


查看版本管理文件



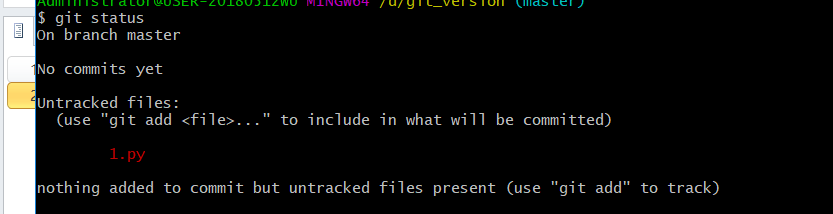
创建一个文件

touch 1.py



git status

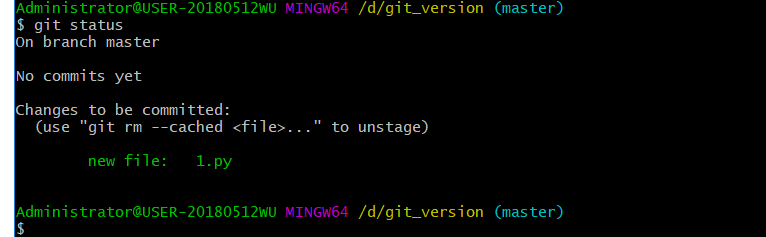
查看git当中的内容



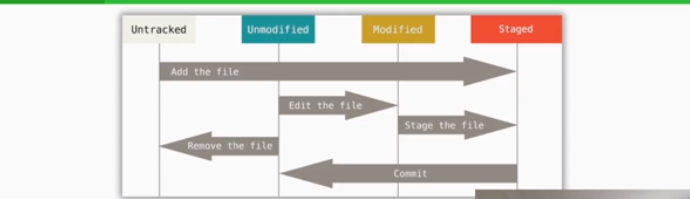
文件还未添加到版本库当中

添加到版本库中

git add 1.py



已经被添加到版本库当中

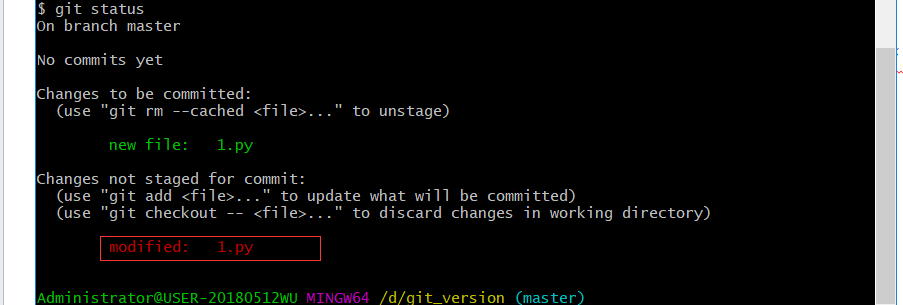


# 5.记录修改

对1.py文件修改‘a=1‘并保存

查看 git status

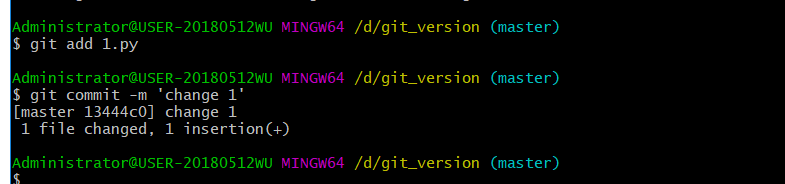
git status –s(缩写形式查看)



如图所示文件变成的修改状态的文件（文件包含3种状态：（untracked，git还不知道有这个文件）modify，unmodify，stage）

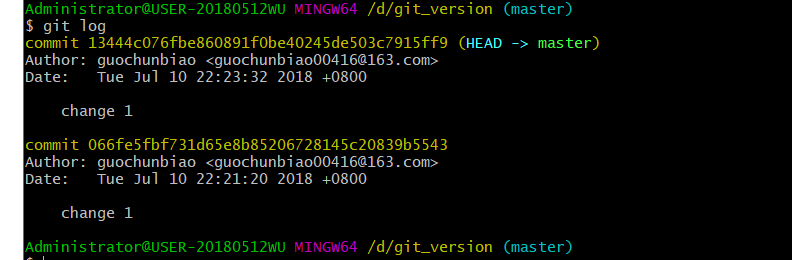
git add 1.py-🡪变成stage 可提交的状态s

git commit –m ‘change 1’（更改信息）

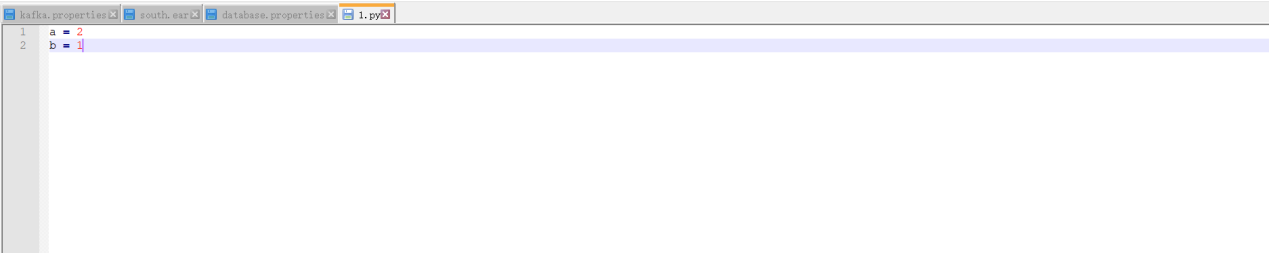


查看记录：

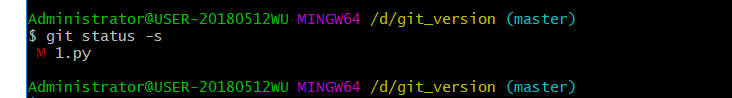
git log



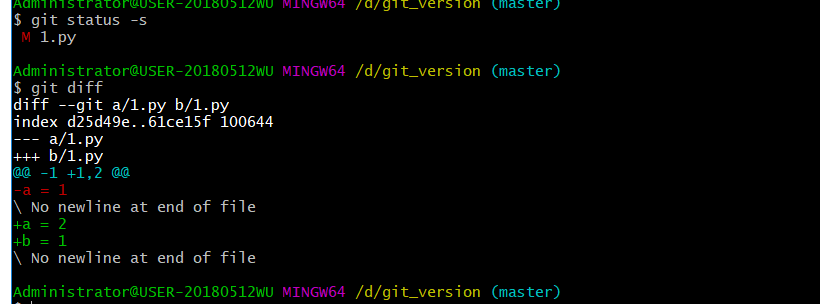
第二次修改保存



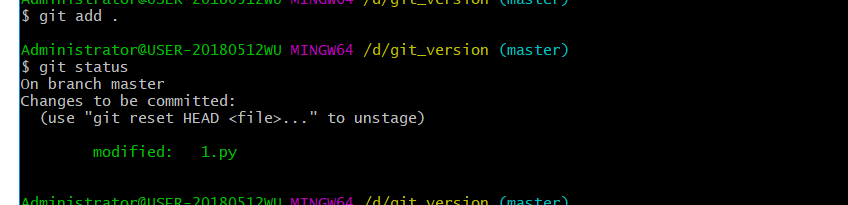
查看git status –s



git diff 对比 onstage 和commit的不同



git add .所有修改变成stage状态

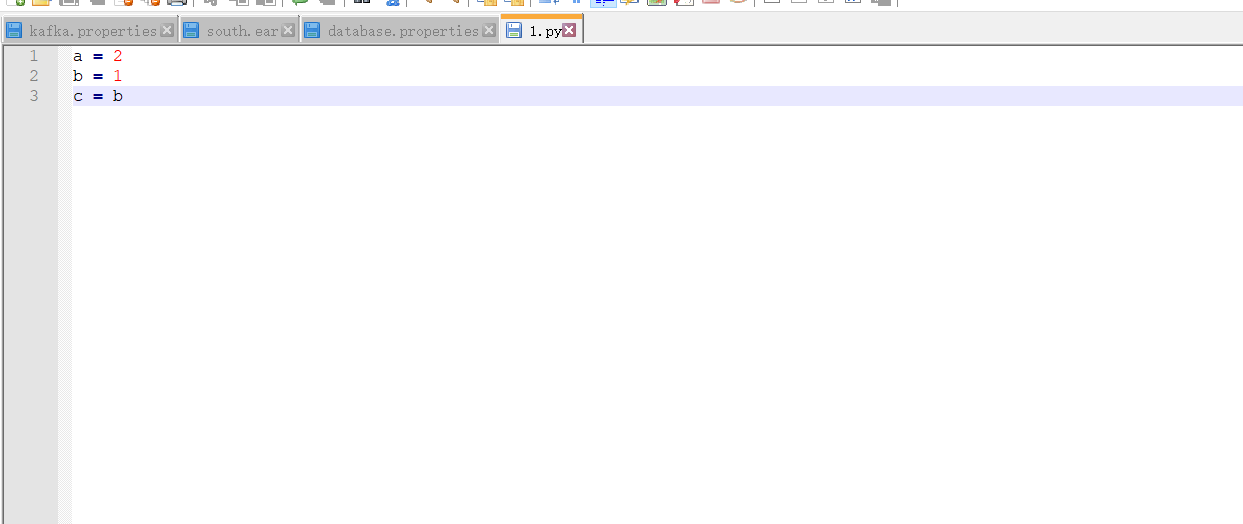


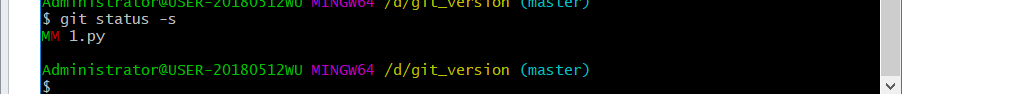
git diff –cached stage 和commit查看对比



同时观看 stage和onstage状态

修改

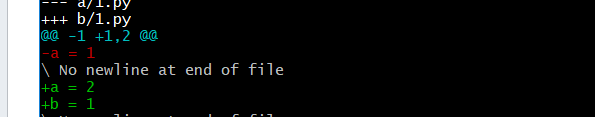




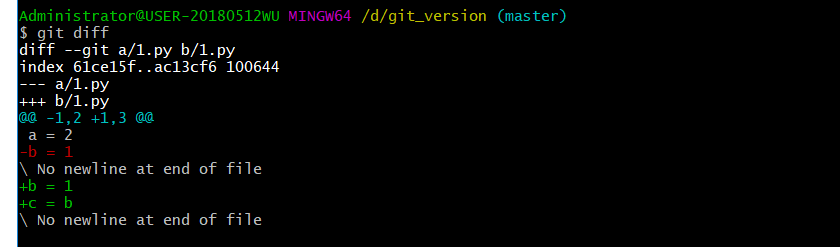
git diff HEAD(包含两种状态)



git diff –cached(只有stage状态)



git diff



# 5．回到从前

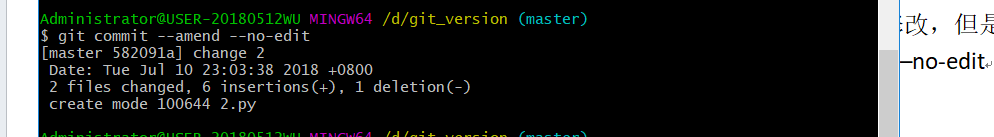
修改记录回到从前

查看git log –online



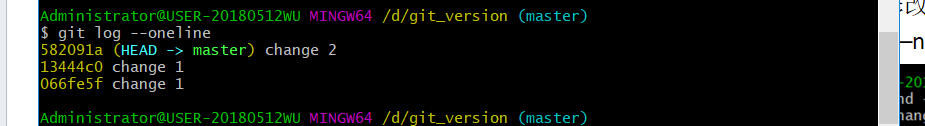
git add 2.py 忘记修改，但是还放到上一个变化当中

git commit –amend –no-edit还放到修改二的当中

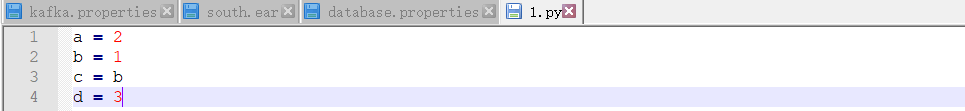


再查看日志

git log –oneline

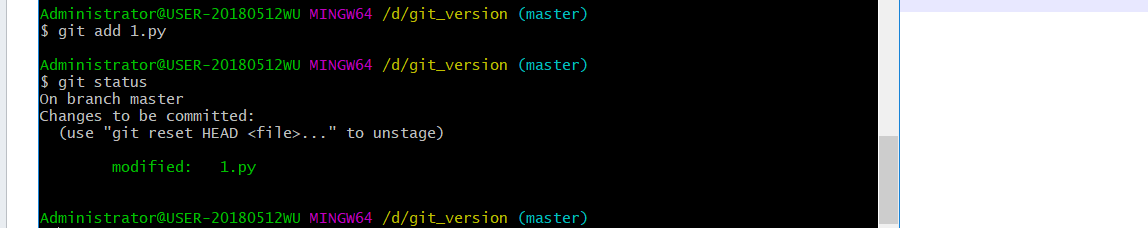


文件部分忘记修改



git add 1.py

git status

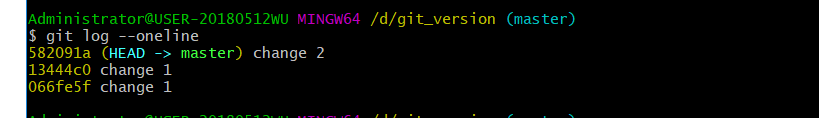


git reset 1.py回到onstage状体



已经提交过

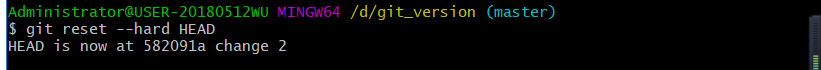
先查看版本



变到stage状态



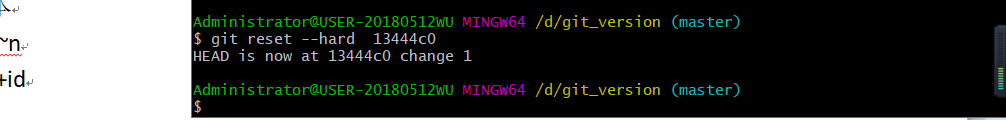
git reset –hard HEAD 回到change 2当中



同理回到第几个版本

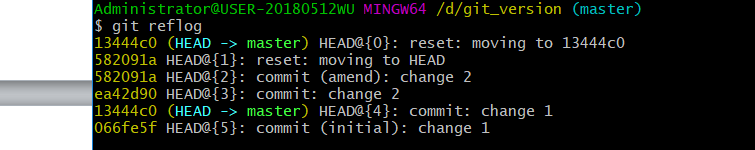
git reset –hard HEAD~n

或 git reset –hard +id

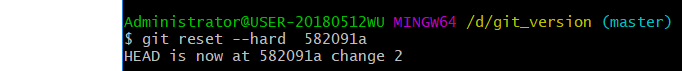


回到将来

git reflog

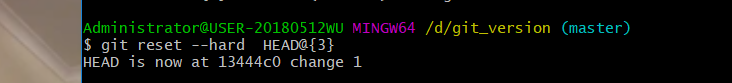
回到change 2

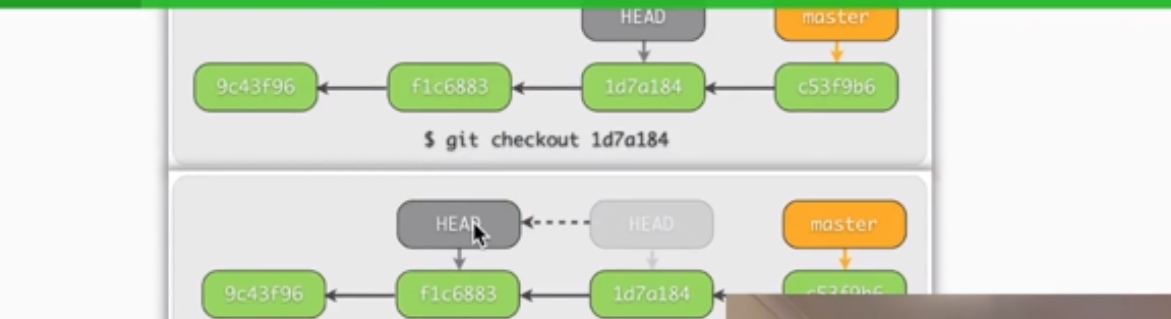
git reset –hard +id



或

git reset --hard HEAD@{3}





# 6.单个文件回到从前chectout

git checkout ac8102b -- 3.py

# 7.分支

* 首先，我们创建dev分支，然后切换到dev分支：
* 图像查看log记录 get log --oneline –graph
* 第一种创建方式：git branch dev 创建分支

把指针放到分支上去：git chectout dev2

删除之前先切换到master ：git checkout master

删除分支git branch -d dev

* 第二种创建方式
* $ git checkout -b dev

Switched to a new branch 'dev'

* + 1
  + 2

git checkout命令加上-b参数表示创建并切换，相当于以下两条命令：

$ git branch dev

$ git checkout dev

Switched to a new branch 'dev'

* + 1
  + 2
  + 3

然后用 git branch 查看当前分支：

$ git branch

\* dev

master

* + 1
  + 2
  + 3
* 然后，我们就可以在dev分支上正常提交，比如对README.md做个修改，加上一行：

Creating a new branch in git is so quick.

* + 1

提交：

git add README.md

git commit -m "branch test"

git commit –am “change4 in dev2”

* + 1
  + 2
* 此时，dev分支的工作完成，我们就可切换回 master 分支：

$ git checkout master

* + 1

切换回master分支后，再查看一个README.md文件，刚才添加的内容不见了！因为那个提交是在dev分支上，而master分支此刻的提交点并没有变。

* 现在，我们把dev分支的工作成果合并到master分支上：

$ git merge dev

git merge --no-ff –m “keep merge info” dev(参数大意：留下你的信息)



* + 1
* 删除分支

$ git branch -d dev

# 8.Merge

git merge nrtest(分支名称)

有冲突的话需要手动修改

停止merge: git merge --abort

# 9 rebase和merge的对比

**git merge**是用来合并两个分支的。

git merge b

      # 将b分支合并到当前分支

同样 git rebase b，也是把 b分支合并到当前分支

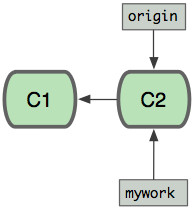
-----------------------------------

他们的 **原理 如下**：

假设你现在基于远程分支"origin"，创建一个叫"mywork"的分支。

$ git checkout -b mywork origin

假设远程分支"origin"已经有了2个提交，如图



现在我们在这个分支做一些修改，然后生成两个提交(commit).

$ vi file.txt

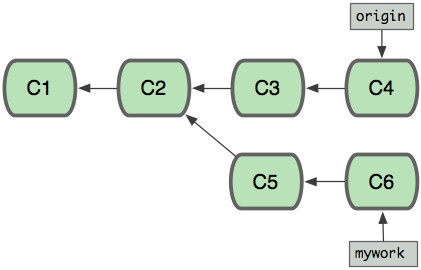
$ git commit

$ vi otherfile.txt

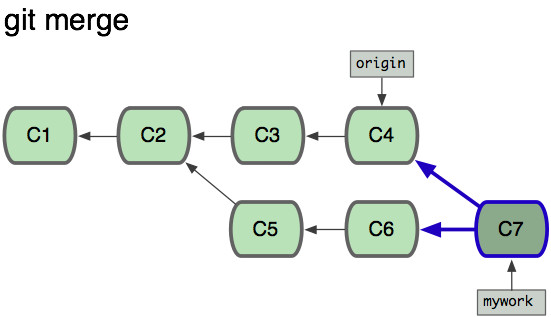
$ git commit

...

但是与此同时，有些人也在"origin"分支上做了一些修改并且做了提交了. 这就意味着"origin"和"mywork"这两个分支各自"前进"了，它们之间"分叉"了。



在这里，你可以用"pull"命令把"origin"分支上的修改拉下来并且和你的修改合并； 结果看起来就像一个新的"合并的提交"(merge commit):

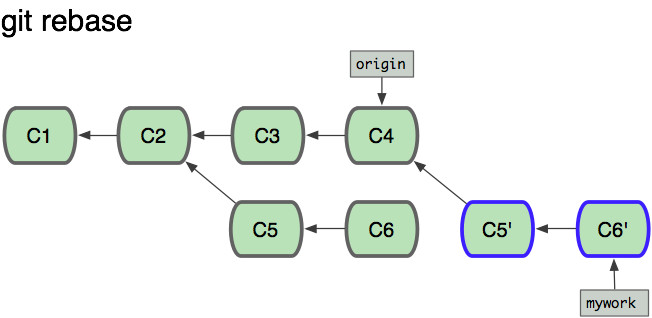


但是，如果你想让"mywork"分支历史看起来像没有经过任何合并一样，你也许可以用 git rebase:

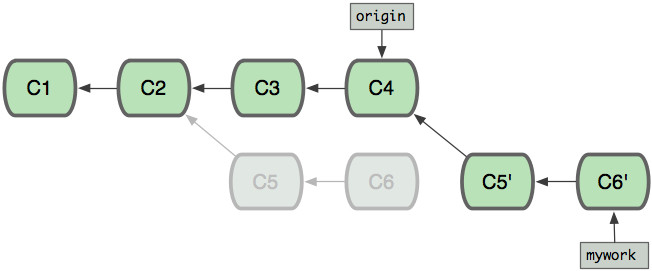
$ git checkout mywork

$ git rebase origin

这些命令会把你的"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 为最新的"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)



二、解决冲突

在rebase的过程中，也许会出现冲突(conflict). 在这种情况，Git会停止rebase并会让你去解决 冲突；在解决完冲突后，用"git-add"命令去更新这些内容的索引(index), 然后，你无需执行 git-commit,只要执行:

$ git rebase --continue

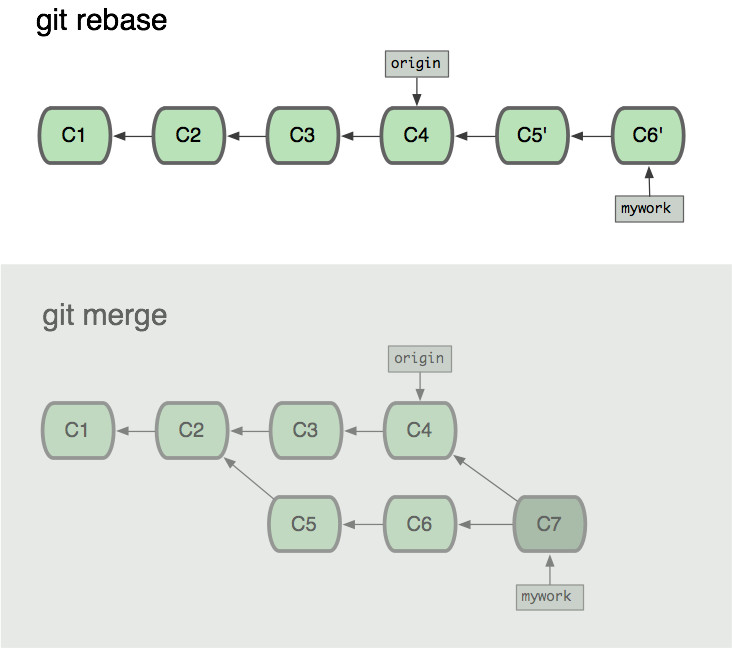
这样git会继续应用(apply)余下的补丁。

在任何时候，你可以用--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。

$ git rebase --abort

三、git rebase和git merge的区别

现在我们可以看一下用合并(merge)和用rebase所产生的历史的区别：



当我们使用Git log来参看commit时，其commit的顺序也有所不同。

假设C3提交于9:00AM,C5提交于10:00AM,C4提交于11:00AM，C6提交于12:00AM,

对于使用git merge来合并所看到的commit的顺序（从新到旧）是：C7 ,C6,C4,C5,C3,C2,C1

对于使用git rebase来合并所看到的commit的顺序（从新到旧）是：C7 ,C6‘,C5',C4,C3,C2,C1

 因为C6'提交只是C6提交的克隆，C5'提交只是C5提交的克隆，

从用户的角度看使用git rebase来合并后所看到的commit的顺序（从新到旧）是：C7 ,C6,C5,C4,C3,C2,C1

# 10 git的分支管理

## 10.1 如何建立与合并分支

### 10.1.1分支的新建与合并指令

新建分支 newBranch，并进入新分支：

|  |  |
| --- | --- |
| 1 | $ git checkout -b newBranch |

相当于：

$ git branch newBranch

$ git checkout newBranch

合并分支 mergeBranch，例如要把mergeBranch，合并入master：

|  |  |
| --- | --- |
| 1  2 | $ git checkout master  $ git merge mergeBranch |

合并后，master和mergeBranch都指向同一个版本，可以删除掉mergeBranch了：

|  |  |
| --- | --- |
| 1 | $ git branch -d mergeBranch |

### 10.1.2　分支创建与合并的实例

#### Task:

1. 开发某项目。
2. 为实现某个新的需求，创建一个分支iss53。
3. 在这个分支上开展工作。

　　　假设临时有个很严重的其他问题需要紧急修补，按照下面的方式处理：

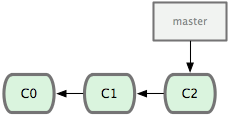
1. 返回master分支。
2. 为这次紧急修补建立一个新分支hotfix，并在其中修复问题。
3. 通过测试后，回到服务器所在的master分支，将修补分支合并进来，然后再推送到服务器上。
4. 删除hotfix
5. 切换到之前实现新需求的分支iss53，继续工作。

　　　工作完成以后，将分支iss53合并入master

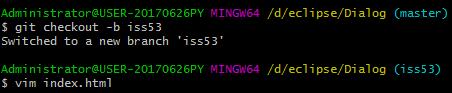
　　如果iss53与hotfix工作没有冲突，则合并成功，如果有冲突则需要再进行处理判断后，再合并。

#### Do:

1、假设当前，master更新为C2版本：

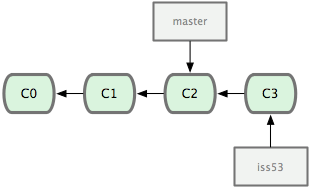


2、为实现某功能创建iss53分支，创建index.html，并上传

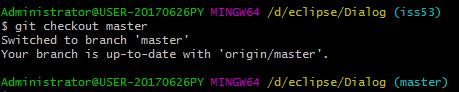




这个时候，状态是这样的：



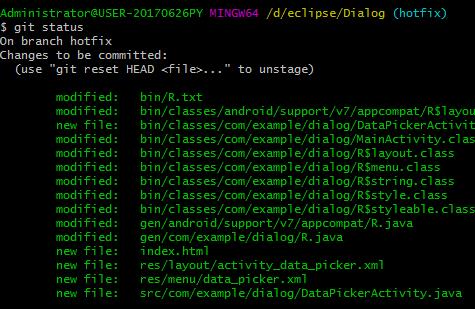
3、紧急任务来了，要先返回master分支

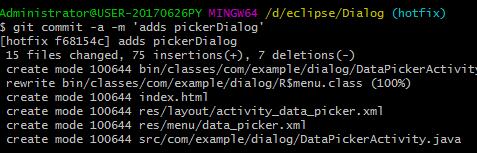


4、现在接到进行紧急修改的要求，然后创建hotfix

https://images2017.cnblogs.com/blog/828709/201712/828709-20171206203412003-1599171270.jpg

5、进行了相应的开发，包括修改index.html(注意，在iss53中也进行了修改)

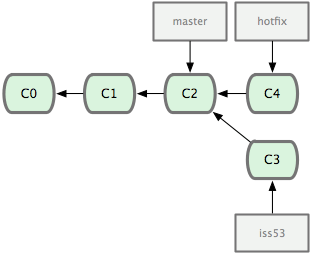




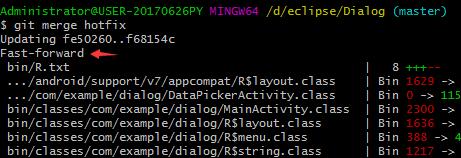
然后git push origin hotfix。

Git 会把工作目录的内容恢复为检出某分支时它所指向的那个提交对象的快照。它会自动添加、删除和修改文件以确保目录的内容和你当时提交时完全一样。

于是当前的状态是：

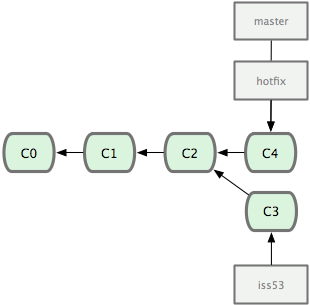


6、当分支hotfix的内容测试通过确认后，就可以合并到master中，并删除hotfix



如果顺着一个分支走下去可以到达另一个分支的话，那么 Git 在合并两者时，只会简单地把指针右移，因为这种单线的历史分支不存在任何需要解决的分歧，所以这种合并过程可以称为快进（Fast forward）。

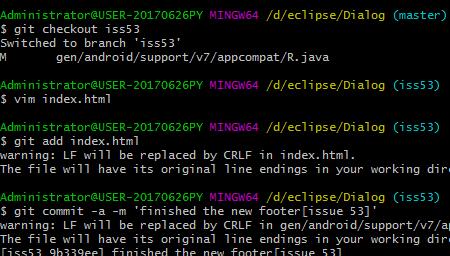
现在最新的修改已经合并到master分支了。可以发布啦。



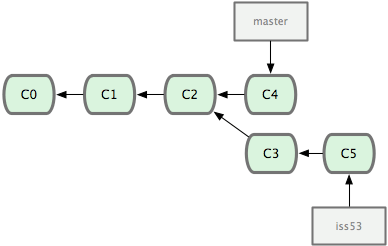
接下来删除hotfix

https://images2017.cnblogs.com/blog/828709/201712/828709-20171206213244753-65797061.jpg

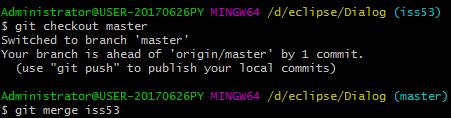
7、现在我们回到iss35继续工作



我们继续推进iss35:

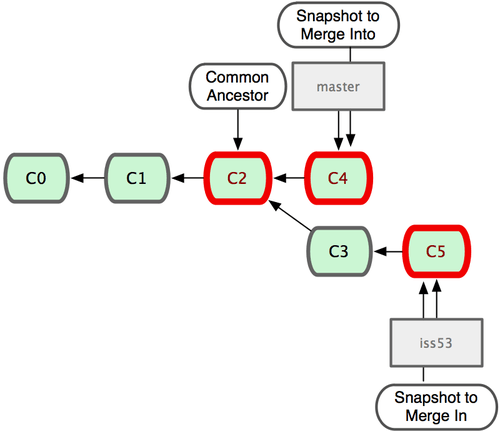


8、现在接下来要合并master和iss53



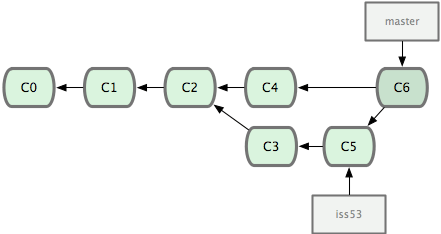
合并后，删除iss53分支，即可。

请注意，这次合并操作的底层实现，并不同于之前 hotfix 的并入方式。因为这次你的开发历史是从更早的地方开始分叉的。由于当前 master 分支所指向的提交对象（C4）并不是 iss53 分支的直接祖先，Git 不得不进行一些额外处理。就此例而言，Git 会用两个分支的末端（C4 和 C5）以及它们的共同祖先（C2）进行一次简单的三方合并计算。

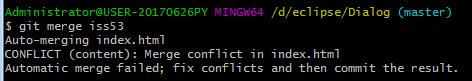


这次，Git 没有简单地把分支指针右移，而是对三方合并后的结果重新做一个新的快照，并自动创建一个指向它的提交对象（C6）（见图 3-17）。这个提交对象比较特殊，它有两个祖先（C4 和 C5）。

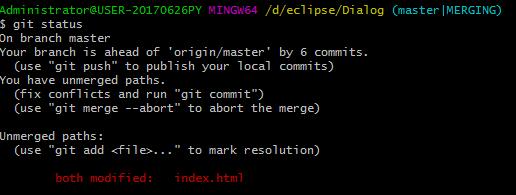
值得一提的是 Git 可以自己裁决哪个共同祖先才是最佳合并基础；这和 CVS 或 Subversion（1.5 以后的版本）不同，它们需要开发者手工指定合并基础。所以此特性让 Git 的合并操作比其他系统都要简单不少。



9、但是如果在不同的分支中都修改了同一个文件的同一部分，Git 就无法干净地把两者合到一起（译注：逻辑上说，这种问题只能由人来裁决。）。如果你在解决问题 #53 的过程中修改了 hotfix 中修改的部分，则会出现：



我们先看一下状态：



接下来我们可以打开 index.html



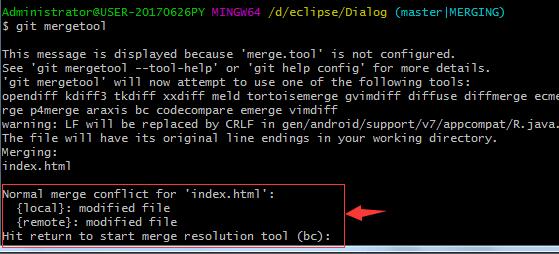
git 已经自动对其做了区分，HEAD表示master分支的内容，=====下面的是iss53分支做的修改

你可以对其再做调整和修改。

再用git add index.html，git commit表示他们已经解决冲突。

然后再融合和删除iss53分支。完成工作。

也可以使用mergetool：



## 10.2. Git分支在实际项目中的灵活运用

这是Git版本控制的分支管理总体图：



 其中最重要的是**主分支**：

* master：用来版本发布，主要反映当前线上运行的代码。
* develop：作为开发分支，用来存放最新开发的代码，有时候会叫它“集成分支”。

develop分支用来存放开发完成的代码，当develop分支上的代码测试没有问题后，merge到master分支上，并发布。



**辅助分支**：

* **feature branches**

feature分支也可以叫做主题分支。例如在新版中要开发的新功能，可以用feature来存放。名字可以另外取，例如要增加insert和search功能，可以增加insertFeature和searcheFeature的分支，开发好后，再分别merge到develop分支中。



### 从 develop 分支建一个 feature 分支，并切换到 feature 分支

|  |  |
| --- | --- |
| 1  2 | $ git checkout -b myfeature develop  Switched to a new branch "myfeature" |

### 合并feature 分支到 develop

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | $ git checkout develop  Switched to branch 'develop'  $ git merge --no-ff myfeature  Updating ea1b82a..05e9557  (Summary of changes)  $ git branch -d myfeature  Deleted branch myfeature (was 05e9557).  $ git push origin develop |

　　其中--no-ff 的参数，ff表示fastforward，--no--ff，就表示会对历史的分支做一个新的提交对象。加参数和不加参数的区别如下：（可以使用 sourceTree 或者命令git log --graph查看）  


* **release branches**

这个分支可以是来自develop；必须合并到develop和master。release分支可以理解为更多的是作为版本控制、缺陷镜像修复和最后发布前的复查工作。

创建一个realase分支

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | $ git checkout -b release-1.2 develop  Switched to a new branch "release-1.2"  $ ./bump-version.sh 1.2  Files modified successfully, version bumped to 1.2.  $ git commit -a -m "Bumped version number to 1.2"  [release-1.2 74d9424] Bumped version number to 1.2  1 files changed, 1 insertions(+), 1 deletions(-) |

　　这里的bump-version.sh ，只是一个虚拟的脚本，是要求读者自己去编写这样的脚本，实现相应的地方都统一更改版本号。

完成release分支：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | $ git checkout master  Switched to branch 'master'  $ git merge --no-ff release-1.2  Merge made by recursive.  (Summary of changes)  $ git tag -a 1.2 |

　　融合到develop分支

|  |  |
| --- | --- |
| 1  2  3  4  5 | $ git checkout develop  Switched to branch 'develop'  $ git merge --no-ff release-1.2  Merge made by recursive.  (Summary of changes) |

　　删除release分支

|  |  |
| --- | --- |
| 1  2 | $ git branch -d release-1.2  Deleted branch release-1.2 (was ff452fe). |

* **hotfix branches**

这个是一个需要紧急修复bug的分支。当线上产品有bug，我们可以开一个hotfix分支，修复完bug后，再merge，删除。

它来自master，融合向develop和master。

关系如下：



具体代码如下：

创建hotfix分支：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | $ git checkout -b hotfix-1.2.1 master  Switched to a new branch "hotfix-1.2.1"  $ ./bump-version.sh 1.2.1  Files modified successfully, version bumped to 1.2.1.  $ git commit -a -m "Bumped version number to 1.2.1"  [hotfix-1.2.1 41e61bb] Bumped version number to 1.2.1  1 files changed, 1 insertions(+), 1 deletions(-) |

|  |  |
| --- | --- |
| 1  2  3 | $ git commit -m "Fixed severe production problem"  [hotfix-1.2.1 abbe5d6] Fixed severe production problem  5 files changed, 32 insertions(+), 17 deletions(-) |

　　完成了hotfix分支：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | $ git checkout master  Switched to branch 'master'  $ git merge --no-ff hotfix-1.2.1  Merge made by recursive.  (Summary of changes)  $ git tag -a 1.2.1 |

　　合并到develop分支

|  |  |
| --- | --- |
| 1  2  3  4  5 | $ git checkout develop  Switched to branch 'develop'  $ git merge --no-ff hotfix-1.2.1  Merge made by recursive.  (Summary of changes) |

　　最后删除hotfix分支

|  |  |
| --- | --- |
| 1  2 | $ git branch -d hotfix-1.2.1  Deleted branch hotfix-1.2.1 (was abbe5d6). |

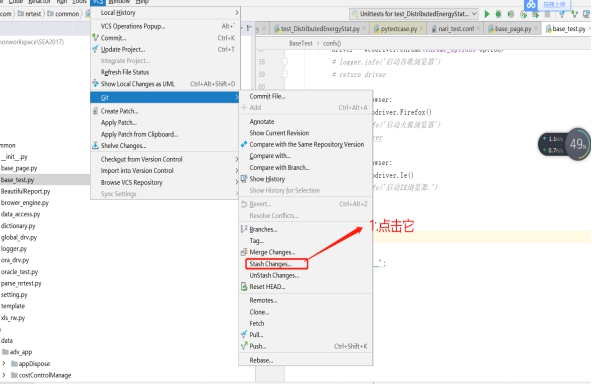
# 11 Pycharm+git 操作流程

## 11.1Git提交步骤

### 11.1 详细步骤

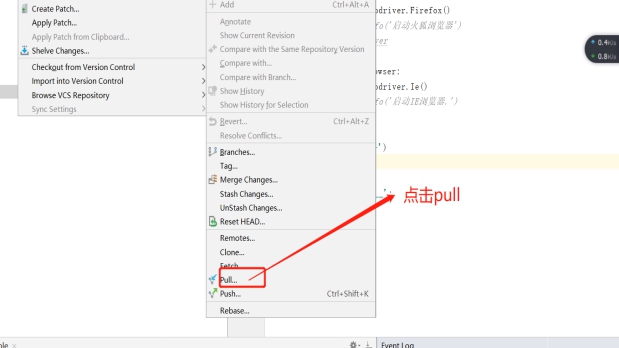
#### 11.1.1 Stash

（保存pycharm当前正在完成的脚本！！！注意是已经添加到本地仓库，但未提交到本地仓库）



#### 11.1.2 Pull

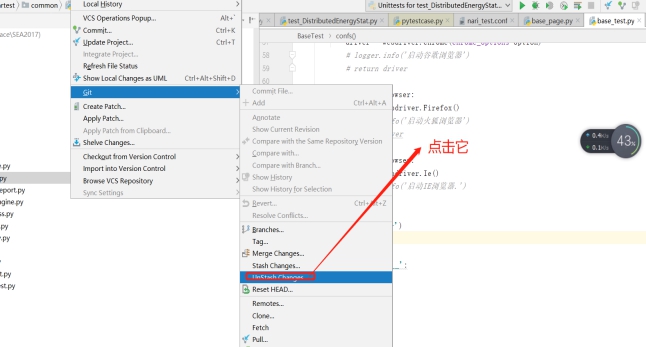
（拉去远程仓库代码，使本地仓库与远程长裤保持一致）

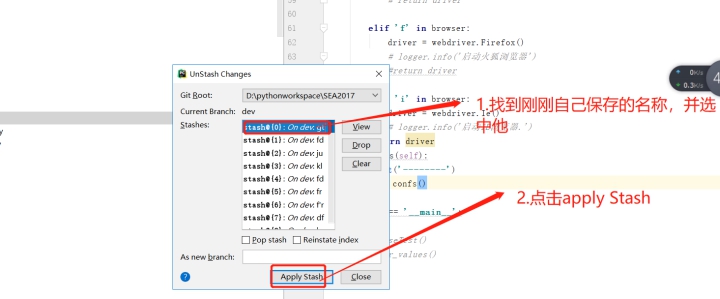


#### 11.1.3 Unstash

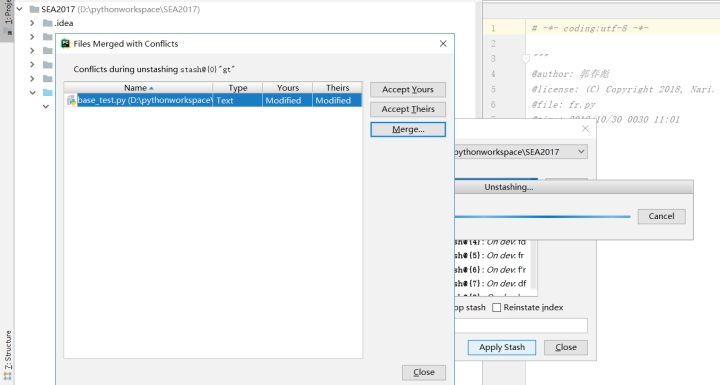
**（恢复自己编写的脚本**）

1 恢复脚本

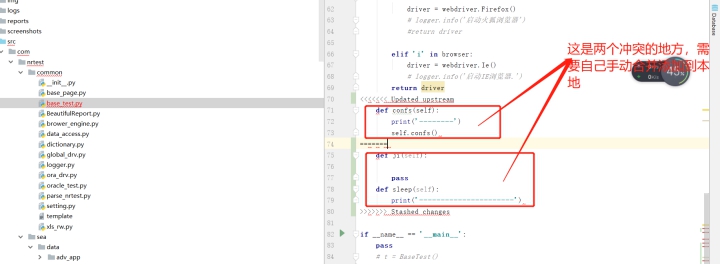




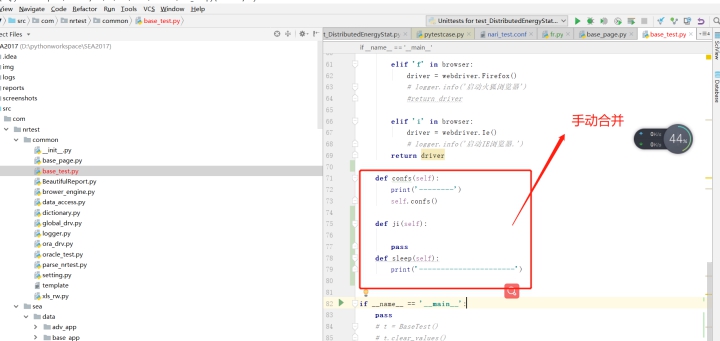
2.假如遇到冲突文件时，如何处理如下图：问题

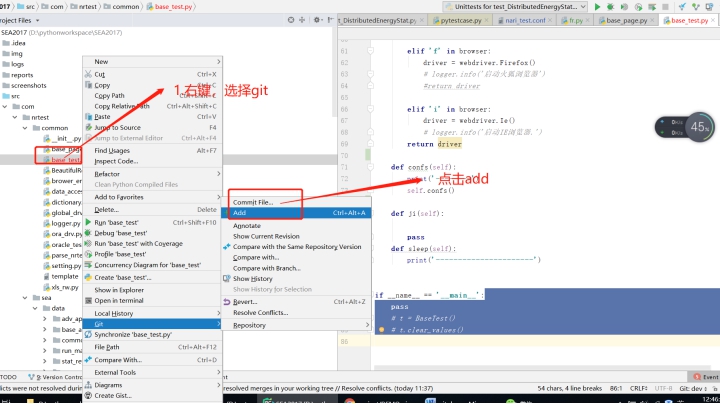


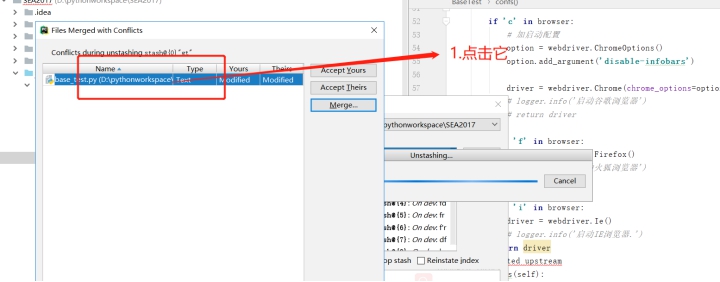
3.如果选择accept their

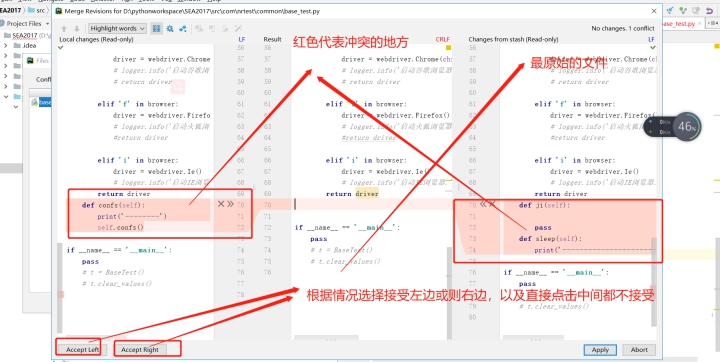


4. 手动合并





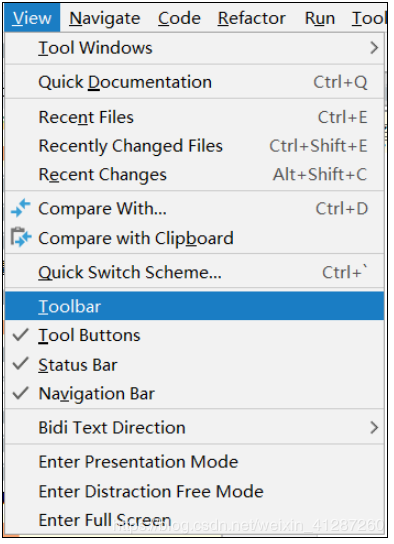


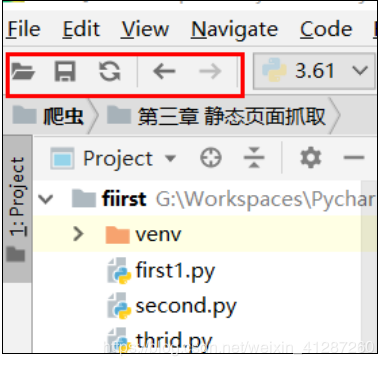


#### 刷新

利用pycharm做项目，有时会有写文件（保存到项目中）的需求，但是因为电脑比较慢，项目目录不会自动更新，可以自行设置：

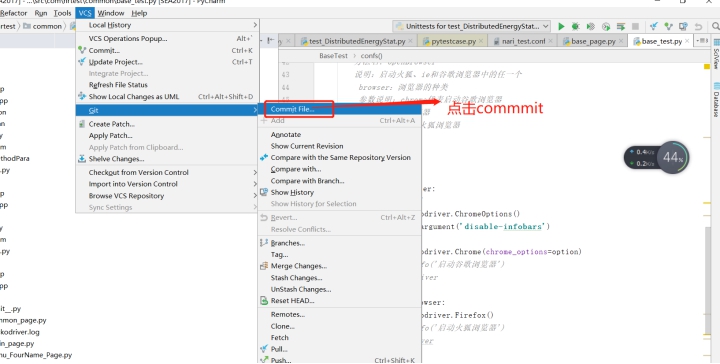
view -->toolbar。

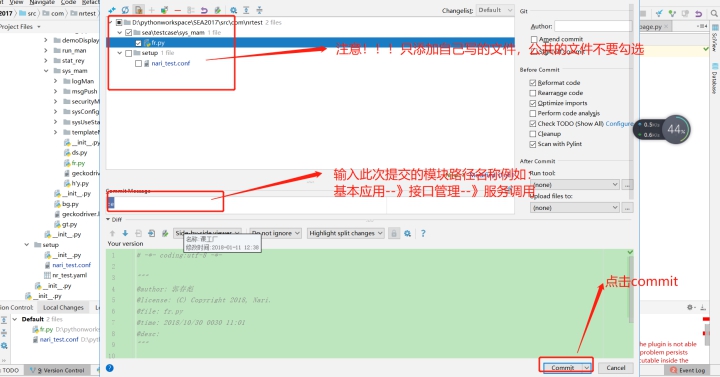




#### 11.1.5 commit提交

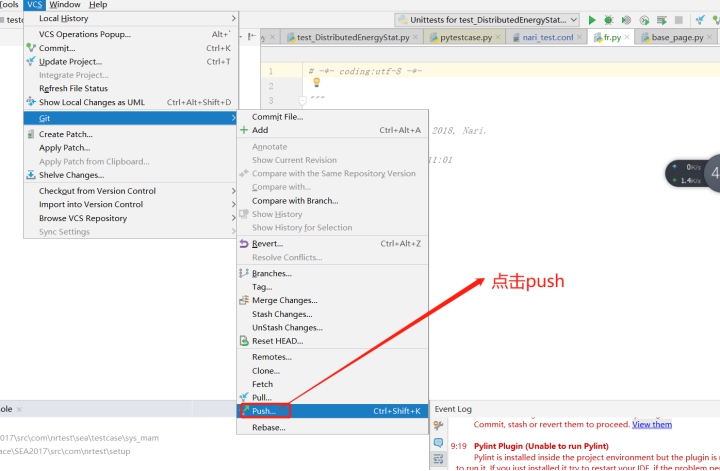
**（提交到本地仓库）**





#### 11.1.6点击push

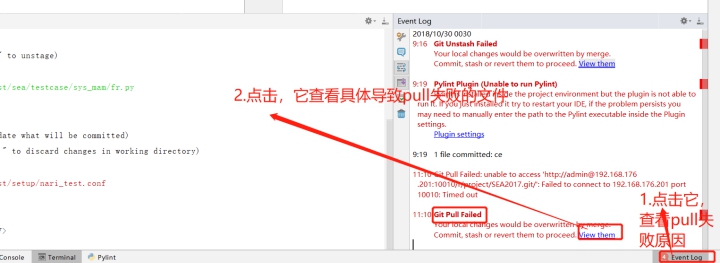
（推送到远程仓库）



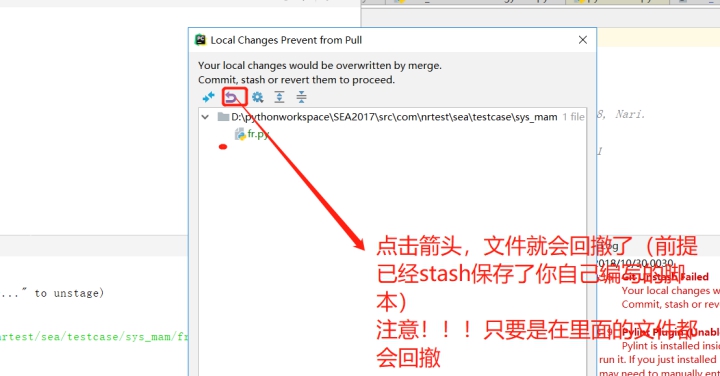
### 11.2. 特殊情况的处理

#### 11.2.1pull失败如何处理

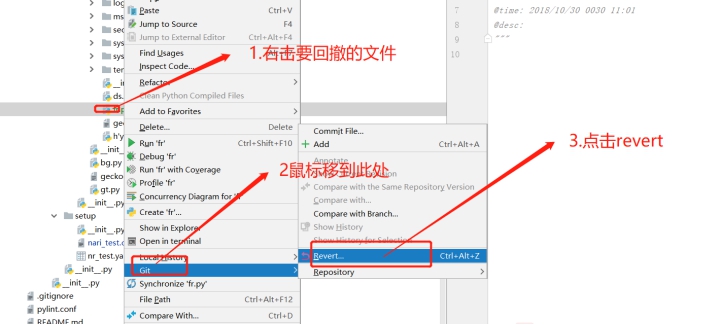
##### 11.2.1.1如何查看pull失败原因（本地仓库发生了变化导致pull失败，stash的作用，保存本地变化。就可以解决这个问题）



##### 11.2.1.2 回撤所有问题文件或先找到问题文件，一个一个的回撤



单个文件的回撤

 回撤所有问题文件后，在pull就可以成功了

# 12 cherry-pick的用法

Git 查看git提交日志：

git log --oneline -3

-3：表示查看3次提交记录

git log --oneline -3

b2c0413c (HEAD -> dev, origin/dev) <E8><B7><B3><E8><BD><AC><E4><BC><98><E5><8C><96>

69d30118 <E8><B7><B3><E8><BD><AC><E4><BC><98><E5><8C><96>

d3cdab88 Merge branch 'dev' of http://192.168.176.201:10010/r/project/SEA2017 into dev

首先在一个分支，挑选需要的commit

如上：git cherry-pick b2c0413c

然后切换到需要合并这个commit的分支：

运行：git cherry-pick b2c0413c

  如果顺利，就会正常提交。结果：

Finished one cherry-pick.

# On branch old\_cc

# Your branch is ahead of 'origin/old\_cc' by 3 commits.

2. 如果在cherry-pick 的过程中出现了冲突

Automatic cherry-pick failed.  After resolving the conflicts,

mark the corrected paths with 'git add <paths>' or 'git rm <paths>'

and commit the result with:

        git commit -c 15a2b6c61927e5aed6718de89ad9dafba939a90b

就跟普通的冲突一样，手工解决：

执行git status 看哪些文件出现冲突

$ git status

both modified:      app/models/user.rb

接着手动解决冲突的文件，然后通过git add把改到添加到索引，最后执行git commit提交修改。

$ vim app/models/user.rb

$ git add app/models/user.rb

git commit -c <原commit号>

3 cherry-pick的几种模式

git cherry-pick <commit id>:单独合并一个提交

git cherry-pick  -x <commit id>：同上，不同点：保留原提交者信息。

Git从1.7.2版本开始支持批量cherry-pick，就是一次可以cherry-pick一个区间的commit。

 git cherry-pick <start-commit-id>..<end-commit-id>

 或者

 git cherry-pick <start-commit-id>^..<end-commit-id>

前者表示把<start-commit-id>到<end-commit-id>之间(左开右闭，不包含start-commit-id)的提交cherry-pick到当前分支；

后者表示把<start-commit-id>到<end-commit-id>之间(闭区间，包含start-commit-id)的提交cherry-pick到当前分支。

其中，<start-commit-id>到<end-commit-id>只需要commit-id的前6位即可，并且<start-commit-id>在时间上必须早于<end-commit-id>

# 13 Git stash 和 git cherry-pick的比较

git stash 和 git cherry-pick

开发人员常常遇到这种情况：花了几天时间一直在做一个新功能，已经改了差不多十几个文件，突然有一个bug需要紧急解决，然后给一个build测试组。在Git问世之前基本上靠手动备份，费时且容易出错。

git stash命令简而言之就是帮助开发人员暂时搁置当前已做的改动，倒退到改动前的状态，进行其他的必要操作（比如发布，或者解决一个bug，或者branch，等等），之后还可以重新载入之前搁置的改动，很cool吧!

步骤:

首先*git add (用git add把所有的改动加到staging area)*

其次git stash(接着用git stash把这些改动搁置)

反馈:git stash save "what you want to say\*\*\*\*\*\*\*\*XXXXXX"

到这里，当前工作平台就回复到改动之前了。该干嘛干嘛，此处省略1万字。

需要找回之前搁置的改动继续先前的工作了？

git stash apply 即可。

也可以用 git stash list 来查看所有的搁置版本（可能搁置了很多次，最好不要这样，容易搞混）

在出现一个搁置栈的情况下，比如如果你想找回栈中的第2个，可以用 git stash apply stash@{1}

如果想找回第1个，可以用 git stash pop

如果想删除一个stash，git stash drop <id>

删除所有stash，git stash clear

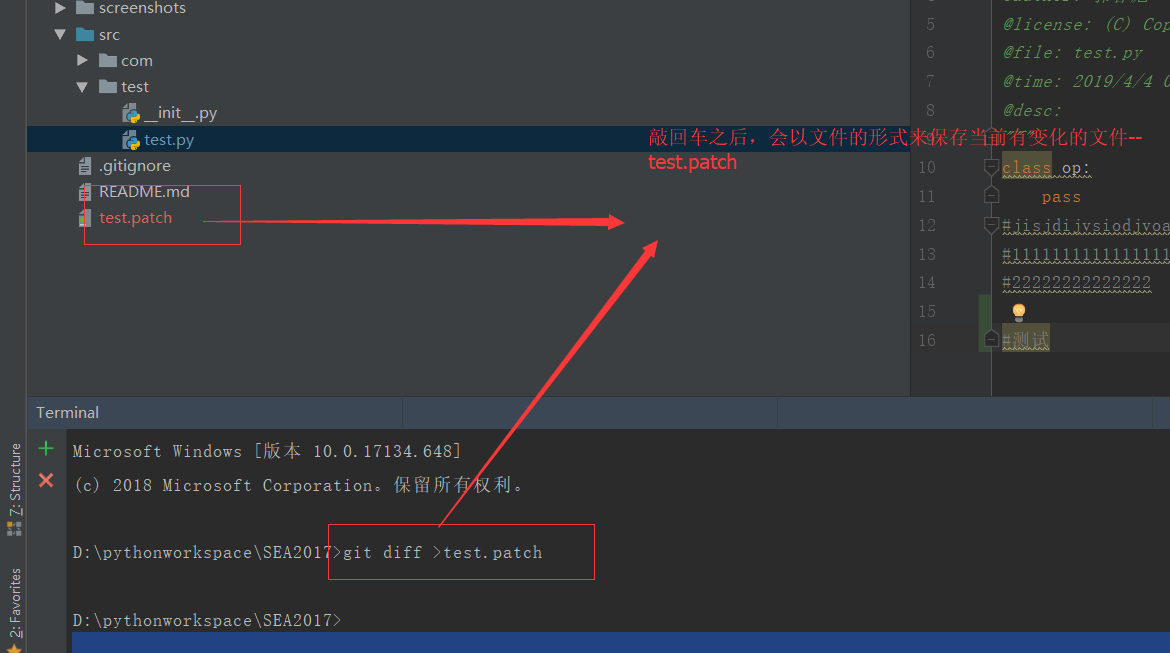
# 14 git patch

git提供了两种简单的patch方案。一是用git diff生成的标准patch，二是git format-patch生成的Git专用Patch。

.diff文件只是记录文件改变的内容，不带有commit记录信息,多个commit可以合并成一个diff文件。  
.patch文件带有记录文件改变的内容，也带有commit记录信息,每个commit对应一个patch文件。

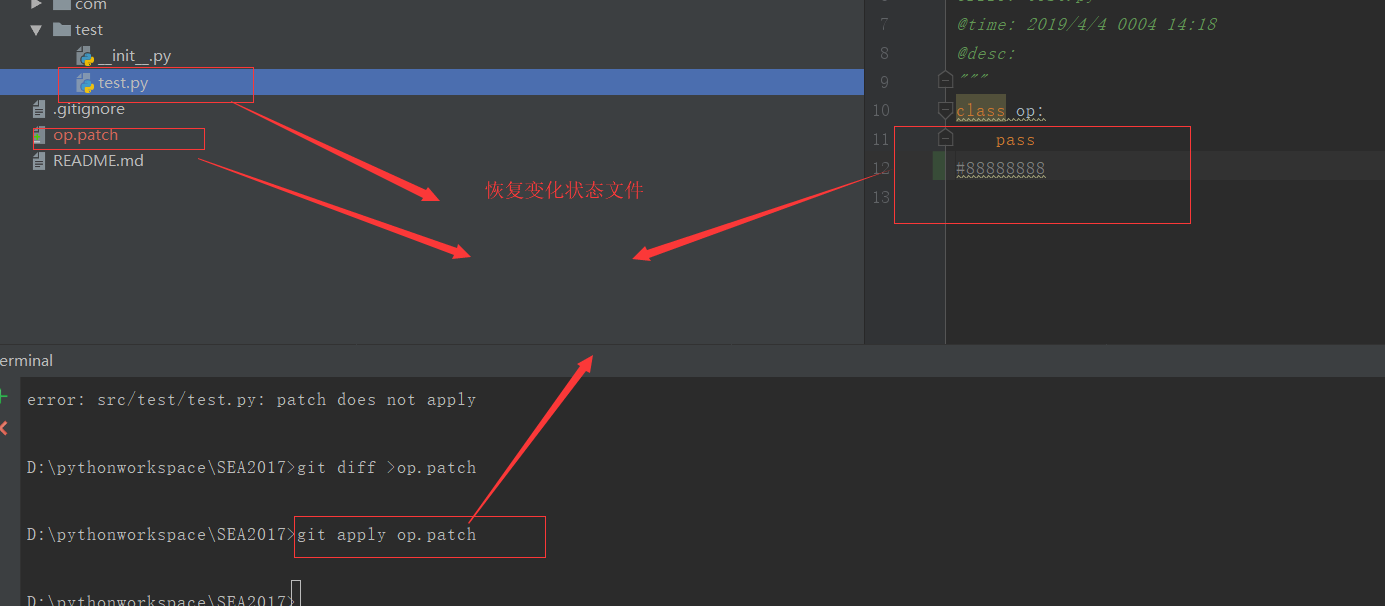
### 14.1 Git diff与stash功能的相同用法，却别就在于，它生成的patch文件来保存当前状态变化的文件

命令 ： git diff > test.patch(注意git是小写)



应用：test.patch文件

g it apply test.patch (注意git是小写)即可



##### 14.1.1使用命令行

git diff 【commit sha1 id】 【commit sha1 id】 > 【diff文件名】

eg：

git diff 2a2fb4539925bfa4a141fe492d9828d030f7c8a8 89aebfcc73bdac8054be1a242598610d8ed5f3c8 > patch.diff

### 14.2 git format-patch

#### 14.2.1 、创建patch 文件的常用命令行

##### 14.2.1.1\*某次提交（含）之前的几次提交：

git format-patch 【commit sha1 id】-n

n指从sha1 id对应的commit开始算起n个提交。  
eg：

git format-patch 2a2fb4539925bfa4a141fe492d9828d030f7c8a8 -2

##### 14.2.1.2\*某个提交的patch：

git format-patch 【commit sha1 id】 -1

eg：

git format-patch 2a2fb4539925bfa4a141fe492d9828d030f7c8a8 -1

##### 14.2.1.3\*某两次提交之间的所有patch:

git format-patch 【commit sha1 id】..【commit sha1 id】

eg：

git format-patch 2a2fb4539925bfa4a141fe492d9828d030f7c8a8..89aebfcc73bdac8054be1a2425986

#### 14.2.2应用git format-patch生成的文件

1. 检查patch文件

git apply --stat 0001-minor-fix.patch

2． 看看能否应用成功

git apply --check 0001-minor-fix.patch

1. 应用

git am -s < 0001-minor-fix.patc