# Git

## Git简介

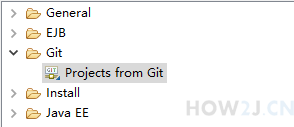
免费开源分布式版本控制系统，用于团队协作开发项目。做到高效，便捷，快速，处理或大或小的项目。

账号注册**1291504564@qq.com**

国外常用的是: <http://github.com>

## Git pull项目

菜单-File->import->Git->Projects from Git

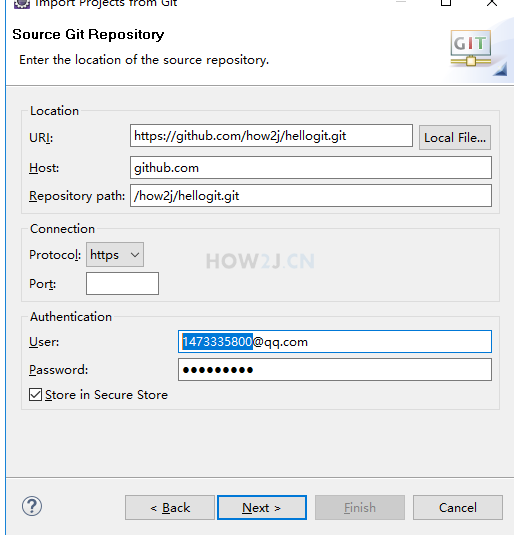


这表示使用某个服务器上的git资源



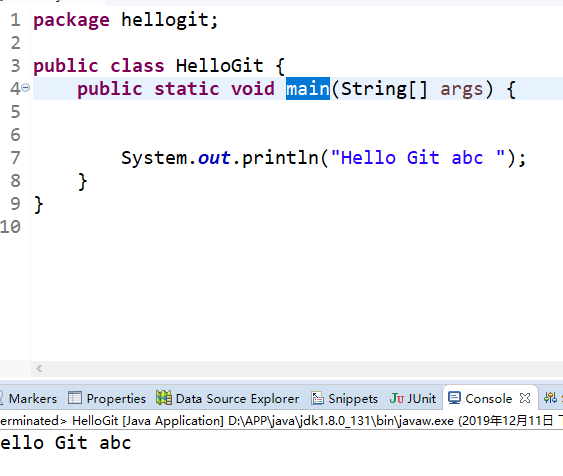
输入git

参数 URI: https://github.com/how2j/hellogit.git  
Host: github.com  
Respository path: /how2j/hellogit.git  
  
按理说只要URI输入了地址，后面两个参数就会自动生成  
然后输入账号密码  
最后Store in Secure Store勾上，表示Eclipse会保持你的账号密码。



接下基本点击next下一步

运行main

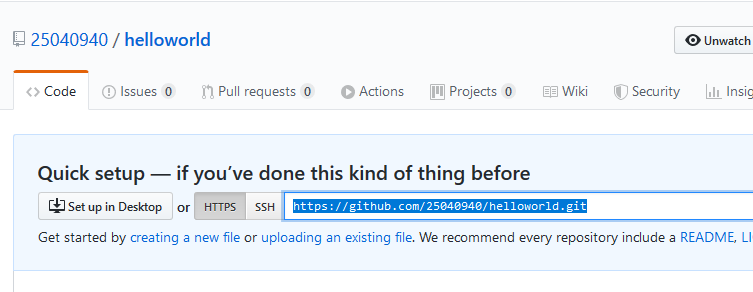


## Git 创建项目

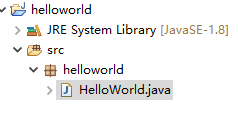
将自己项目共享到git，

### Git创建项目

首先登陆github.com  
然后点击右上角账号左边的加号，点击New repository创建仓库。  
Git上仓库就相当于项目的意思



在Eclipse中创建一个java项目

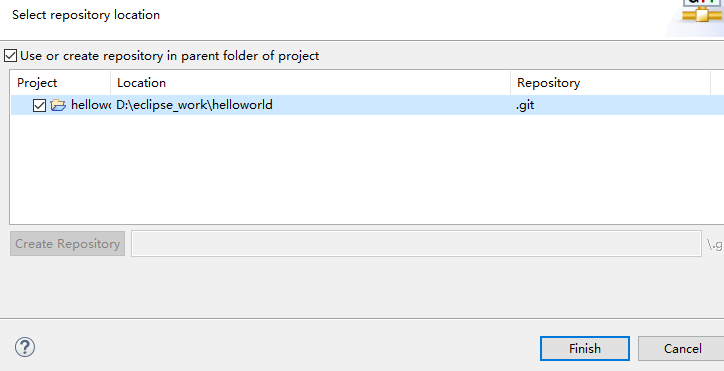


### Share Project

项目右击 team Share Project

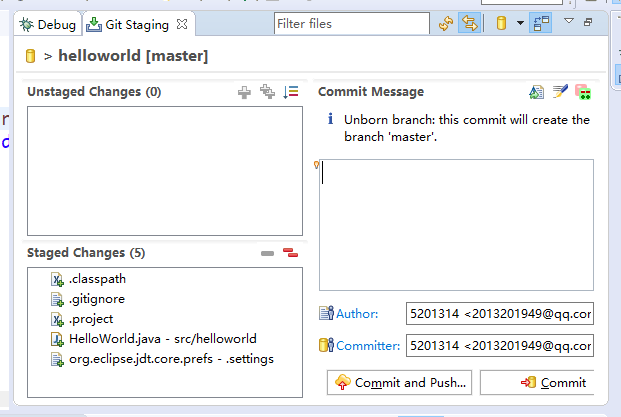
### 配置仓库

依次选中，在创建仓库

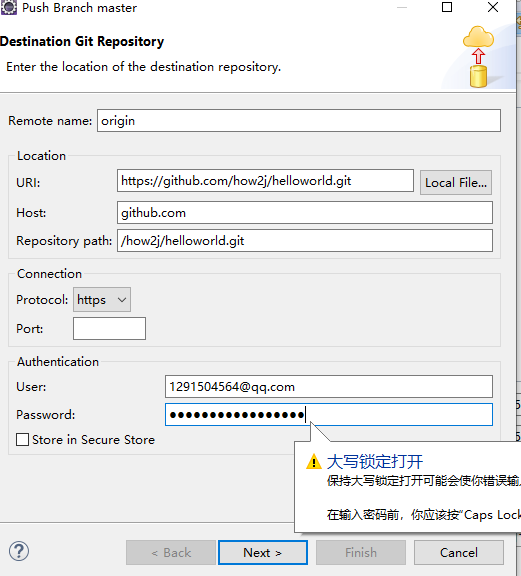


### 提交项目

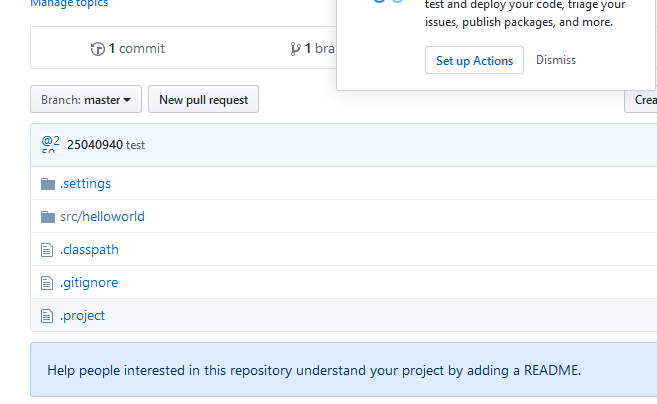
右键项目->Team->Commit



### 设置提交信息

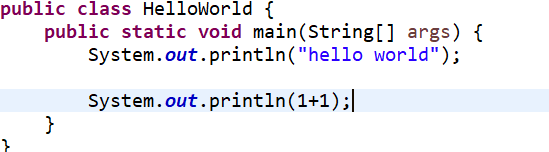


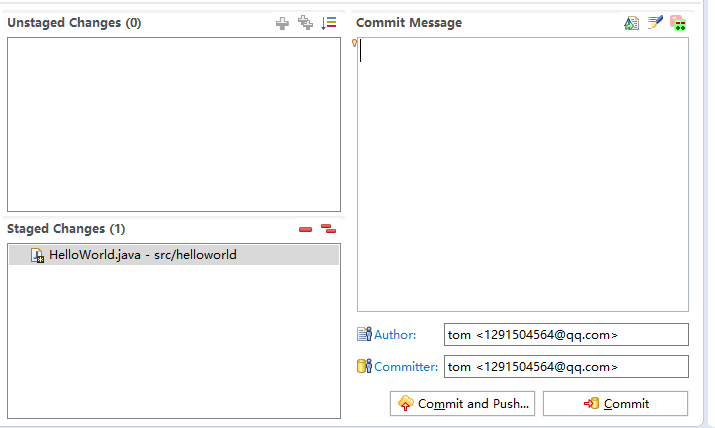
提交完成



## 修改自己的代码

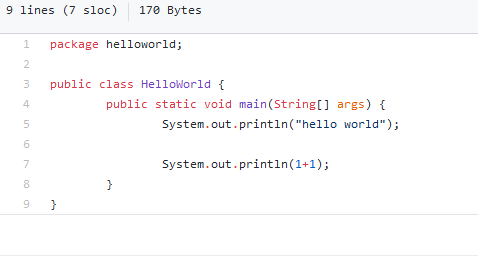
修改代码





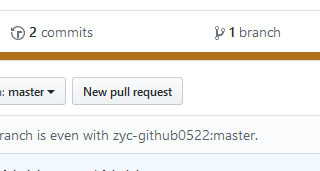
提交已经修改代码。

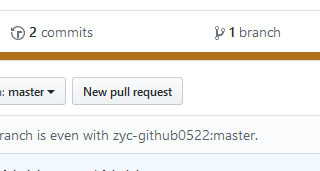
修改成功

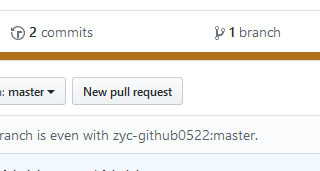


## 修改别人代码

访问别人的资源 fuck 支插 再new pull request 对项目主代码进行修改，在发送给对方你修改过的文件等待对方同意即可







# 一小时学会Git

### Git分布式版本控制

优点：

* 适合分布式开发，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：**

* 模式上比SVN更加复杂。
* 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。

Git下载<https://git-scm.com/>

安装git 打开git-bash.exxe

### Bash基本操作指令

cd 改变目录

pwd显示当前目录路径

ls 列出当前目录下所有文件 ll更加详细

touch : 新建一个文件 如 touch index.js 就会在当前目录下新建一个index.js文件

rm: 删除一个文件, rm index.js 就会把index.js文件删除。

mkdir: 新建一个目录,就是新建一个文件夹

rm -r : 删除一个文件夹, rm -r src 删除src目录， 好像不能用通配符。

mv 移动文件, mv index.html src index.html 是我们要移动的文件, src 是目标文件夹,当然, 这样写,必须保证文件和目标文件夹在同一目录下。

reset 重新初始化终端/清屏。

clear 清屏

history 查看命令历史

help 帮助

exit 退出

#表示注释

输出与注释

小于号：命令默认从键盘获得的输入，改成从文件，或者其它打开文件以及设备输入

Echo >> 是追加内容

Echo > 是覆盖原有内容

显示文件内容 cat

### Git配置 - git config

使用git config -l 可以查看现在的git环境详细配置

查看不同级别的配置文件：

[复制代码](javascript:void(0);)

#查看系统config

git config --system --list

#查看当前用户（global）配置

git config --global --list

#查看当前仓库配置信息

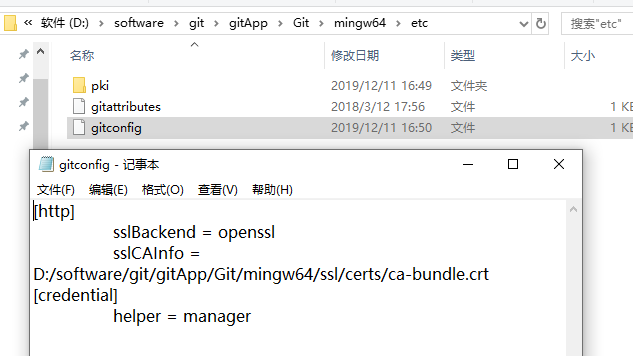
git config --local --list

### Git配置文件分类

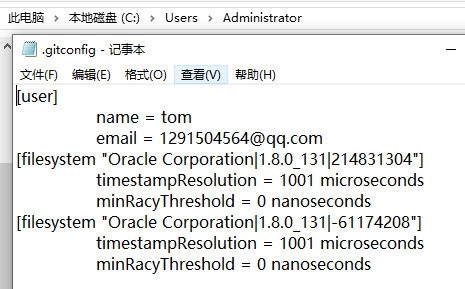
在Windows系统中，Git在$HOME目录中查找.gitconfig文件（一般位于C:\Documents and Settings$USER下）

**Git相关的配置文件有三个**

1）、 /etc/gitconfig：包含了适用于系统所有用户和所有项目的值。(Win：C:\Program Files\Git\mingw64\etc\gitconfig) --system 系统级



2）、~/.gitconfig：只适用于当前登录用户的配置。(Win：C:\Users\Administrator\.gitconfig)  --global 全局



3）、位于git项目目录中的.git/config：适用于特定git项目的配置。(Win：C:\gitProject) --local当前项目

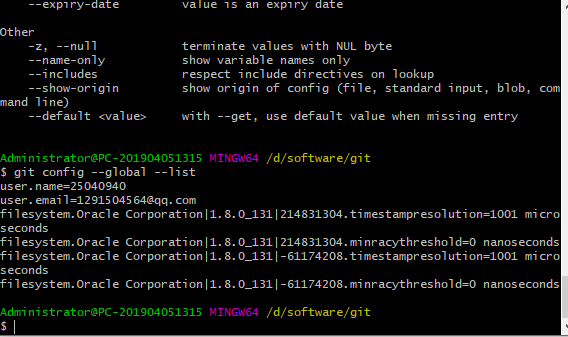
注意：对于同一配置项，三个配置文件的优先级是1<2<3

### 设置用户名与邮箱（用户标识，必要）

当你安装Git后首先要做的事情是设置你的用户名称和e-mail地址。这是非常重要的，因为每次Git提交都会使用该信息。它被永远的嵌入到了你的提交中：

　　$ git config --global user.name **"25040940"** #名称

　　$ git config --global user.email **1291504564@qq.com** #邮箱



### 添加或删除配置项

**1）、添加配置项**

git config [--local|--global|--system] section.key value

[--local|--global|--system] #可选的，对应本地，全局，系统不同级别的设置，请看2.3.2

section.key #区域下的键

value #对应的值

--local 项目级

--global 当前用户级

--system 系统级

**删除配置项**

git config [--local|--global|--system] --unset section.key

 将系统级的height配置项移除

### 更多配置项

[复制代码](javascript:void(0);)

git config --global color.ui true #打开所有的默认终端着色

git config --global alias.ci commit #别名 ci 是commit的别名

[alias]

co = checkout

ci = commit

st = status

pl = pull

ps = push

dt = difftool

l = log --stat

cp = cherry-pick

ca = commit -a

b = branch

user.name #用户名

user.email #邮箱

core.editor #文本编辑器

merge.tool #差异分析工具

core.paper **"less -N"** #配置显示方式

color.diff true #diff颜色配置

alias.co checkout #设置别名

git config user.name #获得用户名

git config core.filemode false #忽略修改权限的文件

[复制代码](javascript:void(0);)

所有config命令参数

[复制代码](javascript:void(0);)

语法: git config [<options>]

文件位置

--global #use global config file 使用全局配置文件

--system #use system config file 使用系统配置文件

--local #use repository config file 使用存储库配置文件

-f, --file <file> #use given config file 使用给定的配置文件

--blob <blob-id> #read config from given blob object 从给定的对象中读取配置

动作

--get #get value: name [value-regex] 获得值：[值]名[正则表达式]

--get-all #get all values: key [value-regex] 获得所有值：[值]名[正则表达式]

--get-regexp #get values for regexp: name-regex [value-regex] 得到的值根据正则

--get-urlmatch #get value specific for the URL: section[.var] URL 为URL获取特定的值

--replace-all #replace all matching variables: name value [value\_regex] 替换所有匹配的变量：名称值[ value\_regex ]

--add #add a new variable: name value 添加一个新变量：name值

--unset #remove a variable: name [value-regex] 删除一个变量名[值]：正则表达式

--unset-all #remove all matches: name [value-regex] 删除所有匹配的正则表达式：名称[值]

--rename-section #rename section: old-name new-name 重命名部分：旧名称 新名称

--remove-section #remove a section: name 删除部分：名称

-l, --list #list all 列出所有

-e, --edit #open an editor 打开一个编辑器

--get-color #find the color configured: slot [default] 找到配置的颜色：插槽[默认]

--get-colorbool #find the color setting: slot [stdout-is-tty] 发现颜色设置：槽[ stdout是TTY ]

类型

--bool #value is "true" or "false" 值是“真”或“假”。

--int #value is decimal number 值是十进制数。

--bool-or-int #value is --bool or --int 值--布尔或int

--path #value is a path (file or directory name) 值是路径（文件或目录名）

其它

-z, --null #terminate values with NUL byte 终止值与null字节

--name-only #show variable names only 只显示变量名

--includes #respect include directives on lookup 尊重包括查找指令

--show-origin #show origin of config (file, standard input, blob, command line) 显示配置（文件、标准输入、数据块、命令行）的来源

# Git理论基础

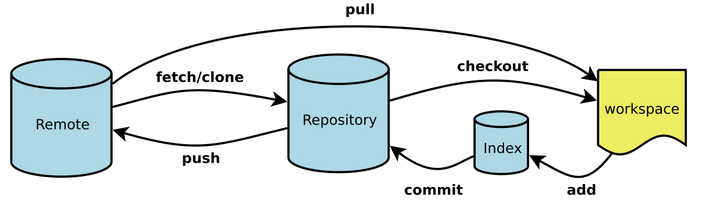
Git本地有三个工作区域：工作目录（Working Directory）、暂存区(Stage/Index)、资源库(Repository或Git Directory)。如果在加上远程的git仓库(Remote Directory)就可以分为四个工作区域。文件在这四个区域之间的转换关系如下：

* Workspace：工作区，就是你平时存放项目代码的地方
* 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
* Repository：仓库区（或本地仓库），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
* 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

## 创建工作目录与常用指令

工作目录（WorkSpace)一般就是你希望Git帮助你管理的文件夹，可以是你项目的目录，也可以是一个空目录，建议不要有中文。



## 获得GIT仓库

创建本地仓库的方法有两种：一种是创建全新的仓库，另一种是克隆远程仓库。

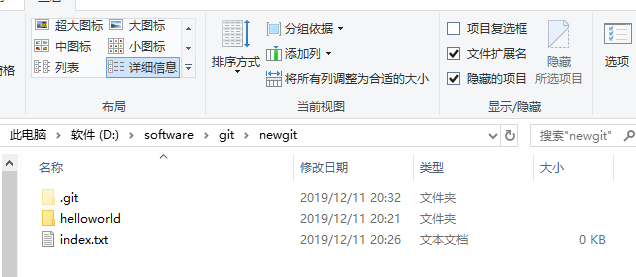
### 创建全新仓库

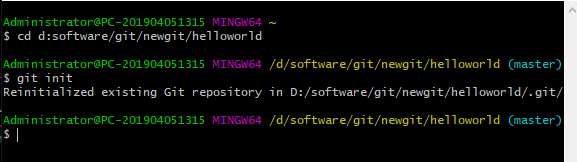
需要用GIT管理的项目的根目录执行：

# 在当前目录新建一个Git代码库

$ git init

记得勾选隐藏的项目





### 克隆远程仓库

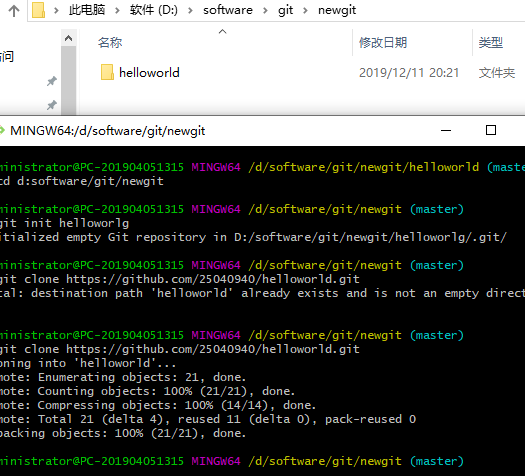
 另一种方式是克隆远程目录，由于是将远程服务器上的仓库完全镜像一份至本地，而不是取某一个特定版本，所以用clone而不是checkout，语法格式如下：

# 克隆一个项目和它的整个代码历史(版本信息)

$ git clone [url]

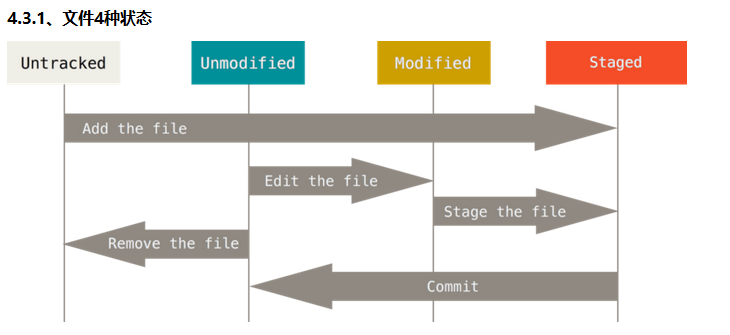
**执行：**

比如我们要从克隆的远程仓库托管在github上，地址为：https://github.com/zhangguo5/SuperPlus.git，这是一个公开的项目



# GIT

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。



* **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.
* **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. 如果使用git rm移出版本库, 则成为Untracked文件
* **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
* **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

### 查看文件状态

上面说文件有4种状态，通过如下命令可以查看到文件的状态：

[复制代码](javascript:void(0);)

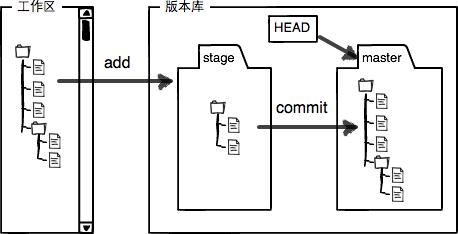
#查看指定文件状态

git status [filename]

#查看所有文件状态

git status

### [复制代码](javascript:void(0);)添加文件与目录



工作区（Working Directory）就是你在电脑里能看到的目录。

版本库（Repository）工作区有一个隐藏目录.git，这个不算工作区，而是Git的版本库。

Git的版本库里存了很多东西，其中最重要的就是称为stage（或者叫index）的暂存区，还有Git为我们自动创建的第一个分支master，以及指向master的一个指针叫HEAD。

将untracked状态的文件添加到暂存区，语法格式如下：

[复制代码](javascript:void(0);)

# 添加指定文件到暂存区

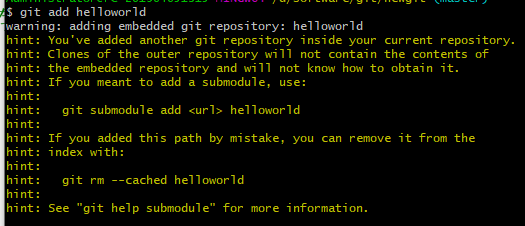
$ git add [file1] [file2] ...

# 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 .

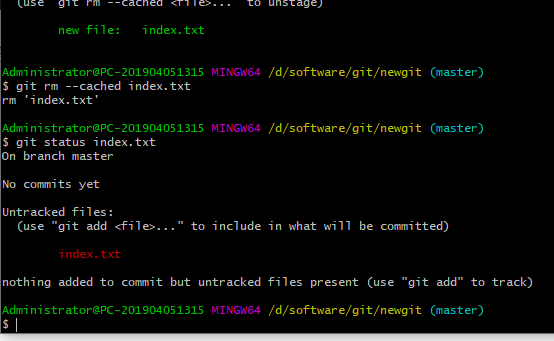


当执行如下命令时，会直接从暂存区删除文件，工作区则不做出改变

#直接从暂存区删除文件，工作区则不做出改变

git rm --cached <file>

当执行 “git reset HEAD” 命令时，暂存区的目录树会被重写，被 master 分支指向的目录树所替换，但是工作区不受影响。



移除所有未跟踪文件

#移除所有未跟踪文件

#一般会加上参数-df，-d表示包含目录，-f表示强制清除。

git clean [options]

### 查看文件修改后的差异

git diff用于显示WorkSpace中的文件和暂存区文件的差异

用"git status"只能查看对哪些文件做了改动，如果要看改动了什么，可以用：

#查看文件修改后的差异

git diff [files]

### ****签出****

如果仓库中已经存在文件f4.txt，在工作区中对f4修改了，如果想撤销可以使用checkout，签出覆盖

检出命令git checkout是git最常用的命令之一，同时也是一个很危险的命令，因为这条命令会重写工作区

语法：

[复制代码](javascript:void(0);)

#用法一

git checkout [-q] [<commit>] [--] <paths>...

#用法二

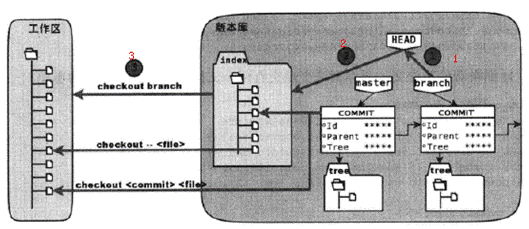
git checkout [<branch>]

#用法三

git checkout [-m] [[-b]--orphan] <new\_branch>] [<start\_point>]

[复制代码](javascript:void(0);)

<commit>是可选项，如果省略则相当于从暂存区（index）进行检出



[复制代码](javascript:void(0);)

$ git checkout branch

#检出branch分支。要完成图中的三个步骤，更新HEAD以指向branch分支，以及用branch 指向的树更新暂存区和工作区。

$ git checkout

#汇总显示工作区、暂存区与HEAD的差异。

$ git checkout HEAD

#同上

$ git checkout -- filename

#用暂存区中filename文件来覆盖工作区中的filename文件。相当于取消自上次执行git add filename以来（如果执行过）的本地修改。

$ git checkout branch -- filename

#维持HEAD的指向不变。用branch所指向的提交中filename替换暂存区和工作区中相 应的文件。注意会将暂存区和工作区中的filename文件直接覆盖。

$ git checkout -- . 或写作 git checkout .

#注意git checkout 命令后的参数为一个点（“.”）。这条命令最危险！会取消所有本地的 #修改（相对于暂存区）。相当于用暂存区的所有文件直接覆盖本地文件，不给用户任何确认的机会！

$ git checkout commit\_id -- file\_name

#如果不加commit\_id，那么git checkout -- file\_name 表示恢复文件到本地版本库中最新的状态。

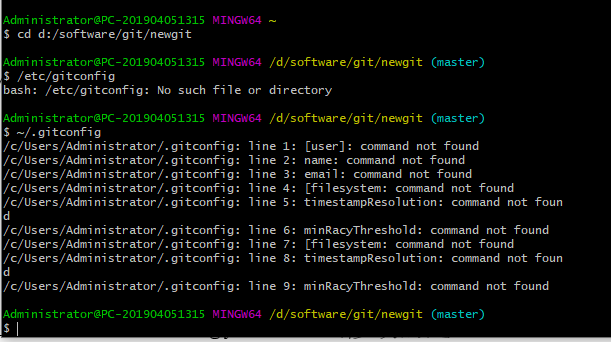
# Git

<https://www.cnblogs.com/wang_yb/p/3867221.html>

常规操作也是我自己平时常用的几个命令, 学自于 pro git 这本书中

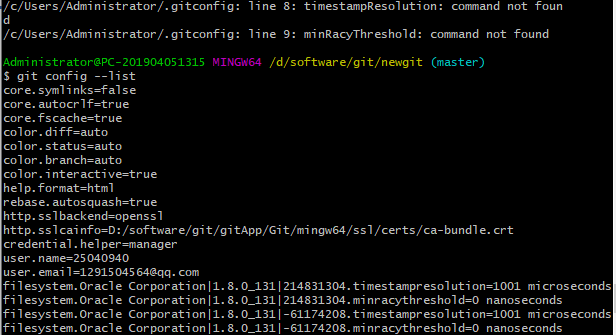
### git 配置文件

1. git的配置文件位置  
   针对所有用户：/etc/gitconfig
2. 针对当前用户： ～/.gitconfig



1. 查看配置的方法

git config –list



1. 修改配置的方法
2. git config --global user.name "wangyubin" (修改的是～/.gitconfig)

git config --system user.name "wangyubin" (修改的是/etc/gitconfig)

### git 基本使用

1. clone现有仓库

git clone URL (URL支持git，ssh，http，https等各种协议)

1. git中文件的各个状态
   * unstaged - git仓库中没有此文件的相关记录
   * modified - git仓库中有这个文件的记录，并且此文件当前有改动
   * staged - 追加,删除或修改的文件被暂时保存，这些追加,删除和修改并没有提交到git仓库
   * commited - 追加或修改的文件被提交到本地git仓库（git仓库中大部分都是这种文件，所以git status不显示这些文件）
2. 查看git仓库中各文件状态

git status

1. 初始化一个仓库

git init

在当前文件夹下生成.git目录，完成初始化，此时此文件夹下的所有文件处于unstaged状态

1. 追加文件

git add test.c

test.c的文件变为staged状态，其他文件还是unstaged状态

5.1 追加文件的结果1 - 恢复为原先状态（变为unstaged）

git rm --cache test.c

5.2 追加文件的结果2 - 提交到git仓库（变为commited）

git commit -m "my message"

1. 修改文件

echo "aaa" >> test.c

test.c的状态变为modified

6.1 修改文件的结果1

git add test.c (暂时保存修改的内容，即变为staged）

下面有2条路可以选择：  
6.1.1 取消刚才的暂时保存

git reset test.c （状态变回modified）

6.2.2 将暂存的修改提交到git仓库

git commit -m "my message"

6.2 修改文件的结果2

git checkout test.c （将test.c恢复为git仓库中的最新版本，即变为commited状态，test.c的内容和5.2节一样）

1. 删除文件  
   7.1 从git仓库和磁盘上删除文件

git rm test.c （当前目录中删除了test.c，在git仓库中暂时删除了test.c，相当于staged状态）

7.1.1 从git仓库中删除test.c

git commit -m "my message" (git仓库以后不再维护test.c)

7.1.2 删错了，恢复刚才的操作

git reset HEAD test.c (恢复到删除前的状态，当前目录中已删除的test.c也恢复了，test.c仍文commited状态)

7.2 仅从git仓库中删除文件

git rm --cache test.c （当前目录中没有删除了test.c，仅在git仓库中暂时删除了test.c，相当于staged状态）

7.2.1 从git仓库中删除test.c

git commit -m "my message" (git仓库以后不再维护test.c，但是当前目录中仍然有test.c)

7.2.2 删错了，恢复刚才的操作

git reset HEAD test.c （和7.1.2一样）

7.3 误删除后的恢复  
如果删除了一个文件，并且commit之后发现删错了。也可以恢复，

git log （查看各次的提交信息）

git checkout commit号 （恢复到未删除前的commint号，此时删除的文件也恢复到磁盘上了）

git checkout master (备份好删除的文件后，再回到最新状态)

### git 远程仓库

1. 查看远程仓库  
   1.1 简单查看-所有仓库

git remote (只能查看远程仓库的名字)

1.2 查看更多内容-所有仓库

git remote -v (远程仓库的名字及git地址)

1.3 查看单个仓库的信息

git remote show [remote-name]

1. 新建远程仓库
2. git remote add [shortname] [url]

ex. git remote add mc git://www.host.com/gitdir/mycode.git

1. 修改远程仓库

git remote rename [oldnanme] [newname]

1. 删除远程仓库

git remote rm [remote-name]

1. 远程仓库的数据  
   5.1 获取数据
2. git fetch [remote-name] (获取仓库的所有更新，但是不自动合并当前分支)

git pull (获取仓库的所有更新, 并且自动合并到当前分支)

5.2 上传数据

git push [remote-name] [branch-name]

ex. git push origin master

### git 标签

1. 列出标签  
   1.1 查看所有tag

git tag

1.2 查看某个tag

git show [tag-name]

1. 新建标签  
   2.1 轻量级tag

git tag [tag-name]

2.2 带标注的tag

git tag -a [tag-name] -m "tag message"

2.3 后期追加tag

git log --pretty=oneline (查看所有的commit号)

git tag -a [tag-name] [commit号前几位即可]

1. 删除标签

git tag -d [tag-name]

1. 提交标签到远程仓库
2. git push [remote-name] --tags

ex. git push origin --tags

### git 分支

1. 查看和切换分支
2. git branch (查看所有的分支及当前处于哪个分支)
3. git branch -v (查看所有的分支的详细信息)
4. git branch --merged (查看已经合并的分支)
5. git branch --no-merged (查看还没合并的分支)

git checkout [branch-name] (切换到某个分支)

1. 新建分支
2. git branch [branch-name] (新建一个分支)

git branch -b [branch-name] (新建一个分支并切换到这个分支上)

1. 合并分支
2. git merge [branch-name]
3. ex. 将分支btest合并到主分支master
4. git checkout master

git merge btest

merge时有冲突的文件会列出来，需要手动合并

将冲突手动解决后，再次用 git status来查看是否还有 unmerged的文件。  
如果没有冲突的文件，就可以 git commit 来提交这次合并了。

1. 删除分支
2. git branch -d [branch-name]

或者 git branch -D [branch-name] (强制删除某个还未合并的分支)

1. 远程分支相关  
   5.1 新建远程分支
   1. git checkout [local\_branch] (首先进入想要上传的分支)
   2. git remote add [remote\_repo] [remote\_branch]  
      (这里的[remote\_branch]是远程分支的名字，一般和[local\_branch]同名,  
      [remote\_repo]是远程仓库的名字)

5.2 向远程分支推送数据

git push [remote\_repo] [remote\_branch]

5.3 删除远程分支

git push [remote\_repo] :[remote\_branch] (注意远程分支前有个":")

1. 合并分支的另一个方法：衍和

衍和可以简化master上的提交记录，使得代码可以方便的回退，  
但是在公共仓库上用衍和有一定的风险。  
衍和我基本用不上，这里就不赘述了。

## 服务器创建 git 仓库, 并将其作为远程仓库

其实 git 是分布式的 SCM. 并不存在谁是服务器, 谁是客户端的问题, 这里所说的服务器上的git仓库, 指的是多人合作开发时, 共用的, 作为最终发布版本的 git 仓库.  
这个 git 仓库就相当于你在 github 上建的仓库, 会将你在各个电脑上做的代码等提交到上面进行统一管理.

1. 服务端 (远程 git 仓库)
   * 生成用于git服务的账户 (一般就用git)
   * groupadd gpxxx

useradd -m -g gpxxx gitxxx

* + 初始化服务端的git 仓库
  + cd ~/
  + mkdir git-repo
  + cd git-repo
  + mkdir test.git
  + cd test.git

git --bare init

1. 客户端 (本地 git 仓库)
   * 新建本地git 仓库
   * cd ~/gitlocal
   * mkdir test
   * cd test

git init

* + 初始化本地仓库
  + touch README
  + git add README

git commit -m 'first commit for init'

* + 设置git用户信息
  + git config --global user.name "wangyubin"

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

* + 关联远程仓库

git remote add origin gituser@<server address>:~/test.git/

* + 将本地仓库提交到远程

git push origin master

## git 使用中遇到的一些问题

### git pull 时, 远程文件与本地文件有冲突

如果远程的仓库被其他人更新了, 并且更新的内容与我自己本地编辑的内容有冲突. 这时执行 git pull 可能有如下message:

Auto-merging path/to/conflict-file

CONFLICT (content): Merge conflict in path/to/conflict-file

Automatic merge failed; fix conflicts and then commit the result.

用文本编辑器 vim 或者 emacs 之类的来编辑冲突的文件 path/to/conflict-file, 冲突的地方有类似如下的显示

<<<<<<< HEAD

App\_Log.logger.debug(u'开始时间: ' + utils.datetime2str(datetime.datetime.now()))

file = request.FILES.get('file-xxx')

App\_Log.logger.debug(u'结束时间: ' + utils.datetime2str(datetime.datetime.now()))

=======

file = request.FILES.get('xxxx')

>>>>>>> 3602514cc2bf1b3a64470b31ad79e07fe372add5

===== 之上的 <<<<<<< HEAD 是本地的内容  
===== 之下的 >>>>>>> 3602514cc2bf1b3a64470b31ad79e07fe372add5 是远程的内容(这个commit号每次都会不同)  
根据实际情况, 删除多余的内容(包括===== >>>>> <<<<<< 之类的), 修改冲突的地方, 如果以本地的代码为准的话, 会得到如下结果:

App\_Log.logger.debug(u'开始时间: ' + utils.datetime2str(datetime.datetime.now()))

file = request.FILES.get('file-xxx')

App\_Log.logger.debug(u'结束时间: ' + utils.datetime2str(datetime.datetime.now()))

然后 git commit -am '提交的信息' 就解决了冲突.  
最后, 也可以将本地的修改同步到远程 git 仓库: git push

### git pull 时, 本地还有未commit 的文件

从远程仓库更新时, 假使本地还有没commit的文件A, 远程仓库的A文件却被修改了. 此时进行 git pull 时有如下信息:

6a707cc..f93575d master -> origin/master

Updating 6a707cc..f93575d

error: Your local changes to the following files would be overwritten by merge:

apps/myapp/utils.py

Please, commit your changes or stash them before you can merge.

Aborting

此时, 如果不想将本地文件commit(可能只是临时的修改), 但是又像将远程的仓库更新下来, 可以这样:

$ git stash # 先将自己的改变保存起来

Saved working directory and index state WIP on master: 6a707cc ...

HEAD is now at 6a707cc ...

$ git pull # 从远程仓库更新

Updating 6a707cc..f93575d

... ...

$ git stash pop # 将自己的修改合并到更新后的代码中

最后一步如果有冲突, 再参照上一节中解决冲突的步骤, 用文本编辑器修改冲突文件.

### git 分支合并时的冲突

正在开发的分支和主分支的编辑了同一个文件时, 在主分支上进行 merge 的时候可能会产生冲突.  
以下构造一个冲突的示例:

$ git branch test # 创建一个分支 test, 但是没有进入test分支, 此时还在 master 分支上.

$ vim xxxx # 编辑 master 分支上的一个已有的文件

$ git commit -am 'xxx message' # 提交 master 分支的修改

$ git checkout test # 切换到 test 分支

$ vim xxxx # 编辑之前在 master 上编辑的文件, 可以编辑同一个地方, 造成冲突

$ git commit -am 'xxx message' # 提交 test 分支的修改

$ git checkout master # 切换到 master 分支

$ git merge test # 将 test 分支合并到 master 分支, 由于上面编辑了同一文件, 这里会产生冲突

Auto-merging xxxx

CONFLICT (content): Merge conflict in xxxx

Automatic merge failed; fix conflicts and then commit the result.

最后, 参照上一节中解决冲突的步骤, 用文本编辑器修改冲突文件.

## 通过 git 提取补丁

提取的补丁的方法有多种:

$ git format-patch -1 # 提取本次 commit 和上次 commit 之间的不同, 并生成patch文件

$ git format-patch -2 # 提取本次 commit 和 上上次 commit 之间的不同, 并生成patch文件

$ git format-patch commit号1 commit号2 # 提取2次commit号之间的不同, 并生成patch文件 (commit号可以通过 git log 来查看)

$ git format-patch tag1 tag2 # 提取2次tag之间的不同, 并生成patch文件 (tag可以通过 git tag 来查看)

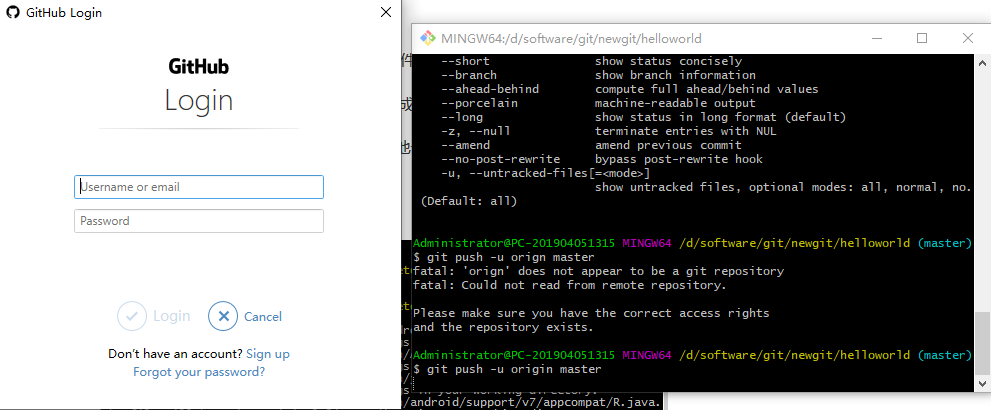
## 通过 git 提取指定版本的源码

这个功能在部署的时候比较有用.

$ git archive --format=tar --prefix="tagxx/" tagxx > ../tagxx.tar # 获取 tagxx 的源码, 加了 --prefix 的作用是在最终的 tagxx.tar 中加了一层文件夹 tagxx

# Git的使用--如何将本地项目上传到Github（两种简单、方便的方法）

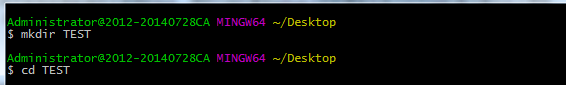
https://cloud.tencent.com/developer/article/1504684



第一步：我们需要先创建一个本地的版本库（其实也就是一个文件夹）。

你可以直接右击新建文件夹，也可以右击打开Git bash命令行窗口通过命令来创建。

现在我通过命令行在桌面新建一个TEST文件夹（你也可以在其他任何地方创建这个文件夹），并且进入这个文件夹

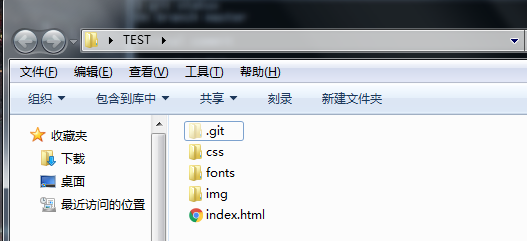




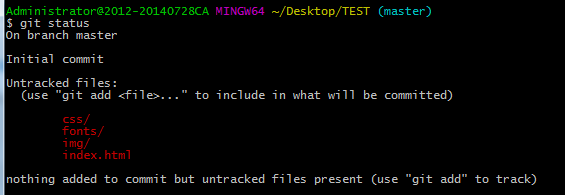
第二步：通过命令git init把这个文件夹变成Git可管理的仓库



这时你会发现TEST里面多了个.git文件夹，它是Git用来跟踪和管理版本库的。如果你看不到，是因为它默认是隐藏文件，那你就需要设置一下让隐藏文件可见。

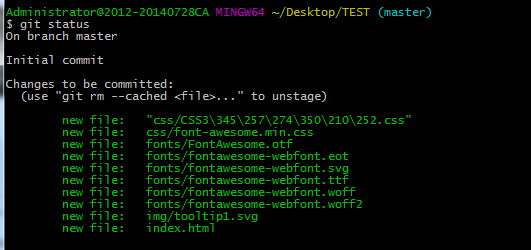


第三步：这时候你就可以把你的项目粘贴到这个本地Git仓库里面（粘贴后你可以通过git status来查看你当前的状态），然后通过git add把项目添加到仓库（或git add .把该目录下的所有文件添加到仓库，注意点是用空格隔开的）。在这个过程中你其实可以一直使用git status来查看你当前的状态。

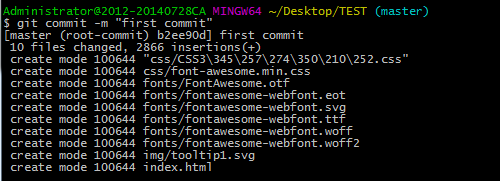


这里提示你虽然把项目粘贴过来了，但还没有add到Git仓库上，然后我们通过git add .把刚才复制过来的项目全部添加到仓库





第四步：用git commit把项目提交到仓库。



-m后面引号里面是本次提交的注释内容，这个可以不写，但最好写上，不然会报错，详情自行Google。 好了，我们本地Git仓库这边的工作做完了，下面就到了连接远程仓库（也就是连接Github）

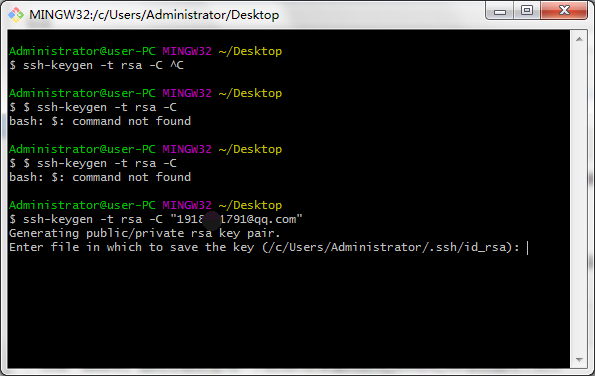
由于本地Git仓库和Github仓库之间的传输是通过SSH加密的，所以连接时需要设置一下：

第五步：创建SSH KEY。先看一下你C盘用户目录下有没有.ssh目录，有的话看下里面有没有id\_rsa和id\_rsa.pub这两个文件，有就跳到下一步，没有就通过下面命令创建。

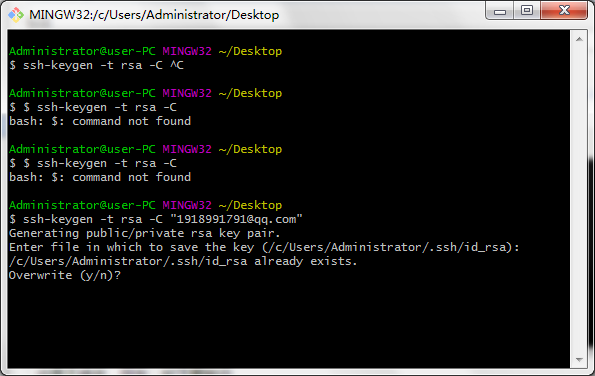
1、输入下边的命令

$ ssh-keygen -t rsa -C "youremail@example.com" 注意ssh-keygen之间没有空格

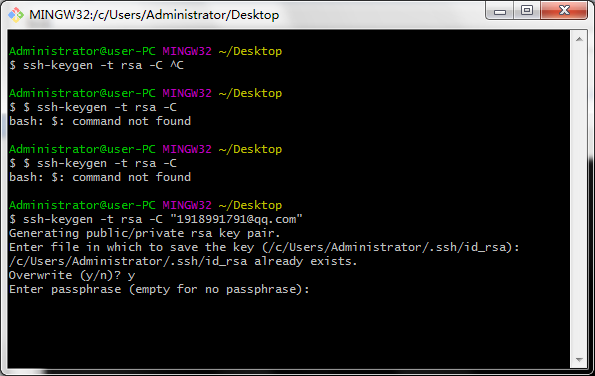
2、然后回车，询问保存key的位置，默认是在括号里的路径下，你可以修改，也可以不做修改。如下图所示：



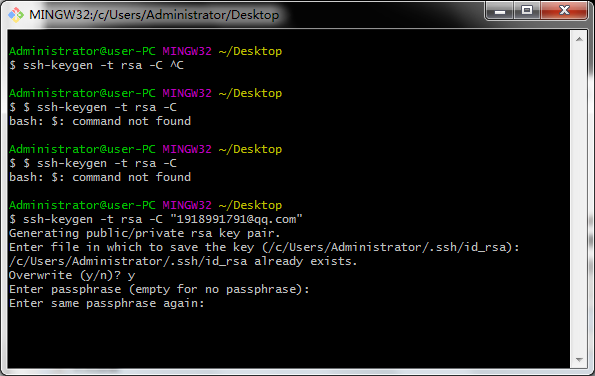
3、这里不修改，回车，提示已存在，是否覆盖。因为我这是第二次，所以有这个提示如下题所示：



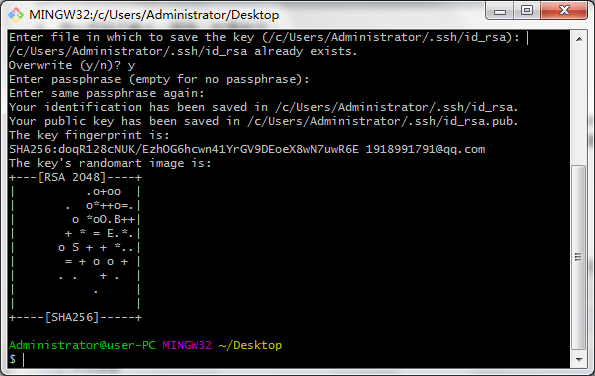
4、输入Y，回车，提示输入密码，如下图所示：



5、为了不必要麻烦，还是不要设置密码，因为容易忘记，不输入密码，回车，如下图所示：

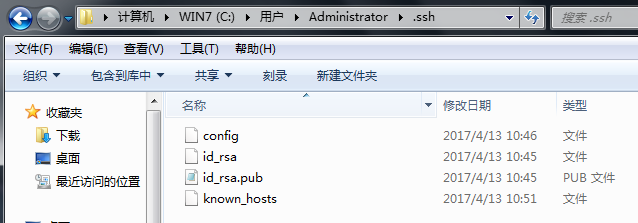


6、确认密码不输入，回车，如图所示：

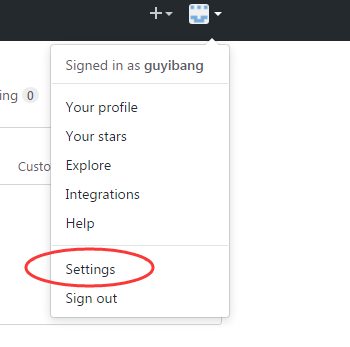


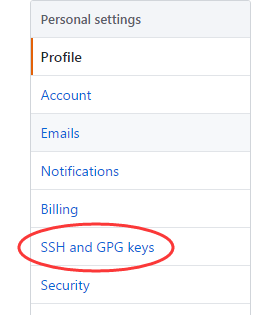
出现上图结果，表明创建key成功！

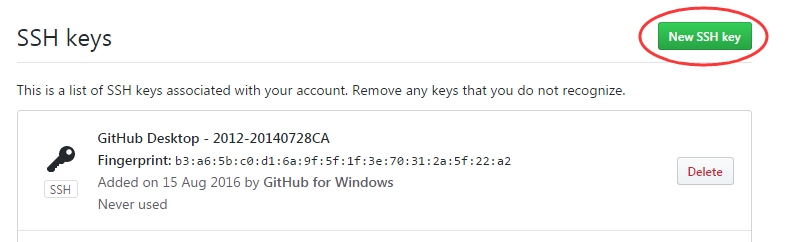
7、这时你就会在用户下的.ssh目录里找到id\_rsa和id\_rsa.pub这两个文件

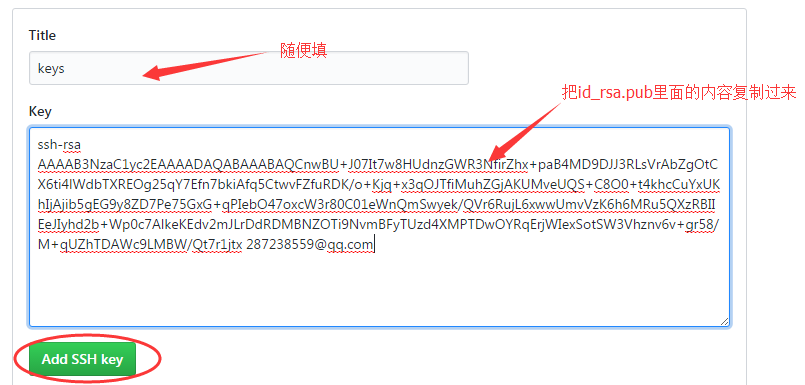


第六步：登录Github,找到右上角的图标，打开点进里面的Settings，再选中里面的SSH and GPG KEYS，点击右上角的New SSH key，然后Title里面随便填，再把刚才id\_rsa.pub里面的内容复制到Title下面的Key内容框里面，最后点击Add SSH key，这样就完成了SSH Key的加密。具体步骤也可看下面：



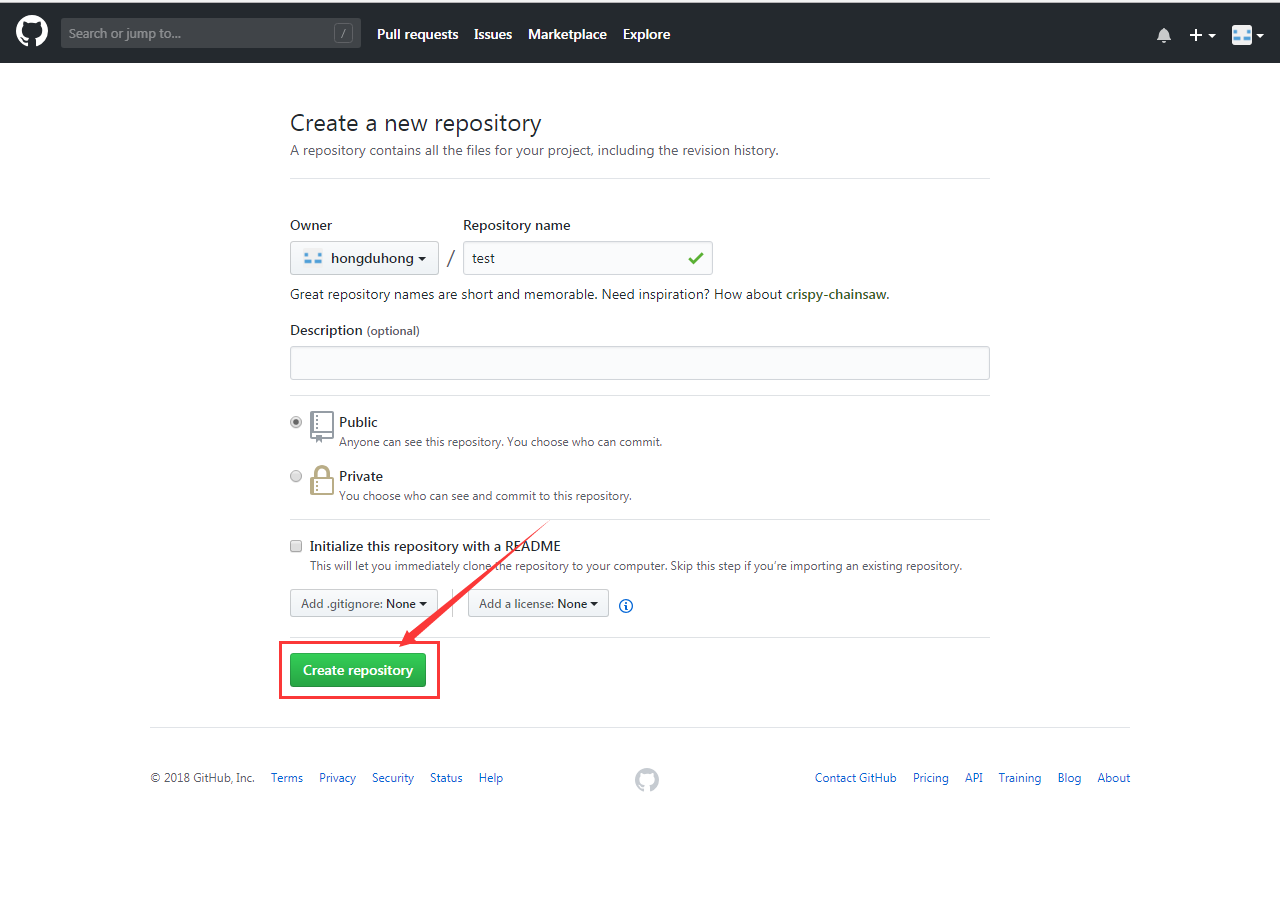






第七步：在Github上创建一个Git仓库。

你可以直接点New repository来创建，比如我创建了一个TEST2的仓库（因为我里面已经有了一个test的仓库，所以不能再创建TEST仓库）。



第八步：在Github上创建好Git仓库之后我们就可以和本地仓库进行关联了，根据创建好的Git仓库页面的提示，可以在本地TEST仓库的命令行输入：

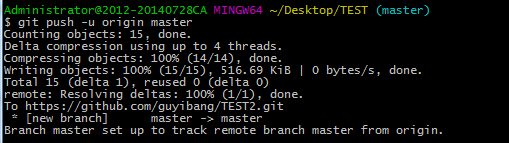
$ git remote add origin https://github.com/guyibang/TEST2.git



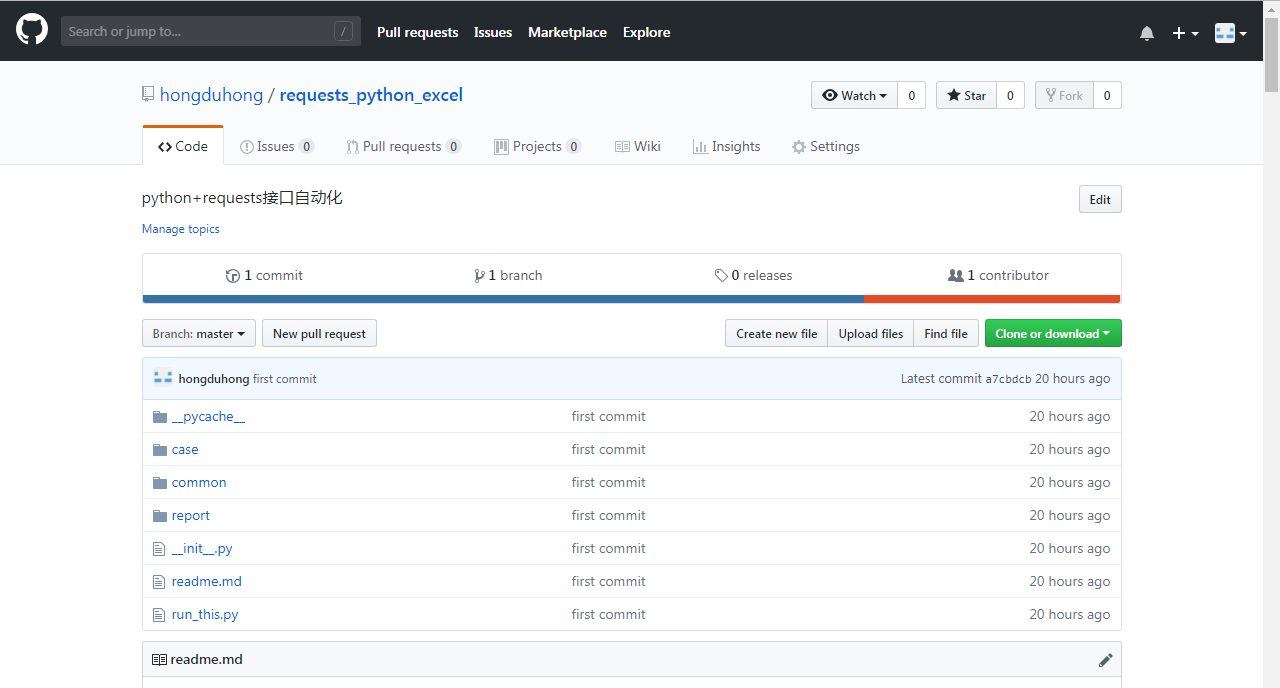
注意origin后面加的是你Github上创建好的仓库的地址。

第九步：关联好之后我们就可以把本地库的所有内容推送到远程仓库（也就是Github）上了，通过：

$ git push -u origin master 由于新建的远程仓库是空的，所以要加上-u这个参数，等远程仓库里面有了内容之后，下次再从本地库上传内容的时候只需下面这样就可以了： $ git push origin master 上传项目的过程可能需要等一段时间，完成之后是这样的：

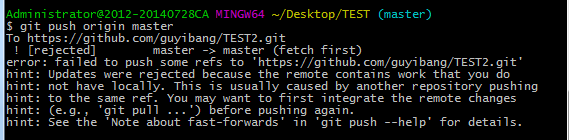


这时候你再重新刷新你的Github页面进入刚才新建的那个仓库里面就会发现项目已经成功上传了：



至此就完成了将本地项目上传到Github的整个过程。

另外，这里有个坑需要注意一下，就是在上面第七步创建远程仓库的时候，如果你勾选了Initialize this repository with a README（就是创建仓库的时候自动给你创建一个README文件），那么到了第九步你将本地仓库内容推送到远程仓库的时候就会报一个failed to push some refs to https://github.com/guyibang/TEST2.git的错。



这是由于你新创建的那个仓库里面的README文件不在本地仓库目录中，这时我们可以通过以下命令先将内容合并以下：

$ git pull --rebase origin master

这时你再push就能成功了。

总结：其实只需要进行下面几步就能把本地项目上传到Github

1、在本地创建一个版本库（即文件夹），通过git init把它变成Git仓库；

2、把项目复制到这个文件夹里面，再通过git add .把项目添加到仓库；

3、再通过git commit -m "注释内容"把项目提交到仓库；

4、在Github上设置好SSH密钥后，新建一个远程仓库，通过git remote add origin https://github.com/guyibang/TEST2.git将本地仓库和远程仓库进行关联；

5、最后通过git push -u origin master把本地仓库的项目推送到远程仓库（也就是Github）上；（若新建远程仓库的时候自动创建了README文件会报错，解决办法看上面）。