TORTOISEGIT 使用指南

哈尔滨工业大学

目录

TortoiseGit 使用指南

Git 是一个开源的分布式版本控制系统，可有效、高速的处理任何规模的项目的版本管理，为团队协作提供便利。那么什么是版本控制呢？

版本控制（Revision control）是一种在开发过程中用于管理我们对文件、目录或工程等内容的修改历史，方便查看更改历史记录，备份以便恢复以前的版本的软件工程技术。

(1). 实现跨区域多人协同开发；

(2). 追踪和记载一个或者多个文件的历史记录；

(3). 组织和保护你的源代码和文档；

(4). 统计工作量；

(5). 并行开发、提高开发效率；

(6). 跟踪记录整个软件的开发过程；

(7). 减轻开发人员的负担，节省时间，同时降低人为错误。

本文以 window 操作系统为例介绍 TortoiseGit 的使用。

1. 安装及基本设置

1.1. 下载安装

TortoiseGit 是 Git 类分布式版本控制系统图形化客户端，并提供多国语言包支持，使用者可能根据需要选择相应的语言包。

要安装软件首先到官网下载相应软件。因为 TortoiseGit 只是一个程序壳,必须依赖一个 Git Core。要完整安装，需要下载三个软件依次安装。

Git 下载地址：https://git-for-windows.github.io；TortoiseGit 及语言包下载地址：https://TortoiseGit.org/download。

安装顺序如下：

(1).首先安装 Git（比如 Git-2.12.0-64-bit.exe）。

(2).安装 TortoiseGit（TortoiseGit-2.4.0.2-64）。

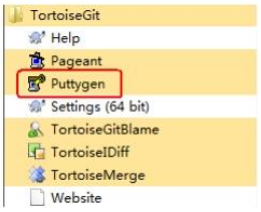
(3).安装汉化包(TortoiseGit-LanguagePack-2.4.0.0-64bit-zh_Cn.msi)，汉化包是可选的，如果习惯了英文界面，也可不装。

1.2. 设置

1.2.1. 设置传输认证方式

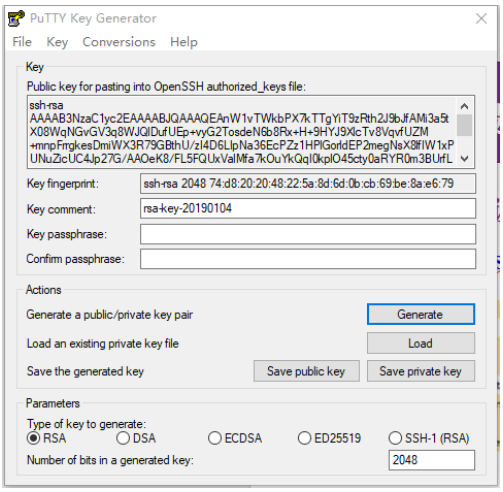
安装完成后,可以选择使用 OpenSSH 类型客户端或是使用 Putty 类型客户端,如果使用的是默认的 Putty 客户端,由于 TortoiseGit GUI 连接不支持 server 端自

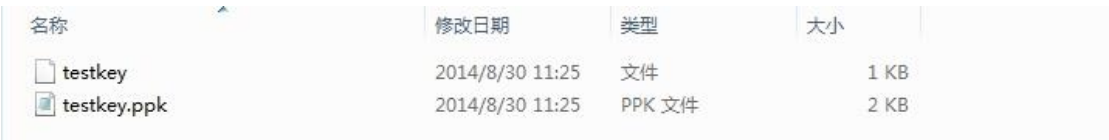
定义端口配置,若 GitLab 使用的 SSH 端口为自定义端口,则需要使用 Putty 的 authentication agent 去做一个本地的端口转发.



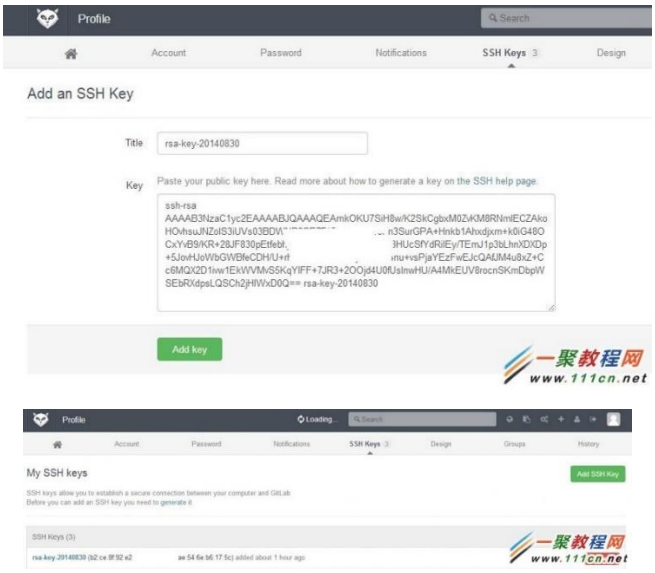
首先使用 TortoiseGit 自带的 Puttygen 创建本地的公/私钥对。

点击 Generate 按钮,在窗口空白处按照提示晃动鼠标,生成公/私钥对,并保存到本地。其中 testkey 为公钥,testkey.ppk 为私钥

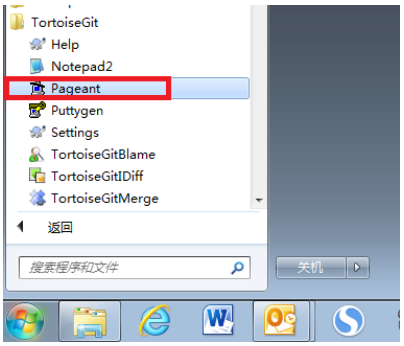




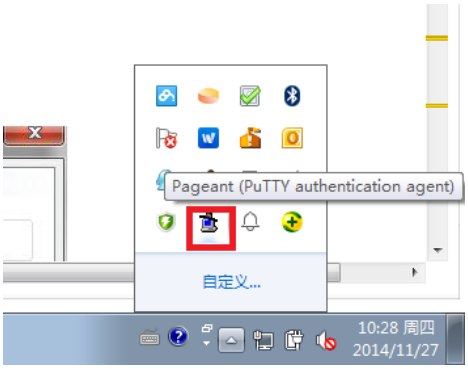
将窗口生成的 Public key 粘贴到 GitHub（或 GitLab）站点具体使用账号的 SSh Keys 内即完成公钥上传.

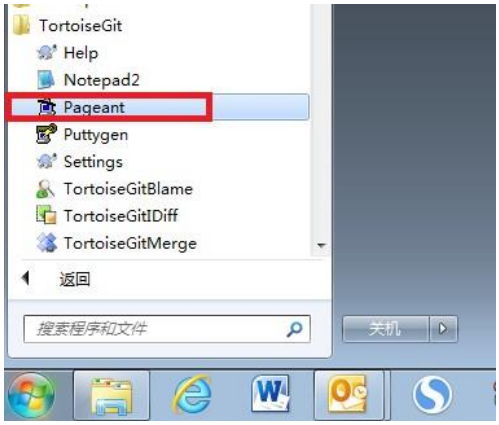


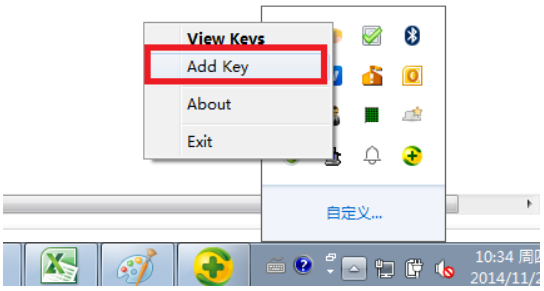
使用 Putty 连接 gitlab 服务器,TortoiseGit 自带了 Putty，即 Pageant。



击打开 Pageant，右下角，右键 Add Key，选中保存的私钥。





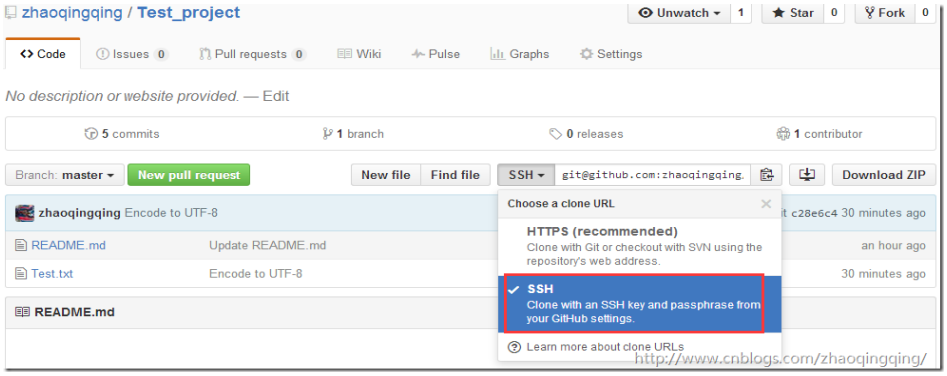


open 之前需选择之前本地生成的私钥文件，配置完成。

连接成功会在右下角任务栏出现任务图标

1.2.2. 服务器端设置

设置 ssh 不用输入密码提交的即在 gitlab 的项目页面，设置项目的拉取方式为 SSH。



1.2.3. 配置 TortoiseGit

(1).TortoiseGit https 保存用户名和密码

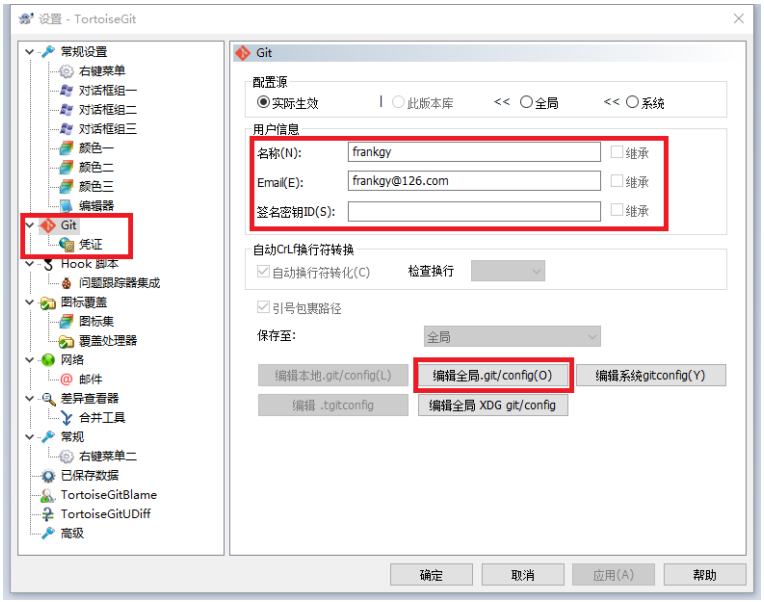
TortoiseGit-设置-凭证-编辑全局.git/config，添加：

[credential]

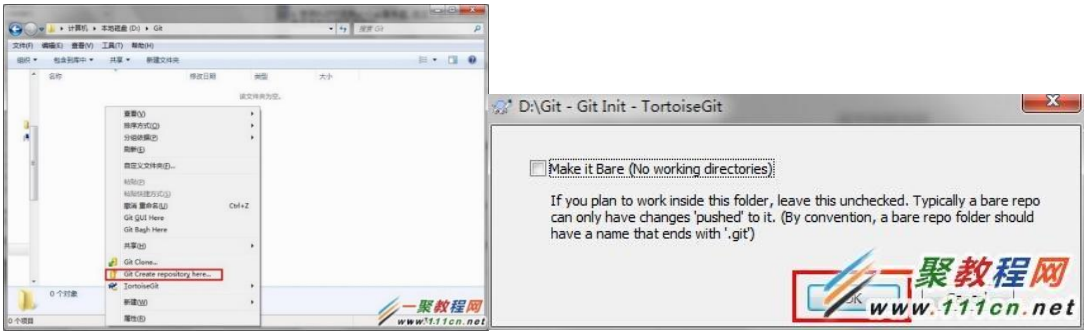
helper=store

用这种方式是把用户名和密码，以明文的方式保存在 C:\Users\你当前用户名 ，例：C:\Users\zhaoq。

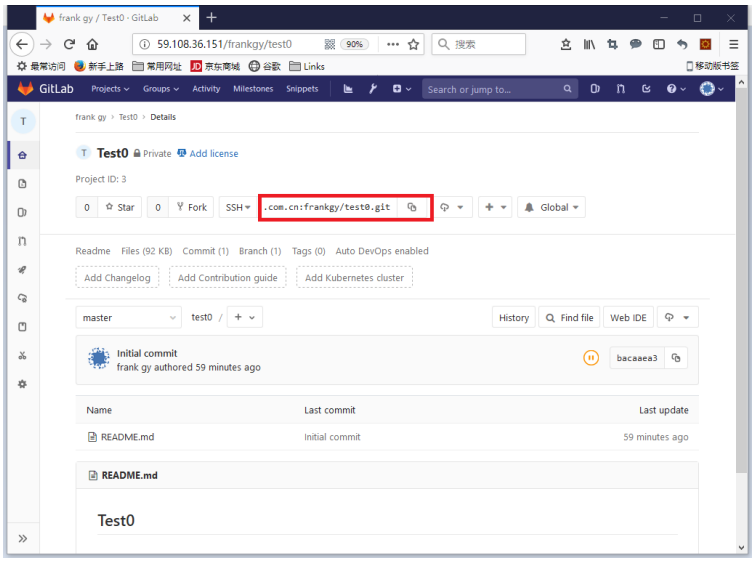
(2).从 windows 开始菜单选择 TortoiseGit-Setting 工具,在对话框里选择 Git 项,填写相关 info。



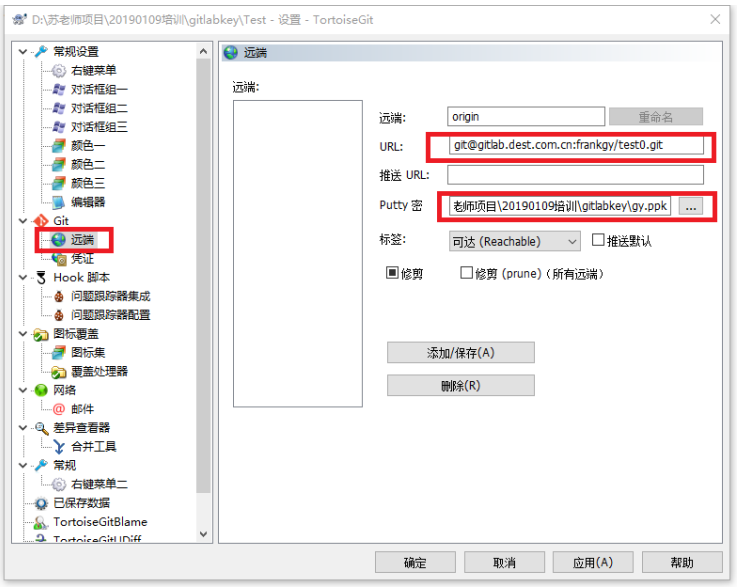
(3).在 D 盘新建一个目录,例如"D:\Git",并进入目录右键目录空白处选择"Git Create repository here...",弹出对话框点确认,这样即建立了一个本地 Git 仓库.



(4).在 TortoiseGit 中设置服务器端地址:在浏览器上登录 Gitlab 网站,如下图所示，拷贝出链接。



然后在机器中，将拷贝的 URL 粘贴到相应位置，并在“Putty 密钥”填入生成的私钥。



点击确定，然后获取远端新建分支。

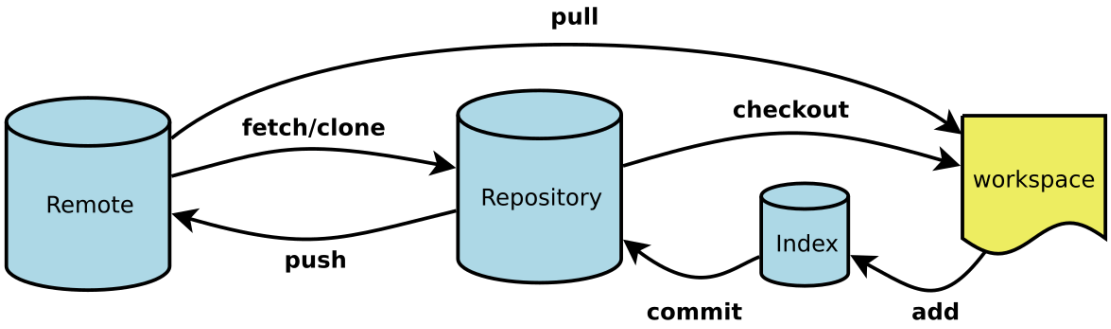
2. 常用术语

2.1. 工作区、暂存区、版本库及远程仓库

2.1.1. 四个工作区域及关系

(1). Workspace：工作区也称工作目录，就是你平时存放项目代码的地方;
(2). Index / Stage：暂存区，用于临时存放你的改动，事实上它只是一个文件，保存即将提交到文件列表信息;
(3). Repository：仓库区(或版本库或称本地仓库)，就是安全存放数据的位置，这里面有你提交到所有版本的数据。其中 HEAD 指向最新放入仓库的版本;
(4). Remote：远程仓库，托管代码的服务器，可以简单的认为是你项目组中的一台电脑用于远程数据交换

Git 本地有四个工作区域:工作目录（Working Directory）、暂存区(Stage/Index)、资源库(Repository 或 Git Directory)、Git 仓库(Remote Directory)。文件在这四个区域之间的转换关系如下：



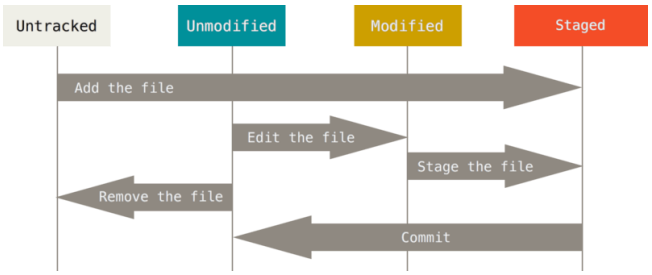
2.1.2. 工作流程

Git 的工作流程一般是这样的：
(1). 在工作目录中添加、修改文件；
(2). 将需要进行版本管理的文件放入暂存区域；
(3). 将暂存区域的文件提交到 Git 仓库。
因此，git 管理的文件有三种状态：已修改（modified）,已暂存（staged），已提交(committed)

2.1.3. 文件的四种状态

版本控制就是对文件的版本控制，要对文件进行修改、提交等操作，首先要知道文件当前在什么状态，不然可能会提交了现在还不想提交的文件，或者要提交的文件没提交上。

GIT 不关心文件两个版本之间的具体差别，而是关心文件的整体是否有改变，若文件被改变，在添加提交时就生成文件新版本的快照，而判断文件整体是否改变的方法就是用 SHA-1 算法计算文件的校验和。



(1). Untracked: 未跟踪，此文件在文件夹中，但并没有加入到 git 库，不参与版本控制. 通过 git add 状态变为 Staged.

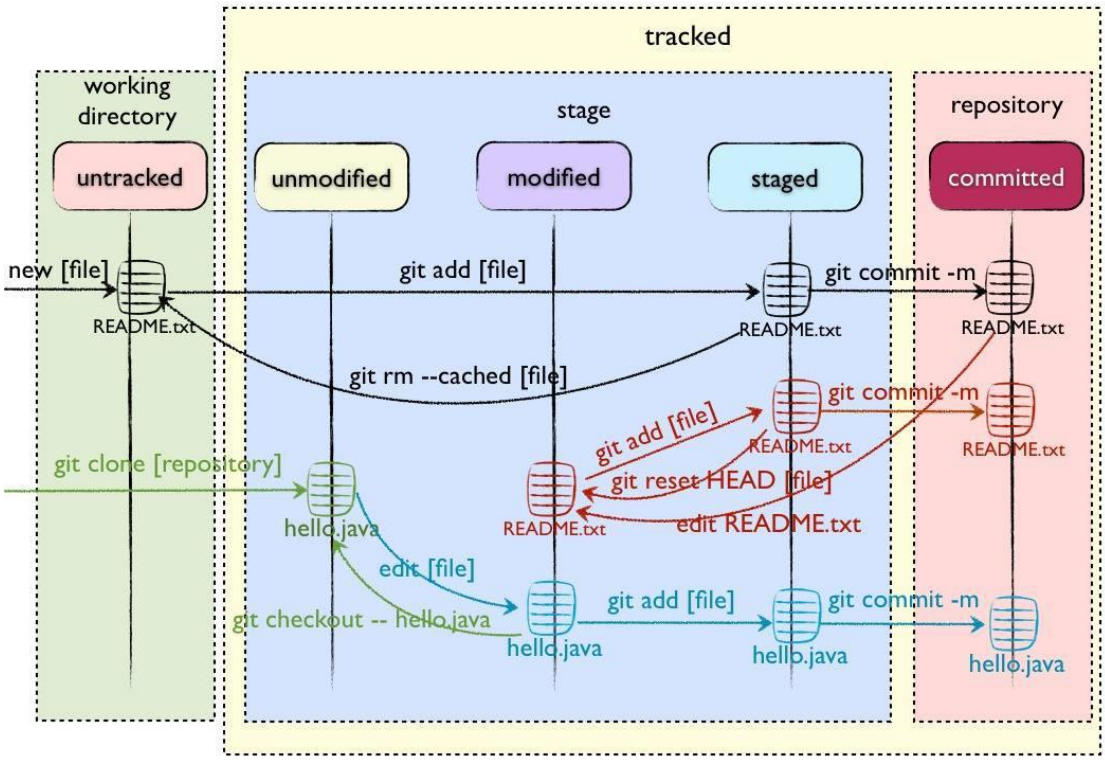
(2). Unmodify: 文件已经入库，未修改，即版本库中的文件快照内容与文件夹中完全一致. 这种类型的文件有两种去处，如果它被修改，而变为 Modified.

(3). 如果使用 git rm 移出版本库，则成为 Untracked 文件。

(4). Modified: 文件已修改，仅仅是修改，并没有进行其他的操作. 这个文件也有两个去处，通过 git add 可进入暂存 staged 状态，使用 git checkout 则丢弃修改过，返回到 unmodify 状态，这个 git checkout 即从库中取出文件，覆盖当前修改。

(5). Staged: 暂存状态. 执行 git commit 则将修改同步到库中，这时库中的文件和本地文件又变为一致，文件为 Unmodify 状态. 执行 git reset HEAD filename 取消暂存,文件状态为 Modified。

下面的图很好的解释了这四种状态的转变：



(1). 新建文件--->Untracked；

(2). 使用 add 命令将新建的文件加入到暂存区--->Staged；

(3). 使用 commit 命令将暂存区的文件提交到本地仓库--->Unmodified；

(4). 如果对 Unmodified 状态的文件进行修改---> modified；

(5). 如果对 Unmodified 状态的文件进行 remove 操作--->Untracked。

2.1.4. 四个区域常用命令

2.1.4.1. 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]
```

2.1.4.2. 查看文件状态

```
#查看指定文件状态
git status [filename]
#查看所有文件状态
git status
```

2.1.4.3. 工作区<-->暂存区

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 .

#当我们需要删除暂存区或分支上的文件, 同时工作区也不需要这个文件了, 可以使用

git rm file_path

#当我们需要删除暂存区或分支上的文件, 但本地又需要使用, 这个时候直接 push 那边这个文件就没有，如果 push 之前重新 add 那么还是会有。

git rm --cached file_path

#直接加文件名　从暂存区将文件恢复到工作区，如果工作区已经有该文件，则会选择覆盖

#加了【分支名】 +文件名　则表示从分支名为所写的分支名中拉取文件并覆盖工作区里的文件

git checkout

2.1.4.4. 工作区<-->资源库（版本库）

#将暂存区-->资源库（版本库）

git commit -m '该次提交说明'

#如果出现:将不必要的文件 commit 或者 上次提交觉得是错的　或者 不想改变暂存区内容，只是想调整提交的信息

#移除不必要的添加到暂存区的文件

git reset HEAD 文件名

#去掉上一次的提交（会直接变成 add 之前状态）

git reset HEAD^

#去掉上一次的提交（变成 add 之后，commit 之前状态）

git reset --soft　HEAD^

2.1.4.5. 远程操作

取回远程仓库的变化，并与本地分支合并

git pull

上传本地指定分支到远程仓库

git push

2.1.4.6. 其它常用命令

```
# 显示当前的 Git 配置
git config --list
# 编辑 Git 配置文件
git config -e [--global]
#初次 commit 之前，需要配置用户邮箱及用户名，使用以下命令：
git config --global user.email "you@example.com"
git config --global user.name "Your Name"
#调出 Git 的帮助文档
git --help
#查看某个具体命令的帮助文档
git +命令  --help
#查看 git 的版本
git --version
```

2.2. 其它术语

(1). **索引（Index）**
索引是暂存区的另一种术语。
(2). **签入（Checkin）**
将新版本复制回仓库
(3). **签出（Checkout）**
从仓库中将文件的最新修订版本复制到工作空间
(4). **提交（Commit）**
对各自文件的工作副本做了更改，并将这些更改提交到仓库
(5). **冲突（Conflict）**
多人对同一文件的工作副本进行更改，并将这些更改提交到仓库
(6). **合并（Merge）**
将某分支上的更改联接到此主干或同为主干的另一个分支
(7). **分支（Branch）**
从主线上分离开的副本，默认分支叫 master
(8). **锁（Lock）**
获得修改文件的专有权限。
(9). **头（HEAD）**
头是一个象征性的参考，最常用以指向当前选择的分支。
(10). **修订（Revision）**
表示代码的一个版本状态。Git 通过用 SHA1 hash 算法表示的 ID 来标识不同的版本。

(11). **标记（Tags）**

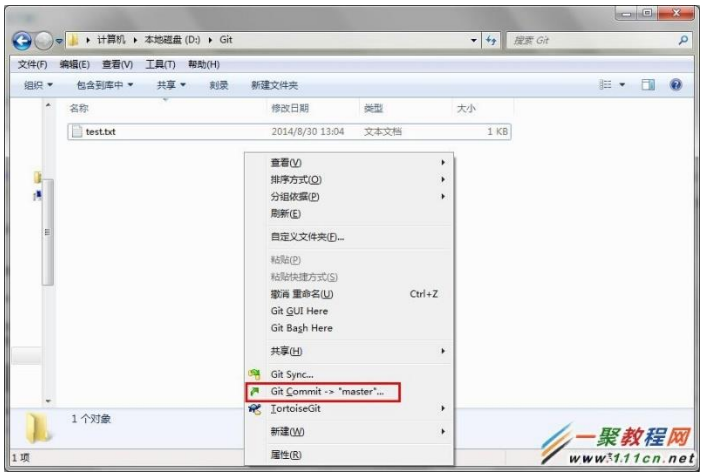
标记指的是某个分支某个特定时间点的状态。通过标记，可以很方便的切换到标记时的状

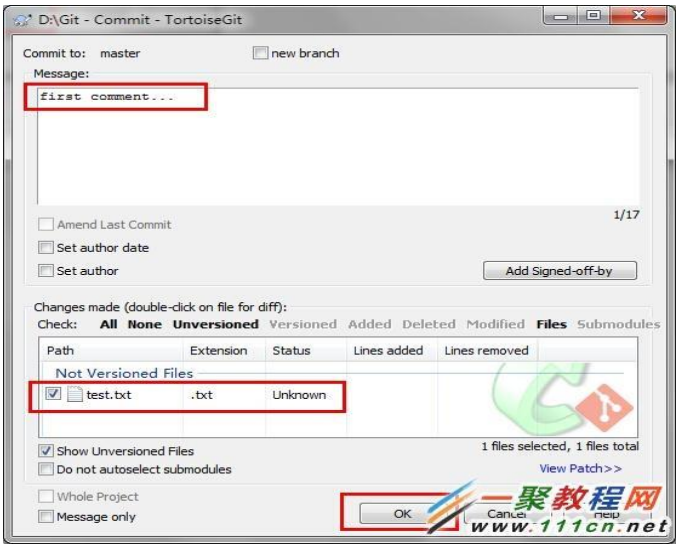
3. 典型操作及菜单概览

3.1. 典型操作

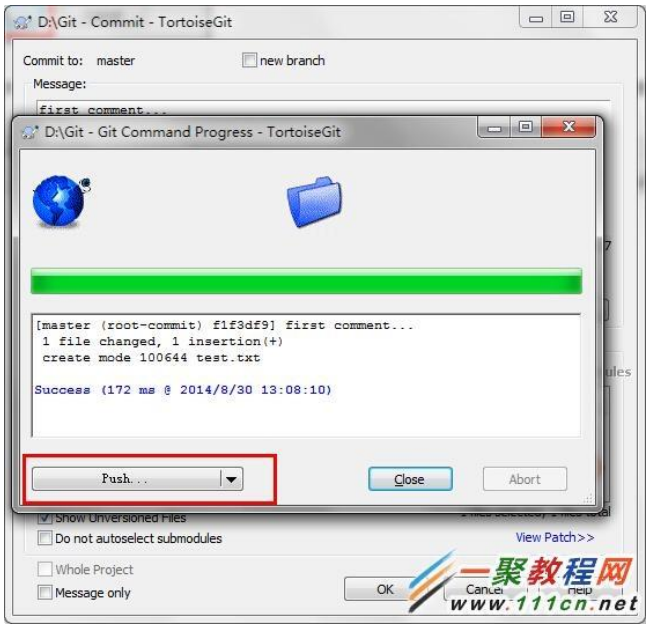
3.1.1. 文件创建及提交

在工作目录下空白处点击右键,选择 Git Commit -> "master" ...,在弹出对话框里输入提示注释,选择要加入的版本控制文件,确定即可提交.



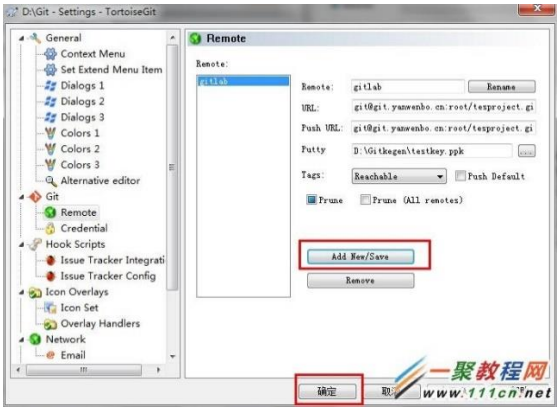


现在只是在本地实现使用 Git 管理项目,在此界面若显示 Success 则本地提交成功,接下来点击 Push..,把我们的改动递交到 Git 服务器上.

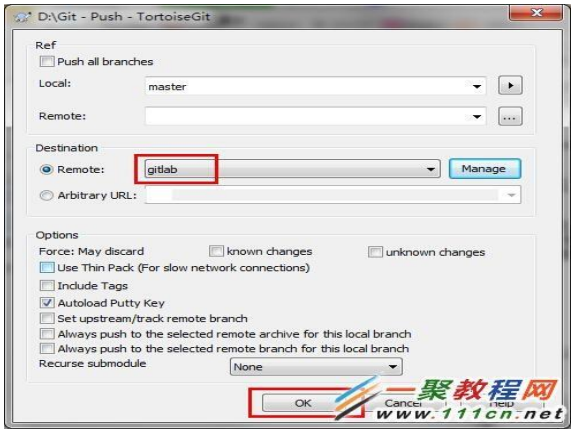


此时会弹出 Push 对话框,在 Ref - Remote 栏里设定当前分支名为 master,然后点击 Destination - Remote 栏的 Manage 按钮.

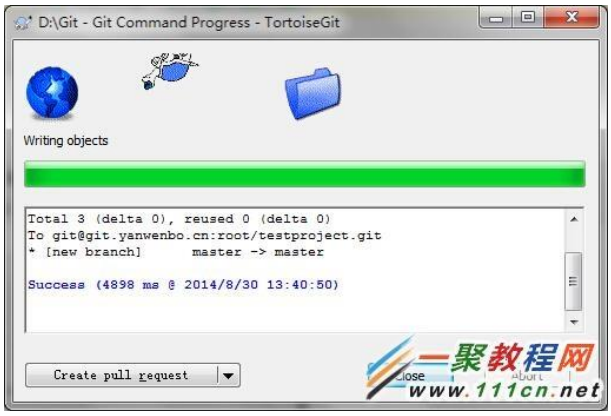
在弹出的设置对话框中按照如图所示,填写服务器 Remote 名称,URL(git@git.yanwenbo.cn:root/tesproject.git), 之前保存的本地私钥 testkey.ppk,点击 Add New/Save 按钮保存这一设置,然后点击确定退出返回之前的对话框.



最终确认提交。



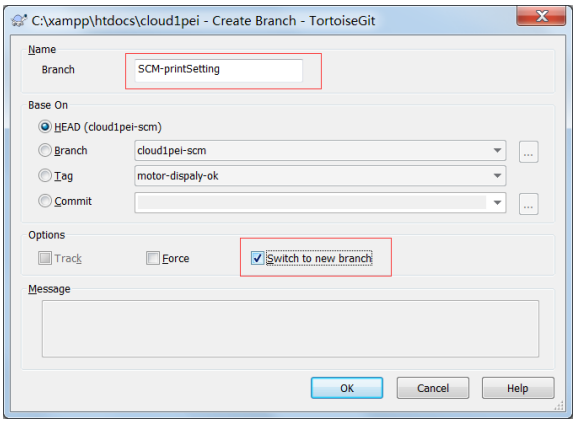
push 成功。



3.1.2. 分支的创建与合并

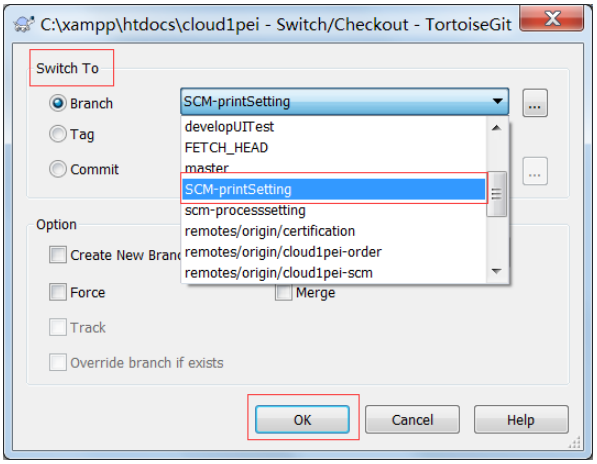
第一步：创建本地分支

点击右键选择 TortoiseGit，选择 Create Branch…，在 Branch 框中填写新分支的名称（若选中”switch to new branch”则直接转到新分支上，省去第二步），点击 OK 按钮。



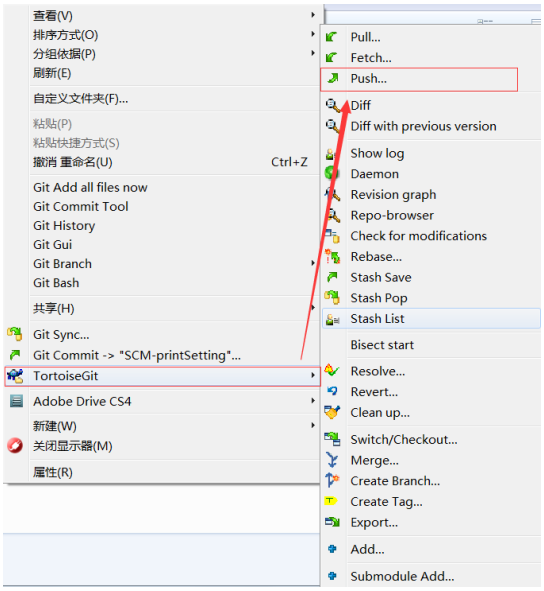
第二步：切换到新创建的分支上

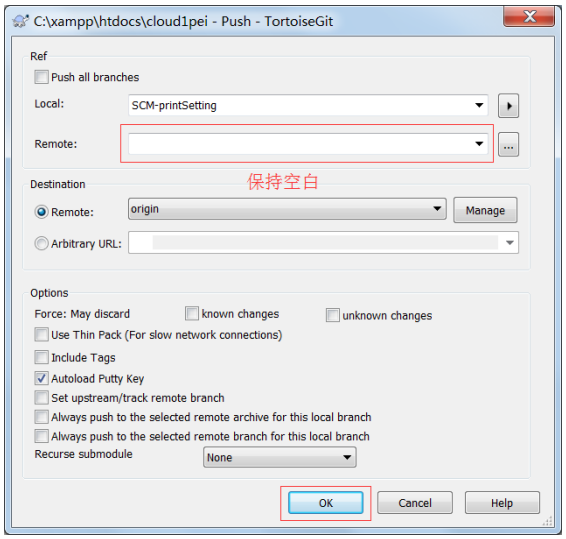
通过“Switch/Checkout”切换到新创建的分支上，点击 OK。

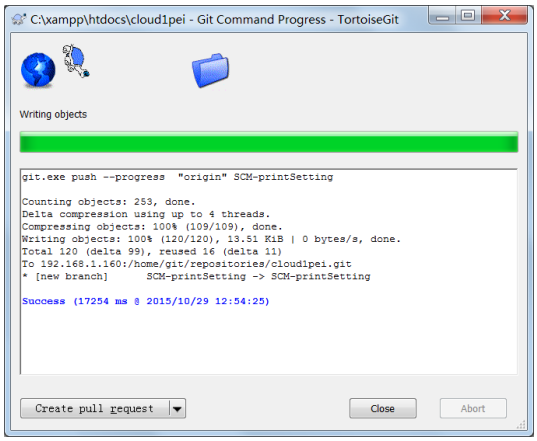


第三步：在新分支下执行 PUSH 操作

在对话框中保持远程分支为空白，点击 OK，则将在远程创建了新的分支（在 PUSH 的时候远程服务器发现远程没有该分支，此时会自动创建一个和本地分支名称一样的分支，并将本地分支的内容上传到该分支）。







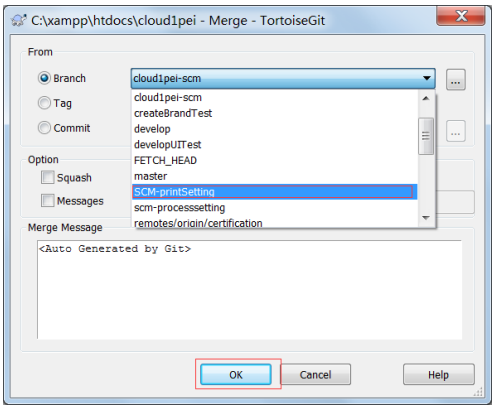
第四步：其他成员切换该新分支

首先进行 pull 操作， 然后进行切换分支（如第二步）。

第五步：分支合并

进行分支合并之前我们需要明确哪个分支将要合并到哪个分支，首先通过“Switch/CheckOut”切换到主干分支（如 develop 分支）,然后通过“Merge”继进行合并操作，在对话框中选择需要合并的分支。

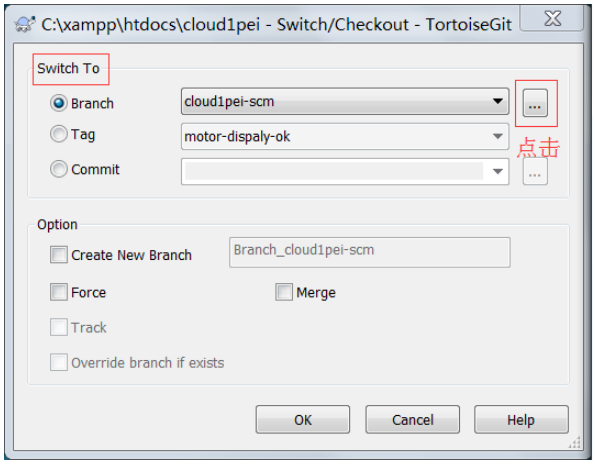
分支合并成功后，我们即可以通过 Commit 与 PUSH 操作将合并上传到中心服务器。

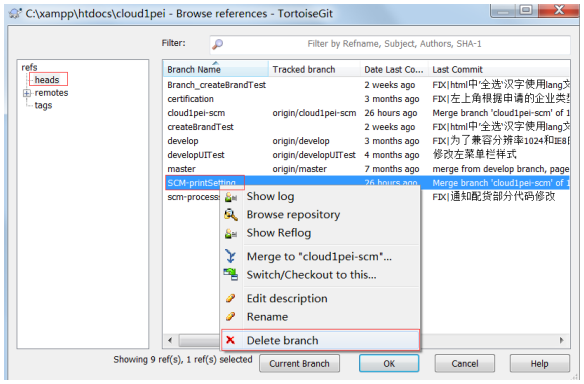


第六步：删除分支

当我们已将新分支合并到主分支后，或者放弃该分支的时候，可以对该分支进行删除操作。

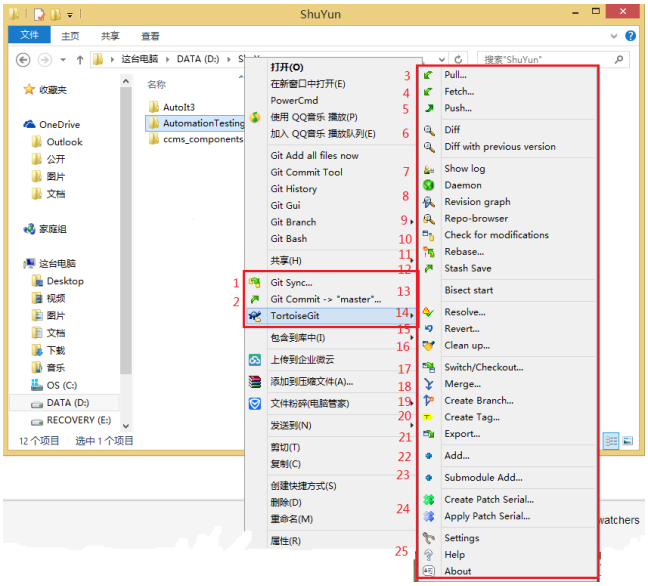
首先通过“CheckOut/Switch”打开对话框，点击 Switch to 区域中 Branch 条目后面的更多按钮，打开分支列表对话框，右键点击要删除的分支，选择 delete branch 进行删除。





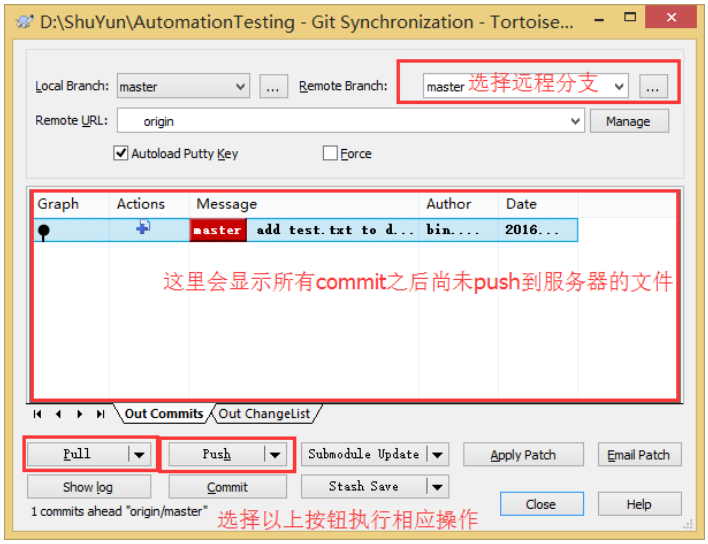
注意，在删除远程分支的时候，本地分支并不会删除，这也说明了本地分支与远程分支并无从属关系。

3.2. TortoiseGit 菜单概览：

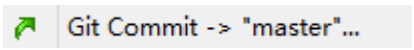


3.2.1. Git Clone(Git 克隆)

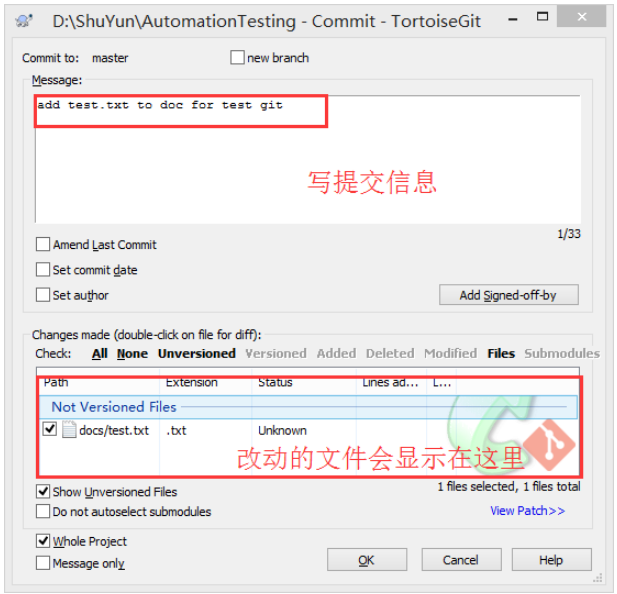
- Git 同步菜单，主要用来跟服务器进行同步操作（pull/push）；
- 也可以在该窗口进行 commit 或查看 log 等操作;

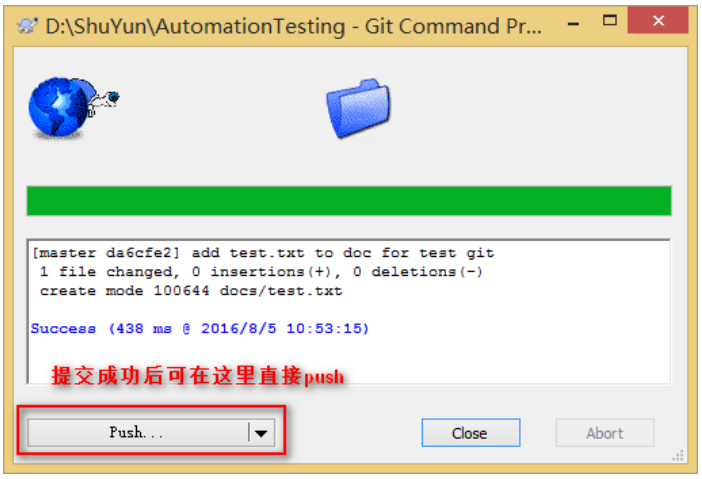


3.2.2. Git Commit->“xxx” (Git 提交)

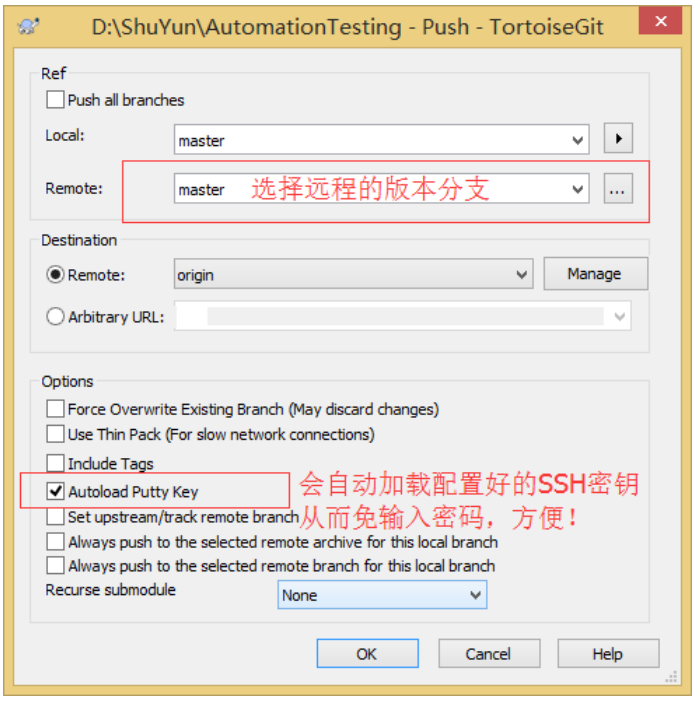


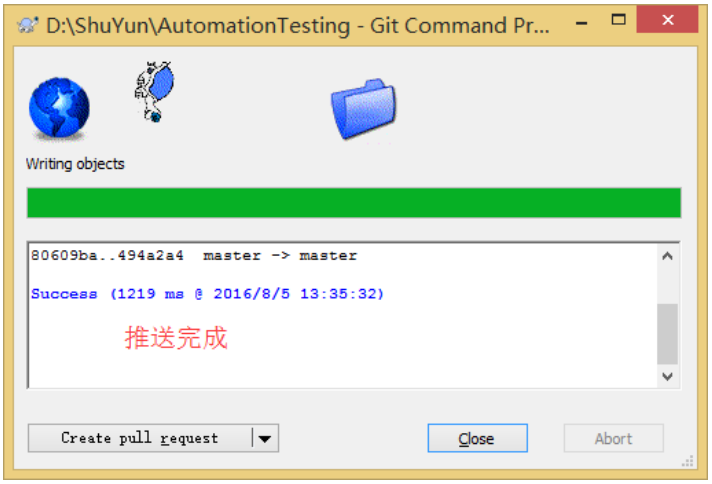
Git 提交工菜单，将本地代码提交到**本地版本库**（默认的分支是 master）。当有文件被改动时，在被改动文件（或者其上层目录）上右键选择此菜单，会弹出提交窗口，如下图：填写信息，点击 OK，提交成功。





p.s.提交成功后，可以直接点击 push，将修改推送到服务器上的远程仓库，如下图：

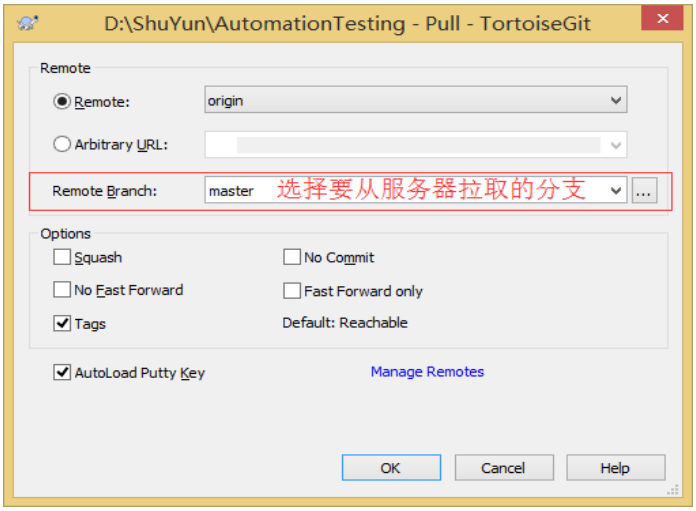




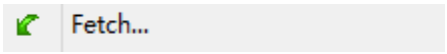
3.2.3. Pull(拉取)

Git 拉取，从服务器上获得更新，这个简单，如下图：

该操作会拉取(fetch)更新到本地仓库并将更新合并(merge)到项目中去，实际上包括 fetch 和 merge 两步操作;



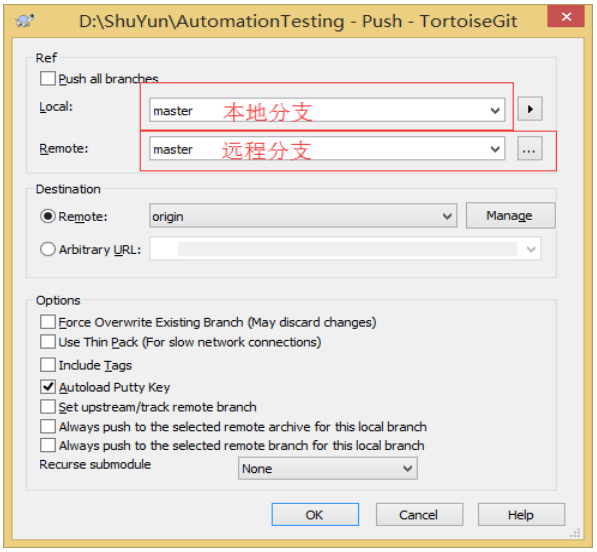
3.2.4. Fetch(获取)



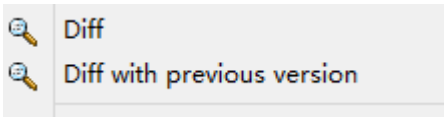
Git 获取，只从服务器获得更新到本地仓库中，并不会合并到项目中去，要合并到项目中去还需要执行 merge 操作，一般不用;

3.2.5. Push(推送)

Git 推送，将提交到本地仓库的修改推送到服务器(远程仓库)，这个简单，如下图：

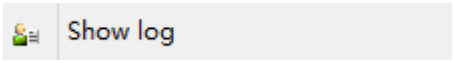


3.2.6. Diff(比较差异)

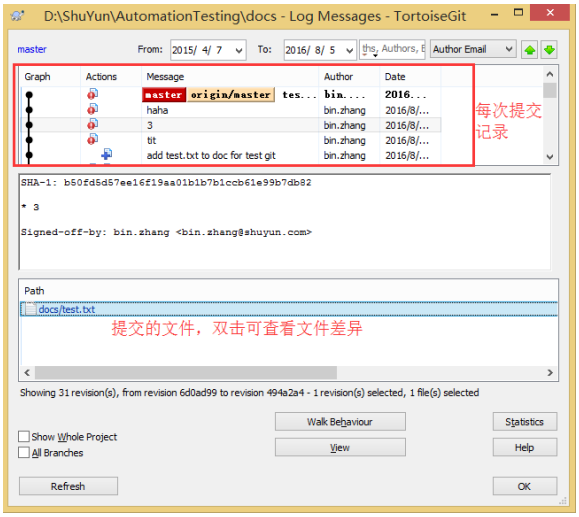


以上两个操作都是用来进行文件对比，一般在需要对比的文件上右键，选择 diff 即可将工作区文件与本地仓库中该文件进行差异对比，主要用来查看改动了啥,不多作介绍;

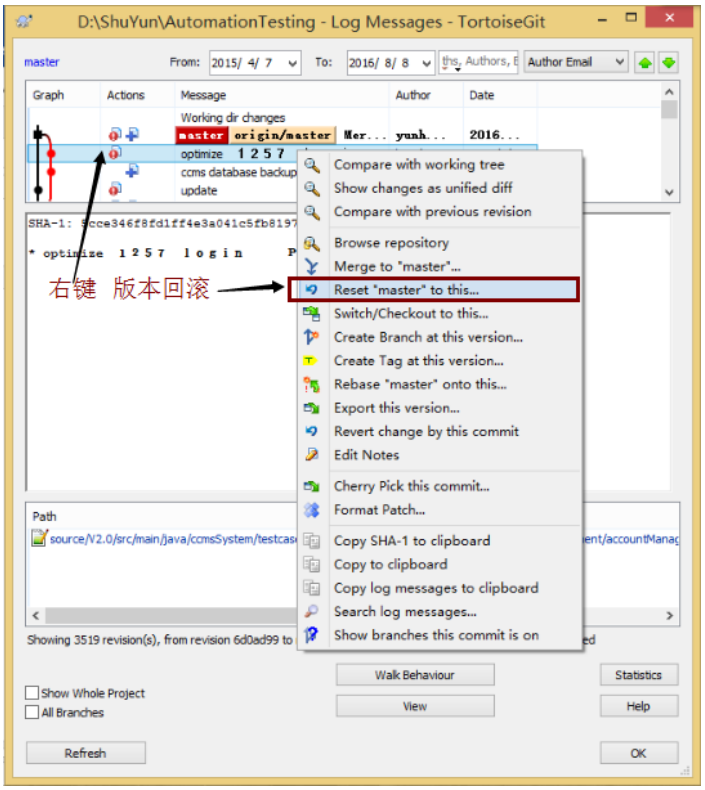
3.2.7. Show Log(显示日志)



查看日志，如下图：



在这里可以选择重置到历史版本，如下图：



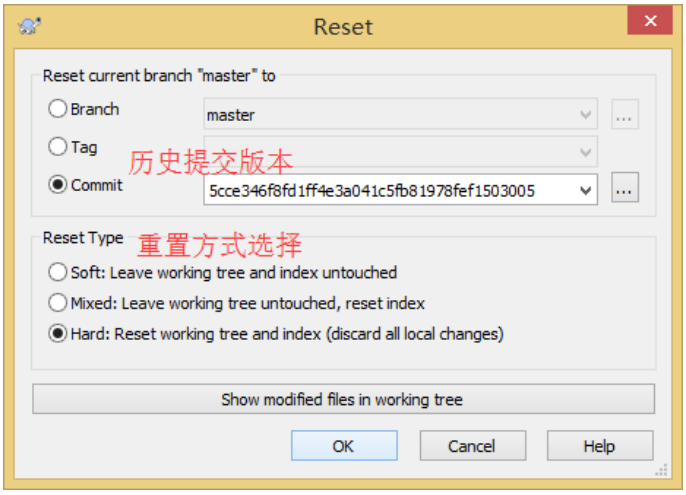
在弹出窗口选择重置方式，点击 OK，即可回到选择的历史版本：

可选择项说明：

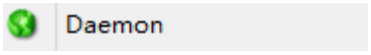
Soft:回到选择的版本，但这个版本之后的所有提交(包括工作区未提交的改动)都会保存;

Mixed:退回到选择的版本，本地仓库也会变为这一版本的内容，但工作区不会变;

Hard:彻底回退到选择的版本，本地仓库也会变为这一版本的内容,工作区所有改动都会丢失;

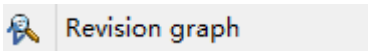


3.2.8. Daemon(后台服务进程)



启动 Git 守护进程，用于快速地将本地存储库与其他人共享，而不使用远程 git 存储库。

3.2.9. **Revision graph**(版本分支图)



如果您需要知道从何处获取分支和标记，可使用该菜单显示分支关系树及显示项目的目录结构。

3.2.10. Check for modifications(检查已修改)

显示被修改的文件;

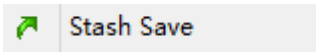
3.2.11. Rebase(变基)

变基或称衍合，这个操作比较复杂，跟 merge 类似，但比 merge 更复杂，更合理;该操作比较复杂，平时也不常用，下面举例作简要说明：

比如当前所分支为 A 分支，使用该菜单来将 B 分支合并进来（AB 都来自 master 分支）,执行过程如下：

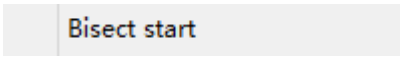
该命令会把当前分支 A 里的每个提交(commit)取消掉，并且把它们临时保存为补丁(patch)(这些补丁放到".git/rebase"目录中),然后把当前分支 A 更新为最新的 B 分支，最后再把保存的这些补丁应用到当前分支 A 上。

3.2.12. Stash Save(贮藏更改)



切换分支时用来保存当前分支尚未提交的修改;

3.2.13. Biset start(二分定位-开始)



很少用。如果您想找出哪个版本引入了 bug，可以使用 bisect 功能。

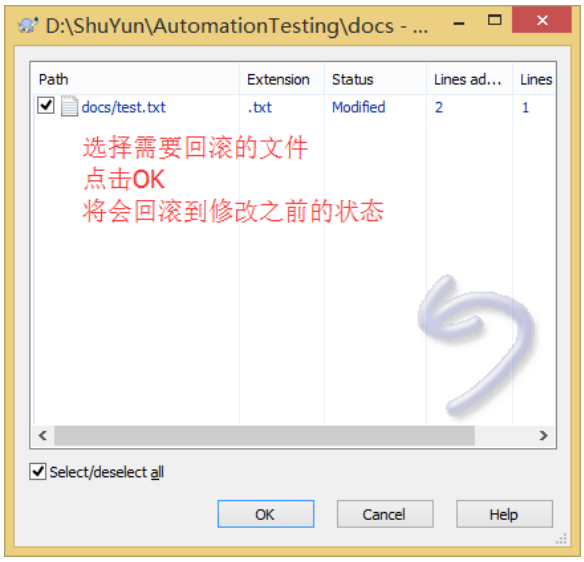
3.2.14. Resolve(解决冲突)

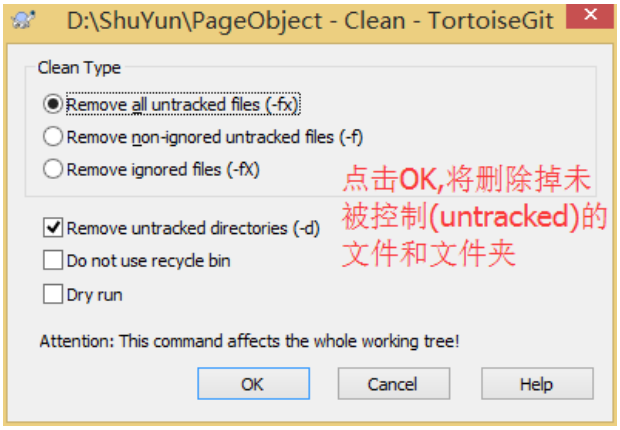
选择处理完冲突的文件，将其标志为解决状态，一般解决完冲突后会直接标志好解决状态，不会再到这里操作;

3.2.15. Revert(还原)

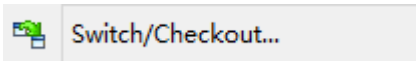
Git 回滚，这个经常用到，在需要回滚的文件(或者其上层目录)上右键，选择该菜单，

操作如下图:

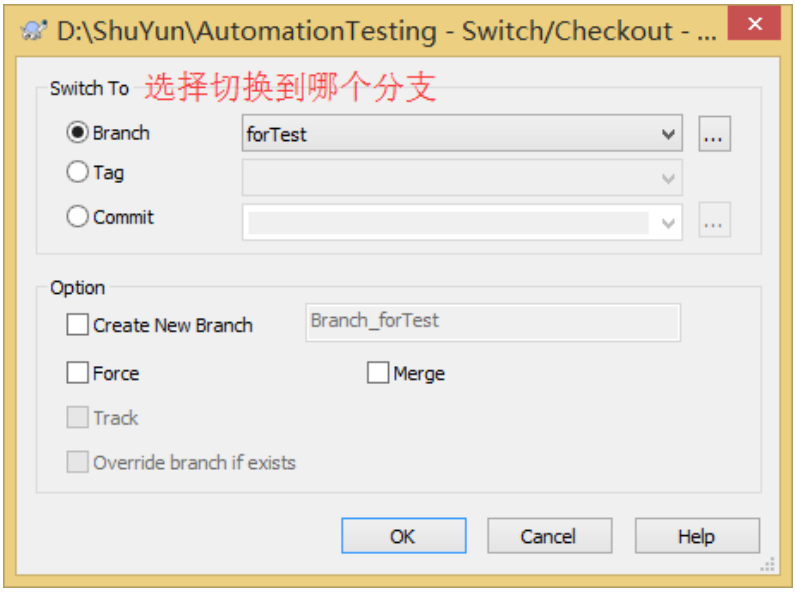


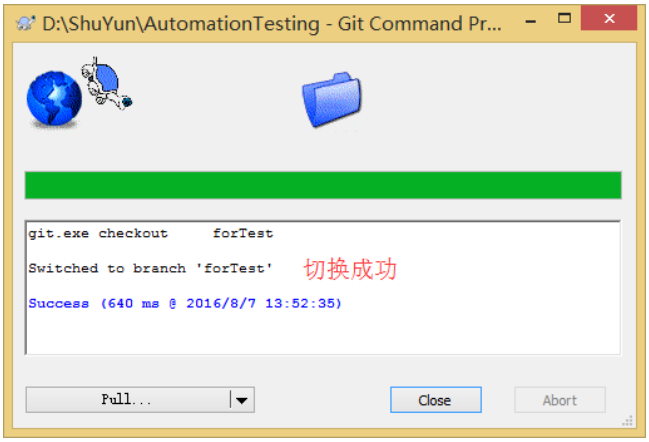


3.2.16. Switch/Checkout(切换/检出)

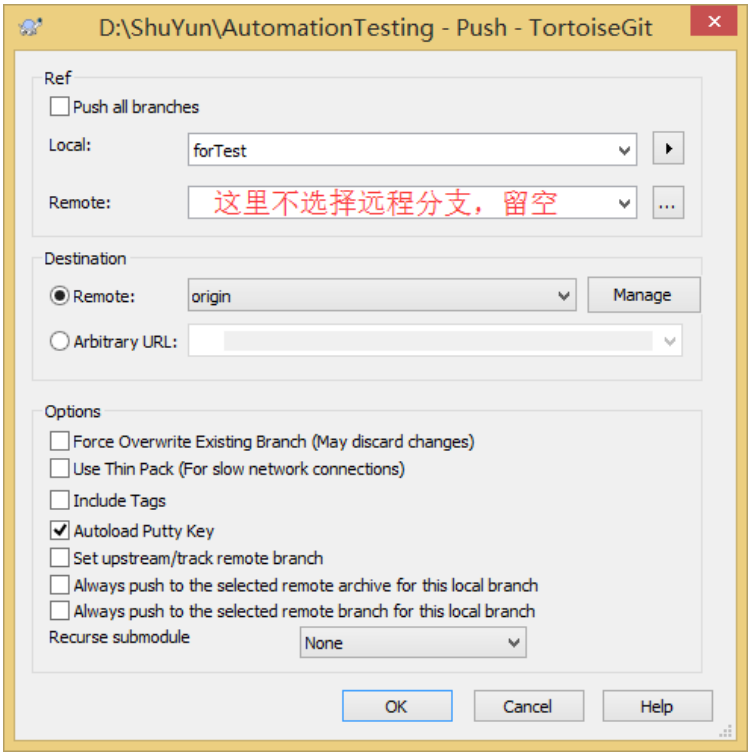


切换分支，当要切换到不同的分支工作时使用该菜单，如下图。切换后所做提交都是针对切换后的分支，原来分支不会受到影响。





新创建分支，并切换到新分支后，此时情况属于远程没有新分支并，本地已经切换到新的分支，若要将本地分支推送到远程服务器(即让服务器端也新增一个分支)，push 操作时可以不选择远程分支，如下图，点击 OK，成功后远程会新增一个分支;



3.2.17. Merge(合并)

Git 合并，可选择项说明：

Squash:勾选则将合并分支的 commit 备注信息也带到当前分支;

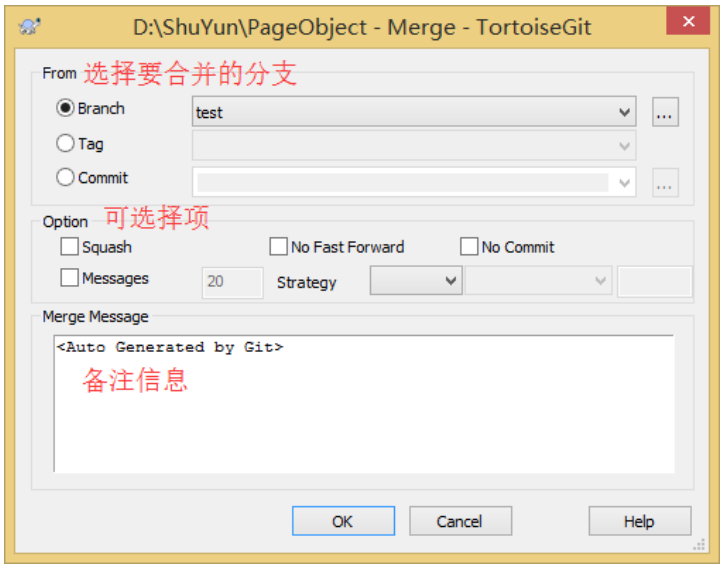
No Fast Forward:非快进式合并（即不会直接把当前分支指向合并分支），合并成功后默认进行提交;(p.s.默认执行”快进式合并”（fast-farwardmerge），直接将当前分支指向合并分支)

NoCommit:合并成功后不提交，默认合并成功后会进行提交;

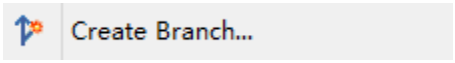
Messages:默认合并成功后提交的备注信息（foranon-fast-forwardmerge);

后面的数字为备注信息的最大长度;

Strategy:合并策略，这个比较复杂的高级操作，我们一般用不着，默认即可;



3.2.18. Create Branch(创建分支)



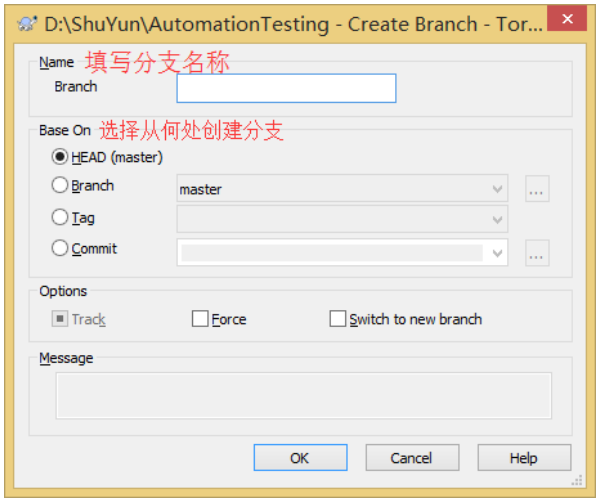
创建分支，填写分支名称，点击 OK 即可，这个简单，如下图：

可选择项说明：

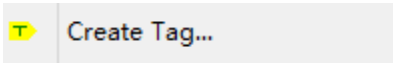
Track:将新创建的本地分支与远程分支建立关联;

Force:强制创建，不论是否存在;

Switchtonewbranch:创建成功后切换到新建的分支;



3.2.19. Create Tag(创建标签)



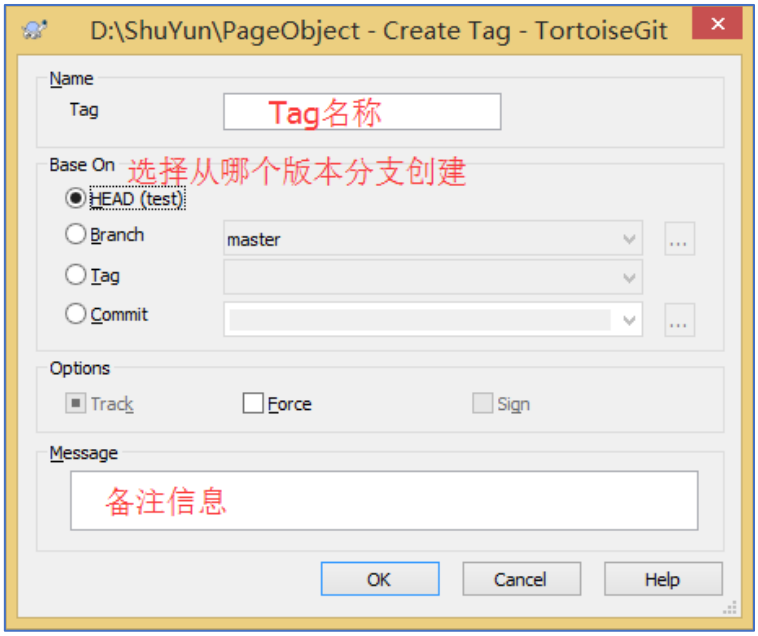
创建标志(里程碑)，填写分支名称，点击 OK 即可，这个简单，如下图：

可选择项说明：

Track:将新创建的 Tag 与远程分支建立关联;

Force:强制创建，不论是否存在;

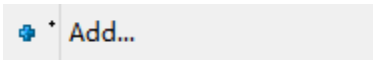
Sign:给标签签名;



3.2.20. Export(导出)

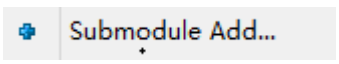
导出项目;

3.2.21. Add(添加)



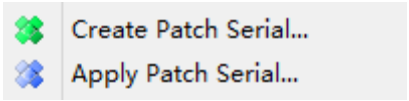
将新增的文件加入版本控制;

3.2.22. Submodule Add(添加子模块)



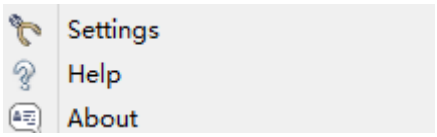
添加子模组，这个功能属于项目中再添加一个 Git 工程依赖，比较复杂,我们用不到;

3.2.23. Create Patch Serial/Apply Patch Serial(创建/应用补丁序列)



Git 补丁，以上两个菜单为创建补丁和应用补丁，就是把自己提交到本地的修改，以补丁的形式发送给别人，别人应用补丁后就能获得自己的修改，这个功能我们用不到;

3.2.24. Settings/Help/About(设置/帮助/关于)



设置，帮助，关于;

4. Git 常用命令

本节讲的命令在 TortoistGit 的菜单中基本都可以找到，如果想使作命令行方式操作 Git，则可参考本节内容。

4.1. 使用 git 恢复未提交的误删数据

不小心将项目中一个文件夹删除还未提交，或者已经提交，此时想要恢复数据该怎么办？

git 记录每次修改 head 的操作，git reflog/git log-g 可以查看所有的历史操作记录，然后通过 git reset 命令进行恢复。

想要将代码恢复到起初时的版本，此时的 head 记录值为"XXXXX"，输入如下命令即可：

git reset--hardXXXXX

4.2. git init

在本地新建一个 repo,进入一个项目目录,执行 git init,会初始化一个 repo,并在当前文件夹下创建一个.git 文件夹.

4.3. git clone

获取一个 url 对应的远程 Git repo,创建一个 localcopy.

一般的格式是 git clone[url].

clone 下来的 repo 会以 url 最后一个斜线后面的名称命名,创建一个文件夹,如果想要指定特定的名称,可以 git clone[url]newname 指定.

4.4. git status

查询 repo 的状态.

git status-s:-s 表示 short,-s 的输出标记会有两列,第一列是对 staging 区域而言,第二列是对 working 目录而言.

4.5. git log

showcommithistoryofabranch.

git log--oneline--number:每条 log 只显示一行,显示 number 条.

git log--oneline--graph:可以图形化地表示出分支合并历史.

git logbranchname 可以显示特定分支的 log.

git log--onelinebranch1^branch2,可以查看在分支 1,却不在分支 2 中的提交.^表示排除这个分支(Window 下可能要给^branch2 加上引号).

git log--decorate 会显示出 tag 信息.

git log--author=[authorname]可以指定作者的提交历史.

git log--since--before--until--after 根据提交时间筛选 log.

--no-merges 可以将 merge 的 commits 排除在外.

git log--grep 根据 commit 信息过滤 log:git log--grep=keywords

默认情况下,git log--grep--author 是 OR 的关系,即满足一条即被返回,如果你想让它们是 AND 的关系,可以加上--all-match 的 option.

git log-S:filterbyintroduceddiff.

比如:git log-SmethodName(注意 S 和后面的词之间没有等号分隔).

git log-p:showpatchintroducedateachcommit.

每一个提交都是一个快照(snapshot),Git 会把每次提交的 diff 计算出来,作为一个 patch 显示给你看.

另一种方法是 git show[SHA].

git log--stat:showdiffstatofchangesintroducedateachcommit.

同样是用来看改动的相对信息的,--stat 比-p 的输出更简单一些.

4.6. git add

在提交之前,Git 有一个暂存区(stagingarea),可以放入新添加的文件或者加入新的改动.commit 时提交的改动是上一次加入到 stagingarea 中的改动,而不是我们 disk 上的改动.

git add.

会递归地添加当前工作目录中的所有文件.

4.7. git diff

不加参数的 git diff:

showdiffofunstagedchanges.

此命令比较的是工作目录中当前文件和暂存区域快照之间的差异,也就是修改之后还没有暂存起来的变化内容.

若要看已经暂存起来的文件和上次提交时的快照之间的差异,可以用:

git diff--cached 命令.

showdiffofstagedchanges.

(Git 1.6.1 及更高版本还允许使用 git diff--staged，效果是相同的).

git diffHEAD

showdiffofallstagedorunstatedchanges.

也即比较 wokingdirectory 和上次提交之间所有的改动.

如果想看自从某个版本之后都改动了什么,可以用:

git diff[versiontag]

跟 log 命令一样,diff 也可以加上--stat 参数来简化输出.

git diff[branchA][branchB]可以用来比较两个分支.

它实际上会返回一个由 A 到 B 的 patch,不是我们想要的结果.

一般我们想要的结果是两个分支分开以后各自的改动都是什么,是由命令:

git diff[branchA]…[branchB]给出的.

实际上它是:git diff$(git merge-base[branchA][branchB])[branchB]的结果.

4.8. git commit

提交已经被 add 进来的改动.

git commit-m“thecommitmessage"

git commit-a 会先把所有已经 track 的文件的改动 add 进来,然后提交(有点像 svn 的一次提交,不用先暂存).对于没有 track 的文件,还是需要 git add 一下.

git commit--amend 增补提交.会使用与当前提交节点相同的父节点进行一次新的提交,旧的提交将会被取消.

4.9. git reset

undochangesandcommits.

这里的 HEAD 关键字指的是当前分支最末梢最新的一个提交.也就是版本库中该分支上的最新版本.

git resetHEAD:unstagefilesfromindexandresetpointertoHEAD

这个命令用来把不小心 add 进去的文件从 staged 状态取出来,可以单独针对某一个文件操作:git resetHEAD--filename,这个--也可以不加.

git reset--soft

moveHEADtospecificcommitreference,indexandstagingareuntouched.

git reset--hard

unstagefilesANDundoanychangesintheworkingdirectorysincelastcommit.

使用 git reset—hardHEAD 进行 reset,即上次提交之后,所有 staged 的改动和工作目录的改动都会消失,还原到上次提交的状态.

这里的 HEAD 可以被写成任何一次提交的 SHA-1.

不带 soft 和 hard 参数的 git reset,实际上带的是默认参数 mixed.

总结:

git reset--mixedid,是将 git 的 HEAD 变了(也就是提交记录变了),但文件并没有改变，(也就是 workingtree 并没有改变).取消了 commit 和 add 的内容.

git reset--softid.实际上，是 git reset–mixedid 后,又做了一次 git add.即取消了 commit 的内容.

git reset--hardid.是将 git 的 HEAD 变了,文件也变了.

按改动范围排序如下:

soft(commit)<mixed(commit+add)<hard(commit+add+localworking)

4.10. git revert

反转撤销提交.只要把出错的提交(commit)的名字(reference)作为参数传给命令就可以了.

git revertHEAD:撤销最近的一个提交.

git revert 会创建一个反向的新提交,可以通过参数-n 来告诉 Git 先不要提交.

4.11. git rm

git rmfile:从 staging 区移除文件,同时也移除出工作目录.

git rm--cached:从 staging 区移除文件,但留在工作目录中.

git rm--cached 从功能上等同于 git resetHEAD,清除了缓存区,但不动工作目录树.

4.12. git clean

git clean 是从工作目录中移除没有 track 的文件.

通常的参数是 git clean-df:

-d 表示同时移除目录,-f 表示 force,因为在 git 的配置文件中,clean.requireForce=true,如果不加-f,clean 将会拒绝执行.

4.13. git mv

git rm--cachedorig;mvorignew;git addnew

git stash

把当前的改动压入一个栈.

git stash 将会把当前目录和 index 中的所有改动(但不包括未 track 的文件)压入一个栈,然后留给你一个 clean 的工作状态,即处于上一次最新提交处.

git stashlist 会显示这个栈的 list.

git stashapply:取出 stash 中的上一个项目(stash@{0}),并且应用于当前的工作目录.

也可以指定别的项目,比如 git stashapplystash@{1}.

如果你在应用 stash 中项目的同时想要删除它,可以用 git stashpop

删除 stash 中的项目:

git stashdrop:删除上一个,也可指定参数删除指定的一个项目.

git stashclear:删除所有项目.

4.14. git branch

git branch 可以用来列出分支,创建分支和删除分支.

git branch-v 可以看见每一个分支的最后一次提交.

git branch:列出本地所有分支,当前分支会被星号标示出.

git branch(branchname):创建一个新的分支(当你用这种方式创建分支的时候,分支是基于你的上一次提交建立的).

git branch-d(branchname):删除一个分支.

删除 remote 的分支:

git push(remote-name):(branch-name):deletearemotebranch.

这个是因为完整的命令形式是:

git pushremote-namelocal-branch:remote-branch

而这里 local-branch 的部分为空,就意味着删除了 remote-branch

4.15. git checkout

git checkout(branchname)

切换到一个分支.

git checkout-b(branchname):创建并切换到新的分支.

这个命令是将 git branchnewbranch 和 git checkoutnewbranch 合在一起的结果.

checkout 还有另一个作用:替换本地改动:

git checkout--<filename>

此命令会使用 HEAD 中的最新内容替换掉你的工作目录中的文件.已添加到暂存区的改动以及新文件都不会受到影响.

注意:git checkoutfilename 会删除该文件中所有没有暂存和提交的改动,这个操作是不可逆的.

4.16. git merge

把一个分支 merge 进当前的分支.

git merge[alias]/[branch]

把远程分支 merge 到当前分支.

如果出现冲突,需要手动修改,可以用 git mergetool.

解决冲突的时候可以用到 git diff,解决完之后用 git add 添加,即表示冲突已经被 resolved.

4.17. git tag

tagapointinhistoryasimport.

会在一个提交上建立永久性的书签,通常是发布一个 release 版本或者 ship 了什么东西之后加 tag.

比如:git tagv1.0

git tag-av1.0,-a 参数会允许你添加一些信息,即 makeanannotatedtag.

当你运行 git tag-a 命令的时候,Git 会打开一个编辑器让你输入 tag 信息.

我们可以利用 commitSHA 来给一个过去的提交打 tag:

git tag-av0.9XXXX

push 的时候是不包含 tag 的,如果想包含,可以在 push 时加上--tags 参数.

fetch 的时候,branchHEAD 可以 reach 的 tags 是自动被 fetch 下来的,tagsthataren’treachablefrombranchheadswillbeskipped.如果想确保所有的 tags 都被包含进来,需要加上--tags 选项.

4.18. git remote

list,addanddeleteremoterepositoryaliases.

因为不需要每次都用完整的 url,所以 Git 为每一个 remoterepo 的 url 都建立一个别名,然后用 git remote 来管理这个 list.

git remote:列出 remotealiases.

如果你 clone 一个 project,Git 会自动将原来的 url 添加进来,别名就叫做:origin.

git remote-v:可以看见每一个别名对应的实际 url.

git remoteadd[alias][url]:添加一个新的 remoterepo.

git remoterm[alias]:删除一个存在的 remotealias.

git remoterename[old-alias][new-alias]:重命名.

git remoteset-url[alias][url]:更新 url.可以加上—push 和 fetch 参数,为同一个别名 set 不同的存取地址.

4.19. git fetch

downloadnewbranchesanddatafromaremoterepository.

可以 git fetch[alias]取某一个远程 repo,也可以 git fetch--all 取到全部 repo

fetch 将会取到所有你本地没有的数据,所有取下来的分支可以被叫做 remotebranches,它们和本地分支一样(可以看 diff,log 等,也可以 merge 到其他分支),但是 Git 不允许你 checkout 到它们.

4.20. git pull

fetchfromaremoterepoandtrytomergeintothecurrentbranch.

pull==fetch+mergeFETCH_HEAD

git pull 会首先执行 git fetch,然后执行 git merge,把取来的分支的 headmerge 到当前分支.这个 merge 操作会产生一个新的 commit.

如果使用--rebase 参数,它会执行 git rebase 来取代原来的 git merge.

4.21. git rebase

--rebase 不会产生合并的提交,它会将本地的所有提交临时保存为补丁(patch),放在".git /rebase"目录中,然后将当前分支更新到最新的分支尖端,最后把保存的补丁应用到分支上.

rebase 的过程中,也许会出现冲突,Git 会停止 rebase 并让你解决冲突,在解决完冲突之后,用 git add 去更新这些内容,然后无需执行 commit,只需要:

git rebase--continue 就会继续打余下的补丁.

git rebase--abort 将会终止 rebase,当前分支将会回到 rebase 之前的状态.

4.22. git push

pushyournewbranchesanddatatoaremoterepository.

git push[alias][branch]

将会把当前分支 merge 到 alias 上的[branch]分支.如果分支已经存在,将会更新,如果不存在,将会添加这个分支.

如果有多个人向同一个 remoterepopush 代码,Git 会首先在你试图 push 的分支上运行 git log,检查它的历史中是否能看到 server 上的 branch 现在的 tip,如果本地历史中不能看到 server 的 tip,说明本地的代码不是最新的,Git 会拒绝你的 push,让你先 fetch,merge,之后再 push,这样就保证了所有人的改动都会被考虑进来.

4.23. git reflog

git reflog 是对 reflog 进行管理的命令,reflog 是 git 用来记录引用变化的一种机制,比如记录分支的变化或者是 HEAD 引用的变化.

当 git reflog 不指定引用的时候,默认列出 HEAD 的 reflog.

HEAD@{0}代表 HEAD 当前的值,HEAD@{3}代表 HEAD 在 3 次变化之前的值.

git 会将变化记录到 HEAD 对应的 reflog 文件中,其路径为.git /logs/HEAD,分支的 reflog 文件都放在.git /logs/refs 目录下的子目录中.

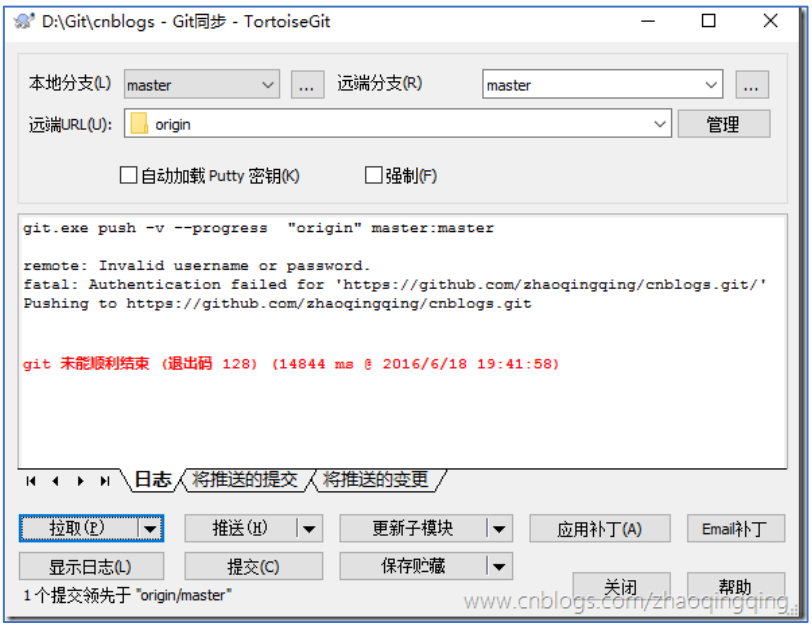
5. 可能遇到的问题

5.1. 推送失败

如果你在推送时遇到失败提示，请仔细查看错误信息，在 git 的错误信息中都会告诉你要怎么做。

5.2. Authentication Failed(验证失败)

如果遇到下方信息，验证失败，有两种解决办法



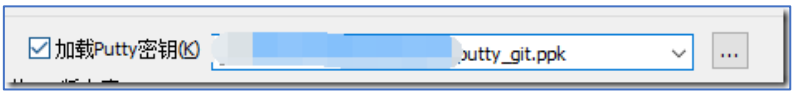
方法一：

在本地机器上创建新的 Putty 密钥，并把密钥添加到 git hub 的后台。

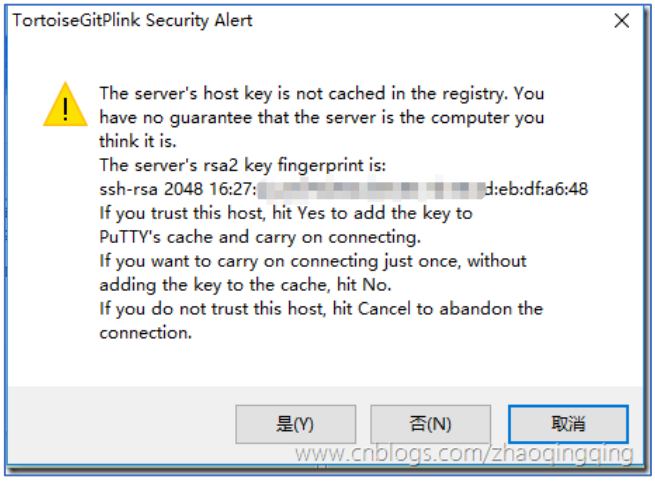
方法二：

检查项目的拉取方式是否是 ssh。

重新克隆此仓库，git clone 时选择 加载 Putty 密钥。



在克隆过程中会弹出框，问你是否要将此计算机添加到你的信任设备列表，选择 是。



再次尝试推送，推送框勾选“自动加载 Putty 密钥”

5.3. 提交之后点推送，远端才更新?

这个要从 git 的原理说起，git 是基于分布式管理的

5.4. git clone 太慢怎么办?

http://www.aneasystone.com/archives/2015/08/git-clone-faster.html

方法 1. 推荐 proxychains 代理，或者挂米国 VPS

方法 2. 把 https 改成 http

方法 3. 在 release 中选择 download 而非 clone 完整仓库，缺点：没有.git 本地仓库，不能 commit，push

方法 4. git clone --depth=1 ,这样只 clone 当前最新的 commit 版本，缺点：没有.git 本地仓库，不能 commit，push

方法 5. 在国内同类代码托管网站查找是否有同样的源码，从国内镜像下载。

参考资料：clone 一个 github 上庞大的代码库，每次 clone 到一半就中断

5.5. 如何断点继传

使用 TortoiseGit 在拉取目录点击鼠标右键 - 同步 - 拉取。就可以继续上次的拉取工作