学习git在vsc中应用√

<https://code.visualstudio.com/docs/editor/versioncontrol#_scm-providers>

学习github基本应用√

<https://guides.github.com/>

github项目学习：

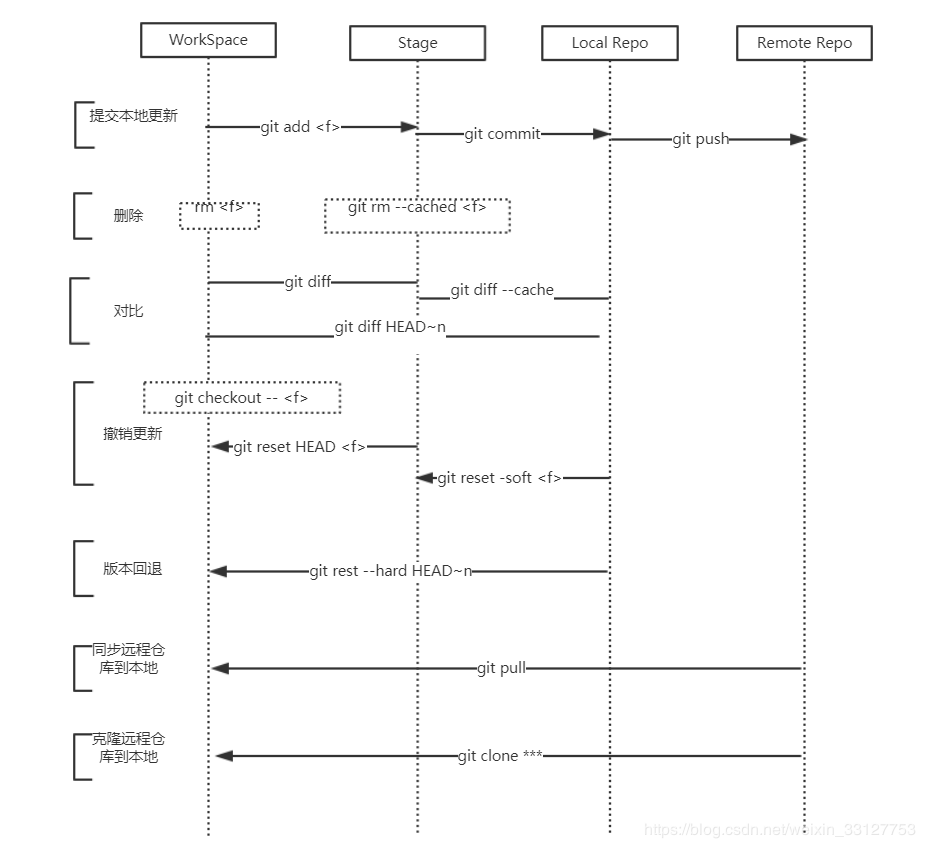
<https://github.com/NovelCOVID/API>

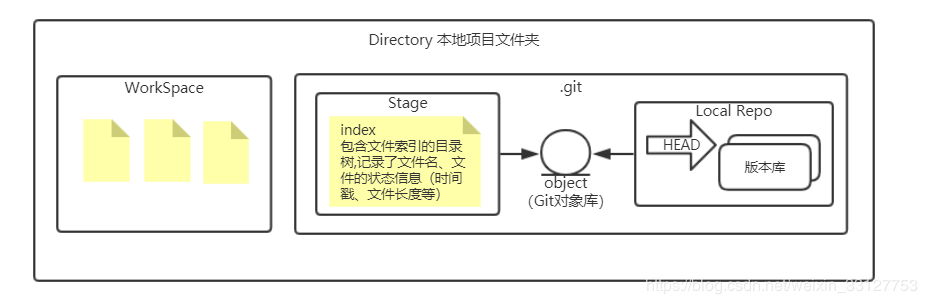
1. 阅读文件，了解使用的技术，文件夹结构和用处，文件的作用
2. 项目管理，团队成员构成，项目推进（功能、）

# Git

## Summary

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Workplace**  **工作区** | **Stage**  **缓存区** | **Local Repository**  **本地仓库** | **Remote Repository远端仓库** |
| **创建仓库** |  |  | git init |  |
| **复制远端到本地** |  |  |  | git clone |
| **获取远端仓库** |  |  |  | git fetch |
| **同步远端到本地** |  |  |  | git pull |
| **添加/更新** | touch | git add | git commit | git push |
| **删除** | rm  git rm -f | git rm --cached  git rm -f |  |  |
| **查看改动** | git diff | git diff --cache |  |  |
| **撤销更新** | git checkout | git reset HEAD |  |  |
| **查看文件/列表** | cat/ls | git status |  |  |
| **移动或重命名** | git mv |  |  |  |
| **查看分支** | git branch |  |  |  |
| **创建分支** | git branch <f> |  |  |  |
| **切换分支** | git checkout |  |  |  |
| **合并分支** | git merge |  |  |  |
| **删除分支** | git branch -d |  |  |  |
| **查看历史** | git log |  |  |  |
| **创建标签** | git tag -a |  |  |  |
| **查看标签** | git tag |  |  |  |
| **创建远端实例** |  |  |  | git remote add |
| **查看远端实例** |  |  |  | git remote |
| **删除远端实例** |  |  |  | git remote rm |





## GIT安装与配置

Git 是一个开源的分布式版本控制系统，用于敏捷高效地处理任何或小或大的项目。

### 安装

从git-scm.com上下载安装包

git主要使用命令行工具操作。

安装完毕后，会在开始菜单中显示有三个git菜单：Git-bash、GIt-CMD、Git-GUI.

### Git 基本配置

#### 配置文件及层级

Git 提供了一个叫做 git config 的工具，专门用来配置或读取相应的工作环境变量。

这些环境变量，决定了 Git 在各个环节的具体工作方式和行为。这些变量可以存放在以下三个不同的地方：

* /etc/gitconfig 文件：系统中对所有用户都普遍适用的配置。若使用 git config 时用 --system 选项，读写的就是这个文件。
* ~/.gitconfig 文件：用户目录下的配置文件只适用于该用户。若使用 git config 时用 --global 选项，读写的就是这个文件。
* 当前项目的 Git 目录中的配置文件（也就是工作目录中的 .git/config 文件）：这里的配置仅仅针对当前项目有效。每一个级别的配置都会覆盖上层的相同配置，所以 .git/config 里的配置会覆盖 /etc/gitconfig 中的同名变量。

#### 配置用户信息

配置个人的用户名称和电子邮件地址：

git config --global user.name "SimonFong"

git config --global user.email "simonangelfong@gmail.com"

如果用了 **--global** 选项，那么更改的配置文件就是位于你用户主目录下的那个，以后你所有的项目都会默认使用这里配置的用户信息。

如果要在某个特定的项目中使用其他名字或者电邮，只要去掉 --global 选项重新配置即可，新的设定保存在当前项目的 .git/config 文件里。

#### 查看配置信息

* 要检查已有的配置信息，可以使用 git config --list 命令.

有时候会看到重复的变量名，那就说明它们来自不同的配置文件（比如 /etc/gitconfig 和 ~/.gitconfig），不过最终 Git 实际采用的是最后一个。

* 直接查阅某个环境变量的设定

git config user.name

## 工作流程workflow

一般工作流程如下：

* 克隆 Git 资源作为工作目录。
* 在克隆的资源上添加或修改文件。
* 如果其他人修改了，你可以更新资源。
* 在提交前查看修改。
* 提交修改。
* 在修改完成后，如果发现错误，可以撤回提交并再次修改并提交。

## 基本概念

工作区work place: 在电脑能看到的目录

暂存区stage/index： 一般存放在.git目录下的index文件中

版本库： 工作区有一个隐藏目录.git，时Git的版本库。



左侧为工作区，右侧为版本库。版本库中index就是暂存区；master代表目录树

“HEAD”是指向master分支的一个“游标”

objects是Git的对象库，位于.git/objects目录，里面包含了创建的各种对象及内容。

**Add：添加/修改暂存区文件**

当对工作区修改（或新增）的文件执行 "git add" 命令时，暂存区的目录树被更新，同时工作区修改（或新增）的文件内容被写入到对象库中的一个新的对象中，而该对象的ID被记录在暂存区的文件索引中。

**Commit：将master目录树更新为暂存区**

当执行提交操作（git commit）时，暂存区的目录树写到版本库（对象库）中，master 分支会做相应的更新。即 master 指向的目录树就是提交时暂存区的目录树。

**reset HEAD：重置暂存区目录树**

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。

**rm cached <file>：对暂存区操作**

当执行 "git rm --cached <file>" 命令时，会直接从暂存区删除文件，工作区则不做出改变。

**checkout：对工作区操作**

当执行 "git checkout ." 或者 "git checkout -- <file>" 命令时，会用暂存区全部或指定的文件替换工作区的文件。这个操作很危险，会清除工作区中未添加到暂存区的改动。

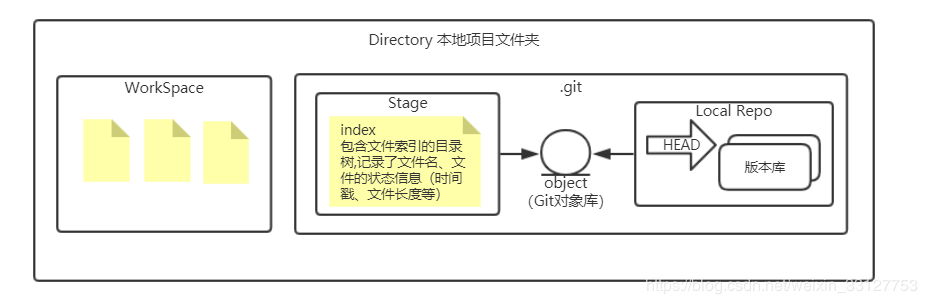
**checkout HEAD：替换暂存区和工作区文件**

当执行 "git checkout HEAD ." 或者 "git checkout HEAD <file>" 命令时，会用 HEAD 指向的 master 分支中的全部或者部分文件替换暂存区和以及工作区中的文件。这个命令也是极具危险性的，因为不但会清除工作区中未提交的改动，也会清除暂存区中未提交的改动。

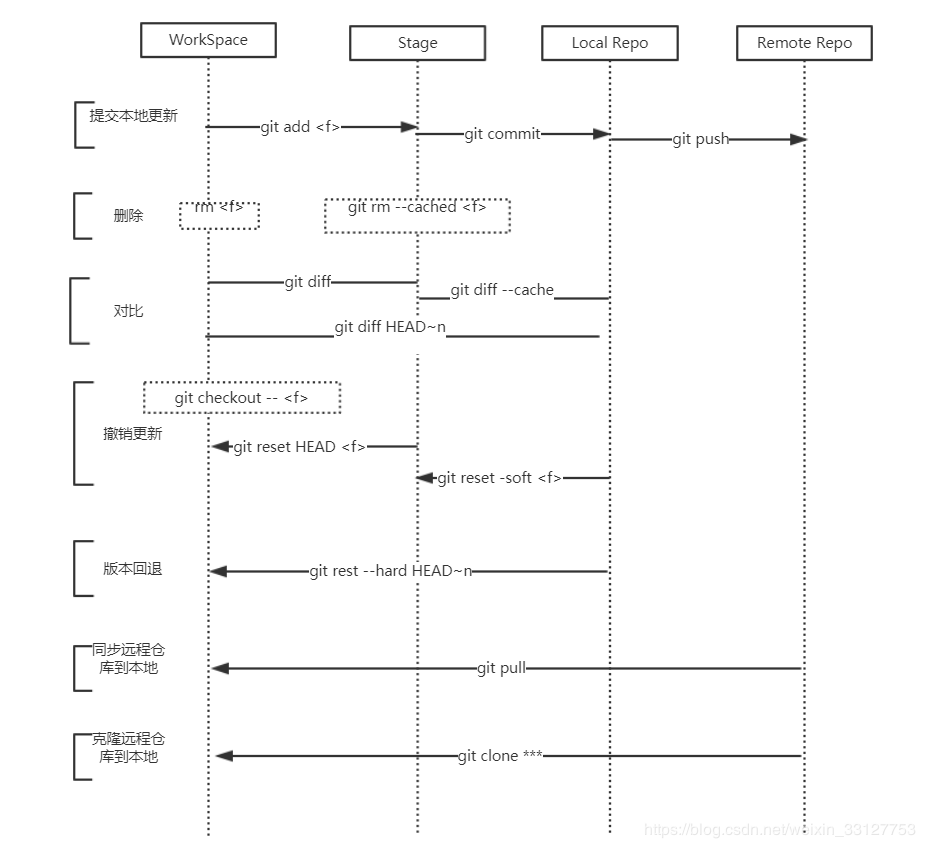
## 基本操作

Git 的工作就是创建和保存你项目的快照及与之后的快照进行对比。

## 操作图解（重要）







## 分支管理

当执行创建仓库时，会创建一个master分支

### 创建分支：git branch

#### 使用：

没有参数时，会列出在本地的分支：git branch (branchname)

### 切换分支：git checkout (branchname)

当你切换分支的时候，Git 会用该分支的最后提交的快照替换你的工作目录的内容， 所以多个分支不需要多个目录。

### 合并分支：git merge

合并并不仅仅是简单的文件添加、移除的操作，文件内容Git 也会合并修改。

### 删除分支：git branch -d (branchname)

## 标签 gti tag

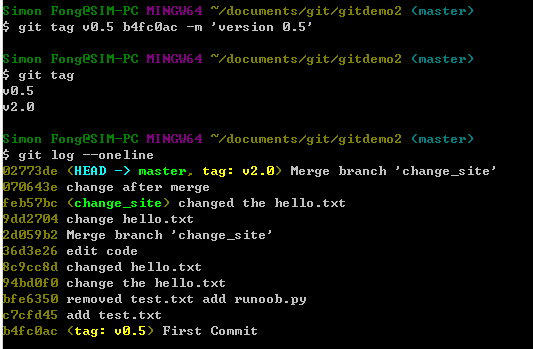
当要记录特别的提交快照时，可以使用标签进行标记。

标签是版本库的快照，但其实它就是指向某个 commit 的指针。

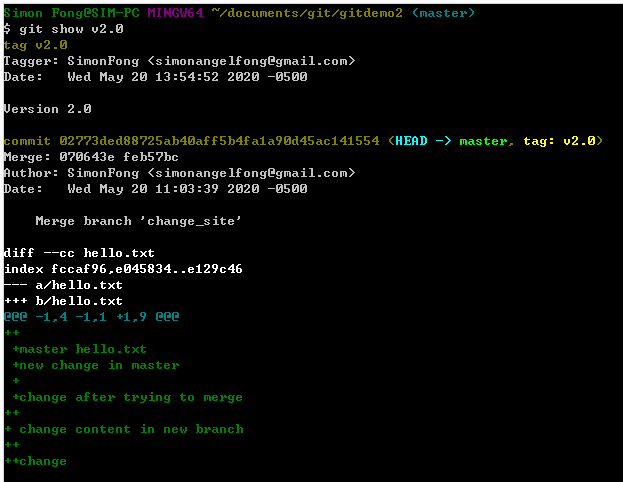
创建和删除标签都是瞬间完成的。

### 使用

* 创建一个带有注解的标签：git tag -a <tagname> -m'<Comment>'
* 追加一个标签：git tag -a <tagname> <commitId> -m'<comment>'



* 查看所有标签：git tag
* 删除标签: git tag -d v1.0
* 查看版本所有修改的内容：git show <tagname>



## 远程仓库——Github应用

### 添加远程库：git remote add

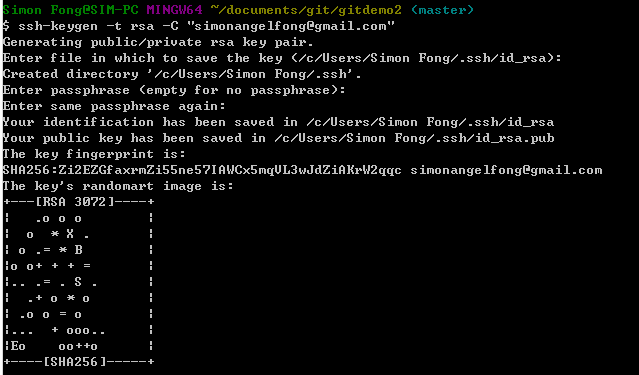
#### 命令：

git remote add [shortname] [url]

### 实例：在本地创建远端仓库实例

1. **创建github ssh key并记录在github账户中**
2. 生成新的SSH Key，打开git命令面板输入命令：

ssh-keygen -t rsa -C "simonangelfong@gmail.com"

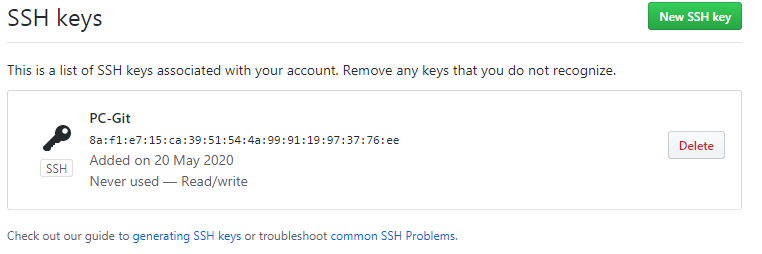


1. 成功后，会生成.ssh文件夹。进入该文件夹，打开id\_rsa.pub,复制里面key
2. 登录github，进入Account Setting，选择ssh keys，粘贴上电脑生成的key。

使用命令复制key:

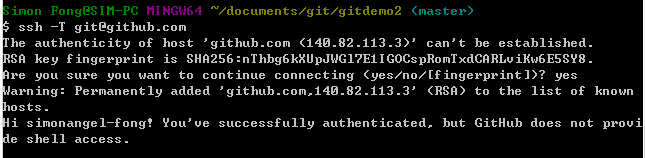


在账户上创建SSH Key

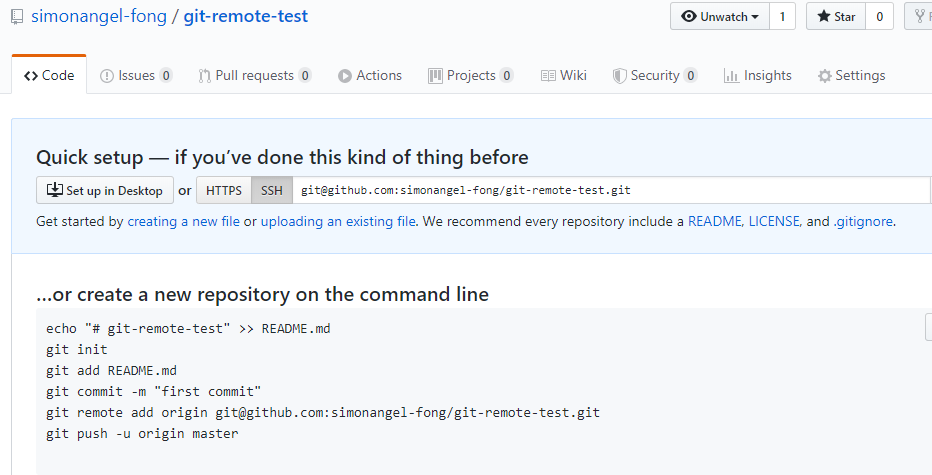


1. 验证是否成功。命令：

ssh -T git@github.com



1. **创建本地远端仓库的实例，并上传本地仓库到github**
2. 在github创建新仓库。



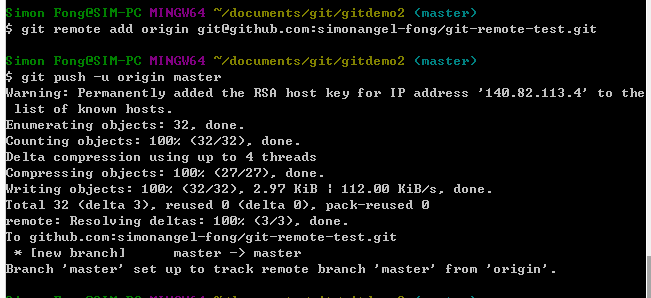
以上信息证明本地git已经与github连接上。

1. 将本地仓库上传到远程仓库

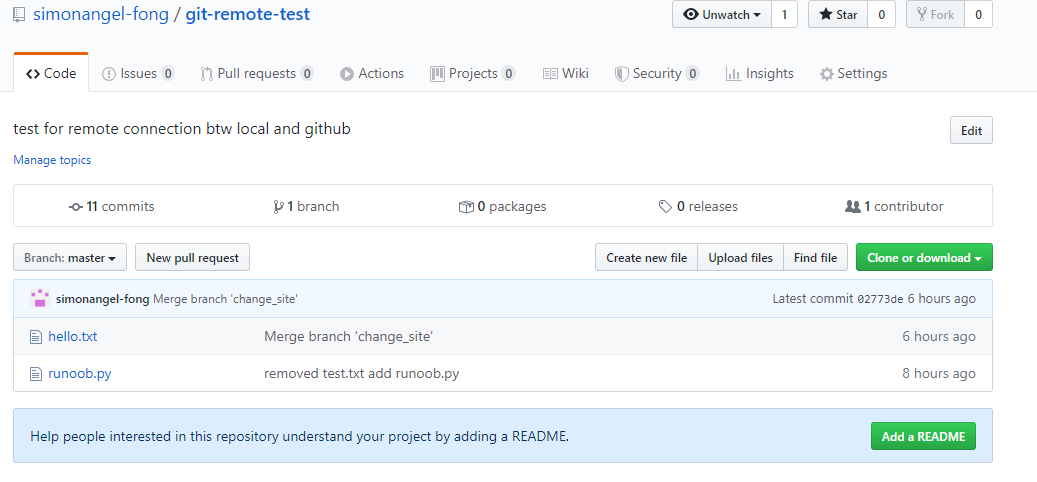
在本机创建一个名为origin的远程仓库：git remote add <name> <url>

git remote add origin git@github.com:simonangel-fong/git-remote-test.git

git push origin master

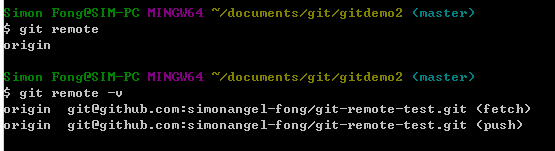


1. 检查github，显示本地仓库的文件已经被上传到github。



### 查看当前配置有哪些远程仓库: git remote

* 查看建立的远程仓库：git remote
* 查看远程仓库的url：git remote -v



### 提取远程仓库的更新：git fetch + git merge/git pull

#### 从远程仓库下载新分支与数据：git fetch

执行下载后，需要在本地执行merge，将远程的分支与本地分支合并

让Git 去获取它有你没有的数据。

git fetch <remoteRepoName> <remoteBranchName>

从名为remoteRepoName的远端仓库的remoteBranchName的分支提取代码到本地的remoteRepoName/remoteBranchName

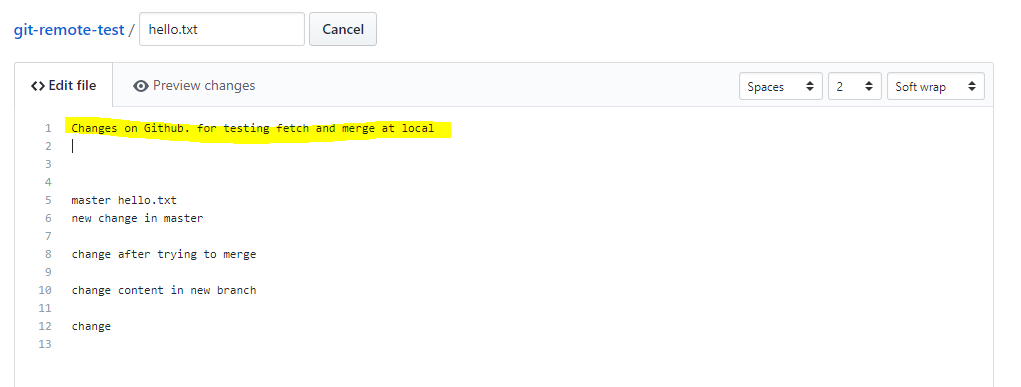
#### 从远程仓库提取数据并尝试合并到当前分支：git merge

git merge <branchname>

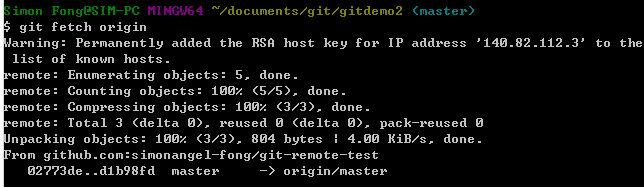
代码被拉到本地后，branchname是remoteRepoName/remoteBranchName。

#### 实例：提取Github最新的仓库

1. 在线修改github文件

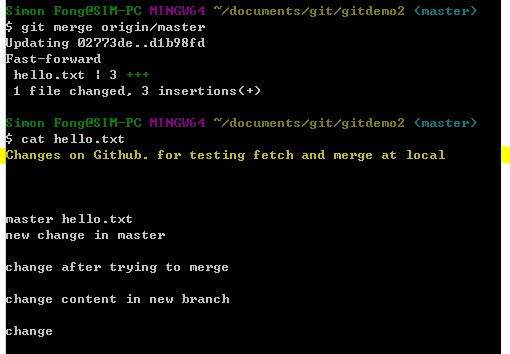


1. 在本地下载Github最新的数据



提示信息显示：master -> origin/master。即本地的master分支已经被更新。

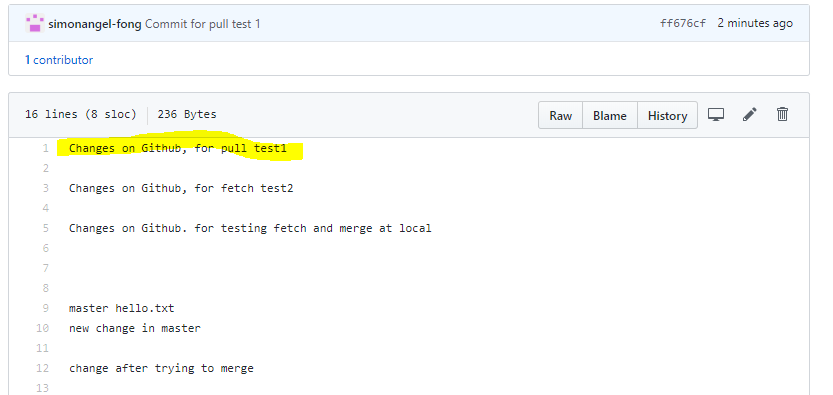
1. 合并origin/master;并查看文件内容，本地master文件已更新。



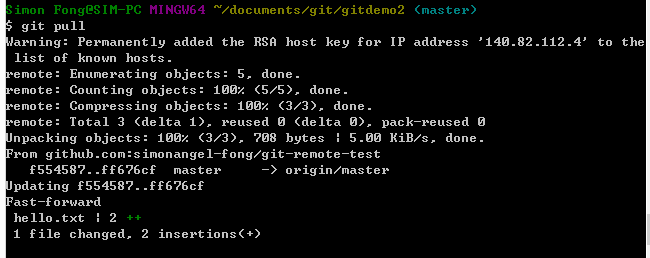
#### 从远程获取最新版本到本地，并自动合并：git pull

相当于pull = fetch+merge

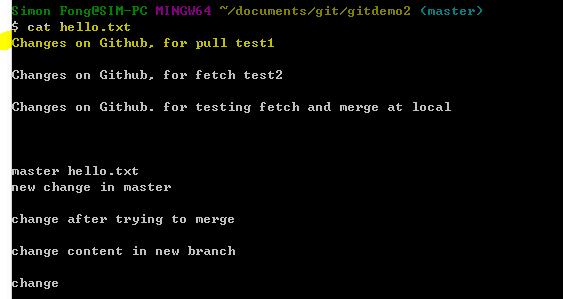
1. 在线修改github文件内容



1. 在本机执行pull



1. 确认更新情况，查看本地仓库文件内容



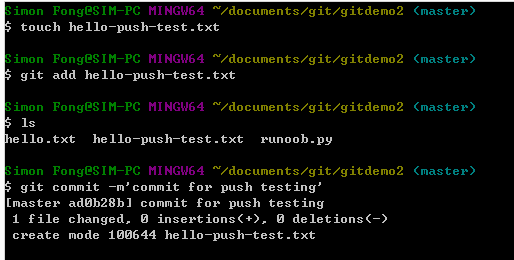
### 将本地分支推送到远端仓库：git push

#### 命令：

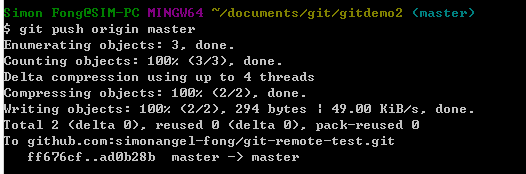
git push [alias] [branch]

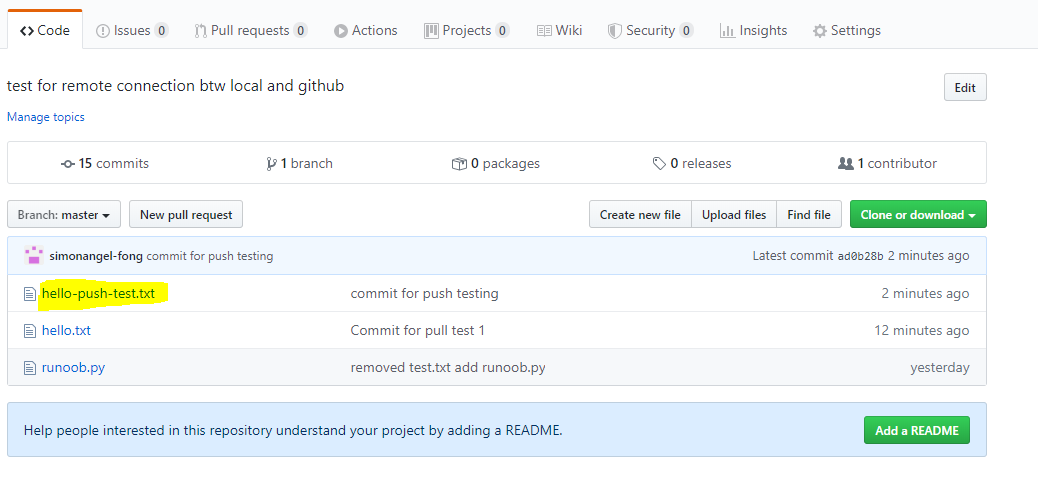
#### 实例：推送到远端仓库

1. 在本地创建、添加、提交文件



1. 向github推送

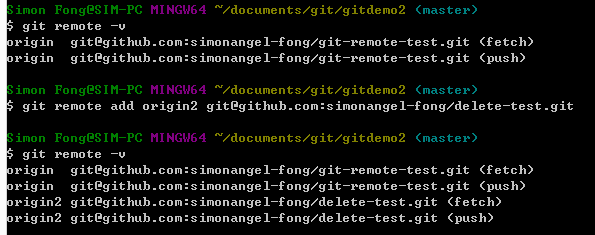




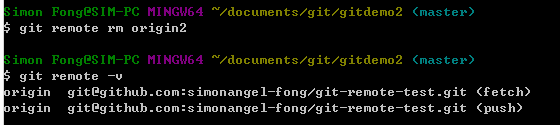
### 删除远端仓库：git remote rm [alias]

#### 实例：删除本机中创建远端仓库实例

1. 查看本机远端仓库实例；新建一个仓库实例，用作演示；确认新建成功。



1. 删除远端仓库实例；确认删除成功。



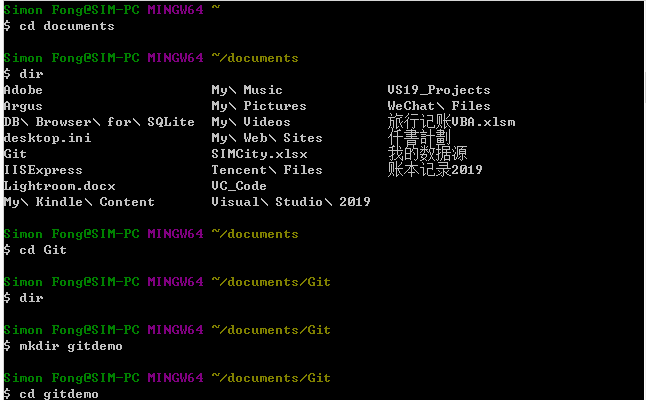
## 墙版Github（略）

## 搭建一台 Git 服务器作为私有仓库使用（略）

## 实例：基本操作

### init:创建仓库

进入"文档|Git"文件夹，创建"gitdemo"文件夹，并以该文件夹创建仓库

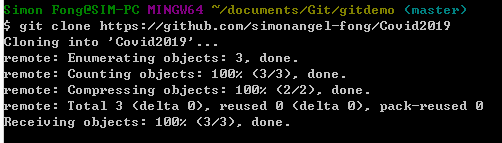




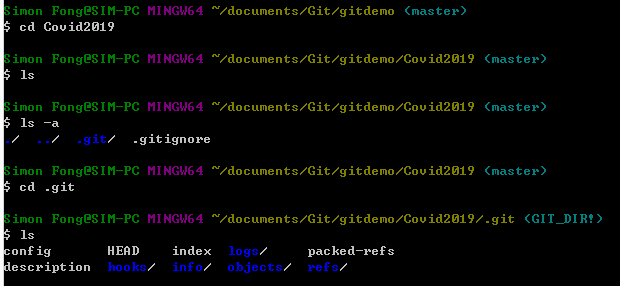
查看仓库内容：



### clone:从github克隆项目到gitdemo

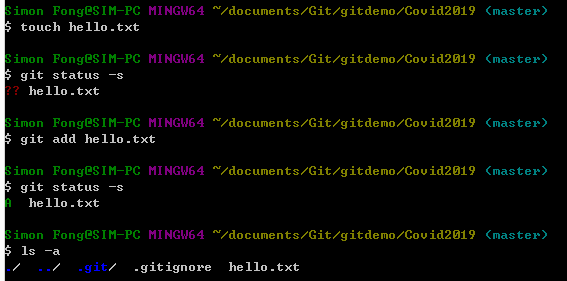


查看项目的全部记录



### add:添加添加文件到暂存区

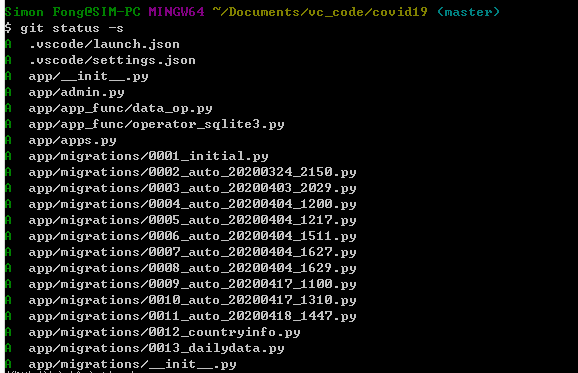
touch文件；查看状态，显示为??；添加add；状态显示为A；查看全部文件



将covid19的所有文件添加到仓库

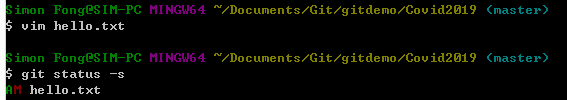


查看文件状态：



### add:修改文件后添加到暂存区

修改hello.txt；查询状态，状态为AM，即该文件添加到暂存区后发生了改动。



将修改后的文件再添加到暂存区中；查看状态；





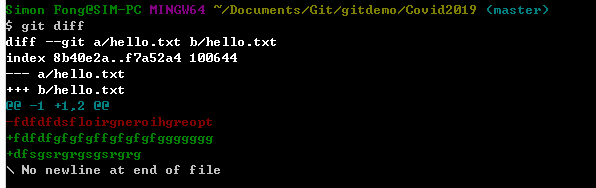
### diff：查看工作区与暂存区文件差异

1.修改文件hello.txt

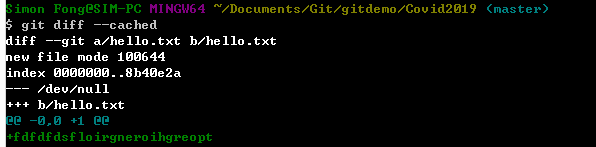
2.查看状态



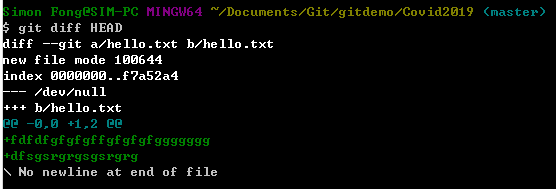
3.查看工作区的改动：git diff



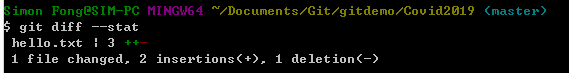
4.查看暂存区的改动: git diff --cached



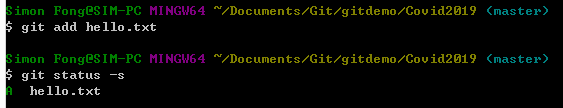
5.查看工作区和暂存区的所有改动: git diff HEAD



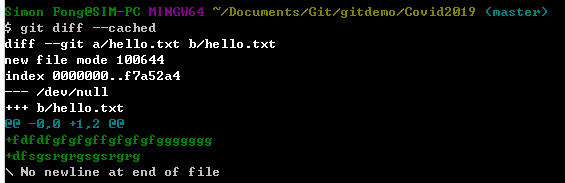
6.显示改动摘要: git diff --stat



7.将工作区改动的文件添加到暂存区,显示状态信息，文件状态为A

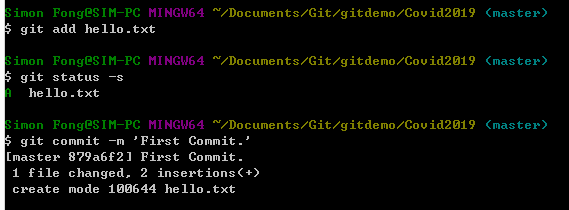


8.再查看暂存区内容文件变动: git diff --cached

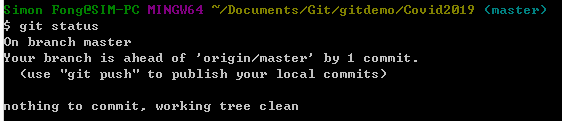


### commit:将暂存区的内容写入仓库

将文件添加到暂存区；检查最近一次提交文件；写入仓库。

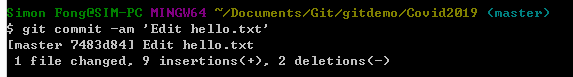


再检查状态，“nothing to commit, working tree clean”表明自上一次提交后，暂存区没有改动。



### commit -a:将工作区文件直接写入仓库

修改hello.txt内容；将工作区文件直接写入仓库

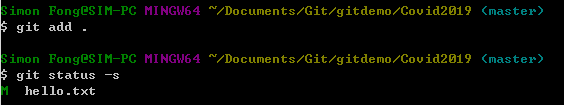


### reset HEAD: 撤销缓存区内文件的更改

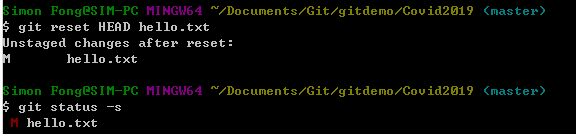
1. 修改工作区文件hello.txt,查看上一次添加后的文件状态为M（颜色），即被工作区修改



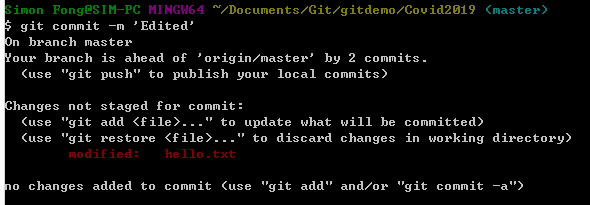
1. 将工作区文件添加到缓存区，再查看文件状态为M，即在缓存区内被修改。



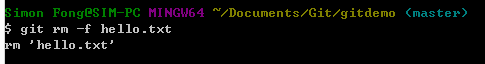
1. 撤销hello.txt文件在缓存区内的更改，并查看文件状态，状态为M（颜色），即在缓存区内容的变动被撤销了。



1. 执行Commit，尝试将缓存区内容写入仓库,显示没有内容被写入仓库。

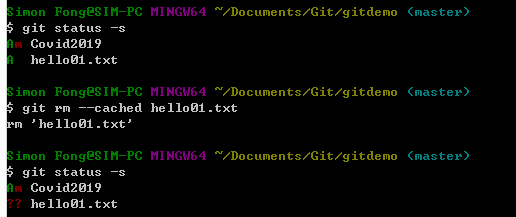


### rm -f: 强制删除工作区的文件



### rm --cached: 删除缓存区中文件

查询文件状态，文件被添加到缓存区；删除缓存区文件；再查询文件状态，文件不再在缓存区。

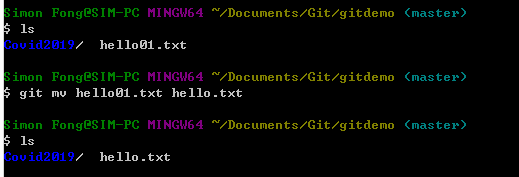


### rm -r \*: 删除所有文件和子目录

删除所有文件和子目录；查看当前文件夹的所有文件和子目录。

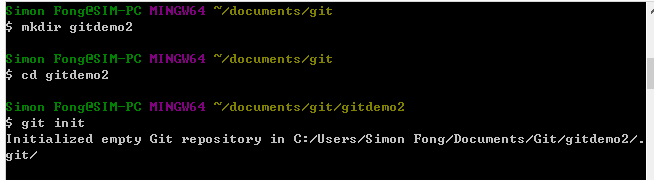


### mv：重命名文件

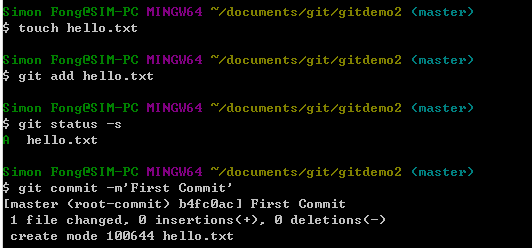


## 实例：分支管理

创建测试仓库：



添加文件：



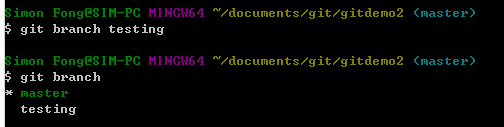
### branch: 显示本地分支

结果：本地有一个‘master’分支



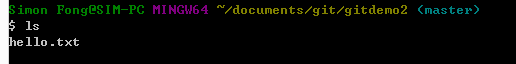
### branch (branchname): 创建一个分支

创建一个testing的分支；显示本地的分支。



### checkout (branch): 切换分支

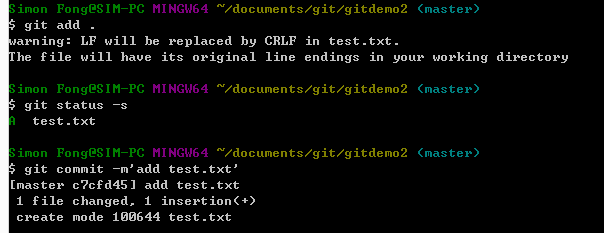
1. 查看本地仓库文件，结果只有hello.txt



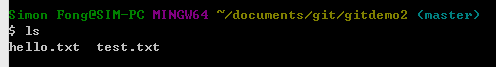
1. 创建test.txt文件，并将'runoob.com'写入文件test.txt



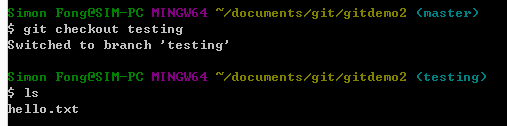
1. 添加并提交test.txt



1. 查看当前文件目录，结果当前分支中有两个文件

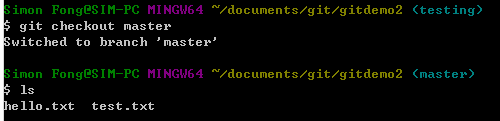


1. 切换到testing分支，并查看目录，只有一个文件。

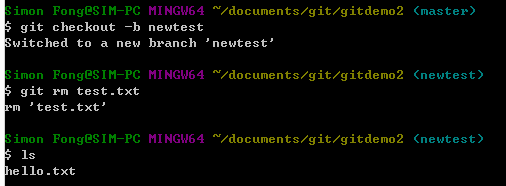


### checkout -b (branchname): 创建新分支并切换到新分支中

1. 切换到主分支,并查看主分支文件

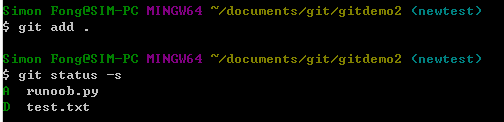


1. 创建新分支'newtest'并立即切换，删除文件test.txt;查看文件。

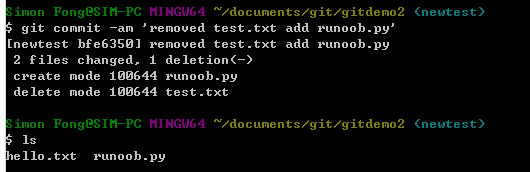


1. 创建runoob.py；添加文件到缓存区





1. 将缓存区内的改变写入仓库；查看仓库文件，test.txt文件被删除，runoob.py被添加



1. 切换到master分支；查看文件：test.txt没有被删除，runoob.py没有被添加

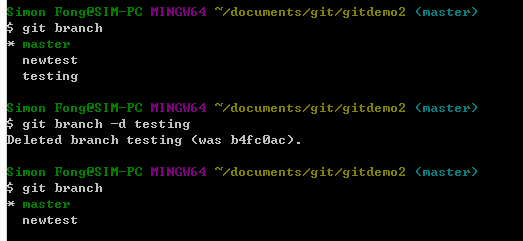




1. 在不同分支上的操作是相互独立的。

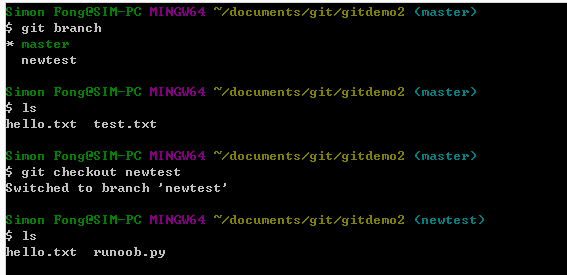
### branch -d: 删除分支

查看分支列表；删除分支；再查看分支列表。

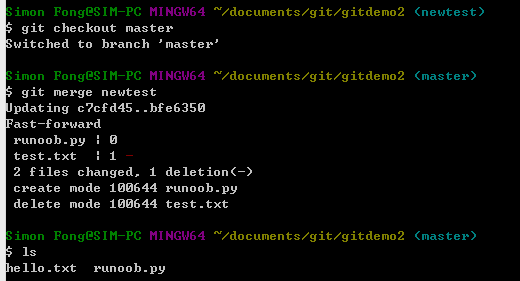


### merge: 合并分支

1. 查看分支列表；查看master文件；切换到newtest分支；查看文件



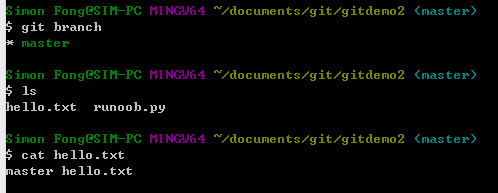
1. 切换到master分支；合并分支newtest；查看文件



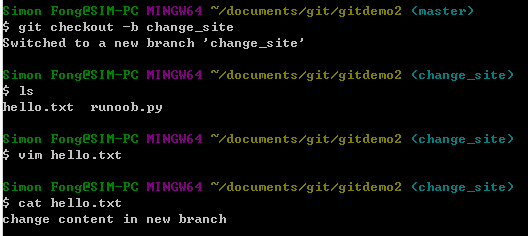
1. 合并的结果是，将指定的分支覆盖master分支

### 合并冲突：当两个分支合并时，如何处理内容有冲突的文件

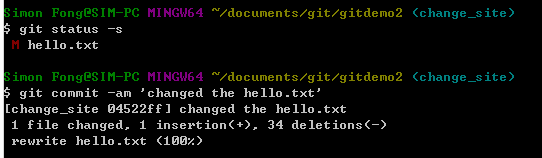
1. 查看分支列表；查看当前分支文件列表；查看hello.txt文件内容



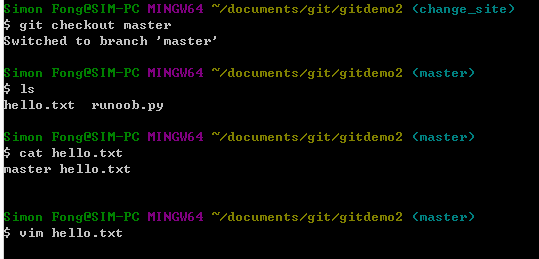
1. 创建change\_site分支并立即切换；修改hello.txt文件内容；显示修改后的文件内容

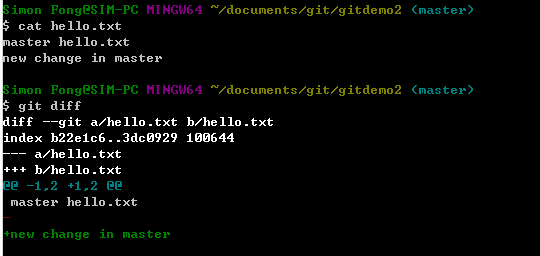


1. 显示当前文件状态；添加并提交文件

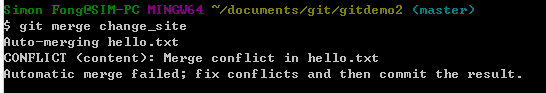


1. 切换到master；查看hello.txt文件内容，结果显示文件内容保留为原有内容;修改hello.txt文件内容；再次显示文件内容；显示文件的改动diff

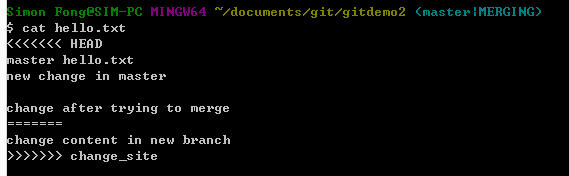




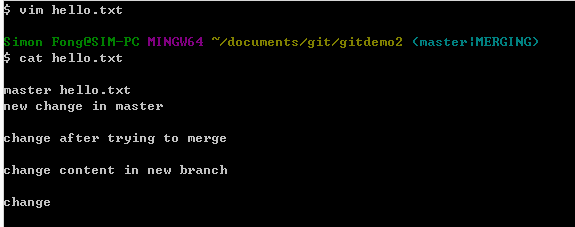
1. 通过以上步骤，在master和change\_site分支中都修改了hello.txt文件。接下来进行合并merge，会显示冲突信息。

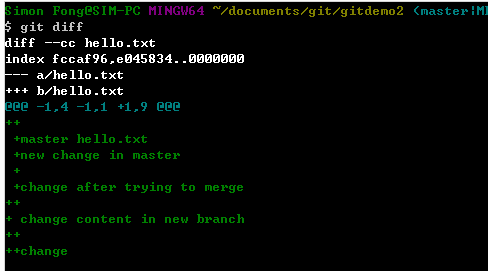


1. 然后手动消除冲突：查看文件内容，会发现master文件内容已经被修改，分别显示了master分支和change\_site分支的内容；

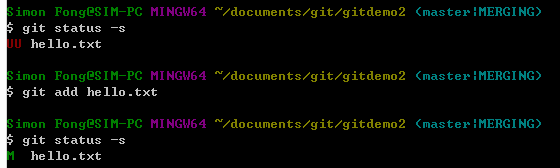


1. 修改master中的hello.txt文件内容；查看改动diff





1. 查看状态，显示文件为‘UU’；添加修改后的文件；查看状态为‘M’



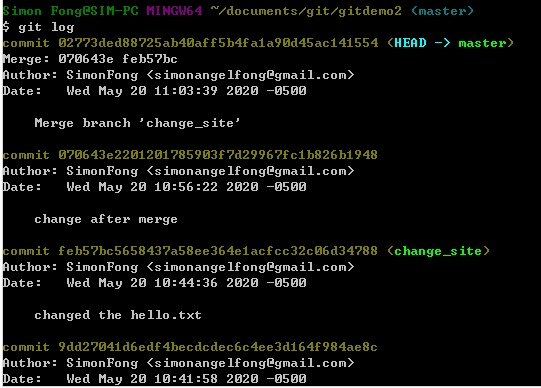
1. 提交冲突的文件，合并完成



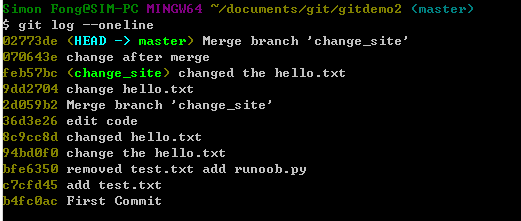


## 实例：查看历史记录

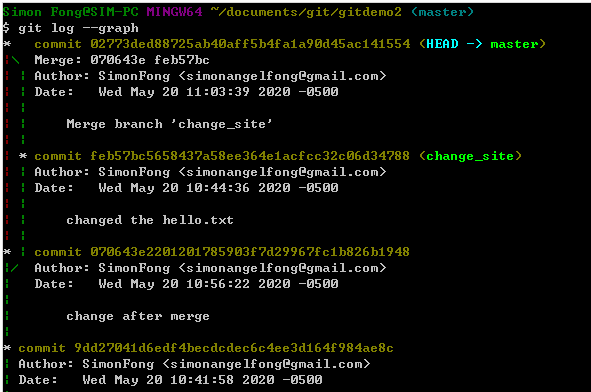
### log: 查看历史提交记录



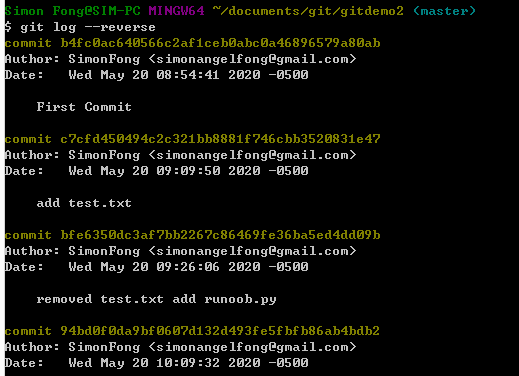
### log --oneline: 查看历史提交记录的简洁版本



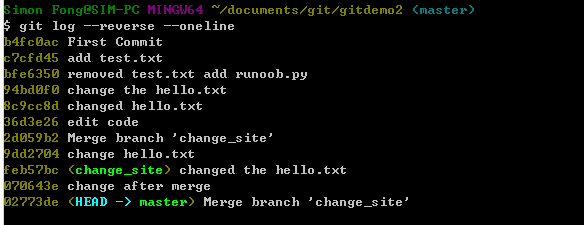
### log --graph: 以拓扑图显示分支和合并情况



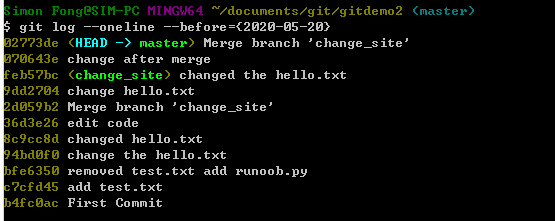
### log --reverse:逆向显示日志



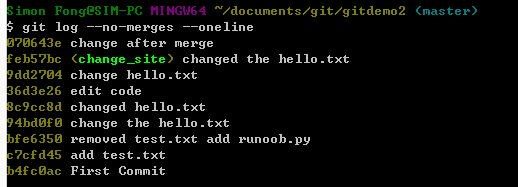
逆向+简短显示



### log --before: 查看指定日期范围的日志



### log --no-mergs:查看隐藏合并的日志



# Github

Github是一个基于git的代码托管平台

官网：github.com

## 名词解释：

**GitHub** ：GitHub is a code hosting platform for version control and collaboration.

**repository**: A repository is usually used to organize a single project.

**Branch**:Branching is the way to work on different versions of a repository at one time.

**Commit**: save changes in repository

**Pull Requests :** proposing changes and requesting review and pull in contribution and merge them.

## Tutorial : Hello world

**Oject:**

* Create a new repository
* Create a new branch
* make a commit
* make a pull request
* merge pull request into master
* delete a branch

### Step 1: Create a Repository

#### create a new repository

1. upper right corner, select "**New repository**"
2. Name repository "hello-world"
3. Click "**Create repository**"

### Step 2: Create a Branch

本教程分支示意图



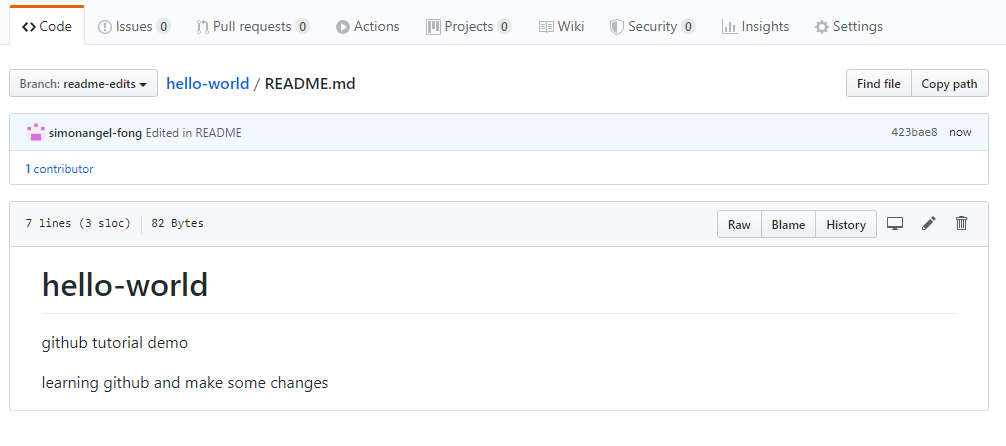
#### To Create a new branch

drop down "**branch:master**" => branch name : "readme-edits"

### Step 3: Make and commit changes

#### Make and commit changes

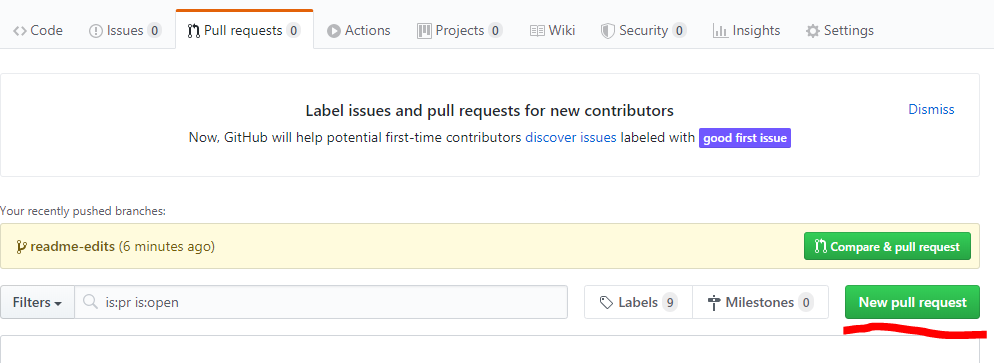
write sth in README.md file => write a commit message => "Commit changes"



### Step 4. Open a Pull Request

#### to open a pull request

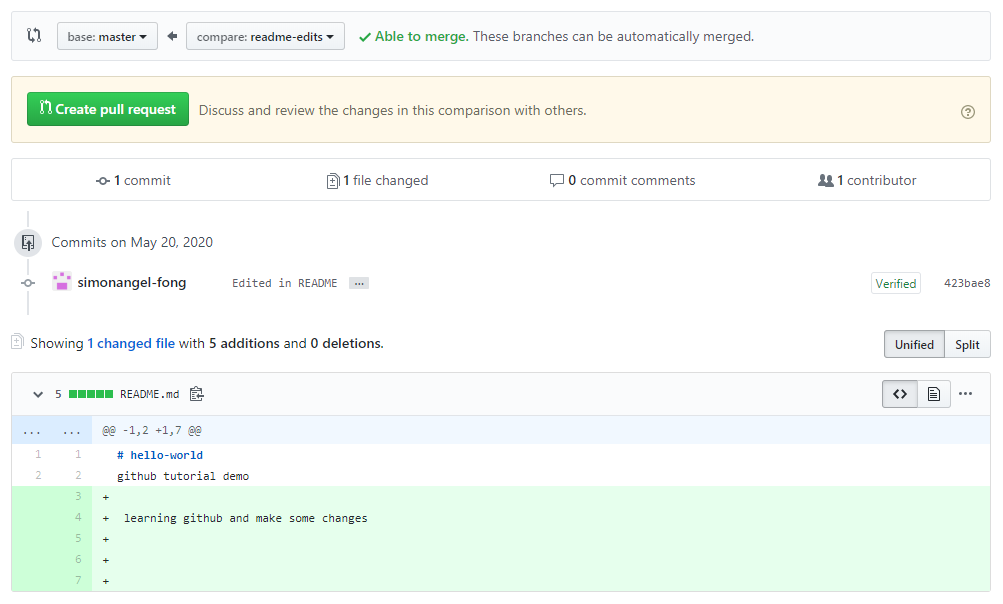
1. click **"Pull Request"** tab => click "**New pull request**" btn



1. In "**Compare changes**" page, select the branch "readme-edits", to compare with "master"



1. Check the diffs, and make true it is what you want to make.



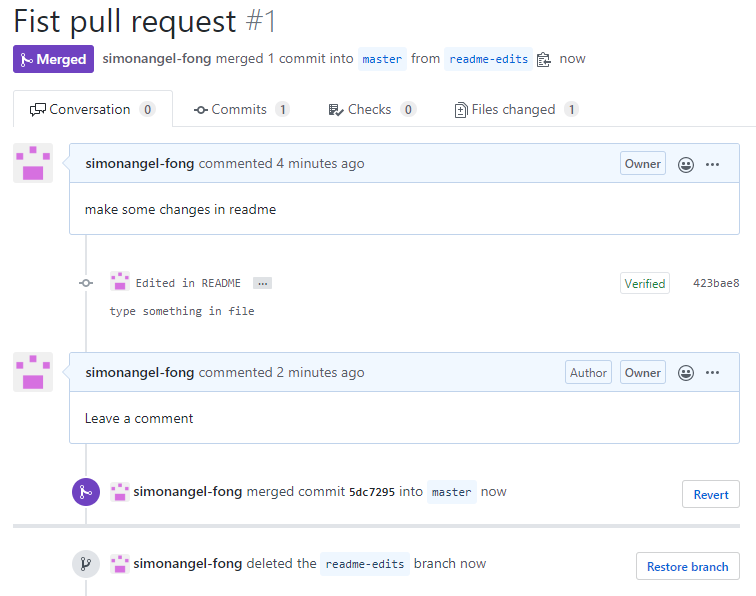
1. While satisfied, submit. '**Create Pull Request**'
2. Give a title for your pull request and write a brief description of changes. => '**Create pull request**'

### Step 5. Merge Pull Request

Click "Merge pull request" btn

Click "Confirm merge"

Delete branch ""



## Tutorial: flow工作流程

**Oject**: to know how and why Github flow works

### Create a branch

Branching is to help you manage features or ideas for a project in a workflow.

While creating a branch in project, creating an environment where can try out new ideas.

### Add commits

Making a commit is to adding operation of files to branch.

Commits are the track of the progress made. Each commit has message explaining why a change was made.

### Open a Pull Request

Pull Request initiate discussion about commits. Anyone can see what changes would be merged if they accept the request.

### Discuss and review your code

Once a Pull Request has been opened, the person or team reviewing your changes may have questions or comments.

### Deploy

Once your pull request has been reviewed and the branch passes your tests, you can deploy your changes to verify them in production. If your branch causes issues, you can roll it back by deploying the existing master into production.

### Merge

Once merged, Pull Requests preserve a record of the historical changes to your code.

When your Pull Request is merged, the related issues are also closed.

# Git in VS Code

## 环境配置：

* vsc
* git

## 名词解释：

SCM ：source control manager

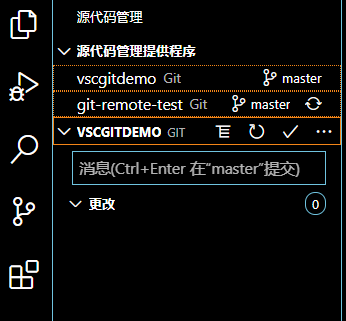
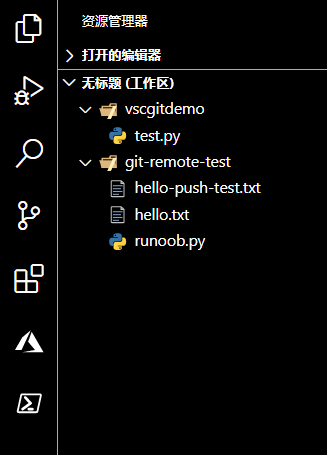
## Commit

可以在消息栏中输入提交信息，CTRL+ENTER



## Clone a repository

弹出命令栏（CTRL + SHIFT + P）=》 输入远端仓库URL =》 选择存储库位置 =》 可以添加到工作区\可以打开存储库



## Branch

### 创建分支

命令栏 =》 "Git: Create Branch" => 输入分支名称

### 切换分支

命令栏 =》 'Git: Checkout to' =>选择分支名称

## Tag

### 创建标签

命令栏：'Git: Create Tag' => 输入标签名称

## Diff

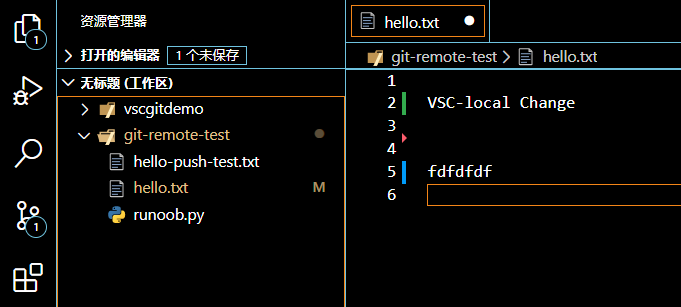
### 修改文档时提示：

在VSC修改文档会提示文档内容修改情况

红色三角：行被删除

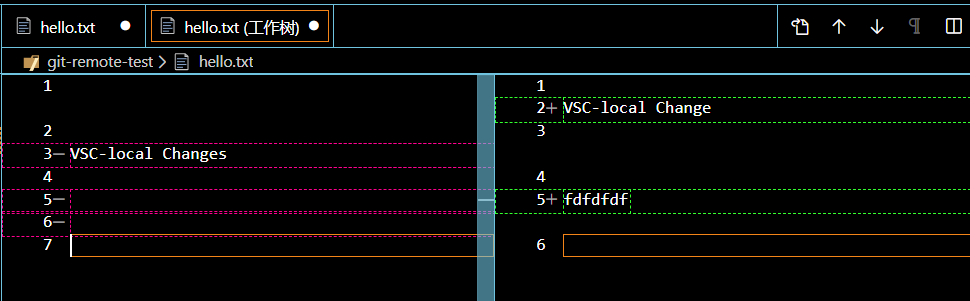
绿色块：新增航

蓝色块：行被修改



### 比较文档差异：

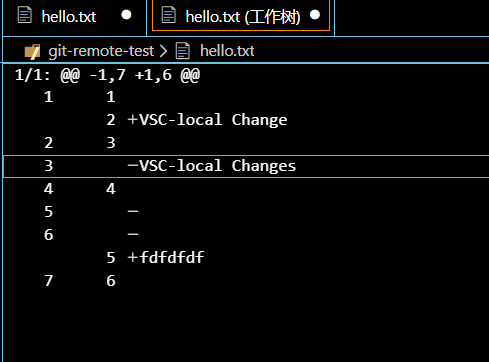
文档编辑器右上角‘打开更改’

****

### 逐行显示更改：

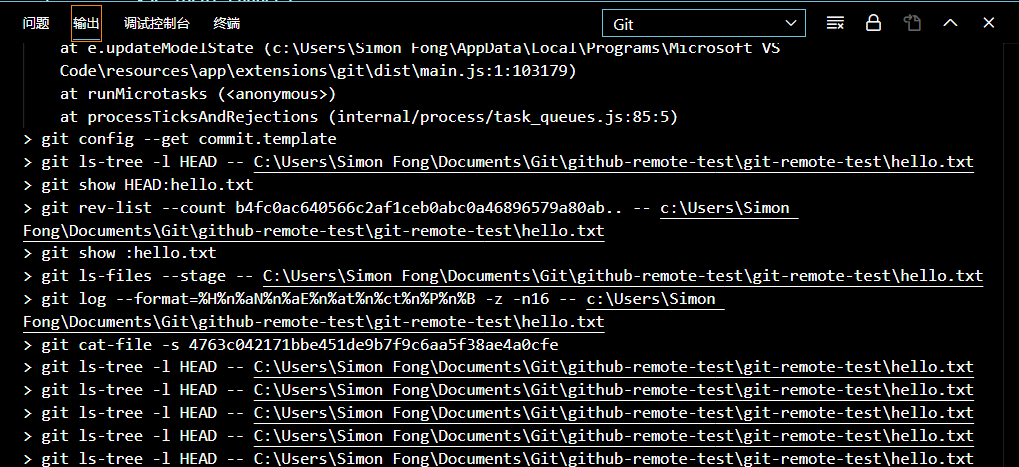
前往下一处更改：‘F7’

前往上一处更改：‘Shift+F7’



## 输出窗口

在输出窗口中可以查看Git的操作记录：



## 创建本地仓库

命令栏：Git: Initialize Repository

# 实例演示：Django-VSC-Git-Github-Azure Web App

## 目标

1. 在本地创建Django实例√
2. 将实例提交到本地Git√
3. 将代码上传到Github√
4. 将代码发布到Azure
5. 通过VSC AZ扩展发布√
6. 通过Github发布
7. 修改代码更新到Azure

## 本地Git管理

### 创建工作区DeployTestDemo√

### 创建远程仓库Github√

### 创立本地仓库√

### 创立远程仓库并获取Github仓库√

## 创建本地Django实例

### 配置VSC环境√

#### 配置Python虚拟环境√

#### 选择解析器√

### 安装Django并调试

#### 安装Django√

#### 创建项目√

#### 创建应用√

#### 本地调试√

## 提交Git并推送到Github

#### 添加并提交到Git√

#### 推送到Github√

## 发布到Azure

### 通过VSC Azure发布

#### 创建Azure Web App√

#### 发布前修改Django设置√

Debug=Fase

ALLOWED\_HOSTS = ['\*'] | 默认设置为[]， 否则部署后出现:( Application Error

#### 创建requirements.txt√

否则出现Application Error

#### 部署App到AZ√

### 通过Github发布

#### 在本地添加并提交Git√

#### 推送到Github√

#### 在AZure门户使用Github部署×

## 更新文件发布到Azure√

### 通过VSC Azure更新√

#### 修改文件√

#### 部署到Azure√

### 通过Github更新√

#### 修改文件√

#### 提交Git

#### 使用Github更新×

## 总结：

### 步骤：

创建本地GIt => 创建本地Django 实例 => 推送到Github => 通过VSC发布到AZure

### 下一步：

了解Github 发布到Azure的设置

了解Azure Devops的代码存储、发布到Azure。

# END