**惠普软件班实训课件**

**主讲：汤燕**

* 大四阶段建议和安排（重要）
* Git工具介绍 （重点）
* 实训项目演示（体验）

# 大四建议和安排

时间过得很快，一转眼大家就进入大四了，本科阶段已接近尾声。

大四应该怎么度过？老师应该怎么做？同学们应该怎么做？如何配合？下面谈谈我的想法。

毕业之前重要的事情：

考研：（统计）

毕业证：（必须）

学位证：（必须）

英语：校线362 （必须）

计算机二级：（必须）

驾照：非必须

毕业设计和论文：必须的。

就业：持续过程。

## 毕业前需要注意的

1. 考研的同学抓紧时间准备考试，但是上课必须到场，实训作品和实训相关文档必须交齐。
2. 先把学分修够，挂科的要补起来，确保能把毕业证和学位证拿到手，要不然要吃一辈子的亏。
3. 确实觉得一点希望都没有的了，彻底放弃的，你应该学好一门技术，哪怕是把微信小程序掌握好。
4. 驾照、英语、计算机等级证书。
5. 完成实训、完成毕业设计和毕业论文、学习其他学科。

## 就业方向

1. 考研。（成绩比较好的，学霸、考霸）。把学历提高点，一方面可以弥补你们高考阶段的失误，曾经因为各种原因，总之是失误了，这次可以证明一下自己。（根据自身实际情况出发，量力而行）。
2. 考公务员/银行（工行、农行、建行、邮政储蓄、中国银行）/事业单位（比如社保局）/村官等。（根据自身实际情况出发，量力而行）。
3. 前端岗位。（大多数）。（培训机构的学生包装过，一般要7000左右，要得高，老板要求也高。 建议大家要6500，讨价还价后还有5000，他们学的知识我们都学过，我们还有价格优势）。
4. 小程序开发岗位。（大多数，最好找工作，同前端岗位，实际是前端的一个细分领域）
5. 测试岗位（喜欢测试的可以找测试老师给大家规划一下，这种岗位比较适合女生，技术要求不是太高）。
6. Java开发工程师。（后端是一个综合了很多技术的岗位，大家目前后端的开发能力大多数同学还处于初级阶段，个别后端做得好的同学适合这个岗位）
7. 软件销售经理、业务经理、销售代表。（这是我要重点讲的岗位，因为班上有一部分同学大学四年玩过来的）。

简单的说就是卖软件。这部分同学也应该做一些准备。

如果你觉得大学四年学的技术不够用，或者是没有学好，或者是学好又忘记了，或者是不想编程怕掉头发，吃不了编程的苦，或者是因为玩游戏耽误了太多的时间，或者是经过四年实践试错后发现自己都没有做程序的天赋，总之不喜欢开发软件，大学四年学的这些技术就白学了吗？不是的，你可以去卖软件。你对软件这个行业也不是一点不了解。

这部分同学需要掌握一些销售技巧。销售做好了，这部分人后面很可能比做开发的混得好，但是前期开发客户比较辛苦。

销售一般以提层为主，底薪低，前期不太稳定，但是做好了之后，以后转行销售其他的产品也是触类旁通。

销售这个行当，萌芽于古代的商业社会，发展于20世纪初，泛滥于今；不同时期的销售，都在影响人们的消费行为。

第一、熟悉自己的产品。 OA\ERP（SAP）\财务软件（用友\金蝶）\供应链系统\考试系统\医疗软件等。 （卖房、卖车、卖软件、卖解决方案）

第二、要学会倾听，少说多听，多听客户的需求，了解客户的痛点，了解客户的兴奋点，了解客户需要解决的问题。

第三、想尽各种办法促成交易。

第四、做好售后的维护工作。

销售也不是简单这几点可以总结完的，主要还是要靠实践积累经验。

## 各种准备

无论你是哪个方向和岗位，你都应该准备以下的资料：

（停下手里的所有的游戏，暂时忍一忍）

第一、准备简历3份。（前端简历-中 / 小程序开发工程师 / 测试简历 / 销售简历 / java开发工程师 / 其他简历 根据自己情况定 ）。

第二、面试宝典。上面就有面试题。（检查）

第三、准备项目作品，精细化，把高度提高，要求提高。

第四、同学之间进行模拟面试。（惠普软件班面试资源共享系统）。

第五、参加各种校园招聘（到其他学校、了解招聘会信息）。举例。

第六、早点准备毕业设计题目和论文。

# Git版本管理器

初学者很难找到一个由浅入深，学完后能立刻上手的Git教程。

## 什么是git

Git是目前世界上最先进的分布式版本控制系统（没有之一）。  
Git有非常高的逼格，这么好的东东如何来学习，网上很多git教程基本上都是支离破碎地介绍Git的几个命令，要么难的发紫，要么简单得一笔带过，初学者学完很难立刻上手。这次给大家准备的教程具有非常强的可操作性，一边学一边练，80分钟就可以轻松学会，可以免费托管自己的代码，甚至可以利用git提供的免费空间和域名搭建自己的静态网站，而且还很稳定哦，绝对的高大上，而且找工作的时候是否会用git也是一个影响你薪资等级的重要因素。

Git虽然功能强大，但是命令繁多，不过常用的就那么十来个，掌握好这十几个常用命令，你已经可以得心应手地使用Git了。



Git is a [free and open source](https://git-scm.com/about/free-and-open-source) distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

Git is [easy to learn](https://git-scm.com/doc) and has a [tiny footprint with lightning fast performance](https://git-scm.com/about/small-and-fast). It outclasses SCM tools like Subversion, CVS, Perforce, and ClearCase with features like [cheap local branching](https://git-scm.com/about/branching-and-merging), convenient [staging areas](https://git-scm.com/about/staging-area), and [multiple workflows](https://git-scm.com/about/distributed).



Linus在1991年创建了开源的Linux操作系统，经过不断发展，已经成为最大的服务器系统软件了。

Linus虽然创建了Linux，但Linux的壮大是靠全世界热心的志愿者参与的，这么多人在世界各地为Linux编写代码，那Linux的代码是如何管理的呢？如何升级程序的呢？

在2002年以前，世界各地的志愿者把源代码文件通过diff的方式发给Linus，然后由Linus本人通过手工方式合并代码！

为什么Linus不把Linux代码放到版本控制系统里呢？不是有CVS、SVN这些免费的版本控制系统吗？因为Linus坚定地反对CVS和SVN，这些集中式的版本控制系统不但速度慢，而且必须联网才能使用。有一些商用的版本控制系统，虽然比CVS、SVN好用，但那是付费的，和Linux的开源精神不符。

到了2002年，Linux系统已经发展了十年了，代码库之大让Linus很难继续通过手工方式管理了，社区的弟兄们也对这种方式表达了强烈不满，于是Linus选择了一个商业的版本控制系统BitKeeper，BitKeeper的东家BitMover公司出于人道主义精神，授权Linux社区免费使用这个版本控制系统。

Linux社区牛人越来越多，2005年，开发Samba的Andrew试图破解BitKeeper的协议（这么干的其实也不只他一个），被BitMover公司发现了（监控工作做得不错！），于是BitMover公司怒了，要收回Linux社区的免费使用权。

Linus本来可以向BitMover公司道个歉，保证以后严格管教弟兄些。但是Linus没这样做，他自己花了两周的时间用C语言写了一个分布式版本控制系统，这就是Git！

Git迅速成为最流行的分布式版本控制系统，尤其是2008年，GitHub网站上线了，它为开源项目免费提供Git存储，无数开源项目开始迁移至GitHub，包括jQuery，PHP，Ruby等等。

可以看出来，很多人都是被逼的。所以你们也需要有人逼你们一把。曾经就有不少应届毕业生单枪匹马闯北上广深杭，随着内地发展加快，也有不少沿海城市的大学生来蓉漂或者渝漂的。实际上到一个陌生的城市从头开始也是一段美好的经历。

最早Git是在Linux上开发的，很长一段时间内，Git也只能在Linux和Unix系统上跑。不过，慢慢地有人把它移植到了Windows上。现在，Git可以在Linux、Unix、Mac和Windows这几大平台上正常运行了。

## 集中式vs分布式



集中式版本控制系统，版本库是集中存放在中央服务器的，而干活的时候，用的都是自己的电脑，所以要先从中央服务器取得最新的版本，然后开始干活，干完活了，再把自己的活推送给中央服务器。（签入和签出）

集中式版本控制系统最大的毛病就是必须联网才能工作，如果在局域网内还好，带宽够大，速度够快，可如果在互联网上，遇到网速慢的话，可能提交一个10M的文件就需要1分钟，这样是没法忍受的。

分布式版本控制系统根本没有“中央服务器”，每个人的电脑上都是一个完整的版本库，这样，你工作的时候，就不需要联网了，因为版本库就在你自己的电脑上。既然每个人电脑上都有一个完整的版本库，那多个人如何协作呢？比方说你在自己电脑上改了文件A，你的同事也在他的电脑上改了文件A，这时，你们俩之间只需把各自的修改推送给对方，就可以互相看到对方的修改了。

## 常见的版本管理工具

CVS、SVN、VSS、TFS都是集中式的版本控制系统。Git是分布式版本控制工具。

和集中式版本控制系统相比，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，某一个人的电脑坏掉了不要紧，随便从其他人那里复制一个就可以了。而集中式版本控制系统的中央服务器要是出了问题，所有人都没法干活了。

在实际使用分布式版本控制系统的时候，其实很少在两人之间的电脑上推送版本库的修改，因为可能你们俩不在一个局域网内，两台电脑互相访问不了，也可能今天你的同事病了，他的电脑压根没有开机。因此，分布式版本控制系统通常也有一台充当“中央服务器”的电脑，但这个服务器的作用仅仅是用来方便“交换”大家的修改，没有它大家也一样干活，只是交换修改不方便而已。



Git的优势：

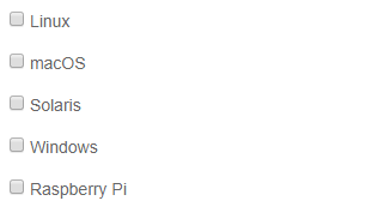
1、不必联网这么简单。（在本地有一个完整的仓库，提交一次就有一次记录，在本地会记录你的修改，可以撤销，可以反悔）。如果是vss,svn,tfs等，只有联网提交到中央服务器才会有修改记录，必须联网。

2、分支管理。

分布式版本控制系统 Git、BitKeeper、 Mercurial、Bazaar等。

这些分布式版本控制系统各有特点，但最快、最简单也最流行的依然是Git！

## Git可以运行在哪些操心系统上



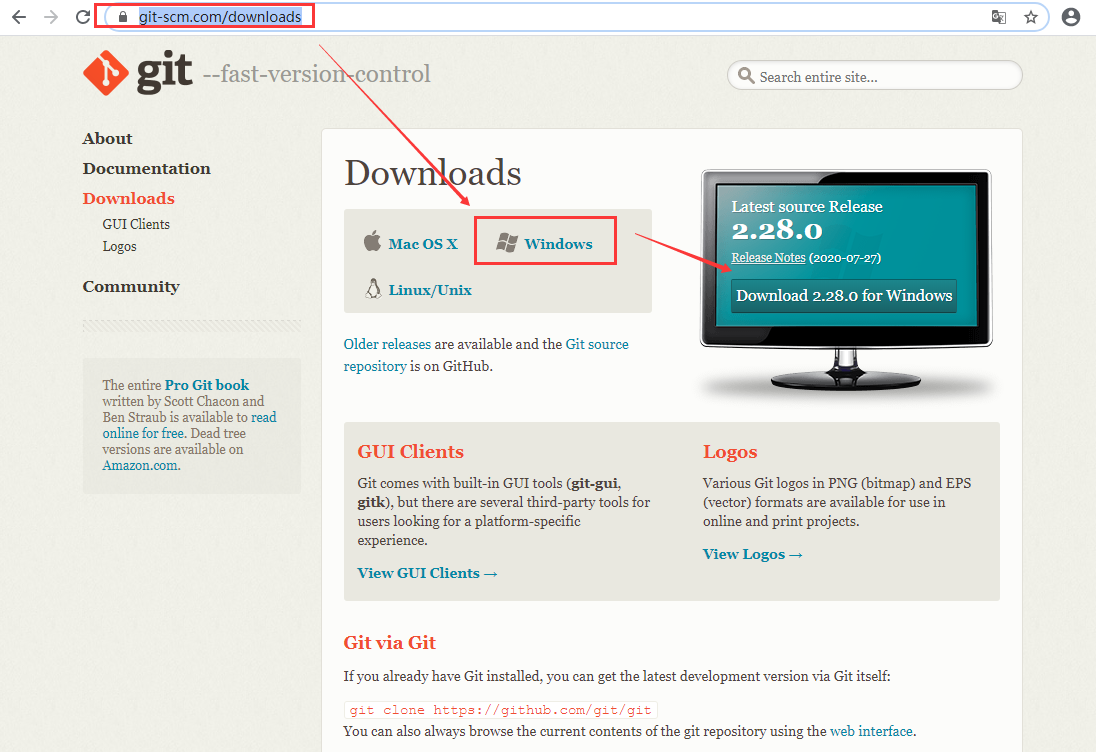
如果你使用Windows系统，为了避免遇到各种莫名其妙的问题，请确保目录名（包括父目录）不包含中文。

## Git官网地址

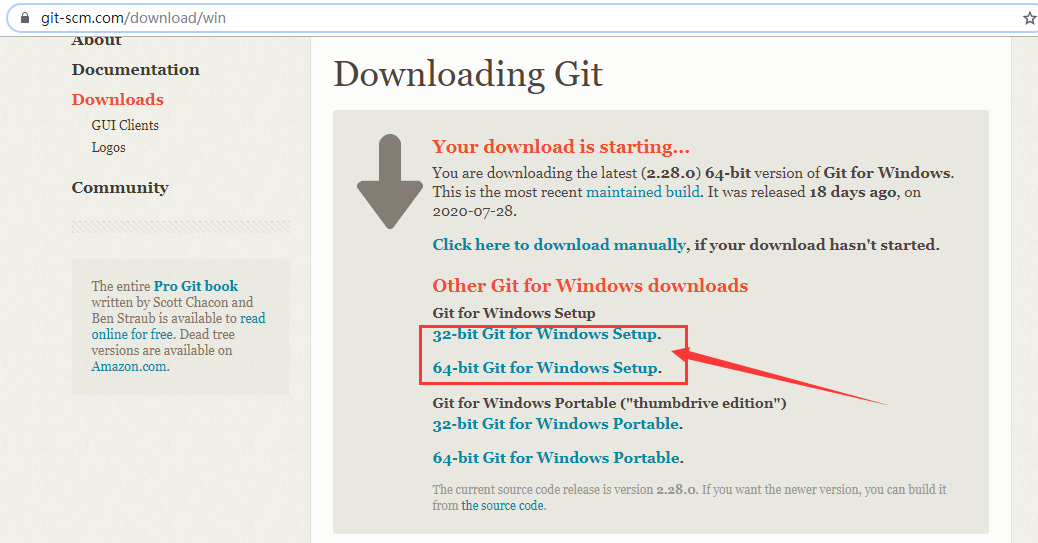
<https://git-scm.com/>

## Git下载地址

<https://git-scm.com/downloads>

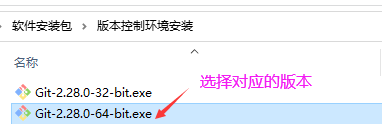


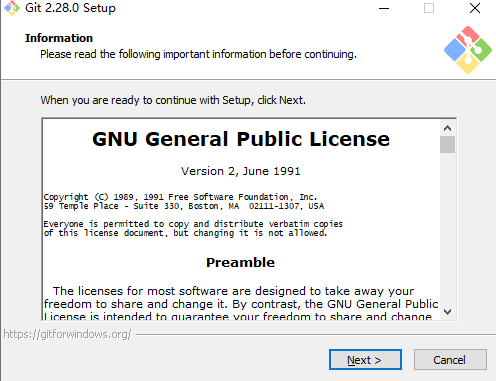
<https://git-scm.com/download/win>

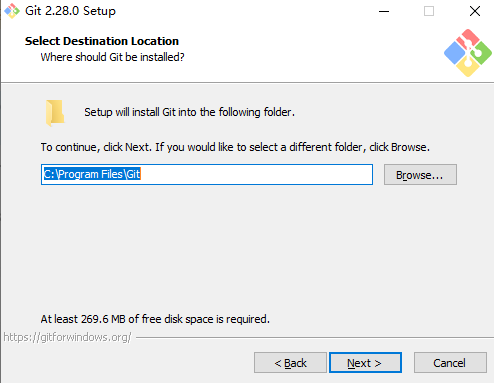


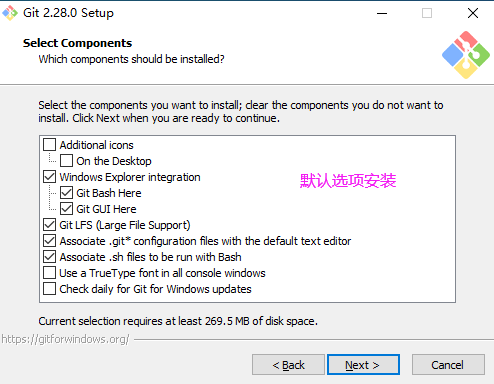
## Git安装教程

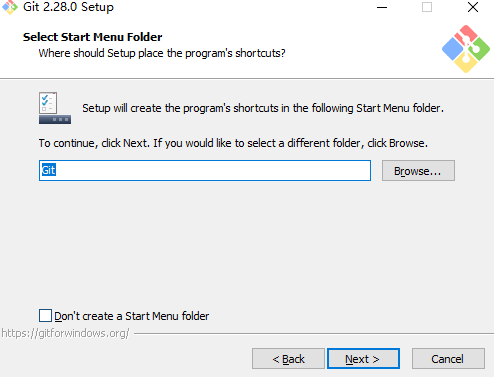
根据你自己的电脑选择对应的版本双击安装。

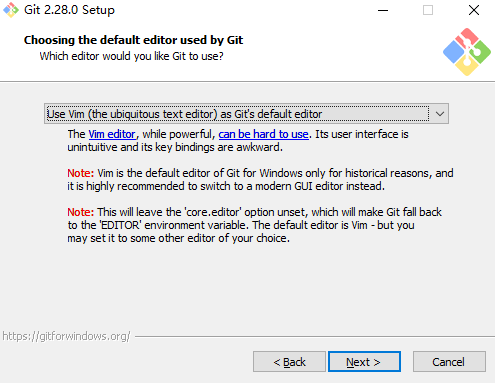












…

一直默认安装

安装完成后，在开始菜单里找到“Git”->“Git Bash”，蹦出一个类似命令行窗口的东西，就说明Git安装成功！



安装完成后，还需要最后一步设置，在命令行输入：

$ git config --global user.name "tangyan"

$ git config --global user.email "1416759661@qq.com"

因为Git是分布式版本控制系统，所以，每个机器都必须自报家门：你的名字和Email地址。你也许会担心，如果有人故意冒充别人怎么办？这个不必担心，首先我们相信大家都是善良无知的群众，其次，真的有冒充的也是有办法可查的。

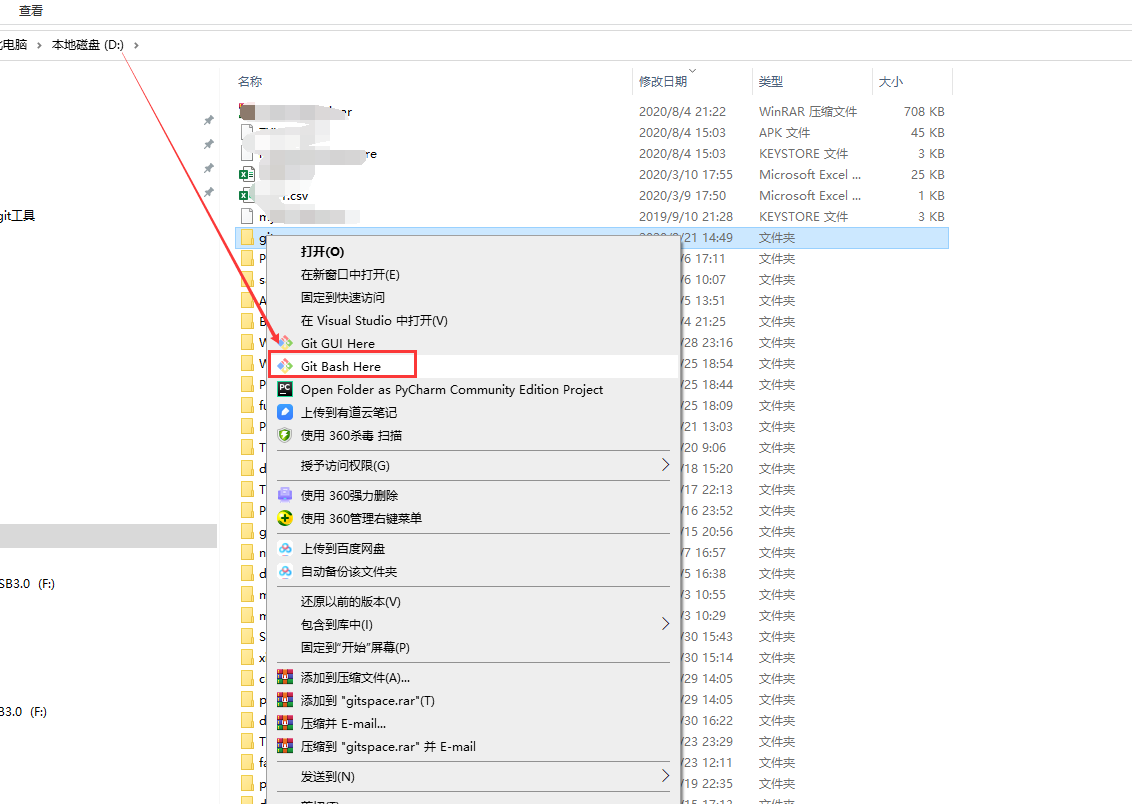
注意git config命令的--global参数，用了这个参数，表示你这台机器上所有的Git仓库都会使用这个配置，当然也可以对某个仓库指定不同的用户名和Email地址。

## 创建版本库

什么是版本库呢？版本库又名仓库，英文名**repository**，你可以简单理解成一个目录，这个目录里面的所有文件都可以被Git管理起来，每个文件的修改、删除，Git都能跟踪，以便任何时刻都可以追踪历史，或者在将来某个时刻可以“还原”。

所以，创建一个版本库非常简单，首先，选择一个合适的地方，创建一个空目录：

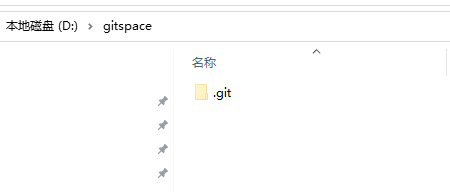
在d盘的根目录建立一个文件夹gitspace,在文件夹上面点击右键，点击Git Bash Here。



通过git init命令把这个目录变成Git可以管理的仓库.

细心的读者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，没事千万不要手动修改这个目录里面的文件，不然改乱了，就把Git仓库给破坏了。

如果你没有看到.git目录，那是因为这个目录默认是隐藏的



也不一定必须在空目录下创建Git仓库，选择一个已经有东西的目录也是可以的。

不建议你使用自己正在开发的公司项目来学习Git，否则造成的一切后果概不负责。

## 把文件添加到版本库

1）所有的版本控制系统，只能跟踪文本文件的改动，比如TXT文件，网页，所有的程序代码等等，Git也不例外。

2）版本控制系统可以告诉你每次的改动，比如在第5行加了一个单词“Linux”，在第8行删了一个单词“Windows”。

3）图片、视频这些二进制文件，虽然也能由版本控制系统管理，但没法跟踪文件的变化，只能把二进制文件每次改动串起来，也就是只知道图片从100KB改成了120KB，但到底改了啥，版本控制系统不知道，也没法知道。

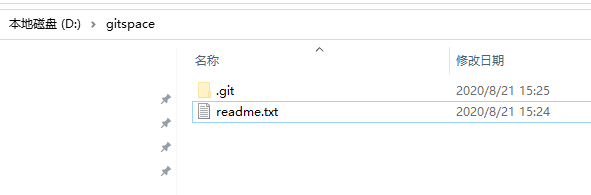
4）Word格式是二进制格式，因此，版本控制系统是没法跟踪Word文件的改动的。

5）如果要真正使用版本控制系统，就要以纯文本方式编写文件。

6）强烈建议使用标准的UTF-8编码，所有语言使用同一种编码，既没有冲突，又被所有平台所支持。

注意：

**记事本**保存UTF-8编码的文件的时候会在每个文件开头添加了0xefbbbf（十六进制）的字符，有时候会出现一些奇怪的问题。推荐使用Notepad++代替记事本，不但功能强大，而且免费！记得把Notepad++的默认编码设置为UTF-8 without BOM即可。

在D盘gitspace文件夹下面建立一个readme.txt文本文件。

使用命令git add readme.txt把readme.txt文件添加到Git仓库（此处的添加仅仅是建立一个添加文件的计划任务，这个任务只有在提交后才生效）。

git add readme.txt

如果要把整个文件夹下面的文件都添加到仓库，就用

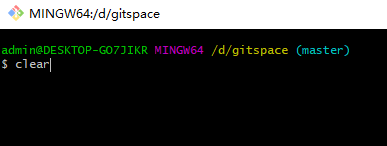
git add . (打一个点表示全部添加)

用命令git commit告诉Git，把文件提交到仓库

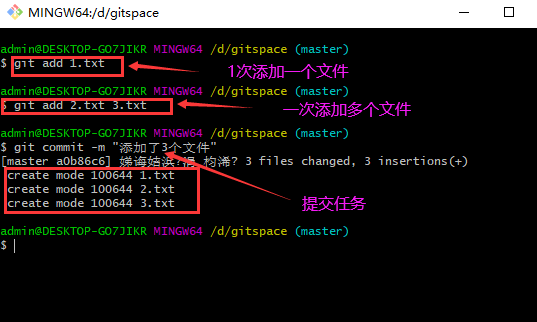


git commit命令，-m后面输入的是本次提交的说明，可以输入任意内容，当然最好是有意义的，这样你就能从历史记录里方便地找到改动记录。

命令写多了，clear命令可以清屏。



你可以多次add文件，每次add一个文件，也可以一次add多个文件。



总结一下两点内容：

初始化一个Git仓库，使用git init命令。

添加文件到Git仓库，分两步：

1. 使用命令git add <file>，注意，可反复多次使用，添加多个文件；
2. 使用命令git commit -m <message>，完成。

## 修改readme.txt文件

我们打开readme.txt文件

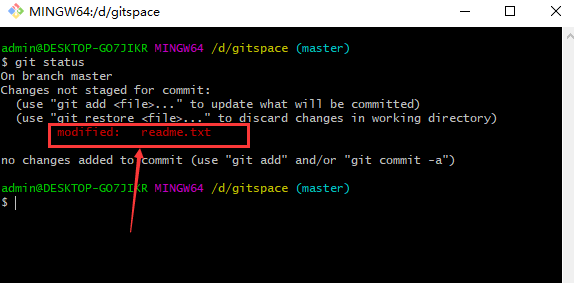
把里面的文字修改为：

Git **is** a distributed version control system.

Git **is** free software.

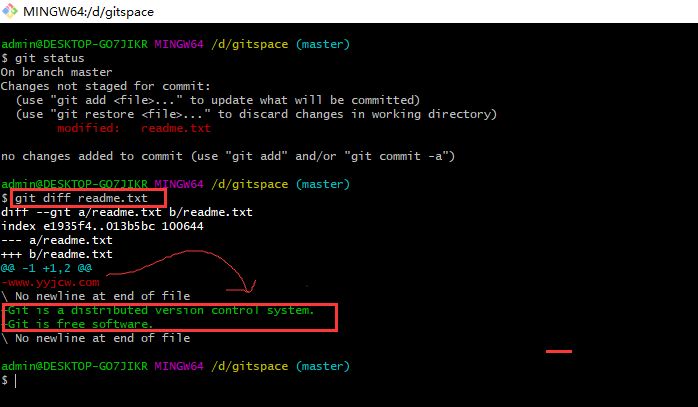
git status可以查看仓库当前的状态。

git status



git status命令可以让我们时刻掌握仓库当前的状态，上面的命令输出告诉我们，readme.txt被修改过了，但还没有准备提交的修改。

虽然Git告诉我们readme.txt被修改了，但如果能看看具体修改了什么内容，自然是很好的。比如你休假两周从国外回来，第一天上班时，已经记不清上次怎么修改的readme.txt，所以，需要用git diff这个命令看看：

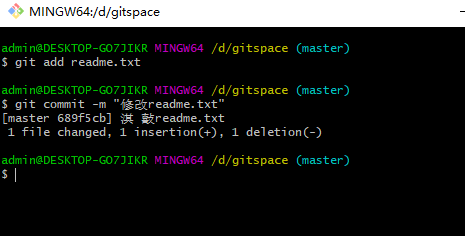


git diff顾名思义就是查看difference，

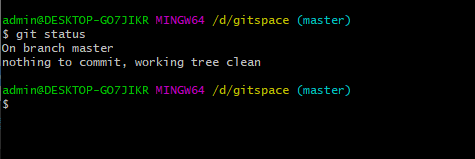
知道了对readme.txt作了什么修改后，再把它提交到仓库就放心多了，提交修改和提交新文件是一样的两步，第一步是git add：

第二步git commit之前，我们再运行git status看看当前仓库的状态

git **commit** -m "修改readme.txt"



提交成功。

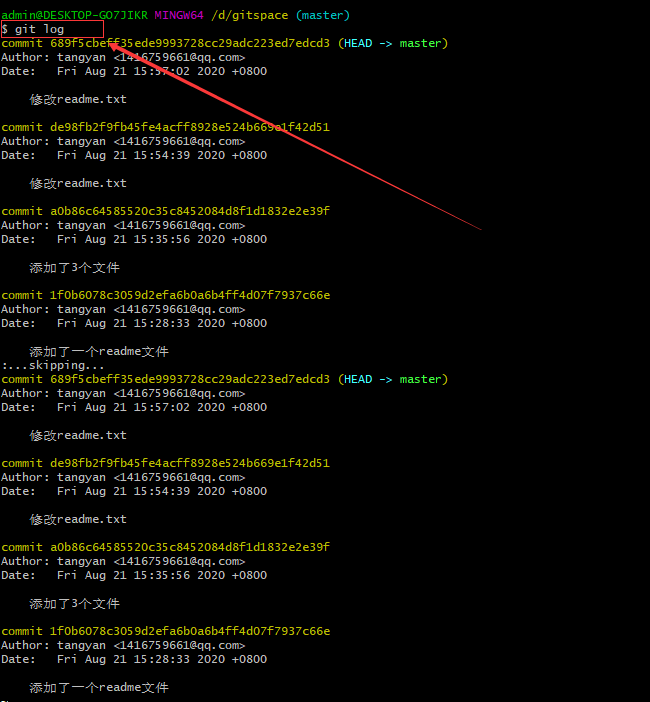


Git告诉我们当前没有需要提交的修改，而且，工作目录是干净（working tree clean）的。

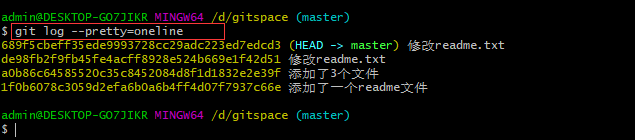
## 版本回退

不断对文件进行修改，然后不断提交修改到版本库里，就好比玩RPG游戏时，每通过一关就会自动把游戏状态存盘，如果某一关没过去，你还可以选择读取前一关的状态。有些时候，在打Boss之前，你会手动存盘，以便万一打Boss失败了，可以从最近的地方重新开始。Git也是一样，每当你觉得文件修改到一定程度的时候，就可以“保存一个快照”，这个快照在Git中被称为commit。一旦你把文件改乱了，或者误删了文件，还可以从最近的一个commit恢复，然后继续工作，而不是把几个月的工作成果全部丢失。

在实际工作中，我们会频繁的修改仓库里面的文件，我们脑子里怎么可能记得一个几千行的文件每次都改了什么内容。版本控制系统肯定有某个命令可以告诉我们历史记录，在Git中，我们用git log命令查看。



如果嫌输出信息太多，看得眼花缭乱的，可以试试加上--pretty=oneline参数：



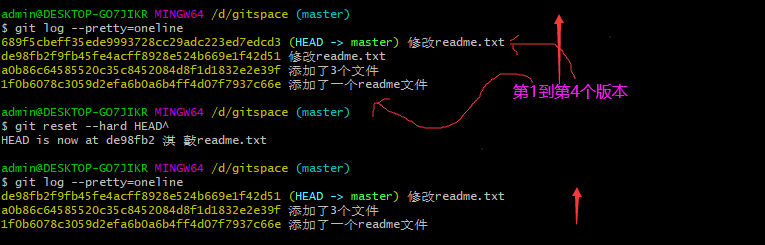
你看到的一大串类似689f5...的是commit id（版本号）

是一个SHA1计算出来的一个非常大的数字，用十六进制表示，而且你看到的commit id和我的肯定不一样，以你自己的为准。

为什么commit id需要用这么一大串数字表示呢？因为Git是分布式的版本控制系统，后面我们还要研究多人在同一个版本库里工作，如果大家都用1，2，3……作为版本号，那肯定就冲突了。

接下来我们退回到上一个版本。

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。



回到特定的版本。

$ git re**set** --hard 1f0b6



Git的版本回退速度非常快。

你回退到了某个版本，关掉了电脑，第二天早上就后悔了，想恢复到新版本怎么办？找不到新版本的commit id怎么办？

在Git中，总是有后悔药可以吃的。当你用$ git reset --hard HEAD^回退到add distributed版本时，再想恢复到append GPL，就必须找到append GPL的commit id。Git提供了一个命令git reflog用来记录你的每一次命令：

$ git reflog

总结一下：

HEAD指向的版本就是当前版本，因此，Git允许我们在版本的历史之间穿梭，使用命令git reset --hard commit\_id。

穿梭前，用git log可以查看提交历史，以便确定要回退到哪个版本。

要重返未来，用git reflog查看命令历史，以便确定要回到未来的哪个版本。

## 工作区和暂存区

D:\gitspace文件夹就是一个工作区

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。



我们把文件往Git版本库里添加的时候，是分两步执行的：

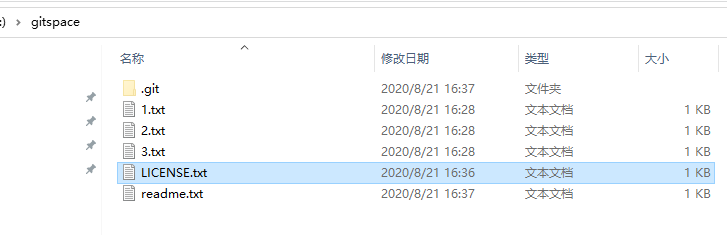
第一步是用git add把文件添加进去，实际上就是把文件修改添加到暂存区；

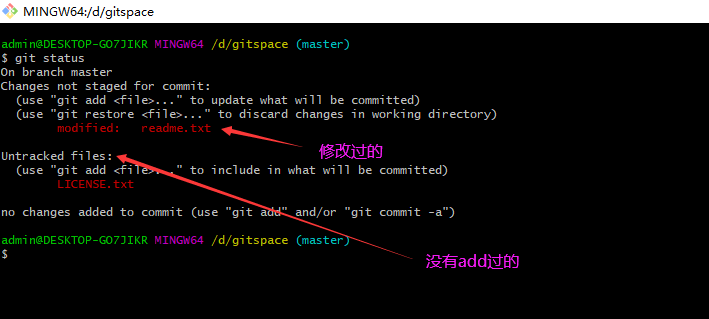
第二步是用git commit提交更改，实际上就是把暂存区的所有内容提交到当前分支。

因为我们创建Git版本库时，Git自动为我们创建了唯一一个master分支，所以，现在，git commit就是往master分支上提交更改。

你可以简单理解为，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。

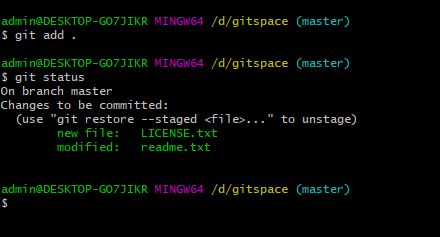
在工作区新增一个LICENSE文本文件（内容随便写）。





Git非常清楚地告诉我们，readme.txt被修改了，而LICENSE还从来没有被添加过，所以它的状态是Untracked。

现在，使用两次命令git add .，把readme.txt和LICENSE都添加后，用git status再查看一下：

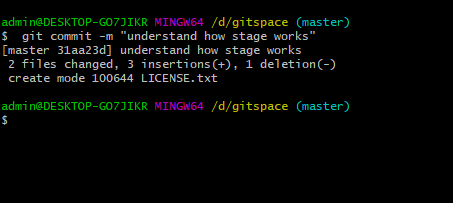


现在，暂存区的状态就变成这样了：



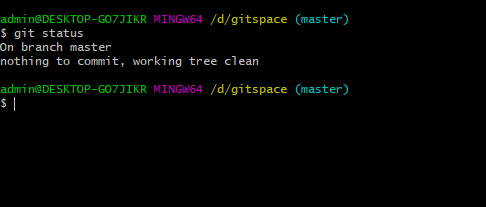
所以，git add命令实际上就是把要提交的所有修改放到暂存区（Stage），然后，执行git commit就可以一次性把暂存区的所有修改提交到分支。

$ git **commit** -m "understand how stage works"



一旦提交后，如果你又没有对工作区做任何修改，那么工作区就是“干净”的：

$ git status



现在版本库变成了这样，暂存区就没有任何内容了：



暂存区是Git非常重要的概念，弄明白了暂存区，就弄明白了Git的很多操作到底干了什么。

注意：

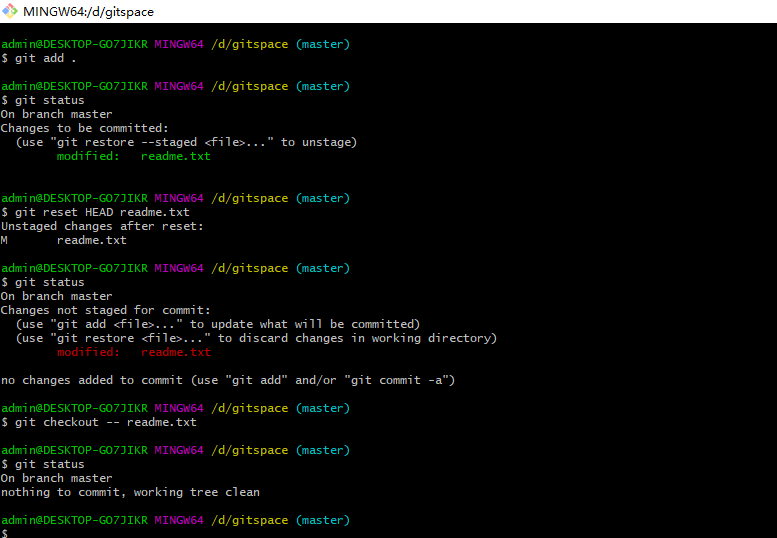
如果你对某个文件（内容为A）进行了修改（内容改为A+），然后把它提交到暂存区（内容为A+），然后又修改了一次这个文件为（A-），然后执行了git commit . 此时提交到master里面的是A+还是A-，答案是A+,因为你没有把A-提交到暂存区。

查看文件的命令：

cat readme.txt

git checkout -- file可以丢弃工作区的修改。

用命令git reset HEAD <file>可以把暂存区的修改撤销掉（unstage），重新放回工作区。



总结下：

场景1：当你改乱了工作区某个文件的内容，想直接丢弃工作区的修改时，用命令git checkout -- file。

场景2：当你不但改乱了工作区某个文件的内容，还添加到了暂存区时，想丢弃修改，分两步，第一步用命令git reset HEAD <file>，就回到了场景1，第二步按场景1操作。

场景3：已经提交了不合适的修改到版本库时，想要撤销本次提交，可以通过版本号退回，不过前提是没有推送到远程库。

## 删除文件



git rm <file>和git add<file>效果是一样的。

工作区和版本库的文件不一致的时候，有两种做法：

1）确实要从版本库中删除该文件，那就用命令git rm删掉，并且git commit。

2）使用 git checkout -- test.txt 命令把暂存区的覆盖到工作区。

# 远程仓库

## 远程仓库基本概念

Git是分布式版本控制系统，同一个Git仓库，可以分布到不同的机器上，一台电脑上也是可以克隆多个版本库。

企业中真实情况是这样的，找一台电脑充当服务器的角色，每天24小时开机，其他每个人都从这个“服务器”仓库克隆一份到自己的电脑上，并且各自把各自的提交推送到服务器仓库里，也从服务器仓库中拉取别人的提交。

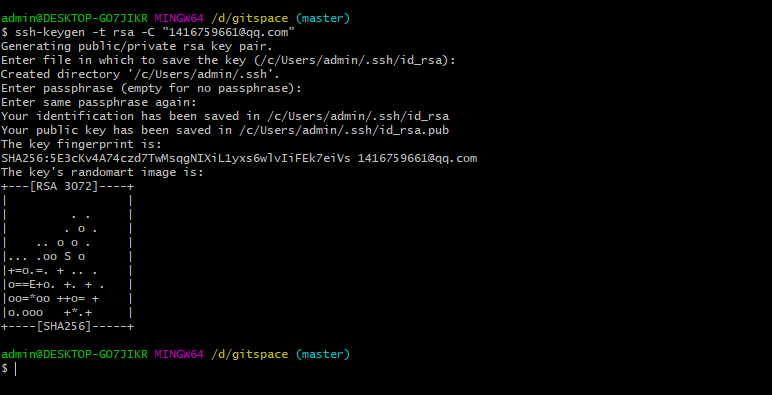
完全可以自己搭建一台运行Git的服务器，不过现阶段，为了学Git先搭个服务器绝对是小题大作。好在这个世界上有个叫GitHub的神奇的网站，从名字就可以看出，这个网站就是提供Git仓库托管服务的，所以，只要注册一个GitHub账号，就可以免费获得Git远程仓库。

## 创建SSH Key

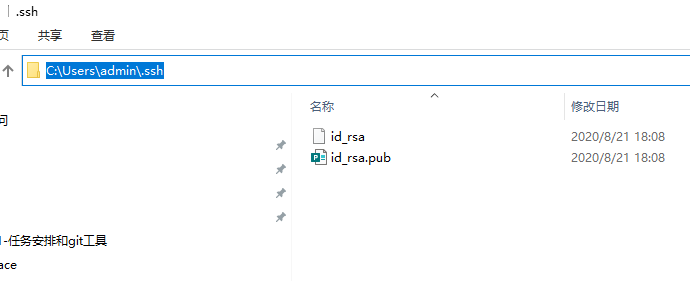
由于你的本地Git仓库和GitHub仓库之间的传输是通过SSH加密的，所以，需要一点设置：

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

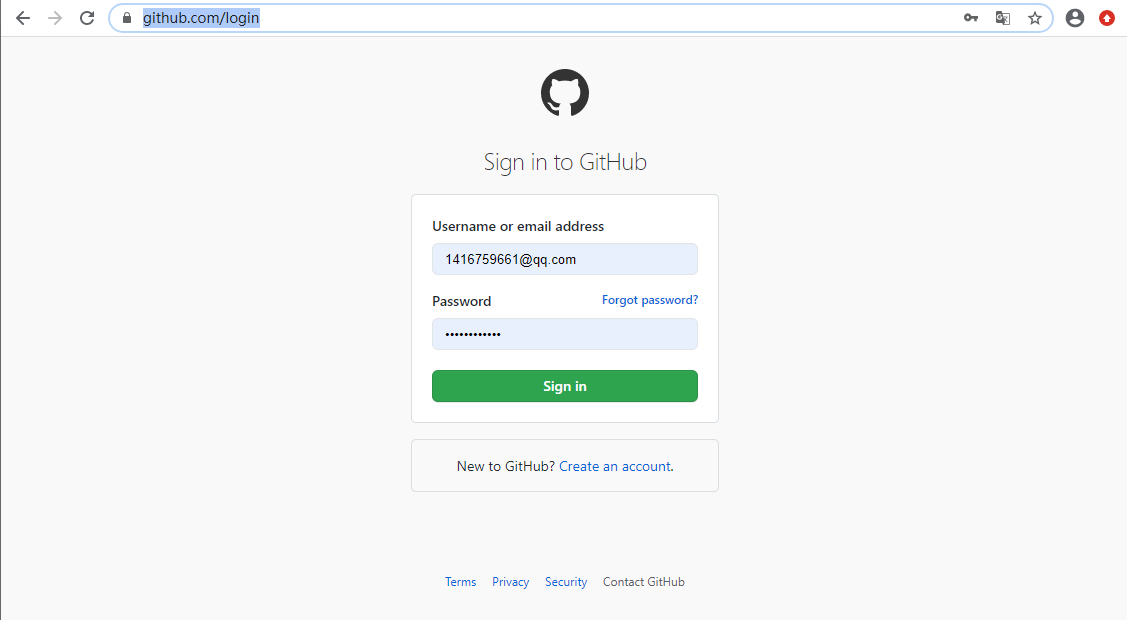
提示选择保存文件位置的时候直接回车，提示输入密码的时候可以设置一个简单的密码，建议统一设置为123456，再次输入密码的时候也输入123456,然后回车就可以了。

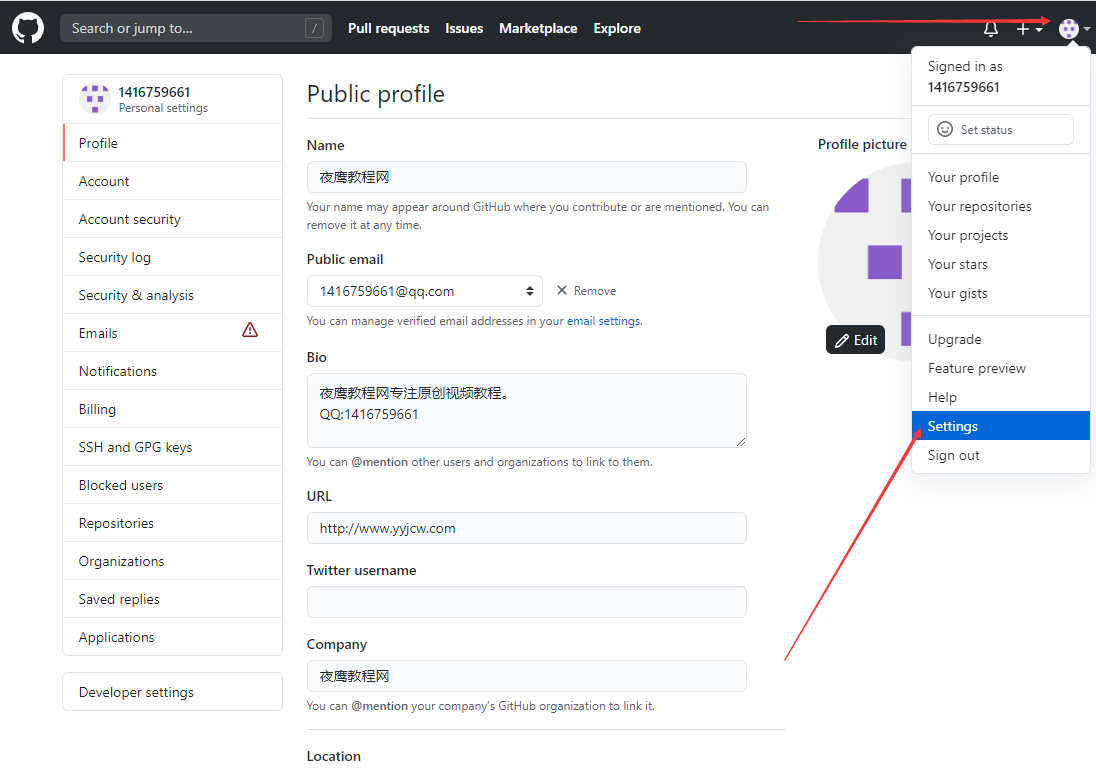


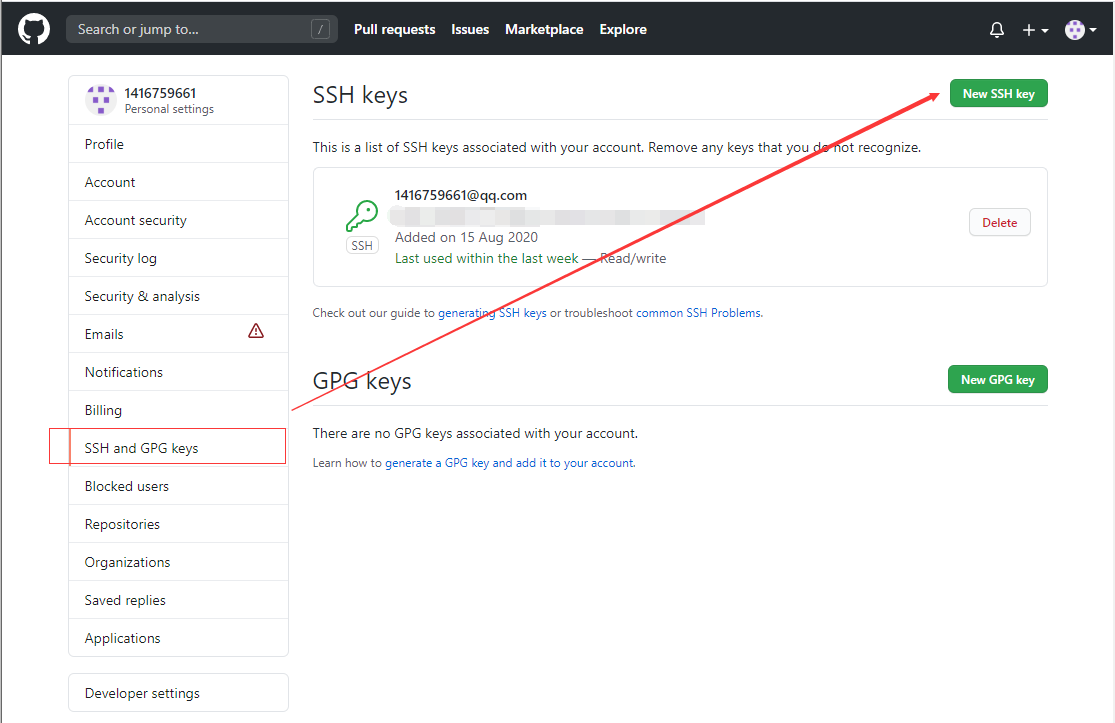
如果一切顺利的话，可以在用户主目录里找到.ssh目录，里面有id\_rsa和id\_rsa.pub两个文件，这两个就是SSH Key的秘钥对，id\_rsa是私钥，不能泄露出去，id\_rsa.pub是公钥，可以放心地告诉任何人。

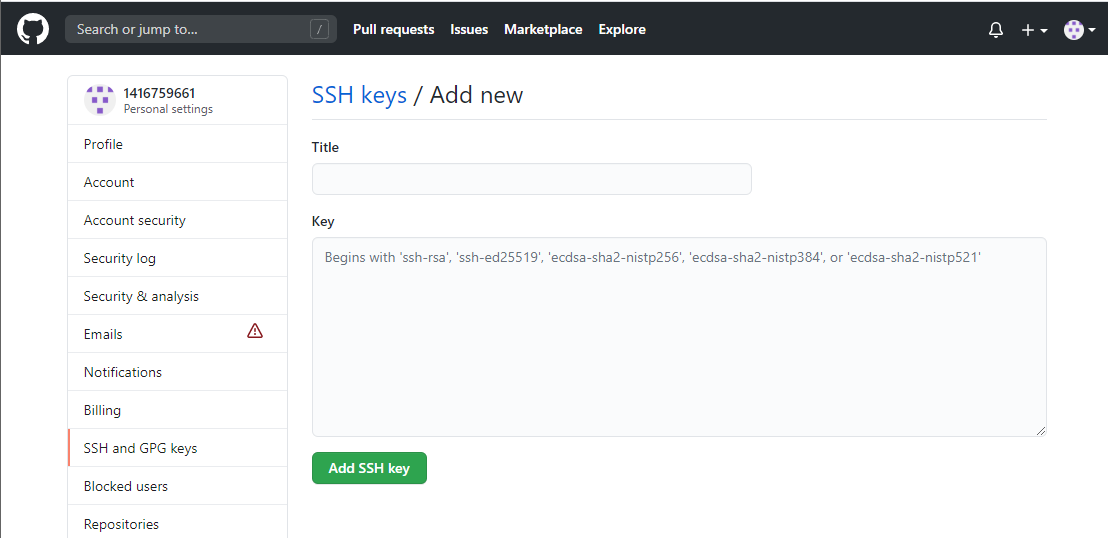


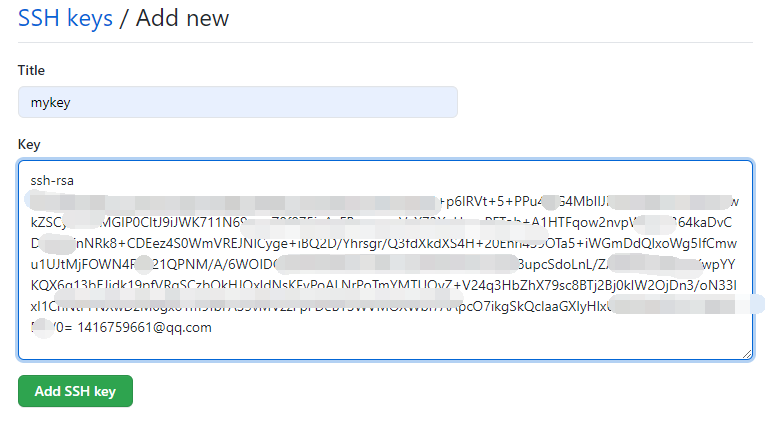
接下来打开网站<https://github.com/login> 如果没有账号需要先注册一个账号。

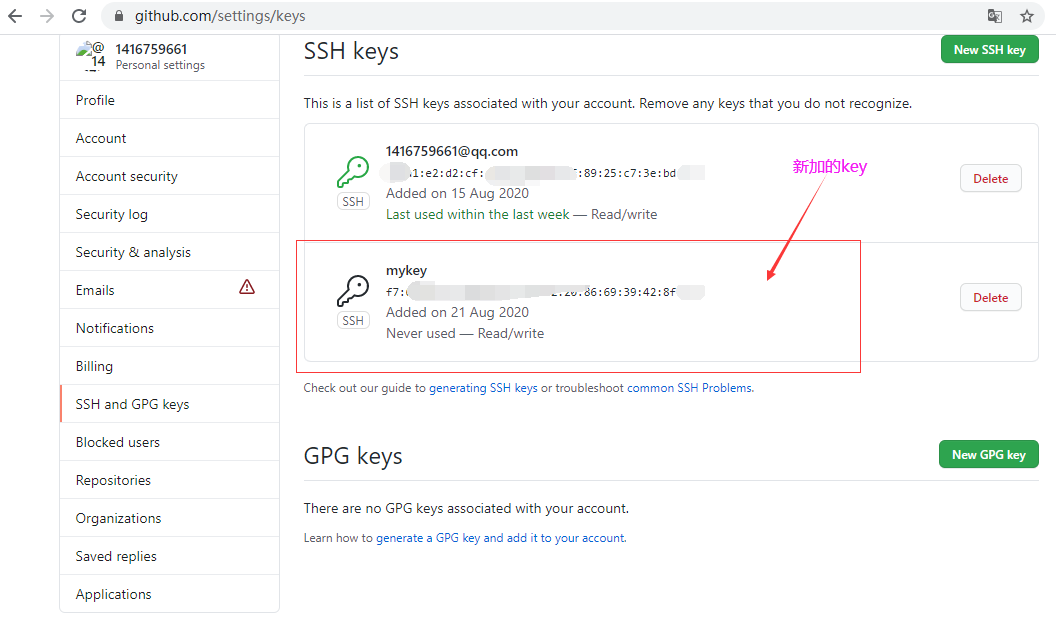


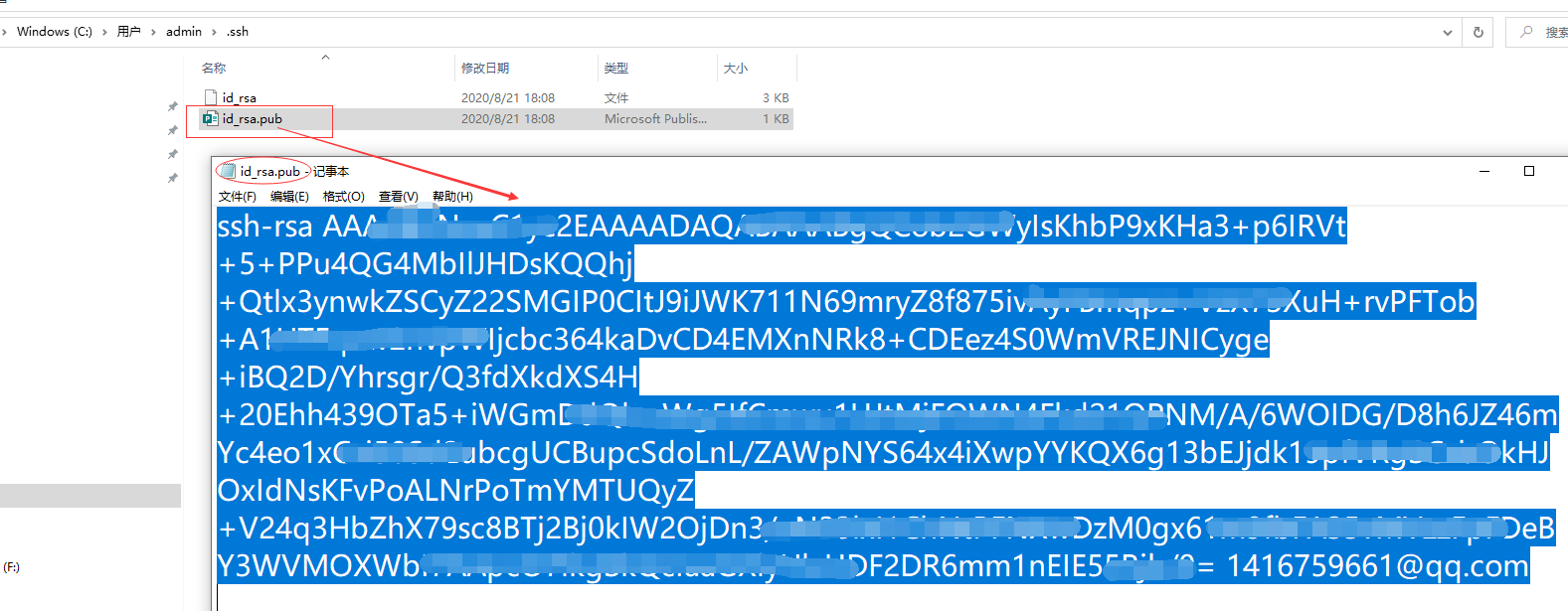












为什么GitHub需要SSH Key呢？因为GitHub需要识别出你推送的提交确实是你推送的，而不是别人冒充的，而Git支持SSH协议，所以，GitHub只要知道了你的公钥，就可以确认只有你自己才能推送。

当然，GitHub允许你添加多个Key。假定你有若干电脑，你一会儿在公司提交，一会儿在家里提交，只要把每台电脑的Key都添加到GitHub，就可以在每台电脑上往GitHub推送了。

最后友情提示，在GitHub上免费托管的Git仓库，任何人都可以看到喔（但只有你自己才能改）。所以，不要把敏感信息放进去。

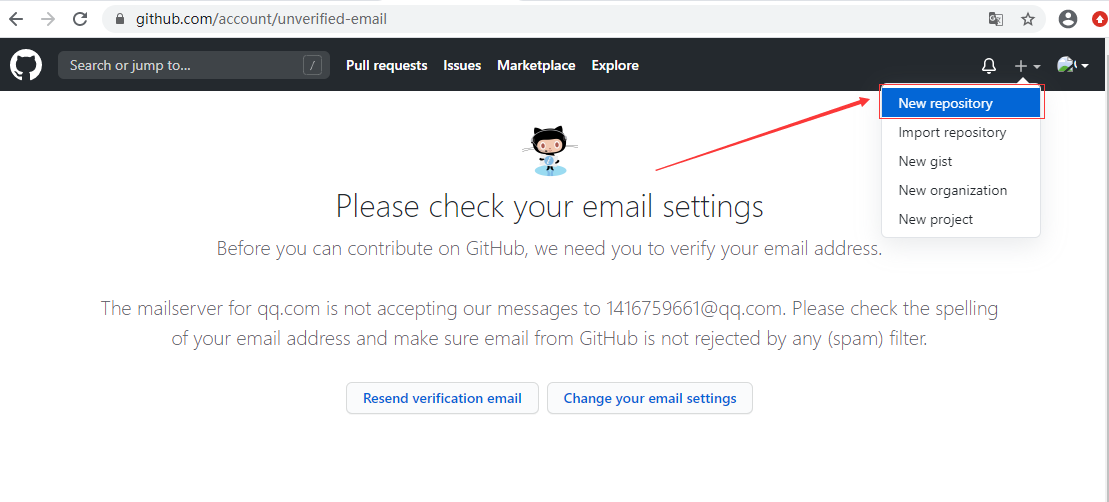
如果你不想让别人看到Git库，有两个办法，一个是交点保护费，让GitHub把公开的仓库变成私有的，这样别人就看不见了（不可读更不可写）。另一个办法是自己动手，搭一个Git服务器，因为是你自己的Git服务器，所以别人也是看不见的。

## 添加远程仓库

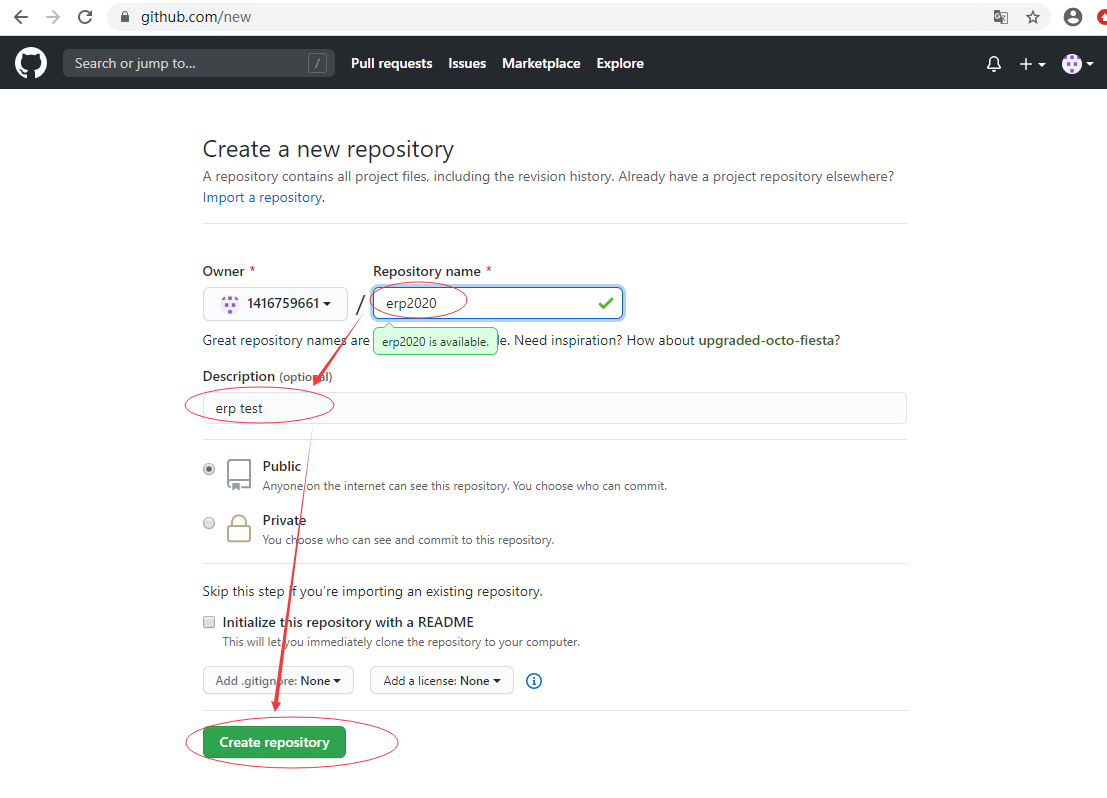
你已经在本地创建了一个Git仓库后，又想在GitHub创建一个Git仓库，并且让这两个仓库进行远程同步，这样，GitHub上的仓库既可以作为备份，又可以让其他人通过该仓库来协作，真是一举多得。

登陆GitHub，然后，在右上角找到“Create a new repo”按钮，创建一个新的仓库。

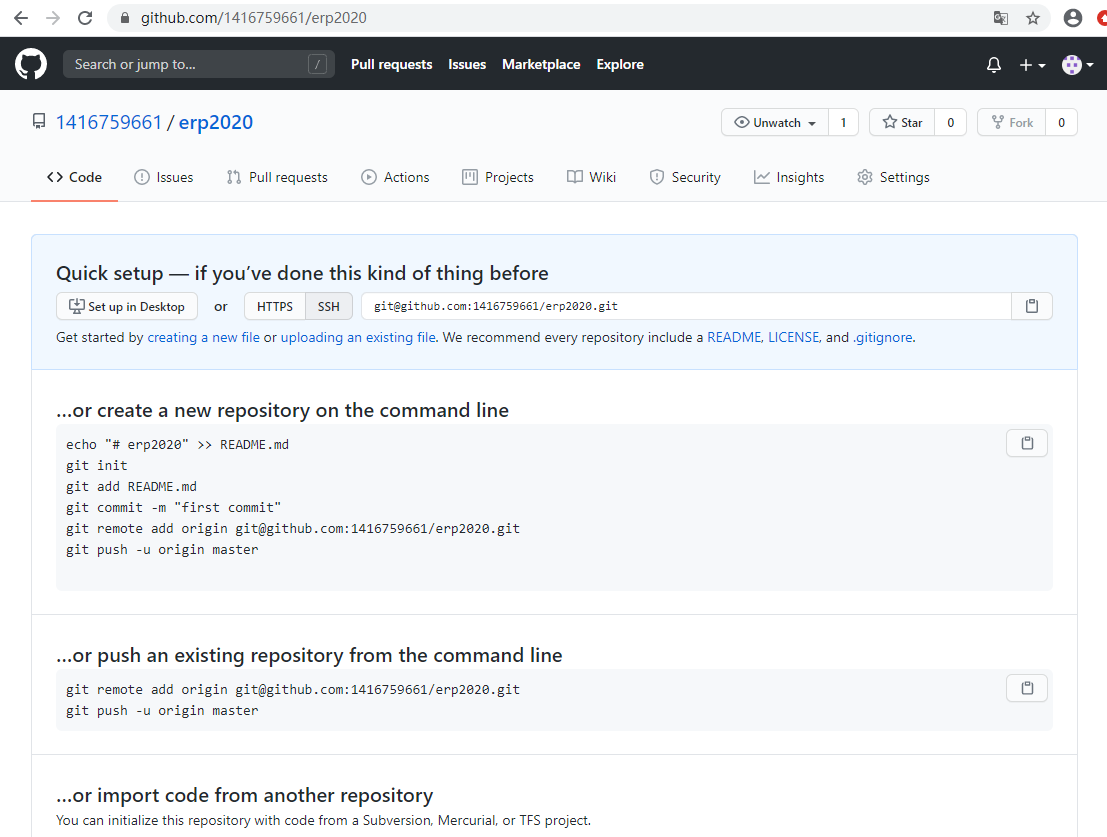
有时候需要验证电子邮箱。



点击new repository进入如下界面：



填写好项目名称和描述后 点击Create respository 创建远程仓库。



## 本地仓库关联远程仓库

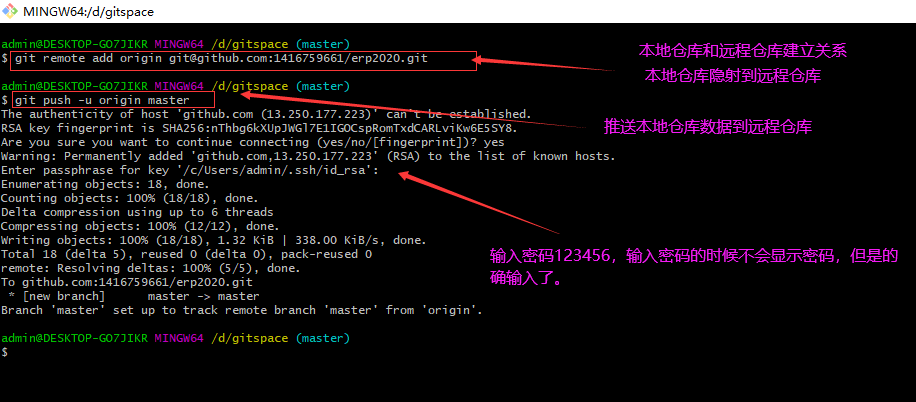
GitHub上的这个erp2020仓库还是空的.

我们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。

根据GitHub的提示，在本地的仓库下运行命令：

git remote add origin [git@github.com:1416759661/erp2020.git](mailto:git@github.com:1416759661/erp2020.git)

[上面的命令表示把 git@github.com:1416759661/erp2020.git](mailto:上面的命令表示把%20git@github.com:1416759661/erp2020.git) 这个作为远程仓库，并且取一个名称叫origin，也可以是别的名称，但是大家习惯都这样取，以看就知道是远程仓库。。



## 推送本地仓库数据到远程仓库

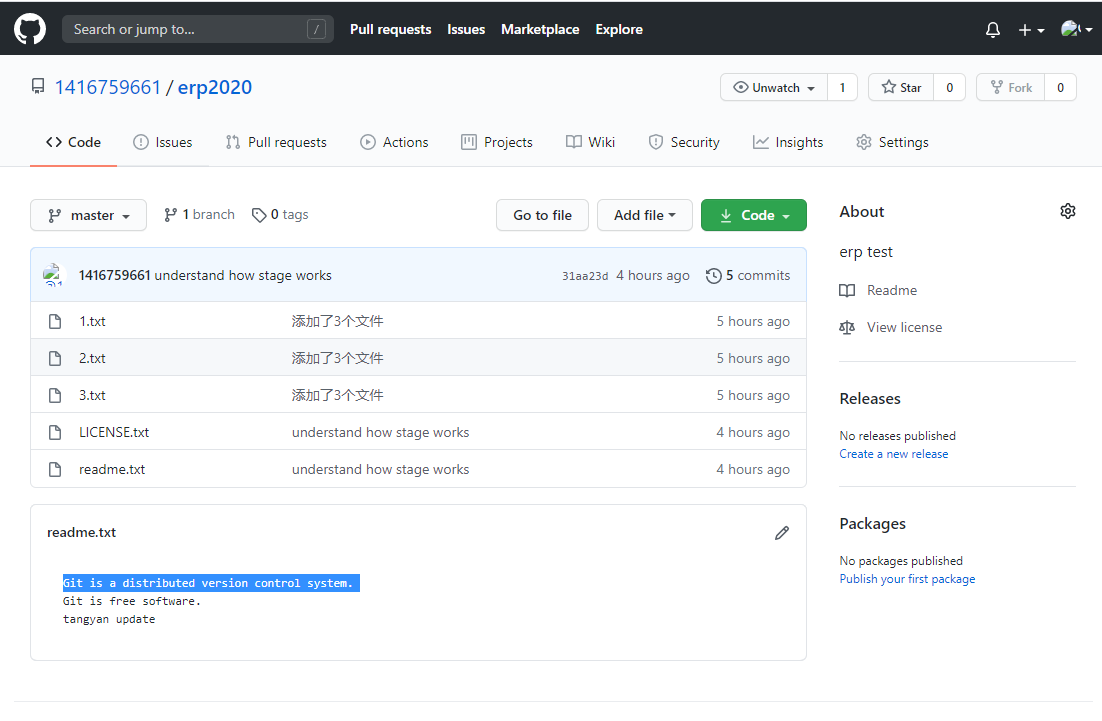
git push -u origin master 推送本地仓库数据到远程仓库。

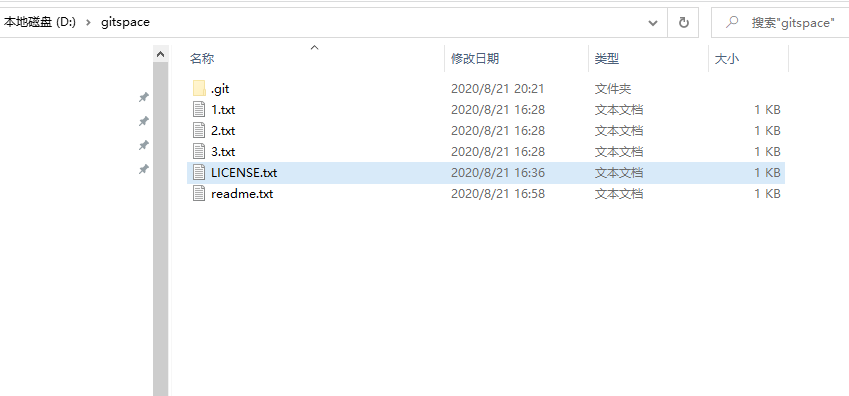
虽然本地仓库和远程仓库建立了关联，SSH Key公钥必须在账户列表中才可以顺利推送。

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，Git不但会把本地的master分支内容推送的远程新的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。

推送成功后，可以立刻在GitHub页面中看到远程库的内容已经和本地一模一样：





从现在起，只要本地作了提交，就可以通过命令：

git push origin master

把本地master分支的最新修改推送至GitHub，现在，你就拥有了真正的分布式版本库！

## SSH警告的解决办法

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.

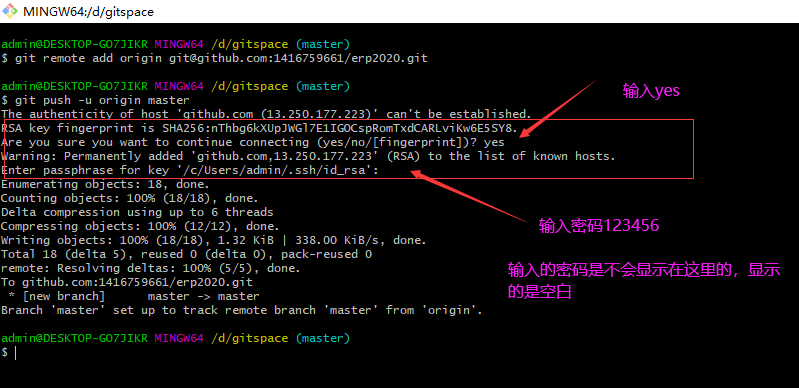
Are you sure you want to continue connecting (yes/no)?

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。



## 推送数据到远程仓库总结

1）要关联一个远程库，使用命令

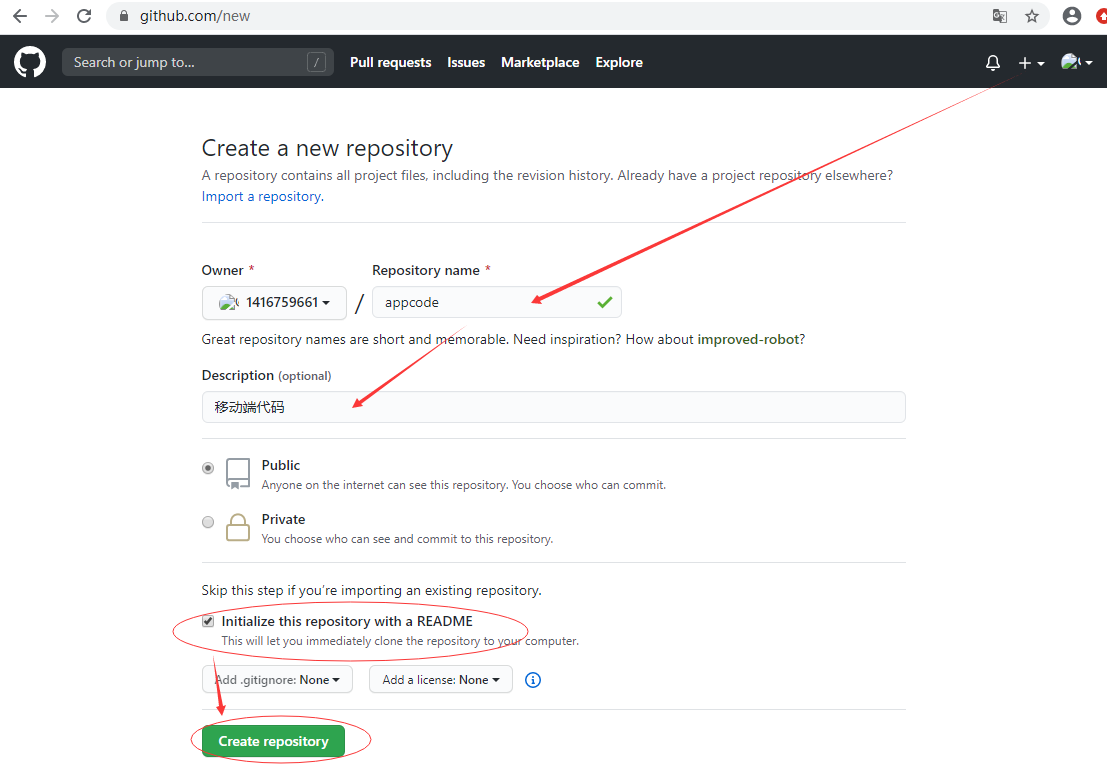
git remote add origin git@github.com:1416759661/erp2020.git

2）关联后，使用命令git push -u origin master第一次推送master分支的所有内容  
3）以后使用命令git push origin master推送本地最新修改到远程仓库

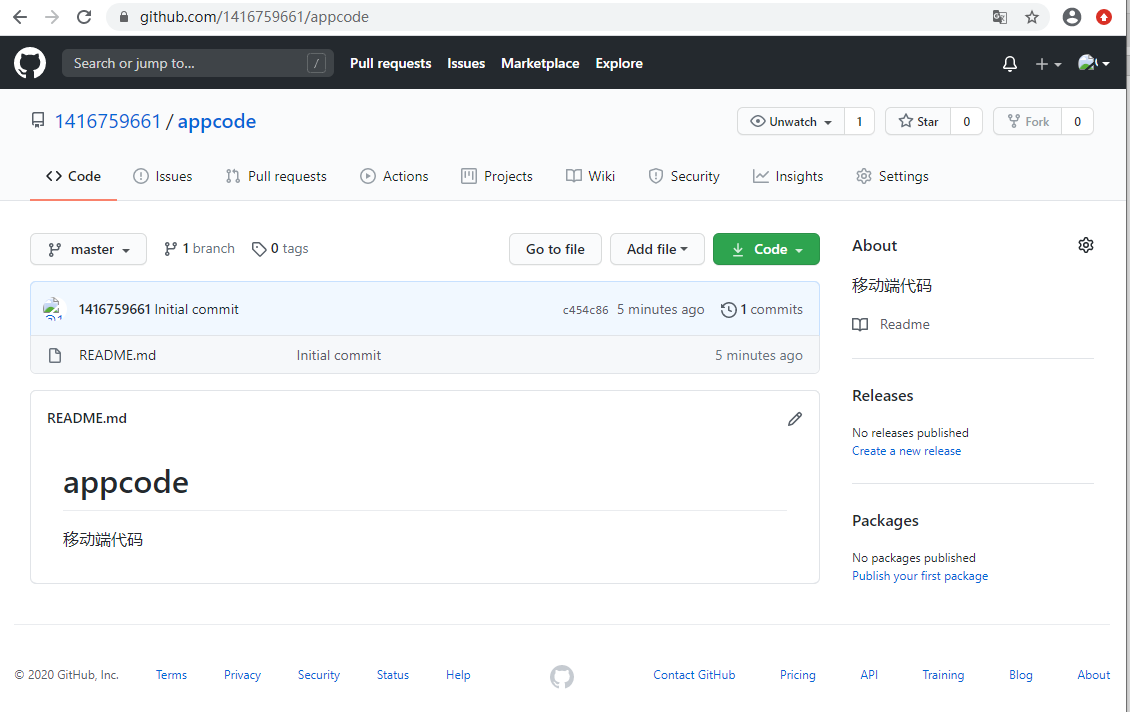
分布式版本系统的最大好处之一是在本地工作完全不需要考虑远程库的存在，也就是有没有联网都可以正常工作，而SVN在没有联网的时候是拒绝干活的！当有网络的时候，再把本地提交推送一下就完成了同步。

## 从远程仓库克隆

登陆GitHub，创建一个新的仓库，名字叫appcode

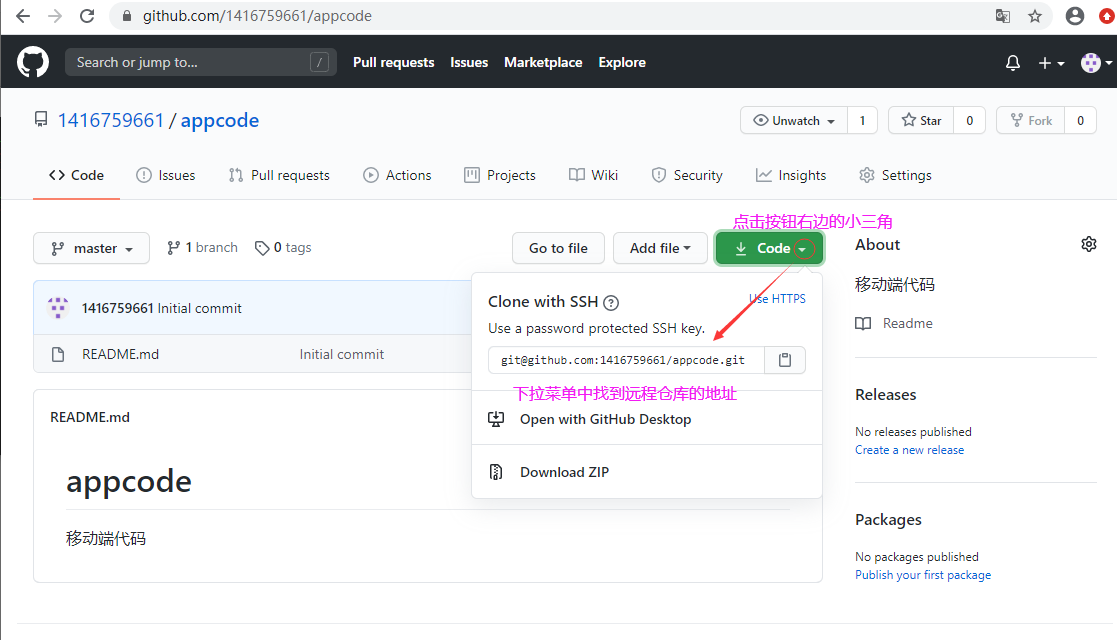


我们勾选Initialize this repository with a README，这样GitHub会自动为我们创建一个README.md文件。创建完毕后，可以看到README.md文件

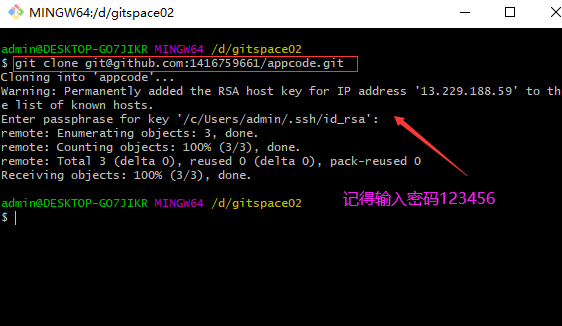


远程库已经准备好了，下一步是用命令git clone克隆一个本地库

在本地建立一个空文件夹，在文件夹上面点击右键,点击Git Bash Here。



使用命令 git clone [git@github.com:1416759661/appcode.git](mailto:git@github.com:1416759661/appcode.git) 下载远程仓库代码到本地仓库。



## 本地修改后重新推送到远程

