

第二周 关于Git

(廖雪峰官方教程的一点总结🐼)

一、Git是什么

- **官方语言：Git是目前世界上最先进的分布式版本控制系统（没有之一）**

那git能干什么呢，主要是以下几个用处：

1. Git 会跟踪每个文件的修改，删除，以便任何时候都可以追踪历史或者在将来某一时刻可以还原（版本控制 最基本）
2. Git的分布式使用的分支管理操作简单，安全可靠

- **git的诞生**

- 说起git的诞生着实不得不让人直呼一声nb！

Linux的创作者仅仅用了两周的时间用**C语言**写出了git！（两周。。同样是两周我闷在家里自我隔离TAT，别人写出了如今最流行的分布式版本控制系统，肃然起敬啊。。）

- **集中式/分布式版本控制系统**

1. **集中式分布系统**有一个大型的中央处理器所有的数据、运算、处理任务全部在中央计算机系统上完成（虽然不太容易坏，不过确实它坏了大家就都完了。。）
2. 每一个终端用来输入输出。
3. 集中式版本控制系统最大的毛病就是必须联网才能工作。
4. 终端多的时候速度会变慢
5. 如果终端用户有不同的需要，要对每个用户的程序和资源做单独的配置，在集中式系统上做起来比较困难，而且效率不高。

pk

1. **分布式版本控制系统**没有“中央服务器”，每个电脑上都是一个完整的版本库，不需要联网
2. 分布式版本控制系统的安全性要高很多，因为每个人电脑里都有完整的版本库，当一台机器发生故障时，可以使用另一台主机的备份。简单的说就是有福同享，有难不同当😁。
3. 用户可以根据自己的需要在自己的主机上安装不同的操作系统、应用软件，使用不同的服务，不再像集中式计算机系统那样受限于中央计算机的功能。

当然分布式系统除了git还有别的，但是git还是最快最简单也最流行~

二、如何安装git

这个这里就不详细写了😁

廖雪峰官方：<https://www.liaoxuefeng.com/wiki/896043488029600/896067074338496>

这里介绍的很详细欧，不行还可以Google，bing，百度，360等。。。

三、版本库

就是储存项目的地方（他就可以追踪还原历史版本啦）

• 首先我们来创建一个仓库

- 打开git (这里我用的是windows 别骂了o(T^To)我知道辣鸡啦)
- *这里我新建了一个名为second的文件夹

```
86153@MINGW64 ~ (master) $ cd second
86153@MINGW64 ~/second (master) $ git init
Initialized empty Git repository in C:/Users/86153/second/.git/
```

1. 选择一个合适的文件夹

2. 用git init命令让他创建一个带有.git的文件

- 然后在这个second文件夹里就会有一个.git的文件，但是一般自动隐藏了，需要取消隐藏



• 好的，然后我们可以在这个仓库上传我们的文件了

- 首先我们把需要上传的文件都写进second里面 (也可子文件夹)
- @@这里我创建一个名为test的txt

```
86153@MINGW64 ~/second/.git (GIT_DIR!) $ cd ../
86153@MINGW64 ~/second (master) $ touch test.txt
```

首先退出.git 这里用cd ../返回上一级

然后创建一个test文本

- 更改test里的内容

test.txt - 记事本

- 文件(F) 编辑(E) 格式(O) 查看(V) 帮

这是一个text

- 然后把文件添加到版本库 (git add)
- 但是没有任何的显示@@ 嗯那就对了√
- 这里可以反复多次使用添加很多文件欧
- emmmm, 偷偷告诉你 git add . 可以把文件夹所有文件一下子添加进去hhh

```
86153@MINGW64 ~/second (master) $ git add test.txt
86153@MINGW64 ~/second (master) $ git commit -m "把test上传到版本库"
[master (root-commit) 79fb4e4] 把test上传到版本库
1 file changed, 1 insertion(+)
create mode 100644 test.txt
```

添加test文本

- 接着把文件提交到仓库----> `git commit -m` 这里一定要写说明!! 为了让你和你的小粉丝都知道这是一个什么文件😊"
- 然后他就会告诉我们👉1.一个文件被改动 (添加的文件) 2.插入了1行内容 (因为我的test里有1行内容)

四、猜猜我变了没有啊

- 现在来看看 `git status` 命令

```
86153@MINGW64 ~/second (master)
$ git status
On branch master
nothing to commit, working tree clean


86153@MINGW64 ~/second (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   test.txt
no changes added to commit (use "git add" and/or "git commit -a")
```

这里我没有做修改

这里我修改了test的内容

这里提示我还没有提交

- 当我修改了test的内容时候

 test.txt - 记事本

文件(E) 编辑(E) 格式(O) 查看(V) 帮助(H)

- 这是一个text
测试修改文本

- 上面的命令输出告诉我们, `readme.txt` 被修改过了, 但还没有提交我的修改。
- 那我们怎么用git知道修改了什么内容呢

```
86153@MINGW64 ~/second (master)
$ git diff test.txt
diff --git a/test.txt b/test.txt
index 1e82f78..483595f 100644
--- a/test.txt
+++ b/test.txt
@@ -1,1,2 @@
-这是一个text
\ No newline at end of file
+这是一个text
+测试修改文本
\ No newline at end of file
```

使用这个命令就可以知道有什么不同啦

这里告诉我将xxx修改成了xxxx😊

- 当我们再次add之后输入 `git status` 就会看见它告诉我们准备提交要修改的内容了

```
$ git add test.txt
```

```
86153@MINGW64 ~/second (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   test.txt
```

- 然后提交 就会告诉我们 +了两个地方 -了一个地方

```
86153@MINGW64 ~/second (master)
$ git commit -m"新的提交"
[master bb87995] 新的提交
1 file changed, 2 insertions(+), 1 deletion(-)
```

五、以前的我

版本回退是git最基础的一个功能

- 首先我们要先看看有多少个版本

```
86153@MINGW64 ~/second (master)
$ git log
commit bb87995f5e4a3f00be59d32bebadf1cec89b8c20 (HEAD -> master)
Author: 2019210924 <2900624398@qq.com>
Date: Tue Mar 3 09:22:05 2020 +0800
    新的提交

commit ee07f4a74acc8c8691f2dcdb71b41e9daa5285e
Author: 2019210924 <2900624398@qq.com>
Date: Tue Mar 3 08:52:13 2020 +0800
    把test上传到版本库

commit 79fb4e498c58a386bd43f465c82aaf269cf133e6
Author: 2019210924 <2900624398@qq.com>
Date: Tue Mar 3 08:31:50 2020 +0800
    把test上传到版本库'
```

用这个命令

好的我提交了3次

- `git log --pretty=oneline` (前面一大串乱码样的东西就是版本号)

```
86153@MINGW64 ~/second (master)
$ git log --pretty=oneline
bb87995f5e4a3f00be59d32bebadf1cec89b8c20 (HEAD -> master) 新的提交
ee07f4a74acc8c8691f2dcdb71b41e9daa5285e 把test上传到版本库
79fb4e498c58a386bd43f465c82aaf269cf133e6 把test上传到版本库'
```

如果嫌刚刚一堆太多了，用这个命令就只看见版本号和注释啦

- 好的，现在我们可以来回退啦
- 在Git中，用 `HEAD` 表示当前版本，上一个版本就是 `HEAD^`，上上一个版本就是 `HEAD^^`，当然往上100个版本写100个 `^` 比较容易数不过来，所以写成 `HEAD~100`。

```
86153@MINGW64 ~/second (master)
$ git reset --hard HEAD^
HEAD is now at ee07f4a 把test上传到版本库'
```

- 这里就回退到上一个版本啦
- 那么现在我后悔了怎么办
- 害！可是世上没有后悔药啊T.T
- hhh但是git有🐼
- 方法一：如果刚刚的命令窗口还在，可以看见版本号

```
86153@MINGW64 ~/second (master)
$ git reset --hard bb8799
HEAD is now at bb87995 新的提交
```

输入版本号，那么它又回了！

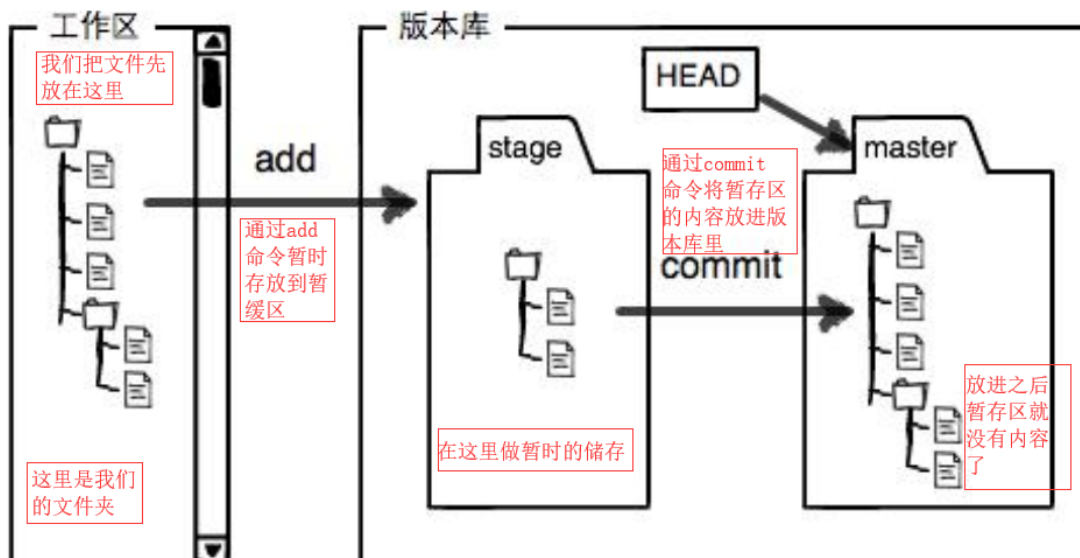
- 方法二：万一我把窗口关了那怎么办呀

```
86153@MINGW64 ~/second (master)
$ git reflog
bb87995 (HEAD -> master) HEAD@{0}: reset: moving to bb8799
ee07f4a HEAD@{1}: reset: moving to HEAD^
bb87995 (HEAD -> master) HEAD@{2}: commit: 新的提交
ee07f4a HEAD@{3}: commit: 把test上传到版本库
79fb4e4 HEAD@{4}: commit (initial): 把test上传到版本库'
```

那我们在这个命令可以看见我们之前的操作欧，
那我们又可以看见版本号啦

六、工作区与暂缓区

- 这里引用了廖雪峰老师的图，加了一点注释☺



七、强大的管理修改

Git跟踪并管理的是修改，而非文件！

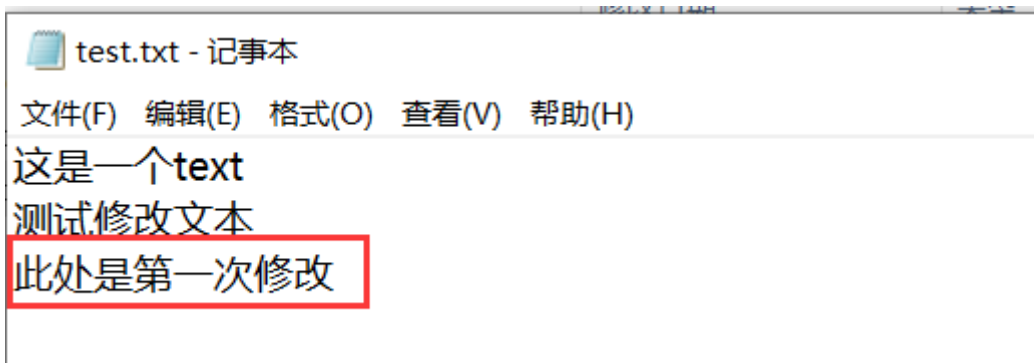
为什么这样说呢（此处引用廖雪峰的例子）

假设我们做出这样的修改：

第一次修改 -> `git add` -> 第二次修改 -> `git commit`

那版本库里的最新版本是什么呢

- 首先修改文本

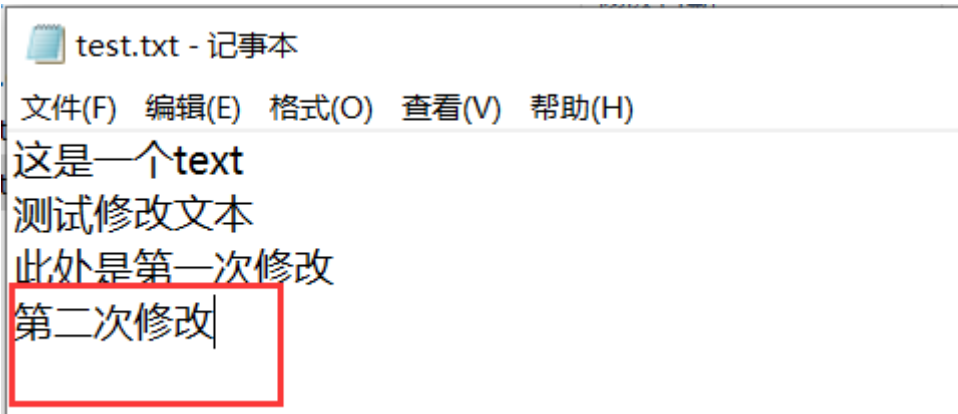


- 添加

```
86153@MINGW64 ~/second (master)
$ git add test.txt

86153@MINGW64 ~/second (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   test.txt
```

- 再修改



- 现在提交 再查看一下

```
86153@MINGW64 ~/second (master)
$ git commit -m"我提交了"
[master ec15d11] 我提交了
 1 file changed, 2 insertions(+), 1 deletion(-)

86153@MINGW64 ~/second (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   test.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

- 噢，看来第二次修改没有被提交

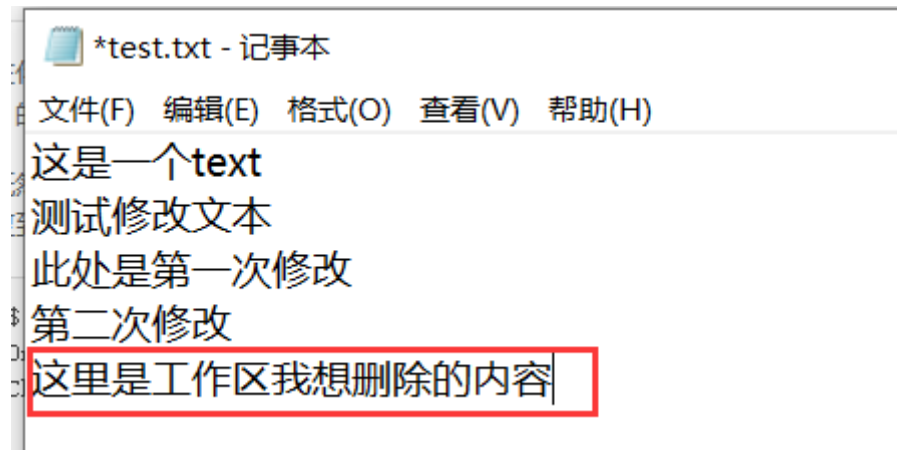
- 其实在六、的那个图里我们就知道，提交的只是在暂缓区的东西而不是原来文件里的东西，这大概就是廖雪峰老师说的管理修改而不是管理文件吧

那看来每次修改之后就要add，commit，修改的内容才能提交到版本库

八、撤销修改

1. 丢弃工作区文件 用命令 `git restore`

- 修改文本



- 这里提示我们使用 `git restore` 可以撤销工作区的修改

```
86153@MINGW64 ~/second (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   test.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

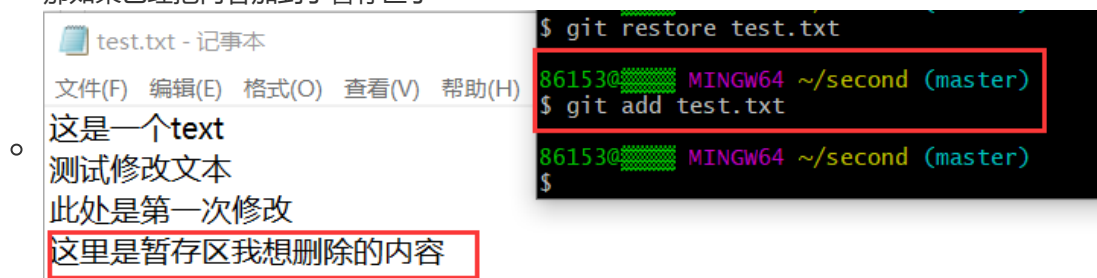
- 好的我们来输入一下



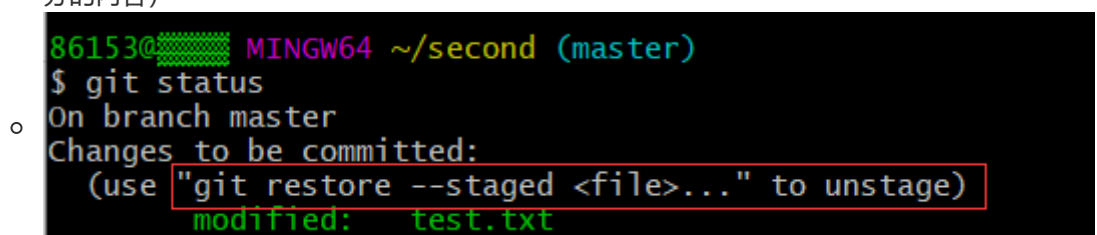
现在我们发现它回到了上一次我们提交的内容

2. 丢弃暂存区的修改 第一步用命令 `git restore --staged` 第二步用命令 `git restore`

- 那如果已经把内容加到了暂存区了



- 然后我们来查看一下
- 它提示我们用这条语句可以把暂存区的内容撤销掉放回工作区 (我觉得就是删掉暂存区这部分的内容)



- 下一步

- ```
86153@MINGW64 ~/second (master)
$ git restore --staged test.txt
```
- 删除暂存区内容
- ```
86153@MINGW64 ~/second (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   test.txt

no changes added to commit (use "git add" and/or "git commit -a")
```
- 现在提示我们暂存区没有内容而工作区被修改了
- 然后就跟1.一样的操作撤销工作区内容
 - test.txt - 记事本
 - 文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
 - 这是一个text
 - 测试修改文本
 - 此处是第一次修改
- ```
86153@MINGW64 ~/second (master)
$ git restore test.txt

86153@MINGW64 ~/second (master)
$
```

3. 已提交到版本库但没有推送到远程库（远程库内容在后面呀），看看五、版本回退叭

## 九、删除文件

这里是两种情况：

- 确实要删除版本库里的文件
    - 用命令 `git rm` 删掉，并且 `git commit`
- \* 这里我新建了一个t.txt并上传到库

```
86153@MINGW64 ~/second (master)
$ touch t.txt

86153@MINGW64 ~/second (master)
$ git add t.txt

86153@MINGW64 ~/second (master)
$ git commit -m"删除测试"
[master c96a2b0] 删除测试
1 file changed, 1 insertion(+), 1 deletion(-)
```

- 然后用 `git rm` 删掉并且提交，这样就确实被删了

- ```
86153@MINGW64 ~/second (master)
$ git rm t.txt
rm 't.txt'
```

- 是不小心在工作区删错了文件

- 输入这一个命令原本被删掉的文件就又回来了

```
86153@MINGW64 ~/second (master)
$ git restore t.txt
```

- 注：其实这个命令是将版本库的版本代替工作区的版本，所以是我们最后一次提交到库的内容

十、远程仓库！！！！

GitHub终于要登场了！

1. 添加远程库

首先注册一个GitHub账户吧（自己注册？？）

<https://github.com/>

然后创建SSH Key

第1步：创建SSH Key。在用户主目录下，看看有没有.ssh目录，如果有，再看看这个目录下有没有id_rsa和id_rsa.pub这两个文件，如果已经有了，可直接跳到下一步。如果没有，打开Shell（Windows下打开Git Bash），创建SSH Key：

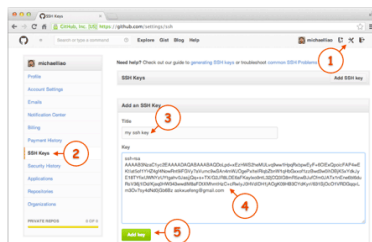
```
$ ssh-keygen -t rsa -C "your_email@example.com"
```

你需要把邮件地址换成你自己的邮件地址，然后一路回车，使用默认值即可，由于这个Key也不是用于军事目的，所以也无需设置密码。

如果一切顺利的话，可以在用户主目录下找到.ssh目录，里面有id_rsa和id_rsa.pub两个文件，这两个就是SSH Key的密钥对，id_rsa是私钥，不能泄露出去，id_rsa.pub是公钥，可以放心地告诉任何人。

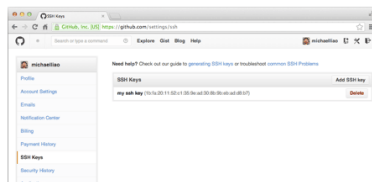
第2步：登陆GitHub，打开“Account settings”，“SSH Keys”页面：

然后，点“Add SSH Key”，填上任意Title，在Key文本框里粘贴id_rsa.pub文件的内容：

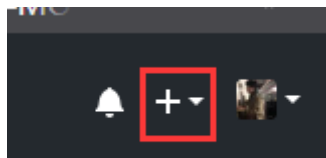


这里参考一下廖雪峰老师的创建方法

点“Add Key”，你就应该看到已经添加的Key：



登录之后右上角，创建一个新的储存库




然后：

创建一个新的仓库

资源库包含所有项目文件，包括修订历史记录。在其他地方已经有项目存储库了吗？[导入存储库](#)。

所有者

仓库名称 *

 2019210924 ▾

/ second ✓

这里写仓库的名字，其他默认就可以了

好的存储库名称简短而令人难忘。需要灵感吗？如何 [重新想象-米姆](#)？

说明 (可选)

第二周作业

☒ 公开

任何人都可以看到此存储库。您选择谁可以提交。

☐ 私人

您可以选择谁可以查看并提交到该存储库。

如果要导入现有存储库，请跳过此步骤。

☐ 使用自述文件初始化此存储库。

这将使您立即将存储库克隆到计算机。

添加.gitignore: 无 ▾

添加许可证: 无 ▾ ⓘ

创建仓库

GitHub, Inc. [条款](#) [隐私](#) [安全](#) [状态](#) [救命](#)

[联系GitHub](#) [价格](#) [API](#) [训练](#) [推广](#)

现在仓库还是空的，GitHub告诉我们，可以从这个仓库克隆出新的仓库，也可以把一个已有的本地仓库与之关联，然后，把本地仓库的内容推送到GitHub仓库。

...or create a new repository on the command line

```
echo "# second" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/2019210924/second.git
git push -u origin master
```

...or push an existing repository from the command line

```
git remote add origin https://github.com/2019210924/second.git
git push -u origin master
```

与本地仓库关联

...or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

Import code

跟着上面提示的操作，就可以把本地仓库的内容推送到远程仓库啦

```

86153@MINGW64 ~/second (master)
$ git remote add origin https://github.com/2019210924/second.git

86153@MINGW64 ~/second (master)
$ git push -u origin master
Enumerating objects: 43, done.
Counting objects: 100% (43/43), done.
Delta compression using up to 8 threads
Compressing objects: 100% (33/33), done.
Writing objects: 100% (43/43), 3.88 KiB | 496.00 KiB/s, done.
Total 43 (delta 4), reused 0 (delta 0)
remote: Resolving deltas: 100% (4/4), done.
To https://github.com/2019210924/second.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.

```





注：因为是第一次推送所以git push后面要加上 -u，以后的修改版本就不用啦

还有就是git是把远程库和本地库关联了起来呢，不仅仅是单纯的提交

现在仓库里就有内容啦

10 commits
Pull requests
Set up Actions
Dismiss

Branch: master
New pull request
Crea

	2019210924 this is rubbish
	rubbish.txt this is rubbish
	t.txt 删除测试
	test.txt bug is ok

Help people interested in this repository understand your project by adding a README.

这里还有一个警告！

SSH警告

当你第一次使用Git的 `clone` 或者 `push` 命令连接GitHub时，会得到一个警告：

```
The authenticity of host 'github.com (xx.xx.xx.xx)' can't be established.  
RSA key fingerprint is xx.xx.xx.xx.xx.  
Are you sure you want to continue connecting (yes/no)?
```

这是因为Git使用SSH连接，而SSH连接在第一次验证GitHub服务器的Key时，需要你确认GitHub的Key的指纹信息是否真的来自GitHub的服务器，输入 `yes` 回车即可。

Git会输出一个警告，告诉你已经把GitHub的Key添加到本机的一个信任列表里了：

```
Warning: Permanently added 'github.com' (RSA) to the list of known hosts.
```

这个警告只会出现一次，后面的操作就不会有任何警告了。

如果你实在担心有人冒充GitHub服务器，输入 `yes` 前可以对照GitHub的RSA Key的指纹信息是否与SSH连接给出的一致。

2.从远程库克隆

一般我们在日常工作中不会直接把项目库和远程库关联，而是克隆一份到远程库里。

先创建一个库

创建一个新的仓库

资源库包含所有项目文件，包括修订历史记录。在其他地方已经有项目存储库了吗？[导入存储库](#)。

所有者

 2019210924 ▾

仓库名称 *

/ demo ✓

好的存储库名称简短而令人难忘。需要灵感吗？如何 [可爱](#)，[两个星期](#)？

说明 (可选)

这是git说明

☒ 公开

任何人都可以看到此存储库。您选择谁可以提交。

这些写好

☐ 私人

您可以选择谁可以查看并提交到该存储库。

如果要导入现有存储库，请跳过此步骤。

☒ 使用自述文件初始化此存储库。

这将使您立即将存储库克隆到计算机。

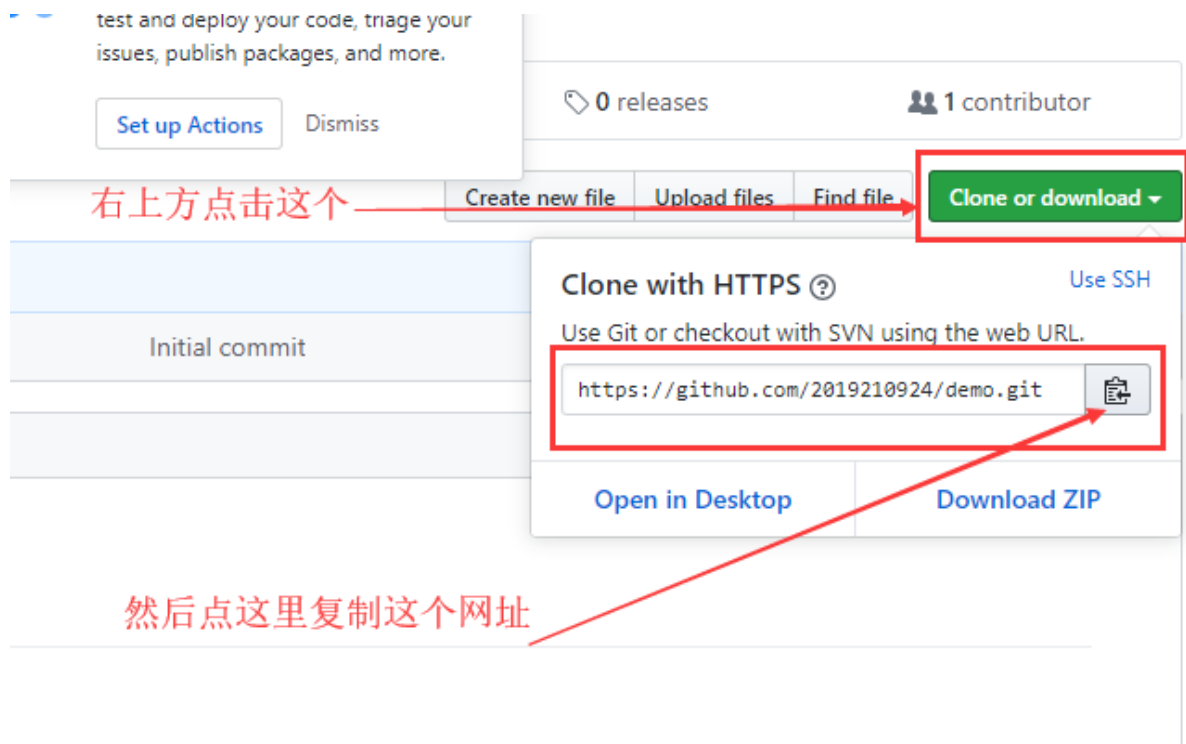
注意选上这个o，这样就有了一个readme在库里啦，通常用来介绍自己的项目。

添加.gitignore: 无 ▾

添加许可证: 无 ▾ ⓘ

创建仓库

创建好后:



下面就是本地操作啦:

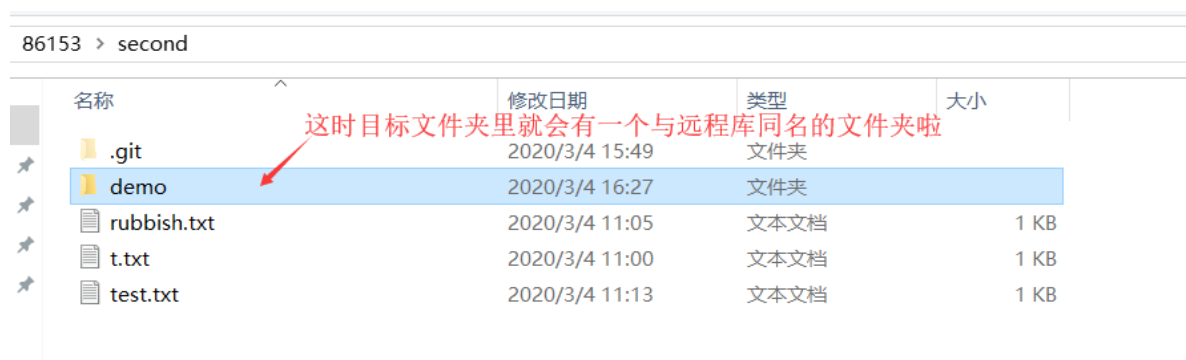
找到你要上传的文件夹项目, 右键点击文件夹 (注意: 不能选单个文件或者压缩包) 在选项里选择**Git Bash Here**

MINGW64:/c/Users/86153/second

```
86153@MINGW64 ~/second (master)
$ git clone https://github.com/2019210924/demo.git
Cloning into 'demo'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
```

弹出这个窗口之后敲 `git clone` 然后加上刚刚复制的网址

然后来看文件夹



然后将我们要上传的文件复制到demo文件夹里面

```
86153@MINGW64 ~/second (master) 切换目录
$ cd demo
86153@MINGW64 ~/second/demo (master)
$ git add .
```

上传

```

86153@MINGW64 ~/second/demo (master)
$ git commit -m "这是一个新的demo" 然后提交
[master fc24052] 这是一个新的demo
3 files changed, 4 insertions(+)
create mode 100644 rubbish.txt
create mode 100644 t.txt
create mode 100644 test.txt

86153@MINGW64 ~/second/demo (master)
$ git push -u origin master 这里上传到远程库里，后面有详细介绍的
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (5/5), 425 bytes | 425.00 KiB/s, done.
Total 5 (delta 0), reused 0 (delta 0)
To https://github.com/2019210924/demo.git
   9ed98e3..fc24052  master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.

```

这里要注意，第一次使用的时候要先配置好git上的用户名和邮箱

```

86153@MINGW64 ~/second/demo (master) 用户名
$ git config --global user.name "2019210924"

86153@MINGW64 ~/second/demo (master) 邮箱
$ git config --global user.email "2019210924@qq.com"

```

然后再提交就OK啦

十一、分支管理

- 什么是分支？相当于一根树枝，既与主干相连又不互相影响

就如廖雪峰所说：

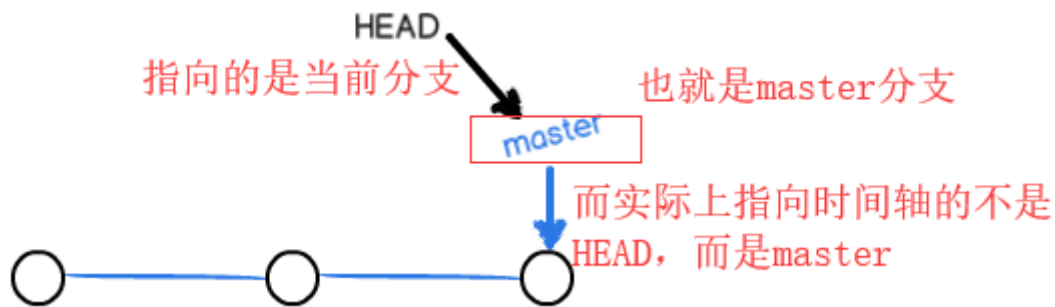
分支在实际中有什么用呢？假设你准备开发一个新功能，但是需要两周才能完成，第一周你写了50%的代码，如果立刻提交，由于代码还没写完，不完整的代码库会导致别人不能干活了。如果等代码全部写完再一次提交，又存在丢失每天进度的巨大风险。

现在有了分支，就不用怕了。你创建了一个属于你自己的分支，别人看不到，还继续原来的分支上正常工作，而你在自己的分支上干活，想提交就提交，直到开发完毕后，再一次性合并到原来的分支上，这样，既安全，又不影响别人工作。

十二、创建合并分支

在上面版本回退的地方，我们已经知道，git把每一次提交串成了一条时间线，也就是一个主分支。

(六、里有一个具体的版本库内部构成图)



现在我们来新建一个分支 `dev`

```
86153@MINGW64 ~/second (master)
$ git checkout -b dev
Switched to a new branch 'dev'

86153@MINGW64 ~/second (dev)
$ git branch
* dev
  master
```

使用这条语句创建一个分支，这时HEAD就指向这个分支了

这里可以查看所有的分支，其中当前正在使用的分支前面有一个*

然后我们可以修改我们的文件啦

test.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) ⋮

这是一个text
测试修改文本
此处是第一次修改
修改分支

然后提交

```
86153@MINGW64 ~/second (dev)
$ git add test.txt

86153@MINGW64 ~/second (dev)
$ git commit -m "我在dev分支里"
[dev 7e25271] 我在dev分支里
1 file changed, 2 insertions(+), 1 deletion(-)
```

这时他就提交到dev分支里了，但是master主分支里是没有变的

如果我们完成了分支里的内容，现在来把他合并到主分支吧

```
86153@MINGW64 ~/second (dev)
$ git checkout master
Switched to branch 'master'

86153@MINGW64 ~/second (master)
$ git merge dev
Updating 6252e6f..7e25271
Fast-forward
 test.txt | 3 ++-
1 file changed, 2 insertions(+), 1 deletion(-)
```

首先，先切换到主分支

然后用这个命令合并dev分支

这里告诉我们直接把master指向dev的当前提交，所以合并速度很快

然后我们就可以删除dev分支了


```
86153@MINGW64 ~/second (master)
$ git branch -d dev
Deleted branch dev (was 7e25271).

86153@MINGW64 ~/second (master)
$ git branch
* master
```

删除分支

这时只剩下一个主分支了

- 因为创建、合并和删除分支非常快，所以Git鼓励你使用分支完成某个任务，合并后再删掉分支，这和直接在 `master` 分支上工作效果是一样的，但过程更安全。

创建/切换分支的第二种命令：

创建并切换到新的 `dev` 分支，可以使用：

```
$ git switch -c dev
```

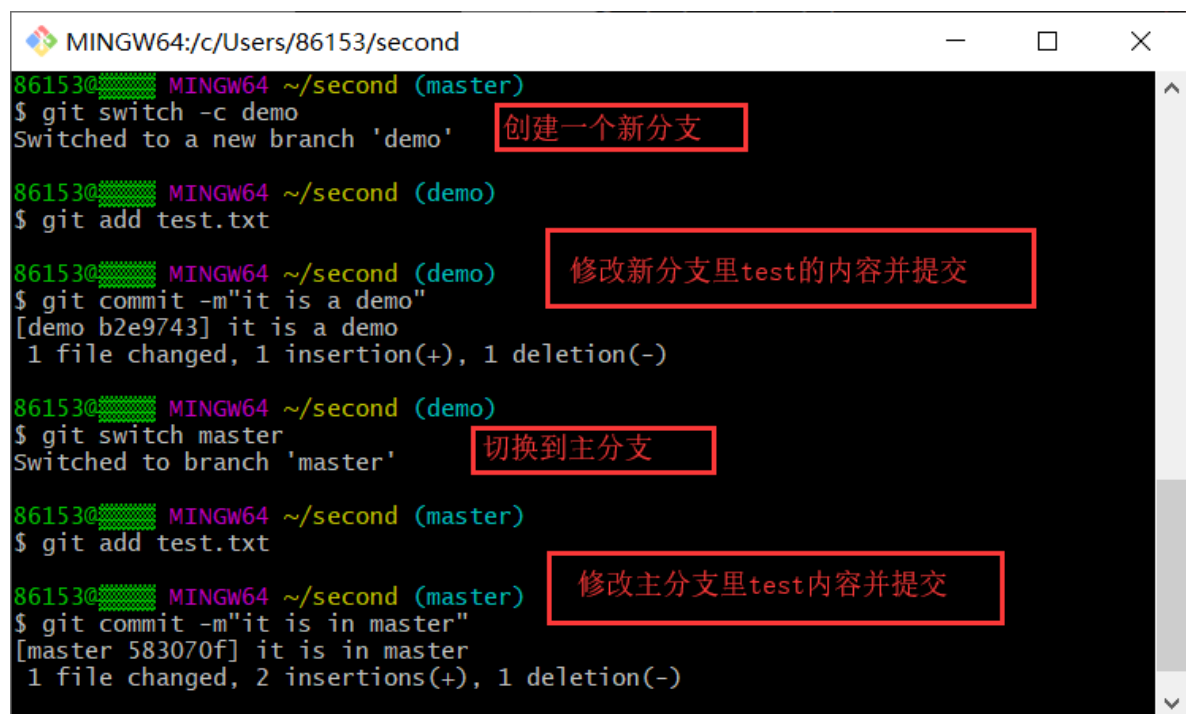
直接切换到已有的 `master` 分支，可以使用：

```
$ git switch master
```

十三、分支间的冲突

假设我们在一个分支上提交了内容，又在主分支（或者另一个分支）提交了另外的内容，那我们需要合并的时候会发生什么奇怪的事呢？

首先：



```
MINGW64:/c/Users/86153/second
86153@MINGW64 ~/second (master)
$ git switch -c demo
Switched to a new branch 'demo'

86153@MINGW64 ~/second (demo)
$ git add test.txt

86153@MINGW64 ~/second (demo)
$ git commit -m "it is a demo"
[demo b2e9743] it is a demo
1 file changed, 1 insertion(+), 1 deletion(-)

86153@MINGW64 ~/second (demo)
$ git switch master
Switched to branch 'master'

86153@MINGW64 ~/second (master)
$ git add test.txt

86153@MINGW64 ~/second (master)
$ git commit -m "it is in master"
[master 583070f] it is in master
1 file changed, 2 insertions(+), 1 deletion(-)
```

创建一个新分支

修改新分支里test的内容并提交

切换到主分支

修改主分支里test内容并提交

然后我们就可以开始试着合并了：

```

86153@MINGW64 ~/second (master)
$ git merge demo
Auto-merging test.txt
CONFLICT (content): Merge conflict in test.txt
Automatic merge failed; fix conflicts and then commit the result.

86153@MINGW64 ~/second (master|MERGING)
$ git status
On branch master
You have unmerged paths
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
    both modified:   test.txt
no changes added to commit (use "git add" and/or "git commit -a")

```

果然，合并的时候告诉我们冲突了

这里这个命令可以告诉我们冲突的文件

然后我们看一下test里的内容：

```

test.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
这是一个text
测试修改文本
此处是第一次修改
<<<<<<< HEAD
修改分支
这是主分支里的
=====
这是一个demo
>>>>>>> demo

```

git还告诉了我们不同分支的内容，害！

那现在我们就手动改一下吧

```

86153@MINGW64 ~/second (master|MERGING)
$ git add test.txt

修改test的内容然后再次提交

86153@MINGW64 ~/second (master|MERGING)
$ git commit -m"it is ok"
[master 8b2df6f] it is ok

```

这里用这个命令可以看见分支合并情况@@

```

86153@MINGW64 ~/second (master)
$ git log --graph
* commit 8b2df6f1fd21f0cee493814dc92eea39fe73bf2f4 (HEAD -> master)
Merge: 583070f b2e9743
Author: 2019210924 <2900624398@qq.com>
Date: Tue Mar 3 22:25:32 2020 +0800

    it is ok

* commit b2e9743ee9a6f2f493c717b5b8f1b3b26128d5d1 (demo)
Author: 2019210924 <2900624398@qq.com>
Date: Tue Mar 3 22:00:14 2020 +0800

    it is a demo

* commit 583070f94c420b2482c276054e5ece6bf2519533
Author: 2019210924 <2900624398@qq.com>
Date: Tue Mar 3 22:02:40 2020 +0800

    it is in master

```

十四、分支管理策略

上面我们提到合并分支时Git会用 `Fast forward` 模式，但这种模式下，删除分支后，会丢掉分支信息。

如果强制禁用 `Fast forward` 模式，Git就会在merge时生成一个新的commit，这样，从分支历史上就可以看出分支信息。

现在我们来实践一下8：

```

86153@MINGW64 ~/second (master)
$ git switch -c dev
Switched to a new branch 'dev'

86153@MINGW64 ~/second (dev)
$ git add test.txt

86153@MINGW64 ~/second (dev)
$ git commit -m"分支管理策略"
[dev 2170284] 分支管理策略
1 file changed, 1 insertion(+), 4 deletions(-)

86153@MINGW64 ~/second (dev)
$ git switch master
Switched to branch 'master'

```

首先我们创建一个分支，更改test的内容，然后提交，再切换回主分支

现在合并 dev 分支：

```

86153@MINGW64 ~/second (master)
$ git merge --no-ff -m"不用fast forward" dev
Merge made by the 'recursive' strategy.
test.txt | 5 +----
1 file changed, 1 insertion(+), 4 deletions(-)

86153@MINGW64 ~/second (master)
$ git log --graph --pretty=oneline --abbrev-commit
* 56249cf (HEAD -> master) 不用fast forward
* 2170284 (dev) 分支管理策略
* 8b2df6f it is ok
* b2e9743 (demo) it is a demo
* 583070f it is in master

```

这里加上 `--no-ff` 表示禁用 `fast forward`

因为要创建一个新的 commit，所以加上 `-m`

然后查看分支历史

这里我们就可以看见是由dev分支传到主分支里的而用了 `fast forward` 则是dev和主分支直接合并了，不信你试试在楼上 `fast forward` 后查看一下分支历史呀 😊

- 在正规的工作中，`master` 分支非常稳定，一般仅用来发布新版本，平时都不在上面干活呢，干活都在 `dev` 分支上。
- 也就是说，`dev` 分支是不稳定的，到某个时候，比如1.0版本发布时，再把 `dev` 分支合并到 `master` 上，在 `master` 分支发布1.0版本。
- 你和你的小伙伴们每个人都在 `dev` 分支上干活，然后每个人再创建自己的分支，时不时地往 `dev` 分支上合并就可以了。

十五、Bug分支

日常工作中我们一般会遇到以下几个问题吧：

1. 主分支发布的版本遇到了bug 0.0
2. 但是手头工作ing，要存档
3. 主分支的bug，工作ing的分支也有

首先先说说存档吧：

```
86153@MINGW64 ~/second (working) 上面我创建了一个working分支
$ git add work.txt 将工作内容添加到工作区，但是还没提交
86153@MINGW64 ~/second (working)
$ git stash 然后我们用这个命令将它存档
Saved working directory and index state WIP on working: 56249cf
86153@MINGW64 ~/second (working)
$ git status 最后查看工作区的内容发现已经干净了 😊
On branch working
nothing to commit, working tree clean
```

现在我们就可以在有bug的分支上创建新的分支来修复啦：

test.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

在分支管理策略
我修复了bug

```
86153@MINGW64 ~/second (working)
$ git switch master 切换到主分支
Switched to branch 'master'

86153@MINGW64 ~/second (master)
$ git switch -c bug 在主分支下创建用来修复bug的分支
Switched to a new branch 'bug'

86153@MINGW64 ~/second (bug)
$ git add test.txt

86153@MINGW64 ~/second (bug) 修复bug并提交 😊
$ git commit -m "bug is ok"
[bug 6419d45] bug is ok
1 file changed, 2 insertions(+), 1 deletion(-)

86153@MINGW64 ~/second (bug)
$ git switch master 切换到主分支
Switched to branch 'master'

86153@MINGW64 ~/second (master)
$ git merge --no-ff -m "master has no bug" bug
Merge made by the 'recursive' strategy.
test.txt | 3 ++-
1 file changed, 2 insertions(+), 1 deletion(-) 合并主分支与bug分支
```

修复完了之后我们回到working分支来继续我们的工作8：

```
86153@MINGW64 ~/second (working) 切换到工作分支
$ git stash pop
On branch working
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   work.txt

Dropped refs/stash@{0} (7396baaa0627eed2e9520c2839d2beea2f361a35)
```

用这个指令可以恢复工作区的东西

```
86153@MINGW64 ~/second (working)
$ git stash list
```

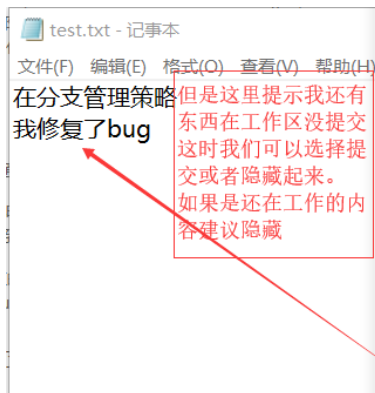
查看列表发现是空的，已经没有了储存工作区内容的地方了

注：其实我们开始是创建了一个地方来储存工作区的内容

```
86153@MINGW64 ~/second
$ git stash list
stash@{0}: WIP on working: 5
```

然后假如我们工作的地方也有上面修改的bug，那怎样快捷的修改呢？

用 `git cherry-pick` 命令，把bug提交的修改“复制”到当前分支



```
86153@MINGW64 ~/second (working)
$ git cherry-pick 6419d45 用这个命令加上bug提交时的commit号就可以直接
error: your local changes would be overwritten by cherry-pick. 改动了
hint: commit your changes or stash them to proceed.
fatal: cherry-pick failed

86153@MINGW64 ~/second (working)
$ git commit -m "it is a test" 我选择了提交
[working b942272] it is a test
2 files changed, 1 insertion(+), 1 deletion(-)
delete mode 100644 t.txt
create mode 100644 work.txt

86153@MINGW64 ~/second (working)
$ git cherry-pick 6419d45 再次输入这个语句，这时候bug就修复啦
[working f997a3d] bug is ok
Date: Wed Mar 4 10:00:43 2020 +0800
1 file changed, 2 insertions(+), 1 deletion(-)
```

十六、强行删除分支

当我们开了一个分支working但是还没work完就不想要这个分支了怎么办？？

这时这个分支没有被合并过，我们要通过 `git branch -D` 强行删除。

```

86153@MINGW64 ~/second (master)  创建一个分支
$ git switch -c feature
Switched to a new branch 'feature'

86153@MINGW64 ~/second (feature)  修改内容
$ git add test.txt

86153@MINGW64 ~/second (feature)  提交
$ git commit -m"dustbin"
[feature f198abb] dustbin
1 file changed, 2 insertions(+), 1 deletion(-)

切换到主分支
86153@MINGW64 ~/second (feature)
$ git switch master
Switched to branch 'master'

86153@MINGW64 ~/second (master)  如果用以前删除分支的方法会发现是不行的，它会提示我们没有合并该分支
$ git branch -d feature
error: The branch 'feature' is not fully merged.
If you are sure you want to delete it, run 'git branch -D feature'.

86153@MINGW64 ~/second (master)
$ git branch -D feature
Deleted branch feature (was f198abb).

```

所以这里要用大写D!

十七、多人协作

记得上面我们已经将本地仓库和远程库连起来了：

```

86153@MINGW64 ~/second (master)
$ git remote  这里可以查看远程库的信息
origin

86153@MINGW64 ~/second (master)
$ git remote -v  这里还显示了origin的地址
origin https://github.com/2019210924/second.git (fetch)
origin https://github.com/2019210924/second.git (push)

```

• 推送分支

就是当我们修改了本地分支需要提交推送到远程库的时候啦，这里可以推送主分支也可以推送其他分支，看自己的需要哦（当然主分支是一定要的啦😊）


```

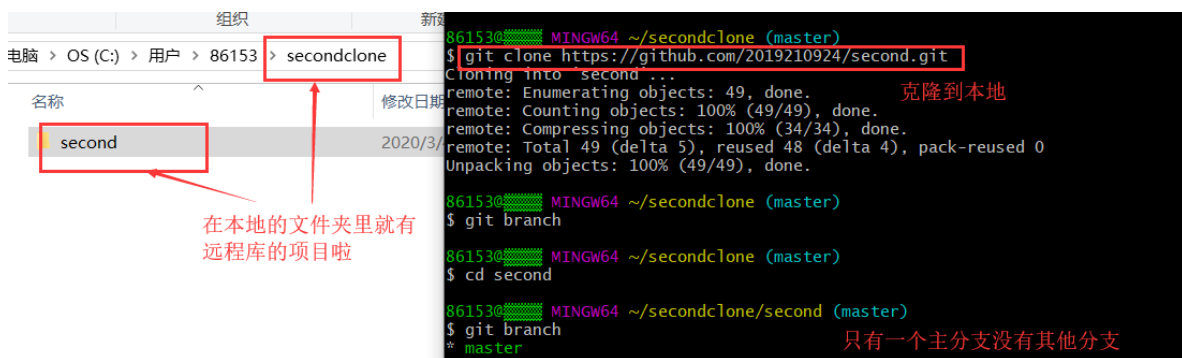
86153@MINGW64 ~/second (master)
$ git push origin master
Logon failed, use ctrl+c to cancel basic credential prompt.
Everything up-to-date

86153@MINGW64 ~/second (master)
$ git push origin working
Enumerating objects: 8, done.
Counting objects: 100% (8/8), done.
Delta compression using up to 8 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (6/6), 559 bytes | 559.00 KiB/s, done.
Total 6 (delta 0), reused 0 (delta 0)
remote:
remote: Create a pull request for 'working' on GitHub by visiting:
remote: https://github.com/2019210924/second/pull/new/working
remote:
To https://github.com/2019210924/second.git
* [new branch] working -> working

```

• 抓取分支

在多人协作中一般都是大家从远程库把主分支clone下来



The image shows a file explorer window on the left and a terminal window on the right. In the file explorer, the path is '电脑 > OS (C:) > 用户 > 86153 > secondclone'. A red box highlights the 'secondclone' directory. Below it, a table shows a file named 'second' with a modification date of '2020/3/'. A red arrow points from the 'second' file to the terminal window. The terminal window shows the following commands and output:

```

86153@MINGW64 ~/secondclone (master)
$ git clone https://github.com/2019210924/second.git
Cloning into 'second'...
remote: Enumerating objects: 49, done.
remote: Counting objects: 100% (49/49), done.
remote: Compressing objects: 100% (34/34), done.
remote: Total 49 (delta 5), reused 48 (delta 4), pack-reused 0
Unpacking objects: 100% (49/49), done.
克隆到本地

86153@MINGW64 ~/secondclone (master)
$ git branch
* master

86153@MINGW64 ~/secondclone (master)
$ cd second

86153@MINGW64 ~/secondclone/second (master)
$ git branch
* master
只有一个主分支没有其他分支

```

在本地的文件夹里就有远程库的项目啦

然后创新的分支来改改改，然后再时不时的push到远程库里

```

86153@MINGW64 ~/secondclone/second (master)
$ git switch -c working origin/working
Switched to a new branch 'working'
Branch 'working' set up to track remote branch 'working' from 'origin'.
现在你的小伙伴创建了一个工作分支

86153@MINGW64 ~/secondclone/second (working)
$ git add test.txt

86153@MINGW64 ~/secondclone/second (working)
$ git commit -m "change it"
[working e91aa1a] change it
1 file changed, 2 insertions(+), 1 deletion(-)

86153@MINGW64 ~/secondclone/second (working)
$ git push origin working
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 312 bytes | 312.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/2019210924/second.git
f997a3d..e91aa1a working -> working
修改并提交

```

那没了，如果我也改了，我的小伙伴也改了那怎么办！！


```

86153@MINGW64 ~/second (master)
$ git switch working
Switched to branch 'working'

86153@MINGW64 ~/second (working)
$ git add test.txt
我也改了!

86153@MINGW64 ~/second (working)
$ git commit -m "also change"
[working 22eeb63] also change
1 file changed, 2 insertions(+), 1 deletion(-)

86153@MINGW64 ~/second (working)
$ git push origin working
To https://github.com/2019210924/second.git
! [rejected]        working -> working (fetch first)
error: failed to push some refs to 'https://github.com/2019210924/second.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
你看冲突了!

```

没事~我们可以先git pull把小伙伴的最新提交抓下来，然后合并再传上去。

```

86153@MINGW64 ~/second (working)
$ git pull
这里我拿下来了
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/2019210924/second
f997a3d..e91aa1a  working    -> origin/working
There is no tracking information for the current branch.
Please specify which branch you want to merge with.
See git-pull(1) for details.

git pull <remote> <branch>

If you wish to set tracking information for this branch you can do so with:

git branch --set-upstream-to=origin/<branch> working

```

但是要注意o，我们抓下来的是远程库的分支和本地库的是不一样的，所以我们要设置他们的链接o

```

86153@MINGW64 ~/second (working)
$ git branch --set-upstream-to=origin/working working
Branch 'working' set up to track remote branch 'working' from 'origin'.
设置

86153@MINGW64 ~/second (working)
$ git pull
Auto-merging test.txt
CONFLICT (content): Merge conflict in test.txt
Automatic merge failed; fix conflicts and then commit the result.
√ 这里才成功拿下来

```

然后再pull（如果合并有冲突记得怎么做吗@@ 不记得啦？那看看十三、）

好的现在就完事啦

```

86153@MINGW64 ~/second (working|MERGING)
$ git add test.txt

86153@MINGW64 ~/second (working|MERGING)
$ git commit -m"ok"
[working cab42bb] ok

86153@MINGW64 ~/second (working)
$ git push origin working
Enumerating objects: 10, done.
Counting objects: 100% (10/10), done.
Delta compression using up to 8 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (6/6), 620 bytes | 620.00 KiB/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To https://github.com/2019210924/second.git
e91aa1a..cab42bb working -> working

```

现在成功上传啦

十八、*Rebase

git rebase是用来干嘛的呢，就是用来整理分支的，因为当有很多人的时候大家相互交错的上传就会让提交很混乱，然后如果要找回想要的版本就很困难。

但是在我们一般的开发中其实并不是特别常见的命令

然后因为我没有创建很多分支😁这里贴一个大佬的详解

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

当然还有我们廖雪峰老师的啦😁

<https://www.liaoxuefeng.com/wiki/896043488029600/1216289527823648>

十九、标签管理

标签 (tag) 其实就是一个版本号，跟commit绑定在一起而比commit号简单有意义。

打标签也很简单的（给当前版本打）：

```

86153@MINGW64 ~/second (master)
$ git switch master
Already on 'master'
Your branch is up to date with 'origin/master'.

86153@MINGW64 ~/second (master)
$ git tag v1.0

86153@MINGW64 ~/second (master)
$ git tag
v1.0

```

切换到要打标签的分支

打标签

查看打过的所有标签

这是标签号

那如果想给以前的版本打标签呢：

```

86153@MINGW64 ~/second (master)
$ git log --pretty=oneline --abbrev-commit
fb4e4ac (HEAD -> master, tag: v1.0, origin/master) this is rubbish
23cbfe9 master has no bug
6419d45 (bug) bug is ok
56249cf 不用fast forward
2170284 分支管理策略
8b2df6f it is ok
583070f it is in master
b2e9743 it is a demo
7e25271 我在dev分支里
6252e6f 删除测试
de3fa0b 我删了
c96a2b0 删除测试
ec15d11 我提交了
bb87995 新的提交
ee07f4a 把test上传到版本库
79fb4e4 把test上传到版本库'

86153@MINGW64 ~/second (master)
$ git tag v0.9 23cbfe9

86153@MINGW64 ~/second (master)
$ git tag
v0.9
v1.0

```

我们可以先找到所有的提交的commit号

然后在标签号后面加上commit号就可以了

注：标签不是按时间顺序列出，而是按字母排序的。

```

86153@MINGW64 ~/second (master)
$ git tag -a a1.0 -m "这里可以有说明文字" 6252e6f

86153@MINGW64 ~/second (master)
$ git tag
a1.0
v0.9
v1.0

```

字母顺序 -a +版本号， -m" " 可有文字说明欧

注：用命令 `git show +版本号` 可以看到说明文字：

```

86153@MINGW64 ~/second (master)
$ git show v1.0
commit fb4e4ac91f60a5f0e63c2d42bb058999d044fbf3 (HEAD -> master, tag: v1.0, origin/master)
Author: 2019210924 <2900624398@qq.com>
Date: Wed Mar 4 11:02:55 2020 +0800

    this is rubbish

diff --git a/rubbish.txt b/rubbish.txt
new file mode 100644
index 0000000..ec7cbc7
--- /dev/null
+++ b/rubbish.txt
@@ -0,0 +1 @@
+垃圾
\ No newline at end of file

```

如果标签打错了：

```

86153@MINGW64 ~/second (master)
$ git tag -d v1.0
Deleted tag 'v1.0' (was fb4e4ac)

```

删除标签

然后我们打的标签只是在本地，那么怎样推送到远程库呢？

```
86153@MINGW64 ~/second (master)
$ git tag v1.1
我重新打了一个标签

86153@MINGW64 ~/second (master)
$ git push origin v1.1
fatal: HttpRequestException encountered.
这是推送某一个标签
Total 0 (delta 0), reused 0 (delta 0)
To https://github.com/2019210924/second.git
* [new tag]          v1.1 -> v1.1

86153@MINGW64 ~/second (master)
$ git push origin --tags
fatal: HttpRequestException encountered.
这是推送所有标签
Enumerating objects: 1, done.
Counting objects: 100% (1/1), done.
Writing objects: 100% (1/1), 192 bytes | 192.00 KiB/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To https://github.com/2019210924/second.git
* [new tag]          a1.0 -> a1.0
* [new tag]          v0.9 -> v0.9
```

那如果要删除标签呢？

```
86153@MINGW64 ~/second (master)
$ git tag -d v0.9
Deleted tag 'v0.9' (was 23cbfe9)
先在本地仓库删除
```

然后：

```
86153@MINGW64 ~/second (master)
$ git push origin :refs/tags/v0.9
输入这行命令删除
To https://github.com/2019210924/second.git
- [deleted]          v0.9
```

这样标签就被删除了

(最后：最后来康康这个配置别名8 hhh)

干啥啥不行偷懒我最行嘻嘻嘻

status, branch。。。这些一堆一堆的单词对我这种英语渣渣真是要晕了

那怎么办呢？

嘻嘻嘻

86153 > second > .git				
名称	修改日期	类型	大小	
hooks	2020/3/2 10:41	文件夹		
info	2020/3/2 10:41	文件夹		
logs	2020/3/3 8:31	文件夹		
objects	2020/3/4 15:10	文件夹		
refs	2020/3/4 11:42	文件夹		
COMMIT_EDITMSG	2020/3/4 15:49	文件	1 KB	
config	2020/3/4 15:55	文件	1 KB	
description	2020/3/2 10:41	文件	1 KB	
HEAD	2020/3/4 14:58	文件	1 KB	
index	2020/3/4 14:58	文件	1 KB	
ORIG_HEAD	2020/3/4 10:21	文件	1 KB	

这里要注意一定要注意T.T

不要乱改.git的文件（但是为了偷懒我还是以身犯险了🐼）

```

config - 记事本
文件(E) 编辑(E) 格式(O) 查看(V) 帮助(H)
[core]
    repositoryformatversion = 0
    filemode = false
    bare = false
    logallrefupdates = true
    symlinks = false
    ignorecase = true
[remote "origin"]
    url = https://github.com/2019210924/second.git
    fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
    remote = origin
    merge = refs/heads/master
[alias]
    ci = commit
    lg = log --color --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(b

```

然后我们来试一下8

啊这真是太爽了~~

```

86153@MINGW64 ~/second (master)
$ git lg
* fb4e4ac - (HEAD -> master, tag: v1.1, origin/master) this is rubbish (5 hours ago) <2019210924>
* 23cbfe9 - master has no bug (6 hours ago) <2019210924>
| \
| * 6419d45 - (bug) bug is ok (6 hours ago) <2019210924>
| /
* 56249cf - 不用fast forward (7 hours ago) <2019210924>
| \
| * 2170284 - 分支管理策略 (7 hours ago) <2019210924>
| /
* 8b2df6f - it is ok (18 hours ago) <2019210924>
| \
| * b2e9743 - it is a demo (18 hours ago) <2019210924>
* | 583070f - it is in master (18 hours ago) <2019210924>
| /
* 7e25271 - 我在dev分支里 (19 hours ago) <2019210924>
* 6252e6f - (tag: a1.0) 删除测试 (28 hours ago) <2019210924>
* de3fa0b - 我删了 (28 hours ago) <2019210924>
* c96a2b0 - 删除测试 (28 hours ago) <2019210924>
* ec15d11 - 我提交了 (29 hours ago) <2019210924>
* bb87995 - 新的提交 (31 hours ago) <2019210924>
* ee07f4a - 把test上传到版本库 (31 hours ago) <2019210924>
* 79fb4e4 - 把test上传到版本库' (31 hours ago) <2019210924>

```

(这里是整理的所有上面涉及的命令)

git命令

- cd 文件名--进入子目录
- cd ../ ---返回上一级
- git init --创建版本库（仓库）
- touch 文件名---创建所需文件
- git add 文件 ---添加到版本库
- git add .* ----添加全部文件到版本库
- git commit -m"xxx"---把文件提交到版本库
- git status ---掌握仓库当前的状态（是否修改）
- git diff ----查看不同
- git log ---查看版本数
- git log --pretty=oneline ---简易版看版本数
- git reset --hard HEAD^ ---回退版本
- git reset --hard commit_id ---回到某版本
- git log ----查看提交历史
- git reflog ---查看命令历史，确定要回到未来的哪个版本
- git restore ---将版本库版本代替工作区版本
- git rm ----删除文件
- git check -b /git switch -c ----创建/切换分支
- git branch -----查看所有分支
- git merge ----合并分支
- git branch -d -----删除分支
- git status ---查看冲突
- git log --graph-----查看时间轴
- git stash-----存档
- git stash pop----恢复
- git cherry-pick+commit号-----复制别的提交到该分支
- git branch -D+xxx-----强行删除
- git remote -v----查看远程库地址信息
- git push origin -----推送分支到远程库
- git tag ----打标签
- git tag -d ---删标签
- git show ----版本说明

