

如果是第一次安装 git 并且上传代码，请在 VS Code 的命令终端 (ctrl+`) 中进行一下操作，以使自己的电脑识别你的 github 账户

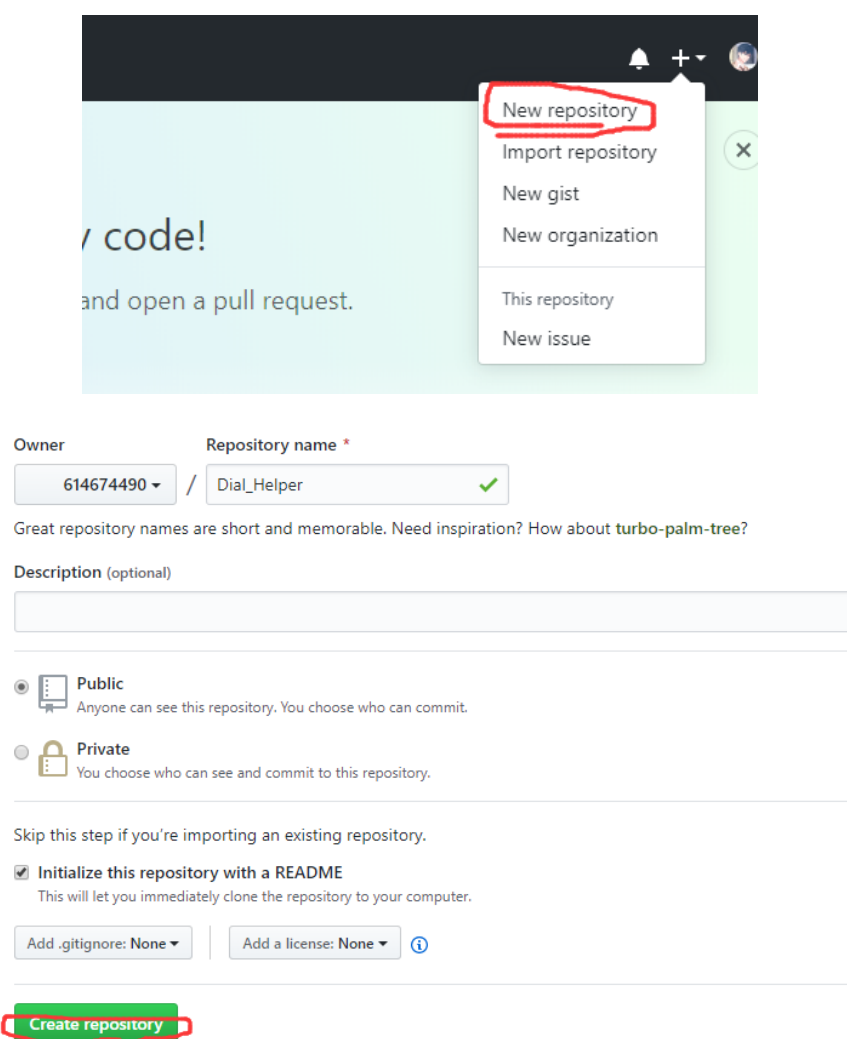
一、Git 全局设置

```
git config --global user.name "用户名"
```

```
git config --global user.email "邮箱"
```

二、新建 github 仓库 并用 vs code+git 拉取、推送、同步、克隆

1. GitHub 新建仓库



Owner: 614674490 / Repository name: Dial_Helper ✓

Great repository names are short and memorable. Need inspiration? How about **turbo-palm-tree**?

Description (optional)

☒ **Public**
Anyone can see this repository. You choose who can commit.

☐ **Private**
You choose who can see and commit to this repository.

Skip this step if you're importing an existing repository.

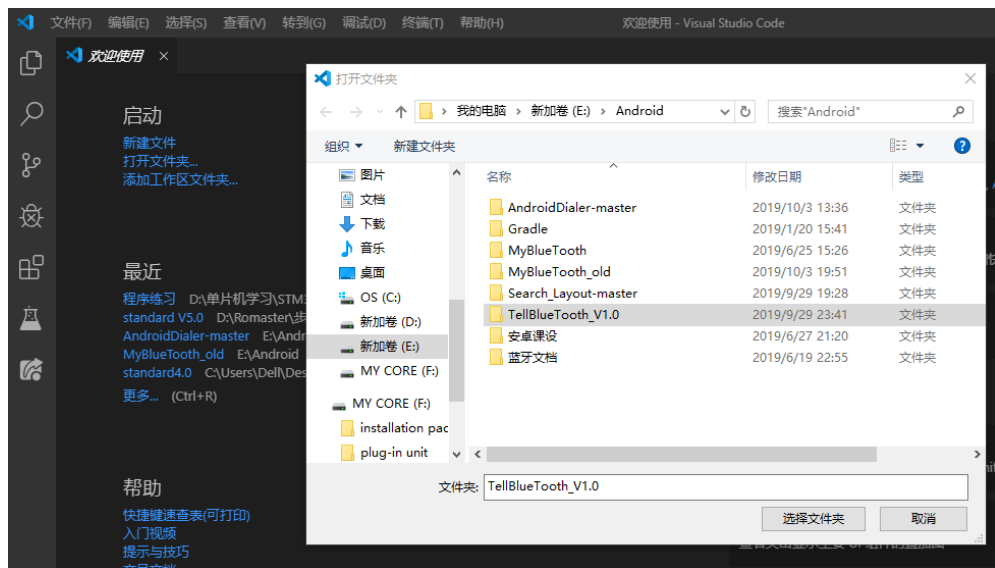
☒ **Initialize this repository with a README**
This will let you immediately clone the repository to your computer.

Add .gitignore: None ▾ | Add a license: None ▾ ⓘ

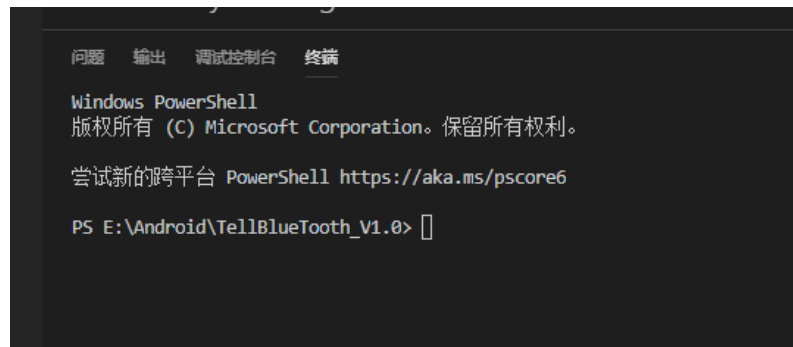
Create repository

2. 在 VS Code 上获取远程仓库，并进行关联

打开要上传工程所在的文件夹 → ctrl+K 然后 ctrl+O



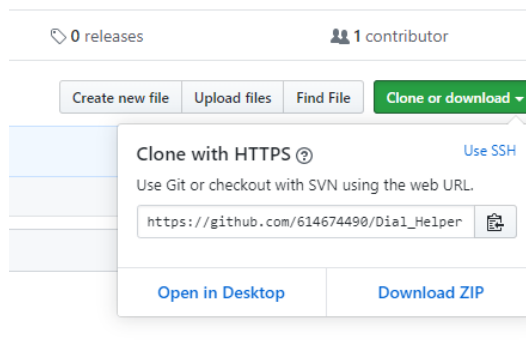
打开命令终端 (ctrl+`)



注意所有的 git 指令操作要在工程文件夹下进行

git init → 初始化仓库 就是建立仓库 生成.git 文件夹

(1) 通过 **Https** 的方法进行关联



获取图中的仓库连接 即获取远程存储库 后面的操作都是基于该存储库的

git remote add origin https://github.com/614674490/Dial_Helper.git

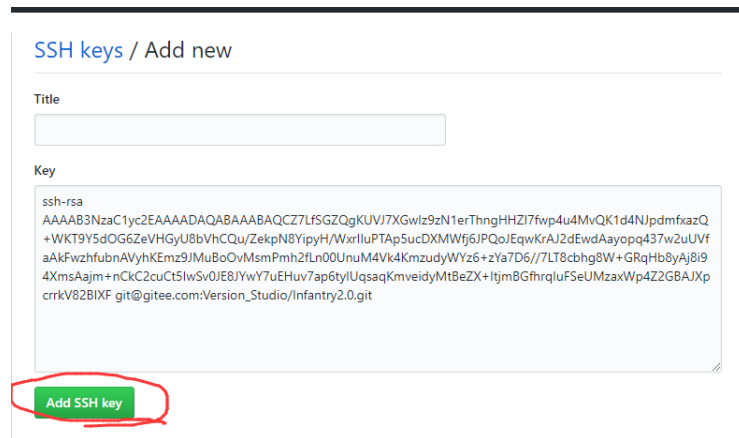
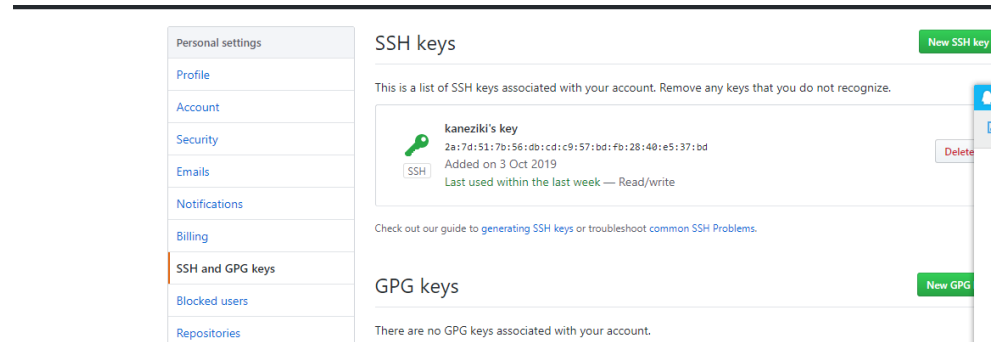
(2) 通过 **SSH** 的方式进行关联

通过密钥进行关联, 首先要生成密钥

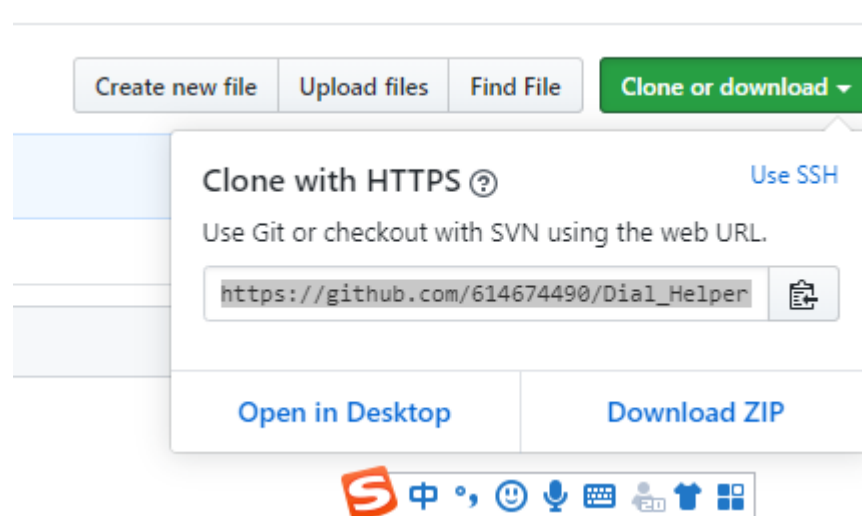
ssh-keygen -t rsa -C "614674490@qq.com" //此处为 github 的个人注册邮箱
生成的密钥默认保存在 C:\Users\Dell\.ssh

名称	修改日期	类型	大小
id_rsa	2019/4/6 12:31	文件	2 KB
id_rsa.pub	2019/4/6 12:31	Microsoft Publis...	1 KB
known_hosts	2019/4/5 10:57	文件	2 KB

打开 id_rsa.pub，复制里面的内容，然后粘贴到 github 的生成密钥的页面中
ssh 的添加在用户的设置页面 头像→Setting→SSH/GPG Keys



由于我之前已经生成的密钥，在此处不再生成密钥
然后在 VS Code 的命令终端输入



`git remote add origin git@github.com:614674490/Dial_Helper.git`

注意：用 Https 的办法的缺点是每次 push 的时候，都需要输入用户名和密码，如果不想输

入，可以通过命令 `git config --global credential.helper store` 记住密码
而用 SSH 的优点是不用输入用户名和密码，因为密钥是用来识别每一个 github 用户的。

（3） 直接修改 config 文件

初始化仓库后，.git 文件中有一个 config 文件



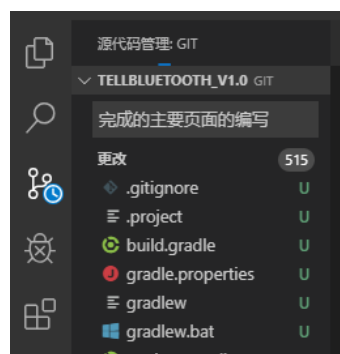
然后添加如下内容

```
[remote "origin"]
    url = https://github.com/614674490/Dial_Helper.git
    fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
    remote = origin
    merge = refs/heads/master

[core]
    repositoryformatversion = 0
    filemode = false
    bare = false
    logallrefupdates = true
    symlinks = false
    ignorecase = true
[remote "origin"]
    url = https://github.com/614674490/Dial_Helper.git
    fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
    remote = origin
    merge = refs/heads/master
```

3. Pull push

在上传代码前，先要点击 ✓（commit）并输入版本说明



如果是第一次上传该工程，需要 pull，将远程存储库拉取到本地，然后 push
在 git pull 可能会出现 fatal: refusing to merge unrelated histories(拒绝合并无关历史)
用该命令解决：`git pull origin master --allow-unrelated-histories`
Push 完成后出现以下页面：

对于刚创建的仓库，没有任何内容，可通过以下命令进行操作

```
echo "# STM32F1_Example" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/614674490/STM32F1_Example.git
git push -u origin master
```

Branch: master ▼	New pull request	Create new file	L
614674490 Merge branch 'master' of github.com:614674490/Dial_Helper			
📁 .gradle	完成主要页面的编写		
📁 .idea	完成主要页面的编写		
📁 .settings	完成主要页面的编写		
📁 app	完成主要页面的编写		
📁 gradle/wrapper	完成主要页面的编写		
📄 .project	完成主要页面的编写		
📄 README.md	Initial commit		
📄 TellBlueTooth_V1.0.iml	完成主要页面的编写		
📄 build.gradle	完成主要页面的编写		
📄 gradle.properties	完成主要页面的编写		
📄 gradlew	完成主要页面的编写		
📄 gradlew.bat	完成主要页面的编写		
📄 local.properties	完成主要页面的编写		
📄 settings.gradle	完成主要页面的编写		
📄 tellAssistant_image.zip	完成主要页面的编写		

此时，基本的仓库关联和 pull push 完成，之后可以进行 sync 远程和本地仓库的同步

注意：如果在 GitHub 上创建仓库后没有勾选自动添加 README.MD，不在仓库中添加任何内容会出现以下页面

Quick setup — if you've done this kind of thing before

📄 Set up in Desktop

 or

HTTPS

SSH

git@github.com:614674490/test.git

Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#)

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

```
echo "# test" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin git@github.com:614674490/test.git
git push -u origin master
```

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

```
git remote add origin git@github.com:614674490/test.git
git push -u origin master
```

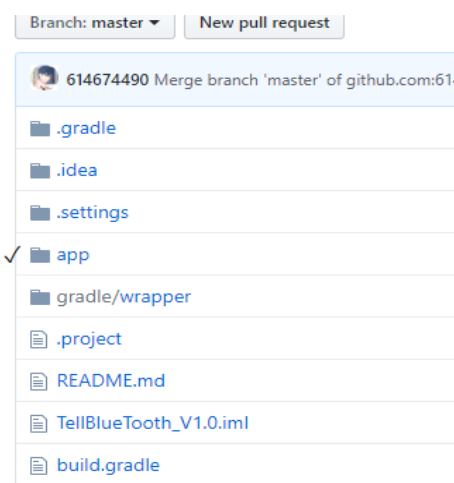
此时若在 VS Code 中进行 pull 和 push 可能会出现 not found any branch
因为此时没有任何分支可言,但是有仓库连接,所以最好勾选自动生成 README.MD,
或者在 VS Code 上更改好文件后,关联远程库,点击发布分支,将默认分支 master
发布到 GitHub 的远程库上,之后就可以随便 pull 和 push 啦

二、克隆远程仓库

复制仓库的 Https 或者 SSH,在 VS Code 上输入以下命令,即可自动克隆远程仓库
`git clone https/ssh` 由于网络原因,大型仓库可能时间比较长,也可以直接把代码以压缩包的形式下载到本地

三、GitHub 上单个文件的下载

在 Google 浏览器 chrome 中的应用商店下载扩展应用“GitZip for github”,然后在仓库中双击某个文件即可下载



四、搜索框快捷键

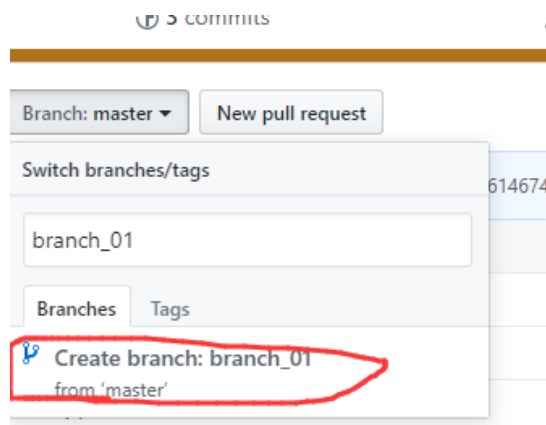
ctrl+shift+P

五、分支的添加与删除

1. 分支的作用

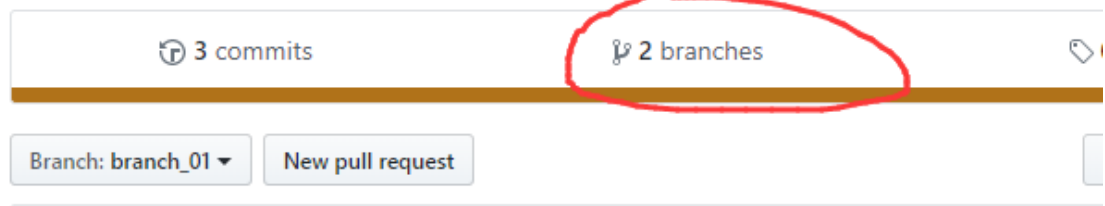
对一个项目同时进行多个版本的控制编写,并且最后可以进行分支合并

2. GitHub 上的添加与删除 (即远程仓库分支)

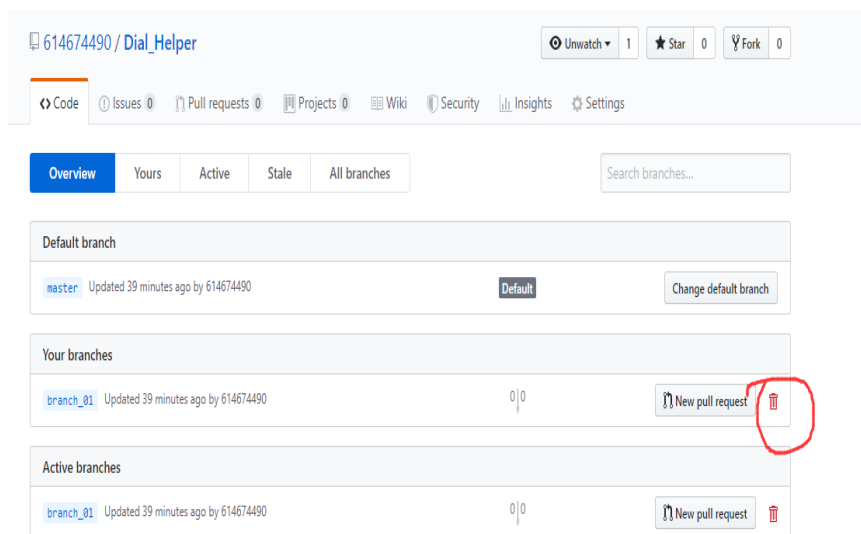


输入你要创建的分支名字，会自动弹出创建选项，进行创建即可
No description, website, or topics provided.

Manage topics

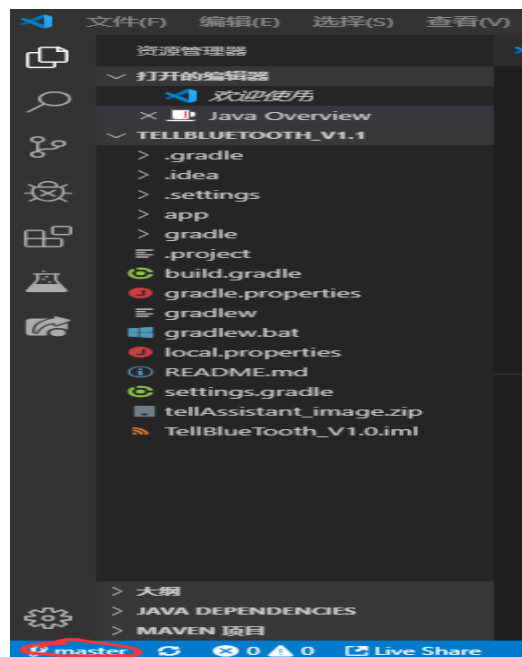


点击 **branches** 进入如下页面



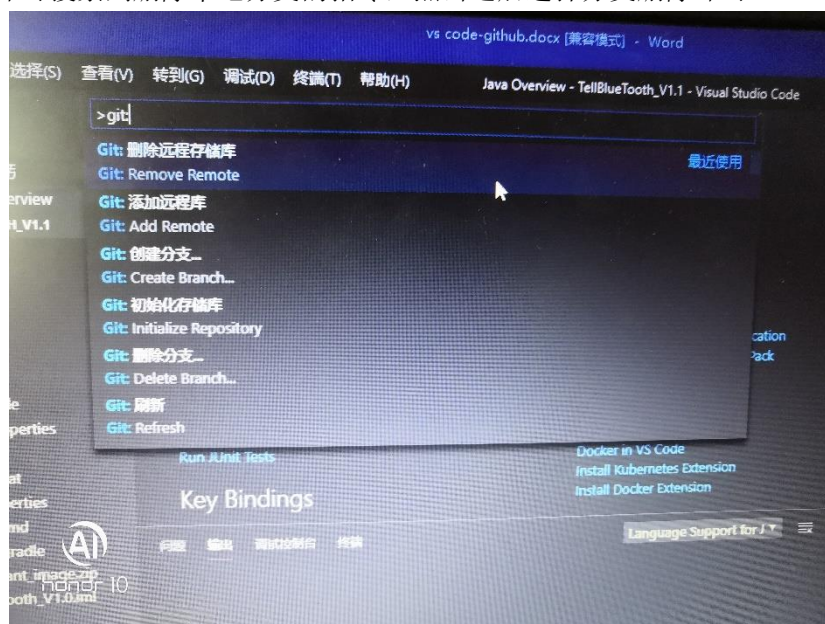
点击垃圾桶即可删除

4. VS Code 上的添加与删除（即本地仓库分支）



点击该处，便会在搜索框中出现提示，点击创建分支即可，也可通过该方式更改当

前的工程所在分支，两个分支之间不会产生影响，通过快捷键 **ctrl+shift+P** 打开搜索框，输入 **git:**，即可搜索到删除本地分支的指令，点击之后选择分支删除即可



创建一个分支后再 **push** 工程，如果远程仓库没有该分支，会提示发布该分支，将其同步到远程仓库，然后再 **push**，如果本地仓库想要获取远程仓库的建立的分支，可以通过 **sync** 同步功能进行获取

六、工作区、版本库和暂存区

<https://brianway.github.io/2016/08/07/git-basic-git-commands/>

- **工作区**：就是你在电脑里能看到的目录，比如我的 **learngit** 文件夹就是一个工作区。
- **版本库**：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。
- **暂存区**：**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**。

我们把文件往 **Git** 版本库里添加的时候，是分两步执行的：

1. 第一步是用 **git add** 把文件添加进去，实际上就是把文件修改添加到暂存区；
2. 第二步是用 **git commit** 提交更改，实际上就是把暂存区的所有内容提交到当前分支。

因为我们创建 **Git** 版本库时，**Git** 自动为我们创建了一个唯一的 **master** 分支，所以，现在 **git commit** 就是往 **master** 分支上提交更改。

简单理解：需要提交的文件修改通通放到暂存区，然后，一次性提交暂存区的所有修改。

详细知识见[工作区和暂存区](#)和[Git 基础 - 记录每次更新到仓库](#)

本地库和远程库

新建仓库

- 建立远程库(为空, 不要加 README.md, 不然后面会 push 不上去)
- 本地新建文件夹
- `git init` 初始化仓库, 可以发现当前目录下多了一个.git 的目录, 这个目录是 Git 来跟踪管理版本库的。勿人为瞎改

- 远程库的名字就是 `origin`, 这是 Git 默认的叫法
- `git remote add origin git@github.com:michaelliao/learngit.git` 这个命令是在本地的 `learngit` 仓库下执行的。这两个地方的 *仓库名不需要相同*, 因为会通过在本地的仓库目录下执行这条命令 (命令中包含远程库的名字) 已经将两者建立了联系
- `git push -u origin master` 把本地库的所有内容推送到远程库上。把本地库的内容推送到远程, 用 `git push` 命令, 实际上是把当前分支 `master` 推送到远程。由于远程库是空的, 我们第一次推送 `master` 分支时, 加上了 *-u 参数(推送和关联)*, Git 不但会把本地的 `master` 分支内容推送到远程新的 `master` 分支, 还会把本地的 `master` 分支和远程的 `master` 分支关联起来, 在以后的推送或者拉取时就可以简化命令。
- `git push origin master` 每次本地提交后, 推送最新修改到远程库

从远程库克隆

假设 `github` 上面已经有一个远程库, 但是本地没有, 需要克隆到本地, 远程库的名字叫 `gitskills`

- `git clone git@github.com:michaelliao/gitskills.git` 克隆一个本地库, 则在当前文件夹下会多一个 `gitskills` 的文件夹。
- `cd gitskills` 进入克隆下来的本地库, 默认的名字是和 `github` 上的一样的

- `git push origin master` 推送分支，就是把该分支上的所有本地提交推送到远程库。推送时，要指定本地分支，这样，Git 就会把该分支推送到远程库对应的远程分支上

常用查看指令

- `git status` 查看仓库当前的状态
- `git diff 文件名` 查看对文件做什么修改
- `git diff 版本号 1 版本号 2 --stat` 查看两个版本的差异的文件列表，包括被修改行数和增删图。参数改为 `--name-status` 前面显示修改说明字母(A,M 等)，无行数
- `git log` 显示从最近到最远的提交日志
- `git log --pretty=oneline` 简化日志输出的显示信息，`commit id` 很长,详细显示见[这里](#)
- `git reflog` 记录你的每一次命令，最先显示的是这个命令执行之后的版本的版本号的前七位，这样就算你清屏了或者重启了，也能找到某个版本的版本号，就可以轻松回退到那个版本
- `git branch` 查看当前所在的分支。`git branch` 命令会列出所有分支，当前分支前面会标一个 `*` 号
- `git log --graph --pretty=oneline --abbrev-commit` 用带参数的 `git log` 可以看到分支的合并情况。用 `git log --graph` 命令可以看到分支合并图
- `git remote` 查看远程库的信息
- `git remote -v` 显示更为详细的信息

常用修改指令

- `git add readme.txt` 添加，但是不提交

- `git commit -m "提交描述"` 提交，只有 **add** 后提交才有效。“改文件->add 文件->再改->提交”，则第二次修改无效,不会被提交，只会成功提交第一次的修改。

撤销修改和版本回退

- `git checkout -- 文件名` 把没暂存(即没 **add**)的干掉，或者说，丢弃工作区，回到到暂存状态
- `git reset HEAD 文件名` 把暂存的状态取消，工作区内容不变，但状态变为“未暂存”。

简单来说，没有 **add** 过的修改，只需要 `git checkout -- 文件名` 即可撤销；**add** 过的修改，先 `git reset HEAD 文件名` 变成没 **add** 过的修改，再 `git checkout -- 文件名` 撤销。操作示例可以看[这张图](#)

- `git reset --hard HEAD^` 会回退到上一个版本
- `git reset --hard 某版本号前几位` 通过命令行上的历史信息（假如你没清屏的话），找到某版本的版本号回到指定版本。不一定要全部版本号，就像这个命令的例子，只要前面的约 7、8 位这样就可以。

分支管理

创建和合并分支

- `git checkout -b dev` 创建一个新的分支：**dev**，并且会切换到 **dev** 分支。所以这条命令有两个作用。`git checkout` 命令加上 `-b` 参数表示创建并切换，相当于以下两条命令：`git branch dev` 和 `git checkout dev`
- `git branch dev`，新建分支是新建指针,指向当前 **commit**
- `git checkout dev` 切换到 **dev** 分支

- `git checkout master` dev 分支的工作完成，我们就可以切换回 master 分支(此时在 dev 分支的修改在 master 上是看不到的)
- `git merge dev` 这是在 master 分支上执行的命令，作用是：把 dev 分支上的工作成果合并到 master 分支上
- `git branch -d dev` 删除已合并的分支。删除分支就是删除指针
- `git branch -D dev` Git 友情提醒，dev 分支还没有被合并，如果删除，将丢失掉修改，如果要强行删除，需要使用 `git branch -D dev` 命令
- `git rebase master` 变基。在当前分支(非 master)下执行该命令，则相当于把当前分支和 master 分支合并，和 merge 操作类似，但提交历史不同，rebase 操作的 log 更干净。具体可参考 [Git 分支 - 变基](#)

解决冲突

假设在 master 分支和 feature1 分支对同一文件做了修改

- `git merge feature1` 在 master 分支上执行该命令，与 feature1 分支合并。这种情况下，Git 无法执行“快速合并”，只能试图把各自的修改合并起来，但这种合并就可能会有冲突，果然冲突了！Git 告诉我们，`readme.txt` 文件存在冲突，必须手动解决冲突后再提交。`git status` 也可以告诉我们冲突的文件

合并分支时，如果可能，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。如果要强制禁用 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

- `git merge --no-ff -m "merge with no-ff" dev` 准备合并 dev 分支，注意 `--no-ff` 参数表示禁用 Fast forward，因为本次合并要创建一个新的 commit，所以加上 -m 参数，把 commit 描述写进去

bug 分支

Git 还提供了一个 **stash** 功能，可以把当前工作现场“储藏”起来，等以后恢复现场后继续工作

- `git stash` 保存工作现场
- `git stash list` 查看工作现场
- `git stash apply` 恢复工作现场，但是恢复后，**stash** 内容并不删除，有多个工作现场时可以通过 `git stash apply stash@{0}` 恢复特定的现场
- `git stash drop` 删除 **stash** 的内容
- `git stash pop` 恢复的同时也把 **stas** 内容删除了

远程分支

这部分只介绍常用的几个操作

- `git fetch origin` 这个命令查找“**origin**”是哪一个服务器，从中抓取本地没有的数据，并且更新本地数据库，移动 `origin/master` 指针指向新的、更新后的位置
- `git push (remote) (branch)` 推送本地的分支来更新远程仓库上的 同名分支。如前文提到的 `git push origin master` 就是将本地 **master** 分支推送到远程 **master** 分支；复杂一点的，`git push origin serverfix:awesomebranch` 将本地的 **serverfix** 分支推送到远程仓库上的 **awesomebranch** 分支
- `git push origin --delete serverfix` 或者 `git push origin :remotebranch`，删除远程的 **serverfix** 分支
- `git pull` 在大多数情况下它的含义是一个 `git fetch` 紧接着一个 `git merge` 命令。具体可参考 [Git 远程操作详解](#)和 [Documentation git-pull](#)