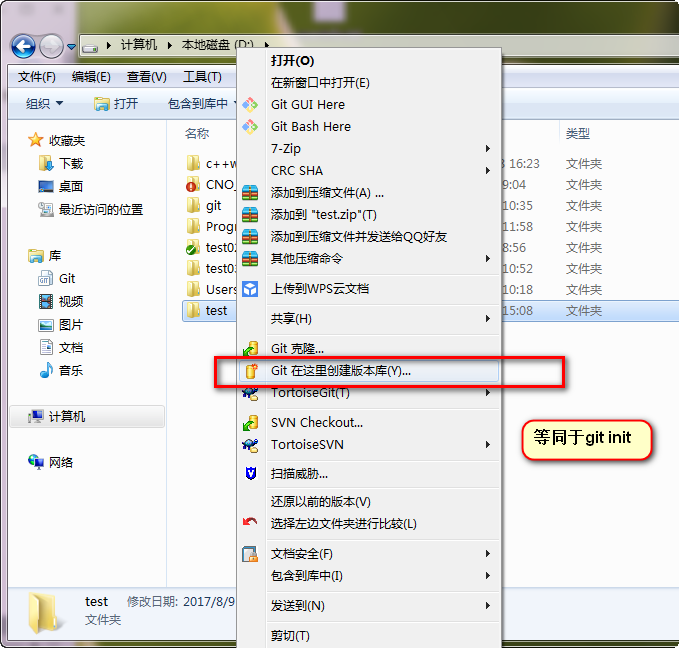
## 1.SVN版本库迁移至Git

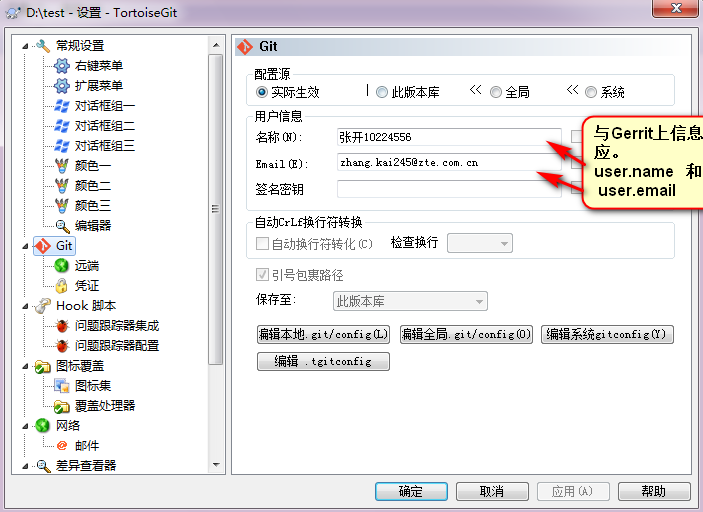
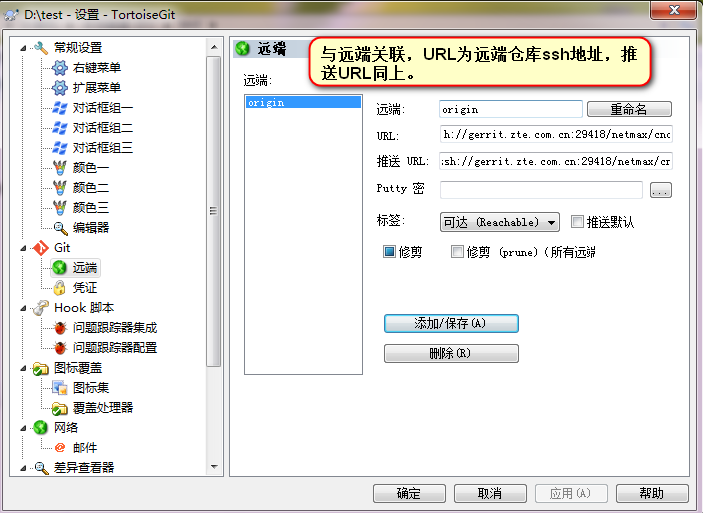
因为项目工程比较大（约7.6G），使用git svn clone url因为延时各种原因断掉。所以采用手动分部分上传项目文件。安装SVN、Git以及图形化工具不在此叙述。

1. 首先采用SVN方式（svn checkout）将远程版本库拉至本地。
2. 创建本地Git仓库，例如在D://test下右键，操作如下：

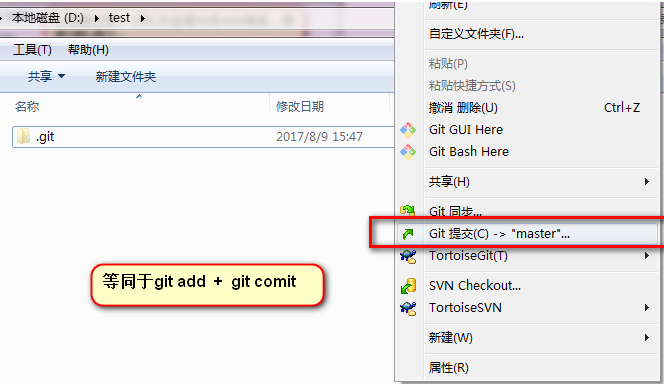
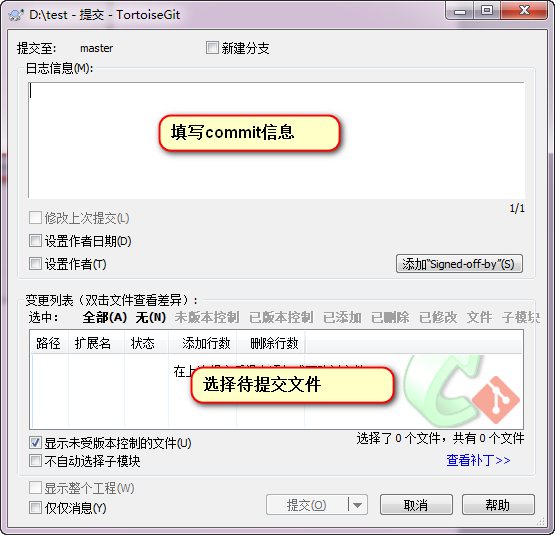
 

1. 配置本地仓库与远程仓库关联。在test文件夹内右键点击TortoiseGit，进而点击设置，进入配置界面。

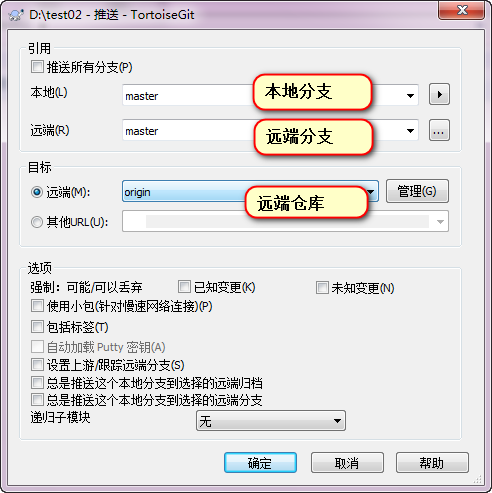


1. 将本地SVN中一部分文件拷贝至test目录下，使用git add，git commit操作，图形化具体操作如下：

1. 完成之后就需要推送至远程仓库。推送（Git push origin master）图形化操作如下：

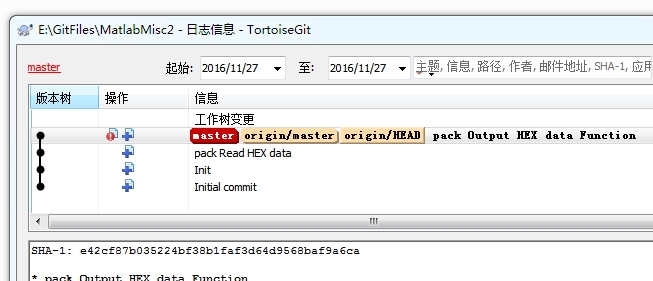
 

至此上传部分文件完成，重复此过程，将剩余代码全部上传。

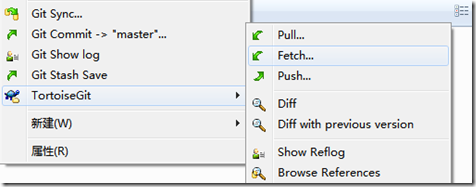
## 2.版本冲突处理

Git的Pull其实是fetch与Merge两个命令的合并。建议使用fetch+merge组合操作。

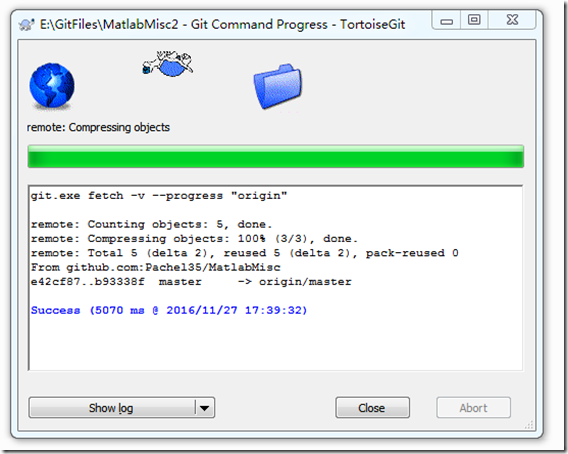
1.未 fetch 之前，本地 master 分支的头指针与 remote/origin/master 分支的头指针指向是同一个位置。



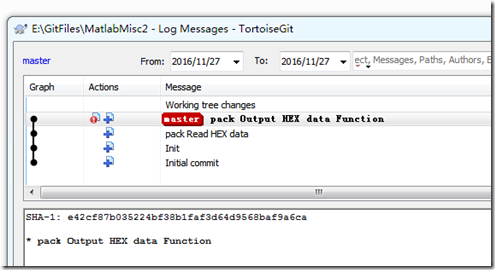
右键菜单 –> fetch



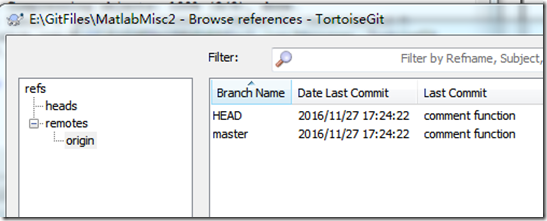
可以看到有 fetch 到更新到本地仓库。



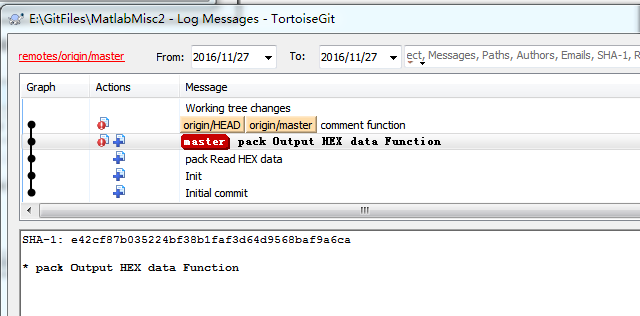
1. 本地查看 log，可以看到此时只有 master 的头指针，已经看不到remote的指针了。



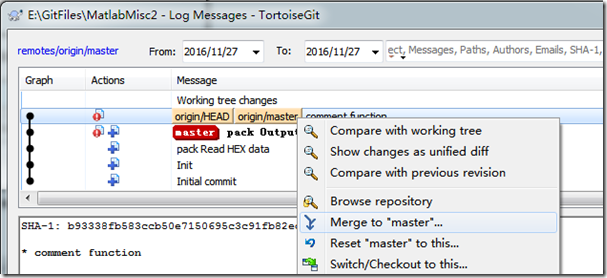
此时，点击左上角的master，并点击选中 remotes/origin/master 分支



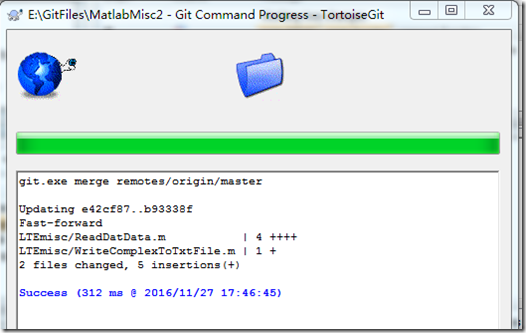
这样便可以看到 remotes/origin/master 分支的头指针，跟本地的 master 分支头指针并不在一个位置了。



1. 把 origin/master 分支合并到 本地的master右击 origin/master 的标签，选择 Merge to “master”…



完成 Merge 操作后，可以看到 Merge 的信息，有两个文件进行了修改。



## 3.Git学习



1. Git clone 网址
2. 拉取代码 git pull <远程主机名> <远程分支>：<本地分支>
3. 推送代码 git push <远程主机名> <本地分支>：<远程分支>
4. 创建分支 git branch 分支名
5. 切换分支 git checkout 分支名
6. 撤销操作

①撤销工作区内容 手动修改文件或者git checkout --filename

②撤销暂存区内容 git reset HEAD filename

③撤销本地仓库提交即版本回退 git reset --hard HEAD^ (HEAD^为上一个版本，回退标识也可以是版本号)

=git log 穿越过去， 查看版本号信息

=git reflog 回到未来， 查看提交历史记录

<https://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/0013744142037508cf42e51debf49668810645e02887691000>

2019.2.15补充：

撤销已提交至远端库的文件：

小乌龟git revert，然后push。 这样会产生一个新的提交记录，不至于像git reset回退，每个用户还需要git reset --hard origin/master 强制同步与远端相同。

****使用“git revert -n 版本号”反做，并使用“git commit -m 版本名”提交.****

1. 类似Svn从仓库中拉取部分文件或文件夹

在.git文件夹中新建文件sparse-checkout文件，其中添加需要拉取的内容，eg：

src/gul/build/DB/CreateAllversionDB.bat src/gul/\*

然后在bash中使其生效 git config core.sparsecheckout true

执行pull命令即可 git pull origin master

<http://www.cnblogs.com/xilifeng/p/5225666.html>

8.强制同步至远程仓库版本



9.操作untracked file

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | # 删除 untracked files  git clean -f    # 连 untracked 的目录也一起删掉  git clean -fd    # 连 gitignore 的untrack 文件/目录也一起删掉 （慎用，一般这个是用来删掉编译出来的 .o之类的文件用的）  git clean -xfd    # 在用上述 git clean 前，墙裂建议加上 -n 参数来先看看会删掉哪些文件，防止重要文件被误删  git clean -nxfd  git clean -nf  git clean -nfd |
|  | **对工作区进行处理**  通常是git status后一片红（包括modified和untrack的）  git checkout . #本地所有修改的。没有的提交的，都返回到原来的状态  git stash #把所有没有提交的修改暂存到stash里面。可用git stash pop回复。  git reset --hard HASH #返回到某个节点，不保留修改。  git reset --soft HASH #返回到某个节点。保留修改  git clean -df #返回到某个节点  git clean 参数  -n 显示 将要 删除的 文件 和 目录  -f 删除 文件  -df 删除 文件 和 目录 |
| 1.  2. | **分支合并--某个分支上的单个commit**  希望只合并某个分支上的某次修改commit到指定的分支上.那么使用cherry-pick命令来操作了.  使用git log 查看提交的信息,记住commit id.  git checkout 要修改的分支  git cherry-pick 某个commit id   // 把某个commit id的提交合并到当前分支.  **合并某个分支上的一系列commits**  dd2e86 - 946992 -9143a9 - a6fd86 - 5a6057 [master]             \              76cada - 62ecb3 - b886a0 [feature]  假设你需要合并feature分支的commit76cada ~62ecb3 到master分支  ①在feature分支，建立新分支，并指明新分支的最后一个commit  Git checkout -b newbranch 62ecb3  ②变基，从76cada前commit至当前分支尾，--onto master 合并至master分支上【注意:^ 因为是半开半闭区间】  Git rabase --onto master 76cada^ |
|  |  |

### 2.git pull –rebase 理解

### 20160926112331953

这个命令做了以下内容：   
a.把你 commit 到本地仓库的内容，取出来放到暂存区(stash)（这时你的工作区是干净的）   
b.然后从远端拉取代码到本地，由于工作区是干净的，所以不会有冲突   
c.从暂存区把你之前提交的内容取出来，跟拉下来的代码合并

所以 rebase 在拉代码前要确保你本地工作区是干净的，如果你本地修改的内容没完全 commit 或者 stash，就会 rebase 失败。

修改commit提交顺序，也就是先做了git commit但忘了git pull origin master，导致git push失败。

|  |
| --- |
| 1. git commit 2. git stash save “xxx” ---- 3. git rebase master 4. git pull origin master 5. git stash pop --- //git pull --rebase 6. git push origin master |

这种是理想的先提交后pull情况，中间没有push失败。

错误情况如下：

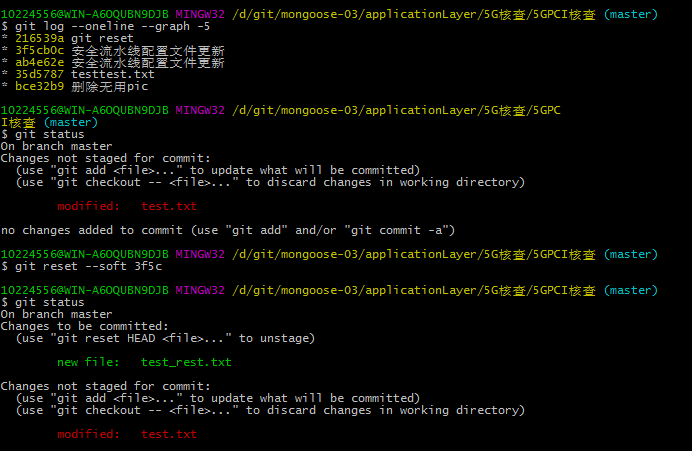


目前尝试git rebase 解决不了。

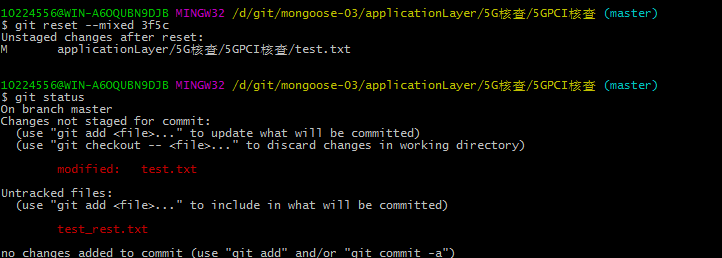
可以先 git reset bce32b9（默认--mixed）

然后重新git commit，然后gitpush。

Git reset --soft commitid 工作区，暂存区均回退（暂存区保留原提交状态）



Git reset --mixed xxx工作区，暂存区干净（暂存区待重新提交）



Git reset --hard xxx 硬回退（谨慎使用）。

### 3.[git stash[贮存] 用法总结和注意点](https://www.cnblogs.com/zndxall/p/9586088.html)

<https://www.cnblogs.com/zndxall/archive/2018/09/04/9586088.html>

|  |
| --- |
| 常用git stash命令：  （1）**git stash** save "save message"  : 执行存储时，添加备注，方便查找，只有git stash 也要可以的，但查找时不方便识别。  （2）**git stash list** ：查看stash了哪些存储  （3）**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@{$num}，比如第二个 git stash show stash@{1}  （4）**git stash show -p** : 显示第一个存储的改动，如果想显示其他存存储，命令：git stash show  stash@{$num}  -p ，比如第二个：git stash show  stash@{1}  -p  （5）**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@{$num} ， 比如第二个：git stash apply stash@{1}  （6）**git stash pop** ：命令恢复之前缓存的工作目录，将缓存堆栈中的对应stash删除，并将对应修改应用到当前的工作目录下,默认为第一个stash,即stash@{0}，如果要应用并删除其他stash，命令：git stash pop stash@{$num} ，比如应用并删除第二个：git stash pop stash@{1}  （7）**git stash drop** stash@{$num} ：丢弃stash@{$num}存储，从列表中删除这个存储  （8）**git stash clear** ：删除所有缓存的stash |

附：参考链接

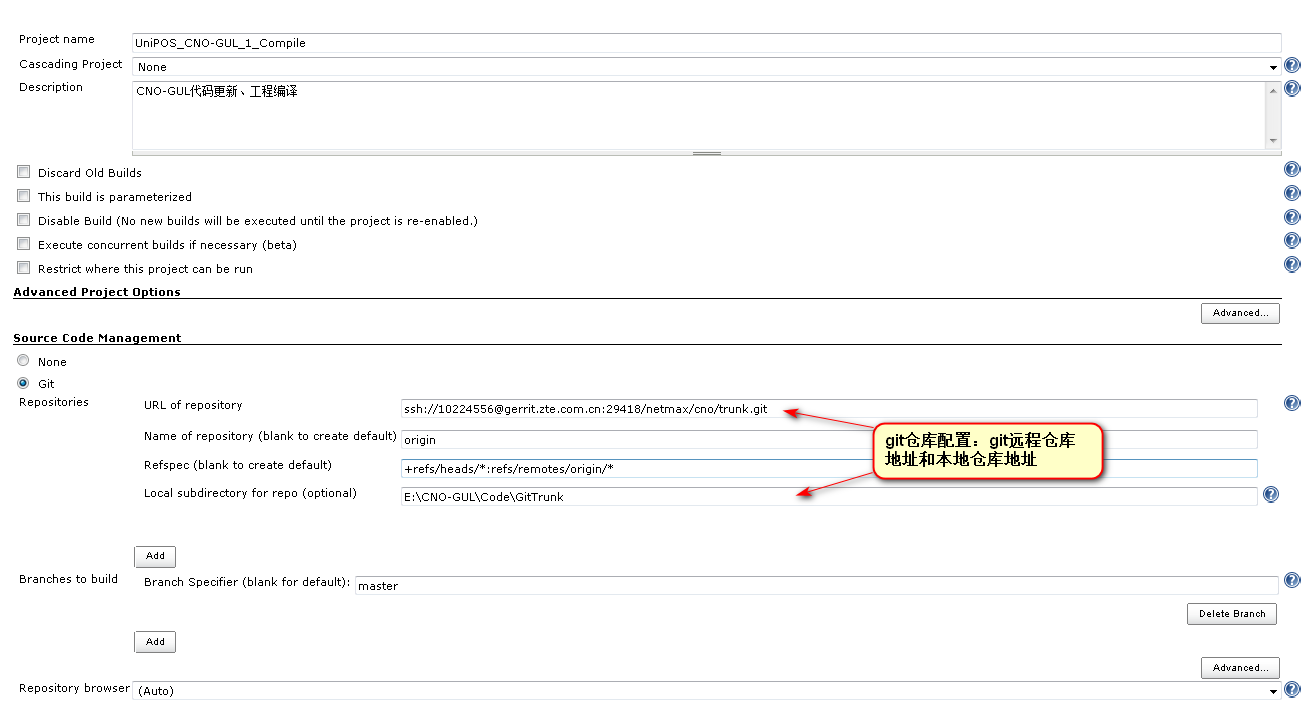
<http://www.ruanyifeng.com/blog/2014/06/git_remote.html>

<https://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000>

## 4.Hadson修改代码版本源

从SVN仓库构建代码转换为从Git仓库构建代码，需修改以下部分：

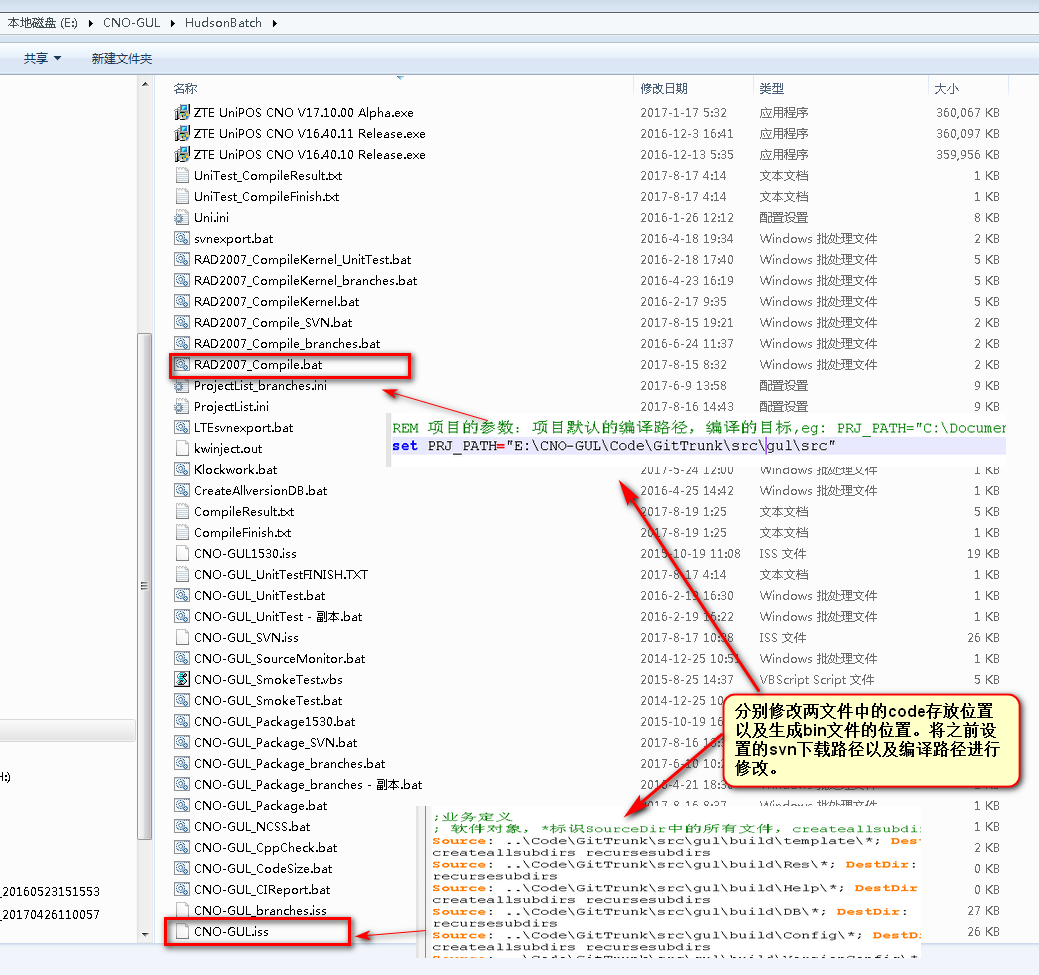
①源代码获取方式修改，从SVN修改问Git方式。



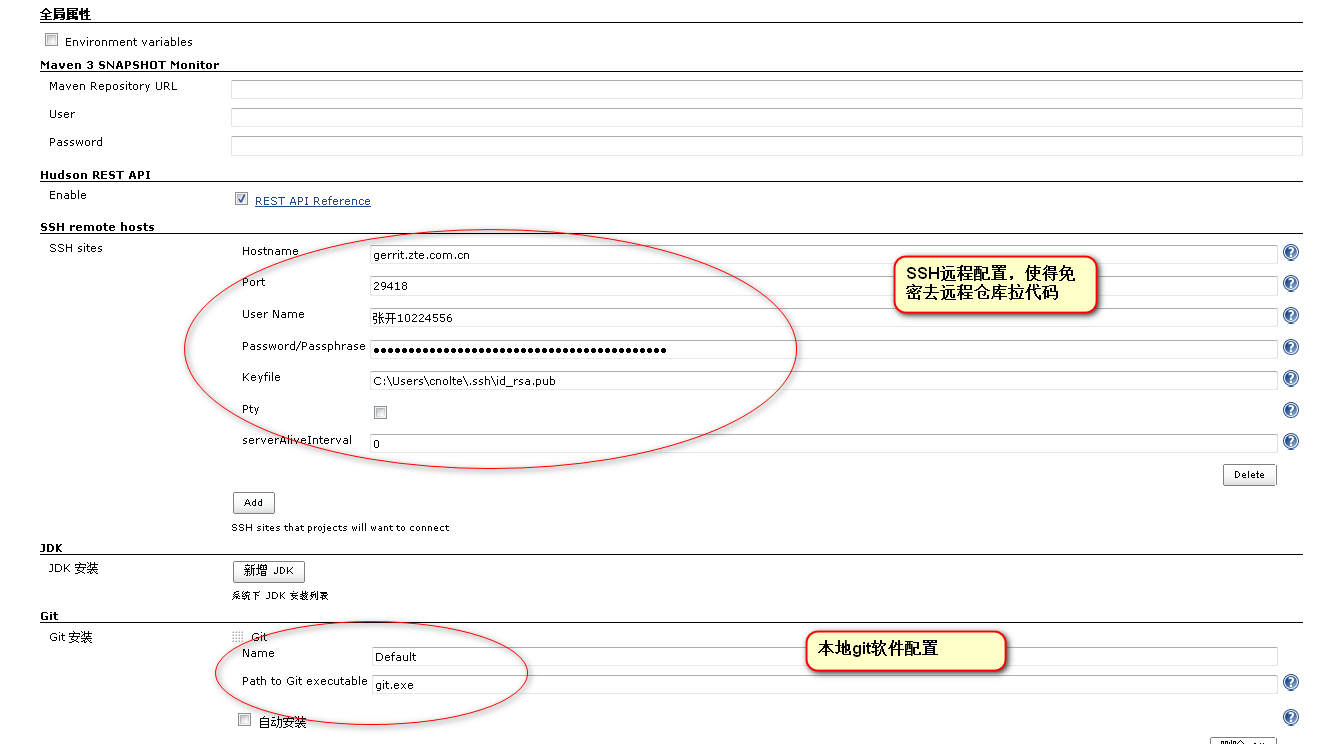
②编译批处理脚本位置修改。



③软件编译批处理脚本以及生成位置需要修改。



此外还需要在Hadson系统管理>>系统设置处，对SSH远端主机进行配置处理。



Hadson编译、打包过程问题总结

一．在hadson上配置git环境以及手动初始化git仓库之后，执行hadson构建，两个半小时后出现以下问题：

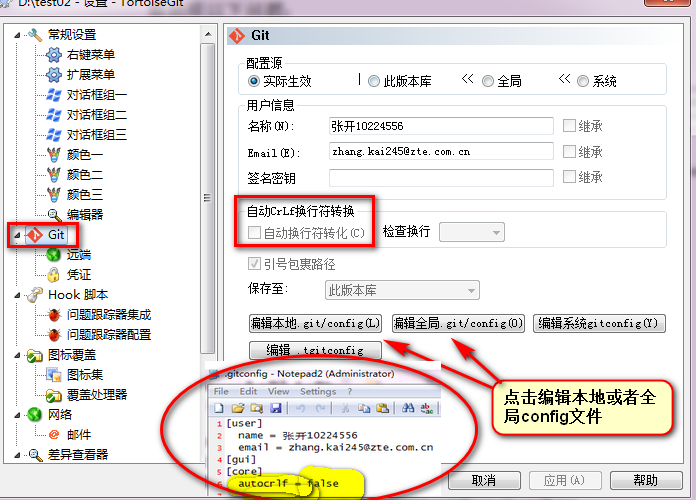


首先工程编译失败，其次发现批处理文件执行语句被莫名其妙拆分。经定位问题，是文件的格式有问题（由gerrit Linux文件格式clone至本地windows上出现问题，也就是换行符出现问题）。

解决方式：在git仓库中右键点击git bash设置git仓库换行符转换是否自动开启。

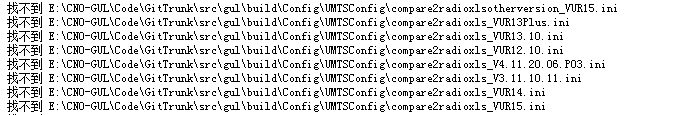
git config --global autocrlf true

或者右键打开Tortoisegit>>setting，具体操作如下：

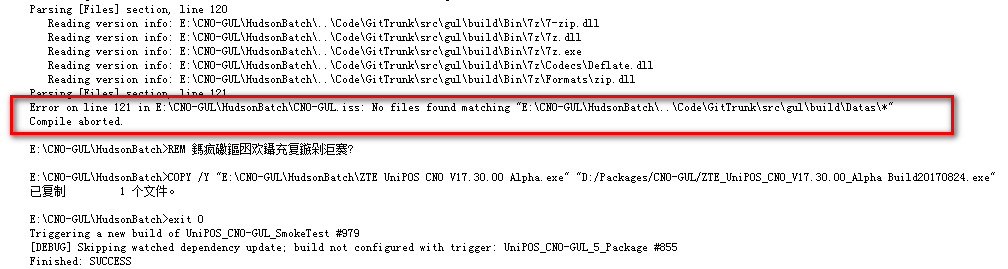


二.修改完换行符问题之后，工程编译成功。但是执行批处理时遇到各种文件找不到。然后定位本地仓库GitTrunk，发现确实没有这类文件。定位远程仓库，也没有此类文件。经过同师傅沟通，这类文件时软件版本文件，非必须。





三．编译构建成功后，执行打包构建。出现如下错误：



定位错误，找不到文件夹Datas，后发现Datas是空文件夹，在Girret远程仓库中并无此文件夹。发现上传空文件夹至远程仓库，git会默认不上传此文件夹。所以在pull时也就拉取不到此文件夹。

解决方式：在空文件夹内新建空文档，重新上传此文件夹至gerrit远程仓库，随后pull的时候便可以将此文件夹拉取至本地。

## 5.Gerrit触发VerifyCI

1. 通过文件生成changeID。



1. Git commit -o 文件 -m Message 或者git add 文件 然后git commit
2. 提交 ：

git push origin HEAD:refs/for/master%r=liu.hong52@zte.com.cn

1. 修改补丁后：

Git add 文件

Git commit --amend （git commit –amend –no-edit, 不修改commit信息）

然后 git push

不需要git commit -o 重新提交，那样就是新的changeId

补充：

1.Git commit -amend 打补丁：<https://www.jianshu.com/p/a8a2ac58f37d>

2.

♥   重新提交时有冲突？（使用rebase把分支重定位到目标分支最新，然后重新提交评审）

git checkout master

git pull origin master

git checkout bugfix/ec61100123456

git rebase -i master

git push origin HEAD:refs/for/master

## 6.交互式的rebase

交互式的 rebase 允许你更改并入新分支的提交。这比自动的 rebase 更加强大,因为它提供了对分支上提交历史完整的控制

一般来说，这被用于将 feature 分支并入 master 分支之前，清理混乱的历史。

命令示例：

1 git checkout feature

2 git rebase -i master

它会打开一个文本编辑器，显示所有将被移动的提交：

1 pick 33d5b7a Message for commit #1

2 pick 9480b3d Message for commit #2

3 pick 5c67e61 Message for commit #3

保存后关闭文件，Git 会根据你的指令来执行 rebase

忽略不重要的提交会让你的 feature 分支的历史更清晰易读。这是 git merge 做不到的

Git rebase失败了，原因很简单，两个分支修改个同一个文件，产生了冲突。所以先需要解决冲突：

打开冲突的文件，解决冲突

原文件：

func NewFunc() {<<<<<<< HEAD=======

fmt.Println("add new func")>>>>>>> add new func}

修改后：

func NewFunc() {

fmt.Println("add new func")}

现在通过add添加

HowiedeiMac:hello howie$ git add newFunc.go

现在是重点，之前的rebase其实只是完成了一半，由于出现冲突而终止，现在冲突解决，可以通过git rebase —continue继续完成之前的rebase操作。

HowiedeiMac:hello howie$ git rebase --continue

Applying: add new func

rebase完成，再查看一下提交历史：

HowiedeiMac:hello howie$ git log --oneline --graph

\* b2593e6 (HEAD -> feature) add new func

\* 0e80f97 (master) do something

\* 94c134b init base

提交记录已经是一条完美的直线。现在切换到主分支master，将feather分支上的提交合并过来。

git checkout master

git merge featur

**rebase黄金法则**

法则：

绝不要在公共的分支上使用它！

在你运行 git rebase 之前，一定要问问你自己「有没有别人正在这个分支上工作？」。如果答案是肯定的，那就不能rebase

**强制推送**

如果你想把 rebase 之后的 master 分支推送到远程仓库，Git 会阻止你这么做，因为两个分支包含冲突。但你可以传入 –force 标记来强行推送

1 git push --force

rebase在开发中的使用场景

rebase 可以或多或少应用在你们团队的 Git 工作流中

**本地清理（合并多次提交为一条）**

隔一段时间执行一次交互式 rebase，你可以保证你 feature 分支中的每一个提交都是专注和有意义的。你在写代码时不用担心造成孤立的提交——因为你后面一定能修复

下面的命令对最新的 3 次提交进行了交互式 rebase：

1 git checkout feature

2 git rebase -i HEAD~3

你实际上没有移动分支——你只是将之后的 3 次提交合并了

交互式 rebase 是在你工作流中引入 git rebase 的的好办法，因为它只影响本地分支。其他开发者只能看到你已经完成的结果，那就是一个非常整洁、易于追踪的分支历史。

但同样的，这只能用在私有分支上。如果你在同一个 feature 分支和其他开发者合作的话，这个分支是公开的，你不能重写这个历史。

————————————————

原文链接：

<https://blog.csdn.net/wangnan9279/article/details/79287631>