# 实验 1 Git与GitHub简介

1. Git 与 GitHub 的来历

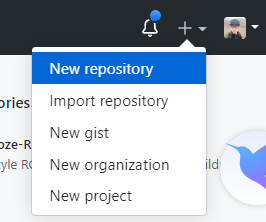
Linux 之父 Linus 在 1991 年创建开源的 Linux 操作系统之后，多年来依靠全世界广大热心志愿者的共同建设，经过长足发展，现已成为世界上最大的服务器系统。系统创建之初，代码贡献者将源码文件发送给 Linus，由其手动合并。这种方式维持多年后，代码量已经庞大到人工合并难以为继，于是深恶集中式版本控制系统的 Linus 选择了一个分布式商业版本控制系统 BitKeeper，不过 Linux 社区的建设者们可以免费使用它。BitKeeper 改变了 Linus 对版本控制的认识，同时 Linus 发现 BitKeeper 有一些不足，而且有个关键性的问题使之不能被广泛使用，就是不开源。

在 2005 年，BitKeeper 所在公司发现 Linux 社区有人企图破解它，BitKeeper 决定收回 Linux 社区的免费使用权。Linus 对此事调节数周无果，找遍了当时已知的各种版本控制系统，没有一个看上眼的，一怒之下决定自己搞一个。Linus 花了十天时间用 C 语言写好了一个开源的版本控制系统，就是著名的 Git。

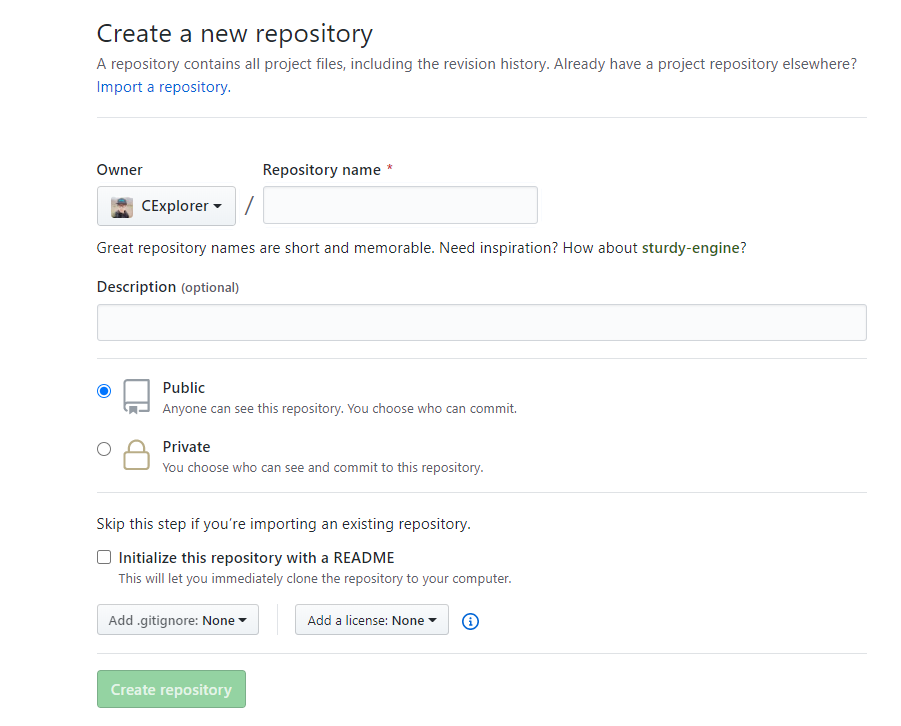
2007 年旧金山三个年轻人觉得 Git 是个好东西，就搞了一个公司名字叫 GitHub，第二年上线了使用 Ruby 编写的同名网站 GitHub，这是一个基于 Git 的免费代码托管网站（有付费服务）。十年间，该网站迅速蹿红，击败了实力雄厚的 Google Code，成为全世界最受欢迎的代码托管网站。2018 年 6 月，GitHub 被财大气粗的 Microsoft 收购。2019 年 1 月 GitHub 宣布用户可以免费创建私有仓库。根据 2018 年 10 月的 GitHub 年度报告显示，目前有 3100 万开发者创建了 9600 万个项目仓库，有 210 万企业入驻。

本课程将以图文的形式逐步讲解 GitHub 的使用以及 Git 实现版本控制。

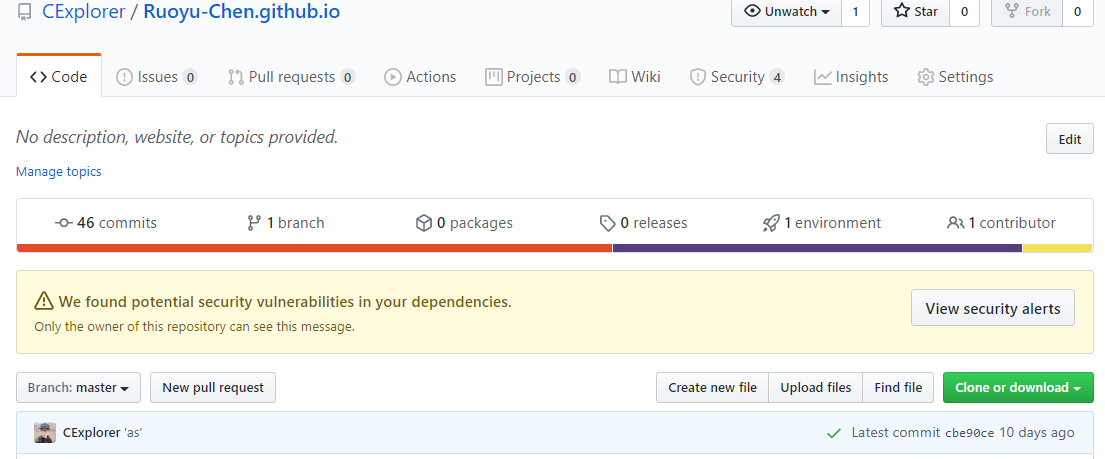
首先，打开 GitHub 注册个人账户并登录。登录后，在个人主页的右上角点击 New repository 创建新的仓库：



打开页面如下图所示，填入相关信息。注意下图紫色框中有两个下拉按钮，左边的用来选择忽略文件，右边的用来选择所属协议，这两项可以不选，后面的课程会讲到。



点击绿色按钮创建新的仓库，成功后自动跳转到新建仓库的主页面，如下图所示：



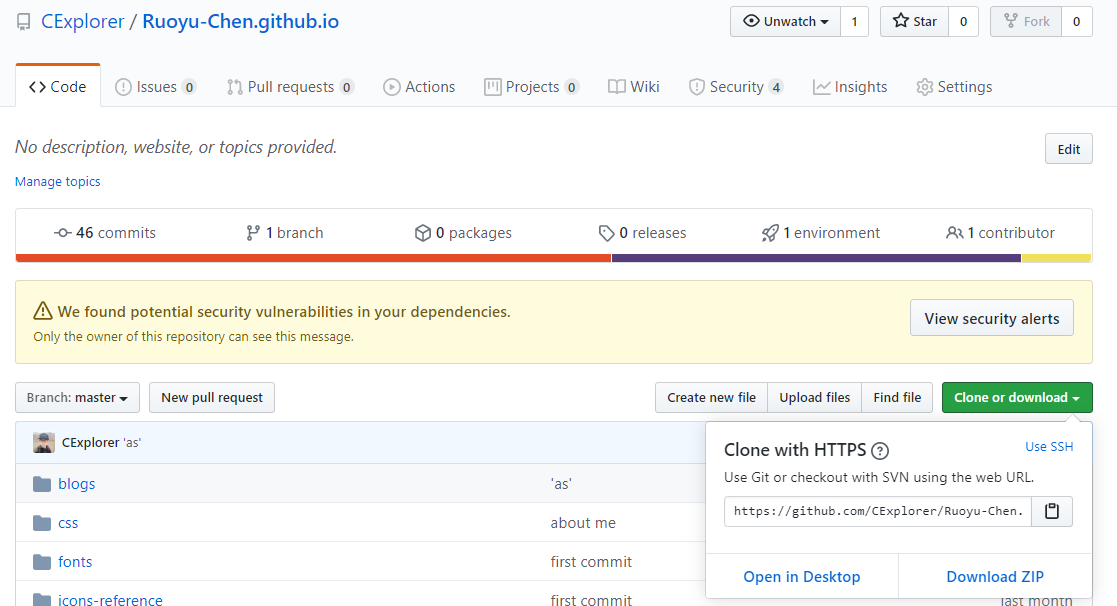
接下来，我们就要尝试使用这个仓库。

首先，打开实验环境。实验环境中内置了 Git 版本控制器，无需下载。打开终端使用 git --version 命令查看版本：

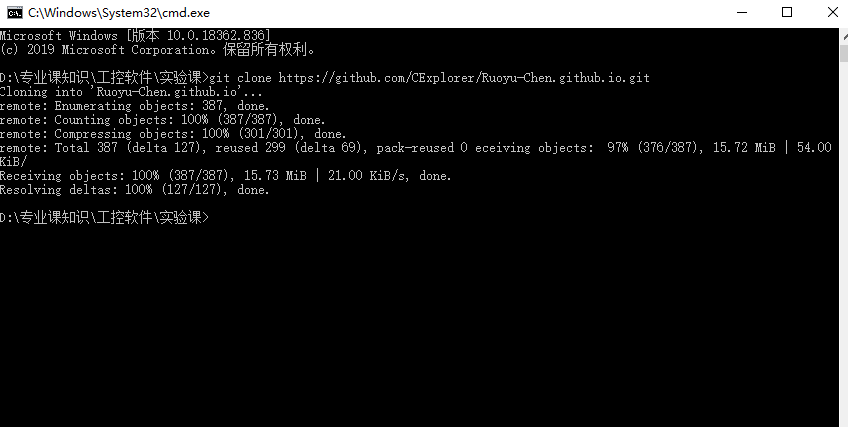


现在克隆前面我们在 GitHub 上创建的仓库，使用 git clone + [仓库地址] 命令即可，这是标准的克隆仓库命令。

点击下图绿色按钮，再点击紫色框中的按钮即可复制仓库地址，当然复制上面地址栏中的内容也是一样的。



克隆仓库到本地：



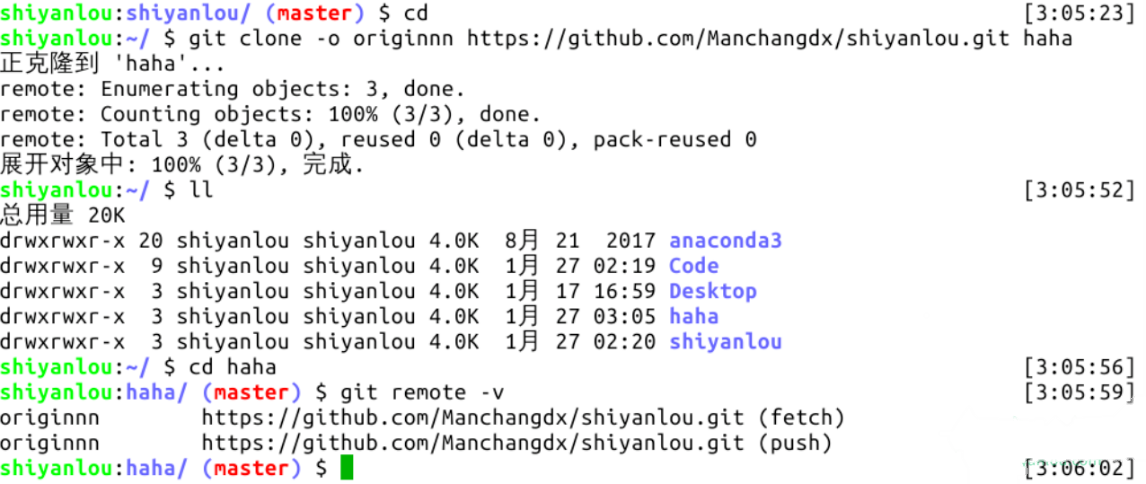
进入仓库主目录，如下图所示，仓库主目录中有个 .git 隐藏目录，它里面包含了仓库的全部信息，删掉这个目录，仓库就变成普通的目录了。进入到仓库目录中，命令行前缀发生了一些变化，出现了红色的 master ，它就是当前所在的分支名：

当我们在 GitHub 上创建一个仓库时，同时生成了仓库的默认主机名 origin，并创建了默认分支 master。GitHub 可以看成是免费的 Git 服务器，在 GitHub 上创建仓库，会自动生成一个仓库地址，主机就是指代这个仓库，主机名就等于这个仓库地址。克隆一个 GitHub 仓库（也叫远程仓库）到本地，本地仓库则会自动关联到这个远程仓库，执行 git remote -v 命令可以查看本地仓库所关联的远程仓库信息：



Git 要求对本地仓库关联的每个远程主机都必须指定一个主机名（默认为 origin），用于本地仓库识别自己关联的主机，git remote 命令就用于管理本地仓库所关联的主机，一个本地仓库可以关联任意多个主机（即远程仓库）。

克隆远程仓库到本地时，还可以使用 -o 选项修改主机名，在地址后面加上一个字段作为本地仓库的主目录名，举例如下：



另一个在其它 Git 教程中常见的命令 git init ，它会把当前所在目录变成一个本地仓库，因为有 GitHub 的存在，这个命令在我们的生产生活中用到的次数应该是零，除非你想费时费力自己搭建服务器。操作截图如下：



本节实验主要介绍了以下知识点：

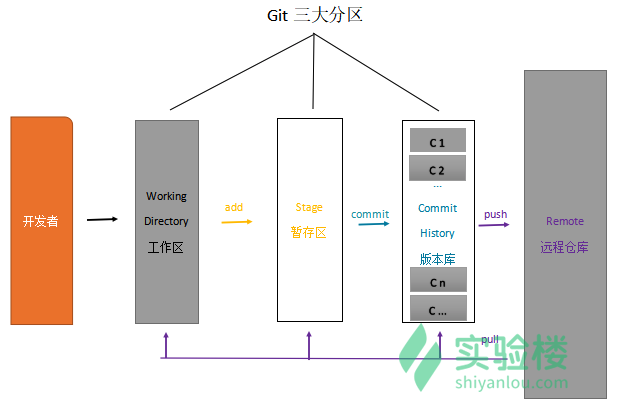
* Git 和 GitHub 的来历
* 使用 GitHub 创建仓库
* 安装 / 更新 Git
* 使用 Git 克隆远程仓库到本地

本节主要介绍了 Git 与 GitHub 概述，以及创建仓库、克隆仓库到本地的基本操作，相对比较简单。大家在学习时重复操作几次即可熟练掌握。下一节将学习 Git 的使用流程，完成一次修改、提交、推送操作。

# 实验 2 Git基础操作

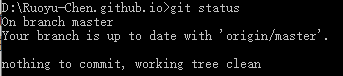
本节课程我们将完成一次从本地仓库修改代码、提交并推送到远程仓库的操作。

Git 本地仓库有三大区域：工作区、暂存区、版本区。这是一个概念，有这个了解即可，随着使用 Git 的时间增多，慢慢就会理解这三个区域的作用以及为何要这么设计，学习阶段只需按照文档逐步操作即可。接下来我们以命令为主线介绍 Git 的操作。

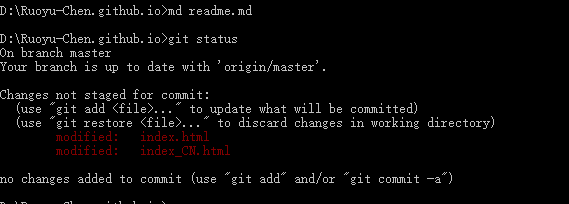


注意，所有 Git 命令都以 git 开头。

首先，进入仓库主目录，执行 git status 查看整个仓库的状态：

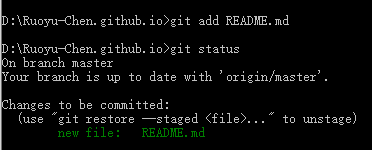


创建一个文件并再次查看仓库状态，这步操作是在工作区中：



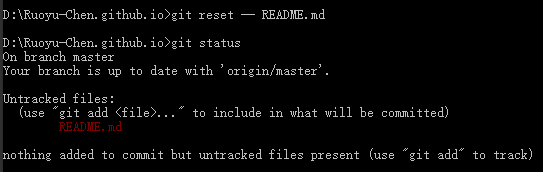
如上图所示，新建文件后，命令行前缀又发生了一些微小的变化，红色 master 后面出现了 \* 星号，这表示工作区或暂存区有变化，对文件进行增删改操作都会出现这个星号，另外使用 git status 命令亦可查看详情。

按照上图的提示，使用 git add [文件名] 命令跟踪此新建文件，即把新增文件添加到暂存区，以备提交：



如果对多个文件或目录进行了增删改，可以使用 git add . 命令全部添加到暂存区。

注意这里有个概念，当我们修改了工作区，git add 命令是将这些修改添加到暂存区，暂存区记录的只是修改。如果要撤销暂存区的修改怎么办？根据上图的提示，执行 git reset -- [文件名] 或者 git rm --cached [文件名] 命令即可：



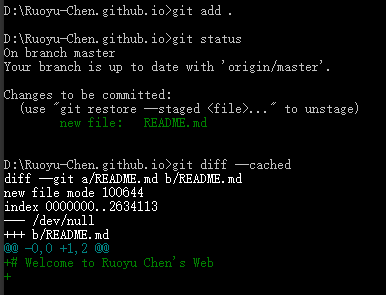
上图的命令，如果省略最后的文件名，把命令写成 git reset -- 即可把暂存区的全部修改撤销。好，现在暂存区的修改被撤销，又回到了工作区。

现在介绍另一个命令 git diff，它可以用来查看工作区被跟踪的文件的修改详情，此时新建文件 one.txt 并未被跟踪，而已被跟踪的文件 README.md 无修改，所以看不到。注意，只有在版本区中存在的文件才是被跟踪文件。

我们先修改 README.md 文件，然后执行此命令：

此时会跳到新的页面，即工作区修改详情页，按 Q 退出此页面：

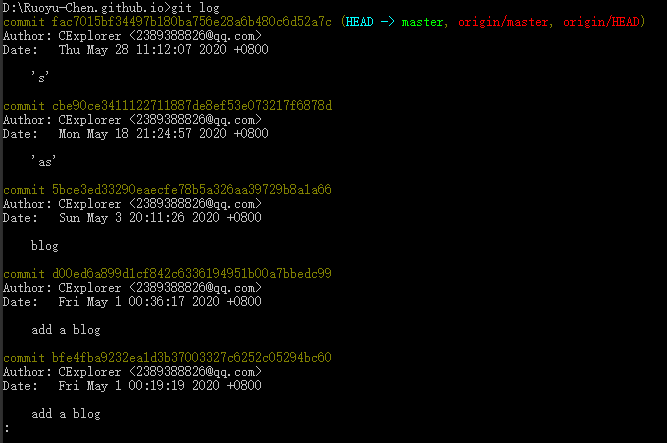
现在，将工作区的两处修改（新增文件 one.txt，修改文件 README.md）全部添加到暂存区，并使用 git diff --cached 查看暂存区的全部修改：



同样，此命令也会跳到新的页面，即暂存区修改详情页：

接下来，将执行 git commit 命令把暂存区的修改提交到版本区，生成一个新的版本。

在此之前，先介绍另一个命令 git log，它用来查看版本区的提交历史记录，当前只有一个提交，就是在 GitHub 上创建新仓库时的初始化提交。同样此命令也会跳到新页面，如下图所示：



关于查看提交历史记录的命令，有些常用的选项介绍一下：

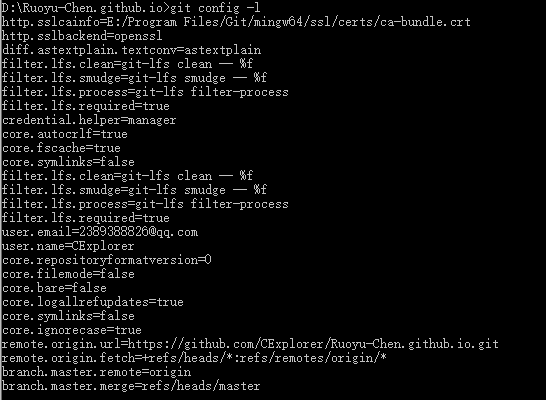
* git log [分支名] 查看某分支的提交历史，不写分支名查看当前所在分支
* git log --oneline 一行显示提交历史
* git log -n 其中 n 是数字，查看最近 n 个提交
* git log --author [贡献者名字] 查看指定贡献者的提交记录
* git log --graph 图示法显示提交历史

接下来需要对 Git 进行一些本地配置：

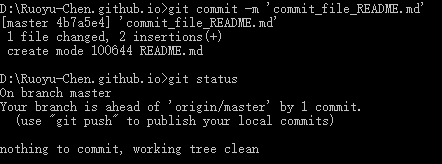
user.email：写入你自己注册 GitHub 账号的邮箱

user.name：你自己的 GitHub 账号名字

这两个命令设置你的身份信息如下图。git config -l 可以查看配置信息



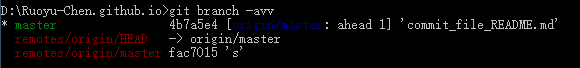
现在执行 git commit 命令生成一个新的提交，一个必须的选项 -m 用来提供该提交的备注：



提交后，暂存区的修改被清空，执行 git log 查看提交记录，紫色框中的十六进制序列号就是提交版本号，这是很重要的信息，每个提交都有自己单独的版本号，就像公民身份证号一样

观察上图的提交信息，提交版本是按时间倒序排列的，也就是最近的提交排在最上面，你可能需要查看时间正序排列的信息，那么可以使用 git log --reverse 命令。

现在介绍一个超级实用、使用频率极高但几乎所有 Git 教程都不重视的命令 git branch -avv，它用来查看全部分支信息：



上图有三行信息，依次说明:

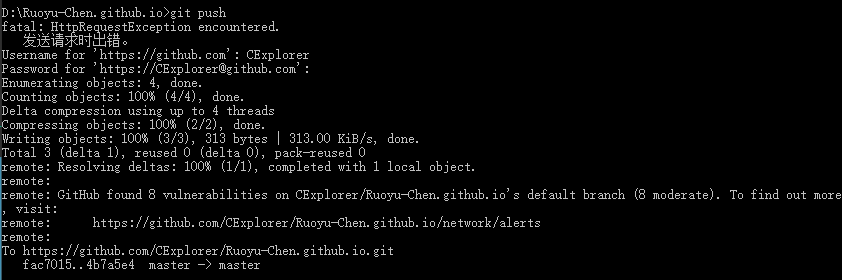
第一行，开头的星号表示当前所在分支，绿色的 master 是分支名，之所以是绿色，也是因为它是当前所在分支。后面第二项是版本号，第三项中括号里面蓝色的字，表示此分支跟踪的远程分支的名字，当然啦，这也是克隆远程仓库到本地时的默认设置 -- 创建 master 分支并自动跟踪远程同名分支；冒号后面黑色文字表示本地分支领先其跟踪的远程分支一个提交。最后一项是提交时填写的备注信息。

第二行，是 Git 指针信息，它指向远程仓库的 master 分支，这行信息暂不重要。

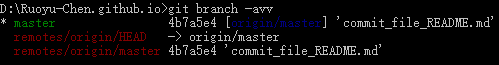
第三行，远程分支信息，详见第一行的解释。

在执行 commit 命令时，再介绍一个我并不推荐的选项 -a ，它的作用是将未添加到暂存区的修改，也就是工作区的修改也一并提交，但会略过未被跟踪的文件，比如新建文件 one.txt，此命令的完整格式：git commit -am xxxxx 。谨慎的做法是按照前文的顺序，修改工作区 - 提交到暂存区 - 随时使用 git status 查看仓库状态 - 将暂存区的修改提交到版本区生成一次新的提交。

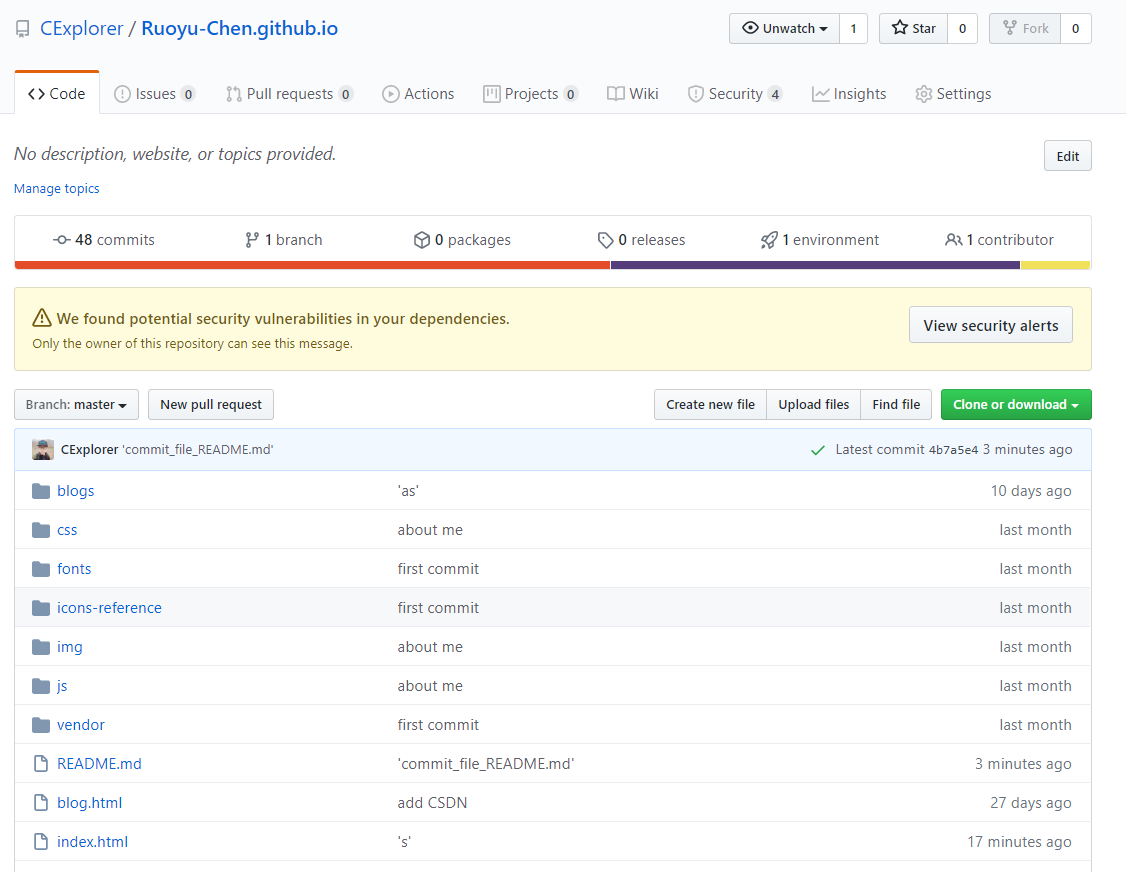
最后一个环节，将本地新增的提交推送到 GitHub 远程仓库中，命令是 git push，后面不需要任何选项和参数，此命令会把本地仓库 master 分支上的新增提交推送到远程仓库的同名分支上，因为当前所在的分支就是 master，而且上文提到，它已经跟踪了远程仓库的同名分支：



此命令需要再次输入用户名和密码，密码为隐藏数据，输入时看不到。推送成功后执行 git branch -avv 查看分支情况：

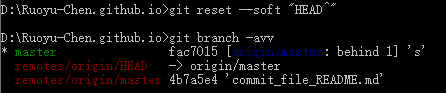


如上图所示，本地分支 master 与远程分支 origin/master 的版本号一致，通常看两个版本号是否一致，只需比对前四位。看一下网页上的情况：



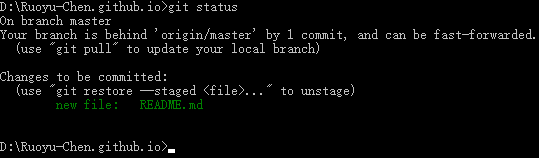
如果发现 one.txt 文件内容有误，怎么做？可以修改此文件然后再次添加到暂存区、提交、推送，也可以撤销最近一次提交，修改文件后重新提交推送。现在使用后一种方法来演示撤销提交的操作流程。

首先执行 git reset --soft “HEAD^” 撤销最近的一次提交，将修改还原到暂存区。--soft 表示软退回，对应的还有 --hard 硬退回，后面会讲到，HEAD^ 表示撤销一次提交，HEAD^^ 表示撤销两次提交，撤销 n 次可以简写为 HEAD~n。软退回一个提交后执行 git branch -avv 命令查看分支信息：

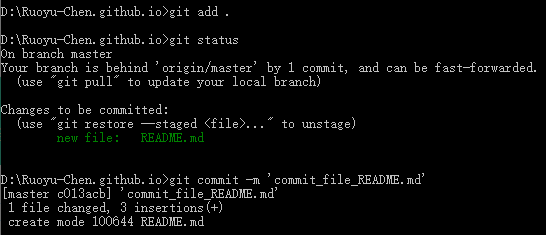


可以看到本地仓库的 master 分支的版本号已经发生了变化，变成了前一次提交的版本号，中括号里也有提示信息，本地分支 master 落后其跟踪的远程分支 origin/master 一个提交。

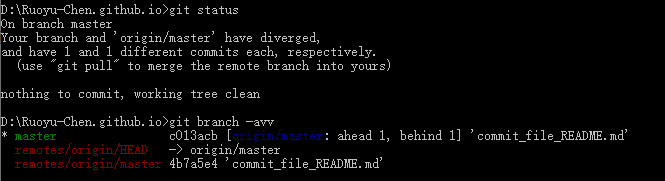
执行 git status 查看仓库状态，果然上一个提交中的修改全部扔回了暂存区：



再次修改 one.txt 文件，执行 git add . 命令将新的修改添加到暂存区，然后执行 git commit 命令生成新的提交：



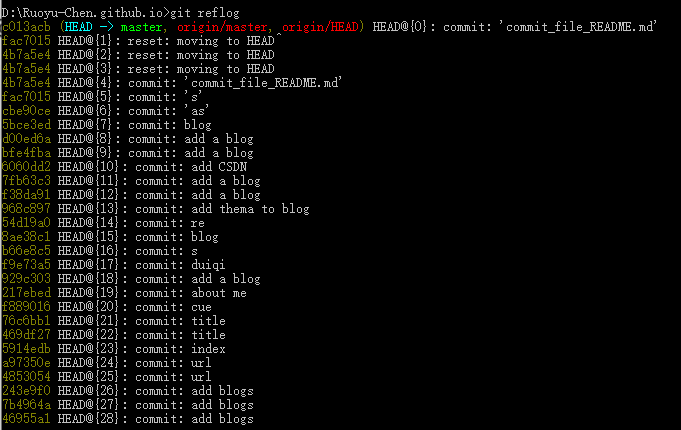
执行 git status 和 git branch -avv 查看仓库状态和分支状态：



可以看到本地仓库的 master 分支与远程仓库的 origin/master 分支在提交版本上有了冲突，又叫做提交时间线分叉。因为刚才的提交操作不是基于远程仓库 origin/master 分支的最新提交版本，而是撤回了一个版本。这种情况下也是可以将本地 master 分支推送到远程仓库的，需要加一个选项 -f ，它是 --force 的简写，这就是强制推送：



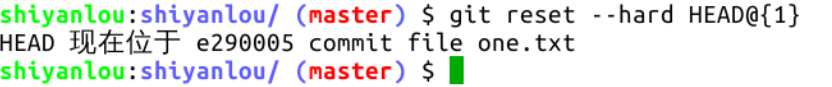
假设此时发现情况不对，之前的那次版本号为 5c04 的提交是正确的，刚才的版本回退操作全都是误操作，怎么办？再次执行一次版本回退吗？当然不需要啦，我们有 git reflog 命令，它会记录本地仓库所有分支的每一次版本变化。实际上只要本地仓库不被删除，随你怎么折腾，都能回退到任何地方。reflog 记录只存在于本地仓库中，本地仓库删除后，记录消失。执行此命令如下图所示：



怎么回退到 5c04 那个版本呢？可以直接执行命令 git reset --hard [版本号] ，如果记不清版本号，也可以根据上图第 3 行的信息，执行 git reset --hard HEAD@{2} 命令，其中 HEAD@{2} 就是上图第 3 行第 2 列所示，这个命令的意思是回到当前分支最近两次提交版本变化前：



还想反悔，刚才还是改对了，怎么办？再执行一次即可，这次大括号里就是 1 了：



重要的一点，本节全部命令中，只有 push 是需要联网执行的，它对远程仓库进行了修改。

本节实验包括以下内容的讲解：

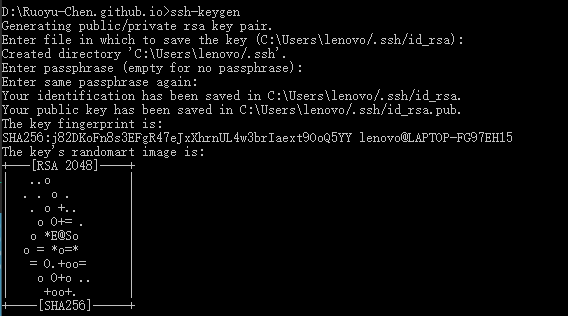
* Git 仓库的三大区域
* 修改工作区
* 将工作区的修改添加到暂存区
* 从暂存区撤销修改到工作区
* 查看提交历史
* 配置个人信息
* 完成一次提交
* 版本回退
* 处理提交时间线分叉问题
* 使用 git reflog 命令查看本地仓库版本变化

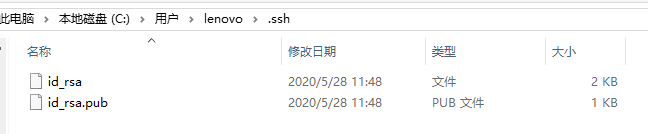
这些内容在实际操作中十分常见，希望大家多多练习，熟练掌握。下节课程，我们将学习设置命令别名、SSH 关联和分支管理的操作。

# 实验 3 Git分支操作

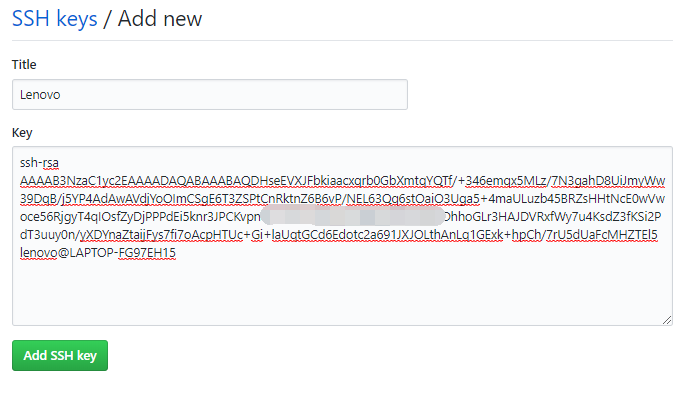
上一节操作中每次提交都要手动输入用户名和密码，若想避免这些麻烦，可以在系统中创建 SSH 公私钥，并将公钥放到 GitHub 指定位置。如此操作即可生成 GitHub 账户对于当前系统中的 Git 授权。

终端执行 ssh-keygen 命令按几次回车生成公私钥，公私钥存放在主目录下的隐藏目录 .ssh 中的两个文件中：

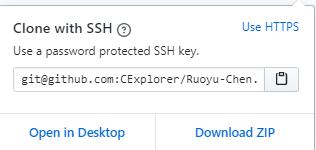




将 ~/.ssh/id\_rsa.pub 文件中的公钥内容复制出来，实验环境中可以使用右侧工具栏中的剪切板复制：



回到仓库主目录，点击下图所示的绿色按钮，点击紫色框中的 “Use SSH”，然后复制这个链接。



在实验环境里删除原仓库，使用此链接重新克隆仓库。克隆仓库是需要确认连接，输入 yes 即可：



重要的一点：只有使用这种 git 开头的地址克隆仓库，SSH 关联才会起作用。

使用 SSH 的好处主要有两点：

* 免密码推送，执行 git push 时不再需要输入用户名和密码了；
* 提高数据传输速度。它不是必须的，比如在实验楼的课程中挑战环境是不可保存的，一次性的，这种环境就没有必要创建 SSH 了，因为相较好处来说，还是太麻烦了。

上一节课程中的操作，有些命令的重复度极高，比如 git status 和 git branch -avv 等，Git 可以对这些命令设置别名，以便简化对它们的使用，设置别名的命令是 git config --global alias.[别名] [原命令]，如果原命令中有选项，需要加引号。别名是自定义的，可以随意命名，设置后，原命令和别名具有同等作用。操作如下：



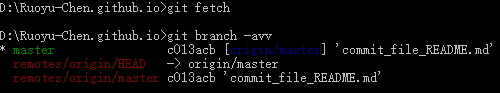
自己设置的别名要记住，也可以使用 git config -l 命令查看配置文件。下面文档中的命令将使用这些别名。

下面介绍 Git 作为分布式版本控制器最强大的功能：分支管理。

在介绍分支前，先讲解另一个命令 git fetch，它的作用是将远程仓库的分支信息拉取到本地仓库，注意，仅仅是更新了本地的远程分支信息，也就是执行 git branch -avv 命令时，查看到的 remotes 开头的行的分支信息。

举例说明一下，首先我们在 GitHub 页面上对 one.txt 文件进行修改并增加一次提交。

提交完成后，提交数变成 3 个，点下图紫色框中的链接可以看到提交记录：



可以看到，本地分支 master 的版本号无变化，而远程分支已经更新。所以，fetch 命令的作用是刷新保存在本地仓库的远程分支信息，此命令需要联网。此时若想使本地 master 分支的提交版本为最新，可以执行 git pull 命令来拉取远程分支到本地，pull 是拉取远程仓库的数据到本地，需要联网，而由于前面执行过 git fetch 命令，所以也可以执行 git rebase origin/master 命令来实现 “使本地 master 分支基于远程仓库的 master 分支”，rebase 命令在后面还会经常用到，这里只需按部就班操作即可：



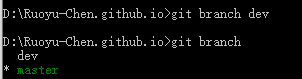
可以看到，远程仓库 master 分支、本地仓库的 origin/master 分支、本地仓库的 master 分支已经一致。

分支在项目开发中作用重大，多人协作时尤其不可或缺。例如一个项目上线了 1.0 版本，研发部门需要开发 1.1、1.2 两个测试版，增加不同的新功能，测试版的代码显然不能在正式版所在的分支上，此时需要新的分支来存放不同版次的代码。再例如实验楼的课程团队在维护课程仓库时，每个人都有各自的分支，在自己的分支上进行修改，然后向 master 分支提 PR（pull request），最后从 master 分支推送到线上。

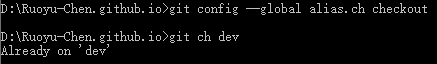
首先，克隆远程仓库到本地，进入仓库主目录，执行 git br 查看分支信息：



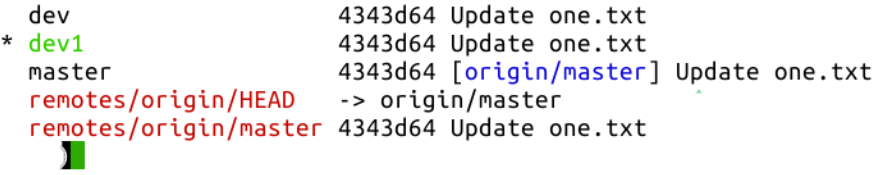
执行 git branch [分支名] 可以创建新的分支：



此命令创建新分支后并未切换到新分支，还是在 master 分支上，执行 git checkout [分支名] 切换分支，checkout 也是常用命令，先给它设置别名，然后切换分支：

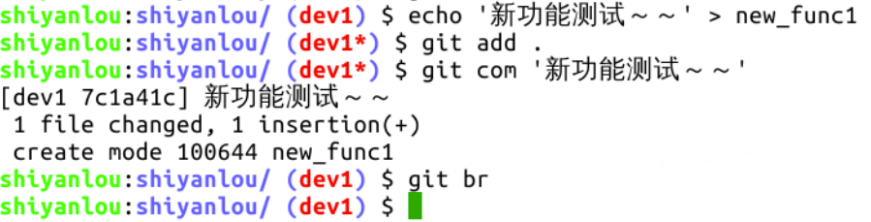


创建新分支还要手动切换太麻烦，介绍另一个常用的命令 git checkout -b [分支名] 创建分支并切换到新分支：

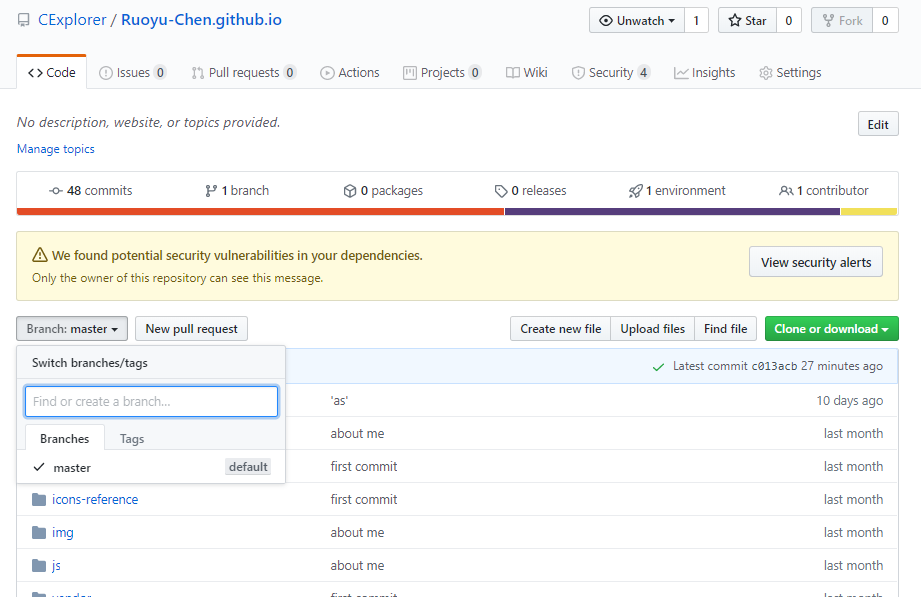


如上图所示的分支信息，前两行是新建的本地分支信息，它们的版本号与主分支 master 一致，这是因为在哪个分支上创建新分支，新分支的提交记录就与哪个分支一致。新建分支并无跟踪任何远程分支，所以没有 master 分支中的中括号和括号内的蓝色远程分支名。

假设我们要在当前分支 dev1 上开发一个新的功能，需要增加一个文件 new\_func1，然后生成一个新的提交：

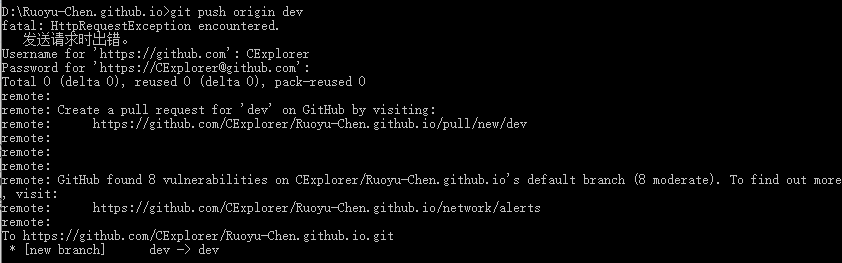


好，新功能已经写好并提交到了版本区，现在要推送了，推送到哪里呢？正常逻辑当然要推送到远程仓库的同名分支，不过现在远程仓库里只有一个分支：



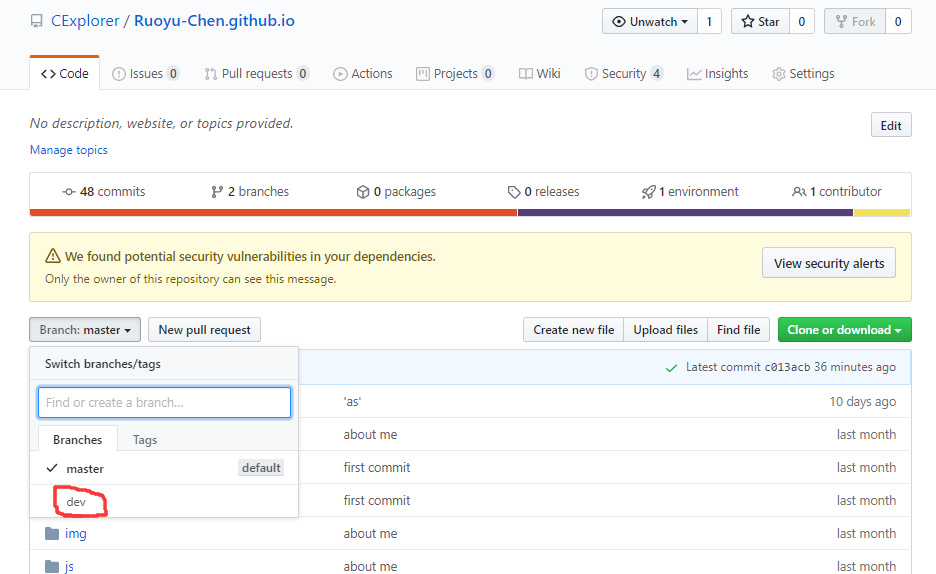
执行 git push [主机名] [本地分支名]:[远程分支名] 即可将本地分支推送到远程仓库的分支中，通常冒号前后的分支名是相同的，如果是相同的，可以省略 :[远程分支名]，如果远程分支不存在，会自动创建：

上图命令可以简写为 git push origin dev1 。注意哦，这是我们创建 SSH 关联后第一次执行 push 命令，可以看到传输速度有明显的提高，更重要的是，不再需要重复输入用户名和密码了，另外打印信息的第一行是警告信息，因为是这个分支的第一次推送嘛，下次执行推送就不会再出现了。现在执行 git br 查看一下分支情况：

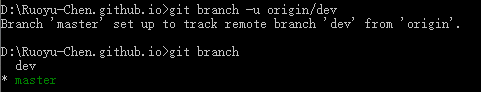




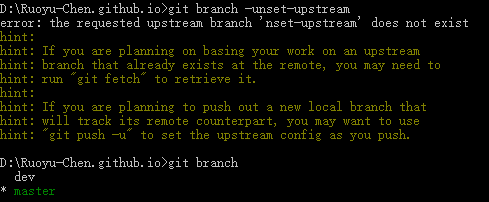
可以看到，远程分支 origin/dev 的信息已经在本地存在，且与本地同名分支一致。再看下 GitHub 页面的情况：



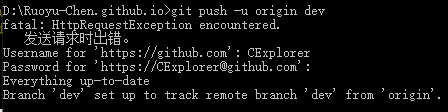
现在有个问题，当我们再次在 dev1 分支上修改并提交，推送到远程仓库时还是要输入上面的那个长长的命令，好不方便。如果能和 master 分支一样跟踪远程同名分支，就可以直接使用 git push 命令推送了。有办法的，执行这个命令 git branch -u [主机名/远程分支名] [本地分支名] 将本地分支与远程分支关联，或者说使本地分支跟踪远程分支。如果是设置当前所在分支跟踪远程分支，最后一个参数本地分支名可以省略不写：



这个命令的 -u 选项是 --set-upstream 的缩写。可不可以让本地分支跟踪远程非同名分支呢？可以的，尽管几乎遇不到这种自找麻烦的需求。可不可以撤销本地分支对远程分支的跟踪呢？也是可以的，执行 git branch --unset-upstream [分支名] 即可撤销该分支对远程分支的跟踪，同样地，如果撤销当前所在的分支的跟踪，分支名可以省略不写：

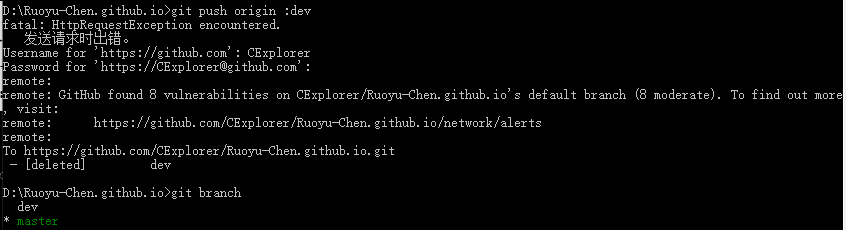


问题又来了，前面的操作是先将本地分支推送到远程仓库，使远程仓库创建新分支，然后再执行命令使本地分支跟踪远程分支，有没有办法在推送时就自动跟踪远程分支呢？有的，在推送的时候，加个 --set-upstream 或其简写 -u 选项即可，现在切换到 dev 分支试一下这个命令：

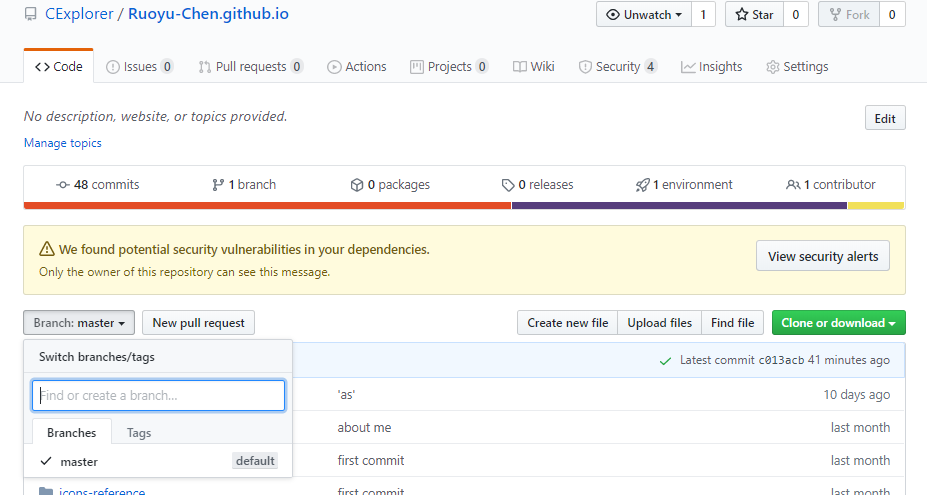


接下来，介绍一下删除分支的方法。

首先，删除远程分支，使用 git push [主机名] :[远程分支名] ，如果一次性删除多个，可以这样：git push [主机名] :[远程分支名] :[远程分支名] :[远程分支名] 。此命令的原理是将空分支推送到远程分支，结果自然就是远程分支被删除。另一个删除远程分支的命令：git push [主机名] --delete [远程分支名]。删除远程分支的命令可以在任意本地分支中执行。两个命令分别试一下：



可以看到本地仓库已经没有远程分支 dev 和 dev1 的分支信息。查看 GitHub 仓库页面：

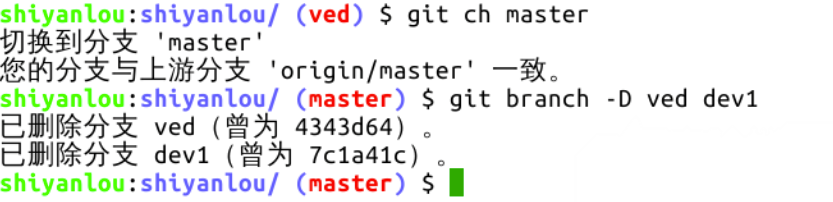


也只剩 master 一个分支。操作成功。

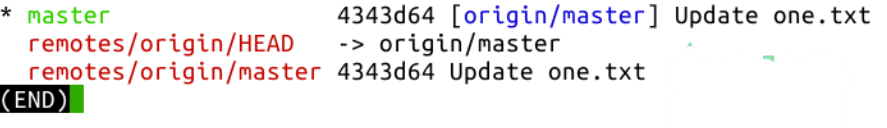
回到实验环境，使用 git branch -D [分支名] 删除本地分支，同样地，此命令也可以一次删除多个，将需要删除的分支名罗列在命令后面即可。在此之前，先介绍一个极少用到的命令：给本地分支改名 git branch -m [原分支名] [新分支名] ，若修改当前所在分支的名字，原分支名可以省略不写：

此处输入图片的描述

好，现在要一次性删除本地分支 ved 和 dev1。需要注意的一点：当前所在的分支不能被删除。切换到 master 分支，然后执行 git branch -D ved dev1 命令：



执行 git branch -avv 查看当前仓库分支状态：



本节实验主要讲解了以下知识点：

* 添加 SSH 关联授权
* 为 Git 命令设置别名
* git fetch 刷新本地分支信息
* 创建新的本地分支
* 将新分支中的提交推送至远程仓库
* 本地分支跟踪远程分支
* 删除远程分支
* 本地分支的更名与删除

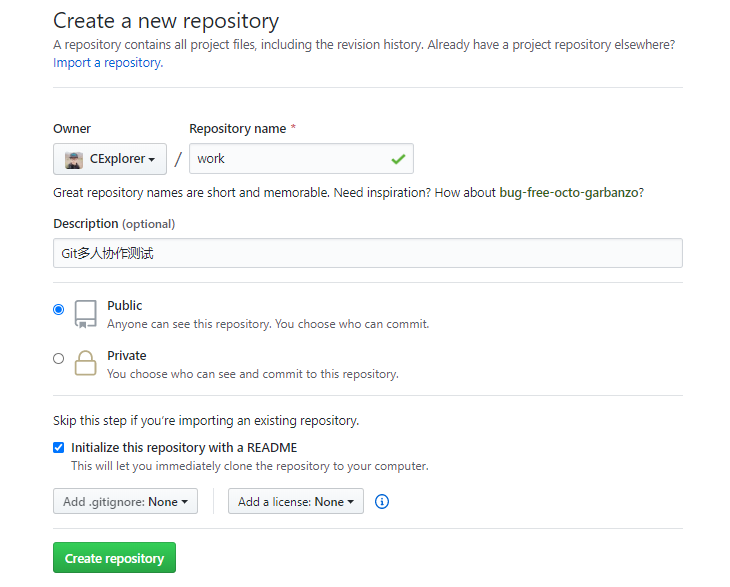
希望大家通过本节课程的学习，真正掌握创建分支、关联远程分支、撤销关联、删除分支等必备技能。下节课程我们将学习多人协作中的 Git & GitHub 操作流程。

# 实验 4 多人协作GitHub部分

本节将介绍 GitHub 多人协作与相关 Git 的操作，所有操作全部在浏览器页面上完成，内容相对较少。

建议大家准备两个浏览器和两个 GitHub 账号以便模拟场景。我的账号 Chuntianlaile 是一个用来测试的账号，假定这是项目组长的账号，Manchangdx 是组员的账号。

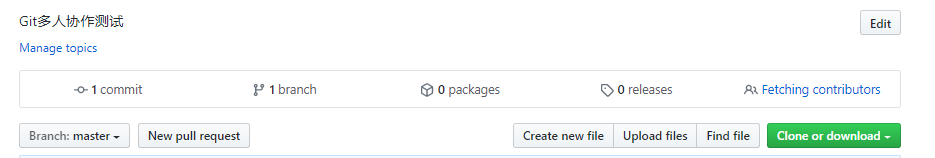
首先，在组长账号中创建一个仓库，名为 work，在创建仓库时，需要说明第一节中提到的两个下拉框：



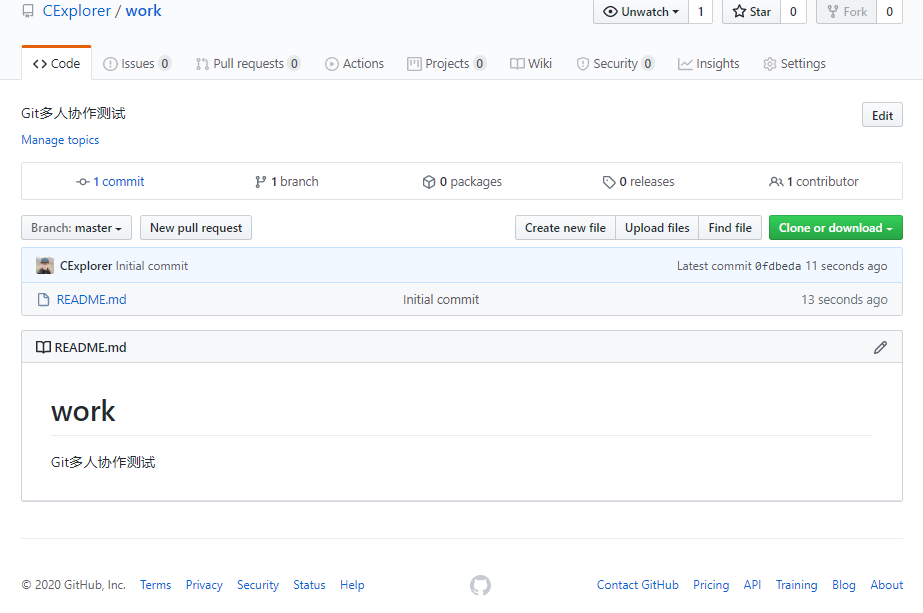
左边的忽略文件下拉框：我们在写代码时，总会出现一些不需要上传到仓库的垃圾文件、缓存文件、备份文件、环境文件等等，可以创建一个忽略文件将这些不需要被上传到远程仓库的文件忽略掉。忽略文件的名字是 .gitignore，它被放置在仓库主目录下，将不需上传的文件的名字写入其中，Git 就会自动忽略它们。比如这个仓库是用来 Windows 开发的，就在下拉框中选择 Windows，如果这是一个保存 Java 项目的仓库，就选择 Java。这样，在仓库创建成功后，忽略文件就自动出现了，这个忽略文件中有对应的语言或工具中绝大部分通用的忽略规则。当然了，你也可以自己手动增删改。

如果在创建仓库时忘记了选择忽略文件，几个提交后突然想起来，怎么办？GitHub 上有人把忽略文件都做好了，打开链接 github / gitignore ，这个仓库里有很多忽略文件，选择你需要的放到自己的仓库即可。

右边的开源许可下拉框：关于开源许可证，不属于本课程所述范围，如有需要大家可以自行搜索。我们的仓库不需要选择这一项。选择这个之后，仓库中会出现相对应的图标，比如上面提到的忽略文件仓库：



在组长账号中创建好新仓库，如下图：



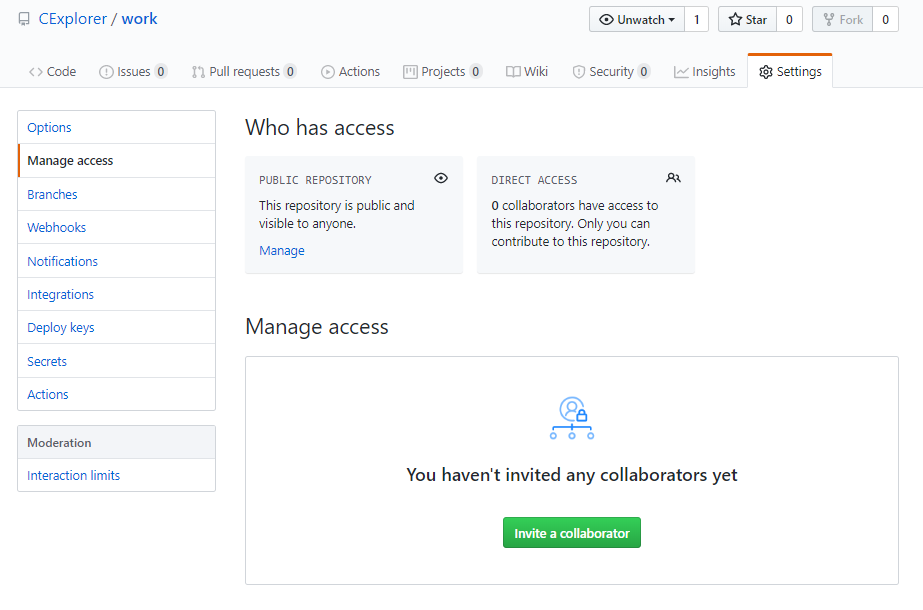
对上图右上角三个按钮进行说明：

Watch：这是一个下拉按钮，可以选择对此仓库关注、不关注、忽略等。

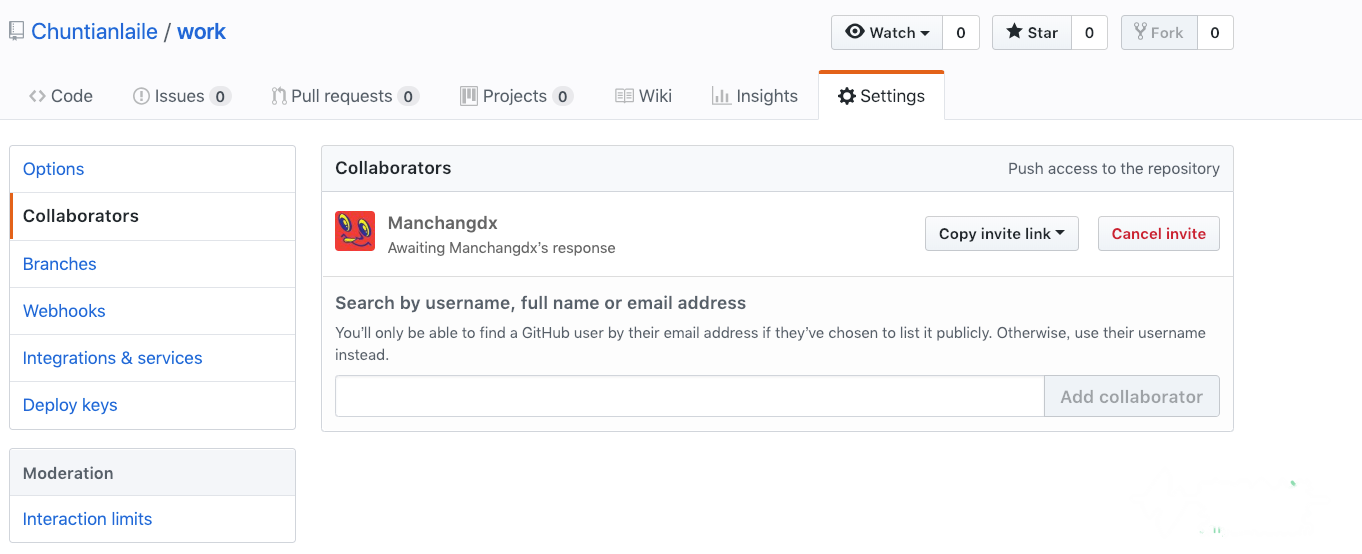
Star：如果觉得这个仓库很好，就点击这个按钮送一颗星，在淘宝提供刷星业务之前，仓库获得的星越多表示该项目越优秀。

Fork：在别人的仓库中点此按钮会克隆一个完全一样的仓库到你自己的账号中，包括所有分支、提交等，但不会克隆 issue（本节后面会讲到），当此仓库发生版本变化，不会自动同步到你克隆的仓库里，反之亦然。

现在在组长账号中增加该仓库的合作者，也就是组员：



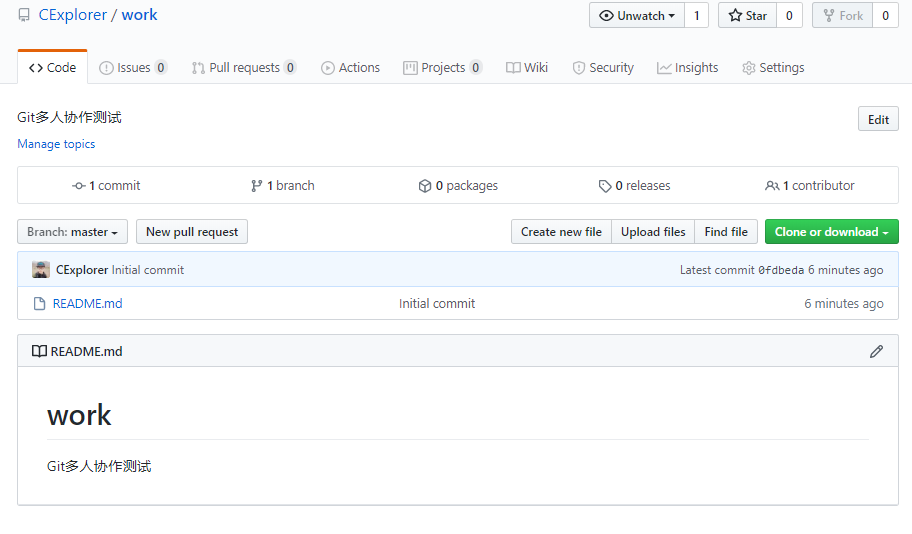
在浅蓝色输入框中写入组员 GitHub 账号的用户名，选择正确的用户，点击右侧按钮就会发送一封邀请邮件给组员：



现在使用另一个浏览器登录组员的 GitHub 账号和邮箱，打开邮件：

再次点击绿色按钮接受邀请，会跳转到组员访问组长仓库的页面：

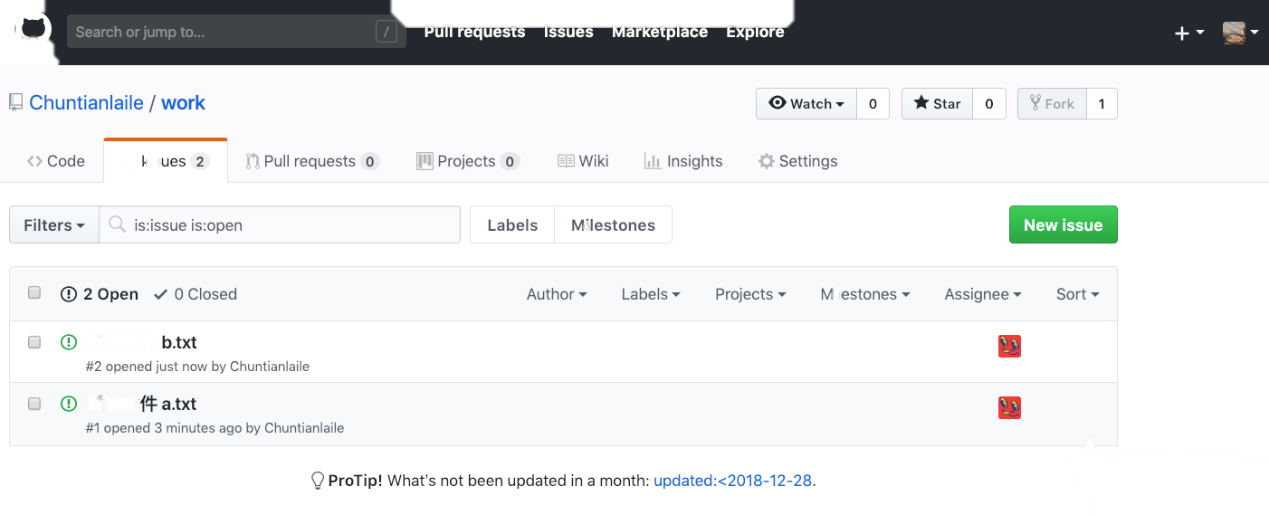
点击上图紫色框中的 Fork 按钮，克隆组长的仓库到组员的账号中，完成后自动跳转到下图页面，也就是组员的仓库页面：



切换到组长的 GitHub 页面，在仓库中添加一些项目任务或待解决问题，这些任务就是 issue：

写好任务标题后，可以在右侧指派一位或多位项目参与者来完成，同样 GitHub 也会给被指派者发邮件的（可以在自己的 GitHub 账号上设置拒收哪类邮件）：

写好两个 issue，前面说过的，组长仓库里的 issue 不会出现在组员仓库中：



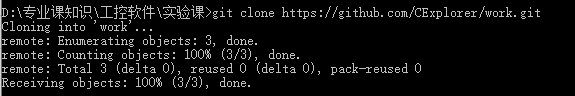
本节实验全部在 GitHub 页面上完成，主要介绍以下内容：

* 忽略文件
* Fork 仓库
* 添加合作者
* 编写 issue

以上是 GitHub 上多人协作最基础的操作，当然还有其它丰富实用的功能可以按需使用。下一节将学习多人协作的 Git 操作部分。

# 实验 5 多人协作Git部分

打开实验环境，以组员的身份克隆自己的 work 仓库到实验环境，由于之前已经设置了实验环境的 SSH 公钥到 GitHub，所以我们使用 git 开头的地址来克隆：



现在我们要完成组长仓库的一个 issue，注意每个 issue 在创建后都会生成一个编号，我们首先完成 1 号 issue：

创建文件，添加到暂存区，提交，查看本地仓库分支状态：

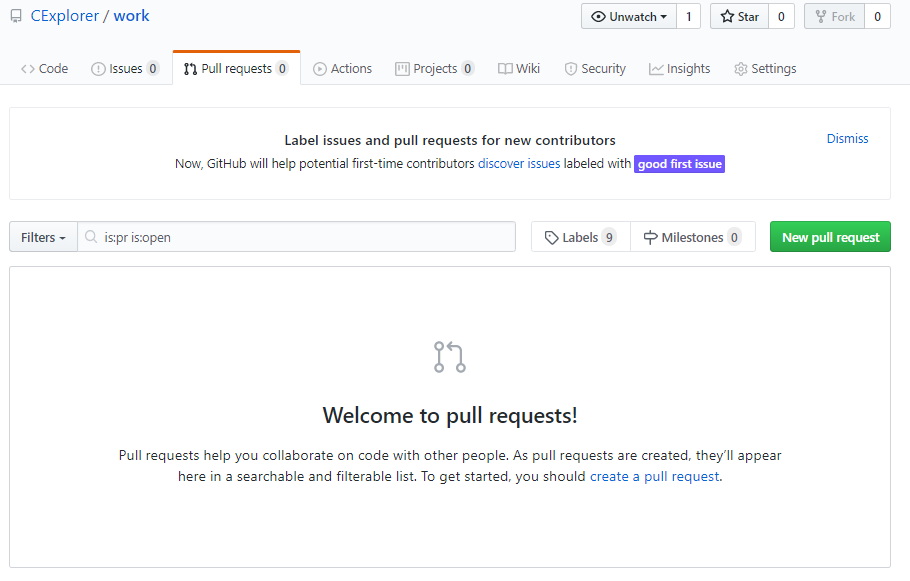


注意在执行 commit 命令时，备注信息里有个 “fix #1”，这是必要的，当备注信息中含有此字样的 commit 出现在组长仓库，仓库中编号为 #1 的 issue 就会自动关闭。类似的字样还有 “fixes #xxx、fixed #xxx、closes #xxx、close #xxx、closed #xxx”，这些并不重要，选择字母最少的 fix 就可以了。当然偶尔忘记写这个字样也不要紧的，issue 可以手动关闭，甚至关掉的 issue 还能再开。

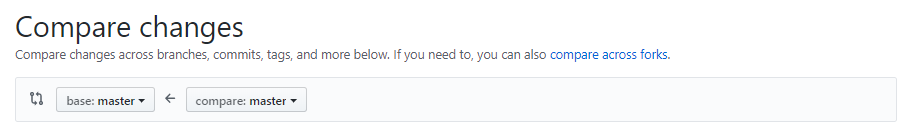
完成以上操作，组员的 GitHub 仓库会发生变化，新增一个版本号为 b374 的提交

接下来，怎么把修改从组员的仓库添加到组长的仓库呢？这就用到了 pull request 方法，简称 PR。这个词组比较费解，两个词都有动词属性，字面意思是 “拉，请求”，可以理解为这是一个名词性词组，意为 “允许被拉取的请求”，创建一个 PR 就是从甲分支向乙分支提一个请求，该请求中有一个或多个提交，对方觉得可以、没问题，就合并（merge) 这个请求，也就是把请求中所有提交的修改增加到乙分支上，整个过程简称 “提 PR”、“检查合并 PR”。提 PR 既可以在仓库内，也可以跨用户跨仓库。

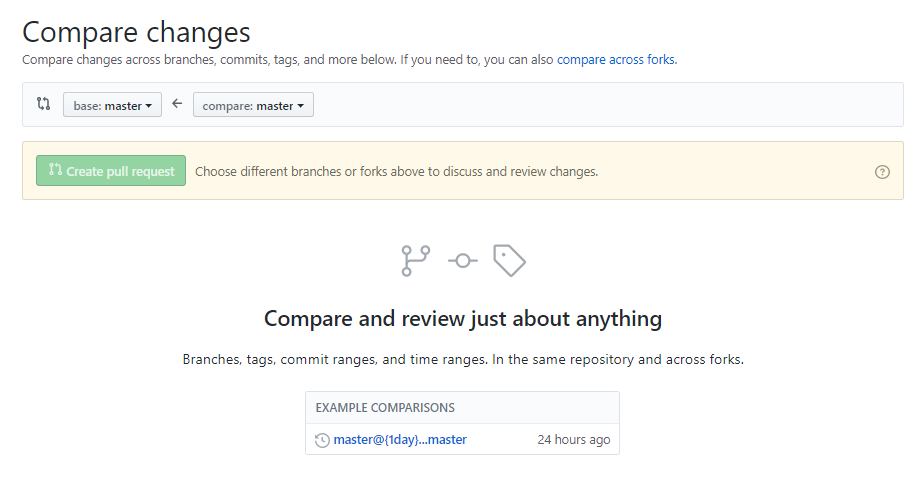
好，现在我们从组员的 work 仓库 master 分支给组长的 work 仓库 master 分支提一个 PR：



如下图所示，仔细检查紫色框中的内容是否正确，再看绿色椭圆形框中的绿色字样 “Able to merge.”，说明这个 PR 中的修改跟目标分支没有冲突：



从上图还可得知一些信息：该 PR 里有 1 个提交，1 个文件改动，1 个贡献者。点击上图绿色按钮跳转到确认页面，再次点击下图绿色按钮即可完成本次 “提 PR” 工作：



该页面只有参与项目协作的成员有权限进入，当前 GitHub 的登录用户是组员，所以可见，且对这个仓库有完全的管理权限，除了删除仓库。当然了，检查合并 PR 的权限也是有的。重要的一点：提了 PR 之后，一定要求参与项目的其他成员来检查合并，不要自己来，尽管自己有权限。

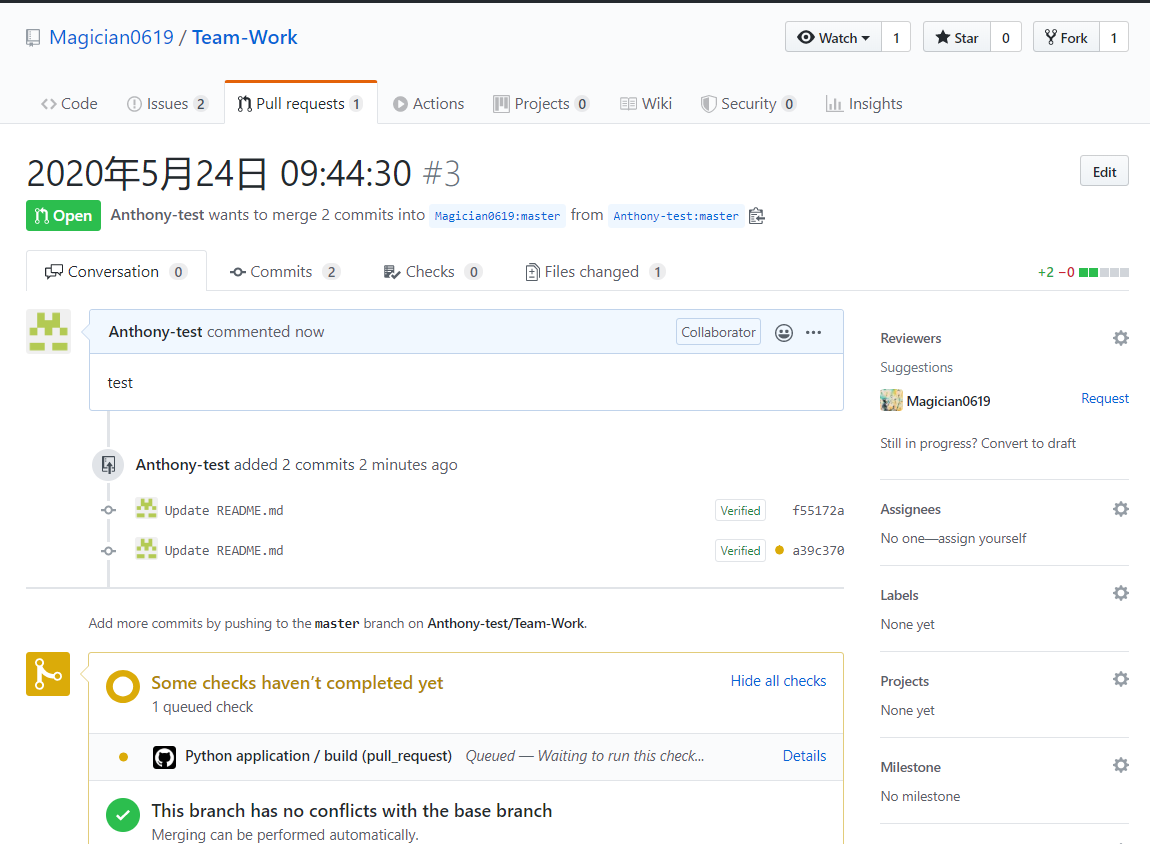
上图中绿色按钮是个下拉按钮，合并 PR 的方法有三种，分别解释一下：

Create a merge commit ：这种方式会在组长仓库的 master 分支上生成一个新的提交，且保留 PR 中的所有提交信息。这是一种常规操作，用得最多。

Squash and merge ：压缩合并，它会把 PR 中的全部提交压缩成一个。此方法的优点就是让提交列表特别整洁。一个 PR 里有很多提交，每个提交都是很细小的改动，保留这些提交没什么意义，这种情况就使用此方法合并 PR。

Rebase and merge ：这种方法不会生成新的提交，例如 PR 中有 6 个提交，用此方法合并后，组长仓库也会新增 6 个提交。注意，这些提交的版本号与组员的提交不同，此外完全一样。

现在切换到另一个登录组长账号的浏览器，打开合并 PR 的页面，用第一种方法合并：



这就是第一种方式合并的结果，生成了一个新的提交，这个提交里没有修改。因为样子不太美观，这是我最不喜欢用的方式。仔细看上图的 issue，变成了 1 个，也就是说在合并 PR 后，#1 issue 被关闭了。

以上就是一次完整的修改、提交、推送、提 PR、合并 PR 的过程。

需要注意的一点：从 A 向 B 提 PR 后，在 PR 合并或关闭前，A 上所有新增的提交都会出现在 PR 里。

因为组长的 master 分支新增了一个空提交，所以需要让组员的仓库同步组长的仓库，使它们的提交版本一致。作为组员，要时刻保持自己的 master 分支与组长的一致，以避免在下次提 PR 时出现冲突，该操作叫做 “同步主仓库”，组长的仓库就是主仓库。

提 PR、合并 PR 只能在 GitHub 页面上操作。同步主仓库是要用 Git 操作的。现在回到实验环境中操作。首先，使用 remote 系列命令来增加一个关联主机，执行 git remote add [主机名] [主仓库的地址]，注意，主仓库的地址使用 https 开头的：

如上图所示，主机名是随意定义的，只要不是 origin 就可以，因为自己的仓库地址对应的主机名是 origin，主仓库的主机名通常定义为 up 或 upstream，这个主机名其实就是一个变量，它的值就是仓库地址，例如 git push origin master 完全等于 git push git@github.com:Manchangdx/work master 。

如此说来，关联主仓库后也没什么变化嘛，确实如此，即使地址写错也不会报出来。现在可以使用前面课程介绍过的 fetch 命令来拉取主仓库的全部分支信息到本地仓库了，我有时使用这个命令看上一个命令是否有拼写错误：

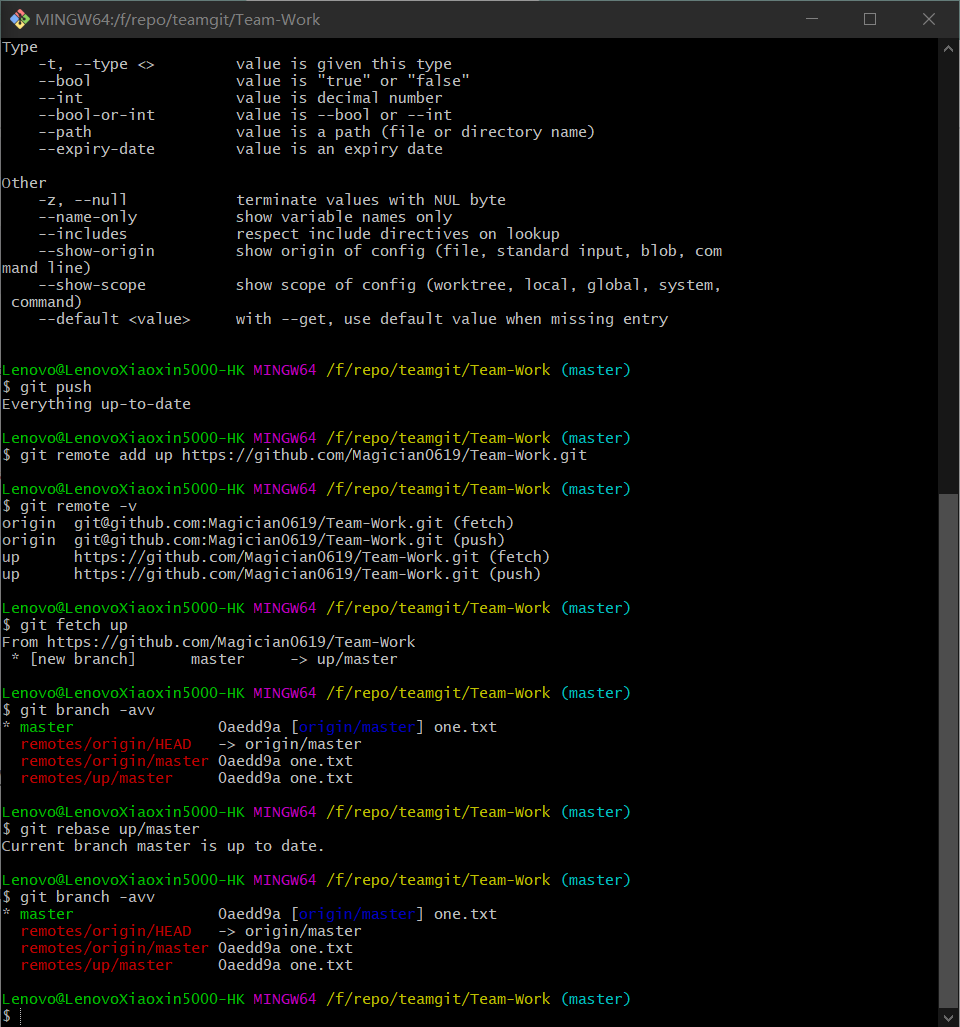
如何同步主仓库哩？方法有二，一是执行 git pull --rebase up master ，此命令需联网，二是执行 git rebase up/master，此命令不联网，因为前面已经执行了 git fetch up 这个需要联网的命令，本地已经有了最新的主仓库 master 分支信息，所以可以这么操作。

总结一下：git pull --rebase = git fetch + git rebase。现在使用方法二来同步：

同步主仓库已完成。现在可以继续修改提交自己的 master 分支了。然后一并推送到自己的远程仓库。

以上是在自己 Fork 的仓库里进行修改的过程。还有一种常用的方式，就是不用 Fork，直接克隆组长的仓库到本地，然后各自创建自己的分支，在自己的分支上进行修改提交，最后从自己的分支向 master 分支提 PR。方式不同，原理一样。

关于多人协作的 Git 操作就到这里了。



本节所讲知识点：

* 备注信息的妙用
* 什么是 PR
* 提 PR、检查合并 PR
* 关联主仓库（上级仓库）
* 同步主仓库

本次课程就暂告一段落。以后还会新增一些功能的讲解。Git 的操作还有很多高级用法，大家可以在掌握课程所讲知识的基础上自学成才。如遇课程问题欢迎在讨论区里留言。

# 实验 6 Git与GitHub简介

知识点

* Git 标签的作用
* 创建标签
* 查看标签
* 推送本地标签至远程仓库
* 删除标签
* 签出标签对应的提交版本
* GitHub releases 简介

在一个项目中，我们可能需要阶段性地发布一个版本，比如 V1.0、V1.0.2、V3.2 Beta 之类的，Git 的标签可以满足这个需求。在一个长期大型项目中，可能会有数千个提交版本，我们可能需要对重要的节点性提交打个记号，这时也可以使用 Git 的标签功能。在一些项目相关的书籍中，我们会看到 “执行 xxx 命令签出这个版本以查看对应的代码” ，这也是使用 Git 的标签功能做到的。本节实验将详细讲解此功能的具体操作。

前面的课程提到过 GitHub 的 issue 功能，issue 是仓库拥有者在 GitHub 上手动创建的，仓库被 Fork 时 issue 不会跟随。Tags 通常在本地使用 git 命令创建后推送到 GitHub 上，与 issue 相同的一点，它也只存在于项目仓库内，Fork 或提 PR 都不会带上它。在多人协作项目中，通常由组长对主仓库设置 Tags，单人项目自然就是自己说了算。

开始操作。首先，克隆仓库、配置信息、查看提交版本历史：

重要的一点，我们创建标签是给具体的某次提交创建的，跟分支无关。创建标签使用 git tag [标签名] -m [备注信息] [提交版本号] 这个命令。其中 -m [备注信息] 可以省略不写，但建议不要省略。[提交版本号] 可以省略，如果是给当前分支最新的提交创建标签的话。

给当前分支当前版本创建一个标签：



这样一个本地标签就创建完成了。

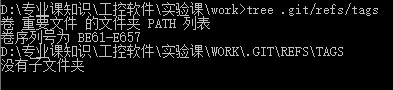
执行 git tag 命令显示仓库中的全部标签列表，执行 git show [标签名] 查看标签详情：





前文已提到，标签是在提交的基础上创建的，如果仓库的多个分支中都有这个提交版本，那么这些分支上就有关于这个提交的相同的标签。

当我们执行 git tag [标签名] 创建本地标签后，在仓库主目录的 .git/refs/tags 目录下就会生成一个标签文件：



执行 git tag -d [标签名] 删除本地标签，标签文件也会被删除：

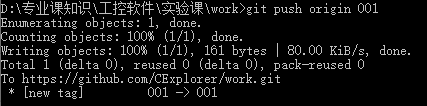


首先对两个提交版本创建对应的标签：

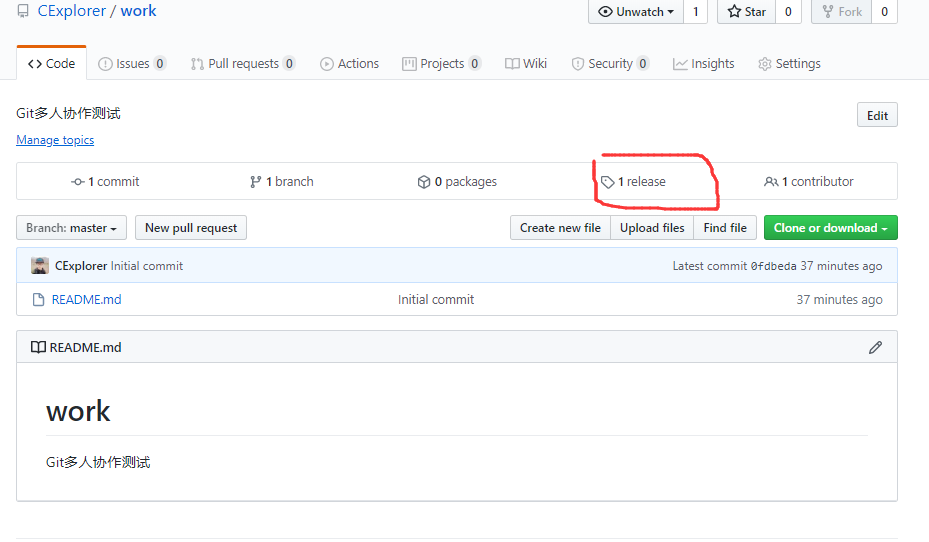




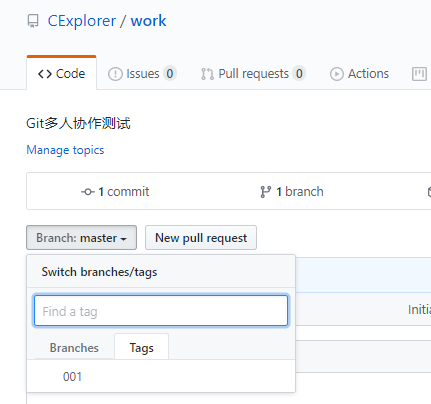
执行 git push origin [标签名] 推送标签到远程仓库，注意前面的命令都只涉及本地操作不需要联网，此命令需要联网：



我们到浏览器上打开仓库主目录，点击下图红色框可以查看 releases 和 tags ：



点 Tags 按钮查看标签：



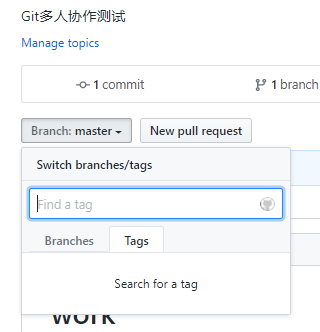
关于 releases 是什么，下文会介绍。

如果你一口气创建了 6 个标签，当然啦，这种情况很少发生，可以使用 git push origin --tags 命令将全部本地标签推送至远程仓库：

如果标签废弃不用或者写错了，可以使用 git push origin :refs/tags/[标签名] 删除远程仓库的标签，命令中的标签名其实也就是文件名：



再次查看远程仓库：



好，删除成功。以上就是关于 Git 标签的创建、查看、推送、删除的操作流程。

查看本地仓库的标签列表：

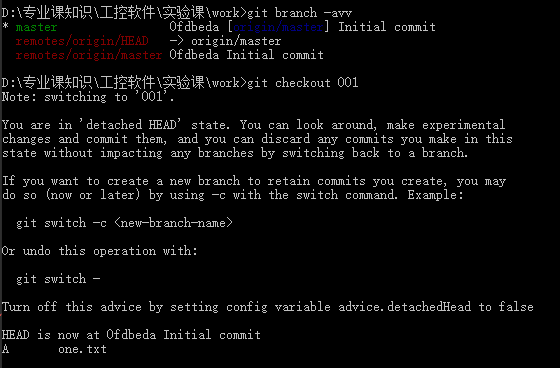


咦，001 标签还在呢？是的，本地标签需要另外手动删除，上文已演示。

现在介绍一下关于 “签出版本” 的操作，我们会见到类似这种说明：“如果你从 GitHub 上克隆了这个程序的仓库，那么可以在仓库主目录下执行 git checkout xxx 签出程序的这个版本。” 其实签出版本就是指定某个提交版本创建一个新的分支。

假定当前的 work 仓库就是一个程序，我们要签出 001 版本，执行以下步骤即可。

首先执行 git checkout [标签名] 切换到之前的某个提交版本，然后执行 git checkout -b [新的分支名] 将此提交版本固定到一个新分支上并切换到此分支：



这样就利用标签完成了提交版本签出的工作。

GitHub 的 releases 是 2013 年发布的新功能，旨在协助软件开发者分发新版本给用户，关于这个功能这里仅作简单介绍。

当项目组织宣布发布一个软件产品的版本，发布过程就是一个将软件交付给最终用户的工作流。版本是具有修改日志和二进制文件的一类对象，它们提供了 Git 工作流之外的完整项目历史，它们也可以从存储库的主页上被访问。发布版 release 附带发布说明和下载软件或源代码的链接。按照许多 Git 项目的约定，发布版本与 Git 的标签 tag 绑定。您可以使用现有的标签，或者让 release 在发布时创建标签。这就是上面查看 GitHub 仓库中标签信息时出现的场景。

标签是 Git 中的概念，而 releases 则是 Github、码云等源码托管商所提供的更高层的概念。Git 本身是没有 releases 这个概念，只有 tag。两者之间的关系则是，release 基于 tag，为 tag 添加更丰富的信息，一般是编译好的文件。

本节所讲知识点：

* Git 标签的作用
* 创建标签
* 查看标签
* 推送本地标签至远程仓库
* 删除标签
* 签出标签对应的提交版本
* GitHub releases 简介

以上就是 Git 标签的全部操作和 GitHub releases 的介绍，相对前面的课程而言比较简单，依照文档的步骤操作一遍即可掌握。有任何问题请随时到讨论区里留言。